UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

GUSTAVO SIQUEIRA VINHAL

Implementação de um Algoritmo Evolutivo Utilizando a Representação Nó-Profundidade-Grau no Processador Nios II do FPGA

Goiânia
2013

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Implementação de um Algoritmo Evolutivo Utilizando a Representação Nó-Profundidade-Grau no Processador Nios II do FPGA

Autor(a): Gustavo Siqueira Vinhal

Goiânia, 19 de Agosto de 2013.

Gustavo Siqueira Vinhal – Autor

Dra. Telma Woerle de Lima Soares – Orientador

GUSTAVO SIQUEIRA VINHAL

Implementação de um Algoritmo Evolutivo Utilizando a Representação Nó-Profundidade-Grau no Processador Nios II do FPGA

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dra. Telma Woerle de Lima Soares

Goiânia
2013

GUSTAVO SIQUEIRA VINHAL

Implementação de um Algoritmo Evolutivo Utilizando a Representação Nó-Profundidade-Grau no Processador Nios II do FPGA

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 19 de Agosto de 2013, pela Banca Examinadora constituída pelos professores:

Prof. Dra. Telma Woerle de Lima Soares
Instituto de Informática – UFG
Presidente da Banca

Prof. Dr. Anderson da Silva Soares
Instituto de Informática - UFG

Prof. Dr. Paulo Henrique Ribeiro Gabriel
Faculdade de Computação - UFU

Gustavo Siqueira Vinhal

Graduou-se em Engenharia de Computação na Pontifícia Universidade Católica de Goiás. Durante sua graduação, foi monitor de Sistemas Digitais no Departamento de Computação da PUC-GO e bolsista CNPq de Iniciação Científica, tendo trabalhado com Filtro de Kalman. Atualmente, é mestrando em Ciência da Computação na Universidade Federal de Goiás.

Dedico este trabalho aos meus pais, em especial a minha mãe pelo apoio em todos os momentos difíceis.

Agradecimentos

Ao longo desses 2 anos e 6 meses de mestrado várias pessoas passaram pela minha vida. Algumas contribuíram diretamente para que eu conseguisse realizar esse sonho e por isso dedico a elas os meus agradecimentos:

- Primeiramente, a Deus por sempre me dar forças e sabedoria nessa longa jornada. Ter me iluminado nos momentos mais difíceis, me mostrando os caminhos certos e me dando forças para nunca desistir.
- Aos meus pais por todo o apoio. Em especial a minha mãe, pelos conselhos, por sempre estar ao meu lado nos momentos mais difíceis, com todo o seu carinho.
- Ao meu irmão Huggo pelo apoio.
- Aos meus primos (irmãos) Frederico e Karoline, pelo apoio nessa jornada. E aos meus primos Rosane e Delfino.
- Aos colegas de mestrado pelas discussões de ideias.
- Ao amigo Fernando Zuher por ter me acolhido em sua residência em São Carlos nos momentos que precisei.
- Ao amigo André Perina da USP - São Carlos pela paciência, ensinamentos e ajuda.
- A Marcilyanne da USP - São Carlos pela ajuda.
- Ao professor Dr. Alexandre Delbem da USP - São Carlos por ter disponibilizado equipamentos necessários para realização deste trabalho.
- Por fim, agradeço muito a uma pessoa que foi fundamental para a conclusão deste trabalho. A professora Dra. Telma Woerle de Lima Soares pela dedicação, paciência e apoio. Por estar sempre ao meu lado dando dicas e ensinamentos valiosos. É uma honra imensa trabalhar com a senhora.

A frase mais empolgante para se ouvir em ciência, aquela que proclama novas descobertas, não é "Eureca!"(Descobri!), mas "Engraçado..."

Resumo

Vinhal, Gustavo Siqueira. **Implementação de um Algoritmo Evolutivo Utilizando a Representação Nó-Profundidade-Grau no Processador Nios II do FPGA**. Goiânia, 2013. 72p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Diversos problemas pertinentes a classe NP-Difícil estão presentes no mundo real. Dentre eles pode-se citar os problemas de projeto de redes (PPRs) que envolvem distribuição de energia elétrica, tráfego de veículos, entre outros. Não existem algoritmos que forneçam uma solução exata para esses tipos de problemas com um tempo de computação aceitável. Ao longo dos anos pesquisas estão sendo desenvolvidas utilizado algoritmos evolutivos (EAs) para fornecer uma solução eficiente com tempo de computção aceitável para tais problemas. Além disso, estruturas de dados adequadas podem melhorar ainda mais o desempenho dos EAs para PPRs. A representação nó-profundidade-grau (NDDE) apresenta resultados significativos para PPRs. A aplicação de EAs em *hardware* pode melhorar o desempenho do algoritmo. Nesse sentido, este trabalho apresenta a implementação de um EA no processador Nios II de uma placa FPGA para solução do PPR da árvore geradora mínima com restrição de grau. Os resultados demonstram que a implementação de EAs em *hardware* traz resultados significativos com melhor desempenho, devido ao poder de paralelismo presente no FPGA.

Palavras–chave

Projeto de Rede, Árvore Geradora Mínima, Algoritmos Evolutivos, Nó-Profundidade-Grau, FPGA.

Abstract

Vinhal, Gustavo Siqueira. **Implementation of a Evolutionary Algorithm Utilizing the Representation Node-Depth-Degree in Nios II processor of FPGA**. Goiânia, 2013. 72p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Many relevant problems to NP-Hard class are present in the real world. Among them we can mention the problems of network design (PNDs) that involve electricity distribution, vehicle traffic, and others. There are not algorithms which provide a exact solution for these types of problems with an acceptable computation time. Over the years, research has been developed used evolutionary algorithms (EAs) to provide an efficient solution with a acceptable computation time for these problems. In addition, appropriate data structures may further improve the performance of EAs to PNDs. The node-depth-degree (NDDE) representation have show significant results for PNDs. The application of EAs in *hardware* can improve the performance of the algorithm. In this sense, this work presents the implementation of a EA in Nios II processor of a FPGA board to solving the PND minimum spanning tree with degree constraint. The results demonstrate that the implementation of EAs in *hardware* brings significant results with better performance, due to the power of parallelism present in the FPGA.

Keywords

Network Design, Minimum Spanning Tree, Evolutionary Algorithms, Node-Depth-Degree, FPGA.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

PPR	Problema de Projeto de Redes
EA	Algoritmo Evolutivo (*Evolutionary Algorithm*)
NDDE	Representação Nó-Profundidade-Grau (*Node-Depth-Degree Encoding*)
FPGA	*Field Programmable Gate Array*
MST	Árvore Geradora Mínima (*Minimum Spanning Tree*)
dc-MSTP	Problema da Árvore Geradora Mínima com Restrição de Grau (*Minimum Spanning Tree with degree constraint Problem*)
OMTP	Problema da Árvore Máxima (*One-Max Tree Problem*)
IA	Inteligência Artificial
GA	Algoritmo Genético (*Genetic Algorithm*)
ES	Estratégias Evolutivas (*Evolutionary Strategy*)
EP	Programação Evolutiva (*Evolutionary Programming*)
GP	Programação Genética (*Genetic Programming*)
CV	Vetor Característica (*Characteristic Vector*)
NetKeys	Chaves Aleatórias (*Network Random Keys*)
PAO	*Preserve Ancestor Operator*
CAO	*Change Ancestor Operator*
ASIC	*Application Specific Integrated Circuit*
CLB	*Configuration Logical Blocks*
IOB	*Input/Output Blocks*
HDL	Linguagem Descritiva de *hardware* (*Hardware Description Language*)
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
BSV	*Bluespec System Verilog*
JTAG	*Joint Test Action Group*
PC	Computador Pessoal (*Personal Computer*)
P-PAO-S	Circuito Paralelo para PAO
LFSR	*Linear Feedback Shift Register*
HP-RNP	*Hardware-Parallelized Representação Nó-Profundidade*
MEF	*Máquina de Estados Finitos*
SDRAM	*Synchronous Dynamic Random Access Memory*
LCD	*Liquid Crystal Display*
PCI	*Peripheral Component Interconnect*
USB	*Universal Serial Bus*
EDA	*Eletronic Design Automation*
SOPC	*System on a Programmable Chip*
H-GA	*Hardware Genetic Algorithm*
EHR	*Evolutionary History Recombitation*

CAPÍTULO 1

Introdução

Problemas de Projeto de Redes (PPRs) estão presentes em diversas aplicações do mundo real no campo da engenharia e da ciência. Tais aplicações envolvem problemas como distribuição de energia elétrica [18], roteamento de veículos [60], redes de computadores [48], entre outros. Em escala do mundo real, a grande quantidade de elementos (vértices) e conexões (arestas) restringe os algoritmos utilizados para esses problemas. O problema do caixeiro viajante [29] é um exemplo para o qual não existe algoritmo determinístico de complexidade polinomial para resolvê-lo quando o grafo possui grande dimensão.

Esse tipo de problema é classificado na classe NP-Difícil em que não possui algoritmo determinístico que execute em tempo polinomial para resolvê-lo. Para resolver tais problemas, utilizam-se algoritmos aproximativos, tais como metaheurísticas e heurísticas. No entanto, esses algoritmos não fornecem uma solução exata ao problema, mas uma próxima da solução ótima. Dentre essas técnicas destacam-se os Algoritmos Evolutivos (EAs, *Evolutionary Algorithms*).

EAs possuem como ponto principal a simulação da evolução natural das espécies. Dado um conjunto (população) de indivíduos (possíveis soluções), a cada iteração (geração) são aplicados operadores para que melhores indivíduos sejam encontrados. Entretanto, um EA com codificação padrão pode não ser eficiente quando aplicado a uma rede grande, ou seja, que possui vários vértices. Em geral, esses EAs aplicados a redes de grande porte demandam tempo de processamento e memória para a validação ou correção de soluções infactíveis.

Para melhorar a eficiência de um EA aplicado a um PPR com alta escala de complexidade, estruturas de dados dinâmicas (codificação) mais adequadas vêm sendo pesquisadas [15]. Tais codificações aumentam a qualidade das soluções geradas pelo EA, já que possibilita uma exploração mais adequada do espaço de busca. Além disso, o uso de codificações específicas minimiza a necessidade de processos de correção e validação das soluções.

Dentre várias codificações existentes na literatura, a codificação Nó-Profundidade-Grau (NDDE, *Node-Depth-Degree Encoding*) se destaca por possuir

complexidade $O(\sqrt{n})$ (sobre a operação de codificação e decodificação) enquanto outras requerem o tempo de $O(n)$, onde n é o número de vértices do grafo [19]. Sendo assim, uma codificação adequada para PPRs de larga escala.

Nesse sentido, a implementação de EAs que utilizem a representação NDDE em FPGA (*Field Programmable Gate Array*) proporciona resultados significativos em desempenho, que permitem a sua utilização para PPRs com milhares e até milhões de vértices [25]. Esse resultado é possível devido à propriedade intrínseca de paralelização do FPGA que adapta os processos do EA com o objetivo de reduzir o tempo computacional [25]. Além disso, a implementação em FPGA permite que o tempo computacional fique limitado por uma constante para grafos com diferentes números de vértices, o que torna uma vantagem a utilização de FPGA para o desenvolvimento do EA em relação ao *software*.

Este trabalho consiste no desenvolvimento de um EA para solucionar o problema de árvore geradora mínima com restrição de grau no processador Nios II de uma placa FPGA. O problema da árvore geradora mínima é um modelo que pode ser aplicado a diferentes PPRs. Esse problema consiste em encontrar uma subárvore (subrede) acíclica que conecte todos os vértices de um grafo (rede). O EA proposto utiliza a representação NDDE implementada em FPGA conforme apresentada em [25]. O principal objetivo com o desenvolvimento do EA proposto é analisar a capacidade de otimização do mesmo com a manutenção do tempo de processamento constante.

O trabalho está organizado como segue: o Capítulo 2 apresenta uma revisão bibliográfica dos principais PPRs encontrados na literatura. O Capítulo 3 aborda uma introdução aos algoritmos evolutivos. O Capítulo 4 ilustra as diferentes formas de representação de PPRs em EAs. No Capítulo 5 é descrita a computação reconfigurável e suas ferramentas. A arquitetura desenvolvida para a proposta do trabalho também é apresentada no Capítulo 5. O Capítulo 8 apresenta os resultados obtidos da aplicação do EA ao PPR de árvore geradora mínima com restrição de grau em FPGA. No Capítulo 9 conclui-se o trabalho.

