UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMO GENÉTICO EMBARCADO APLICADO À INDUÇÃO DE OPERADORES MORFOLÓGICOS

ÉTTORE LEANDRO TOGNOLI

ORIENTADOR: PROF. DR. EMERSON CARLOS PEDRINO

São Carlos – SP

Março/2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMO GENÉTICO EMBARCADO APLICADO À INDUÇÃO DE OPERADORES MORFOLÓGICOS

ÉTTORE LEANDRO TOGNOLI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Processamento de Imagens e Sinais e Arquitetura de Computadores

Orientador: Prof. Dr. Emerson Carlos Pedrino

São Carlos – SP

Março/2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Éttore Leandro Tognoli, realizada em 07/03/2016:

Prof. Dr. Emerson Carlos Pedrino
UFSCar

Profa. Dra. Maria do Carmo Nicoletti
UFSCar

Profa. Dra. Adriane Beatriz de Souza Serapião
UNESP

Dedico este trabalho a meus amigos e familiares, que sempre estiveram do meu lado,
me incentivando e auxiliando.

AGRADECIMENTOS

Obrigado, Bárbara, pela paciência e dedicação em todo esse tempo em que estivemos juntos, mesmo próximos ou distantes um do outro.

Obrigado, família, por sempre terem me apoiado e incentivado em todas as minhas decisões.

Obrigado, Persys e seus integrantes, por cederem espaço de trabalho e motivar o desenvolvimento deste projeto. Em especial para o João pelo café de todos os dias.

Obrigado, funcionários da UFSCar, pelo trabalho e esforço que possibilitaram a realização deste trabalho.

Obrigado, Prof. Emerson Carlos Pedrino, por ter compartilhado conhecimento, por sua dedicação e orientação.

Agradeço a todos meus colegas de "turma"que se tornaram companheiros nas dificuldades e espero que amigos até o final da vida.

A simplicidade é o último grau de sofistificação.

Leonardo da Vinci

RESUMO

A utilização de algoritmos genéticos, estratégias evolutivas e programação genética tem sido notoriamente abordada nas áreas de processamento de imagens e visão computacional e tem contribuído com resultados diversificados. Entretanto, esse tipo de abordagem necessita de grande poder computacional para que o tempo de processamento seja viável. Alguns trabalhos acadêmicos trazem esses algoritmos para as áreas de sistemas embarcados e dedicados, melhorando dessa forma, o desempenho dos algoritmos e minimizando o custo computacional. Este projeto aborda a implementação de um desses algoritmos, aplicado ao processamento morfológico de imagens, utilizando FPGAs e Verilog-HDL. O algoritmo genético desenvolvido é adaptado a uma implementação mais intuitiva e simples para o desenvolvimento de um *hardware* dedicado, podendo contribuir no desempenho do sistema, reduzir o tempo de treinamento e reduzir o consumo de recursos lógicos. Os estudos e testes realizados durante o projeto demonstraram bons resultados para a adaptação proposta.

Palavras-chave: algoritmo genético, *hardware*, FPGA, morfologia matemática, sistema embarcado, processamento de imagens

ABSTRACT

The use of genetic algorithms, evolutionary strategies and genetic programming has been notoriously addressed in the areas of image processing and computer vision and has contributed with diverse results. However, this approach requires large computing power. Some academic papers bring these algorithms for the areas of embedded and dedicated systems, improving in this way the performance of algorithms and minimizing the computational cost. This design addresses the implementation of these algorithms applied to morphological processing of images using FPGAs and Verilog-HDL. The genetic algorithm is adapted for a more intuitive and simple implementation for the development of a dedicated hardware and it may contribute to reduce logical resources and improve performance. Studies and tests carried out during the project showed good results for the proposed design.

Keywords: genetic algorithm, mathematical morphology, FPGA, embedded system, image processing,hardware

LISTA DE FIGURAS

2.1 Fluxograma de um Algoritmo Genético. 24

2.2 Representação gráfica de uma função multimodal. 25

2.3 Roda da Roleta com oito indivíduos. 28

2.4 Exemplo de cruzamento uniforme. 37

2.5 Exemplo de cruzamento de *n-pontos*, com $n = 2$. 38

2.6 Exemplo de cruzamento de *n-pontos*, com $n = 4$. 38

2.7 Exemplos de máscaras de cruzamento. 38

2.8 Algoritmo Genético Insular 44

2.9 Algoritmo Genético Celular 45

2.10 Exemplo de Super-indivíduo. 48

2.11 Exemplo da Formação de Subpopulações na Função Objetivo 49

2.12 Exemplo do resultado de uma dilatação (WANGENHEIM, 2001). 51

2.13 Processo de Dilatação (PEDRINO, 2008). 51

2.14 Dilatação binária com lógica digital, em que *i* é a imagem e *e* o elemento estruturante. 52

2.15 Exemplo de Erosão (WANGENHEIM, 2001). 52

2.16 Exemplo do processo de erosão (PEDRINO, 2008). 53

2.17 Erosão binária com lógica digital, em que *i* é a imagem e *e* o elemento estruturante. 53

2.18 Exemplo do resultado de uma abertura. Adaptado de Wangenheim (2001) . . . 54

2.19 Exemplo do resultado de um fechamento. Adaptado de Wangenheim (2001) . . 55

2.20 Exemplo de detecção de borda com erosão e dilatação. Adaptado de Wangenheim (2001) 56
2.21 Dimensões de Autômatos Celulares (GREMONINI; VICENTINI, 2008). . . . 58
2.22 Formato de Células (GREMONINI; VICENTINI, 2008). 58
2.23 Tipos de Vizinhaça (TEIXEIRA, 1996) 58
2.24 AC Binário Unidimensional de regra 30 59
2.25 AC Binário Unidimensional de regra 90 60
2.26 AC Binário Unidimensional de regra 150 60
2.27 Representação gráfica da regra 30. 61
3.1 Unidade de dilatação com circuito de dualidade e de *bypass*. 69
3.2 Unidade de erosão com circuito de dualidade e de *bypass*. 69
3.3 Unidade morfológica. 70
3.4 Diagrama do Processador Morfológico. 73
4.1 Exemplo de Decodificação de um Segmento do Cromossomo. 80
4.2 Exemplo de Decodificação de um Cromossomo de 32 *bits*. 80
4.3 Exemplo do Processo de Comparação das Imagens 82
4.4 Módulo de Avaliação 83
4.5 Cálculo da paridade dos segmentos de um cromossomo. 87
4.6 Memória da População. 88
4.7 *Hardware* do Autômato Celular. 92
4.8 Autômato Celular mais Deslocador. 93
4.9 Máscara de Cruzamento. 97
4.10 Cruzamento 97
4.11 Exemplo Gráfico do Processo de Seleção. 98
4.12 Produção da Máscara de Mutação 99
4.13 Mutação com *ou exclusivo* 100

4.14 Diagrama de Blocos do *Hardware* Genético 102
4.15 Conversor USB para Serial TTL 110
4.16 Diagrama de Blocos do *Hardware* Genético 110
4.17 Placa de Circuito Impresso com o FPGA Cyclone II EP2C5T144C6 111
4.18 Gravador do FPGA 111
4.19 Caso de Teste 1, 0x10381000 e 0x387C3810 112
4.20 Caso de Teste 2, 0x00100000 e 0x387C3810 112
4.21 Caso de Teste 3, 0x00100000 e 0x386C3810 112
4.22 Caso de Teste 4, 0x00181800 e 0x183C3C18s 112
4.23 Caso de Teste 5, 0x00400400 e 0x04E44E04 113
A.1 *Hardware* Evolutivo. 124

LISTA DE TABELAS

2.1 Regras de transição da regra 30. 60

3.1 Tabela de instruções da unidade lógica. 64

3.2 Tabela de instruções da unidade morfológica. 70

3.3 Consumo Lógico do Processador Morfológico. 75

4.1 Valores Pseudoaleatórios Gerados com um AC de Regra 30. 94

4.2 Tabela Verdade da Lógica *e* com 3 Entradas 99

4.3 Resultados dos Testes de DeJong 109

4.4 Consumo Lógico das Sínteses do AG. 110

4.5 Resultado do AG Serial no FPGA Cyclone II 112

LISTA DE LISTAGENS

2.1 Código Python do método de seleção roda da roleta 29

2.2 Código Python do método de seleção roda da roleta com ordenação linear . . . 30

2.3 Código Python do método de seleção por torneio 32

2.4 Código Python para mutação binária 34

2.5 Código Python da mutação *creep* para representação real 34

2.6 Código Python para Cruzamento 35

2.7 Código Python para cruzamento uniforme 37

2.8 Código Python para cruzamento de *n-pontos* 39

2.9 Código Python para cruzamento BLX-α 40

2.10 Código Python de um AG completo, com representação binária, seleção por torneio e cruzamento uniforme 40

2.11 Código Python de um AG completo, com representação real, seleção por torneio e cruzamento BLX-α 42

2.12 Código Python de um Algoritmo Genético Compacto 46

3.1 Código Verilog da unidade lógica 64

3.2 Código Verilog da unidade de dilatação 66

3.3 Código Verilog da unidade de erosão 67

3.4 Código Verilog da unidade morfológica 70

3.5 Código Verilog do processador morfológico 72

4.1 Código Verilog da Comparação *bit* a *bit* 82

4.2 Código Verilog da Função Objetivo ou Módulo de *fitness* 83

4.3 Código Verilog da População do AG 88
4.4 Código Verilog do AC 91
4.5 Código Verilog do Gerador de Números Aleatórios 93
4.6 Trecho de Código Verilog Responsável pela Mutação 99
4.7 Código Verilog do Módulo Genético 101
4.8 Trecho de Código Verilog Responsável pela Mutação 106

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO **17**

1.1 Contexto 17

1.2 Motivação e Objetivos 18

1.3 Metodologia de Desenvolvimento do Trabalho 20

1.4 Organização do Trabalho 20

CAPÍTULO 2 – FUNDAMENTAÇÃO TEÓRICA **22**

2.1 Algoritmos Genéticos 22

2.1.1 Representação 24

2.1.1.1 Representação Binária 25

2.1.1.2 Representação Real 26

2.1.2 Métodos de Seleção 27

2.1.2.1 Seleção Roda da Roleta 28

2.1.2.2 Seleção por Torneio 31

2.1.3 Métodos de Mutação 32

2.1.3.1 Mutação com Representação Binária 33

2.1.3.2 Mutação com Representação Real 34

2.1.4 Métodos de Cruzamento 35

2.1.5 Cruzamento com Representação Binária 36

2.1.6 Cruzamento com Representação Real 39

2.1.7 Exemplos de Algoritmos Genéticos 40
2.1.8 AGs Distribuídos 43
2.1.8.1 AGs Insulares 43
2.1.8.2 AGs Celulares 44
2.1.9 Algoritmo Genético Compacto 45
2.1.10 Tópicos Especiais 47
2.2 Morfologia Matemática 49
2.2.1 Dilatação Binária 50
2.2.2 Erosão Binária 51
2.2.3 Dualidade entre Dilatação e Erosão 53
2.2.4 Abertura Binária 54
2.2.5 Fechamento Binário 55
2.2.6 Outras Operações 55
2.3 Autômatos Celulares 56
2.3.1 Dimensão 57
2.3.2 Formato 58
2.3.3 Classificação 59
2.3.4 ACs Binários Unidimensionais 59

CAPÍTULO 3 – PROCESSADOR MORFOLÓGICO **62**

3.1 Trabalhos Correlatos 62
3.2 Desenvolvimento do Processador Morfológico 63
3.2.1 Unidade Lógica 64
3.2.2 Unidade de Dilatação 65
3.2.3 Unidade de Erosão 67
3.2.4 Unidade Morfológica 68
3.2.5 Organização do Processador Morfológico 72

3.3 Resultados 74
3.4 Considerações Finais 75

CAPÍTULO 4 – ALGORITMO GENÉTICO EMBARCADO **76**

4.1 Trabalhos Correlatos 76
4.2 Desenvolvimento do *Hardware* Dedicado 77
4.2.1 Representação Cromossômica 79
4.2.2 Função Objetivo 81
4.2.3 População 86
4.2.4 Números Aleatórios 90
4.2.5 Cruzamento 96
4.2.6 Seleção 97
4.2.7 Mutação 98
4.2.8 Fluxo Completo 100
4.3 Resultados 107
4.3.1 Implementação em *software* 108
4.3.2 Implementação em *Hardware* 109
4.4 Considerações Finais 114

CAPÍTULO 5 – CONCLUSÃO **115**

5.1 Contribuições e Limitações 115
5.2 Lições Aprendidas 115
5.3 Trabalhos Futuros 116

REFERÊNCIAS **118**

GLOSSÁRIO **122**

APÊNDICE A – *HARDWARE* EVOLUTIVO **123**

Capítulo 1

INTRODUÇÃO

Neste capítulo são abordados de forma breve os assuntos envolvidos no trabalho e seu contexto de aplicação.

1.1 Contexto

O processamento digital de imagens tem sido cada vez mais utilizado na sociedade atual em diversas áreas de conhecimento, tais como biologia, medicina e meteorologia, somente para citar alguns. A morfologia matemática é uma das diversas ferramentas utilizadas no processamento de imagens e baseia-se na teoria de conjuntos. Tem como diferencial o estudo das formas dos objetos presentes em imagens binárias. Também, pode ser aplicada em imagens coloridas ou em níveis de cinza, podendo assim obter resultados semelhantes ou até mesmo melhores que os de filtros lineares e não lineares. (FACON, 1996; BROGGI, 1994)

Geralmente, somente a utilização de processamento de imagens não é suficiente para determinadas aplicações. Na robótica, por exemplo, muitas vezes é necessário analisar as imagens mais detalhadamente para viabilizar a extração de dados úteis ao sistema, assim, possibilitando a tomada de decisões baseadas nos dados extraídos dessas imagem. O conceito de análise ou interpretação de imagem, transformando-a em dados, é abordado em Visão Computacional. O processamento de imagens consegue apenas realizar transformações nas imagens, ou seja, produz novas imagens a partir da imagem de entrada. Para transformar uma imagem em dados úteis é necessária a utilização de algoritmos de classificação e/ou Inteligência Artificial. Porém, a utilização desses tipos de algoritmos não dispensa necessariamente o processamento de imagens. O processamento de imagens pode contribuir com os resultados dos algoritmos supracitados tratando a imagem de entrada, realçando pontos críticos ou filtrando ruídos, por exemplo.

A inteligência computacional é um ramo da ciência que inspirada pela natureza busca simular aspectos comportamentais dos seres humanos, como o aprendizado, a percepção e a adaptação. Essa área possui uma variedade de algoritmos consolidados, como os de Redes Neurais Artificiais, Lógica Fuzzy, Sistemas Especialistas e Algoritmos Genéticos.(PACHECO, 1999)

Dentre alguns algoritmos de Inteligência Artificial, os algoritmos genéticos têm sido muito utilizados na solução de problemas complexos, tendo obtido bons resultados, inclusive na área de processamento digital de imagens (HRBACEK; SIKULOVA, 2013; PEDRINO, 2008). Basicamente, os algoritmos genéticos são algoritmos de busca e otimização, com o diferencial de se adaptarem bem a diversos contextos, trabalharem com um conjunto de possíveis soluções e serem apropriados para a resolução de problemas multimodais.

Em muitas aplicações de processamento de imagens ou detecção de padrões é de extrema importância a baixa latência do sistema, principalmente nos casos de sistemas críticos ou de tempo real. Inclusive, em algumas aplicações é necessário que o sistema seja embarcado, possibilitando-se assim sua utilização em projetos de automação e robótica.

As FPGAs [1] são dispositivos lógicos digitais que podem ser aplicadas em diversas áreas, tanto para o desenvolvimento de *hardware* ou para simulação. Dentro dessas áreas destacam-se: criptografia, reconhecimento de padrões, processamento de imagens, processamento de sinais e compressão de dados. As FPGAs facilitam e aceleram o desenvolvimento de protótipos de dispositivos físicos necessários para qualquer projeto digital (BOURIDANE et al., 1999; ORDOÑEZ et al., 2003). Este projeto teve por objetivo o desenvolvimento de um algoritmo genético embarcado, utilizando FPGAs para a indução de operadores lógicos e morfológicos, no contexto de processamento digital de imagens.

1.2 Motivação e Objetivos

Os algoritmos genéticos possuem algumas características que inspiram seu desenvolvimento em *hardware*. As operações binárias utilizadas em alguns modelos são facilmente reproduzidas com *hardware*. Com *software* a manipulação de *bits* pode ser custosa, já que nem todos os processadores possuem operações para manipular os *bits* separadamente.

Dependendo do tipo de problema a ser abordado, pode ser necessário um grande poder computacional, normalmente exigindo muito tempo de execução. Como os algoritmos genéticos trabalham com um conjunto de possíveis soluções, eles geralmente são mais lentos e custosos do que outros algoritmos de otimização. Muitas vezes um algoritmo genético ainda

[1] Do inglês Field-Programmable Gate Array

está avaliando sua primeira população, enquanto muitos métodos de Subida de Encosta já têm encontrado a solução (GALVÃO, 1999).

O principal objetivo deste projeto foi acelerar o processo de otimização de um algoritmo genético, com a utilização de uma implementação de baixo nível com FPGAs. A implementação em *hardware* desses tipos de algoritmos levantam diversas questões. Algumas partes do algoritmo são complexas para serem implementadas em *hardware*, podendo aumentar o consumo de recursos lógicos. Isso poderia ser contornado utilizando uma abordagem mista entre *software* e *hardware*, porém, a utilização excessiva de *software* poderia prejudicar o desempenho. Também é possível realizar modificações em partes do algoritmo, com o intuito de simplificá-las e consequentemente diminuir o consumo lógico e/ou aumentar a frequência de trabalho do *hardware*. A implementação final deste projeto contribui apresentando uma modificação que simplifica uma das partes dos algoritmos genéticos.

Além do custo computacional dos algoritmos genéticos, também deve ser levado em consideração o custo do processamento de uma imagem. Geralmente é necessário aplicar um conjunto de operações para cada *pixel* da imagem. No caso de um vídeo, ainda é necessário executar esse processo em várias imagens por segundo. Dependendo do conjunto de operações que está sendo utilizado e do tamanho da imagem, a visualização em tempo real pode se tornar inviável em computadores tradicionais. Durante o processo de aprendizado de um algoritmo genético, é necessário testar possíveis soluções para o problema em questão diversas vezes. Se esse teste necessitar de uma quantidade de tempo significativa, o desempenho do algoritmo genético, como um todo, pode ser drasticamente prejudicado. Para viabilizar o processamento de imagens em tempo real e a utilização do algoritmo genético, também foi desenvolvido um *hardware* para processamento de imagens.

Muitas vezes a escolha de um algoritmo ou de um filtro para processar uma imagem não é trivial, nem mesmo para um especialista da área. Detectar objetos ou formas dentro de imagens pode ser um processo difícil. O sistema embarcado proposto neste projeto visa determinar o filtro ou algoritmo, assim, o usuário só precisa determinar o resultado esperado. A partir de uma imagem de entrada e uma imagem objetivo, o algoritmo genético fará a busca pelo melhor conjunto de operações que consegue transformar a imagem original na desejada. Esse conjunto de operações poderá, então, ser aplicado a outras imagens. Assim, se o conjunto de operações foi produzido a partir de um par de imagens, sendo a original uma imagem qualquer e a desejada o resultado de uma detecção de borda sobre a imagem original, espera-se que esse conjunto de operações tenha o mesmo efeito que uma detecção de borda, em outras imagens.

1.3 Metodologia de Desenvolvimento do Trabalho

Inicialmente, foi realizada uma pesquisa bibliográfica sobre algoritmos genéticos e similares, na busca de um modelo de implementação que se adaptasse melhor a uma implementação em *hardware*.

Após a pesquisa, alguns modelos foram implementados em *software*, para verificar seu funcionamento e desempenho. Algumas características foram simplificadas e, posteriormente, verificado os impactos sobre o desempenho do algoritmo. Com esses resultados foi possível determinar um modelo de implementação mais transparente para *hardware*, com bom desempenho e consumo reduzido de recursos. Após as simulações, o modelo foi sintetizado, com intuito de analisar o consumo de recursos lógicos, a frequência máxima de trabalho e garantir o real funcionamento.

A implementação do *hardware* dedicado deste projeto, foi realizada utilizando a linguagem de descrição de *hardware* (HDL[2]) Verilog, optou-se pela utilização do Icarus Verilog, que é uma implementação *open-source* (WILLIAMS, 2016; WILLIAMS; BAXTER, 2002). Para visualizar as simulações dos circuitos desenvolvidos, foi utilizado o GTKWave (GTKWAVE, 2016).

As FPGAs utilizadas são fabricados pela Altera, assim, foi necessária a utilização das ferramentas fornecidas pela empresa. A IDE[3] da Altera, Quartus II, fez todo o processo de síntese, mensuração do consumo lógico e mesuração da frequência máxima de trabalho.

1.4 Organização do Trabalho

O trabalho foi dividido em duas implementações principais: a implementação de um processador morfológico para imagens binárias, abordada no Capítulo 3; e a implementação de um algoritmo genético, abordada no Capítulo 4. Cada um desses capítulos aborda alguns pontos críticos e possibilidades de implementação, justifica a forma escolhida, descreve como foi realizado o desenvolvimento e apresenta alguns resultados.

No Capítulo 2 são explicados os conceitos utilizados no trabalho. Devido à diversidade de variações de algoritmos genéticos, o trabalho abordou somente os conceitos mais fundamentais para justificar a forma de implementação. Optou-se pela utilização de código Python para exemplificar os algoritmos, ao invés de pseudo-código. Assim é possível testar ou uti-

[2]Do Inglês *Hardware Description Language*
[3]Do inglês *Integrated Development Environment*

lizar efetivamente os algoritmos. Os autômatos celulares, também explicados no Capítulo 2, foram utilizados na geração de números aleatórios, necessária para a implementação dos algoritmos genéticos. Os conceitos de morfologia matemática, área onde foi aplicado os algoritmos genéticos, também foram resumidos no Capítulo 2.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados o conceitos fundamentais para o entendimento do trabalho, esses foram utilizados durante a implementação do protótipo final. Os conceitos abordados são algoritmos genéticos, morfologia matemática e autômatos celulares. Todos são conceitos extensos, por isso, são somente apresentados os princípios essenciais.

Como o trabalho diz respeito a implementação de um algoritmo genético aplicado ao processamento morfológico de imagens, aqui foram adicionadas as seções de algoritmos genéticos e morfologia matemática, com maior ênfase no que foi utilizado no projeto. Para o desenvolvimento do algoritmo genético, como será visto, é necessária a utilização de números aleatórios, esses foram gerados por meio de autômatos celulares. Por esse motivo, também há uma seção dedicada a eles.

2.1 Algoritmos Genéticos

Segundo Darwin e Green (2005), as espécies evoluem de acordo com sua adaptação ao ambiente, os indivíduos melhores adaptados tem uma maior chance de sobrevivência, logo, uma chance maior de se reproduzirem e espalhar seus genes. Um princípio importante de Darwin, é que o código genético dos indivíduos não são modificados durante sua existência, a modificação dos genes somente ocorre na transição de uma geração para outra, através de recombinação, em casos de reprodução sexuada, ou, através de mutação. Então toda geração carrega de alguma forma uma herança genética, afinal, seus genes são em maioria recombinações e mutações dos genes passados que melhor se adaptaram ao ambiente.

> Algoritmos Genéticos (AGs) são métodos computacionais de busca baseadas nos mecanismos de evolução natural e na genética. Em AGs, uma população

> de possíveis soluções para o problema em questão evolui de acordo com operadores probabilísticos concebidos a partir de metáforas biológicas, de modo que há uma tendência de que, na média, os indivíduos representem soluções cada vez melhores à medida que o processo evolutivo continua. (TANOMARU, 1995)

Os AGs estão dentro do contexto da computação evolucionária, que deve ser entendida como uma forma genérica e adaptável para a resolução de problemas complexos. Dentro da computação evolucionária também se encontram os conceitos de programação evolucionária e estratégias evolucionárias (BÄCK; HAMMEL; SCHWEFEL, 1997; LARRANAGA et al., 1999). Existem diversas variantes de algoritmos evolucionários, porém, todos possuem o mesmo conceito de funcionamento: a partir de uma população de indivíduos, a pressão seletiva faz com que apenas os indivíduos mais aptos sobrevivam e, por consequência, a aptidão da população é melhorada (EIBEN; SMITH, 2003). Inicialmente, a programação evolucionária foi concebida como uma tentativa de se produzir inteligência artificial (FOGEL, 1992, 1964).

Os AGs trabalham com um conjunto de possíveis soluções, em que cada possível solução é chamada de *indivíduo* e o conjunto de indivíduos chamado de *população*. Cada indivíduo possui um código genético que pode ser chamado de cromossomo. O código genético deve ser decodificado para poder ser aplicado como resolução de um determinado problema. AGs são facilmente adaptáveis a vários tipos de problemas, geralmente sendo necessário apenas a criação da representação genética do problema e da função que realiza o cálculo da aptidão dos indivíduos. A representação genética do problema é como uma possível solução é codificada, ou seja, uma estrutura de dados capaz de representar uma solução para o problema. A partir disso, os outros métodos utilizados nos AGs são basicamente os mesmos podendo ser reutilizados.

O fluxo de funcionamento de um AG pode ser divido em três etapas: reprodução, avaliação e seleção. Durante a etapa de reprodução, o AG combina seus indivíduos utilizando procedimentos referentes a metáforas biológicas da própria reprodução. No período de avaliação, os indivíduos recém gerados são aplicados ao problema, viabilizando a mensuração do grau de aptidão de cada indivíduo. Por fim, na etapa de seleção, os indivíduos mais aptos são selecionados para dar origem às próximas gerações, enquanto os menos aptos são descartados. Esses procedimentos são repetidos até que um critério de parada seja satisfeito. Limitar a quantidade de gerações e/ou determinar um erro mínimo a ser atingido, são critérios de parada comuns (TANOMARU, 1995). Para a execução dessas etapas é primeiramente necessário inicializar a população, essa pode ser iniciada com indivíduos aleatórios ou por um conjunto predeterminado. Na Figura 2.1 é apresentado um fluxograma com as etapas supracitadas.

As formas de se realizar essas três etapas do AG variam. A avaliação está intimamente re-

Figura 2.1: Fluxograma de um Algoritmo Genético.

lacionada ao problema que se pretende solucionar. A reprodução está intimamente relacionada com a forma de codificação dos indivíduos. A seleção também tem diversas formas de implementação, mas depende apenas da aptidão dos indivíduos e da implementação da população. Assim, os métodos de seleção podem ser reutilizados em praticamente todos os AGs.

2.1.1 Representação

A codificação de um individuo da população pode ser chamada de cromossomo ou de genótipo e sua decodificação de fenótipo. A codificação do genótipo deve conseguir representar as possíveis soluções dentro do espaço de busca desejado, deve também ser levado em consideração a precisão desejada.(PEDRINO, 2008)

As representações mais básicas dos cromossomos são a binária e a real. A representação binária foi utilizada nos trabalhos de Holland (1975) e é muito tradicional. A representação real normalmente gera cromossomos menores e mais compreensíveis por seres humanos (GALVÃO, 1999). Ambas as formas de codificação serão explicadas a seguir nas seções 2.1.1.1 e 2.1.1.2 respectivamente.

2.1.1.1 Representação Binária

Um cromossomo binário é basicamente um vetor de uns e zeros, sendo que o tamanho deste vetor varia conforme a precisão desejada para a solução do problema. Por exemplo, um vetor de 8 posições poderia representar 256 valores diferentes dentro do espaço de busca. Em casos que a precisão necessária é muito alta, os cromossomos tendem a ficar muito grandes, prejudicando o desempenho do AG. Por outro lado a representação binária é simples e fácil de analisar. (GALVÃO, 1999; TANOMARU, 1995)

Considere a Equação 2.1 que tem sua representação gráfica ilustrada na Figura 2.2. Essa é uma função multimodal, com máximo em $x = 0$.

$$f(x) = \frac{20\cos(x\pi)}{|x/2| + 2} \; \forall \, x \; \in \; [-20; 20] \tag{2.1}$$

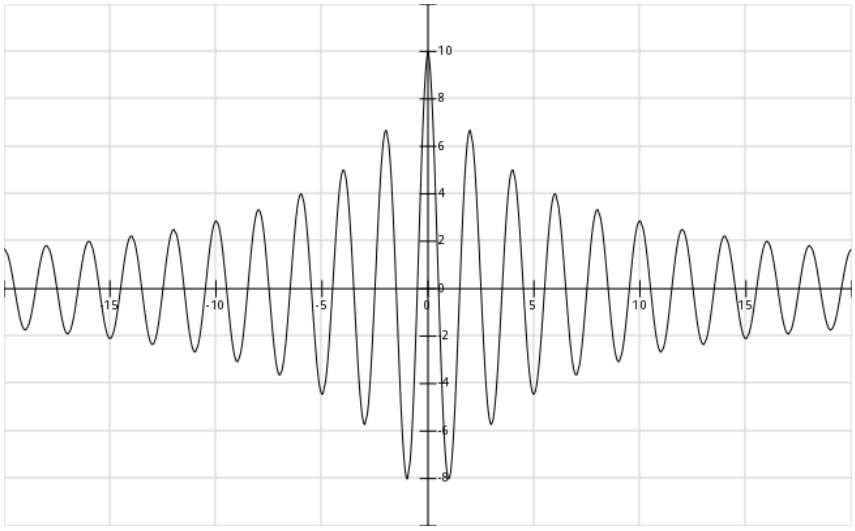

Figura 2.2: Representação gráfica de uma função multimodal.