CAPÍTULO 2

Problemas de Projetos de Redes

Diversos problemas na computação podem ser modelados como Problemas de Projetos de Redes (PPRs). Em grande parte dos casos a versão de decisão dos PPRs é classificada na classe dos problemas NP-Completos [51]. No mundo real, a solução desses problemas envolve variáveis como larga escala, resposta em tempo real e, na maioria das vezes, múltiplos critérios. A consideração dessas características aumenta a complexidade do problema no ponto de vista computacional.

Em geral, os PPRs são modelados por grafos não orientados $G = (V, E)$, onde V é o conjunto de vértices (pontos da rede) do grafo e E o conjunto de arestas (ligações da rede). A rede pode conter um conjunto de pesos W utilizado para associar um peso a cada aresta do grafo, o qual representa o custo daquela ligação.

De acordo com Johnson et al. [9], um PPR pode ser escrito da seguinte forma: dado um grafo $G = (V, E)$, uma função de pesos $L : E \rightarrow (N)$, um orçamento B e um limitante C para o valor critério ($B, C \in N$), a solução do PPR consiste em encontrar o subgrafo $H = (V, E_H)$ de G com pesos tais que $\sum_{(i,j)\in E_H} L(i, j) \leq B$ e o valor de critério $f(H) \leq C$, em que $f(H)$ representa a soma dos comprimentos de caminhos mínimos entre todos os vértices em H.

Assim, a resolução de um PPR consiste em encontrar um subgrafo G' de um grafo G que satisfaça, de forma otimizada, os critérios utilizados na avaliação da rede, sem violar as restrições do problema. As restrições do problema podem variar dependendo da rede. Por exemplo, as restrições podem ser com relação aos pesos das arestas ou ao grau dos vértices. Um exemplo de critério de avaliação da rede é a minimização do caminho entre todos os vértices [14].

Em PPRs do mundo real, além da minimização dos pesos entre os vértices da rede, outros fatores devem ser analisados [14]. Em redes de telecomunicações, de computadores e elétricas deve-se também analisar o custo, desempenho e confiança da rede. O custo da rede está associado ao custo monetário de construção e/ou manutenção da rede em funcionamento. O desempenho da rede corresponde a quantidade de tráfego e fluxo presente na mesma. A confiança da rede representa o nível de integridade ou qualidade de um caminho na rede.

Dentre os vários PPRs encontrados na literatura, pode-se citar os mais conhecidos e estudados: árvore geradora mínima com restrição de grau [33, 50, 55] e árvore máxima [54, 50, 55].

2.1 Árvore Geradora Mínima

Seja um grafo não orientado $G = (V, E)$, onde V é o conjunto de vértices, E o conjunto das arestas e, para cada aresta $(u, v) \in E$, tem-se um peso $w(u, v)$ que representa o custo para conectar u e v. O problema da Árvore Geradora Mínima (MST, *Minimum Spanning Tree*) consiste em encontrar um subconjunto acíclico $T \subseteq E$ que conecte todos os vértices em V e cujo peso total $w(T) = \sum_{(u,v)\in T} w(u, v)$ seja mínimo.

Seja M uma árvore geradora mínima. M possui duas propriedades [21]:

1. Propriedade de corte: seja e uma aresta tal que $e \in M$. e será a menor aresta de alguma poda de G.
2. Propriedade de ciclo: seja e uma aresta tal que $e \notin M$. e será a maior aresta de alguma poda de G.

Dois algoritmos clássicos são utilizados para resolver o problema da MST: o algoritmo de Kruskal [35] e o algoritmo de Prim [49]. Ambos algoritmos são gulosos, ou seja, a cada passo do algoritmo, é feita a melhor escolha no momento.

O conceito de árvore geradora pode ser expandido para floresta geradora [20]. Uma floresta geradora de G é formada pelo conjunto de árvores geradoras. Cada árvore geradora é obtida a partir de componentes conexos do grafo G. A floresta geradora mínima é obtida pelo conjunto de todas as árvores geradoras mínimas extraídas de G.

MSTs são comumente aplicadas em diversos problemas de roteamento como redes de telecomunicações, rotas de cabos elétricos, entre outros [21]. Nesses problemas as MSTs são utilizadas para determinar os melhores caminhos para as redes. Além disso, o problema da MST tem sido utilizado como simplicação de PPRs mais complexos em busca de uma solução próxima da ótima para os mesmos.

2.1.1 Árvore Geradora Mínima com Restrição de Grau

O problema da Árvore Geradora Mínima com Restrição de Grau (dc-MSTP, *Minimum Spanning Tree with degree constraint Problem*) é uma variação do problema MST, em que o grau do vértice é restringido por uma constante k [33]. Ao adicionar a restrição do grau ao problema da árvore geradora mínima, esse passa a pertencer a classe de problemas NP-Difícil [2].

O dc-MSTP pode ser definido como segue. Seja $G = (V,E)$ um grafo não orientado em que cada vértice $v_i \in V$ possui um grau $deg(v_i)$ e k é a restrição de grau do problema, com $k > 2$. Seja W a matriz de pesos, onde $w_{i,j}$ é o peso do par de vértices v_i, v_j. Seja T uma árvore geradora de G. Então, o dc-MSTP pode ser definido como

$$min_T \sum_{(i,j)\in T} w_{v_i,v_j} \tag{2-1}$$

sujeito à restrição

$$deg(v_i) \leq k, \text{para todo} \quad v_i \in T. \tag{2-2}$$

A Figura 2.1 ilustra um exemplo do problema do dc-MSTP.

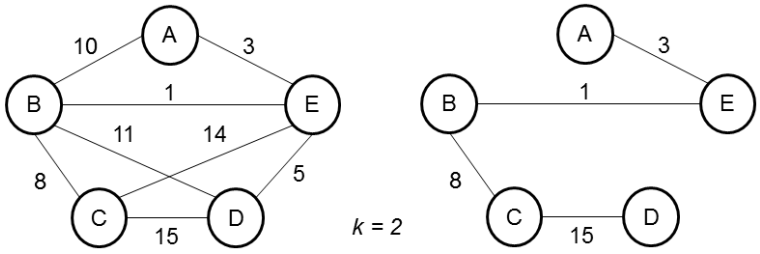


Figura 2.1: *Árvore Geradora Mínima com Restrição de Grau com k = 2*

Conforme a Figura 2.1 o grafo da esquerda representa o grafo original do dc-MSTP. O grafo da direita representa uma solução para o problema quando a restrição de grau k é igual a 2. Nesse exemplo foi aplicado um algoritmo guloso onde a cada iteração ele vai formando o novo grafo com as arestas de menor peso, sempre respeitando a restrição k.

Na literatura é possível encontrar várias heurísticas para solução do dc-MSTP, tais como algoritmos evolutivos e *Simulated Annealing*. A seguir são descritas algumas dessas propostas de solução para esse problema.

Krishnamoorthy et al. utilizam o método *Simulated Annealing* para resolver o dc-MSTP e demonstram que o seu uso traz resultados medianos [34]. Nesse trabalho, os autores utilizam a técnica de *cortar e colar* para geração de novas árvores. Essa técnica fornece uma nova árvore geradora podando uma árvore T_1 em T. Para isso, sorteia-se de forma aleatória um vértice i e remove a aresta que o conecta ao seu pai j. A remoção da aresta $(i, j) \in T$ gera duas árvores. Então, essas árvores são reconectadas escolhendo outro nó aleatório j para conectar i. Para implementar essa técnica Krishnamoorthy et al. utiliza a permutação dos vértices, chamada de B. B é utilizado para evitar a decodificação de um número a cada iteração. A implementação do *Simulated Annealing* em um espaço de

busca que oferece apenas soluções factíveis reduz o número de escolhas para reconexão da árvore podada T_1 devido à restrição de grau.

Narula et al. [40] utiliza uma modificação do algoritmo de Prim na resolução do dc-MSTP. Prim [49] utiliza dois princípios para gerar uma MST. O primeiro deles diz que um nó isolado i pode ser conectado ao vizinho j mais próximo de i. O segundo princípio diz que qualquer subconjunto $V_S \subset V$, tal que V é o conjunto de vértices, pode ser conectado ao vizinho mais próximo j do subconjunto. Narula et al. modificam esse segundo princípio para obter dc-MSTs factíveis. Essa modificação consiste em conectar o subconjunto ao vizinho mais próximo pela aresta de menor valor, cuja conexão não viole a restrição de grau.

A implementação do algoritmo modificado de Prim gera uma melhor solução inicial factível em menos tempo, além de manter a factibilidade a cada passo do algoritmo. Porém, Narula et al. demonstram que, em relação ao tempo médio de computação para árvores de 30, 50 e 100 vértices com restrições de grau de 3 e 4, a modificação do algoritmo de Prim é pior do que outros algoritmos, como por exemplo, *branch and bound*.

Além disso, dc-MSTPs tem sido utilizados para avaliar o desempenho de EAs quando aplicados em PPRs [62, 17, 34]. Em [14] é utilizada uma nova forma de representação chamada NDDE para solução de dc-MSTPs. Essa representação faz uso de dois operadores baseados no princípio de mutação. Os resultados obtidos para diversos dc-MSTs são satisfatórios, principalmente para redes de alta escala, com baixo custo computacional.

2.2 Árvore Máxima

O problema da árvore máxima (OMTP, *One-Max Tree Problem*) é amplamente utilizado para verificar o desempenho de algoritmos de otimização quando aplicados em problemas que envolvem projeto topológico de redes [54].

Nesse problema, uma árvore geradora é escolhida como sendo uma solução ótima e o objetivo consiste em buscar por essa árvore. A estrutura dessa árvore pode ser estrela, lista ou qualquer outra estrutura de árvore que contenha n vértices [56].

Tendo uma árvore ótima T_{opt} avalia-se o quão bom outra árvore T_b representa para a solução utilizando a distância $d_{opt,b}$ entre elas, representada por

$$d_{opt,b} = \frac{1}{2} \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} |l_{ij}^{opt} - l_{ij}^{b}|. \tag{2-3}$$

Na Equação 2-3, l_{ij}^{opt} é 1 se existe uma aresta que conecta o vértice i ao vértice j na árvore T_{opt} e 0 caso contrário, n representa o número de vértices. A distância entre as

duas árvores é baseada nos conceitos de distância de Hamming e $d_{opt,b} \in \{0, 1, ..., n-2\}$ [56].

Pesquisas com EAs têm utilizado a OMTP com o objetivo de avaliar diferentes representações e suas vantagens [32, 57]. Em [54] a OMTP é utilizada para avaliar o desempenho de EAs com relação a redundância, localidade e escalabilidade das representações.

Schindler et al. utilizam Estratégias Evolutivas com Chaves Aleatórias como forma de representação para resolver o OMTP [57]. É mostrado que a utilização das Chaves Aleatórias com Estratégias Evolutivas resolvem de forma satisfatória o OMTP em relação à GAs.

CAPÍTULO 3

Computação Evolutiva

A computação evolutiva surgiu dos conceitos da Teoria da Evolução das Espécies. Desde então, modelos e funcionalidades naturais foram desenvolvidos para sistemas computacionais com o objetivo de otimizar o desempenho dos mesmos [47].

Devido a grande diversidade de problemas de otimização, a computação evolutiva obteve grande destaque no meio científico. Por meio da computação evolutiva é possível obter resultados aceitáveis em problemas com alta complexidade de tempo computacional, como os da classe NP-Difícil. Dentre os algoritmos que utilizam o paradigma de computação evolutiva destacam-se os Algoritmos Genéticos [30], Estratégias Evolutivas [58, 52], Programação Evolutiva [23] e Programação Genética [59], que serão abordados a seguir.

3.1 Algoritmos Evolutivos

Os Algoritmos Evolutivos (EAs, *Evolutionary Algorithm*) representam a estrutura geral do paradigma de computação evolutiva. Esses algoritmos não fornecem uma solução exata, mas uma solução próxima a ótima com complexidade de tempo relativamente menor [10].

Os EAs advêm da área de Inteligência Artificial (IA), como métodos de pesquisa em que se objetiva conseguir uma boa solução para problemas que utilizam um grande espaço de busca. Apesar do seu surgimento na área de IA, EAs são aplicados em diversos campos da ciência tais como a biologia, otimização numérica e engenharia.

Baseados em mecanismos evolutivos naturais, os EAs são métodos de procura estocástica e otimização que utilizam informações codificadas em um formato que imita a informação genética e opera de acordo com uma lógica que permite que tais unidades codificadas troquem informações entre si. Como correlação à evolução natural é necessário alguns conceitos, como cromossomo, genótipo, fenótipo e seleção natural.

O cromossomo é a representação computacional das possíveis soluções do EA. Como na biologia, a sequência de genes que compõe o cromossomo é denominada genótipo da solução. O fenótipo representa a solução no seu domínio real, ou seja, é

a decodificação do cromossomo com base nas informações do genótipo. Cada posição dentro do cromossomo é dito gene e representa uma característica da solução do problema [31].

Os EAs se comportam de forma semelhante ao processo de seleção natural que ocorre na natureza [53]. A seleção é realizada através de uma função de avaliação (também chamada de função *fitness* ou aptidão) que atribui um valor a cada indivíduo da população. Quanto melhor esse valor, maior a chance do indivíduo ser selecionado para o cruzamento e transferir as melhores características para a próxima geração. Esse valor também é utilizado na determinação dos indivíduos que irão sobreviver para a próxima geração.

Três componentes básicos podem ser identificados nos EAs:

- **Indivíduos:** representam as possíveis soluções do problema;
- **Seleção:** mecanismo responsável pela sobrevivência dos indivíduos mais aptos, ou seja, os que representam as melhores soluções para o problema;
- **Operadores:** mecanismos que exploram o espaço de busca. Em geral, existem dois principais operadores: mutação e recombinação.

A cada indivíduo são aplicados operadores genéticos (mutação e recombinação) de forma a modificá-los para encontrar uma melhor solução ao problema. Uma vez modificados, os indivíduos são avaliados em relação à sua qualidade como solução ao problema e, os mais aptos, são selecionados para formar novos indivíduos.

Os EAs modelam um processo de aprendizagem coletivo, em meio de uma população de indivíduos, em que cada um representa um ponto no espaço de soluções potenciais de um dado problema, e, além disso, um depósito temporal de conhecimento sobre as leis do ambiente [36].

Nas subseções a seguir são apresentados, detalhadamente, os componentes necessários para o funcionamento do EA.

3.1.1 Codificação

O primeiro passo para a construção de um algoritmo evolutivo é a definição de como os indivíduos serão representados computacionalmente. Cada indivíduo pode ter diversas representações que variam de acordo com o problema a ser tratado. A codificação é o processo que consiste em definir como os cromossomos serão representados.

O sucesso do algoritmo depende diretamente da escolha da codificação: uma codificação inadequada pode dificultar o processo de evolução do EA. No entanto, a escolha de uma codificação adequada pode evitar testes de viabilidade de cada solução gerada, por exemplo.

Ao final do processo de execução do EA, é necessário o processo de decodificação para utilizar a solução no espaço real do problema. Existem diversas formas de representação, tais como binária, inteira e real [24].

Codificação Binária A mais simples e utilizada em EAs. Nesse tipo de representação, cada gene do cromossomo assume os valores 1 ou 0. A Figura 3.1 apresenta um exemplo de codificação binária.

Figura 3.1: *Codificação binária.*

Codificação Inteira Os genes são representados por valores inteiros. A Figura 3.2 mostra essa forma de codificação.

Figura 3.2: *Codificação inteira.*

Codificação Real Nessa representação cada gene assume um valor real. Esse tipo de codificação é mais utilizado quando os genes são distribuídos em um intervalo de valores contínuos ao invés de valores discretos. A Figura 3.3 ilustra esse tipo de representação.

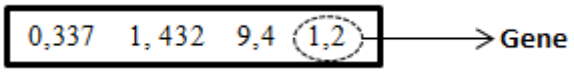


Figura 3.3: *Codificação real.*

3.1.2 Avaliação

A avaliação é o processo que determina o quão adequado um indivíduo é como solução do problema [38]. Essa forma de avaliação é realizada aplicando a cada indivíduo uma função de avaliação e obtendo o valor de seu *fitness*. Um indivíduo é selecionado para os operadores genéticos (cruzamento e mutação) de acordo com o valor de seu *fitness* [31].

3.1.3 Seleção

O processo de seleção elege os melhores indivíduos para o processo evolutivo do EA. Esse processo tem como base o valor de *fitness* de cada cromossomo. Na etapa de seleção podem ocorrer alguns problemas:

- **Convergência prematura:** quando existe um indivíduo muito melhor que todos os outros, ele sempre será selecionado fazendo com que haja uma convergência prematura para um ótimo local (que não necessariamente representa o ótimo global);
- **Pressão de seleção:** quando os valores de *fitness* são muito próximos uns dos outros não há uma pressão de seleção, fazendo com que a seleção fique uniformemente aleatória e valores de *fitness* ligeiramente melhores não influenciem na escolha de indivíduos.

Existem algumas técnicas de seleção como torneio, *ranking*, elitista e proporcional [31].

Seleção por Torneio Na seleção por torneio um subconjunto de indivíduos da população é selecionado de forma aleatória. Os indivíduos desse subconjunto são comparados em relação ao *fitness*. O indivíduo que possuir o maior valor de *fitness* vence o torneio e é selecionado [27, 63]. O tamanho do subconjunto é determinado por uma constante k, onde k é a quantidade de indivíduos selecionados para comparação.

Um ponto importante é o tamanho do subconjunto a ser escolhido. Um subconjunto muito grande aumenta a probabilidade de indivíduos com melhores *fitness* serem selecionados, porém aumenta o tempo de computação.

A Figura 3.4 mostra um exemplo de seleção por torneio. A caixa da esquerda representa o subconjunto de n indivíduos, sendo $1 < n \leq$ tamanho população, extraídos da população. Desse subconjunto são selecionados pares de indivíduos para disputarem entre si. Aquele que tiver o maior valor de *fitness* é selecionado.

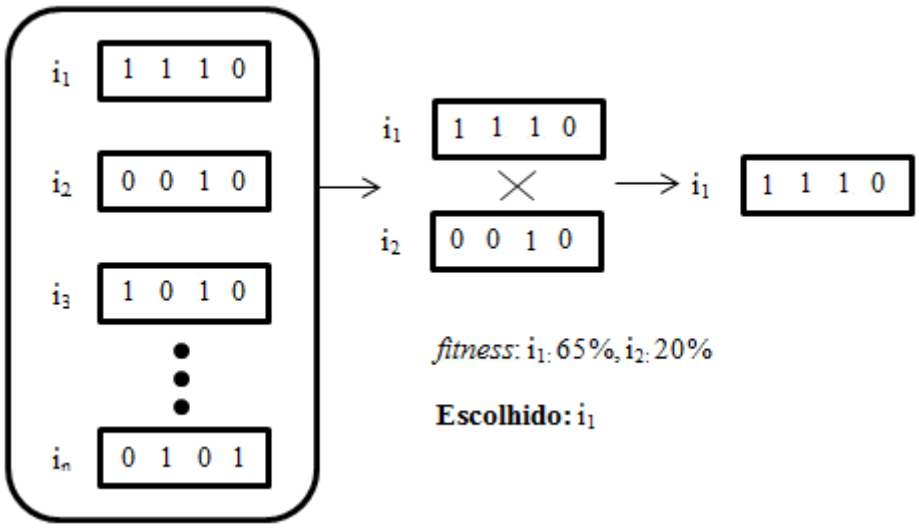


Figura 3.4: *Seleção por torneio.*

Seleção por *Ranking* A seleção por *ranking* trabalha criando um *rank* com os indivíduos de acordo com o valor de *fitness* de cada um. O *rank* é ordenando de forma decrescente. São escolhidos x indivíduos para gerarem a nova população. Os indivíduos que ocuparem a melhor posição no *rank* possuem a maior probabilidade de serem escolhidos.

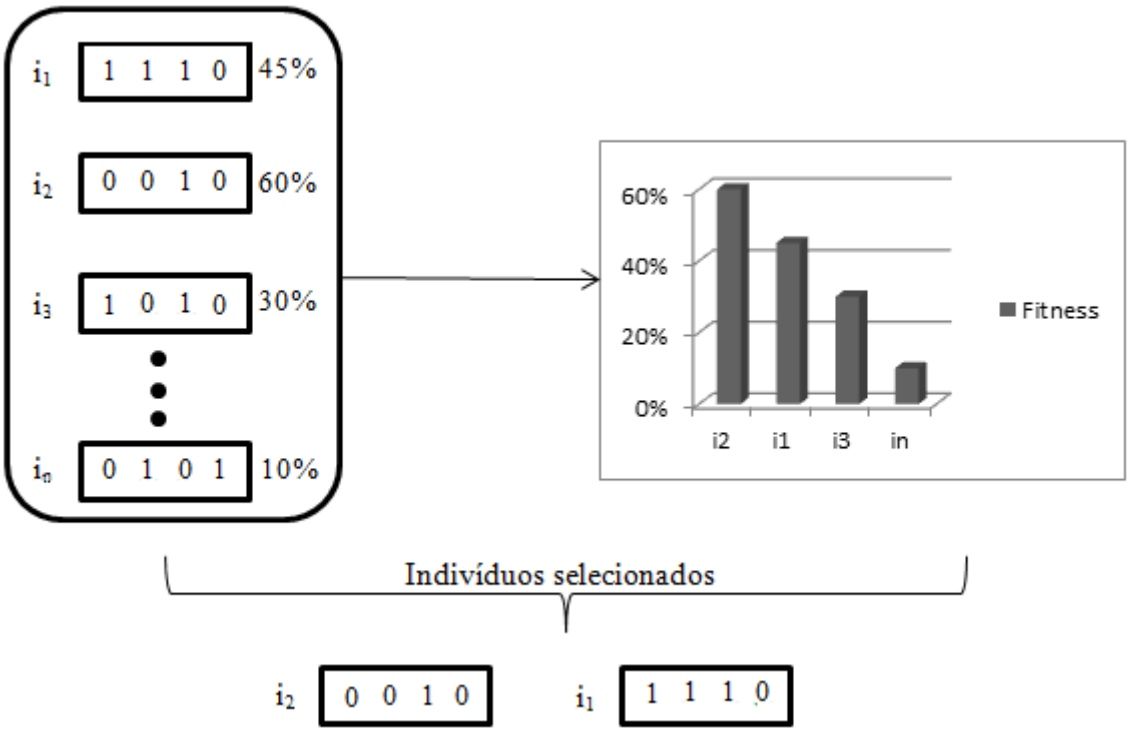


Figura 3.5: *Seleção por ranking.*

Na Figura 3.5 uma população de n indivíduos, $0 < n \leq$ tamanho população, é dada (caixa da esquerda). Os indivíduos são ordenados em um *rank* em ordem decrescente com base no valor do *fitness*. Então são escolhidos 2 indivíduos ($x = 2$) que ocupam a melhor posição no *rank*, ou seja, que possuem um maior valor de *fitness*.

Seleção Elitista Nesse tipo de seleção é escolhido um subconjunto de indivíduos com os melhores valores de *fitness* para serem inseridos diretamente na nova população, sem a necessidade de passar pelos operadores de cruzamento e mutação.

Essa seleção é útil quando se quer manter certos indivíduos que possuem altos valores de *fitness* e que representam boas soluções ao problema, mas que são eliminados no processo de cruzamento.

Seleção Proporcional (ou Roleta) Na seleção proporcional (também chamada de seleção por roleta) cada indivíduo possui a probabilidade de ser selecionado proporcional-

mente ao seu *fitness* [3]. Essa probabilidade é expressa por,

$$p(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)}. \tag{3-1}$$

De acordo com a Equação 3-1, a probabilidade $p(i)$ de um indivíduo i ser selecionado é proporcional ao seu fitness, $f(i)$.

Essa seleção pode ser vista como uma roleta em que cada fatia dela é proporcional ao *fitness* do indivíduo. Assim, quanto maior o *fitness* do indivíduo maior a probabilidade dele ser escolhido.

De acordo com a Figura 3.6, uma população de n indivíduos, sendo $0 < n \leq$ tamanho população, é dada (caixa da esquerda). O *fitness* de cada indivíduo determina o tamanho de sua fatia na roleta. Quanto maior o *fitness* maior a fatia na roleta e mais alta é a probabilidade do indivíduo ser escolhido.