Para codificarmos um indivíduo com uma representação binária para esse problema com uma precisão de 4 casas decimais, vamos precisar representar $40 \times 10^4 = 400.000$ pontos do espaço de busca, já que o domínio do problema tem largura $20 - (-20) = 40$. Então são necessários 19 *bits* para a representação cromossômica, segundo a Equação 2.2.

$$262.144 = 2^{18} < 400.000 < 2^{19} = 524.288 \tag{2.2}$$

Utilizando cromossomos binários com largura de 19 *bits*, o domínio do problema, ou espaço de busca, é segregado em 2^{19} partes iguais. O código genético é então codificado na forma de

um vetor de *bits*, como representado na Equação 2.3, onde C é um possível cromossomo, b_i os genes do cromossomo e o conjunto $\{0,1\}$ os alelos. Nos termos da biologia, alelos representam as formas possíveis de um gene (GALVÃO, 1999).

$$C = [b_{18}, b_{17}, ..., b_1, b_0] \;\Big|\; b_i \in \{0,1\} \tag{2.3}$$

Para decodificar o cromossomo é necessário converter o código genético da base 2 para a base 10 e posteriormente mapeá-lo para o domínio do problema. A conversão para base 10 utilizando a notação binária posicional[1] pode ser feita de acordo com a Equação 2.4, onde x_{10} representa o valor convertido para a base decimal.

$$x_{10} = \sum_{i=0}^{18} b_i 2^i \;\forall\; b_i \in C \implies x_{10} \in [0; 2^{19} - 1] \tag{2.4}$$

O mapeamento pode ser feito de acordo com a Equação 2.5, onde x representa o valor convertido para o domínio do problema, x_{max} o limite superior do domínio e x_{min} o limite inferior.

$$x = \frac{x_{max} - x_{min}}{2^{19} - 1} x_{10} + x_{min} \;\Big|\; x_{max} = 20,\; x_{min} = -20 \tag{2.5}$$

Dessa maneira os cromossomos $[00000000000000000000]$ e $[11111111111111111111]$ representam os valores -20 e 20, respectivamente. Os cromossomos mais próximos do valor 0, ponto máximo da função, tem os códigos $[01111111111111111111]$ e $[10000000000000000000]$, representando os valores $-0,000038$ e $0,000038$, respectivamente.

2.1.1.2 Representação Real

Utilizando representação real os cromossomos tendem a ficar menores e sua assimilação mais natural aos seres humanos, se comparado com a representação binária. A representação binária varia de tamanho conforme a precisão desejada, enquanto a real utiliza a codificação de números reais disponibilizada pela arquitetura do processador, mantendo seu tamanho constante. A criação de operadores para esse tipo de representação é simples, normalmente utilizando métodos aritméticos. Diversos pesquisadores tem discutido sobre as representações, bi-

[1] Representação mais comum, onde os *bits* mais significativos ficam para esquerda e os menos para direita. O valor de cada *bit* é igual á 2^i sendo i a posição do algarismo contando da direita para esquerda iniciando-se em 0.

nária e real, demonstrando experimentos que favorecem a representação real (GALVÃO, 1999).

Considerando a Equação 2.1, utilizada como exemplo para exemplificar a representação binária na seção 2.1.1.1, um cromossomo com a representação real seria apenas um número pertencente ao domínio do problema, como representado da Equação 2.6. Pode-se dizer que esse cromossomo tem apenas um gene, que é um valor real. Funções com N parâmetros necessitam de cromossomos com N genes.

$$C = c | c \in \mathbb{R}, c \in [-20; 20] \tag{2.6}$$

Computacionalmente, esse valor real também será convertido em um valor binário dentro do computador. Porém, dessa maneira a precisão é definida pela forma como o computador codifica um número real. Esse modelo de representação também altera como o AG trabalha com as operações de reprodução.

2.1.2 Métodos de Seleção

Os métodos de seleção tem a responsabilidade de representar metáforas para a pressão seletiva. A pressão seletiva faz com que os indivíduos mais aptos tenham maior chance de se reproduzirem, enquanto os menos aptos tem menores chances. Dessa forma, os genes dos indivíduos com maior aptidão terão maior probabilidade de serem propagados para as próximas gerações. Caso a pressão seletiva esteja muito elevada, ou seja, a chance dos indivíduos mais aptos se reproduzirem for muito maior do que a dos indivíduos menos aptos, a população tende a perder diversidade genética, podendo prejudicar os resultados do AG. Os métodos de seleção normalmente selecionam indivíduos de forma aleatória, mas proporcional as suas aptidões. Esses indivíduos são utilizados para a geração da próxima população, com a utilização de métodos que representam metáforas da reprodução.

Existem diversos métodos de seleção e cada um deles ainda pode ter algumas pequenas variações. Neste capítulo são somente apresentados os métodos de seleção *roda da roleta*, que é muito tradicional e o método de seleção por *torneio*, que tem características interessantes para uma implementação em *hardware*.

2.1.2.1 Seleção Roda da Roleta

Nó método de seleção roda da roleta, é simulada uma roleta em que cada casa representa um indivíduo e tem tamanho proporcional a aptidão de seu respectivo indivíduo. É importante que as aptidões não possuam valores negativos ou que as mesmas sejam normalizadas, pois, os valores negativos para a aptidão inviabilizariam a representação das casas da roleta, já que uma casa de tamanho negativo não seria passível de implementação. Depois de construir a roleta, são sorteados N pontos de seu perímetro, sendo N o tamanho da população. Cada ponto sorteado pelo perímetro da roleta, seleciona uma casa que representa o indivíduo sorteado.

Na Figura 2.3 é ilustrada uma roleta com 8 casas, cada uma delas representa um indivíduo, sendo esses diferenciados pela textura. Então 8 pontos do perímetro foram sorteados aleatoriamente, esses representados pelas setas ao redor da roleta. As setas possuem as texturas de seus respectivos indivíduos.

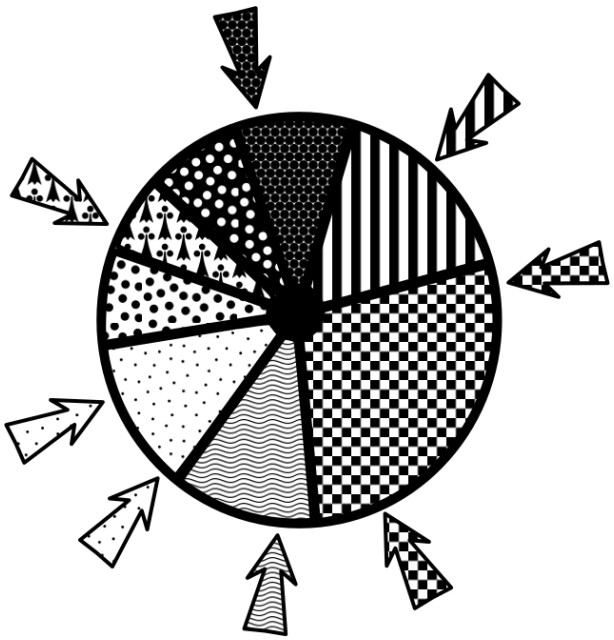

Figura 2.3: Roda da Roleta com oito indivíduos.

Para determinar o tamanho de cada casa da roleta é necessário calcular o valor da aptidão de cada indivíduo, para isso, precisaremos utilizar uma função de adequabilidade juntamente a função objetivo, assim garantindo valores positivos para a aptidão. Utilizando a Equação 2.1 como função objetivo, tendo como intenção encontrar o máximo global do intervalo especificado, podemos deduzir que o valor da aptidão é proporcional ao resultado da função objetivo, porém, os resultados negativos impedem a utilização direta desses resultados. Por esse motivo, é necessária a função de adequabilidade. Nesse caso uma possível função de adequabilidade é apresentada na Equação 2.7, onde a é a função de adequabilidade, f a função objetivo, C_i um

cromossomo e $\mathbb{P}$ a população. Com essa função de adequabilidade nenhuma aptidão negativa será atribuída aos indivíduos da população.

$$a(C_i) = f(C_i) + \alpha \Big| \alpha = |min(f(C_i)| \; \forall \, C_i \in \mathbb{P} \tag{2.7}$$

Na Listagem 2.1 é apresentado um código em Python que implementa o método de seleção roda da roleta, com tamanho da população igual á 50, representação real, utilizando a Equação 2.1 como função objetivo e a Equação 2.7 como função de adequabilidade.

Listagem 2.1: Código Python do método de seleção roda da roleta

```
1  import random, math
2
3  population_size = 50
4  domain = [-20,20]
5
6  def f(x):
7          return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
8
9  population = [
10         random.uniform(*domain)
11         for i in range(population_size)
12 ]
13
14 def roulette_selection(pop):
15         selected = []
16         alpha = abs(min(*[f(c) for c in pop]))
17         def a(c):
18                 return f(c) + alpha
19
20         pop_with_fitness = [
21                 (c,a(c))
22                 for c in pop
23         ]
24         roulette_size = sum([
25                 i[1] for i in pop_with_fitness
26         ])
27
28         for i in range(population_size):
29                 random_point = random.uniform(0,roulette_size)
30                 acc = 0
```

```
31                 for chromossome, fitness in pop_with_fitness:
32                     if random_point <= acc + fitness:
33                         selected += [chromossome]
34                         break
35                     acc += fitness
36         return selected
37
38 roulette_selection(population)
```

Utilizar a Equação 2.7 como função de adequabilidade pode trazer alguns problemas. Em casos que a função objetivo é muito amena, todos os indivíduos acabam tendo probabilidade de seleção muito próxima, fazendo com que o AG trabalhe quase que de forma aleatória. Em casos que a função objetivo é muito abrupta, as probabilidades de seleção entre os indivíduos ficam muito distantes, fazendo com que a pressão seletiva fique muito elevada. Uma maneira de se contornar esse problema, é utilizando uma função de adequabilidade baseada em ordenação. Os indivíduos da população são ordenados pelo valor da função objetivo de forma decrescente, no caso de uma maximização[2]. Depois são atribuídos valores de aptidão de acordo com a posição de cada indivíduo, como apresentado na Equação 2.8, onde i é o índice do cromossomo na população[3] e N o tamanho da população.

$$a(C_i) = min + (max - min)\frac{N-i}{N-1} \tag{2.8}$$

Geralmente, $max \in [1;2] \big| max + min = 2$. Assim. a aptidão representa o números de filhos esperados do indivíduo e $max - min$ a pressão seletiva (GALVÃO, 1999). Na Listagem 2.2 é apresentado um código Python que implementa o método da roleta utilizando essa função de adequabilidade, que tem o nome de *ordenação linear*.

Listagem 2.2: Código Python do método de seleção roda da roleta com ordenação linear

```
1 import random, math
2
3 population_size = 50
4 domain = [-20,20]
5
6 def f(x):
7         return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
8
```

[2]Para minimizar uma função, pode ser ordenado de forma crescente

[3]O primeiro indivíduo da população tem o índice igual a 1.

```
9  def a(p,max=2.0,min=0.0):
10         N = population_size
11         return min + ( max - min ) * (N-p)/(N-1)
12
13 population = [
14         random.uniform(*domain)
15         for i in range(population_size)
16 ]
17
18 def sorted_roulette_selection(pop):
19         sorted_pop = sorted(pop,key=f,reverse=True)
20         selected = []
21         pop_with_fitness = [
22                 (sorted_pop[i],a(i+1))
23                 for i in range(population_size)
24         ]
25         roulette_size = sum([
26                 i[1] for i in pop_with_fitness
27         ])
28
29         for i in range(population_size):
30                 random_point = random.uniform(0,roulette_size)
31                 acc = 0
32                 for chromossome,fitness in pop_with_fitness:
33                         if random_point <= acc + fitness:
34                                 selected += [chromossome]
35                                 break
36                         acc += fitness
37         return selected
38
39 sorted_roulette_selection(population)
```

2.1.2.2 Seleção por Torneio

No método de seleção por torneio são sorteados de forma aleatória N grupos de tamanho n, sendo N o tamanho da população e $n \in [1;N]$. Depois de selecionados os grupos, o melhor indivíduo de cada grupo é selecionado para dar origem a nova geração. O valor de n determina proporcionalmente a pressão seletiva, ou seja, quanto maior o valor de n maior será a pressão seletiva. Valores comuns para n são 2 e 3. Uma característica interessante do método de seleção por torneio, é que ele não necessita de uma função de adequabilidade. (GALVÃO, 1999)

Baseando-se no mesmo exemplo da Equação 2.1, o código Python apresentado na Listagem 2.3 implementa o método de seleção por torneio, utilizando uma população de tamanho 50, representação real e *n* igual á 3.

Listagem 2.3: Código Python do método de seleção por torneio

```
1  import random, math
2
3  population_size = 50
4  domain = [-20,20]
5
6  def f(x):
7          return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
8
9  population = [
10         random.uniform(*domain)
11         for i in range(population_size)
12 ]
13
14 def tournament_selection(pop,p=population_size,n=3):
15         selected = []
16         for i in range(population_size):
17                 group = random.sample(pop,n)
18                 selected += [max(*group,key=f)]
19         return selected
20
21 tournament_selection(population)
```

2.1.3 Métodos de Mutação

Os métodos de mutação são metáforas para as alterações que podem acontecer nos genes no momento da reprodução. Essas alterações normalmente são definidas pelos meios externos, eventos ocorridos dentro do ambiente onde os indivíduos estão inseridos. Porém, é inviável simular os eventos de um ambiente completo para calcular a mutação, assim, geralmente as mutações utilizam métodos baseados no caos, ou seja, valores aleatórios ou pseudoaleatórios.

Os métodos de mutação variam conforme a forma de representação do indivíduo, de acordo com a sua codificação. O processo de mutação dentro dos AGs, se assemelha a uma busca aleatória, onde é explorado o espaço de busca praticamente sem nenhum critério. Basicamente a mutação percorre um cromossomo e modifica alguns genes aleatoriamente, de acordo com uma taxa de mutação. (TANOMARU, 1995)

A mutação ajuda a manter a diversidade genética da população, porém acaba destruindo informações do cromossomo, por esse motivo a taxa de mutação normalmente é pequena, variando entre 0,5% e 1%(GALVÃO, 1999). Também é possível variar a taxa de mutação conforme a evolução das gerações. Iniciando-se com uma taxa de mutação mais elevada, a distribuição dos indivíduos pelo espaço de busca seria melhorada. Finalizando-se com uma taxa de mutação menor, os indivíduos da população tenderiam a convergir mais rapidamente.

2.1.3.1 Mutação com Representação Binária

A mutação aplicada a uma representação binária é muito simples. Os genes, ou no caso os *bits*, de cada cromossomo são invertidos com alguma probabilidade de acordo com a taxa de mutação. Assim, é possível que um cromossomo não sofra nenhuma mutação, ou que sofra mutação em um ou mais de seus genes.

Utilizando o exemplo de representação binária descrita na Seção 2.1.1.1, considere um vetor de *bits* com largura 19. Essa será a representação cromossômica utilizada para exemplificação. A mutação de um desses cromossomos pode ser definida de acordo com a Equação 2.9, sendo C_m o cromossomo com mutação, c_i os genes do cromossomo original, $\wedge$ a operação lógica *ou exclusivo* e $\bar{m}$ a taxa de mutação dentro do intervalo $[0;1]$[4].

$$C_m = [c_{n-1} \wedge m_{n-1}, c_{n-2} \wedge m_{n-2}, ..., c_0 \wedge m_0] \; \forall \; c_i \; \in \; C \; \Big| \; m_i = \begin{cases} 0 \; se \; \sim U([0;1]) \geq \bar{m} \\ 1 \; se \; \sim U([0;1]) < \bar{m} \end{cases} \tag{2.9}$$

Em outras palavras, é realizado um *ou exclusivo* entre o vetor de *bits* que representa o cromossomo e um vetor de *bits* de mesmo tamanho, esse gerado de forma aleatória com a probabilidade determinada pela taxa de mutação. Por exemplo, um cromossomo em que o vetor de *bits* corresponda á $[0100011101010011100]$ e um vetor de mutação igual á $[0000000000001000000]$, resultará em um novo cromossomo com o código genético $[0100011101011\bar{0}11100]$, como demonstra a Equação 2.10. O *bit* que sofreu modificação devido a mutação foi destacado por um traço na parte superior.

$$\begin{array}{r} (0100011101010011100) \\ \wedge\,(0000000000001000000) \\ \hline (0100011101011\bar{0}11100) \end{array} \tag{2.10}$$

[4] O valor 0 representa 0% e o 1 representa 100%.

Na Listagem 2.4 é apresentado um código em Python que realiza a mutação binária, utilizando a representação do exemplo supracitado, taxa de mutação de 1% e população com 50 indivíduos.

Listagem 2.4: Código Python para mutação binária

```
from random import choice, uniform

population_size = 50
population = [
        "".join(choice(['0','1']) for j in range(19))
        for i in range(population_size)
]

def mutate(chromossome, rate=0.01):
        return "".join(map(str,[
                int(gen,2) ^ ( 1 if uniform(0,1) <= rate else 0 )
                for gen in chromossome
        ]))

for chromossome in population:
        mutate(chromossome)
```

2.1.3.2 Mutação com Representação Real

Existem diversas formas de se realizar mutações com a representação real. Pode-se substituir um ou mais genes do cromossomo por valores aleatórios, ou por um dos limites do domínio, ou multiplicar alguns genes por um valor aleatório próximo de um, entre outros. Como nos exemplos utilizados o cromossomo com representação real possui apenas um gene, será exemplificado a mutação *creep*. Na mutação *creep*, são multiplicados um ou mais genes do cromossomo por valores aleatórios próximos a um (GALVÃO, 1999). Uma mutação que substitui genes seria muito destrutiva para esse caso. Na Equação 2.11 é exemplificado como realizar essa mutação, onde $\bar{m}$ é a taxa de mutação.

$$C_m = (1 + \sim U([-\bar{m}/2; \bar{m}/2]))C \;\Big|\; \bar{m} \in [0;1] \tag{2.11}$$

Na Listagem 2.5 é apresentado um exemplo de uma implementação em Python da mutação *creep*.

Listagem 2.5: Código Python da mutação *creep* para representação real

```
from random import uniform

population_size = 50
domain = [-20,20]

population = [
        uniform(*domain)
        for i in range(population_size)
]

def mutate(chromossome, rate=0.01):
        cm = chromossome * (1 + uniform(-rate/2, rate/2))
        return max(min(domain[1],cm),domain[0])

for chromossome in population:
        mutate(chromossome)
```

Como a representação real não foi utilizada nas implementações do projeto, as outras operações de mutação para essa representação não são abordadas.

2.1.4 Métodos de Cruzamento

Os métodos de cruzamento representam uma metáfora para a reprodução sexuada, onde a geração de um novo indivíduo é realizada combinando genes de indivíduos diferentes. Diferente da mutação que se assemelha a busca aleatória, o cruzamento busca soluções combinando valores já verificados dentro do espaço de busca. O cruzamento combina as informações contidas em dois cromossomos na busca de cromossomos melhores. Porém, nem sempre essa combinação resultará em indivíduos mais aptos.

O cruzamento é aplicado com uma taxa aos indivíduos selecionados pelo método de seleção. Os indivíduos são separados em pares, sorteados aleatoriamente, e o cruzamento é aplicado de acordo com a taxa de cruzamento. O código Python, apresentado na Listagem 2.6, exemplifica como aplicar a operação de cruzamento com uma probabilidade definida. Nesse código os métodos de seleção e de cruzamento não foram implementados, somente comentados. A taxa de cruzamento utilizada foi de 80%.

Listagem 2.6: Código Python para Cruzamento

```
import random, math

```

```
3  population_size = 50
4  domain = [-20,20]
5  R = 0.8
6
7  population = [
8          random.uniform(*domain)
9          for i in range(population_size)
10 ]
11
12 def selection(pop):
13         # apply some selection method
14         return pop
15
16 def crossover(dad,mum):
17         # apply some crossover method
18         return dad,mum
19
20 sub_pop = selection(population)
21 random.shuffle(sub_pop)
22 for i in range(0,len(sub_pop),2):
23         dad,mum = i,(i+1)%(len(sub_pop))
24         if random.uniform(0,1) > R:
25                 continue
26         sub_pop[dad],sub_pop[mum] = crossover(sub_pop[dad],sub_pop[mum])
```

2.1.5 Cruzamento com Representação Binária

Os métodos de cruzamento mais conhecidos para a representação binária, são o de *n-pontos* e o *uniforme*. No cruzamento uniforme é gerado de forma aleatória um vetor binário do mesmo tamanho de um cromossomo, esse vetor é utilizado para selecionar de quais dos cromossomos pais será copiado cada gene. Um exemplo de cruzamento uniforme com cromossomos de 9 *bits* pode ser visto na Figura 2.4. Nessa figura um vetor de cruzamento, [11010001], é utilizado para gerar dois novos indivíduos, $Filho_0$ e $Filho_1$, com a combinação dos genes de outros dois indivíduos, Pai_0 e Pai_1.

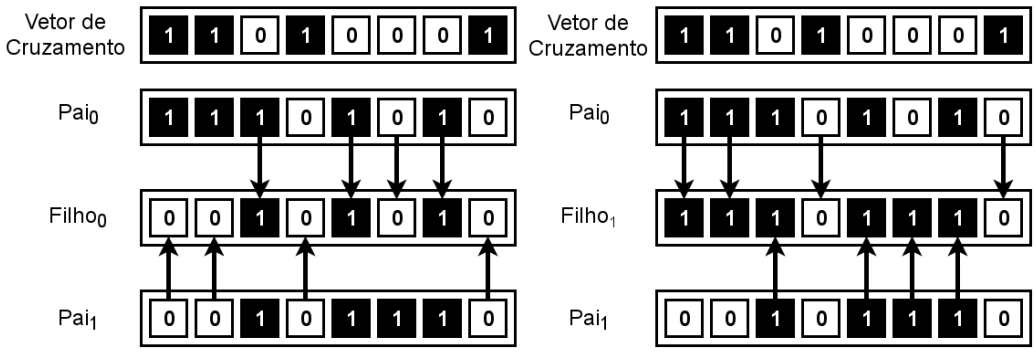

Figura 2.4: Exemplo de cruzamento uniforme.

Na Listagem 2.7 é aprestada uma implementação em Python do cruzamento uniforme, utilizando a mesma representação apresentada na Seção 2.1.1.1.

Listagem 2.7: Código Python para cruzamento uniforme

```
1  from random import choice
2
3  population_size = 50
4  population = [
5          "".join(choice(['0','1']) for j in range(19))
6          for i in range(population_size)
7  ]
8
9  def crossover(d,m):
10         if len(d) != len(m):
11                 raise Exception()
12         return map("".join,zip(*[
13                 (d[i], m[i]) if choice([True,False]) else (m[i],d[i])
14                 for i in range(len(d))
15         ]))
```

No cruzamento de *n*-pontos são escolhidos aleatoriamente *n* pontos diferentes de um determinado cromossomo, o separando em seções. Então, essas seções são trocadas com um outro cromossomo. Um exemplo desse cruzamento com $n = 2$ pode ser visto na Figura 2.5 e outro exemplo com $n = 4$ na Figura 2.6. Nessas figuras os traços serrilhados verticais representam os pontos sorteados. Em ambas as figuras dois novos cromossomos, $Filho_0$ e $Filho_1$, são gerados com a combinação de outros dois cromossomos, Pai_0 e Pai_1.

Para a implementação desse método de cruzamento, também pode ser utilizado um vetor binário para selecionar os *bits* dos pais. Porém, esse vetor deve ser gerado respeitando as definições do cruzamento de *n*-pontos, ou seja, esse vetor deve ser formado por segmentos de *bits* que são alternados nos *n* pontos sorteados. Na Figura 2.7 são apresentados alguns exemplos

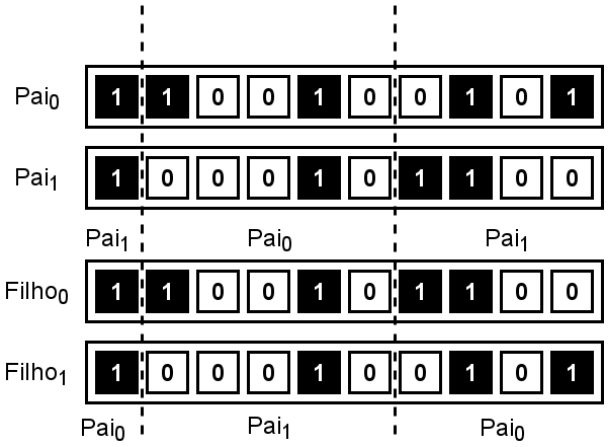

Figura 2.5: Exemplo de cruzamento de *n-pontos*, com $n = 2$.

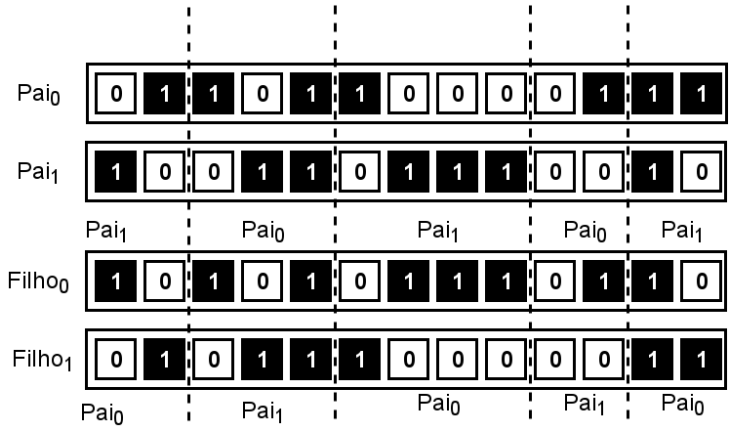

Figura 2.6: Exemplo de cruzamento de *n-pontos*, com $n = 4$.

de vetores, ou máscaras, para o cruzamento uniforme e cruzamento de n-pontos.

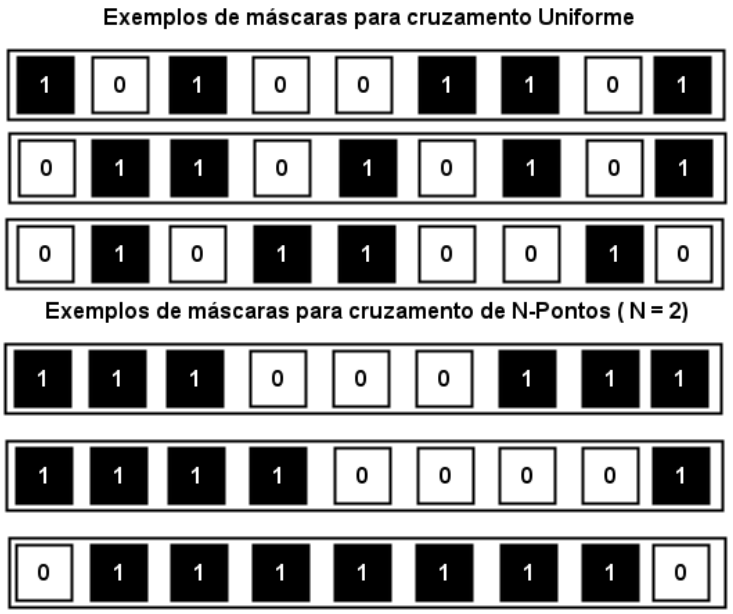

Figura 2.7: Exemplos de máscaras de cruzamento.

Na Listagem 2.8 é apresentada uma implementação do cruzamento de n-pontos com $n = 2$ em Python, onde foi utilizada a abordagem que faz uso de máscaras.

Listagem 2.8: Código Python para cruzamento de *n-pontos*

```
from random import choice, sample

population_size = 50
population = [
        "".join(choice(['0','1']) for j in range(19))
        for i in range(population_size)
]

def crossover(d,m,n=2):
        if len(d) != len(m):
                raise Exception()
        ps = [0] + sorted(sample(range(19),n)) + [len(d)]
        mask = "".join([
                ('1' if i % 2 == 0 else '0') * (ps[i] - ps[i-1])
                for i in range(1,len(ps))
        ])
        return map("".join, zip(*[
                (d[i],m[i]) if mask[i] == '1' else (m[i],d[i])
                for i in range(len(d))
        ]))
```

2.1.6 Cruzamento com Representação Real

Um dos métodos mais utilizados para o cruzamento na representação real é o cruzamento mistura ou BLX- α[5] (GALVÃO, 1999; ESHELMAN; SCHAFFER, 1993). Para o caso da representação real exemplificada na Seção 2.1.1.2, o cruzamento BLX-α pode ser realizado de acordo com a Equação 2.12, onde C_n é um novo cromossomo e C_1 e C_2 os cromossomos geradores.

$$C_n = C_1 + \beta(C_2 - C_1) \Big| \ \beta \ \in \sim U([-\alpha; 1+\alpha]) \tag{2.12}$$

Em casos em que a representação real contém mais de um gene, essa equação é replicada para cada um deles. O valor de β pode variar para cada gene, ou ser o mesmo para todos. Na

[5] Do inglês *blend crossover*

Listagem 2.9 é apresentada uma implementação em Python para o cruzamento BLX- α com $\alpha = 0,5$, utilizando a representação real descrita na Seção 2.1.1.2.

Listagem 2.9: Código Python para cruzamento BLX-α

```
1  from random import uniform
2  import math
3
4  P = 50
5  alpha = 0.5
6  domain = [-20,20]
7
8  population = [
9          uniform(*domain)
10         for i in range(P)
11 ]
12
13 def crossover(dad,mum):
14         beta = uniform(-alpha,1+alpha)
15         return min(max(dad + beta * (mum - dad),domain[0]),domain[1])
```

2.1.7 Exemplos de Algoritmos Genéticos

Com o conteúdo explicado durantes as seções anteriores já é possível realizar a implementação de um AG. Para melhorar o entendimento da sequencia de aplicação e funcionamento das operações, serão apresentados exemplos de implementações de AGs.

Na Listagem 2.10 é apresentada uma implementação em Python de um AG com todas as etapas funcionais. Esse AG está sendo aplicado na maximização da Equação 2.1, utilizando representação binária com 19 *bits* de precisão, taxa de mutação igual á 1%, taxa de cruzamento de 80%, população com 50 indivíduos, cruzamento uniforme e seleção por torneio. A quantidade de gerações foi limitada a 100.

Listagem 2.10: Código Python de um AG completo, com representação binária, seleção por torneio e cruzamento uniforme

```
1  import math
2  from random import choice,uniform,shuffle,sample
3
4  P, L, CR, MR = 50, 19, 0.8, 0.01
5  domain = [-20,20]
6  population = [
7          "".join(choice(['0','1']) for j in range(L))
```

```
8           for i in range(P)
9   ]
10
11  def decode(chromossome):
12          b10 = int(chromossome,2)
13          return b10 * (domain[1] - domain[0]) / (2**L -1.0) + domain[0]
14
15  def f(x):
16          return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
17
18  def fitness(chromossome):
19          return f(decode(chromossome))
20
21  def tournament_selection(pop,p=P,n=3):
22          selected = []
23          for i in range(P):
24                  group = sample(pop,n)
25                  selected += [max(*group,key=fitness)]
26          return selected
27
28  def mutate(chromossome,rate=MR):
29          return "".join(map(str,[
30                  int(gen,2) ^ ( 1 if uniform(0,1) < rate else 0 )
31                  for gen in chromossome
32          ]))
33
34  def crossover(d,m):
35          if len(d) != len(m):
36                  raise Exception()
37          return map("".join,zip(*[
38                  (d[i], m[i]) if choice([True,False]) else (m[i],d[i])
39                  for i in range(len(d))
40          ]))
41
42  for i in range(100):
43          sp = tournament_selection(population)
44          shuffle(sp)
45          for i in range(0,len(sp),2):
46                  dad,mum = i,(i+1)%(len(sp))
47                  if uniform(0,1) > CR:
48                          continue
49                  sp[dad],sp[mum] = crossover(sp[dad],sp[mum])
50          for i in range(0,len(sp)):
```

```
51                  sp[i] = mutate(sp[i])
52          population = sp
53
54  best = max(*population,key=fitness)
55  print best,decode(best),fitness(best)
```

Na Listagem 2.11 é apresentada uma implementação em Python de um AG com todas as etapas funcionais. Esse AG está sendo aplicado na maximização da Equação 2.1, utilizando representação real, taxa de cruzamento de 80%, população com 50 indivíduos, cruzamento BLX-α com $\alpha = 0,5$, mutação *creep* com taxa de 1% e seleção por torneio. A quantidade de gerações foi limitada a 100.

Listagem 2.11: Código Python de um AG completo, com representação real, seleção por torneio e cruzamento BLX-α

```
1   from random import uniform,sample,shuffle
2   import math
3
4   alpha = 0.5
5   domain = [-20,20]
6   P, CR, MR = 50, 0.8, 0.01
7
8   population = [
9           uniform(*domain)
10          for i in range(P)
11  ]
12
13  def crossover(dad,mum):
14          beta = uniform(-alpha,1+alpha)
15          return min(max(dad + beta * (mum - dad),domain[0]),domain[1])
16
17  def f(x):
18          return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
19
20  def mutate(chromossome,rate=0.01):
21          cm = chromossome * (1 + uniform(-rate/2,rate/2))
22          return max(min(domain[1],cm),domain[0])
23
24  def tournament_selection(pop,p=P,n=3):
25          selected = []
26          for i in range(P):
27                  group = sample(pop,n)
28                  selected += [max(*group,key=f)]
```

```
29          return selected
30
31  for i in range(100):
32          sp = tournament_selection(population)
33          shuffle(sp)
34          for i in range(0,len(sp),2):
35                  dad,mum = i,(i+1)%(len(sp))
36                  if uniform(0,1) > CR:
37                          continue
38                  sp[dad] = crossover(sp[dad],sp[mum])
39                  sp[mum] = crossover(sp[mum],sp[dad])
40          for i in range(0,len(sp)):
41                  sp[i] = mutate(sp[i])
42          population = sp
43
44  best = max(*population,key=f)
45  print best,f(best)
```

2.1.8 AGs Distribuídos

Devido ao baixo desempenho dos AGs, que podem ser ainda mais prejudicados com grandes populações ou funções de aptidão de alto custo computacional, muitos modelos de implementações paralelas vem sendo propostos. Esses modelos utilizam desde computadores paralelos até a distribuição por rede de computadores, tendo basicamente dois tipos de classificação, AGs insulares e AGs celulares.(TANOMARU, 1995)

2.1.8.1 AGs Insulares

Os AGs insulares, são praticamente vários AGs trabalhando de forma cooperativa. Cada AG possui sua própria população e periodicamente os melhores indivíduos de cada população são compartilhados. O compartilhamento de indivíduos pode ser interpretado como uma metáfora biológica para a migração. Esse modelo de implementação é relativamente simples, podendo cada AG ser executado em um processador ou computador diferente.(TANOMARU, 1995)

Na Figura 2.8 é ilustrado a ideia dos AGs insulares, vários AGs independentes com suas próprias populações cooperando entre si através da migração de indivíduos. Assim, é possível que cada AG trabalhe com configurações diferentes, como, tamanho da população, método de seleção, taxa de mutação, taxa de cruzamento e etc. A única restrição é que a representação cromossômica seja a mesma, ou que os AGs estejam preparados para trabalharem com

representações cromossômicas diferentes.

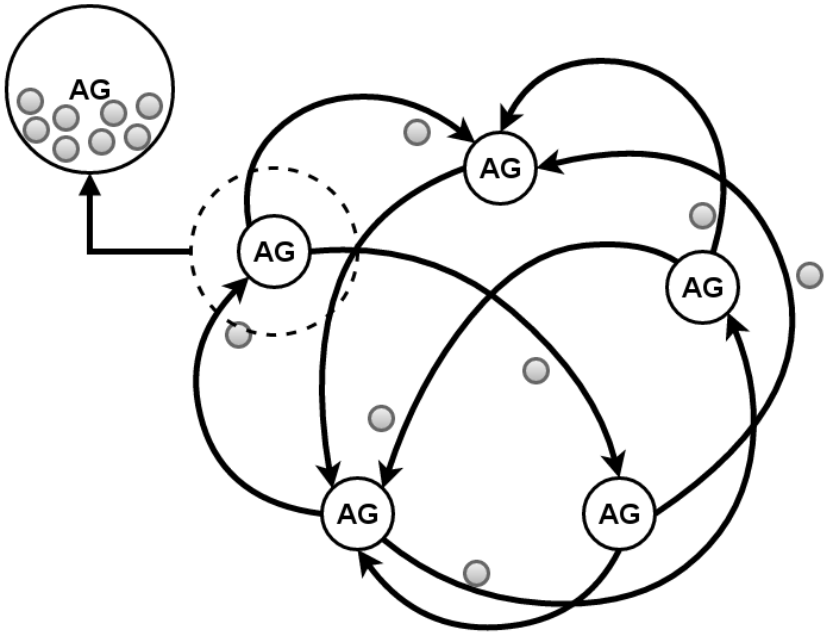

Figura 2.8: Algoritmo Genético Insular

2.1.8.2 AGs Celulares

Os AGs celulares, também chamados de AGs massivamente paralelos ou AGs de granularidade fina, podem ser considerados uma ramificação de autômatos celulares [6]. O princípio desse modelo, é que idealmente cada processador controle apenas um indivíduo e que cada processador realize todas as etapas do AG, seleção, reprodução e avaliação. Esses processadores são distribuídos em uma malha e são interconectados, como exemplificado na Figura 2.9. Com essas interconexões, os processadores tem acesso aos indivíduos controlados por seus vizinhos. Assim, apesar de cada processador controlar apenas um indivíduo, todos tem acesso a $n+1$ indivíduos, sendo n a quantidade de vizinhos, no caso da Figura 2.9, totalizando 5 indivíduos. Esses indivíduos são utilizados na etapa de seleção de cada processador, ou seja, a seleção dos indivíduos é feita dentro da vizinhança dos processadores. (TANOMARU, 1995)

Assim como com os autômatos celulares, a forma de distribuir os processadores pela malha pode variar. Por exemplo, a malha poderia ser distribuída em apenas uma dimensão, ou em três, ou qualquer outro número inteiro positivo. As vizinhanças formadas pela malha, formam pequenas subpopulações que estão contidas na população total, união de todos os indivíduos da malha de processadores. Essas subpopulações estão constantemente migrando seus indivíduos para as subpopulações vizinhas, de acordo com a seleção dos processadores.

[6] Veja mais sobre autômatos celulares na Seção 2.3.

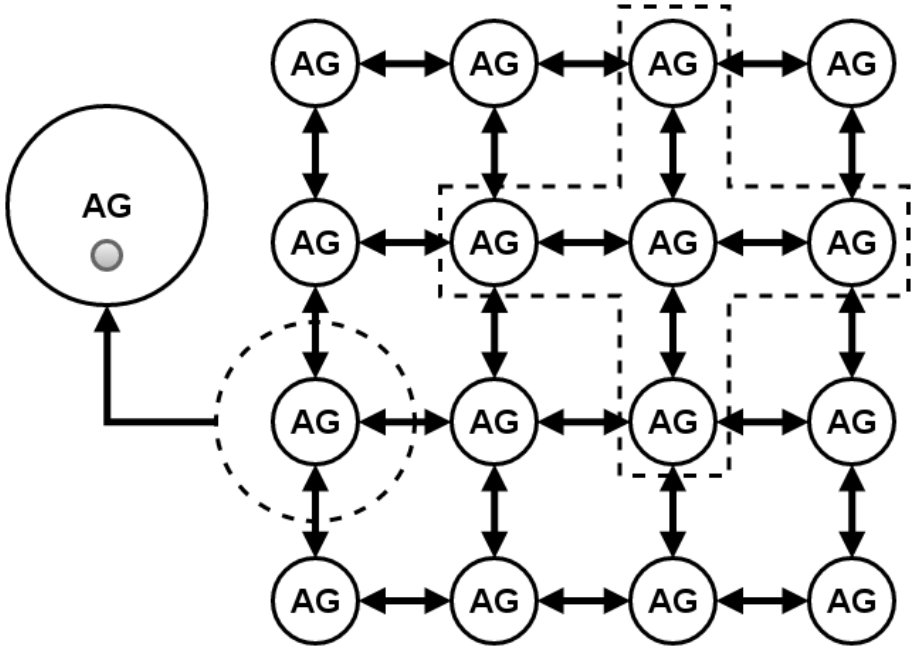

Figura 2.9: Algoritmo Genético Celular

Como há um processador para cada indivíduo da população, é possível realizar cada etapa do AG ao mesmo tempo. Todos os indivíduos são avaliados, em seguida todos são selecionados e por fim são reproduzidos, paralelamente.

2.1.9 Algoritmo Genético Compacto

Os AGs Compactos economizam memória utilizando um método alternativo para representar a população e não simplesmente armazenando uma coleção de indivíduos. Ao invés de uma população tradicional, é utilizado um vetor de probabilidades. Esse vetor de probabilidades dá origem a todos os indivíduos que o AG necessitar. Geralmente os AGs compactos são baseados em AGs *steady-state*, onde a cada geração um número limitados de indivíduos da população são substituídos e não toda a população. No caso dos AGs compactos, ao invés da substituição de indivíduos, o vetor de probabilidades é adequado. (HARIK; LOBO; GOLDBERG, 1999; APORNTEWAN; CHONGSTITVATANA, 2001)

Na Equação 2.13 é apresentado como é realizada a representação da população de um AG compacto. Sendo $\mathbb{P}$ a população, p_i uma probabilidade e n o comprimento dos indivíduos.

$$\mathbb{P} = [p_{n-1}, p_{n-2}, ..., p_0] \Big| p_i \in [0;1] \tag{2.13}$$

Assim um novo indivíduo pode ser gerado de acordo com a Equação 2.14. Sendo C o novo

cromossomo e c_i seus genes.

$$C = [c_{n-1}, c_{n-2}, ..., c_0] \left| c_i = \begin{cases} 0 \; se \; \sim U([0;1]) \geq p_i \\ 1 \; se \; \sim U([0;1]) < p_i \end{cases} \right. \tag{2.14}$$

A adequação das probabilidades pode ser realizada com a Equação 2.15. Onde Φ é a taxa de aprendizado.

$$p_i = p_i + \Phi(2c_i - 1) \; \forall \; p_i \in \mathbb{P}, \; c_i \in C \tag{2.15}$$

Na Listagem 2.12 é apresentado um código em Python que implementa um AG Compacto, considerando a representação binária utilizada na Seção 2.1.1.1 com 19 *bits*, a Equação 2.1 como função objetivo e uma taxa de aprendizado igual á $0,0005$.

Listagem 2.12: Código Python de um Algoritmo Genético Compacto

```
1  import math
2  from random import uniform
3
4  L, R, N = 19, 0.0005, 2
5  domain = [-20.0,20.0]
6  P = [0.5] * L
7
8  def gen():
9          return "".join([
10                 '1' if uniform(0,1) < P[i] else '0'
11                 for i in range(L)
12         ])
13
14 def decode(chromossome):
15         b10 = int(chromossome,2)
16         return b10 * (domain[1] - domain[0]) / (2**L -1.0) + domain[0]
17
18 def f(x):
19         return (20.0 * math.cos(x*math.pi))/(abs(x/2) + 2)
20
21 def fitness(chromossome):
22         return f(decode(chromossome))
23
```

```
24 def coach(chromossome):
25         for i in range(L):
26                 P[i] += R*(2*int(chromossome[i],2)-1)
27
28 def converged():
29         for i in range(L):
30                 if 0 < P[i] < 1:
31                         return False
32         return True
33
34 while not converged():
35         group = [gen() for i in range(N)]
36         best = max(*group,key=fitness)
37         coach(best)
38
39 best = gen()
40 print best,decode(best),fitness(best)
```

2.1.10 Tópicos Especiais

Alguns conceitos comumente abordados dentro do contexto de AGs são descritos sucintamente a seguir:

- Convergência Prematura - A cada geração a pressão seletiva faz que alguns indivíduos sejam eliminados da população. A eliminação desses indivíduos pode acarretar no desaparecimento de possíveis genes, os alelos. Isso pode fazer com que o espaço de busca não seja totalmente explorado e, consequentemente o AG pode convergir para uma solução não global. A convergência para soluções não globais é chamada de *convergência prematura*. A mutação e uma pressão seletiva balanceada, podem ajudar a manter a diversidade genética, assim, evitando esse tipo de problema. (GALVÃO, 1999)

- Super-indivíduos - No decorrer da execução de um AG, indivíduos com boas aptidões começam a ser encontrados, mas nem sempre esses indivíduos representam a melhor solução do problema. Porém, esses indivíduos possuem maior chande de espalhar seus genes e, muitas vezes, acabam dominando grande parte da população. Esses indivíduos são chamados de *super-indivíduos*. São indivíduos que, a cada geração espalham mais e mais seus genes pela população e não necessariamente a encaminha para a melhor solução. Na Figura 2.10 é exemplificado um caso em que, um super indivíduo fez com

que a população se concentrasse para o lado direito da função objetivo, em máximos locais, atrapalhando a convergência para o máximo global.

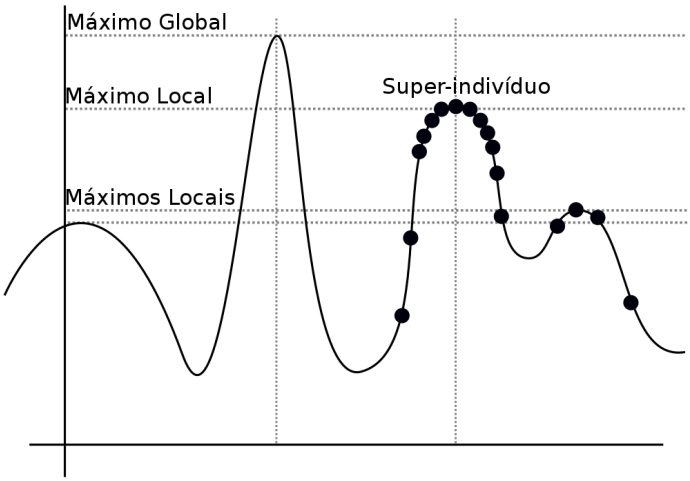

Figura 2.10: Exemplo de Super-indivíduo.

- Nichos - A pressão seletiva na maior parte dos AGs, age sobre toda a população, tornando todos os indivíduos concorrentes. Porém, na natureza espécies distintas com papéis (*nichos*) ecológicos diferentes, não competem entre si podendo conviver no mesmo ambiente. Uma das maneiras de aplicar esse principio nos AGs, é modificar a função de adequabilidade, fazendo com que a competição entre indivíduos distantes um do outro no espaço de busca seja menor. Essa abordagem faz com que o AG tenha tendencia a criar subpopulações, cada uma concentrada em um pico da função objetivo. Esse comportamento foi exemplificado na Figura 2.11, onde se formaram 4 subpopulações nos picos da função objetivo. Sem essa abordagem, a tendencia é que a população toda se concentre em apenas um pico, que não necessariamente seria o maior. (TANOMARU, 1995)

- Esquemas - Os esquemas são padrões encontrados nos códigos genéticos dos indivíduos da população. Esquemas presentes em indivíduos com boas aptidões tendem a se espalhar mais que outros pela população (TANOMARU, 1995; GALVÃO, 1999). Para entender melhor o que são e o comportamento dos esquemas, foi inicialmente desenvolvido por Holland (1975) o teorema dos esquemas. Alguns pesquisadores acreditam que a compreensão do teorema dos esquemas pode melhorar a utilização dos AGs e/ou ajudar a criar novas operações de cruzamento, mutação e outras mais. Porém, Tanomaru (1995) afirma que o teorema dos esquemas ainda não trouxe contribuições significativas.

- Elitismo - A ideia do elitismo é manter os indivíduos mais aptos na população. Esses poderiam ser perdidos devido a mutação e/ou o cruzamento durante a geração de uma nova

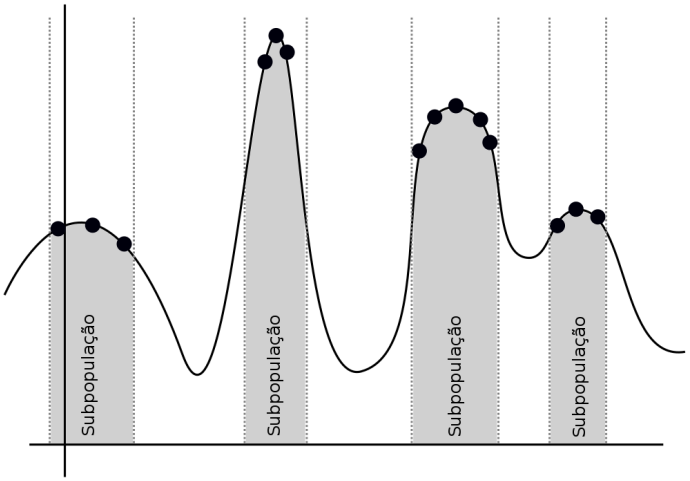

Figura 2.11: Exemplo da Formação de Subpopulações na Função Objetivo

população. O elitismo foi proposto por DeJong (1975) e hoje é comumente aplicado na maioria das implementações de AGs. A cada geração, o indivíduo, ou os indivíduos mais aptos, devem ser copiados de uma geração para outra, assim, é garantido que as melhores soluções já encontradas não serão perdidas. Essa abordagem ajuda os AGs convergirem mais rapidamente para uma solução. (GALVÃO, 1999; TANOMARU, 1995)

2.2 Morfologia Matemática

Morfologia é a forma e estrutura de um objeto ou o modo como suas partes se relacionam. Basicamente, com morfologia matemática é possível extrair informações topológicas e geométricas de um conjunto desconhecido, como uma imagem, realizando transformações com um outro conjunto previamente definido, denominado *elemento estruturante* (FILHO; NETO, 1999). Com esse tipo de abordagem é possível detectar, realçar ou ocultar formas.

A teoria original da morfologia matemática foi desenvolvida para imagens binárias, estendida para imagens níveis de cinza e, posteriormente, para reticulados completos, possibilitando a aplicação em imagens coloridas. A morfologia matemática, diferente da maioria dos outros métodos de análise de imagens, consegue extrair informações de formas geométricas. (CANDEIAS; BANON, 1997; KIM, 1997)

A representação lógica de uma imagem colorida tem uma complexidade bem maior que a de uma imagem binária e a implementação dos operadores morfológicos para imagens coloridas não é algo trivial (COMER; DELP, 1999). Consequentemente, sua operações tem maior custo computacional. Para simplificar e otimizar o desenvolvimento do processador morfoló-

gico apresentado no Capítulo 3.5, optou-se pelo desenvolvimento de uma arquitetura capaz de trabalhar somente com imagens binárias. Por esse motivo, esta seção fundamenta somente a morfologia matemática para imagens binárias, não descrevendo as operação para imagens tons de cinza e coloridas.

Dentre as operações da morfologia matemática, existem duas que podem ser vistas como operações básicas para a construção de operações complexas, sendo elas a dilatação e a erosão. Com a combinação dessas duas operações e mais algumas operações lógicas entre imagens binárias, é possível tratar muitos problemas práticos de análise de imagens (BANON; BARRERA, 1994).

A seguir são brevemente descritas essas duas principais operações da morfologia matemática, a dilatação e a erosão. Também são descritas algumas operações que são derivadas ou compostas por elas. As definições utilizadas foram extraídas dos trabalhos de Pedrino (2008), Gonzalez e Woods (2000), Banon e Barrera (1994) e Wangenheim (2001).

2.2.1 Dilatação Binária

A dilatação tem como característica principal alargar objetos e preencher suas lacunas. Ela combina dois objetos utilizando adição vetorial e é, geralmente, representada pelo símbolo $\oplus$. Segundo Wangenheim (2001) a dilatação de um conjunto A por um conjunto B pode ser definida pela Equação 2.16.

$$A \oplus B = \{c | c = a + b, a \in A, b \in B\} \tag{2.16}$$

Na Equação 2.16, A é a imagem alvo da dilatação e B o elemento estruturante. Em outras palavras, a implementação da dilatação consiste na união das imagens resultantes das translações da imagem original pelo elemento estruturante. O efeito ocorrido, após uma dilatação em uma imagem binária, pode ser visto na Figura 2.12. Nessa figura é perceptível a expansão dos objetos, a diminuição de seus orifícios, causando até o desaparecimento dos menores, e a redução das distâncias entre os objetos, criando até algumas conexões entre alguns.

Na Figura 2.13 é possível notar a diferença que o elemento estruturante pode causar. Nessa figura dois objetos retangulares são dilatados por dois elementos estruturantes diferentes, um quadrado e um círculo, gerando resultados distintos. O retângulo dilatado por um círculo resultou em um retângulo maior com os cantos arredondados, enquanto o retângulo dilatado pelo quadrado ficou maior e manteve os cantos quadrados.

Figura 2.12: Exemplo do resultado de uma dilatação (WANGENHEIM, 2001).

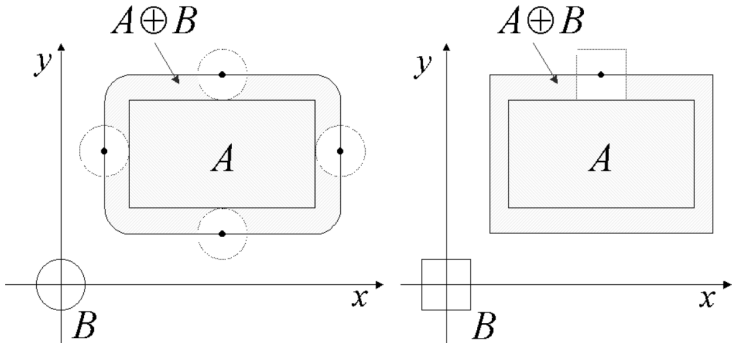

Figura 2.13: Processo de Dilatação (PEDRINO, 2008).

A dilatação pode ser descrita com álgebra booleana como uma soma de produtos, como representado na Figura 2.14. Nessa figura, a matriz *i* representa a imagem a ser dilatada e a matriz *e* representa o elemento estruturante. Para cada *pixel* da imagem é necessário a replicação dessa soma de produtos. Nesse caso, o *pixel* que está sendo calculado é o do centro da parte mais escura da matriz *i*, localizado na segunda linha (linha 1) e terceira coluna (coluna 2). A parte mais escura são os *pixels* utilizados no cálculo da dilatação do *pixel* em questão.

2.2.2 Erosão Binária

A erosão tem como característica principal afinar objetos e aumentar as lacunas, mas não é o inverso da dilatação, ou seja, uma dilatação não necessariamente anula o efeito de uma erosão. A erosão combina duas imagens utilizando vetores de subtração e é, geralmente, representada pelo símbolo $\ominus$. Segundo Wangenheim (2001) a erosão de um conjunto A por um conjunto B pode ser representada pela Equação 2.17.

$$A \ominus B = \{c | c + b \in A \ \forall \ b \in B\} \tag{2.17}$$

A implementação da Equação 2.17 consiste na intersecção das imagens resultantes das

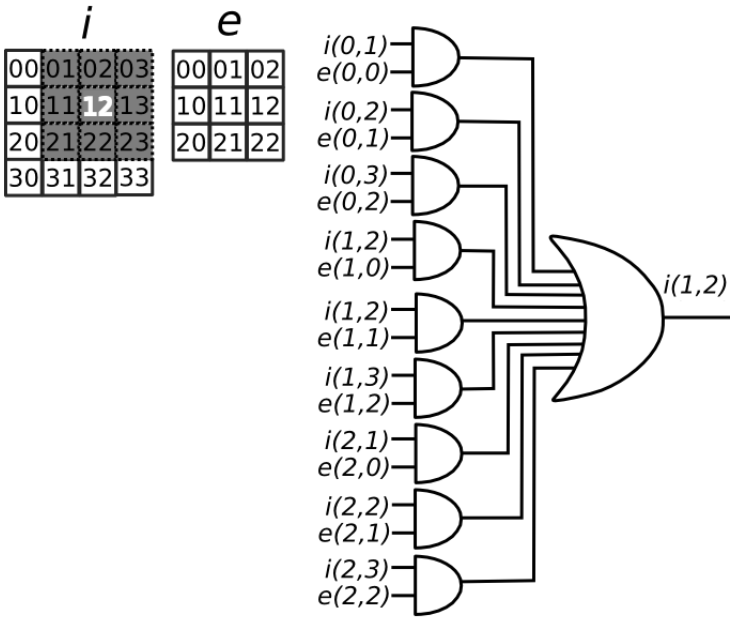

Figura 2.14: Dilatação binária com lógica digital, em que *i* é a imagem e *e* o elemento estruturante.

translações da imagem original *A* pelo elemento estruturante *B*. O efeito ocorrido após uma erosão pode ser visto na Figura 2.15. O resultado é quase o inverso de uma dilatação, os objetos foram encolhidos, os orifícios expandidos e aumentou-se a distância entre os objetos. Alguns orifícios até romperam as bordas de seus objetos e uma conexão estreita entre dois objetos foi removida.

Figura 2.15: Exemplo de Erosão (WANGENHEIM, 2001).

Na Figura 2.16 é demonstrado o processo de erosão de um objeto retangular *A* por um elemento estruturante circular *B*.

A erosão também pode ser descrita com álgebra booleana como um produto de somas, como representado na Figura 2.17. Na figura a imagem a ser erodida é representada pela matriz *i* e a matriz *e* representa o elemento estruturante. Para cada *pixel* da imagem é necessário replicar esse produto de somas. No caso dessa imagem, o *pixel* que está sendo calculado é o do centro da parte mais escura da matriz, localizado na segunda linha (linha1) e terceira coluna (coluna 2). A parte mais escura são os *pixels* utilizados durante o cálculo da erosão do *pixel* em questão.

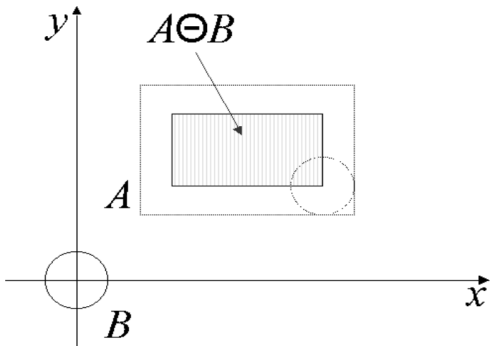

Figura 2.16: Exemplo do processo de erosão (PEDRINO, 2008).

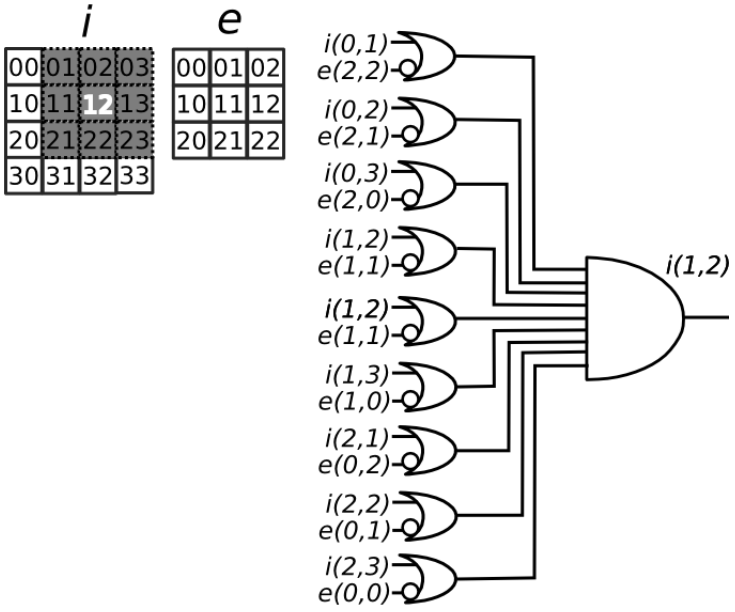

Figura 2.17: Erosão binária com lógica digital, em que i é a imagem e e o elemento estruturante.

2.2.3 Dualidade entre Dilatação e Erosão

Há uma relação de dualidade entre as operações de dilatação e erosão. Pode-se dizer que a erosão de uma imagem por um elemento estruturante, gera o mesmo resultado que a dilatação do fundo da imagem pelo mesmo elemento estruturante refletido. O mesmo acontece com a dilatação. A dilatação de uma imagem por um elemento estruturante tem o mesmo efeito que uma erosão do fundo da imagem pelo mesmo elemento estruturante refletido. Assim é possível realizar uma dilatação utilizando uma erosão, como sugere a Equação 2.18, e também é possível, realizar uma erosão com uma dilatação, como sugere a Equação 2.19. Em ambas as equações ^{C} indica complemento e ^{R} reflexão.

$$A \oplus B = (A^C \ominus B^R)^C \tag{2.18}$$

$$A \ominus B = (A^C \oplus B^R)^C \tag{2.19}$$

2.2.4 Abertura Binária

A abertura é uma das operações derivadas da combinação da dilatação e da erosão. Geralmente, a abertura suaviza contornos, quebra istmos estreitos, elimina proeminências delgadas e alguns tipos de ruídos (WANGENHEIM, 2001). A abertura é a combinação das operações de erosão e dilatação utilizando o mesmo elemento estruturante, nessa ordem. Segundo Gonzalez e Woods (2000), pode ser definida pela Equação 2.20.

$$A \circ B = (A \ominus B) \oplus B \tag{2.20}$$

Uma característica importante da abertura é que ela é idempotente, ou seja, sua aplicação repetidas vezes não cria um novo efeito, como representado na Equação 2.21.

$$A \circ B = (A \circ B) \circ B \tag{2.21}$$

Na Figura 2.18 foi ilustrado um exemplo do resultado de uma abertura. É possível observar que alguns pequenos orifícios dos objetos acabaram se tornando maiores, inclusive, rompendo as bordas dos objetos. Após a abertura, também pode ser visto que uma pequena linha desapareceu, separando dois objetos da figura.