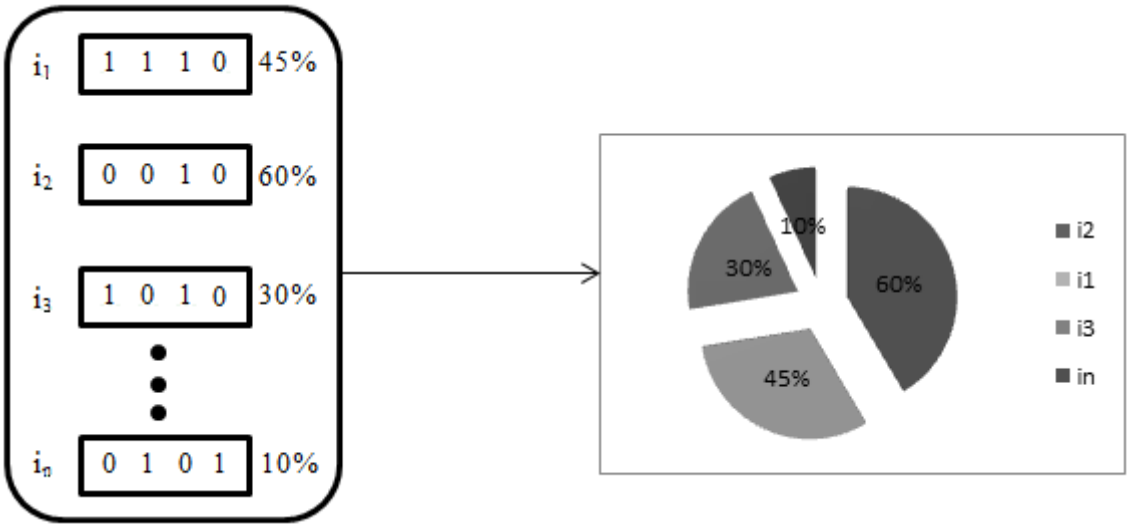


Figura 3.6: *Seleção proporcional.*

3.1.4 Cruzamento

O operador de cruzamento, também chamado de *crossover*, visa realizar a recombinação de dois indivíduos, denominados pais, para gerar dois novos indivíduos, denominados filhos. Os filhos gerados são utilizados para compor a nova população de indivíduos, ou seja, um novo conjunto de possíveis soluções para o problema. Uma vez gerada, essa nova população passa por todo o processo do EA, o processo é repetido de forma iterativa.

No cruzamento, os filhos são formados por parte dos genes de um pai e parte de outro. Espera-se que os filhos herdem as melhores características dos pais, assim propaga-se essas características ao longo do processo [26].

Há uma probabilidade para que o cruzamento ocorra. Usualmente essa probabilidade é alta (cerca de 60% a 90%, de acordo com o problema em questão) [43], tendo em vista que esse é o principal operador responsável pelo surgimento de novos indivíduos. Os principais tipos de cruzamento são de um ponto, multiponto e uniforme.

Cruzamento de Um Ponto Nesse tipo de cruzamento, um ponto de corte é escolhido aleatoriamente em um intervalo de $[0, l-1]$, sendo l o tamanho do indivíduo. Uma vez definido o ponto de corte, os segmentos são utilizados para formar os novos filhos. O ponto de corte é a posição onde o indivíduo pai será divido para formar os filhos.

A Figura 3.7 apresenta um exemplo de cruzamento de um ponto, com o ponto de corte igual a 2.

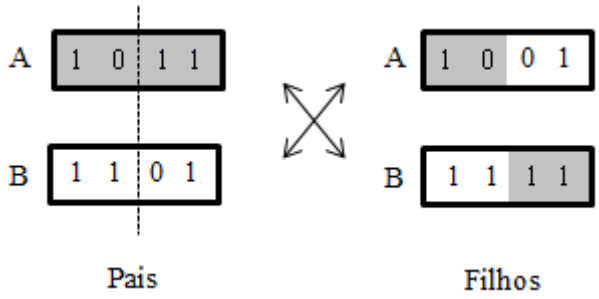


Figura 3.7: *Cruzamento de um ponto.*

Conforme a Figura 3.7, dado um ponto de corte igual a 2, os pais são divididos e os genes são distribuídos entre os filhos. O filho A será composto pelos dois primeiros genes do pai A e pelos dois últimos genes do pai B. O filho B será composto pelos dois primeiros genes do pai B e pelos dois últimos genes do pai A.

Cruzamento Multiponto O cruzamento multiponto é igual ao de um ponto, com a diferença de que agora vários pontos de corte são selecionados.

A Figura 3.8 ilustra um exemplo de um cruzamento multiponto.

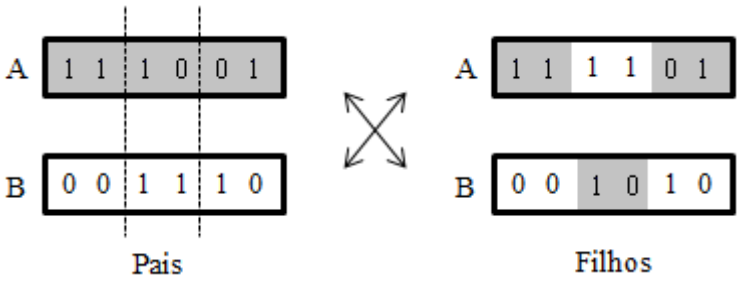


Figura 3.8: *Cruzamento multiponto.*

Conforme a Figura 3.8, dois pontos de corte de tamanho 2 são selecionados, os pais são divididos e os genes são distribuídos entre os filhos. O filho A será composto

pelos dois primeiros e dois últimos genes do pai A e pelos dois genes intermediários do pai B. O filho B será composto pelos dois primeiros e dois últimos genes do pai B e pelos dois genes intermediários do pai A.

Cruzamento Uniforme No cruzamento uniforme os genes dos filhos são herdados dos pais a partir de um sorteio.

Neste tipo de cruzamento cria-se um vetor v de tamanho l, sendo l o tamanho do indivíduo, de números aleatórios com distribuição entre 0 e 1. Cada posição de v corresponde a um gene. Para gerar o filho A, verifica-se o valor do número aleatório em v. Se for menor que um dado valor i (geralmente 0,5) o gene do filho (para uma dada posição p) será herdado do pai A. Caso contrário, será herdado do pai B. Para o filho B utiliza-se o inverso: caso o número aleatório for menor que i o gene será herdado do pai B, caso contrário do pai A.

A Figura 3.9 apresenta um exemplo do cruzamento uniforme. Seja v um vetor de números aleatórios de tamanho $l = 6$, tal que $v = [0,23; 0,58; 0,79; 0,09; 0,46; 0,64]$. Os genes dos filhos gerados são determinados com a comparação dos números aleatórios do vetor v com o valor de $i = 0,5$.

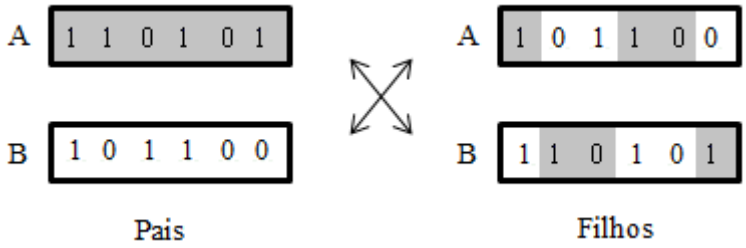


Figura 3.9: *Cruzamento uniforme.*

De acordo com a Figura 3.9 o filho A será composto pelos genes 1, 4 e 5 do pai A, pois os valores das respectivas posições no vetor v são menores que i. Os genes restantes são formados pelos genes do pai B, pois os valores são maiores que i. O filho B é formado pelos genes 2, 3 e 6 do pai A dado que os valores de v são maiores que i. O restante dos genes do filho B é composto pelos genes do pai B onde os valores em v são menores que i.

3.1.5 Mutação

Com o intuito de diversificar os indivíduos da população, o GA possui o operador de mutação em seu processo [31, 23, 61]. Através dele altera-se um ou mais genes de cada cromossomo, após a etapa de cruzamento, com probabilidade igual a p_m. A alteração de genes dos cromossomos pode levar a um indivíduo pior que o original. Por isso, geralmente, a taxa de mutação é ajustada entre o intervalo de $1\% \leq p_m \leq 5\%$.

O operador de mutação auxilia no processo evolutivo do algoritmo fazendo com que seja feito uma maior exploração no espaço de soluções. Após algumas gerações, o processo de cruzamento tende a deixar a população mais homogênea, ou seja, com indivíduos semelhantes entre si. Este fato pode levar a população a um ótimo local. Com a mutação é possível realizar alterações nos cromossomos de tal forma a direcionar a pesquisa para outros locais no espaço de soluções [11].

A Figura 3.10 mostra um exemplo de mutação.

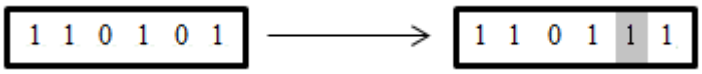


Figura 3.10: *Exemplo de mutação.*

De acordo com a Figura 3.10, o gene na posição 5 foi trocado mostrando um exemplo de mutação. Os indivíduos estão codificados na forma binária, entretanto a mutação varia de acordo com a representação utilizada [4].

Mutação Para Representações Binárias Nesse tipo de representação, a mutação ocorre apenas alterando os bits de 0 para 1 e vice-versa.

Mutação Para Representações Inteiras Para esse tipo de representação existem dois tipos de mutação: aleatória e deformação.

- **Aleatória:** os genes são trocados aleatoriamente por outros que estão dentro de um conjunto de valores permitidos;
- **Deformação:** a cada gene a ser modificado é somado um pequeno valor (positivo ou negativo).

Mutação Para Representações Reais A mutação de cromossomos representados de forma real é realizada considerando os valores de forma contínua, ignorando a parte discreta deles. Duas formas de mutação existem:

- **Uniforme:** os genes são trocados aleatoriamente de forma uniforme. Essa forma é semelhante à mutação aleatória para representações inteiras;
- **Não Uniforme:** assim como a mutação por deformação para representações inteiras, a mutação não uniforme altera cada gene somando um pequeno valor a ele. Esse valor é obtido de uma distribuição Gaussiana.

3.2 Algoritmos Genéticos

Os princípios que regem os Algoritmos Genéticos (GAs, *Genetic Algorithms*) surgiram entre as décadas de 1950 e 1960 [30]. Nesse período, cientistas da computação

estudaram sistemas evolutivos com o intuito de utilizá-los como ferramentas de otimização para problemas na engenharia. Os sistemas desenvolvidos tinham como objetivo gerar uma população de candidatos para solução de um dado problema.

Apesar de terem sido propostos por John Holland na década de 60, os GAs apenas foram desenvolvidos em meados de 1970 por seus alunos na Universidade de Michigan. Para Holland, o principal objetivo não era o desenvolvimento de algoritmos que solucionassem problemas específicos, mas sim dedicar-se ao estudo formal do fenômeno da evolução e desenvolver métodos de transportá-lo aos sistemas de computação. Holland acreditava que se poderia implementar de forma algorítmica todas as particularidades da evolução natural, alcançando uma versão computacional de todo o processo de evolução.

Atualmente, os GAs são aplicados com sucesso em diversas áreas, em especial na área de otimização ou em problemas relacionados com processos adaptativos. Eles são técnicas de heurísticas de otimização global, ou seja, que utilizam heurísticas para tentar encontrar um ponto de máximo global (melhor solução) em um espaço de busca.

Baseados em modelos computacionais de pesquisa probabilística que se inspiram no processo de seleção natural e na genética, os GAs possuem os componentes básicos:

- **População de indivíduos:** representam potenciais soluções para o problema em questão;
- **Avaliação:** mecanismo que avalia o quão boa está a solução do esperado;
- **Seleção:** escolhe as soluções mais aptas para compor a nova população;
- **Operadores:** responsáveis por pesquisar/gerar novas soluções para o problema. Os mais comuns são:

  Mutação: altera as informações de um indivíduo com o intuito de aumentar a diversidade da população.

  Cruzamento: as informações de dois indivíduos (pais) são recombinadas com o objetivo de gerar novos indivíduos (filhos) na população. A ideia é que os filhos sejam melhores que os pais se herdarem as melhores características de cada um.

O processo de funcionamento do GA é mostrado de acordo com o fluxograma da Figura 3.11.