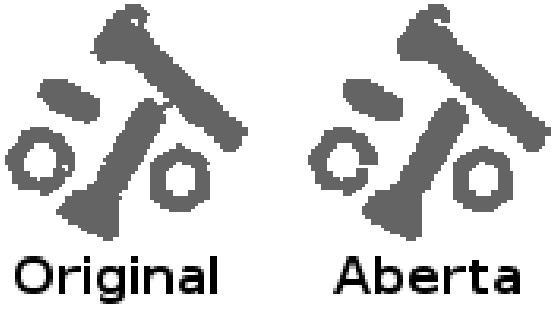

Figura 2.18: Exemplo do resultado de uma abertura. Adaptado de Wangenheim (2001)

2.2.5 Fechamento Binário

O fechamento, assim como a abertura, também é derivado das operações de dilatação e erosão. Ele pode ser utilizado para alargar golfos estreitos, unir pequenas quebras, eliminar pequenos orifícios e alguns tipos de ruídos (WANGENHEIM, 2001). O fechamento é a combinação das operações de dilatação e erosão utilizando o mesmo elemento estruturante, nessa ordem. Segundo Gonzalez e Woods (2000), pode ser definido pela Equação 2.22.

$$A \bullet B = (A \oplus B) \ominus B \tag{2.22}$$

Assim como a abertura, o fechamento também é idempotente, como demonstra a Equação 2.23.

$$A \bullet B = (A \bullet B) \bullet B \tag{2.23}$$

Na Figura 2.19 é ilustrado um exemplo do resultado de um fechamento. Pode ser visto que alguns orifícios dos objetos foram preenchidos, uma nova conexão entre dois objetos foi criada e que uma pequena linha que conectava dois objetos foi expandida.

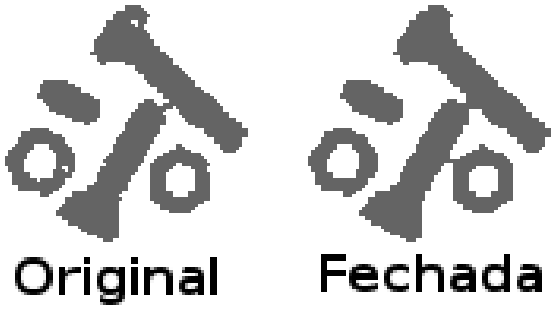

Figura 2.19: Exemplo do resultado de um fechamento. Adaptado de Wangenheim (2001)

2.2.6 Outras Operações

Utilizando operações morfológicas e lógicas é possível implementar diversos filtros. Por exemplo, para extrair as bordas internas de um objeto, pode-se realizar a operação descrita na Equação 2.24.

$$C = (A \ominus B) - A \tag{2.24}$$

Na Equação 2.24, A é a imagem original, B um elemento estruturante não vazio e C as bordas internas de A. A operação de subtração pode ser facilmente substituída por uma operação lógica *ou exclusivo*, no caso de imagens binárias.

Para a extração das bordas externas, pode-se realizar uma operação semelhante, basicamente substituindo a erosão pela dilatação, como descrito na Equação 2.25.

$$C = (A \oplus B) - A \tag{2.25}$$

Os resultados obtidos com essas duas equações podem ser vistos na Figura 2.20.

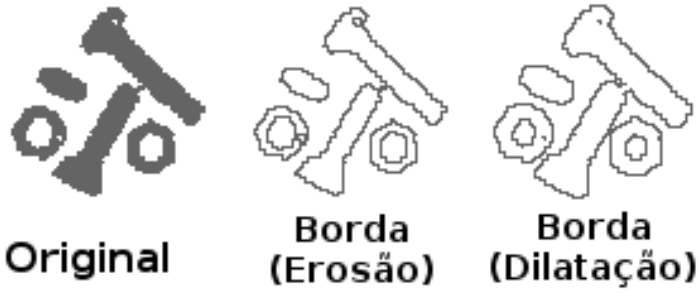

Figura 2.20: Exemplo de detecção de borda com erosão e dilatação. Adaptado de Wangenheim (2001)

2.3 Autômatos Celulares

Um autômato celular (AC) é um modelo matemático discreto, inspirado nas interações entre células em um organismo vivo. John Von Neumann e Stannislaw Ulam foram os pioneiros dos conceitos de ACs, visando o desenvolvimento de mecanismos artificias reprodutivos. John Conway em 1970 apresentou o jogo da vida (*game of life*), que é um AC binário bidimensional, e demonstrou como regras simples aplicadas inúmeras vezes sobre estados aleatórios, podem se assemelhar a sistemas presentes no mundo real. (WEINERT, 2010)

> Os Autômatos Celulares têm sido estudados e aplicados em diversos campos do conhecimento, como: Ciência da Computação, Física, Química, Geografia, Biologia, Ciências Sociais, entre outras e é uma importante ferramenta para

> simular e estudar sistemas físicos, químicos e biológicos, vida artificial, computadores universais, teoria de sistemas dinâmicos, estudos sobre dinâmicas populacionais. (GREMONINI; VICENTINI, 2008)

Basicamente um autômato celular é uma malha de *n*-dimensões de autômatos, podendo ser *n* qualquer número inteiro positivo. Cada autômato nessa malha é chamado de célula e, cada uma possui seu próprio estado e uma regra de transição. As regras de transição de cada célula se baseiam no estado da célula atual e nos estados das células vizinhas. Em geral essas regras são aplicadas simultaneamente por toda a malha. Os vizinhos de uma célula podem ser determinados por diversas regras, geralmente envolvendo a distância entre as células e sua geometria.

Os estados dos autômatos devem ser discretos, determinísticos e finitos, sendo assim, passiveis de implementação computacional. No caso do autômato celular proposto por John Von Neuman existia um total de 29 estados. Porém, suas regras nunca foram implementadas em um computador, por serem muito complexas. (GREMONINI; VICENTINI, 2008)

Neste projeto pretende-se explorar a utilização de autômatos celulares na geração de números pseudoaleatórios. ACs unidimensionais binários são de simples implementação em *hardware*. Como a regra de cada autômato ou célula deve ser aplicada simultaneamente, torna-se intuitivo o desenvolvimento de um *hardware* paralelo para sua implementação.

2.3.1 Dimensão

Os autômatos celulares podem ter qualquer dimensão inteira positiva, com apenas uma dimensão as células podem ser dispostas em uma linha, com duas dimensões as células podem ser organizadas em um plano como uma matriz e com três dimensões as células podem ser distribuidas pelo espaço, sendo esse o modelo mais semelhante com células em um organismo. Dimensões superiores a terceira são passiveis de implementação, mas, não são facilmente compreendidas, sendo difícil visualizar seu funcionamento. A quantidade de vizinhos de cada célula está fortemente relacionada com a quantidade de dimensões do autômato celular, quanto mais dimensões mais complexas poderão ser suas regras e consequentemente o comportamento das células.

Para exemplificar as dimensões de um AC, na Figura 2.21 são ilustrados ACs de uma, duas e três dimensões. Nessa figura cada célula é representada por um quadrado, nos casos dos autômatos de uma e duas dimensões, enquanto no AC de três dimensões, cada célula é representada por um cubo. Os níveis de cinza utilizados nos autômatos de uma e duas dimensões, embora não citado pelo autor da imagem, provavelmente representam o estado atual de cada célula.

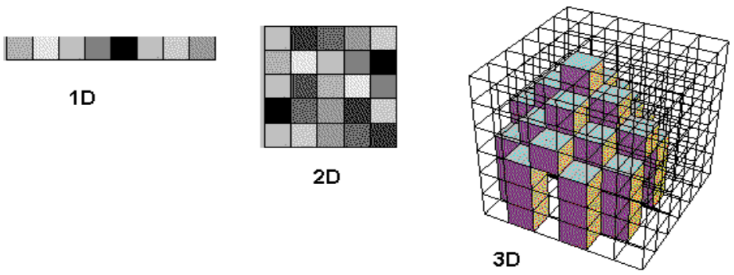

Figura 2.21: Dimensões de Autômatos Celulares (GREMONINI; VICENTINI, 2008).

2.3.2 Formato

Um outro fator que pode influenciar a vizinhança, além da quantidade de dimensões do AC, é o formato das células. Na Figura 2.22 são ilustrados alguns formatos de células, sendo esses, triangular, quadrangular e hexagonal. É intuitivo pensar que no caso de ACs bidimensionais a quantidade de vizinhos é igual ao número de lados da forma geométrica adotada, mas isso nem sempre é o correto.

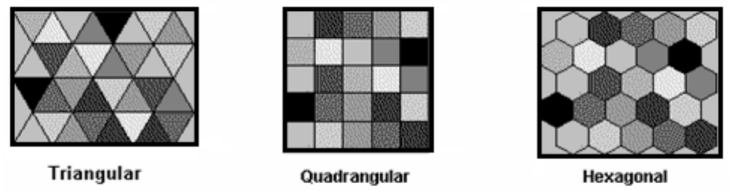

Figura 2.22: Formato de Células (GREMONINI; VICENTINI, 2008).

Continuando com os ACs bidimensionais, que de certa forma inspiram a maior parte dos ACs, é normal de se considerar dois tipos de vizinhança, a de Von-Neumann e a de Moore. A vizinhança de Von-Neumann considera células vizinhas, as imediatamente mais próximas a norte, sul, leste e oeste. A vizinhança de Moore considera vizinhas todas as células em contato. (TEIXEIRA, 1996)

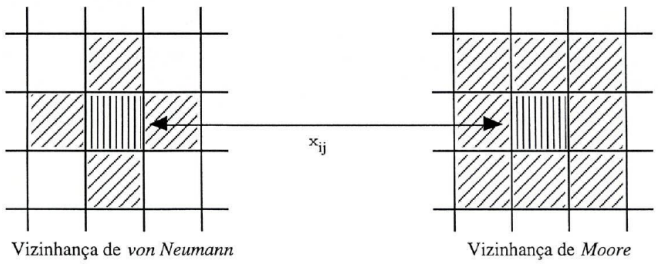

Figura 2.23: Tipos de Vizinhaça (TEIXEIRA, 1996)

Na Figura 2.23 é ilustrada uma comparação entre os dois modelos de vizinhança, de Moore e de Von-Neumann para o caso de um AC bidimensional em que as células tem a forma de um

quadrado, assim, a vizinhança de Moore tem um total de oito vizinhos e a de Von-Neumann quatro vizinhos.

2.3.3 Classificação

Stephen Wolfram classificou os ACs em 4 classes, analisando como os autômatos evoluem a partir de uma sequencia binária aleatória. A primeira classe inclui os autômatos que conforme a evolução temporal acabam se estabilizando de forma homogênea, ou seja, todas as células convergem para um mesmo estado. Na segunda classe a evolução do autômato o leva a um estado estável, cíclico e não homogêneo. Na terceira classe os autômatos criam padrões caóticos não aleatórios, mas com padrão não reconhecível. Na quarta classe, a evolução temporal faz com que o autômato gere estruturas complexas imprevisíveis. (TEIXEIRA, 1996; GREMONINI; VICENTINI, 2008)

2.3.4 ACs Binários Unidimensionais

Os estados possíveis de cada célula em um AC binário unidimensional são apenas um ou zero e geralmente cada célula possui dois vizinhos. Os ACs binários unidimensionais são frequentemente utilizados para a geração de fractais. Alguns exemplos podem ser vistos nas figuras 2.24, 2.25 e 2.26, onde são apresentados resultados de ACs binários unidimensionais de regras 30, 90 e 150, respectivamente. Em cada figura são apresentados dois padrões. Os padrões do lado esquerdo, são iniciados com apenas uma célula com estado 1, sendo ela a do centro. Os padrões do lado direito, são iniciados com valores aleatórios. As linhas da imagens representam a dimensão temporal do AC, ou seja, o histórico dos estados. As colunas mostram os estados de cada célula, preto para o valor um e branco para o valor zero.

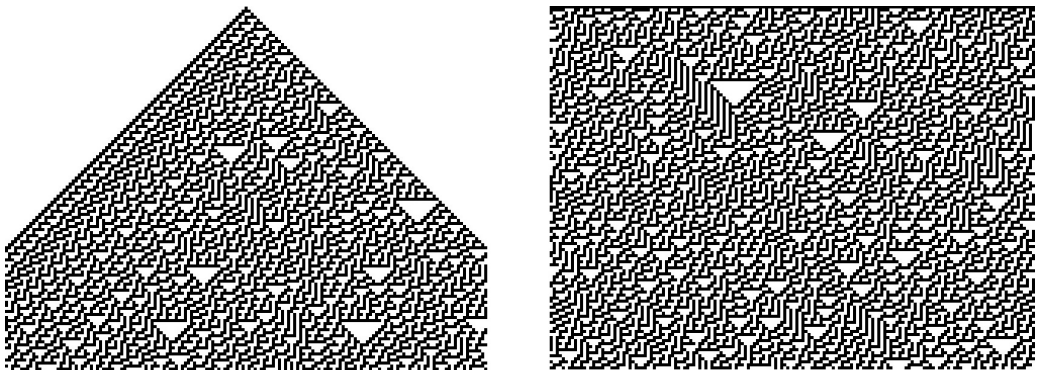

Figura 2.24: AC Binário Unidimensional de regra 30

Esses ACs com esse tipo de vizinhança, podem ter 256 regras diferentes. Cada célula possui

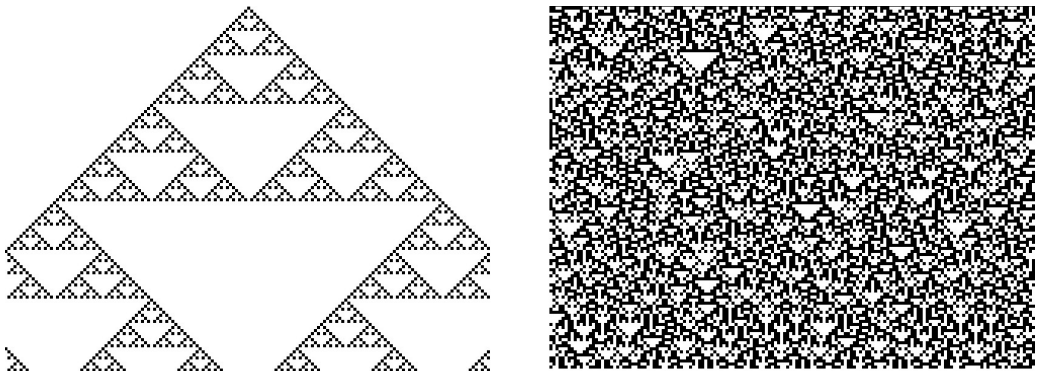

Figura 2.25: AC Binário Unidimensional de regra 90

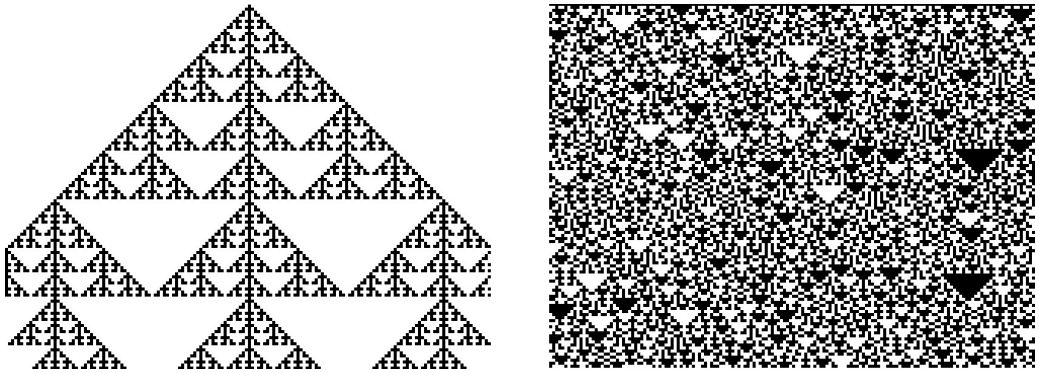

Figura 2.26: AC Binário Unidimensional de regra 150

duas células vizinhas, logo, seu próximo estado depende do estados de seus dois vizinhos e de seu estado atual. Como se trata de um AC binário, cada célula só pode estar no estado 0 ou 1, dando um total de 8 combinações diferentes[7]. Ao definir o próximo estado das células de acordo com essas 8 possibilidades é possível definir a regra do AC. Na Tabela 2.1 foram definidas as regras de transição de um AC de regra 30.

Tabela 2.1: Regras de transição da regra 30.

Vizinho Esquerdo	Célula Atual	Vizinho Direito	Próximo Estado
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Ordenando as transições de acordo com os estados anteriores, podemos extrair um número binário com 8 algarismos. Realizando a conversão desse número binário para decimal, conse-

[7] $2^3 = 8$ e $2^8 = 256$

guimos descobrir a regra do AC, como pode ser visto na Figura 2.27.

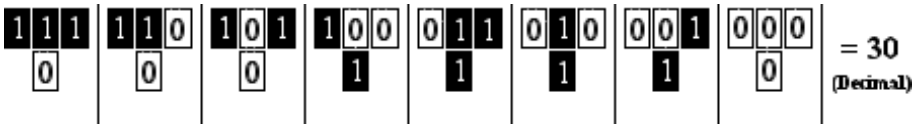

Figura 2.27: Representação gráfica da regra 30.

No trabalho de Chen et al. (2008) foi utilizado um AC binário unidimensional não uniforme para gerar números aleatórios. Um AC não uniforme é um AC em que cada célula pode ter regras diferentes, enquanto em um uniforme, todos a células tem necessariamente as mesmas regras. Esse AC foi implementado com 14 células, sendo as mais extremas com regra 90 e as internas com regra 150. Os estados das células resultam em um vetor de 14 *bits*, que foi utilizado como um valor aleatório. As células mais extremas consideravam seus vizinhos inexistentes como células de estado constante, no caso com estado zero.

De acordo com Serra et al. (1990) geradores de números aleatórios (RNGs [8]) que utilizam ACs tem demonstrado bons resultados. Sendo esses até melhores que alguns métodos tradicionais, como o deslocamento com realimentação linear (LFSR[9]). Na implementação feita por Aporntewan e Chongstitvatana (2001) também foram utilizados ACs para a implementação de RNGs.

[8]Do inglês *random number generator*
[9]Do inglês *linear feedback shift register*

Capítulo 3

PROCESSADOR MORFOLÓGICO

O objetivo deste capítulo é apresentar os conceitos utilizados para a confecção do processador morfológico, justificar a forma de implementação e apresentar os resultados obtidos a partir dos testes realizados.

Um dos pontos críticos para o desenvolvimento de um AG é a especificação da sua função de aptidão. Essa geralmente envolve a execução de um processo para solucionar o problema abordado. No caso do processamento de imagens, a execução da função de aptidão pode exigir muito tempo de processamento, comprometendo o desempenho final do AG. Com intuito de evitar esse problema, foi desenvolvido um processador morfológico binário de alto desempenho.

O objetivo principal desse processador é possibilitar que o AG o utilize como parte de sua função de aptidão. O restante da implementação da função de aptidão será abordada no Capítulo 4, que descreve o desenvolvimento do AG embarcado.

Para melhorar a abrangência do processador morfológico, também foram implementadas operações lógicas. A combinação de operações lógicas e morfológicas possibilita vários tipos de filtros de imagens, dando ao processador maior flexibilidade.

3.1 Trabalhos Correlatos

É possível encontrar diversas implementações diferentes de processadores morfológicos na literatura. Os trabalhos de Neto (2012), Pedrino (2008) , Abbott, Haralick e Zhuang (1988), Andreadis, Gasteratos e Tsalides (1996), Klein e Serra (1972) apresentam implementações em *hardware* de processadores morfológicos, alguns para imagens binárias, alguns para imagens tons de cinza e outras para imagens coloridas.

No trabalho de Neto (2012), foi desenvolvido um processador para a execução de operadores morfológicos em imagens binárias utilizando FPGAs. Para a execução das operações, a imagem é dividida em pedaços para serem processados. O tempo de processamento é proporcional ao tamanho da imagem, sendo reaproveitado o núcleo do processador para processar cada parte da imagem. Também é possível alterar o tamanho da imagem através de parâmetros do código VHDL[1] do processador. Aumentando o núcleo do processador, é possível diminuir a quantidade de ciclos necessários para processar toda a imagem.

No projeto de Andreadis, Gasteratos e Tsalides (1996) foi realizada uma implementação em ASIC, capaz de realizar dilatações em imagens com nível de cinza. Porém, a técnica utilizada inviabiliza o processamento de imagens de alta resolução, por consumir muitos recursos lógicos.

No trabalho de Kojima, Ebisawa e Miyakawa (1994) foi utilizado um algoritmo de decomposição do elemento estruturante, possibilitando a redução da complexidade das operações.

Apesar de existirem diversas implementações de processadores morfológicos, o acesso a seus códigos fonte não é fácil. Os poucos repositórios de HDL não possuem tais tipos de implementação, dificultando o reúso. Esse foi um dos motivos que influenciaram na decisão de desenvolver um processador de autoria própria.

3.2 Desenvolvimento do Processador Morfológico

Para a implementação do AG proposto nesta pesquisa, foi indispensável a utilização de um processador morfológico, pois é utilizado pelo AG para verificar a aptidão dos indivíduos de sua população. O resultado do AG também é especifico para esse processador. É importante que a transformação dos indivíduos em um código executável por esse processador seja a mais transparente possível.

Durante o levantamento bibliográfico foram encontradas algumas implementações de processadores morfológicos para imagens binárias, níveis de cinza e coloridas, essas foram abordadas na Seção 3.1. Porém, optou-se pelo desenvolvimento de um processador próprio. Construindo um processador próprio foi possível focar nos principais interesses da pesquisa, ou seja, tornar uniformes as instruções do processador, otimizar o desempenho e simplificar as operações. Assim, o acoplamento entre o processador morfológico e o AG se tornou eficiente, já que o processador foi projetado com essa finalidade.

O processador morfológico desenvolvido pode realizar as operações básicas da morfologia

[1] Do inglês *VHSIC Hardware Description Language*

matemática: dilatação, erosão, abertura e fechamento. Também pode combinar essas operações morfológicas com operações lógicas. O levantamento bibliográfico apontou que essa combinação de operações lógicas e morfológicas pode trazer diversos resultados, como apresentado na Seção 2.2.6. Esse tipo de combinação já contribuiu para a resolução de vários problemas práticos de processamento de imagens (BANON; BARRERA, 1994)

O processador morfológico possui duas unidades, uma lógica e uma morfológica, além de componentes responsáveis em acoplar essas unidades e realizar a interconexão de todos os componentes internos. As seções seguintes descrevem essas duas unidades, a lógica e a morfológica.

3.2.1 Unidade Lógica

A unidade lógica conta com a disposição total de duas imagens (*A* e *B*) e aplica uma operação lógica entre as duas de acordo com o código da instrução. Além das operações lógicas entre as imagens, também é possível realizar uma passagem direta (*by-pass*) de uma delas ou realizar a operação lógica *não*. O tamanho das duas imagens é sempre o mesmo e é definido através de parâmetros no código Verilog.

Na Tabela 3.1 é apresentado o conjunto de instruções implementado pela unidade lógica. A primeira e a segunda coluna apresentam o código da instrução, em decimal e binário, respectivamente e a terceira coluna descreve sucintamente as operações.

Tabela 3.1: Tabela de instruções da unidade lógica.

Código Decimal	Código Binário	Descrição
0	000	*Bypass* A
1	001	*Bypass* B
2	010	A negado
3	011	B negado
4	100	*E* lógico entre A e B
5	111	*Ou* lógico entre A e B
6	110	*Ou exclusivo* entre A e B
7	111	*Ou excluisivo negado* entre A e B

Na Listagem 3.1 é apresentado o código Verilog que implementa essa unidade.

Listagem 3.1: Código Verilog da unidade lógica

```
1 `ifndef __LOGIC_UNIT__
2 `define __LOGIC_UNIT__
3
4 module LogicUnit #(
```

```
5           parameter ImageWidth = 8,
6           parameter ImageHeight = 8
7   ) (
8           input [ImageHeight*ImageWidth-1:0]imgA,
9           input [ImageHeight*ImageWidth-1:0]imgB,
10          input [2:0]op,
11          output [ImageHeight*ImageWidth-1:0]result
12  );
13
14  assign result =
15          op == 3'b000 ? imgA : //a bypass
16          op == 3'b001 ? imgB : //b bypass
17          op == 3'b010 ? ~imgA : //not a
18          op == 3'b011 ? ~imgB : //not b
19          op == 3'b100 ? imgA & imgB : //and
20          op == 3'b101 ? imgA | imgB : //or
21          op == 3'b110 ? imgA ^ imgB : //xor
22          op == 3'b111 ? ~(imgA ^ imgB) :0; //not xor
23
24  endmodule
25
26  `endif
```

3.2.2 Unidade de Dilatação

A unidade de dilatação é, basicamente, uma soma de produtos replicada para cada *pixel* da imagem binária. Essa soma de produtos foi exemplificada na Figura 2.14 na Seção 2.2.1, que explica e define a operação de dilatação. Os *pixels* que ficam na borda da imagem necessitaram de um tratamento especial, já que alguns de seus vizinhos não existem. Poderiam ser utilizados os *pixels* do outro lado da imagem, como se a imagem fosse a face de uma esfera, mas optou-se por utilizar valores padrões para os *pixels* não existentes. Essa escolha foi influenciada pela ferramenta MatLab, que utiliza a mesma abordagem para a implementação da dilatação. No caso da dilatação, os *pixels* vizinhos não existentes foram substituídos por zero. Assim, ao aplicar a dilatação em uma imagem sem conteúdo, ela continua sem conteúdo, caso contrário, as bordas da imagem seriam dilatadas formando um quadro na imagem.

O código Verilog da unidade de dilatação é apresentado na Listagem 3.2. Esse código reposiciona dois vetores, um sendo a imagem e outro o elemento estruturante, os transformando em duas matrizes. Depois, replica a soma de produtos supracitada para cada item (*pixel*) da

matriz que representa a imagem.

Listagem 3.2: Código Verilog da unidade de dilatação

```
1  `ifndef __DILATE_UNIT__
2  `define __DILATE_UNIT__
3
4  module Dilate #(
5          parameter Width = 32,
6          parameter Height = 32
7  ) (
8          input [Width*Height-1:0]imageIn,
9          input [8:0]mask,
10         output [Width*Height-1:0]imageOut
11 );
12
13 wire [Width-1:0]point[Height-1:0];
14 wire [Width-1:0]dil[Height-1:0];
15
16 generate
17 genvar l,c;
18
19 for (l=0;l < Height ; l = l+1)
20 begin: _assign
21   assign point[l] = imageIn[ (l * Width) + (Width - 1) :l * Width];
22   assign imageOut[ (l * Width) + (Width - 1) :l * Width] = dil[l];
23 end
24
25 for (l =0; l< Height ; l = l+1)
26 begin: _dil
27   for(c =0;c<Width;c = c+1)
28   begin: _dil_
29     assign dil[l][c] = |{
30       ((l > 0 && c > 0)?(point[l-1][c-1] & mask[8]):1'b0),
31       ((l>0)?(point[l-1][c] & mask[7]):1'b0),
32       ((l > 0 && c < Width-1)?(point[l-1][c+1] & mask[6]):1'b0),
33       ((c > 0)?(point[l][c-1] & mask[5]) : 1'b0 ),
34       point[l][c]   & mask[4],
35       ((c < Width-1)?(point[l][c+1] & mask[3]):1'b0),
36       ((l < Height-1 && c > 0)?(point[l+1][c-1] & mask[2]):1'b0),
37       ((l < Height-1)?(point[l+1][c] & mask[1]):1'b0),
38       ((l < Height-1 && c < Width-1)?(point[l+1][c+1] & mask[0]):1'b0)
39     };
40   end
```

```
41 end
42
43 endgenerate
44
45 endmodule
46
47 `endif
```

3.2.3 Unidade de Erosão

A unidade de erosão é, basicamente, um produto de somas replicado para cada *pixel* da imagem binária, semelhante à dilatação. O produto de somas da erosão já foi exemplificado na Figura 2.17 na Seção 2.2.2, que explica e define a erosão. Os *pixels* da borda da imagem, na erosão, também precisam de um tratamento especial e, assim como na dilatação, optou-se por inserir um valor padrão para os vizinhos inexistentes que, no caso da erosão, foi escolhido o valor um. Assim, ao aplicar uma erosão em uma imagem totalmente preenchida com uns, essa imagem continuaria da mesma forma, mas caso os vizinhos inexistentes tivessem o valor zero, essa imagem teria as bordas recuadas. O código Verilog da unidade de erosão é apresentado na Listagem 3.3.

Listagem 3.3: Código Verilog da unidade de erosão

```
1  `ifndef __ERODE__
2  `define __ERODE__
3
4  module Erode #(
5          parameter Width = 32,
6          parameter Height = 32
7  ) (
8          input [Width*Height-1:0]imageIn,
9          input [8:0]mask,
10         output [Width*Height-1:0]imageOut
11 );
12
13 wire [Width-1:0]point[Height-1:0];
14 wire [Width-1:0]ero[Height-1:0];
15
16 genvar l,c;
17 generate
18
19 for (l=0;l < Height ; l = l+1)
```

```
20 begin: _assign
21   assign point[l] = imageIn[ (l * Width) + (Width - 1) :l * Width];
22   assign imageOut[ (l * Width) + (Width - 1) :l * Width] = ero[l];
23 end
24
25 for (l =0; l< Height ; l = l+1)
26 begin: _ero
27   for(c =0;c<Width;c = c+1)
28   begin: _ero_
29     assign ero[l][c] = &{
30       ((l>0 && c>0 )?(point[l-1][c-1] | ~mask[0] ):1'b1),
31       (( l >0 )?(point[l-1][c] | ~mask[1]):1'b1),
32       (( l> 0 && c < Width-1)?(point[l-1][c+1] | ~mask[2]):1'b1),
33       ((c >0)?(point[l][c-1] | ~mask[3]):1'b1),
34       (point[l][c] | ~mask[4]),
35       ((c<Width-1)?(point[l][c+1] | ~mask[5]):1'b1),
36       ((l < Height-1 && c > 0)?(point[l+1][c-1] | ~mask[6]):1'b1),
37       ((l<Height-1)?(point[l+1][c] | ~mask[7]):1'b1),
38       ((l<Height-1 && c < Width-1)?(point[l+1][c+1] | ~mask[8]):1'b1)
39     };
40   end
41 end
42
43 endgenerate
44
45 endmodule
46
47 `endif
```

3.2.4 Unidade Morfológica

A unidade morfológica combina a unidade de dilatação com a de erosão, adiciona um circuito para aproveitar a dualidade de ambas as operações e, um circuito para habilitar ou desabilitar as duas unidades individualmente.