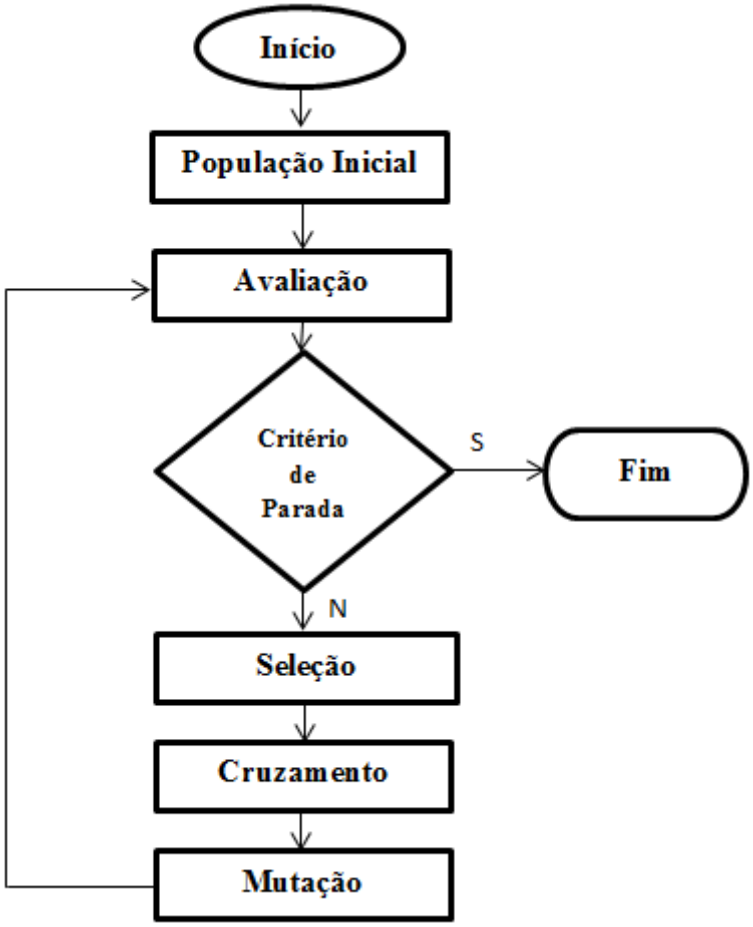


Figura 3.11: *Fluxograma Algoritmo Genético.*

De acordo com a Figura 3.11, os indivíduos (soluções) da população inicial são avaliados e se o critério de parada (que é diferenciado de acordo com o problema em questão) for atingido, o algoritmo finaliza. Caso contrário, o algoritmo segue realizando a seleção dos melhores indivíduos para os operadores evolutivos: cruzamento e mutação. O cruzamento é realizado através de mecanismos baseados na genética, assim os novos indivíduos são compostos a partir da recombinação dos progenitores e herdam algumas das suas características. Objetivando o aparecimento de novas características e soluções melhores, a cada geração é aplicado sobre os indivíduos o operador de mutação. O GA tem como base o princípio da sobrevivência dos melhores, ou seja, os indivíduos competem entre si pela sobrevivência. Assim, os mais aptos possuem alta probabilidade de serem escolhidos para compor a nova população e com isso são selecionados e substituem a população anterior, fazendo parte do novo conjunto de possíveis soluções. Após a substituição ser feita o processo iterativo do algoritmo é novamente realizado.

A busca de uma solução no espaço de soluções no GA ocorre de forma direcionada, sendo definida através da seleção, apesar de que a determinação do conjunto de pontos a serem percorridos é feito de forma aleatória. Ou seja, o GA explora informações históricas para encontrar novos pontos e assim bons desempenhos.

Os GAs possuem algumas vantagens em relação a outros métodos de otimização, tais como:

- Permite avaliação paralela das possíveis soluções uma vez que trabalha com toda população e não apenas com um único indivíduo;
- Facilmente pode ser utilizado com outras heurísticas;
- A presença de mínimos locais não reduz a eficiência do algoritmo, devido a avaliação de todas as potenciais soluções.

Um ponto importante é o desempenho dos GAs. Ele é diretamente influenciado por alguns parâmetros, os quais devem ser observados e adaptados de acordo com o problema e disponibilidade de recursos [38]:

- **Número de gerações:** um pequeno número de gerações causa uma diminuição no grau de evolução dos indivíduos devido a redução no espaço de busca. Um número muito alto de gerações exige maior tempo de processamento.
- **Tamanho da população:** uma maior população gera maior diversidade ampliando o espaço de busca, contudo necessitará de mais esforço computacional. Uma população pequena diminui o espaço de busca diminuindo a possibilidade de cruzamentos.
- **Cruzamento:** altas taxas de cruzamento favorece a evolução das gerações, porém pode levar a perda de atributos importantes em alguns indivíduos.
- **Mutação:** altas taxas de mutação podem destruir algumas informações prejudicando a evolução.

Um problema do GA é a convergência prematura. Essa situação ocorre quando, em poucas gerações, o GA converge para um ponto excelente (ótimo local), mas o qual não representa o ponto de ótimo global.

3.3 Estratégias Evolutivas

Esse tipo de algoritmo evolutivo foi desenvolvido por Rechenberg, Schwefel e Bienert na Universidade Técnica de Berlim por volta do ano de 1964 [58, 52]. Incialmente, Rechenberg desenvolveu as Estratégias Evolutivas (ESs, *Evolutionary Strategy*) com o objetivo de resolver problemas na área de otimização com base nas estruturas e processos que ocorrem na natureza, problemas focados principalmente em mecânicas dos fluídos. Schwefel, mais tarde, desenvolveu novos esquemas evolutivos com base nesses mesmos princípios.

As ESs podem ser descritas como um GA que utiliza como forma de codificação a representação real e que possui um operador de mutação que é baseado em uma

distribuição normal. Apesar de parecido com o GA, as ESs não foram desenvolvidas em conjunto com aquela técnica.

O operador de mutação usado na ES tem como base uma distribuição normal padrão, com média zero e desvio padrão $\sigma, N(0,\sigma)$. Se mantido σ em um valor constante, é provado que as ESs convergirão para uma solução ótima. Porém, é indefinido o tempo que levará para que isso ocorra. Rechenberg criou uma regra chamada *Regra de 1/5 de sucesso* para atualizar o valor de σ, fazendo com que haja convergência rápida para uma solução ótima. De acordo com essa regra, 1/5 dos descendentes gerados são melhores que os progenitores. Se o índice de melhora for menor do que 1/5 significa que o máximo local está próximo e deve-se diminuir o valor de σ. Se o índice de melhora for maior que 1/5 significa que o máximo local está longe e deve-se aumentar o valor de σ, aumentando assim a busca no espaço de soluções.

Uma ES pode utilizar dois tipos diferentes de seleção, uma representada pelo símbolo "+"e outra pelo símbolo ",". A primeira, representada por $(\alpha+\beta)-ES$, determina que em uma geração qualquer exista uma população de α progenitores que geram β descendentes por mutação. Todos os indivíduos, tanto os progenitores quanto os descendentes, são avaliados e ordenados para que sejam selecionados novos progenitores para formarem uma nova população. A segunda abordagem, representada por $(\alpha,\beta)-ES$, é parecida com a anterior, com a diferença que agora somente os descendentes gerados são selecionados para serem os novos progenitores.

3.4 Programação Evolutiva

A Programação Evolutiva (EP, *Evolutionary Programming*) foi desenvolvida por Lawrence J. Fogel no ano de 1960 [23]. Fogel acreditava que as abordagens da inteligência artificial daquela época eram limitadas pelo fato de modelar os humanos ao invés da particularidade da evolução das espécies. Nas décadas de 1980 e 1990 grandes avanços foram feitos de forma a otimizar os mecanismos da EP, com isso, diversas aplicações de EP podem ser encontradas tais como automação de controles, otimização de redes neurais, otimização de rotas, entre outros.

A EP possui características comuns de um algoritmo evolutivo: inicialização, avaliação, mutação e seleção. Um algoritmo de EP inicia seu funcionamento com um conjunto de soluções gerado de forma aleatória ou por meio de heurística apropriada para o problema. Ao decorrer do processo, o tamanho do conjunto permanece constante e as soluções são avaliadas por uma função de avaliação específica. Uma das diferenças entre um EP e outros algoritmos evolutivos é a falta do operador de cruzamento. Neste caso cada indivíduo gera descendentes que são modificados através do operador de mutação. O operador de mutação permite que todas as variáveis sejam mudadas ao mesmo tempo.

Outra característica que a difere de outros EAs é a abordagem top-down, em que os ótimos locais nem sempre levam a um ótimo global.

3.5 Programação Genética

A Programação Genética (GP, *Genetic Programming*) consiste basicamente em tentar evoluir programas de forma a resolver um problema em questão, ou seja, adaptar um programa de maneira automática para resolver um problema específico [59].

A GP é semelhante aos GAs, em que é necessário encontrar uma forma de codificação do programa e aplicar a operação de cruzamento e mutação a fim de encontrar a melhor solução para o problema. Como forma de codificação, geralmente, os indivíduos podem ser representados como uma linguagem. Essa forma pode ser útil para modelagem dos operadores e da função de avaliação. Há uma evolução de programas ou funções ao invés de indivíduos e os mesmos são avaliados através da execução de cada programa. Em seguida é verificado o quão bom a saída do programa representa a saída desejada. O espaço de busca na GP é composto por diversos programas que solucionam o problema, dentre os quais existe um programa que produz o melhor resultado.

Em [39] a GP é definida como sendo uma combinação de algoritmos genéticos com estruturas de dados. Em outras palavras, a GP consiste em aplicar GA em estruturas de dados que representam programas de forma a resolver problemas específicos.

Uma vantagem da GP é que, por se tratar de um procedimento *automático*, o algoritmo não se prende a crenças e axiomas existentes, o que é um problema natural do ser humano, assim encontrando soluções que nunca seriam imaginadas por cientistas e engenheiros.

Assim como em GA, a GP gera uma população inicial de indivíduos em que cada indivíduo representa um programa para resolver um problema em específico. Esses programas, geralmente, estão representados por meio de uma estrutura em forma de árvore. De forma aleatória, cada programa é executado e o resultado de cada execução é utilizado pela função de avaliação. A nova população é composta por cópias de alguns programas selecionados e de programas gerados pela aplicação de operadores genéticos em programas já existentes, os quais foram selecionados com base na aptidão de cada um. Novamente todos os indivíduos são avaliados e o ciclo se repete até que o critério de parada seja satisfeito.

CAPÍTULO 4

Representações de Algoritmos Evolutivos Aplicados em Projetos de Redes

A escolha da representação é importante para o desempenho dos algoritmos evolutivos (EAs). A escolha de uma representação inadequada pode afetar o tempo de convergência do algoritmo, além de tornar o processo similar aos métodos de busca aleatória [54].

Problemas de projeto de redes (PPRs) são, em geral, representados por árvores geradoras de um grafo. A codificação da árvore por meio de uma *string* para o PPR resulta em uma grande diferença entre o genótipo e a estrutura da árvore no fenótipo. Essa diferença pode ser vista no resultado que representa uma rede inviável na prática. Assim, representações convencionais em EAs para PPRs não tem gerado resultados satisfatórios, uma vez que o tempo de computação para gerar redes adequadas é alto para grafos de larga escala [14]. Dessa forma, diversas pesquisas vêm sendo realizadas com o objetivo de encontrar representações mais adequadas para EAs aplicados a PPRs [54].

Além da escolha da representação, é necessário realizar uma avaliação das codificações com o objetivo de identificar se a representação é adequada ou não ao problema. Alguns critérios de avaliação foram propostos por Palmer [44] e, posteriormente, complementados por Raidl e Julstrom [51]. Esses critérios são:

1. Espaço: as representações devem ocupar o mínimo de memória necessário;
2. Tempo: o processo de avaliação (em PPRs pode ser incluído o processo de decodificação), recombinação e mutação das soluções deve possuir complexidade de tempo baixa;
3. Factibilidade: todas as representações (incluindo as obtidas por recombinação e mutação) devem ser factíveis, ou seja, devem codificar apenas árvores. Métodos para trabalhar representações não factíveis devem ser especificados;
4. Cobertura: a representação deve ser capaz de representar todas as árvores possíveis presentes no espaço de busca;

5. Tendência: todas as soluções devem ser igualmente prováveis de serem representadas. Só é favorável uma representação possuir uma tendência maior de codificar certos tipos de soluções se essas soluções estiverem próximas do ótimo global;
6. Localidade: pequenas alterações na representação devem causar pequenas alterações na respectiva árvore representada, ou seja, devem possuir alta localidade. Baixa localidade pode direcionar o EA a um processo de busca aleatória;
7. Hereditariedade: os operadores de recombinação devem manter a maioria das arestas dos pais nos filhos. O ideal seria se todas as arestas fossem preservadas nas futuras gerações, ou seja, todas as arestas dos pais estivessem presentes nos filhos;
8. Restrições: restrições do problema podem ser facilmente incluídas, sem causar grandes modificações;
9. Hibridismo: operadores de representação devem permitir a inserção de heurísticas do problema;
10. Grafos densos e esparsos: a representação deve ser capaz de codificar soluções em grafos não completos (com menos de $n(n-1)/2$ arestas, onde n é a quantidade de vértices do grafo) preservando as características anteriores.

Deve-se analisar também três propriedades fundamentais para o desempenho de EAs [54]:

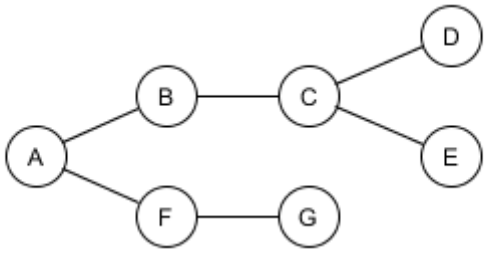
1. Localidade: assim como descrito em Raidl e Julstrom [51], representações com baixa localidade fazem com que os EAs se tornem parecidos com métodos de busca aleatória;
2. Escalabilidade: as representações não devem possuir escalabilidade dos genes. A preferência de um conjunto de genes faz com que tempo de convergência dos EAs reduze;
3. Redundância: ocorre quando mais de um genótipo produz o mesmo fenótipo. Se mais de uma codificação representar uma mesma árvore a complexidade do algoritmo pode ser alterada. Caso a redundância for sinônima (codificações similares), a complexidade do EA não é alterada.

Na literatura, é possível encontrar diversas representações para PPRs em algoritmos evolutivos, das quais se destacam:

1. Vetor de Características [44, 13];
2. Número de Prüffer [45, 62];
3. Predecessores [1];
4. Conjunto de Arestas [50];
5. Chaves Aleatórias [55, 57];
6. Nó-Profundidade-Grau [17, 19, 14].

4.1 Vetor de Características

Vetor de Características (CV, *Characteristic Vector*) é uma representação do tipo *array* binário. Seja um vetor de m posições, em que m representa o número de arestas de um grafo completo. Cada aresta do grafo é associada a um índice, 1...m, de uma posição do vetor [44, 13]. A codificação da árvore por CV é realizada pela inserção do valor 1 no vetor, na posição correspondente da aresta, caso haja uma aresta conectando dois vértices e 0 caso contrário. A Figura 4.1 ilustra um exemplo do CV.



1	0	0	0	1	0	1	0	0	0	0
A-B	A-C	A-D	A-E	A-F	A-G	B-C	B-D	B-E	B-F	B-G

1	1	0	0	0	0	0	0	0	1
C-D	C-E	C-F	C-G	D-E	D-F	D-G	E-F	E-G	F-G

Figura 4.1: *Árvore e sua respectiva representação por CV.*

Uma das propriedades do CV é conseguir representar todas as árvores possíveis, incluindo grafos que não representam árvores. Outra propriedade é a alta localidade. Alterar o valor de uma posição no vetor implica na inclusão ou remoção direta da aresta no grafo. A localidade não depende da topologia da árvore [55].

Uma característica importante do CV é sua alta complexidade de tempo e espaço. Para grafos esparsos a complexidade de espaço é de $O(m)$, enquanto para grafos densos e completos é de $O(n^2)$. Para decodificar a árvore é necessário percorrer todo o *array* de comprimento m. Isso gera a complexidade de tempo de $O(m)$ para grafos esparsos e $O(n^2)$ para grafos completos [51]. Essas complexidades são obtidas considerendo todas as operações sobre o vetor. Dentre as representações existentes na literatura a CV é a que possui a maior complexidade. Além disso, o uso de operadores de cruzamento de pontos e mutação de *flip* podem gerar soluções infactíveis. Tais soluções podem conter ciclos ou representar grafos desconexos. Assim, são necessários mecanismos de correção para garantir que as soluções obtidas são factíveis.

4.2 Número de Prüfer

O Número de Prüfer é a representação mais conhecida na literatura [28]. Essa representação correlaciona uma única *string* de tamanho $n-2$ a uma árvore geradora ou seja, trata-se de uma função bijetora. Essa propriedade garante que a codificação de árvores por número de Prüfer não possui tendência para nenhum tipo de solução.

Formalmente, o número de Prüfer é definido como: dada uma árvore T com n vértices, o número de Prüfer, P, é um número de $n-2$ dígitos com valores entre 1 e n. A codificação por o número de Prüfer funciona por meio de uma busca na árvore pelo vértice com menor rótulo, v_i e de grau 1. A seguir, o predecessor do vértice é adicionado em P. Após a inserção do vértice v_{i-1} em P, v_i é removido da árvore. Essa iteração se repete até sobrar apenas dois vértices na árvore. Ao término do processo é o obtido o número de Prüfer P.

A Figura 4.2 apresenta um exemplo do número de Prüfer.

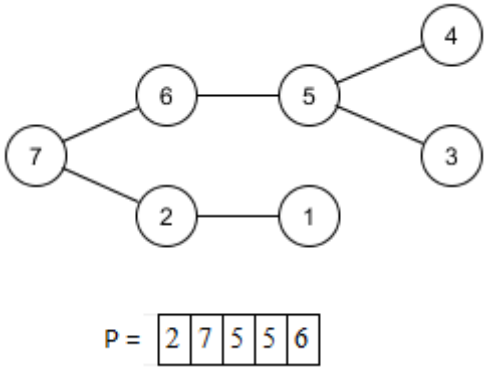


Figura 4.2: *Árvore e sua respectiva representação por Número de Prüfer.*

A complexidade de tempo do número de Prüfer para os processos de codificação e decodificação é de $O(n\log n)$ [28]. O número de Prüfer possui como principais vantagens: a certeza de que toda árvore pode ser representada, cada número de Prüfer representa apenas uma árvore, e não há tendência, ou seja, todas as árvores são representadas de forma uniforme.

Entretanto, apesar das vantagens a sua localidade é baixa, já que a mudança de apenas um dígito no número de Prüfer pode alterar drasticamente a árvore resultante [28]. Desta forma, o operador de mutação pode produzir indivíduos descendentes muito diferentes dos pais, aproximando o comportamento do EA a um algoritmo de busca aleatória.

Outra desvantagem dessa representação é quando se utiliza grafos esparsos, como é o caso de alguns tipos de PPRs reais. Nesta situação é inviável a utilização do

número de Prüfer por não gerar árvores válidas, uma vez que o processo de codificação e decodificação considera que os grafos são completos.

4.3 Predecessores

A representação por predecessores tem como fundamento os teoremas de Even [22]. Nessa representação, uma árvore é codificada escolhendo aleatoriamente um vértice que será chamado de raiz. Para cada vértice v_i (exceto a raiz) é armazenado o seu vértice predecessor, v_j. Esse vértice está imediatamente antes de v_i no caminho que começa na raiz até v_i. Se $Pred[v_i] = v_j$ significa que v_j é o primeiro vértice no caminho entre v_i e a raiz. A raiz nunca será armazenada por não ter um vértice predecessor. Como consequência disso, o vetor da representação sempre terá comprimento igual a $n-1$. A Figura 4.3 ilustra a representação por predecessores.

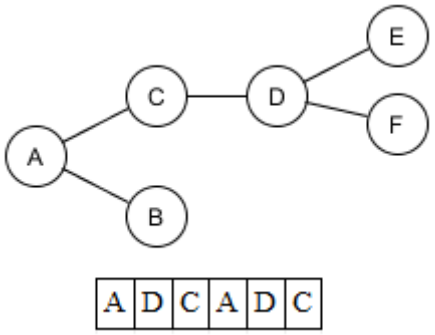


Figura 4.3: *Árvore e sua respectiva representação por predecessores.*

O processo de codificação e decodificação para essa representação possui complexidade de $O(n)$ [45, 1]. Porém, a representação por predecessores gera uma grande quantidade de soluções infactíveis, como ciclos ou árvores desconexas ao serem utilizados os operadores de cruzamento de pontos.

4.4 Conjunto de Arestas

A representação por conjunto de arestas consiste em representar uma árvore diretamente pelo conjunto de suas arestas. Computacionalmente, o conjunto de arestas pode ser representada por um vetor ou uma tabela *hash* [8]. As entradas da tabela *hash* são compostas por pares de vértices que representam as arestas da árvore. Essa implementação permite obter um tempo constante para operações de inserção, remoção e buscas das arestas. Tanto a tabela *hash* quanto a implementação por vetor possuem complexidade de espaço de $O(n)$, onde n é o número de vértices do grafo.

Soluções para representação por conjunto de arestas podem ser obtidas por modificações dos algoritmos clássicos de Prim e Kruskal e um algoritmo de geração por caminhos aleatórios [50]. A complexidade de tempo dos algoritmos de Prim e Kruskal varia entre $O(n)$ e $O(m)$ [50]. A Figura 4.4 ilustra a representação por conjunto de arestas.

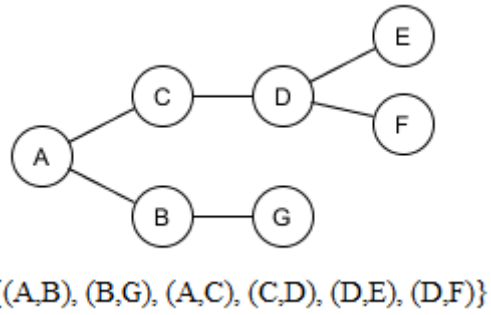


Figura 4.4: *Árvore e sua respectiva representação por Conjunto de Arestas.*

A representação conjunto de arestas para garantir a obtenção de somente soluções factíveis faz uso de operadores de cruzamento e mutação específicos para a codificação [50, 51]. Os operadores de cruzamento tem o seguinte funcionamento básico: primeiramente insere no filho as arestas que estão presentes em ambos os pais, depois de forma aleatória adiciona arestas que pertencem a um dos pais e que não formem ciclos na solução.

4.5 Chaves Aleatórias

A representação Chaves Aleatórias (NetKeys, *Network Random Keys*) representa uma árvore por meio de um vetor de números reais de comprimento $m = n(n-1)/2$. Cada índice nesse vetor representa uma aresta e cada número associado ao índice representa a importância da aresta.

A representação NetKeys possui alta localidade, o que resulta em uma mutação que altera uma aresta ou nenhuma. Além disso, a construção de árvores nessa representação garante que não existirão soluções infactíveis, pois o processo de decodificação aplica o algoritmo de Kruskal ao vetor de pesos. Porém, a NetKeys possui complexidade de espaço e tempo superiores a outras representações presentes na literatura [55].

4.6 Nó-Profundidade-Grau

A representação Nó-Profundidade-Grau (NDDE) é baseada nos conceitos de caminhos, profundidade e grau do vértice em uma árvore [17]. A representação consiste

em triplas de valores (de_i, v_i, deg_i) armazenadas em forma de vetor, onde de_i representa a profundidade e deg_i o grau do vértice v_i ($i \in \{1, 2, ..., n\} \mid n$ é a quantidade de vértices da árvore). A raiz da árvore sempre terá a profundidade 0. A ordem dos demais vértices no vetor é determinada por meio de algum método de busca em grafos, como por exemplo, busca em profundidade (DFS, *Depth-First Search*) [8]. Delbem et al.[19] demonstram teoricamente e experimentalmente que o tempo médio de computação da NDDE é de $O(\sqrt{n})$.

A Figura 4.5 ilustra um exemplo de uma árvore. A Tabela 4.1 apresenta a representação NDDE da árvore apresentada na Figura 4.5.

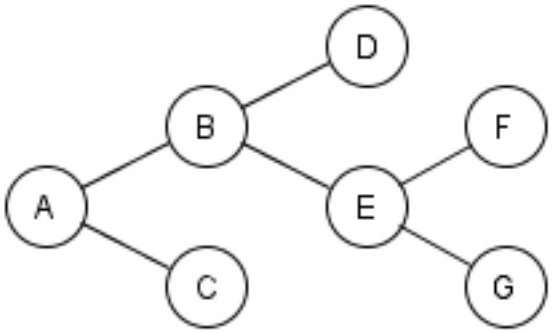


Figura 4.5: *Árvore com sete vértices.*

Tabela 4.1: *NDDE da árvore da Figura 4.5.*

i	1	2	3	4	5	6	7
de	0	1	2	3	3	2	1
v	A	B	E	G	F	D	C
deg	2	3	3	1	1	1	1

A partir da codificação da árvore em NDDE, podem-se aplicar operadores de mutação sobre essa representação para obter novas soluções. Em especial os operadores PAO (*Preserve Ancestor Operator*) e CAO (*Change Ancestor Operator*) podem ser utilizados.