Em cada unidade, foi adicionado dois inversores condicionais, um na entrada da imagem e, um na saída do resultado. Também foi adicionado um multiplex para selecionar o elemento estruturante original ou sua reflexão. Assim, é possível realizar uma dilatação com a unidade de erosão ou uma dilatação com a unidade de erosão, aproveitando a dualidade entre ambas as operações. Também foi adicionado um multiplex para selecionar a imagem de entrada ou a

resultante da unidade, possibilitando a realização de um *by-pass*. Na Figura 3.1 é apresentado um diagrama de blocos, evidenciando as conexões realizadas.

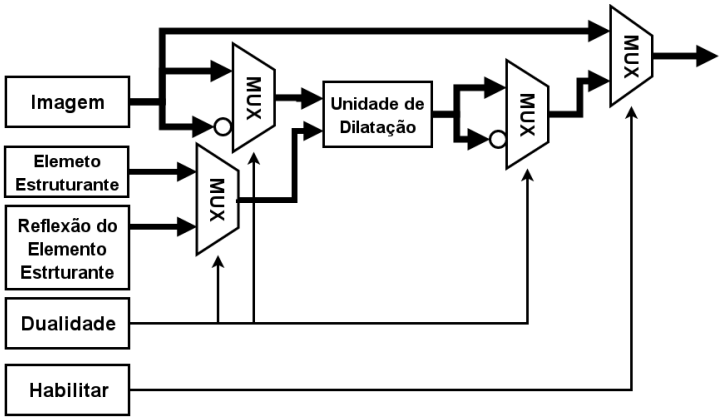

Figura 3.1: Unidade de dilatação com circuito de dualidade e de *bypass*.

Os mesmos circuitos foram replicados para a unidade de erosão, como exemplificado na Figura 3.2

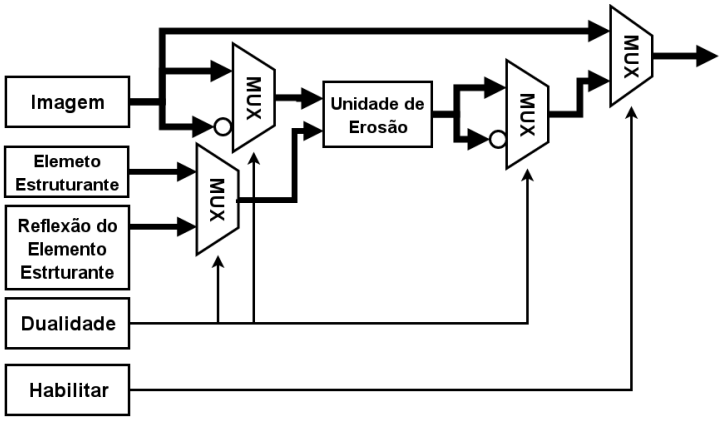

Figura 3.2: Unidade de erosão com circuito de dualidade e de *bypass*.

A imagem resultante da unidade de dilatação é utilizada como entrada na unidade de erosão, e ambas as unidades usam o mesmo elemento estruturante. Com essa organização, é possível realizar operações um pouco mais complexas, por exemplo, a abertura e o fechamento.

Com a combinação das unidades, se tornaram necessários alguns *bits* de controle, dois para habilitar ou desabilitar as unidades de dilatação e erosão e mais dois para ativar a dualidade, tendo assim, um total de dezesseis possibilidades de combinação. Porém, muitas das combinações acabam gerando o mesmo resultado. Por isso, foi adicionado um decodificador, reduzindo os *bits* de controle para três. As operações resultantes são apresentadas na Tabela 3.2.

A adição do decodificador é importante para a redução das instruções da unidade morfológica e consequentemente do processador final. Como o processador será posteriormente

Tabela 3.2: Tabela de instruções da unidade morfológica.

Código Decimal	Código Binário	Descrição
0	000	*by-pass*
1	001	Dilatação
2	010	Erosão
3	011	Fechamento
4	100	Abertura
5	111	2 x Dilatação
6	110	2 x Erosão
7	111	*by-pass*

utilizado juntamente com o AG, a redução de suas instruções possibilitou a redução do tamanho do código genético dos indivíduos do AG, o que pode contribuir para o seu desempenho. Na Figura 3.3 é apresentado um diagrama de blocos da organização da unidade morfológica. Em seguida na Listagem 3.4 é apresentado o código Verilog da unidade morfológica.

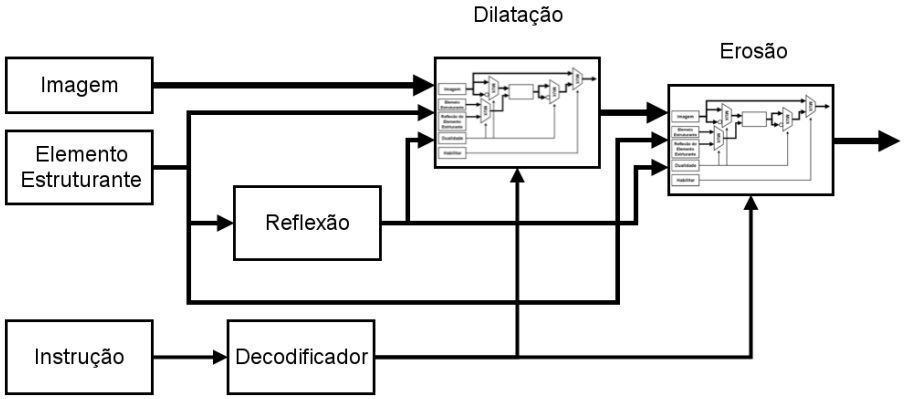

Figura 3.3: Unidade morfológica.

Listagem 3.4: Código Verilog da unidade morfológica

```
1  `ifndef __MORPHOLOGIC_UNIT__
2  `define __MORPHOLOGIC_UNIT__
3
4  `include "morphology/Dilate.v"
5  `include "morphology/Erode.v"
6
7  module MorphologicUnit #(
8          parameter ImageWidth=32,
9          parameter ImageHeight=32
10 ) (
11         input [ImageWidth*ImageHeight-1:0]img,
12         input [8:0]el,
13         input [2:0]op,
14         output [ImageWidth*ImageHeight-1:0]result
```

```
15 );
16
17 assign {ceDilate,swapDilate,ceErode,swapErode} =
18         op == 3'b000 ? 4'b0000: //bypass
19         op == 3'b001 ? 4'b1000: //dil
20         op == 3'b010 ? 4'b0010: //ero
21         op == 3'b011 ? 4'b1010: //dil ero
22         op == 3'b100 ? 4'b1111: //ero dil
23         op == 3'b101 ? 4'b1011: //dil dil
24         op == 3'b110 ? 4'b1110: //ero ero
25         op == 3'b111 ? 4'b0000:4'b0000; //bypass
26
27 wire [ImageWidth*ImageHeight-1:0]image = img;
28 wire [8:0]element = el;
29 wire [8:0]relement;
30
31 generate
32         genvar l,c;
33         for(l=0;l<3;l=l+1)
34         begin:_el_line_
35                 for(c=0;c<3;c=c+1)
36                 begin: _el_column_
37                         assign relement[(3-l-1)*3+(3-c-1)] = el[l*3+c];
38                 end
39         end
40 endgenerate
41
42 wire ceDilate;
43 wire swapDilate;
44 wire [ImageWidth*ImageHeight-1:0]dilateIn = image;
45 wire [ImageWidth*ImageHeight-1:0]dilated;
46 wire [ImageWidth*ImageHeight-1:0]dilateResult = ceDilate ? dilated ^ {
       ImageWidth*ImageHeight{swapDilate}} : dilateIn;
47 Dilate #(
48         .Width(ImageWidth),
49         .Height(ImageHeight)
50 )
51 dilate(
52         .imageIn(dilateIn ^ {ImageHeight*ImageWidth{swapDilate}}),
53         .mask(swapDilate?relement:element),
54         .imageOut(dilated)
55 );
56
```

```
57 wire ceErode;
58 wire swapErode;
59 wire [ImageWidth*ImageHeight-1:0]erodeIn = dilateResult;
60 wire [ImageWidth*ImageHeight-1:0]eroded;
61 wire [ImageWidth*ImageHeight-1:0]erodeResult = ceErode ? eroded^{ImageWidth
       *ImageHeight{swapErode}} : erodeIn;
62 Erode #(
63         .Width(ImageWidth),
64         .Height(ImageHeight)
65 )
66 erode(
67         .imageIn(erodeIn ^ {ImageWidth*ImageHeight{swapErode}}),
68         .mask(swapErode?relement:element),
69         .imageOut(eroded)
70 );
71
72 assign result = erodeResult;
73
74 endmodule
75
76 `endif
```

3.2.5 Organização do Processador Morfológico

Para acoplar a unidade lógica e a morfológica de maneira eficaz, foi adicionado um registrador para armazenar as imagens obtidas a partir das operações lógicas e morfológicas. O registrador pode ser usado como entrada para outras operações. Também foi adicionado um contador de operações com tamanho parametrizável pelo código Verilog. Quando esse contador está com o valor zero, a imagem de entrada da unidade morfológica é a entrada de imagem convencional do processador, nos outros casos, pode ser selecionada a imagem de entrada ou a imagem armazenada no registrador.

Para cada *clock* o contador de operações é incrementado. Dessa forma, um circuito externo pode alterar os parâmetros de execução. Os parâmetros de execução são: a operação morfológica, o elemento estruturante, a operação lógica e um *bit* de seleção para a entrada da unidade morfológica. Na Figura 3.4 é ilustrado um diagrama de blocos do circuito implementado.

O código Verilog que implementa o processador morfológico é apresentado na Listagem 3.5

Listagem 3.5: Código Verilog do processador morfológico

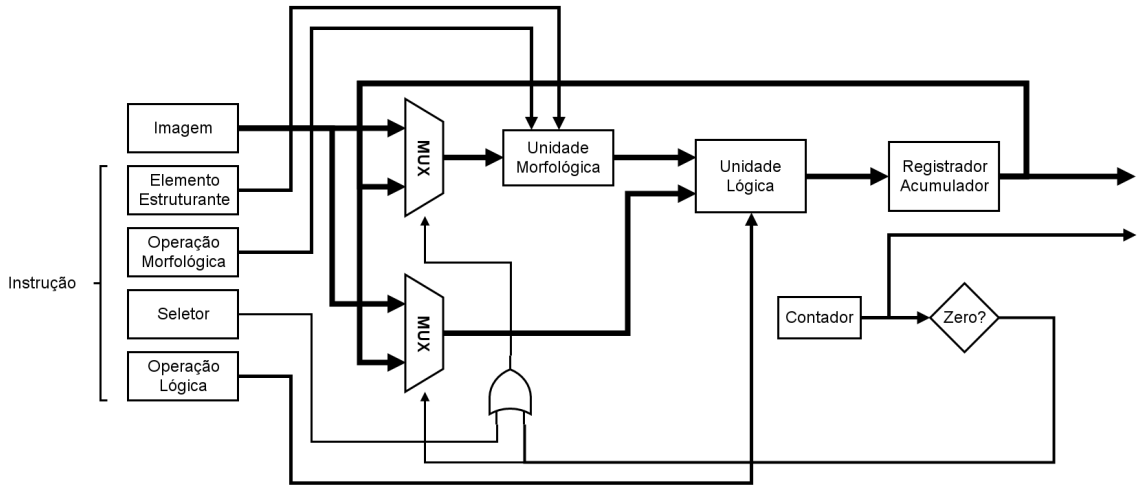

Figura 3.4: Diagrama do Processador Morfológico.

```
1  `ifndef __MORPHOLOGIC_PROCESSOR__
2  `define __MORPHOLOGIC_PROCESSOR__
3
4  `include "image/LogicUnit.v"
5  `include "morphology/MorphologicUnit.v"
6
7  module MorphologicProcessor #(
8    parameter ImageWidth = 32,
9    parameter ImageHeight =32,
10   parameter OpCounterWidth = 2
11 ) (
12   input clk,
13   input rst,
14   input ce,
15   input [ImageWidth*ImageHeight-1:0]image,
16   input [8:0]el,
17   input [2:0]morphOp,
18   input morphInSelect,
19   input [2:0]logicOp,
20   output reg [OpCounterWidth-1:0]opCounter,
21   output reg [ImageHeight*ImageWidth-1:0]imageAcc
22 );
23
24 wire [ImageHeight*ImageWidth-1:0]morpholicResult;
25 MorphologicUnit #(
26   .ImageWidth(ImageWidth),
27   .ImageHeight(ImageHeight)
28 )
29 morphologicUnit (
```

```
30    .img((morphInSelect | opCounter ==0)?image:imageAcc),
31    .el(el),
32    .op(morphOp),
33    .result(morpholicResult)
34  );
35
36  wire [ImageHeight*ImageWidth-1:0]logicResult;
37  LogicUnit #(
38    .ImageWidth(ImageWidth),
39    .ImageHeight(ImageHeight)
40  )
41  logicUnit (
42    .imgA(opCounter == 0 ? image : imageAcc),
43    .imgB(morpholicResult),
44    .op(logicOp),
45    .result(logicResult)
46  );
47
48  always @(posedge clk or posedge rst)
49  begin
50    if (rst)
51    begin
52      opCounter = {OpCounterWidth{1'b0}};
53      imageAcc = {ImageHeight*ImageWidth{1'b0}};
54    end
55    else
56    if (ce)
57    begin
58      opCounter = opCounter + 1'b1;
59      imageAcc = logicResult;
60    end
61  end
62
63  endmodule
64
65  `endif
```

3.3 Resultados

O processador foi sintetizado para três modelos de FPGAs: Cyclone II EP2C5T144C6, Cyclone IV E EP4CE115F29C7 e Stratix III EP3SL150F1152C2. A síntese e o cálculo do

consumo dos recursos, foram realizados com o Quartus II, IDE fornecida pela fabricante dos FPGAs utilizados, a Altera. O consumo de recursos em cada FPGA pode ser verificado na Tabela 3.3. Pode ser observado que FPGAs com menor densidade não possuem recursos suficientes para sintetizar esse processador para imagens "grandes", como no caso da Cyclone II. FPGAs mais atuais e/ou com mais recursos já tem capacidade para processar imagens com tamanhos mais significativos.

Tabela 3.3: Consumo Lógico do Processador Morfológico.

Tamanho da Imagem pixel x pixel	Cyclone II EP2C5T144C6		Cyclone IV E EP4CE115F29C7		Stratix III EP3SL150F1152C2	
	Elementos Lógicos	Registradores	Elementos Lógicos	Registradores	ALUTs	Registradores
8x8	1.478 (32%)	66 (1%)	1.449 (1%)	66 (<1%)	959 (<1%)	66 (<1%)
16x16	5.871 (127%)	258 (6%)	5.949 (5%)	258 (<1%)	3.871 (3%)	258 (<1%)
32x32			20.585 (18%)	1026 (<1%)	20.159 (18%)	1.026 (<1%)

Imagens com tamanhos usuais precisam de uma abordagem diferente da utilizada nesse processador. Esse processador aborda a imagem de forma totalmente paralela, exigindo muitos recursos lógicos. Para processar imagens grandes, é necessário utilizar uma organização com *pipelines*.

3.4 Considerações Finais

O processador desenvolvido possui um desempenho elevado, uma vez que realiza as operações morfológicas e lógicas na imagem completa em apenas um ciclo de máquina. Porém, isso exige a utilização de um barramento extremamente largo, inviabilizando sua utilização em aplicações convencionais. Considerando que seu propósito é apenas para o treinamento do AG, onde foi utilizado um barramento largo e imagens pequenas de até 32x32, esse processador alcançou seu objetivo.

Capítulo 4

ALGORITMO GENÉTICO EMBARCADO

A implementação de um AG em *hardware* levanta alguns questionamentos. Deve-se levar em consideração a complexidade do algoritmo e das metáforas biológicas utilizadas, para ser possível implementar um *hardware* mais otimizado. Partes do algoritmo podem ser simplificadas para reduzir o consumo de recursos lógicos, porém, isso pode prejudicar o desempenho do algoritmo, caso suas características essenciais não forem levadas em consideração. No decorrer deste capítulo será apresentado como foi realizado o desenvolvimento do *hardware* genético, explicando as abordagens utilizadas para resolver os problemas que apareceram durante a implementação e os questionamentos supracitados.

4.1 Trabalhos Correlatos

Pedrino (2008) desenvolveu uma arquitetura *pipeline* para processamento morfológico de imagens binárias, níveis de cinza e, inclusive, imagens coloridas, em tempo real. Para realizar operações morfológicas em imagens coloridas o autor utilizou a teoria dos reticulados. A arquitetura é configurada por meio de instruções geradas por um AG executado na ferramenta MatLab. Dessa forma, foi possível gerar diversos filtros para imagens com a combinação de operações lógicas e morfológicas.

Já no projeto de Chen et al. (2008), foi desenvolvida uma arquitetura dedicada para a execução de um AG. Para que a arquitetura fosse facilmente alterada, foi desenvolvido um *software* que gera código Verilog a partir de configurações fornecidas por um usuário. O usuário pode configurar o tamanho da população, a taxa de mutação, o formato da operação de cruzamento, a forma de avaliação e o tamanho do indivíduo da população. Os números aleatórios necessários para a implementação do AG, foram fornecidos por um AC binário unidimensional com as regras 90 e 150 combinadas. Com essa implementação, foi possível realizar buscas por máximos

e mínimos globais dentro de determinados intervalos na ordem de milissegundos.

Foi proposto por Yuen e Chow (2009) um AG que nunca revisita pontos já explorados no espaço de busca. Os indivíduos já testados são armazenados em um arquivo, para posteriores consultas. O AG apresentou bom desempenho, principalmente em casos que a função de aptidão tem grande custo computacional, tornando quase que insignificativo o custo da consulta por indivíduos já visitados.

Já Povinelli e Feng (1999) propuseram a utilização de uma tabela *hash* para armazenar os resultados da função de *fitness*. AGs com funções de *fitness* com custo computacional significativo tiveram seu desempenho melhorado.

Hrbacek e Sikulova (2013) implementaram um algoritmo co-evolucionário de programação genética cartesiana, utilizando FPGAs e *soft-cores*. O algoritmo utiliza duas populações que são sempre avaliadas em paralelo por dois processadores e os resultados armazenados em uma memória compartilhada. O módulo de programação genética cartesiana consiste em um circuito virtual reconfigurável, onde é utilizado um indivíduo da população como configuração. O circuito utiliza as operações geradas pelos indivíduos, para realizar filtros em imagens digitais e conseguiu ganhar um desempenho significativo comparado com implementações em *software*.

Coury et al. (2011) implementaram um AG em um FPGA para a estimação da frequência em circuitos elétricos. Esse trabalho é um ótimo exemplo da necessidade de aprendizado em tempo real, onde o AG era executado a cada amostragem de sinal.

Alguns outros projetos também utilizam a mesma ideia do projeto de Pedrino (2008), onde são utilizados AGs para gerar uma sequência de operações morfológicas, como nos trabalhos de Bala e Wechsler (1993), de Yoda, Yamamoto e Yamada (1999) e de Harvey e Marshall (1996). Todos esses projetos usaram implementações tradicionais de AGs, alguns com pequenas peculiaridades da plataforma em que se foi desenvolvido o AG. Com esses trabalhos foi possível perceber o potencial de otimização dos AGs e a possibilidade de melhorar seu desempenho com a utilização de sistemas embarcados.

4.2 Desenvolvimento do *Hardware* Dedicado

O desenvolvimento de um *hardware* dedicado para execução de um AG levanta vários questionamentos. Geralmente, os AGs utilizam grande quantidades de indivíduos em sua população. Isso ajuda a preservar a diversidade genética e abranger uma maior área do espaço de busca. Porém, uma quantidade demasiadamente grande de indivíduos pode prejudicar o desempenho

do algoritmo, pois esses indivíduos devem ser de alguma forma comparados para posteriormente serem selecionados os indivíduos mais aptos. Deve ser lembrado que a cada geração, todos os indivíduos da população são testados, ou seja, a complexidade de uma geração do AG é diretamente proporcional ao tamanho da população. Outro ponto crítico em relação a quantidade de indivíduos na população, que deve ser levado em consideração, é a memória necessária para armazenar esses indivíduos. Todos esses pontos são de extrema importância para o desenvolvimento do *hardware* dedicado.

A função de aptidão, também chamada de função de *fitness*, que mensura a aptidão de cada indivíduo, é um ponto crítico para o desempenho, pois será utilizada para cada indivíduo de cada geração. Caso a avaliação de um único indivíduo seja um processo demorado, todo o desempenho do AG será prejudicado. A função de aptidão define o objetivo do AG, sendo ela uma função que mensura a aptidão do indivíduo aplicado a resolução do problema real ou a uma aproximação do problema. Dessa forma, todo o resto do AG pode ser reutilizado, enquanto essa parte varia para cada tipo de problema. Devido a essa característica, manter o módulo de aptidão desacoplado do restante do sistema pode tornar o sistema muito mais flexível, facilitando seu reúso.

A maioria dos métodos de seleção dos AGs envolve a ordenação dos indivíduos da população de acordo com suas aptidões, possibilitando a seleção dos indivíduos de acordo com essas aptidões. Esses métodos utilizam uma abordagem estatística, selecionando indivíduos de forma aleatória, porém, proporcional ás aptidões. A ordenação de grandes quantidades de indivíduos é um processo custoso para ser implementado em *hardware* ou *software*. Utilizar um método que evite essa abordagem pode elevar o desempenho do AG. O método de seleção por torneio ou o utilizado em AGs Celulares podem ser uma boa opção, já que não precisam ordenar toda a população, apenas pequenos grupos de indivíduos.

A reprodução dos indivíduos utilizando mutação e cruzamento, podem variar conforme a escolha de representação dos indivíduos. As formas de representação mais básicas são a binária e a real. Para a representação binária, os métodos de mutação e cruzamento envolvem a inversão e trocas de *bits*. A representação real envolve operações aritméticas, como médias, somas, etc. A representação binária é, sem dúvida, a mais simples e mais intuitiva para uma implementação em *hardware*.

Os métodos de seleção, cruzamento e mutação, envolvem a utilização de números aleatórios. Para fornecer esses números optou-se pela utilização de ACs, como utilizado em um dos trabalhos correlatos, de Chen et al. (2008).

4.2.1 Representação Cromossômica

Os indivíduos da população do AG proposto, devem conseguir representar operações lógicas e morfológicas. No caso, essas operações foram implementadas pelo processador morfológico apresentado no Capítulo 3.5, então os indivíduos devem seguir a codificação das instruções desse processador. Para isso o cromossomo deve obedecer a Equação 4.1. Essa define que o cromossomo é representado por um vetor de *bits* com tamanho n.

$$C = [c_{n-1}, c_{n-2}, ..., c_0] \;\Big|\; c_i \in \{0,1\} \tag{4.1}$$

O tamanho do vetor tem a restrição de ser múltiplo de uma potência de 2 multiplicado por 16, como define a Equação 4.2, onde n é o tamanho do vetor. Isso é necessário pois o processador morfológico executa somente uma instrução de 16 *bits* por vez e contem um contador de instruções para auxiliar algum circuito externo a multiplexar os segmentos de 16 *bits*.

$$n = 2^i 16 \;\Big|\; i \in \mathbb{N} \tag{4.2}$$

Na Figura 4.1 é exemplificada a decodificação do primeiro segmento de 16 *bits* de um cromossomo. O *bit* de índice 3, o quarto da direita para esquerda, foi marcado com um X, pois o processador morfológico o ignora na execução da primeira instrução. Esse *bit* determina qual será a entrada da unidade morfológica: 0 para a imagem de entrada do processador e 1 para a imagem do registrador. Como na primeira instrução nenhuma imagem foi inserida no registrador, o próprio processador morfológico ignora esse *bit*.

A decodificação exemplificada na Figura 4.1 pode ser descrita como na Equação 4.3, sendo I^{in} a imagem de entrada, I^{out} a imagem resultante e E o elemento estruturante.

$$I^{out} = (I^{in} \ominus E) \wedge I^{in} \;\Big|\; E = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{4.3}$$

O protótipo final utilizou $i = 1$, assim o código genético é codificado por meio de um vetor de 32 *bits*. Assim, é possível realizar operações lógicas com resultados de operações morfológicas, ou acumular operações morfológicas. Na Figura 4.2 é exemplificado a decodificação

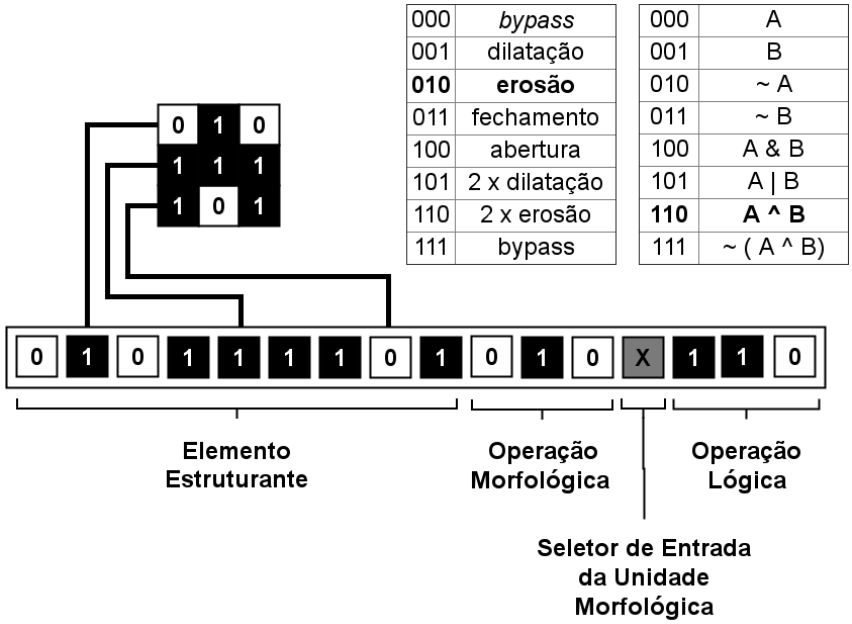

Figura 4.1: Exemplo de Decodificação de um Segmento do Cromossomo.

completa de um cromossomo de 32*bits*. Pode ser observado nessa figura, que o *bit* de índice 3 do segundo segmento de 16 *bits*, não foi demarcado com X. Isso é para indicar que dessa vez houve sua utilização de forma efetiva. No caso, fez com que a segunda instrução fosse aplicada ao resultado da primeira, pois o resultado da primeira instrução foi armazenado no registrador do processador, que posteriormente foi utilizado como entrada para unidade morfológica. Caso esse *bit* fosse 0, a segunda instrução seria executada novamente sobre a imagem de entrada, possibilitando realizar uma operação lógica entre dois resultados morfológicos.

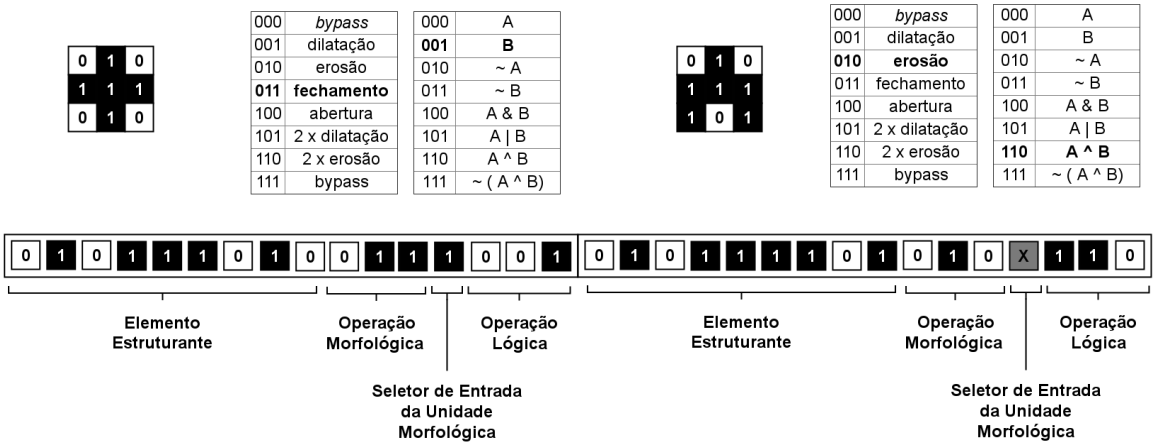

Figura 4.2: Exemplo de Decodificação de um Cromossomo de 32 *bits*.

A decodificação exemplificada na Figura 4.2 pode ser descrita como na Equação 4.4, sendo I^{in} a imagem de entrada, I^{out} a imagem resultante, E_o o elemento estruturante da primeira instrução e E_1 o elemento estruturante da segunda instrução.

$$I^{out} = ((I^{in} \ominus E_0) \wedge I^{in}) \bullet E_1 \left| E_0 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}, E_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \right. \tag{4.4}$$

4.2.2 Função Objetivo

A ideia da função objetivo para o AG implementado neste projeto, é mensurar a aptidão de uma sequência de operações lógicas e morfológicas na transformação de uma imagem de entrada, também chamada de original, em uma imagem objetivo. Para isso, é necessário comparar a imagem resultante da sequência de operações com a imagem objetivo.