Com o objetivo de se obter novas soluções factíveis aplicam-se os operadores de mutação PAO e CAO nas árvores representadas pela NDDE. Basicamente, ambos os operadores podam uma árvore em um dado vértice e transfere a subárvore podada para outra árvore. A árvore que foi podada é chamada de árvore de origem, T_{ori}. A árvore que recebe a subárvore podada de T_{ori} é chamada de árvore de destino, T_{des}.

O PAO necessita de dois vértices, um de origem e um de destino. O vértice de origem p indica a raiz da subárvore podada da árvore de origem T_{ori}. O vértice de destino a em T_{des} representa o vértice o qual a subárvore será conectada.

O CAO é semelhante ao operador PAO com a diferença que agora existe um terceiro vértice r que representa a nova raiz da subárvore.

4.6.1 Operador PAO

Para aplicação do PAO, escolhe-se aleatoriamente um vértice p na árvore de origem T_{ori} e um vértice a na árvore de destino T_{des}. Após a escolha do vértice p o processo de transferência ocorre de acordo com os seguintes passos:

1. Determinar o intervalo $(v_p - v_l)$ que representa a subárvore podada na árvore de origem T_{ori}. Como o vértice v_p é conhecido, basta encontrar o vértice v_l de acordo com a NDDE. Esse vértice é o último vértice da subárvore podada. O intervalo $(v_p - v_l)$ é composto por v_i vértices em que $v_i > v_p$ e $de_i > de_p$.
2. Inserir o intervalo $(v_p - v_l)$ na árvore de destino T_{des} na posição $v_a + 1$.
3. Atualizar os valores de grau e profundidade dos vértices.

O operador PAO causa poucas mudanças na árvore. A Figura 4.6 ilustra um exemplo da aplicação do PAO na árvore da Figura 4.5. A representação NDDE de T_{des} é apresentada pela Tabela 4.2. Os vértices escolhidos são a = C (v_4) e p = E (v_5).

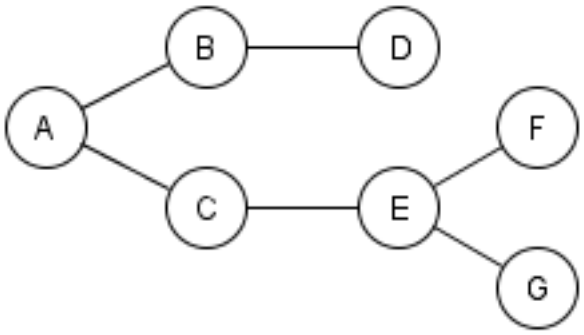


Figura 4.6: *Árvore da Figura 4.5 com aplicação do operador PAO.*

Tabela 4.2: *NDDE da árvore da Figura 4.6.*

i	1	2	3	4	5	6	7
de	0	1	2	1	2	3	3
v	A	B	D	C	E	F	G
deg	2	2	1	2	3	1	1

4.6.2 Operador CAO

No operador CAO a subárvore que será transferida recebe uma nova raiz representada pelo vértice r. O vértice r é escolhido de forma aleatória dentro do intervalo

$(v_p - v_l)$ e deve ser diferente do vértice p. A ordem dos vértices da subárvore é trocada. Ao invés de copiar a subárvore enraizada em p, primeiramente é armazenado em um vetor temporário a subárvore enraizada no vértice r. Em seguida, armazena-se a subárvore enraizada no primeiro vértice anterior a r, $(v_r - 1)$, que está presente no intervalo de $(v_p - v_r)$ e que não contenha a subárvore já presente no vetor temporário. O vetor temporário é preenchido até que o vértice p seja atingindo.

CAO funciona de acordo com os seguintes passos:

1. Escolher o vértice p e, dentro do intervalo $(v_p - v_l)$, a nova raiz representada pelo vértice r.
2. Copiar para o vetor temporário a subárvore enraizada no vértice r.
3. Copiar para o vetor temporário a subárvore enraizada em cada vértice anterior a r no intervalo $(v_p - v_r)$. Cada subárvore não pode conter vértices já presentes no vetor temporário.
4. Inserir os vértices do vetor temporário na árvore de destino T_{des} na posição $v_a + 1$.
5. Atualizar os valores de grau e profundidade dos vértices.

O operador CAO causa mudanças significativas na árvore. A Figura 4.7 ilustra um exemplo da aplicação do CAO na árvore da Figura 4.5. A representação NDDE é apresentada pela Tabela 4.3 .Os nós escolhidos são a = C, p = B e r = G.

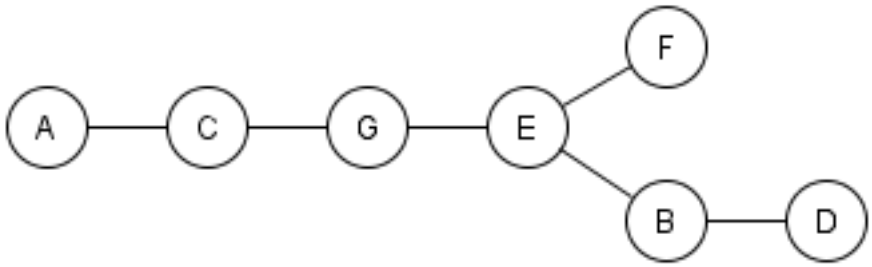


Figura 4.7: *Árvore da Figura 4.5 com aplicação do operador CAO.*

Tabela 4.3: *NDDE da árvore da Figura 4.7.*

i	1	2	3	4	5	6	7
de	0	1	2	3	4	4	5
v	A	C	G	E	F	B	D
deg	1	2	2	3	1	2	1

CAPÍTULO 5

Computação Reconfigurável

Computadores de propósito gerais são utilizados em diversas aplicações que não exigem desempenho elevado. Esses computadores possuem arquitetura e conjunto de instruções fixas, as quais restringem as aplicações e afetam o seu desempenho.

Existem aplicações cujos requisitos excedem as capacidades dos computadores de propósito gerais. Dessa forma, é necessário recorrer a sistemas de propósito específicos, como por exemplo, ASIC (*Application Specific Integrated Circuit*). Porém, esse tipo de sistema possui um custo de projeto e de implementação elevados, além das soluções serem inflexíveis.

A computação reconfigurável é uma abordagem que combina as duas metodologias descritas anteriormente. Essa tem como princípio base de funcionamento os dispositivos lógicos reprogramáveis. Esses dispositivos permitem programar e reprogramar diversas vezes diretamente em *hardware*. Tal característica torna a aplicação flexível e com desempenho elevado.

Dispositivos de *hardware* bastante utilizados em computação reconfigurável são os FPGAs (*Field Programmable Gate Array*). Esses dispositivos possuem velocidade e recursos semelhantes aos ASICs, ao mesmo tempo em que são flexíveis como os computadores de propósito gerais.

FPGAs são dispositivos que possuem alto grau de paralelismo [7], sendo amplamente utilizados em aplicações mais complexas, tanto de propósito específico quanto em propósito geral. Esses dispositivos permitem que o usuário reconfigure suas portas lógicas, fazendo com que o FPGA trabalhe de acordo com a necessidade desejada. Assim, FPGAs oferecem flexibilidade no desenvolvimento de soluções complexas de alto desempenho com o custo de projeto baixo [25].

O FPGA é composto por três componentes:

- Blocos lógicos (CLB, *Configuration Logical Blocks*): circuitos constituídos de *flip-flops* que podem ser configurados independentemente;
- Blocos de entrada e saída (IOB, *Input/Output Block*): interface das conexões dos pinos do FPGA com o mundo externo;

- Chaves de interconexões: trilhas que conectam os CLBs e IOBs.

A Figura 5.1 apresenta a arquitetura do FPGA.

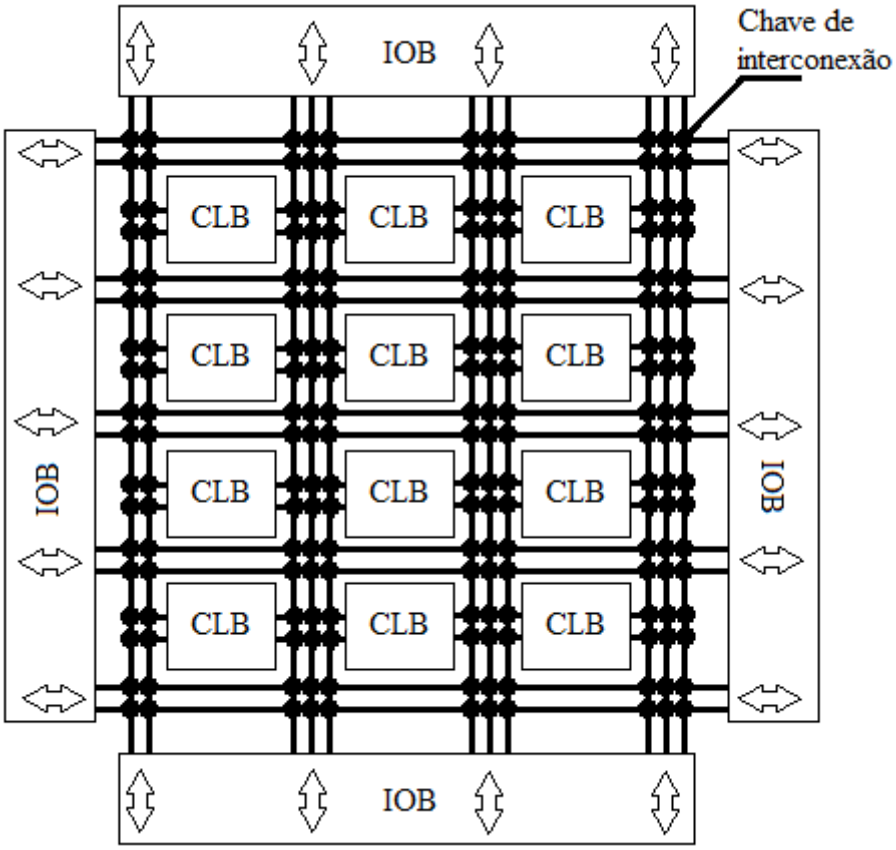


Figura 5.1: *Arquitetura do FPGA.*

Conforme a Figura 5.1 os CLBs enviam e recebem dados entre outros CLBs através das chaves de interconexão. Quando é necessário a troca de dados entre os CLBs e o mundo externo, os dados também são transmitidos pelas chaves de interconexão. A quantidade de CLBs varia de acordo com o modelo da placa FPGA.

5.1 Linguagem descritiva de *hardware*

As técnicas tradicionais para descrever *hardware* fazem uso de interconexões dos elementos lógicos [7]. Porém, devido ao elevado número desses elementos lógicos, o uso dessas técnicas se torna inviável necessitando de outras estratégias para a descrição de *hardware*. Nesse sentido, tem-se utilizado linguagens de descrição de *hardware* (HDL, *Hardware Description Language*) [5]. As HDLs permitem o desenvolvimento de sistemas em nível físico e comportamental, com alto nível de abstração. O uso dessas linguagens possibilita ao projetista testar o funcionamento dos circuitos digitais por meio de simulações, com o objetivo de sintetizá-los após a descrição.

Dentre as diversas HDLs existentes, como VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) [6] e Verilog [46], destaca-se a BSV (*Bluespec System Verilog*) [12, 41, 42].

5.1.1 Linguagem *Bluespec System Verilog*

A BSV é uma linguagem descritiva de *hardware* de alto nível proposta por Arvind em 2003. Ela possui vantagem sobre as outras linguagens descritivas de hardware por ser de fácil implementação com menor tempo de desenvolvimento e número de linhas de código. Essa facilidade deve-se ao alto nível de abstração para lidar com projetos de *hardware*. Outras vantagens da BSV é a alta flexibilidade e melhor desempenho, além de ter uma maior portabilidade entre dispositivos FPGAs [41].

A BSV mapeia os blocos lógicos do projeto de *hardware* em uma representação de alto nível chamada de módulos e que podem ser sintetizados em Verilog RTL [41]. Cada módulo é composto por três componentes [37, 12, 42]:

- Estados: podem ser registradores, *flip-flops* ou memórias descritas no próprio módulo;
- Ações Atômicas Protegidas ou Regras: operam sobre o estado para especificar um conjunto de critérios para a sua execução;
- Métodos de Interface: possibilitam a comunicação do módulo descrito.