A função objetivo mensura a diferença entre, a imagem resultante da decodificação e execução do cromossomo sobre a imagem original e a imagem objetivo. Essa mensuração é realizada comparando *bit* a *bit* as imagens, como representado na Equação 4.5, onde H é a quantidade de linhas das imagens ,W a quantidade de colunas, C o indivíduo, I^{in} a imagem original, I^{obj} a imagem objetivo e I^{out} a imagem resultante.

$$f(C) = \sum_{i=0}^{H} \sum_{j=0}^{W} |I_{i,j}^{obj} - I_{i,j}^{out}| \left| \; I^{out} = decode(C, I^{in}) \right. \tag{4.5}$$

Na Figura 4.3 é exemplificado o processo de comparação das imagens. Nessa figura são representadas duas imagens 4x4 que são comparadas *bit* a *bit*. Só foram ilustrados o *bits* diagonais para simplificar a imagem. A soma das diferenças é executada por um circuito assíncrono, não sendo necessários *clocks*. Isso exige mais elementos lógicos, porém, tem desempenho elevado.

No caso de imagens 4x8, que foi o tamanho utilizado no protótipo final, o resultado da função objetivo está dentro do intervalo $[0;32]$, podendo assumir 33 valores diferentes. Sendo assim necessários 6 *bits* para representação, segundo a Equação 4.6. Como a função objetivo resulta em valores pequenos para bons indivíduos e altos para os demais, o AG implementado neste capítulo é aplicado a um problema de minimização.

$$32 = 2^5 < 33 < 2^6 = 64 \tag{4.6}$$

O código Verilog que realiza a comparação entre as imagens de forma assíncrona é apresen-

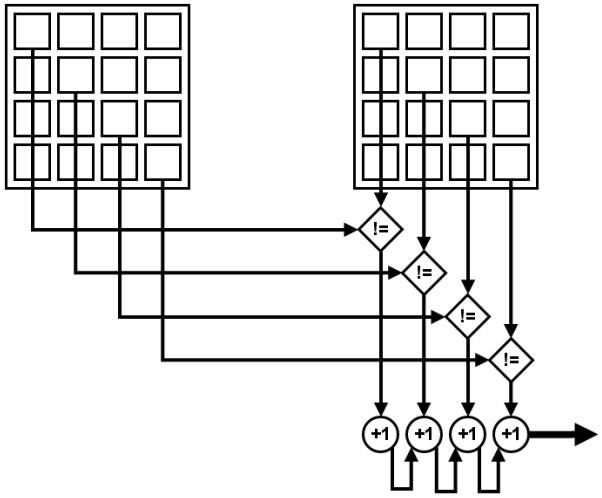

Figura 4.3: Exemplo do Processo de Comparação das Imagens

tado na Listagem 4.1. Esse código descreve um módulo que é posteriormente utilizado dentro do módulo de *fitness*, que implementa toda a função objetivo.

Listagem 4.1: Código Verilog da Comparação *bit* a *bit*

```
1  `ifndef __DIFF__
2  `define __DIFF__
3
4  module Diff #(
5          parameter Width = 32,
6          parameter DiffWidth = $clog2(Width)
7  ) (
8          input [Width-1:0]A,
9          input [Width-1:0]B,
10         output reg [DiffWidth-1:0]Diff
11 );
12
13 wire [Width-1:0]diffBits = A ^ B;
14
15 integer i;
16 always @(*)
17 begin
18         Diff = 0;
19         for (i=0;i<Width;i=i+1)
20         begin
21                 Diff = Diff + diffBits[i];
22         end
23 end
24
25 endmodule
```

```
26
27 `endif
```

O módulo de *fitness* é composto por um processador morfológico, apresentado no Capítulo 3, um comparador e um multiplexador. O processador morfológico é utilizado para decodificar e executar os segmentos, selecionados pelo multiplexador, do cromossomo sobre a imagem original. A imagem resultante é então comparada, utilizando o comparador, com a imagem objetivo. A imagem original e a objetivo são entradas desse módulo. A organização interna do módulo de *fitness* é exemplificada na Figura 4.4.

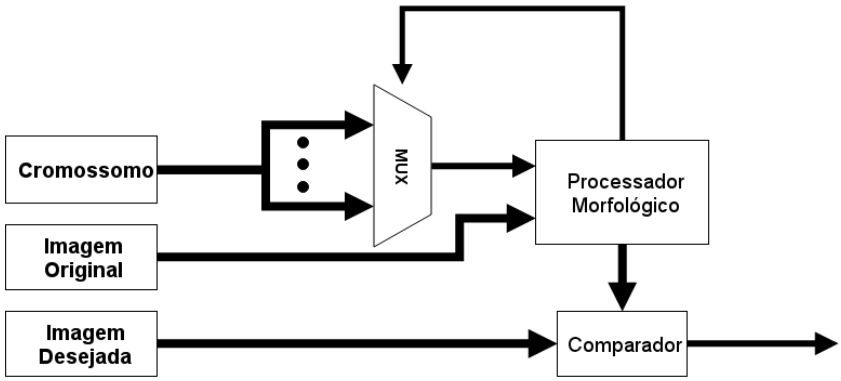

Figura 4.4: Módulo de Avaliação

O módulo de *fitness* obedece uma interface simples de comunicação. Essa interface foi definida para facilitar a troca do módulo de *fitness*, tornando o resto da implementação do AG mais simples de reutilizar em outros problemas.

A interface define 3 *bits* de controle e 2 barramentos de dados. Um dos barramentos é para o módulo de *fitness* receber o indivíduo que será testado e o outro para devolver o resultado da função objetivo. Um dos *bits* de controle serve para iniciar o processo de avaliação, um outro para indicar que o módulo está ocupado e por o último, um para indicar que o processo foi concluído. O código Verilog que implementa a função objetivo e essa interface é apresentado na Listagem 4.2.

Listagem 4.2: Código Verilog da Função Objetivo ou Módulo de *fitness*

```
1 `ifndef __MORPHOLOGIC_FITNESS__
2 `define __MORPHOLOGIC_FITNESS__
3
4 `include "image/MorphologicProcessor.v"
5 `include "util/Diff.v"
6 `include "util/Multiplex.v"
7
8 module MorphologicFitness #(
```

```
9     parameter ImageWidth = 8,
10    parameter ImageHeight = 8,
11    parameter ErrorWidth = $clog2(ImageWidth*ImageHeight),
12    parameter OpcodeWidth = 16,
13    parameter OpCounterWidth = 2,
14    parameter InstructionWidth = OpcodeWidth * (2**OpCounterWidth)
15  ) (
16    input clk,
17    input rst,
18    input [ImageHeight*ImageWidth-1:0]origin,
19    input [ImageHeight*ImageWidth-1:0]objetive,
20    input [InstructionWidth-1:0]individual,
21    input start,
22    output [ErrorWidth-1:0]error,
23    output reg finish,
24    output reg buzy
25  );
26
27    MorphologicProcessor #(
28      .ImageWidth(ImageWidth),
29      .ImageHeight(ImageHeight),
30      .OpCounterWidth(OpCounterWidth)
31    )
32    processor(
33      .clk(clk),
34      .rst(rst),
35      .ce(buzy),
36      .image(origin),
37      .el(opcode[15:7]),
38      .morphOp(opcode[6:4]),
39      .morphInSelect(opcode[3]),
40      .logicOp(opcode[2:0]),
41      .opCounter(opCounter),
42      .imageAcc(product)
43    );
44
45    Diff #(
46      .Width(ImageWidth*ImageHeight),
47      .DiffWidth(ErrorWidth)
48    )
49    diff(
50      .A(product),
51      .B(objetive),
```

```
52      .Diff(error)
53    );
54
55    Multiplex #(
56      .Width(OpcodeWidth),
57      .AddressSize(OpCounterWidth)
58    )
59    multiplex(
60      .D(individual),
61      .S(opCounter),
62      .Q(opcode)
63    );
64
65    wire [ImageWidth*ImageHeight-1:0]product;
66    wire [OpCounterWidth-1:0]opCounter;
67    wire [OpcodeWidth-1:0]opcode;
68
69    always @(negedge clk or posedge rst)
70    begin
71      if (rst)
72      begin
73        buzy = 1'b0;
74        finish = 1'b0;
75      end
76      else if (~buzy && ~finish && start)
77      begin
78        buzy = 1'b1;
79        finish = 1'b0;
80      end
81      else if(~buzy && finish)
82      begin
83        finish = 1'b0;
84      end
85      else if(buzy && opCounter == 0)
86      begin
87        buzy = 1'b0;
88        finish = 1'b1;
89      end
90
91    end
92
93  endmodule
94
```

```
95 `endif
```

4.2.3 População

O armazenamento da população é um ponto crítico para o desenvolvimento. Utilizar uma população grande poderia requerer a utilização de uma memória externa, criando um gargalo. Porém, é um desafio manter a diversidade genética com uma população pequena.

Uma abordagem que economiza memória é a dos AGs Compactos, que representam a população por meio de um vetor de probabilidades. Dependendo do tamanho desse vetor, é bem possível o manter em registradores internos. Porém, a adequação do vetor de probabilidades e a geração dos indivíduos podem acabar sendo um processo custoso.

Uma opção interessante é utilizar a abordagem dos AGs Celulares, assim evitando a ordenação de grandes listas de indivíduos. Mas, uma das características desse algoritmo é a utilização de processadores paralelos para realizar a avaliação, seleção e cruzamento dos indivíduos, que no caso desse projeto, pode ser inviável já que o processo de avaliação envolve processamento de imagens.

Algumas implementações de AG eliminam indivíduos duplicados da população, o que pode melhorar o desempenho do algoritmo. A utilização de um código *hash*[1] para identificar e possibilitar a eliminação de indivíduos duplicados foi apresentada por Ronald (1998).

No incio da pesquisa, foi desenvolvido um pequeno protótipo em *hardware* de um algoritmo evolutivo (AE) simplificado, onde não se armazenava toda a população, mas somente o melhor indivíduo[2]. A população era gerada apenas com a mutação do melhor indivíduo e, caso um indivíduo melhor fosse gerado, substituía o anterior. Essa pequena implementação não é muito eficaz, levando várias vezes a convergência prematura. Porém, algumas vezes os ótimos globais de alguns problemas foram encontrados.

Uma opção para melhorar a implementação desse AE seria utilizar vários deles de forma insular. Essa abordagem poderia consumir muitos recursos lógicos, pois seria necessário replicar todas as etapas do AE: seleção, mutação e avaliação. Para diminuir o consumo de recursos, as etapas comuns que precisariam ser replicadas, poderiam ser compartilhadas através de escalonamento. Porém, assim que um AE encontra-se um super-indivíduo, esse seria compartilhado com os outros AEs, podendo fazer com que todos convergissem prematuramente. Utilizar o conceito de nichos ecológicos poderia evitar a propagação de super-indivíduos. Cada AE se-

[1] Ou resumo em Português

[2] Veja mais detalhes dessa implementação no Apêndice A

ria responsável por um nicho, assim, seria evitado que o super-indivíduo de um AE dominasse a população dos outros. Dessa forma, fazendo-se necessário um método para classificar os indivíduos, possibilitando a divisão em nichos.

Mantendo a ideia elitista do AE implementado, o conceito de nichos e a utilização de *hash* para identificar indivíduos duplicados, foi projetado um banco de memória, onde em cada endereço armazena-se somente o melhor indivíduo de um nicho. Cada nicho, ou endereço no banco de memoria, é determinado por meio de um *hash*, gerado a partir do código genético dos indivíduos. Assim, indivíduos com o mesmo código competem pelo mesmo endereço de memoria, forçando a eliminação de indivíduos duplicados. Normalmente as funções de *hash* são projetadas para gerar a menor quantidade de colisões[3] possíveis, porém, nessa implementação seria interessante gerar algumas colisões, forçando a competição pelos endereços de memoria. Pode-se comparar cada endereço de memória com um nicho: somente os indivíduos que tenham um código genético que derive aquele endereço de memória competirão por ele. A função de *hash* utilizada foi baseada em paridade. Os códigos genéticos dos indivíduos são separados em n segmentos, então é calculado o *bit* de paridade de cada segmento. Na Figura 4.5 é exemplificado esse processo com $n = 4$ para um cromossomo de 12 *bits*.

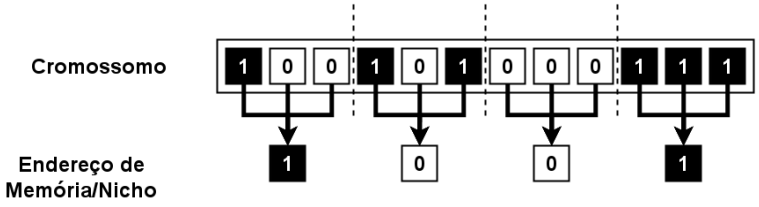

Figura 4.5: Cálculo da paridade dos segmentos de um cromossomo.

O valor de n determina a quantidade de endereços do banco de memória. A razão entre o tamanho dos indivíduos e a quantidade de endereços de memória influencia a diversidade genética e a pressão seletiva. Essa razão determina a quantidade de possíveis indivíduos que competirão por cada endereço, como mostra a Equação 4.7, onde N são os possíveis indivíduos por nicho, n a quantidade de segmentos, w a quantidade de *bits* dos cromossomos, M a quantidade de endereços de memória e P o tamanho do espaço de busca.

$$N = \frac{2^w}{2^n} = 2^{w-n} \Big| M = 2^n, P = 2^w \tag{4.7}$$

Junto a cada endereço de memória, também é armazenada a aptidão do indivíduo, no caso,

[3]Uma colisão dentro do conceito de funções de *hash* é quando a função aplicada a dois valores diferentes têm o mesmo resultado.

o resultado da função objetivo. Esse valor é posteriormente utilizado para decidir se o indivíduo continuará na população, quando um outro indivíduo do mesmo nicho for gerado. O AG implementado busca a minimização da função objetivo, assim, o indivíduo será substituído somente caso um outro indivíduo que concorra pelo mesmo endereço tenha o valor da função objetivo menor. No momento de armazenar um indivíduo na memória, é verificado se o mesmo é mais apto do que o já presente na memória em seu respectivo endereço, do contrário, o mesmo não é armazenado e é descartado. Essa verificação faz o papel da pressão seletiva, posteriormente simplificando o procedimento de seleção.

Na Figura 4.6 é ilustrada a organização interna do módulo da população, onde uma memória de acesso aleatório (RAM[4]), foi utilizada para armazenar o indivíduo e sua aptidão, os indivíduos são endereçados por seu código *hash* e a escrita na memória só é habilitada caso o novo indivíduo seja mais apto que o já presente na memória.

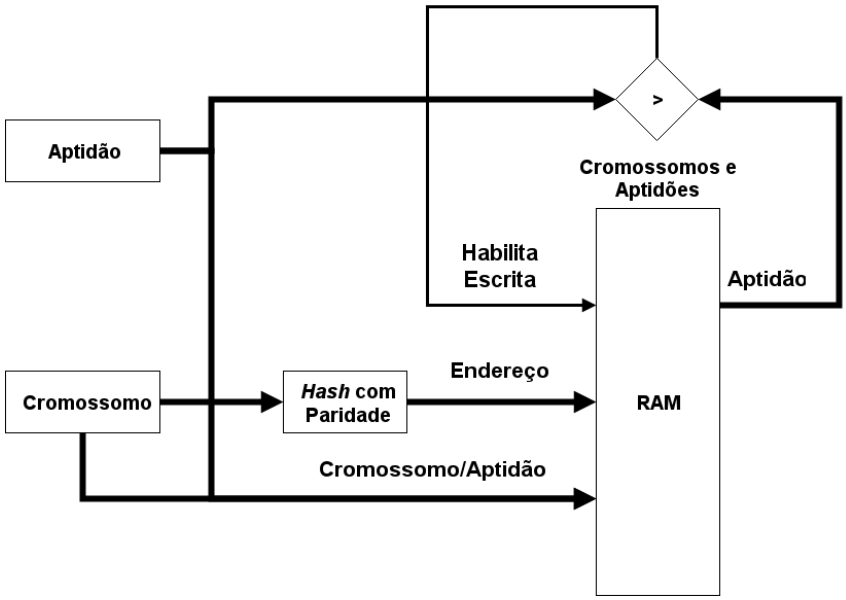

Figura 4.6: Memória da População.

O *hardware* responsável pelo armazenamento da população é descrito no código Verilog apresentado na Listagem 4.3.

Listagem 4.3: Código Verilog da População do AG

```
1 `ifndef __INDIVIDUAL_CACHE__
2 `define __INDIVIDUAL_CACHE__
3
4 `include "memory/RAM3.v"
5
6 module IndividualsCache #(
```

[4]Do inglês *Random Access Memory*

```
7           parameter IndividualWidth = 32,
8           parameter AddressWidth = 4,
9           parameter ErrorWidth = 32
10  ) (
11          input clk,
12          input rst,
13          input we,
14          input [IndividualWidth-1:0]inIndividual,
15          input [ErrorWidth-1:0]inError,
16          input [AddressWidth-1:0]addrIndividual1,
17          output [IndividualWidth-1:0]outIndividual1,
18          output [ErrorWidth-1:0]outError1,
19          input [AddressWidth-1:0]addrIndividual2,
20          output [IndividualWidth-1:0]outIndividual2,
21          output [ErrorWidth-1:0]outError2
22  );
23
24  wire [AddressWidth-1:0]hashAddress;
25  wire [IndividualWidth-1:0]oldIndividual;
26  wire [ErrorWidth-1:0]oldError;
27
28  generate
29          genvar i;
30          for (i=0;i<AddressWidth;i=i+1)
31          begin: __hash__
32                  assign hashAddress[i] = ^ inIndividual[(i+1)*(
                        IndividualWidth/AddressWidth)-1:i*(IndividualWidth/
                        AddressWidth)];
33          end
34  endgenerate
35
36  wire replaceIndividual = (inError < oldError) & we;
37
38  RAM3
39          #(.AddressWidth(AddressWidth),.Width(IndividualWidth))
40  individuals (
41          .clk(clk),
42          .rst(rst),
43          .pst(1'b0),
44          .we(replaceIndividual),
45          .waddr(hashAddress),
46          .D(inIndividual),
47          .r1addr(hashAddress),
```

```
48              .Q1(oldIndividual),
49              .r2addr(addrIndividual1),
50              .Q2(outIndividual1),
51              .r3addr(addrIndividual2),
52              .Q3(outIndividual2)
53  );
54
55  RAM3
56              #(.AddressWidth(AddressWidth),.Width(ErrorWidth))
57  errors (
58              .clk(clk),
59              .rst(1'b0),
60              .pst(rst),
61              .we(replaceIndividual),
62              .waddr(hashAddress),
63              .D(inError),
64              .r1addr(hashAddress),
65              .Q1(oldError),
66              .r2addr(addrIndividual1),
67              .Q2(outError1),
68              .r3addr(addrIndividual2),
69              .Q3(outError2)
70  );
71
72  endmodule
73
74  `endif
```

Além do controle de endereçamento dos nichos e da aptidão dos indivíduos, o módulo da população também disponibiliza a leitura simultânea de dois indivíduos. Essa leitura é posteriormente utilizada pelo módulo de seleção.

A maneira proposta neste trabalho para o armazenamento da população é um grande diferencial. Não foi encontrado nenhum outro trabalho com essa proposta, durante o levantamento bibliográfico.

4.2.4 Números Aleatórios

Como já comentado, em alguns procedimentos do AG é necessário o uso de números aleatórios, como, na mutação, seleção e cruzamento dos indivíduos. A geração de números aleatórios, normalmente, utiliza um valor gerado através da mensuração de fenômenos naturais, que

podem ser imprevisíveis. Exemplos de fenômenos naturais utilizados são, ruídos térmicos ou atmosféricos e decaimento nuclear (JUN; KOCHER, 1999). Porém, geralmente a tradução do fenômeno natural para um valor digital é um processo demorado e/ou pode exigir um *hardware* específico. Para evitar a demora da tradução, o valor gerado naturalmente é utilizado somente para iniciar um sistema, que produz novos valores a partir do inicial. O valor utilizado para iniciar o sistema, nesse caso, pode ser chamado de semente.

No trabalho de Coury et al. (2011) foi utilizada uma tabela com valores aleatórios pré-definidos, mas essa abordagem acaba exigindo mais memória do FPGA. A utilização de LFSR foi abordada no trabalho de Vinhal (2013). Existem diversos tipos de sistemas para geração de números aleatórios a partir de uma semente. Neste projeto, optou-se pela utilização dos autômatos celulares, que já são utilizados em outros trabalhos e têm demonstrado bons resultados (CHEN et al., 2008; APORNTEWAN; CHONGSTITVATANA, 2001; SERRA et al., 1990). Porém, não foi utilizada a saída bruta do AC como número aleatório. Já que, após um número de iterações em qualquer AC com tamanho finito, os estados de suas células começam a se repetir. Esse comportamento poderia prejudicar o funcionamento do AG. Por exemplo, se esse número gerado fosse utilizado para realizar a mutação nos cromossomos, após algumas gerações, os mesmos ciclos de mutação começariam a ocorrer, podendo prejudicar a diversidade genética da população.

Com intuito de melhorar a distribuição dos valores gerados pelos estados do AC, foi desenvolvido um circuito de um AC unidimensional binário, que pode alterar sua regra durante a execução, não somente durante a síntese do código Verilog. Assim, depois de alguns ciclos de máquina onde as células do AC interagiram com uma determinada regra, a regra pode ser trocada e as células do AC começam a interagir de outra forma.

O *hardware* que implementa o AC utiliza um multiplexador de 8 endereços para cada célula, assim, cada multiplexador contém 3 *bits* de endereçamento. Esses 3 *bits* são conectados a célula esquerda, a célula corrente e a célula direita, a saída do multiplexador é utilizada para realimentar a célula corrente. Isso se repete por todas as células. Assim, as entradas do multiplexador definem a regra do AC e as células selecionam os *bits* dos próximos estados. Na Figura 4.7 é exemplificado esse circuito utilizando a regra 30[5].

O código Verilog referente a esse AC é apresentado na Listagem 4.4.

Listagem 4.4: Código Verilog do AC

```
1 `ifndef __DYNAMIC_BINARY_CELLULAR_AUTOMATA_1D__
2 `define __DYNAMIC_BINARY_CELLULAR_AUTOMATA_1D__
```

[5] 30 em binário com 8 algarismos é igual á 00011110.

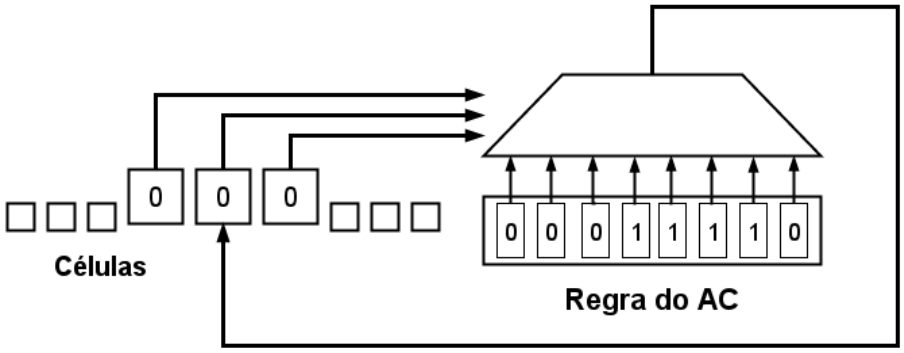

Figura 4.7: *Hardware* do Autômato Celular.

```
3
4  module DynamicBinaryCellularAutomata1D #(
5    parameter Width = 16,
6    parameter Initial= {(Width/2){2'b01}}
7  ) (
8    input clk,
9    input rst,
10   input ce,
11   input [7:0]rule,
12   input [Width-1:0]set,
13   output reg [Width-1:0]state = Initial
14 );
15
16 genvar i;
17 generate
18   for (i=0;i<Width;i=i+1)
19   begin: _cell_
20     always @(posedge clk or posedge rst)
21     begin
22       if (rst)
23       begin
24         state[i] = set[i];
25       end
26       else if (ce)
27       begin
28         state[i] =
29           rule[{state[(i+1)%Width],state[i],state[i-1<0?0:i-1]}];
30       end
31     end
32   end
33 endgenerate
34
```

```
35 endmodule
36
37 `endif
```

Também foi adicionado um registrador para auxiliar na geração dos números aleatórios. A cada ciclo do AC, o registrador é deslocado um *bit* à esquerda de forma cíclica, e é realizado um *ou exclusivo* com o estado atual do AC. Esse processo é exemplificado na Figura 4.8

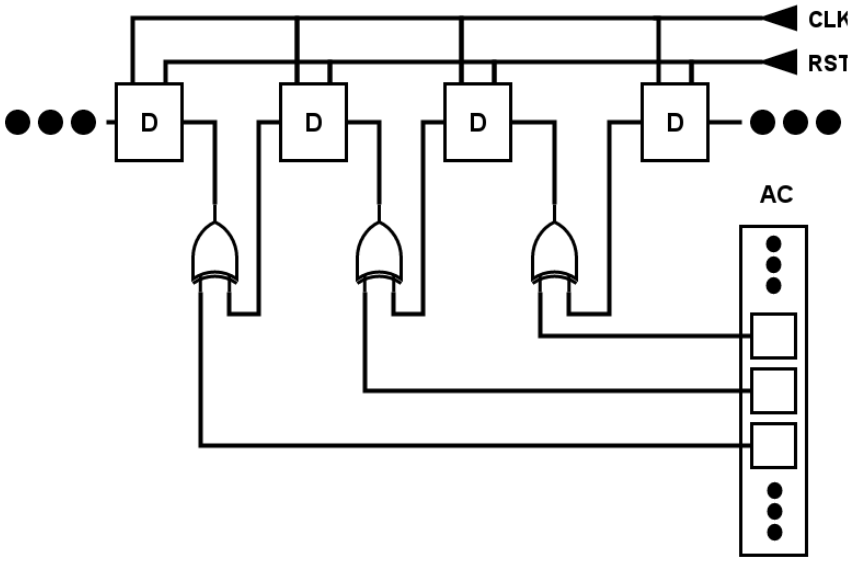

Figura 4.8: Autômato Celular mais Deslocador.

Essa abordagem melhorou a distribuição dos números aleatórios, quando comparada à utilização direta do estado do AC. Na Tabela 4.1 são apresentados valores gerados por um AC com regra 30 e os valores gerados pelo registrador com deslocamento, ambos com 7 *bits*. A partir da iteração 49 os valores do AC começam a se repetir, enquanto os valores do registrador continuam com uma sequência não conhecida. Nas iterações 4 e 9 o registrador contém os mesmos valores (109), porém os valores subsequentes são distintos (79 e 7). Essa característica não seria obtida apenas com o uso do AC.

A semente que define o estado inicial do AC foi gerada uma vez em um computador convencional e inserida estaticamente no código Verilog. Sempre que circuito é reiniciado o estado do AC é redefinido com essa semente.

O gerador de números aleatórios trabalha alternando a regra do AC entre as regras 30, 90 e 150. O código Verilog responsável em fazer essa alternação e de fazer o deslocamento do registrador é apresentado na Listagem 4.5.

Listagem 4.5: Código Verilog do Gerador de Números Aleatórios

```
1 `ifndef __RANDOM__RANDOMIC_LCA__
```

Tabela 4.1: Valores Pseudoaleatórios Gerados com um AC de Regra 30.

Iteração	AC30		AC30 + XOR	
0	1111101	125	1111101	125
1	0000111	7	1111001	121
2	1000101	69	0111001	57
3	1100011	99	0111111	63
4	0110010	50	1101101	109
5	0111001	57	1001111	79
6	1101100	108	0001011	11
7	1111110	126	0111011	59
8	1000011	67	0011110	30
9	1100010	98	1101101	109
10	1110001	113	0000111	7
11	0011001	25	1011010	90
12	1011100	92	1110001	113
13	0110110	54	1001110	78
14	0111111	63	0011000	24
15	1100001	97	1101101	109
16	0110001	49	1000111	71
17	1111000	120	0011011	27
18	1001100	76	0000001	1
19	0101110	46	1101110	110
20	0011011	27	0101100	44
21	1011111	95	1001001	73
22	1110000	112	0010100	20
23	1011000	88	1010010	82
24	0111100	60	0010101	21
25	0100110	38	1101100	108
26	0010111	23	0100001	33
27	1001101	77	0011101	29
28	1101111	111	0100001	33
29	0111000	56	1101000	104
30	0101100	44	0011000	24
31	0011110	30	0010010	18
32	0010011	19	0011010	26
33	1001011	75	1000110	70
34	1100110	102	1000101	69
35	1110111	119	0010101	21
36	0011100	28	1010110	86
37	0010110	22	0111101	61
38	0001111	15	1010001	81
39	1001001	73	0100001	33
40	1100101	101	0110101	53
41	0110011	51	1101001	105
42	1111011	123	0001111	15
43	0001110	14	1001001	73
44	0001011	11	1101111	111
45	1000111	71	0110000	48
46	1100100	100	1111100	124
47	1110010	114	1001100	76
48	1011001	89	1111111	127
49	1111101	125	0000010	2
50	0000111	7	0000110	6
51	1000101	69	1000110	70
52	1100011	99	1000000	64
53	0110010	50	0010010	18
54	0111001	57	0110000	48
55	1101100	108	1110100	116
56	1111110	126	1000100	68
57	1000011	67	1100001	97
58	1100010	98	0010010	18
59	1110001	113	1111000	120
60	0011001	25	0100101	37
61	1011100	92	0001110	14
62	0110110	54	0110001	49
63	0111111	63	1100111	103

```
2  `define __RANDOM__RANDOMIC_LCA__
3
4  `include "ca/DynamicBinaryCellularAutomata1D.v"
5
6  module RandomicLCA #(
7    parameter Width = 8
8  ) (
9    input clk,
10   input rst,
11   input ce,
12   input [Width-1:0]seed,
13   output reg [Width-1:0]random =0
14 );
15
16   reg [5:0] counter = 0;
17   wire [7:0] rule;
18   wire [Width-1:0]caState;
19
20   assign rule =
21       counter[5:4]== 2'b00 ? 8'b00011110 :
22       counter[5:4]== 2'b01 ? 8'b00111100 :
23       counter[5:4]== 2'b10 ? 8'b01011010 : 8'b10010110;
24
25   DynamicBinaryCellularAutomata1D #(
26     .Width(Width)
27   )
28   automata (
29     .clk(clk),
30     .rst(rst),
31     .ce(ce),
32     .rule(rule),
33     .set(seed),
34     .state(caState)
35   );
36
37   always @(posedge clk or posedge rst)
38   begin
39     if (rst)
40     begin
41       counter = 0;
42       random = 0;
43     end
44     else if (ce)
```

```
45      begin
46        random = {random[Width-2:0],random[Width-1]} ^ caState;
47        counter = counter + 1'b1 + ^caState;
48      end
49    end
50
51  endmodule
52
53  `endif
```

4.2.5 Cruzamento

No caso da codificação binária, os métodos de cruzamento possíveis são o uniforme e o de *n*-pontos. Existe uma pequena diferença nos resultados desses dois métodos de cruzamento, no que diz respeito à preservação de esquemas dos indivíduos da população. O cruzamento de *n*-pontos preserva os esquemas dos indivíduos, enquanto muitas vezes o cruzamento uniforme os destroem (TANOMARU, 1995; GALVÃO, 1999). Porém, não existe diferença significativa, que tenha sido comprovada, de desempenho entre os dois. Mesmo assim, optou-se pela utilização do cruzamento *n*-pontos, já que tende a preservar os esquemas dos indivíduos.

O método de cruzamento utilizado foi o cruzamento de *n*-pontos, com *n* igual á 2. Para a implementação desse método, são utilizados dois números gerados pelo RNG com AC, proposto na Seção 4.2.4, esses são utilizados para definir os pontos do cruzamento. Posteriormente os segmentos gerados por esses pontos são permutados entre os indivíduos geradores.

A implementação desse método em *hardware* foi feita utilizando uma máscara de cruzamento, um vetor que define quais *bits* serão copiados para os novos indivíduos. Na Figura 4.9 é exemplificado como essa máscara é gerada. Primeiro um valor aleatório é utilizado para rotacionar um vetor totalmente preenchido com uns, descartando os *bits* excedentes. Depois um outro valor aleatório é utilizado para rotacionar de forma cíclica. Assim, são sempre formadas máscaras condizentes com o cruzamento de *n*-pontos, com $n = 2$.

Essa máscara é posteriormente utilizada para multiplexar o *bits* dos indivíduos geradores (pais), formando dois novos indivíduos (filhos). Esse processo é ilustrado na Figura 4.10, onde foi ilustrada apenas a multiplexação do primeiro *bit*, para simplificação. Porém, o circuito apresentado com dois multiplexadores é replicado para cada um dos *bits* do código genético. Os indivíduos geradores são fornecidos simultaneamente pelo módulo de seleção, que é explicado na Seção 4.2.6.

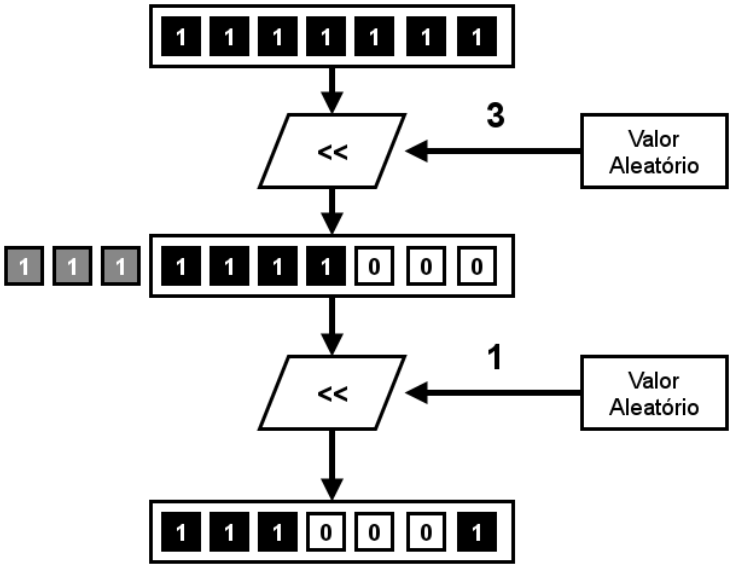

Figura 4.9: Máscara de Cruzamento.

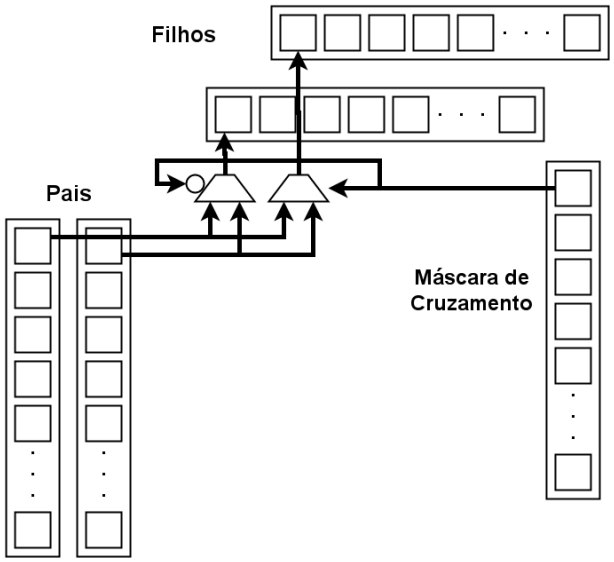

Figura 4.10: Cruzamento

Esse circuito também pode ser utilizado para o cruzamento uniforme, simplesmente trocando a máscara de cruzamento.

4.2.6 Seleção

Como no próprio módulo de armazenamento da população já são realizadas algumas comparações, mantendo somente indivíduos com as melhores aptidões, o módulo de seleção pôde ser simplificado. No módulo de seleção são apenas sorteados aleatoriamente indivíduos da população. Esses são posteriormente utilizados para a reprodução de novos indivíduos.

O módulo da população permite a leitura simultânea de dois indivíduos e o módulo de

cruzamento necessita de dois indivíduos simultaneamente. Assim, o módulo de seleção apenas define dois endereços aleatoriamente e os usa para endereçar os indivíduos para o módulo de cruzamento.

A seleção é feita com a utilização de dois números pseudoaleatórios como endereços de memória. Esses números são gerados a partir de um AC, como apresentado na Seção 4.2.4, e, se por algum motivo os dois forem iguais, em um deles é invertido o *bit* menos significativo. Assim, pode-se dizer que a seleção simplesmente seleciona dois indivíduos da população de forma aleatória. Diferentemente da maioria dos métodos de seleção, não é feita nenhuma comparação para executar a seleção. Isso se deve a como os indivíduos são inseridos na população, afinal, no momento de inserção já é feita uma comparação, eliminando os indivíduos menos aptos da população. A metáfora da pressão seletiva acaba ficando dividida nos módulos da população e da seleção. Na Figura 4.11 é ilustrado o processo de seleção.

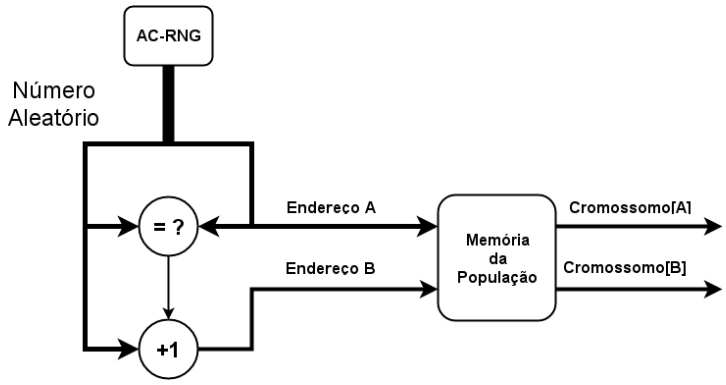

Figura 4.11: Exemplo Gráfico do Processo de Seleção.

4.2.7 Mutação

A mutação para a representação binária, necessita apenas inverter alguns *bits* com uma determinada probabilidade. A probabilidade utilizada foi de 12,5%, uma mutação alta, se comparada com o comum, que é entre 0,5% e 1%. Mas, como a população foi armazenada de uma forma diferente, onde os indivíduos mais aptos não são descartados, a mutação acaba não sendo tão destrutiva. Essa taxa de mutação elevada também simplificou o *hardware*, ao invés de se utilizar comparadores para verificar se um número aleatório gerado, foi maior ou menor do que um valor que representa a taxa de mutação, foi utilizado apenas, a combinação de algumas portas lógicas *e*.

A mutação é realizada utilizando a operação lógica *ou exclusivo*, *bit* a *bit*, entre um indivíduo e uma máscara de mutação. A probabilidade de cada *bit* ter o valor um, nessa máscara, é igual a taxa de mutação (12,5%). A máscara é gerada com a combinação de três vetores aleató-

rios com a operação lógica *e*. Para cada *bit* desses vetores, é realizada a lógica *e* entre os três, como exemplificado na Figura 4.12. Os três vetores aleatórios são extraídos da segmentação de um vetor aleatório maior, gerado com o RNG proposto na Seção 4.2.4.

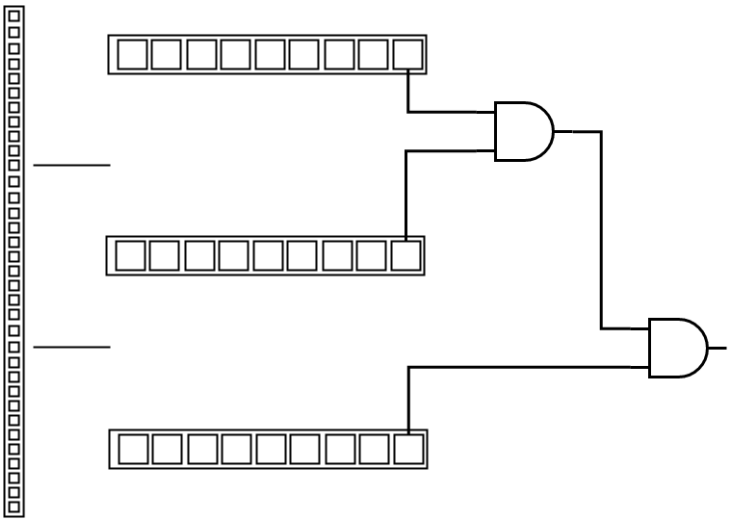

Figura 4.12: Produção da Máscara de Mutação

A tabela verdade da lógica *e* com três entradas é apresentada na Tabela 4.2. Observando a tabela, podemos deduzir que a chance de um *bit* ter o valor 1 é de uma em oito, ou seja, os 12,5% da taxa de mutação.

Tabela 4.2: Tabela Verdade da Lógica *e* com 3 Entradas

A	B	C	Resultado
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Com a máscara pronta, basta executar a operação lógica *ou exclusivo*. A Figura 4.13 exemplifica esse processo.

Não foi desenvolvido um módulo separado para a execução da mutação, mas o trecho de código que faz a mutação é apresentado na Listagem 4.6.

Listagem 4.6: Trecho de Código Verilog Responsável pela Mutação

```
1 RandomicLCA #(
2     .Width(IndividualWidth*3)
```

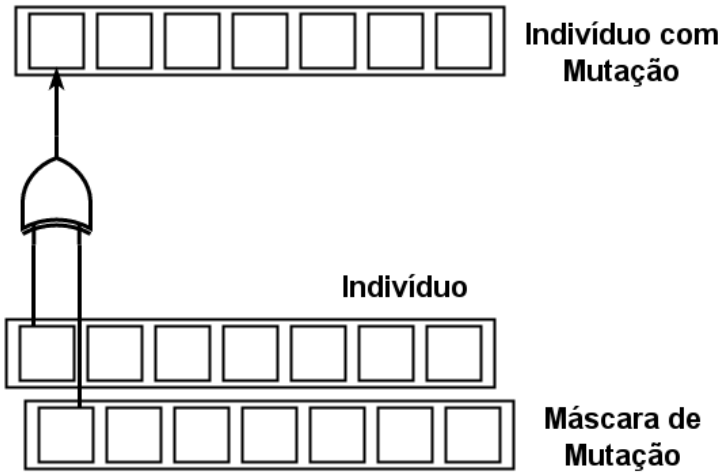

Figura 4.13: Mutação com *ou exclusivo*

```
3  )
4  randomicMutation(
5     .clk(clk),
6     .rst(rst),
7     .ce(~fitnessQueueVoid & fitnessCacheHit),
8     .seed({IndividualWidth*3/32{32'b11000000100010010101111010000001}}),
9     .random(randomMutation)
10 );
11
12 wire [IndividualWidth*3-1:0]randomMutation;
13 wire [IndividualWidth-1:0]mutationMask =
14    randomMutation[IndividualWidth*3-1:IndividualWidth*2] &
15    randomMutation[IndividualWidth*2-1:IndividualWidth] &
16    randomMutation[IndividualWidth-1:0];
17
18 wire[IndividualWidth-1:0]toMutateInidividual;
19 wire [IndividualWidth-1:0]toTestIndividual =
20          toMutateInidividual ^ mutationMask;
```

4.2.8 Fluxo Completo

Agora que todos os componentes que compõe a implementação do AG foram explicados, será apresentado como foi realizada as conexões entre eles e o fluxo de funcionamento desse AG. Para realizar as conexões efetivamente, foi necessário adicionar mais dois componentes,

uma fila e um sistema de *cache*, esses logo serão explicados.

Inicialmente a memória de população inicia zerada, com todos os códigos genéticos iguais a um vetor de zeros. As aptidões começam com o valor mínimo, no caso, como o que é armazenado é o resultado da função objetivo, e se trata de um problema de minimização, são vetores com o valor um. O módulo de seleção sorteia dois indivíduos aleatoriamente, que são enviados para o módulo de cruzamento. Esses indivíduos são combinados gerando dois novos indivíduos. Nesse primeiro momento, o módulo de cruzamento acaba combinando dois vetores de zeros, resultando em outros dois vetores de zeros. Os dois novos indivíduos são inseridos em uma fila. Um dos motivos para a inserção da fila, foi para afunilar os novos indivíduos, pois esses são gerados de dois em dois, enquanto o módulo de mutação e o de aptidão trabalham com apenas um de cada vez. O módulo de seleção e o de cruzamento, continuam trabalhando até que a fila fique cheia.

Na saída da fila fica conectado o módulo de mutação, esse por sua vez retira indivíduos da fila e realiza a mutação. Os primeiros indivíduos são vetores de zeros, pois o módulo de cruzamento não tinha material genético para combinar, assim, o módulo de mutação fica responsável em gerar a população inicial. Antes de enviar o indivíduo para o módulo de avaliação, ele é procurado em um sistema de *cache*, esse sistema armazena o resultado da função objetivo. Caso o resultado da função objetivo não esteja em *cache*, o indivíduo é enviado para o módulo de aptidão, utilizando a interface que foi definida, essa comentada na Seção 4.2.2 referente ao módulo de aptidão. Assim que o módulo de aptidão termina seu processo, o indivíduo e o resultado da função objetivo é inserido no *cache* e enviado para o módulo da população. No início a maioria dos indivíduos são aceitos pelo módulo de população, já que ele inicia zerado. O módulo de mutação aguarda todo esse processo pata retirar outro indivíduo da fila.

Depois de alguns ciclos, a memória da população já está cheia de indivíduos não nulos, assim, o módulo de cruzamento começa efetivamente combinar material genético. O sistema de *cache* não armazena todos os resultados da função objetivo, apenas dos indivíduos recém testados. Durante as primeiras gerações do AG, muitos indivíduos iguais são gerados, até que a memória da população seja preenchida, nesse momento o sistema de *cache* é muito eficaz.

Todas essas interconexões são ilustradas na Figura 4.14.

O código Verilog que descreve o módulo genético na Figura 4.14 , é apresentado na Listagem 4.7.

Listagem 4.7: Código Verilog do Módulo Genético

```
1 `ifndef __ETTORE_AG__
2 `define __ETTORE_AG__
```

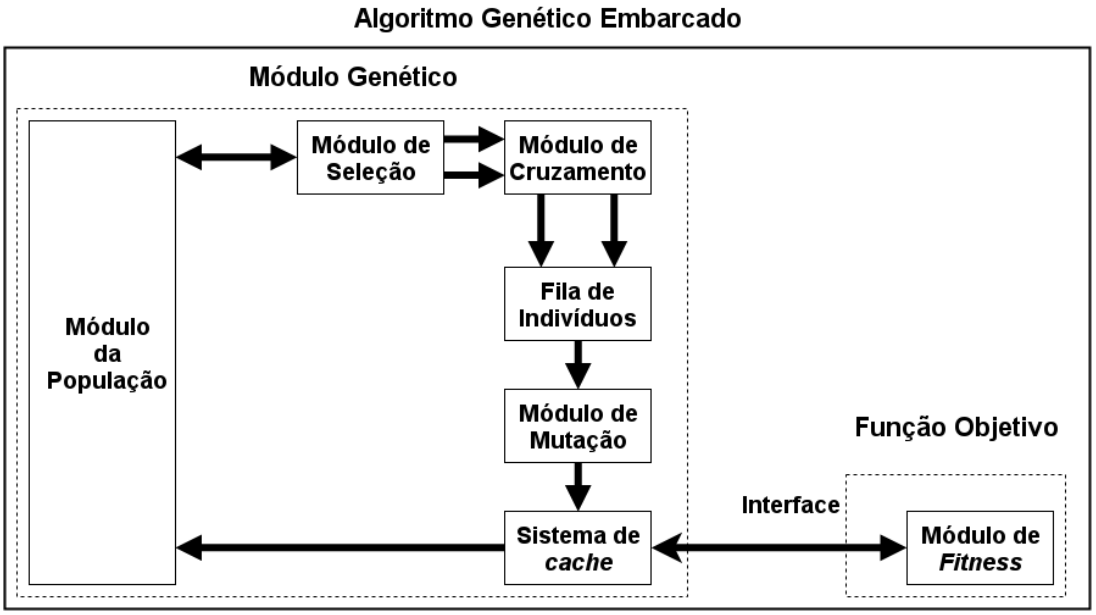

Figura 4.14: Diagrama de Blocos do *Hardware* Genético

```
3
4  `include "ga/IndividualsCache.v"
5  `include "ga/Crossover.v"
6  `include "random/RandomicLCA.v"
7  `include "util/Queue2To1.v"
8  `include "memory/DirectCache.v"
9
10 module EttoreAG #(
11   parameter ErrorWidth=32,
12   parameter IndividualWidth=32,
13   parameter PopulationAddressWidth=5
14 ) (
15   input clk,
16   input rst,
17
18   output fitnessStart,
19   input fitnessFinish,
20   output [IndividualWidth-1:0]fitnessIndividual,
21   input [ErrorWidth-1:0]fitnessError,
22   output reg [IndividualWidth-1:0]bestIndividual,
23   output reg [ErrorWidth-1:0]bestError
24 );
25
26 IndividualsCache #(
27   .IndividualWidth(IndividualWidth),
28   .ErrorWidth(ErrorWidth),
```

```
29      .AddressWidth(PopulationAddressWidth)
30  )
31  individualsCache (
32      .clk(clk),
33      .rst(rst),
34      .we(fitnessCacheHit),
35      .inIndividual(toTestIndividual),
36      .inError(cachedError),
37      .addrIndividual1(dadAddress),
38      .outIndividual1(dadIndividual),
39      .addrIndividual2(momAddress==dadAddress?momAddress^1'b1:momAddress),
40      .outIndividual2(momIndividual)
41  );
42
43  wire [PopulationAddressWidth-1:0]dadAddress;
44  wire [PopulationAddressWidth-1:0]momAddress;
45  wire [IndividualWidth-1:0]dadIndividual;
46  wire [IndividualWidth-1:0]momIndividual;
47  wire [IndividualWidth-1:0]sonIndividual;
48  wire [IndividualWidth-1:0]daughterIndividual;
49
50  RandomicLCA #(
51      .Width(PopulationAddressWidth*2)
52  )
53  randomicSelector(
54      .clk(clk),
55      .rst(rst),
56      .ce(~fitnessQueueFull),
57      .seed({PopulationAddressWidth{32'b11000000100010010101111010000001}}),
58      .random({dadAddress,momAddress})
59  );
60
61  Crossover #(
62      .IndividualWidth(IndividualWidth)
63  )
64  crossover(
65      .clk(clk),
66      .rst(rst),
67      .ce(~fitnessQueueFull),
68      .dad(dadIndividual),
69      .mom(momIndividual),
70      .son(sonIndividual),
71      .daughter(daughterIndividual)
```

```
72 );
73
74 wire fitnessQueueVoid;
75 wire fitnessQueueFull;
76
77 Queue2To1 #(
78    .Width(IndividualWidth),
79    .AddressWidth(3)
80 )
81 fitnessQueue(
82    .clk(clk),
83    .rst(rst),
84    .push(~fitnessQueueFull),
85    .pull(~fitnessQueueVoid & fitnessCacheHit),
86    .D({sonIndividual, daughterIndividual}),
87    .Q(toMutateInidividual),
88    .void(fitnessQueueVoid),
89    .full(fitnessQueueFull)
90 );
91
92 RandomicLCA #(
93    .Width(IndividualWidth*3)
94 )
95 randomicMutation(
96    .clk(clk),
97    .rst(rst),
98    .ce(~fitnessQueueVoid & fitnessCacheHit),
99    .seed({IndividualWidth*3/32{32'b11000000100010010101111010000001}}),
100   .random(randomMutation)
101 );
102
103 wire [IndividualWidth*3-1:0]randomMutation;
104 wire [IndividualWidth-1:0]mutationMask =
105   randomMutation[IndividualWidth*3-1:IndividualWidth*2] &
106   randomMutation[IndividualWidth*2-1:IndividualWidth] &
107   randomMutation[IndividualWidth-1:0];
108
109 wire[IndividualWidth-1:0]toMutateInidividual;
110 wire [IndividualWidth-1:0]toTestIndividual =
111   toMutateInidividual ^ mutationMask;
112 wire [ErrorWidth-1:0]cachedError;
113 wire fitnessCacheHit;
114 wire fitnessCacheMiss;
```

```
115
116 assign fitnessStart = fitnessCacheMiss & ~fitnessQueueVoid;
117 assign fitnessIndividual = toTestIndividual;
118
119 DirectCache #(
120   .Width(ErrorWidth),
121   .AddressWidth(IndividualWidth),
122   .CacheAddressWidth(4)
123 )
124 fitnessCache (
125   .clk(clk),
126   .rst(rst),
127   .we(fitnessFinish),
128   .hit(fitnessCacheHit),
129   .miss(fitnessCacheMiss),
130   .raddr(toTestIndividual),
131   .Q(cachedError),
132   .waddr(toTestIndividual),
133   .D(fitnessError)
134 );
135
136 always @(posedge clk or posedge rst) begin
137   if (rst)
138   begin
139     bestIndividual = {IndividualWidth{1'b0}};
140     bestError = {ErrorWidth{1'b1}};
141   end
142   else
143   if (fitnessCacheHit) begin
144     if(cachedError < bestError)
145     begin
146       bestError = cachedError;
147       bestIndividual = toTestIndividual;
148     end
149   end
150 end
151
152 endmodule
153
154 `endif
```

A conexão entre o módulo genético e o módulo da função objetivo, é descrita pelo código

Verilog apresentado na Listagem 4.8.

Listagem 4.8: Trecho de Código Verilog Responsável pela Mutação

```
1  `ifndef __GA__MORPHOLOGIC_GENETIC_ALGORITHM__
2  `define __GA__MORPHOLOGIC_GENETIC_ALGORITHM__
3
4  `include "ga/MorphologicFitness.v"
5  `include "ga/EttoreAG.v"
6
7  module MorphologicGeneticAlgorithm #(
8          parameter ImageWidth = 8,
9          parameter ImageHeight = 4,
10         parameter ErrorWidth = $clog2(ImageHeight*ImageWidth),
11         parameter OpcodeWidth = 16,
12         parameter OpCounterWidth = 2,
13         parameter InstructionWidth = OpcodeWidth * (2**OpCounterWidth),
14         parameter IndividualWidth = InstructionWidth
15 ) (
16         input clk,
17         input rst,
18         input [ImageHeight*ImageWidth-1:0]origin,
19         input [ImageWidth*ImageHeight-1:0]objetive,
20         output [IndividualWidth-1:0]bestIndividual,
21         output [ErrorWidth-1:0]bestError,
22         output cycle
23 );
24
25 wire fitnessStart;
26 assign cycle = fitnessStart;
27 wire fitnessFinish;
28 wire [IndividualWidth-1:0]fitnessIndividual;
29 wire [ErrorWidth-1:0]fitnessError;
30
31 EttoreAG #(
32         .ErrorWidth(ErrorWidth),
33         .IndividualWidth(IndividualWidth),
34         .PopulationAddressWidth(4)
35 )
36 ag(
37         .clk(clk),
38         .rst(rst),
39         .bestIndividual(bestIndividual),
40         .bestError(bestError),
```

```
41          .fitnessIndividual(fitnessIndividual),
42          .fitnessError(fitnessError),
43          .fitnessStart(fitnessStart),
44          .fitnessFinish(fitnessFinish)
45  );
46
47  MorphologicFitness #(
48          .ImageWidth(ImageWidth),
49          .ImageHeight(ImageHeight),
50          .ErrorWidth(ErrorWidth),
51          .OpcodeWidth(OpcodeWidth),
52          .OpCounterWidth(OpCounterWidth),
53          .InstructionWidth(InstructionWidth)
54  )
55  fitness(
56          .clk(clk),
57          .rst(rst),
58          .origin(origin),
59          .objetive(objetive),
60          .individual(fitnessIndividual),
61          .start(fitnessStart),
62          .finish(fitnessFinish),
63          .error(fitnessError)
64  );
65
66  endmodule
67
68  `endif
```

4.3 Resultados

O novo modelo proposto para o armazenamento da população, foi concebido visando uma implementação em *hardware*, mas sua implementação em *software* também é viável. Antes da implementação em *hardware* o modelo foi testado em *software*, sendo comparado com algoritmos tradicionais. Os resultados das implementações em *hardware* e em *software* são apresentados nesta seção.

4.3.1 Implementação em *software*

Para verificar o desempenho do modelo proposto, foi realizado um *benchmark* entre ele e outros AGs. O objetivo do *benchmark*, apresentado nesta seção, não é comparar o modelo proposto com soluções comerciais, onde o código já está otimizado e cada solução pode estar utilizando tecnologias diferentes. O foco é comparar a diferença de desempenho com apenas diferenças conceituais e, por esse motivo, todos os algoritmos foram reimplementados utilizando a mesma linguagem de programação. A linguagem utilizada foi o Python versão 2.7.5.

O computador utilizado para os testes foi um *notebook* pessoal, com as seguintes configurações: processador Intel Core I7 (2620M, 2.7GHz), 8GB de memória RAM (DDR3, 1333Mhz) e sistema operacional Fedora 19.

O *benchmark* foi realizado com quatro algoritmos genéticos, sendo um deles o modelo proposto nesse trabalho, identificado como Paridade. Os testes foram realizados sobre as cinco funções de testes propostas por DeJong (1975). Essas funções foram enumeradas como F1, F2, F3, F4 e F5, e são representadas pelas equações 4.8, 4.9, 4.10, 4.11 e 4.12 respectivamente.

$$f(x,y,z) = x^2 + y^2 + z^2 \; \forall \, x,y,z \; \in [-5,15;5,12] \tag{4.8}$$

$$f(x,y) = 100(x^2 - y^2) + (1-x)^2 \; \forall \, x,y \; \in \; [-2,048;2,048] \tag{4.9}$$

$$f(x_1,...,x_5) = \sum_{i=1}^{5} trunc(|x_i|) \; \forall \, x_i \; \in \; [-5,12;5,12] \tag{4.10}$$

$$f(x_1,...,x_{30}) = \sum_{i=1}^{30} ix^4 + \phi \; \forall \, x_i \; \in \; [-1,28;1,28] \tag{4.11}$$

$$f(x,y) = 0,002 + \sum_{j=1}^{26} \frac{1}{j + (x-10)^6 + (y-10)^6} \; \forall \, x,y \; \in \; [-65,536;65,536] \tag{4.12}$$

Para cada teste, cada algoritmo foi executado 100 vezes, os dados apresentados na Ta-

bela 4.3 são uma média dos resultados desses testes. A coluna Compacto é referente a uma implementação utilizando um AG Compacto. As colunas roleta e torneio, são referentes a dois AGs com variação nos métodos de seleção, um com o método de seleção Roda da Roleta utilizando ordenação linear e o outro utilizando seleção por torneio com grupos de 3 indivíduos. Todos os algoritmos utilizaram população de 256 indivíduos exceto o Paridade que utilizou 32, já que a intenção desse algoritmo é conseguir resultados semelhantes aos demais com uma população menor. Os AGs foram executados até 2000 gerações ou até atingir 10^{-5} de erro, foi necessário limitar a quantidade de gerações pois nem sempre os AGs alcançavam o erro mínimo. A representação cromossômica escolhida foi a binária, possibilitando realizar a paridade de *bits*, necessária para o modelo proposto. Para cada parâmetro das funções, foi utilizado 20 *bits*. A taxa de cruzamento foi de 80% exceto para o paridade que foi de 100%, como implementado em *hardware*. A taxa de mutação foi de 1% exceto para o paridade que foi de 12%.

Tabela 4.3: Resultados dos Testes de DeJong

		Paridade	**Compacto**	**Roleta**	**Torneio**
Duração (Segundos)	F1	10,36379354	1,879268429	82,18831701	80,80279752
	F2	7,602419217	63,01063721	78,60783545	23,10633699
	F3	1,137697849	0,2180329061	0,2428336835	0,2051712155
	F4	93,98048776	968,130286	126,2219337	150,8650511
	F5	10,46557855	83,81429649	75,86356176	79,14254826
	Média	**24,70999538**	**223,4105042**	**72,62489632**	**66,82438103**
Erro	F1	$2,43 \times 10^{-2}$	$6,91 \times 10^{-6}$	0,0002695130501	$6,03 \times 10^{-5}$
	F2	$1,13 \times 10^{-3}$	$2,81 \times 10^{-4}$	0,0004682196962	$6,46 \times 10^{-6}$
	F3	0	0	0	0
	F4	38,92423835	4,257249218	60,78557954	46,39949322
	F5	0,00200000007	0,002000000067	0,002000000067	0,002000000067
	Média	**7,79**	**0,852**	**12,15766345**	**9,28**
Gerações	F1	1979.18	34,27	1990,91	1893,34
	F2	2000	1805,66	1933,41	743,82
	F3	134.28	1,74	6,03	4,84
	F4	2000	2000	2000	2000
	F5	2000	2000	2000	2000
	Média	**1622,692**	**1169,334**	**1587,07**	**1329,4**

4.3.2 Implementação em *Hardware*

Para realizar testes de maneira mais efetiva, foi adicionado um circuito para comunicação serial entre o AG e um computador tradicional. Para isso foi necessário utilizar um conversor USB para serial compatível com TTL[6], esse apresentado na Figura 4.15.

[6]Do inglês *Transistor-Transistor Logic*

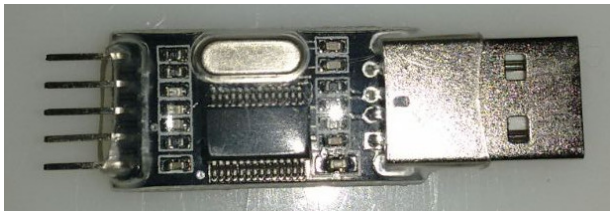

Figura 4.15: Conversor USB para Serial TTL

O computador envia um par de imagens e recebe, como resposta, um indivíduo, a quantidade de indivíduos testados e o erro do indivíduo. Na Figura 4.16 é exemplificada essa comunicação.

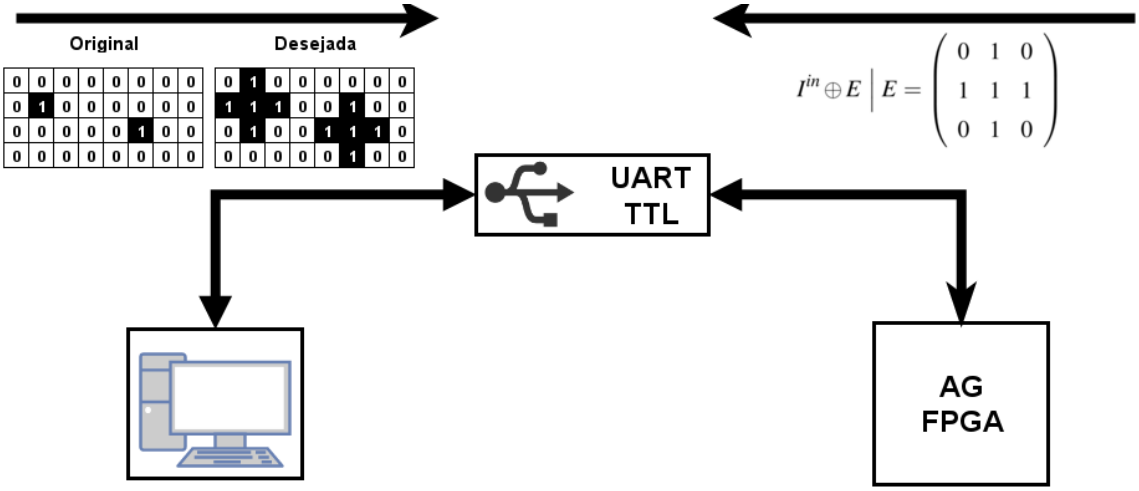

Figura 4.16: Diagrama de Blocos do *Hardware* Genético

Nessa fase, o AG foi configurado pelo código Verilog para trabalhar com uma população de 16 indivíduos, cada um com 32 *bits* e cada imagem com tamanho 8x4. Esse circuito foi sintetizado para três FPGAs e o consumo lógico correspondente pode ser visto na Tabela 4.4.

Tabela 4.4: Consumo Lógico das Sínteses do AG.

FPGA	Elemetos Lógicos	Funções de Combinação	Registradores Lógicos		Frequencia Máxima
Cyclone II EP2C5T144C6	4.363 (95%)	3.621 (79%)	2.396 (52%)		40,57 MHz
Cyclone IV EP4CE115F29C7	4.526 (4%)	3.623 (3%)	2.396 (2%)		43,2 MHz
FPGA	Utilização Lógica	ALUTs	ALUTs de Memória	Registradores Lógicos	Frequencia Máxima
Stratix III EP3SL150F1152C2	3%	2.771 (2%)	0 (0%)	2.400 (2%)	81,98 MHz

O FPGA utilizado para os testes práticos foi o Cyclone II EP2C5T144C6, a placa de circuito impresso com esse FPGA é apresentado na Figura 4.17. Para realizar a gravação do FPGA foi

utilizado um gravador USB, esse apresentado na Figura 4.18. O *kit* da placa do FPGA e seu gravador pode se encontrado no *site* do Prof. Emerson Martins[7].

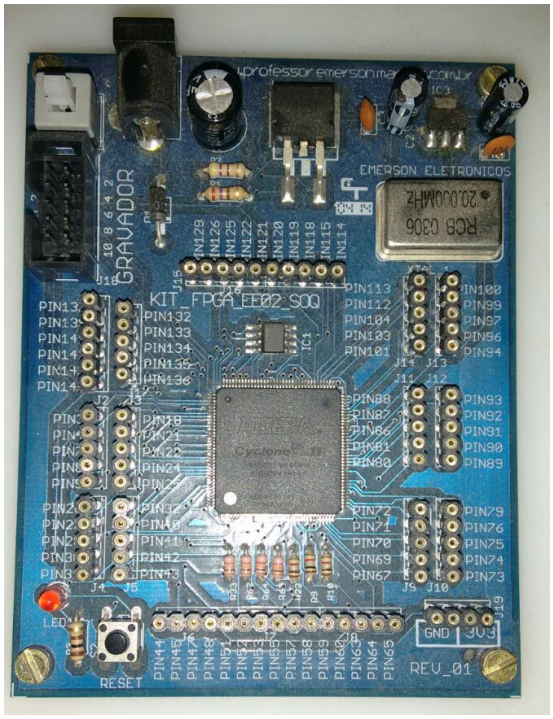

Figura 4.17: Placa de Circuito Impresso com o FPGA Cyclone II EP2C5T144C6

Figura 4.18: Gravador do FPGA

Para verificar a velocidade de processamento, cinco pares de imagens foram enviados para o FPGA, e o tempo de resposta foi calculado. Esse processo foi repetido 100 vezes para cada par de imagem. A velocidade da comunicação serial foi configurada para 4800 *bits* por segundo. A Tabela 4.5 apresenta a média dos 100 resultados em cada caso de teste para a síntese no FPGA Cyclone II EP2C5T144C6[8].

As figuras correspondentes para os códigos hexadecimais da tabela, são apresentados a seguir.

[7] http://www.professoremersonmartins.com.br

[8] Vídeo da execução dos testes disponível em: https://www.youtube.com/watch?v=ogeisHdAo6I

Tabela 4.5: Resultado do AG Serial no FPGA Cyclone II

Nº	Original	Desejada	Indivíduo	Ciclos	Erro	Tempo
1	0x10381000	0x387C3810	0x55192C68	408	0	35,75ms
2	0x00100000	0x387C3810	0x5D5DBF90	1976	0	38,26ms
3	0x00100000	0x386C3810	0xCF8E5D5D	5545	0	43,98ms
4	0x00181800	0x183C3C18	0x151DA3B5	753	0	36,34ms
5	0x00400400	0x04E44E04	0x5515CCA8	592	0	36,06ms

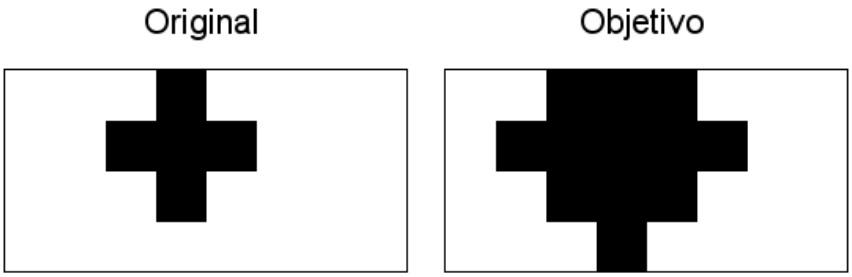

Figura 4.19: Caso de Teste 1, 0x10381000 e 0x387C3810

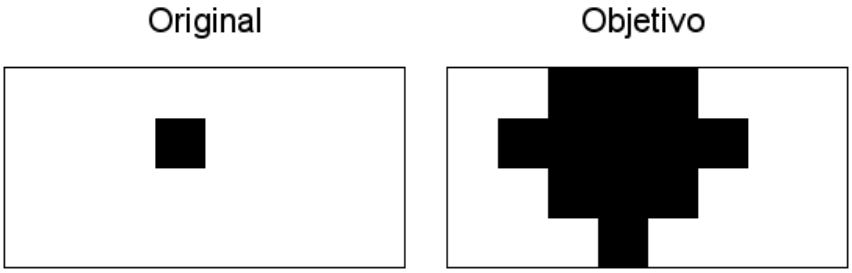

Figura 4.20: Caso de Teste 2, 0x00100000 e 0x387C3810

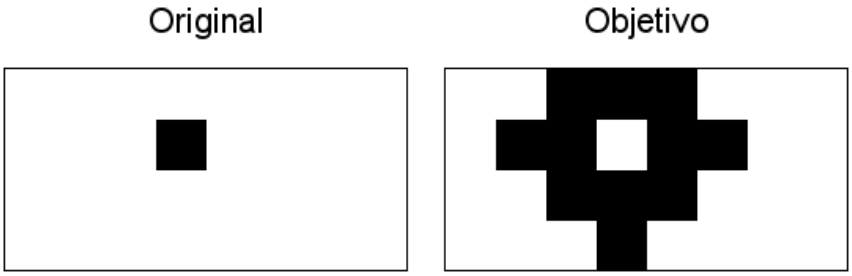

Figura 4.21: Caso de Teste 3, 0x00100000 e 0x386C3810

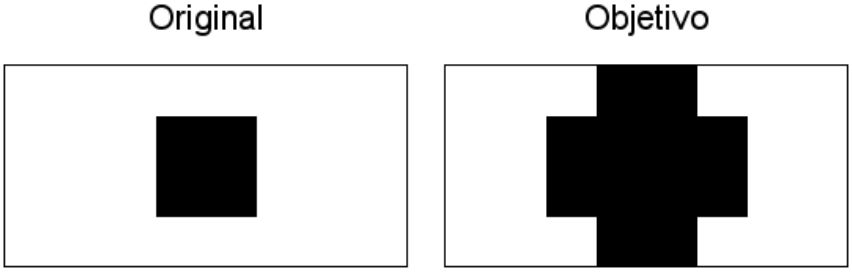

Figura 4.22: Caso de Teste 4, 0x00181800 e 0x183C3C18s

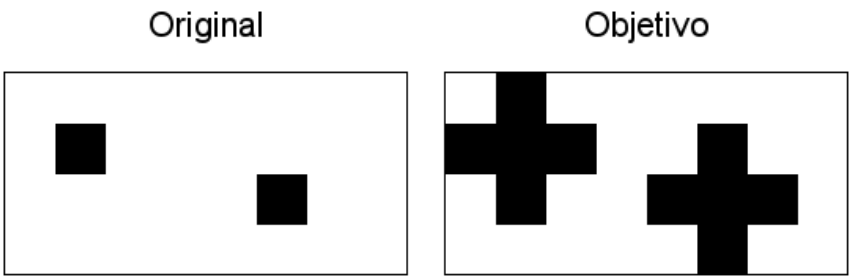

Figura 4.23: Caso de Teste 5, 0x00400400 e 0x04E44E04

4.4 Considerações Finais

Os testes utilizando os FPGAs foram executados praticamente de forma instantânea e encontraram soluções para pequenos problemas. Com os testes realizados não é possível afirmar que problemas, com imagens de alta resolução ou que necessitem de muitas operações, serão resolvidos tão rapidamente. Porém, é possível afirmar que o AG funcionou corretamente e não teve problemas com relação ao desempenho ou com convergências prematuras.

O consumo lógico do circuito desenvolvido é muito dependente da função objetivo. Nesse caso, a função objetivo tem grande consumo. Para problemas com funções objetivo mas simples, ou com circuitos mais otimizados, o *hardware* poderia ser utilizado em FPGAs de pequeno porte. Como no caso deste trabalho, onde a função objetivo apesar de complexa, foi utilizada somente com pares de imagens pequenas, possibilitando sua síntese em um FPGA de baixa densidade.

Mesmo conseguindo processar a imagem e executar o AG dentro do FPGA, ainda existe um gargalo no sistema. O envio do par de imagens e a recepção da resposta, ambos feitos por um computador tradicional, ainda são feitos de forma ineficiente. Provavelmente ao trabalhar com imagens de alta resolução ou com *frames* de um vídeo, seria necessário planejar uma forma melhor de realizar essa comunicação.

Capítulo 5

CONCLUSÃO

5.1 Contribuições e Limitações

A contribuição principal deste projeto é o novo modelo de implementação de AG, que é mais intuitivo para o desenvolvimento em *hardware*, sem comprometimento significativo de desempenho. O novo modelo não mostrou ter desempenho superior ou ser mais robusto que os outros comparados neste projeto, mas se manteve em uma média razoável; entretanto traz grande economia de memória comparado com os AGs tradicionais, que armazenam toda a população em memória, e também, torna a implementação em *hardware* mais intuitiva.

A implementação do novo modelo e do processador morfológico binário em *hardware* pode ser vista com uma contribuição secundária. A implementação do AG em *hardware* pode servir de referência para implementações posteriores. O processador morfológico também pode servir de referência, porém, sua utilização pode ser inviável devido à largura de barramento necessária para trabalhar com imagens grandes.

Todos os códigos fonte desenvolvidos neste projeto ficarão disponíveis em um repositório *online*, chamado *github*[1]. Espera-se com isso fomentar a contribuição de códigos de HDL e facilitar o reúso dos componentes desenvolvidos neste projeto.

5.2 Lições Aprendidas

Manter o código fonte compatível com os dois principais fabricantes de FPGAs, Xilinx e Altera, é uma tarefa muito trabalhosa. Alguns pequenos detalhes no código podem fazer com que o projeto não funcione em FPGAs de fabricantes diferentes. As simulações lógicas

[1] https://github.com/ettoreleandrotognoli/genetic-hardware

realizadas pelo Icarus Verilog e o GTKWave não detectam esses tipos de problemas. Por esse motivo, é muito complicado possibilitar o reúso efetivo de módulos e, muitas vezes, o código fica dependente da tecnologia utilizada. A IDE da Xilinx, também não facilita a utilização de bibliotecas customizadas de HDL, esse fato acabou motivando o uso exclusivo de FPGAs da Altera.

As linguagens de descrição de *hardware* não parecem ser tão populares como as linguagens de programação mais tradicionais, como C/C++, Java, PHP e Python, que possuem grandes comunidades ativas, diversas bibliotecas e *frameworks*. Os repositórios de HDL parecem ainda estar engatinhando, quando comparados com os repositórios de linguagens de programação mais comuns.

5.3 Trabalhos Futuros

O novo modelo sugerido neste projeto necessita de um algoritmo para determinar os endereços dos indivíduos. Por enquanto, foi utilizada a paridade de segmentos dos códigos genéticos. Talvez um pequeno algoritmo de *hash* possa ser utilizado no lugar, mas não se sabe os efeitos que essa modificação pode acarretar. Uma pesquisa futura poderia explorar novas formas para determinar os endereços dos indivíduos. É possível que para cada problema seja mas eficiente ter um novo método para determinação dos endereços.

A implementação produzida neste trabalho não está preparada para paralelizar a função de *fitness*. Paralelizar a função de *fitness* poderia aumentar drasticamente o desempenho, principalmente em casos de problemas mais complexos. Porém, outra abordagem seria a utilização de AGs insulares. Dessa forma, não somente a função de *fitness* seria paralelizada, mas também todo o restante do algoritmo. Preparar a implementação deste trabalho para qualquer uma dessas abordagens poderia melhorar muito seu desempenho.

AGs possuem diversos tipos de variações que podem melhorar seu funcionamento, como, por exemplo, variar a taxa de mutação no decorrer das gerações, iniciando com uma taxa elevada e gradativamente diminuí-la. Essa abordagem pode melhorar a distribuição dos indivíduos no inicio e acelerar a convergência no final. Outro exemplo pode ser o genocídio periódico, que pode ser executado quando o AG converge para uma solução não ótima e acaba ficando preso. Inserir indivíduos aleatórios na população periodicamente para melhorar a diversidade genética, também seria um exemplo. Enfim, existem diversos tipos de abordagens que poderiam ser adicionadas a implementação. Mas encontrar uma abordagem que poderia contribuir significativamente sem aumentar drasticamente o consumo lógico pode ser um desafio.

Pretende-se prosseguir com a manutenção da biblioteca HDL desenvolvida, corrigindo possíveis erros e adicionando novos recursos úteis. Espera-se que essa biblioteca possa ser utilizada em diversos projetos de desenvolvimento de *hardware* com Verilog.

REFERÊNCIAS

ABBOTT, L.; HARALICK, R. M.; ZHUANG, X. Pipeline architectures for morphologie image analysis. *Machine Vision and Applications*, Springer, v. 1, n. 1, p. 23–40, 1988.

ANDREADIS, I.; GASTERATOS, A.; TSALIDES, P. An asic for fast grey-scale dilation. *Microprocessors and Microsystems*, Elsevier, v. 20, n. 2, p. 89–95, 1996.

APORNTEWAN, C.; CHONGSTITVATANA, P. A hardware implementation of the compact genetic algorithm. In: CITESEER. *IEEE Congress on Evolutionary Computation*. [S.l.], 2001. p. 624–629.

BÄCK, T.; HAMMEL, U.; SCHWEFEL, H.-P. Evolutionary computation: Comments on the history and current state. *Evolutionary computation, IEEE Transactions on*, IEEE, v. 1, n. 1, p. 3–17, 1997.

BALA, J.; WECHSLER, H. Shape analysis using genetic algorithms. *Pattern Recognition Letters*, Elsevier, v. 14, n. 12, p. 965–973, 1993.

BANON, G. J. F.; BARRERA, J. *Bases da Morfologia Matemática para a análise de imagens binárias*. [S.l.]: Universidade Federal de Pernambuco, Departamento de Informática, 1994.

BOURIDANE, A. et al. A high level fpga-based abstract machine for image processing. *Journal of systems architecture*, Elsevier, v. 45, n. 10, p. 809–824, 1999.

BROGGI, A. Speeding-up mathematical morphology computations with special-purpose array processors. In: IEEE. *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*. [S.l.], 1994. v. 1, p. 321–330.

CANDEIAS, A. L. B.; BANON, G. J. F. Aplicação da morfologia matemática e analise de imagens de sensoriamento remoto. 1997.

CHEN, P.-Y. et al. Hardware implementation for a genetic algorithm. *Instrumentation and Measurement, IEEE Transactions on*, IEEE, v. 57, n. 4, p. 699–705, 2008.

COMER, M. L.; DELP, E. J. Morphological operations for color image processing. *Journal of electronic imaging*, International Society for Optics and Photonics, v. 8, n. 3, p. 279–289, 1999.

COURY, D. V. et al. Evolutionary algorithm for frequency estimation in electrical systems using fpgas. *Sba: Controle & Automação Sociedade Brasileira de Automatica*, SciELO Brasil, v. 22, n. 5, p. 495–505, 2011. Disponível em: <http://www.scielo.br/scielo.php?pid=S0103-17592011000500005&script=sci_arttext>.

DARWIN, C.; GREEN, J. *A origem das espécies*. [S.l.]: Lelo & Irmão, 2005.

DEJONG, K. A. *Analysis of the behavior of a class of genetic adaptive systems*. Tese (Doutorado), 1975.

EIBEN, A. E.; SMITH, J. E. *Introduction to evolutionary computing*. [S.l.]: Springer, 2003. v. 53.

ESHELMAN, L. J.; SCHAFFER, D. J. Real-coded genetic algorithms and interval-schemata. *Foundations of Genetic Algorithms*, Morgan-Kaufmann, v. 2, p. 187–203, 1993.

FACON, J. *Morfologia Matemática, Teoria e Exemplos. Editora Universitária Champagnat*. [S.l.]: Pontifica Universidade Católica do Paraná Curitiba, 1996.

FILHO, O. M.; NETO, H. V. *Processamento digital de imagens*. [S.l.]: Brasport, 1999.

FOGEL, D. B. *Evolving artificial intelligence*. Tese (Doutorado), 1992.

FOGEL, L. J. *On the organization of intellect*. Tese (Doutorado), 1964.

GALVÃO, C. d. O. *Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais*. [S.l.]: Ed. Universidade, 1999. (Colecao ABRH de Recursos Hidricos). ISBN 9788570255273.

GONZALEZ, R. C.; WOODS, R. E. *Processamento de imagens digitais*. [S.l.]: Edgard Blucher, 2000.

GREMONINI, L.; VICENTINI, E. Autômatos celulares: revisão bibliográfica e exemplos de implementações. *Revista eletrônica lato sensu–UNICENTRO*, v. 6, 2008.

GTKWAVE. *GTKWave Main Page*. 2016. Disponível em: <http://gtkwave.sourceforge.net/>. Acesso em: 16 mar. 2016.

HARIK, G. R.; LOBO, F. G.; GOLDBERG, D. E. The compact genetic algorithm. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 3, n. 4, p. 287–297, 1999.

HARVEY, N. R.; MARSHALL, S. The use of genetic algorithms in morphological filter design. *Signal Processing: Image Communication*, Elsevier, v. 8, n. 1, p. 55–71, 1996.

HOLLAND, J. H. Adaptation in natural and artificial systems. 1975.

HRBACEK, R.; SIKULOVA, M. Coevolutionary cartesian genetic programming in fpga. In: *Advances in Artificial Life, ECAL*. [S.l.: s.n.], 2013. v. 12, p. 431–438.

JUN, B.; KOCHER, P. The intel random number generator. *Cryptography Research Inc. white paper*, 1999.

KIM, H. Y. *Construção automática de operadores morfológicos por aprendizagem computacional*. Tese (Doutorado) — Universidade de São Paulo, 1997.

KLEIN, J.; SERRA, J. The texture analyser. *Journal of microscopy*, Wiley Online Library, v. 95, n. 2, p. 349–356, 1972.

KOJIMA, S.; EBISAWA, Y.; MIYAKAWA, T. Fast morphology hardware using large-sized structuring element. *Systems and computers in Japan*, Wiley Online Library, v. 25, n. 6, p. 41–49, 1994.

LARRANAGA, P. et al. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, Springer, v. 13, n. 2, p. 129–170, 1999.

NETO, A. R. S. H. C. Processamento morfológico de imagens digitais em fpga. *VIII Jornadas sobre Sistemas Reconfiguráveis*, p. 105–112, 2012.

ORDOÑEZ, E. D. M. et al. *Projeto, desempenho e aplicações de sistemas digitais em circuitos programáveis (FPGAs)*. [S.l.]: Pompéia: Bless, 2003.

PACHECO, M. A. C. Algoritmos genéticos: princípios e aplicações. *ICA: Laboratório de Inteligência Computacional Aplicada.*, 1999.

PEDRINO, E. C. *Arquitetura Pipeline reconfigurável através de instruções geradas por programação genética para processamento morfológico de imagens digitais utilizando FPGAs*. Tese (Doutorado) — Escola de Engenharia de São Carlos, Universidade de São Paulo, 2008.

POVINELLI, R. J.; FENG, X. Improving genetic algorithms performance by hashing fitness values. *proceedings of Artificial Neural Networks in Engineering, St. Louis, Missouri*, p. 399–404, 1999.

RONALD, S. Duplicate genotypes in a genetic algorithm. In: IEEE. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*. [S.l.], 1998. p. 793–798.

SERRA, M. et al. The analysis of one-dimensional linear cellular automata and their aliasing properties. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, IEEE, v. 9, n. 7, p. 767–778, 1990.

TANOMARU, J. Motivação, fundamentos e aplicações de algoritmos genéticos. In: *II Congresso Brasileiro de Redes Neurais*. [S.l.: s.n.], 1995. p. 373–403.

TEIXEIRA, A. M. *Grupos livres e autómatos celulares*. Dissertação (Mestrado) — Faculdade de Ciências da Universidade do Porto, 1996.

VINHAL, G. S. *Implementação de um algoritmo evolutivo utilizando a representação nó-profundidade-grau no processador Nios II do FPGA*. [S.l.]: Universidade Federal de Goiás, 2013.

WANGENHEIM, A. v. *Introdução à Visão Computacional*. Universidade Federal de Santa Catarina, 2001. Disponível em: <https://www.inf.ufsc.br/~visao/morfologia.pdf>. Acesso em: 16 mar. 2016.

WEINERT, W. R. *Computação evolucionária para indução de regras de autômatos celulares multidimensionais*. Tese (Doutorado) — Universidade Tecnológica Federal do Paraná, 2010.

WILLIAMS, S. *Icarus Verilog Main Page*. 2016. Disponível em: <http://iverilog.icarus.com>. Acesso em: 16 mar. 2016.

WILLIAMS, S.; BAXTER, M. *Icarus Verilog: Open-Source Verilog More Than a Year Later*. Linux Journal, 2002. Disponível em: <http://www.linuxjournal.com/article/6001>. Acesso em: 19 mar. 2016.

YODA, I.; YAMAMOTO, K.; YAMADA, H. Automatic acquisition of hierarchical mathematical morphology procedures by genetic algorithms. *Image and Vision Computing*, Elsevier, v. 17, n. 10, p. 749–760, 1999.

YUEN, S. Y.; CHOW, C. K. A genetic algorithm that adaptively mutates and never revisits. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 13, n. 2, p. 454–472, 2009.

GLOSSÁRIO

AC – *Autômato Celular*

AG – *Algoritmo Genético*

FPGA – *Field-Programmable Gate Array*

HDL – *Hardware Description Language*

IDE – *Integrated Development Environment*

LFSR – *Linear Feedback Shift Register*

RNG – *Random Number Generator*

TTL – *Transistor-Transistor Logic*

VHDL – *VHSIC Hardware Description Language*

VHSIC – *Very High Speed Integrated Circuits*

Apendice A

Hardware EVOLUTIVO

Este apêndice descreve o funcionamento de um protótipo desenvolvido durante o projeto, que é uma simplificação de um algoritmo evolutivo. Esse protótipo foi importante, pois proporcionou algumas conclusões obtidas de forma empírica.

O *hardware* desenvolvido utiliza um processador morfológico para cada indivíduo da população, sendo possível dessa maneira medir a aptidão de todos simultaneamente. Esse modelo de implementação traz grandes vantagens para o desempenho. Porém, o consumo de recursos é muito elevado e acabou inviabilizando a síntese em FPGAs de pequeno porte.

A principal simplificação realizada nessa implementação diz respeito ao método de seleção, que somente seleciona o indivíduo com melhor aptidão, todo restante da população é descartado. Foi utilizado o conceito de elitismo, que sugere que o melhor indivíduo da população passada só é substituído caso o melhor indivíduo da população atual seja mais apto. As novas populações são geradas sempre a partir da mutação desse indivíduo, já que não existem outros indivíduos para combinar seus códigos genéticos.

A geração da população é feita totalmente em paralelo, para cada novo indivíduo é gerado um vetor de mutação utilizando um AC. Na Figura A.1 é ilustrado um diagrama que representa o funcionamento dessa implementação. A quantidade de processadores morfológicos para avaliação, o comprimento e a quantidade de indivíduos e o tamanho das imagens são parametrizáveis pelo código Verilog.

Essa implementação tem um grande problema com consumo de recursos, uma vez que para gerar um novo indivíduo por geração, é necessário acrescentar um novo processador morfológico. E para aumentar a resolução da imagem a ser trabalhada, é necessário aumentar o tamanho de todos os processadores morfológicos.

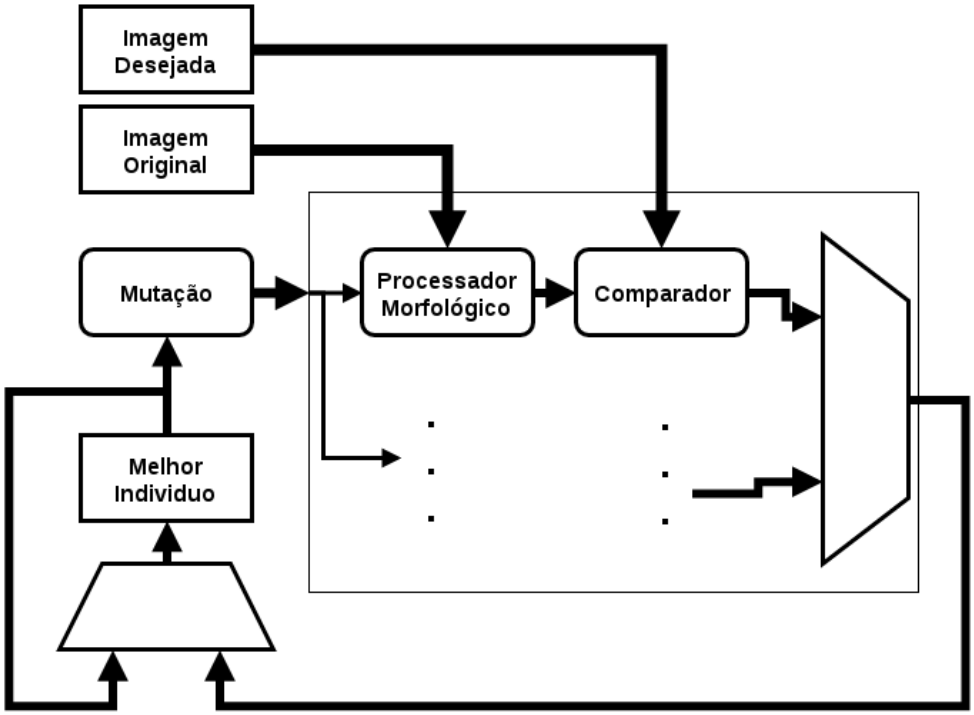

Figura A.1: *Hardware* Evolutivo.

Além dos problemas de consumo de recursos, essa implementação também propicia convergências prematuras, já que sua pressão seletiva é muito elevada. Mesmo com essa característica negativa, foi possível induzir diversas operações morfológicas para alguns pares de imagens binárias.

Apesar de não ser uma implementação ideal, o algoritmo teve a capacidade de encontrar algumas soluções. Imaginar um algoritmo genético insular com esse *hardware*, o que poderia ajudar a resolver o problema da pressão seletiva, foi uma das principais inspirações para a implementação final do projeto.