A definição de um módulo especifica um esquema que pode ser instanciado múltiplas vezes [42]. Essa divisão em módulos torna a BSV mais vantajosa em relação a outras HDLs, devido à possibilidade de análise isolada de cada regra em cada ciclo. Além disso, a BSV é totalmente sintetizável permitindo que o código HDL seja traduzido em estruturas de *hardware* disponíveis no dispositivos.

CAPÍTULO 6

Arquitetura e Implementação

A arquitetura de *hardware* utilizada é ilustrada pela Figura 6.1 e é constituída pelos seguintes componentes:

- Microprocessador *softcore* - Nios II;
- Memória externa, memória do sistema de *hardware* responsável pelo armazenamento das NDDEs, dos pesos das arestas do grafo G e código do GA implementado para ser executado no microprocessador Nios II;
- Barramento Avalon, responsável pela comunicação entre os dispositivos internos do sistema em *hardware*;
- Interface JTAG, responsável pela comunicação do sistema em *hardware* com dispositivos externos (por exemplo, um PC);
- Módulo P-PAO-S (Circuito Paralelo para PAO), desenvolvido em [25] que executa o operador de mutação PAO.

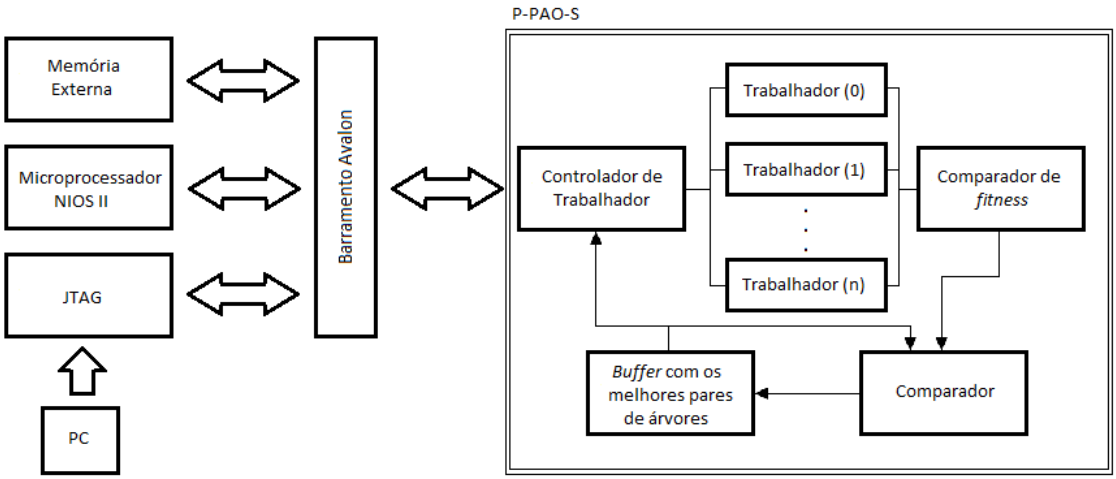


Figura 6.1: *Arquitetura utilizada.*

De acordo com a Figura 6.1, o computador externo (PC) gera sementes aleatórias para definir o estado inicial do gerador de número pseudoaleatórios LFSR (*Linear Feedback Shift Register*). Esse gerador é utilizado para a obtenção dos vértices p e a

utilizados no operador PAO. As sementes são transferidas do computador para o sistema em *hardware* por meio da interface JTAG.

O microprocessador Nios II realiza a execução de todo processo evolutivo do GA, além de ser responsável pela leitura e transferência das NDDEs da memória externa para o P-PAO-S e a cópia para a memória externa das melhores NDDEs encontradas pelo P-PAO-S. Toda essa comunicação é realizada pelo barramento Avalon.

No módulo P-PAO-S, o Controlador de Trabalhador é responsável pela comunicação entre o microprocessador Nios II e a memória externa de forma a enviar e receber T_{ori} e T_{des} para os Trabalhadores. Cada T_{ori} e T_{des} obtido na memória externa é replicado para todos os Trabalhadores. Cada Trabalhador executa o operador PAO com os vértices p e a diferentes entre si. Após a execução dos Trabalhadores o Comparador de *fitness* realiza a comparação do *fitness* das árvores geradas. O melhor par de árvores encontrado é comparado então com as árvores antigas e o melhor par encontrado é armazenado em um *Buffer*. Por fim, o Controlador de Trabalhador acessa o *Buffer* e copia o melhor par de árvores para a memória externa.

6.1 Representação Nó-Profundidade-Grau em FPGA

Em [25] foi realizada a paralelização da NDDE em FPGA. Essa arquitetura implementada foi utilizada neste trabalho. A paralelização foi realizada levando em consideração que cada NDDE é representada por um conjunto de vetores de valores inteiros que pode ser implementado na forma de blocos de memória, cujo número de *bits* é proporcional ao tamanho do vetor [16].

Cada vetor é manipulado em paralelo no FPGA e possui tamanho médio de $O(\sqrt{n})$, onde n é o número de vértices do grafo. Em [25] a estratégia de manipulação das NDDEs de forma paralela em *hardware* foi denominada HP-RNP (*Hardware-Parallelized Representação Nó-Profundidade*). A Equação 6-1 apresenta o tempo de processamento do HP-RNP.

$$\frac{\textit{Número de florestas modificadas}}{\textit{Tempo de processamento}} = \frac{\sqrt{n}}{O(\sqrt{n})} = O(1). \tag{6-1}$$

De acordo com a Equação 6-1 o processo de paralelização do HP-RNP realiza $\sqrt{n}$ modificações na floresta geradora. Como as novas florestas são geradas em tempo médio de $O(\sqrt{n})$, tem-se que o tempo de processamento para gerar uma nova floresta é limitado por uma constante $O(1)$.

6.1.1 Implementação do Operador PAO no Trabalhador

Para cada trabalhador existe uma MEF (Máquina de Estados Finitos). A MEF é constituída pelo operador PAO, gerador de números pseudoaleatórios LFSR e um bloco de memória. A Figura 6.2 apresenta a MEF utilizada.

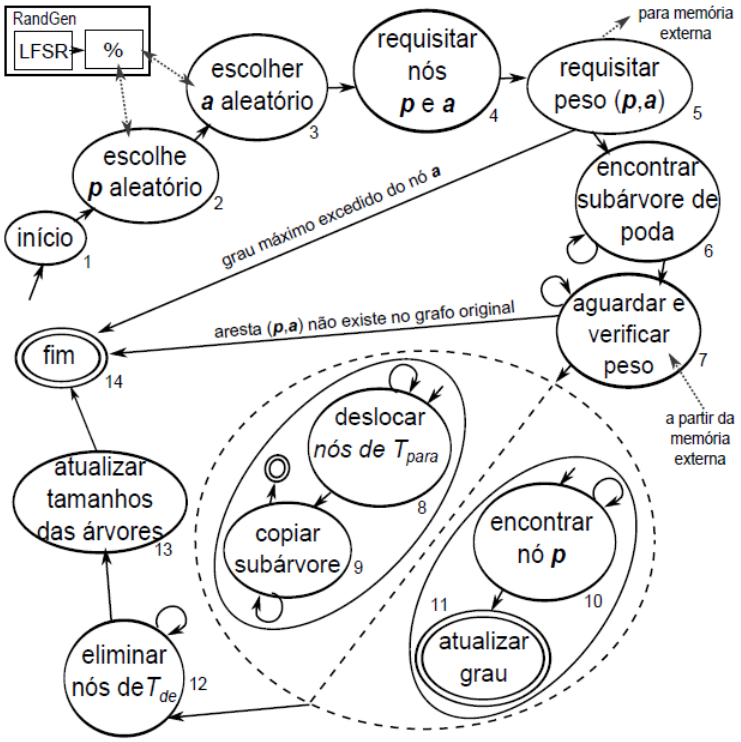


Figura 6.2: *Máquina de estados finitos de cada Trabalhador - Retirado de [25], Capítulo 6, Página 50.*

De acordo com a Figura 6.2, a MEF executa uma série de etapas que são descritas a seguir:

- Estado 1: inicia a MEF;
- Estados 2 e 3: requisita ao gerador de números pseudoaleatórios LFSR dois números aleatórios que representam os vértices de origem p em T_{ori} e de destino a em T_{des};
- Estado 4: requisita da memória os vértices p e a com suas respectivas NDDEs. Em seguida, elas são armazenadas nos blocos de memória dos Trabalhadores;
- Estado 5: requisita para memória externa o peso da aresta referente a conexão entre os vértices p e a do grafo original. Nesse estado também é realizada uma verificação de grau para o vértice a. Se o grau de a for igual ao limite da restrição de grau definida a não pode receber mais nenhuma subárvore, pois o grau máximo permitido será excedido. Então, segue-se para o estado final da MEF. Caso contrário, avança-se para o Estado 6;

- Estado 6: busca a subárvore de poda percorrendo todos os vértices de T_{ori};
- Estado 7: aguarda e verifica se o peso requisitado no Estado 5 é nulo ou não. Caso seja, significa que não existe conexão entre os vértices p e a, avançando para o estado final da MEF. Senão, segue-se para o Estado 8;
- Estados 8, 9, 10 e 11: esses estados são executados em paralelo, uma vez que os estados 8 e 9 alteram a árvore de destino e os estados 10 e 11 alteram a árvore de origem. O Estado 8 desloca em T_{des} a quantidade de vértices necessária para alocar a subárvore podada em T_{ori}. O Estado 9 copia os vértices da subárvore podada de T_{ori} para T_{des}. O Estado 10 procura o vértice ancestral de p em T_{ori}. O Estado 11 decrementa o grau do vértice ancestral de p em T_{ori};
- Estado 12: elimina a subárvore podada de T_{ori};
- Estado 13: atualiza os tamanhos das novas árvores. O tamanho da subárvore podada é subtraído do tamanho de T'_{ori} e adicionado ao tamanho de T'_{des};
- Estado 14: finaliza a MEF.

6.2 Implementação do Algoritmo Genético

A implementação do GA no processador Nios II visa a redução do peso das NDDEs por meio do processo evolutivo com um tempo de execução inferior se comparado com a implementação realizada em *software* Essa redução de tempo se deve ao poder de paralelismo do FPGA. A implementação do GA é a principal contribuição deste trabalho.

Antes da execução do GA pelo processador Nios II, o *software* executa alguns passos: leitura das matrizes de pesos, geração das florestas iniciais e armazenamento das florestas geradas na memória externa. É utilizado o algoritmo modificado de Kruskal [35] para geração das florestas iniciais. A modificação consiste em incluir o grau de cada vértice na estrutura de dados que armazena a floresta. O grau é utilizado pelo operador PAO e a sua inclusão é verificada a fim de evitar a geração de soluções iniciais infactíveis. O algoritmo utiliza uma matriz de pesos aleatórios para gerar as florestas iniciais. Após a geração das florestas, os pesos das arestas são atualizados com os valores dos conjuntos de dados utilizados (ver Capítulo 8). Cada floresta gerada possui apenas uma árvore, de acordo com a formulação do problema de árvore geradora mínima com restrição de grau.

Uma vez geradas as florestas iniciais, o GA é executado. A cada iteração do GA, o processo evolutivo realiza a seleção da floresta por meio do torneio de 3. Aleatoriamente são selecionadas 3 florestas para competirem entre si. A de melhor *fitness* (menor peso) é selecionada e então enviada para o operador PAO. A quantidade de vezes que o operador

PAO é executado em uma única geração do GA é dada pela Equação 6-2.

$$\text{Execuções} = \frac{2.400.000}{\text{Número de trabalhadores} * \text{Número de gerações}} \tag{6-2}$$

Segundo a Equação 6-2 a quantidade de execuções depende do número de trabalhadores alocados e da quantidade de gerações do GA. Quanto maior o número de trabalhadores maior o número de execuções do operador em uma única geração. Portanto, a quantidade de execuções total do operador é reduzida quando ocorre o aumento de trabalhadores.

A implementação do operador PAO apresentada em [25] funciona com a entrada de duas árvores, uma de origem e outra de destino. Para representar NDDEs do dc-MST cada floresta possui apenas uma árvore. Para contornar essa restrição a árvore da floresta é dividida em duas. A Figura 6.3 ilustra como ocorre o processo de divisão e união da floresta.

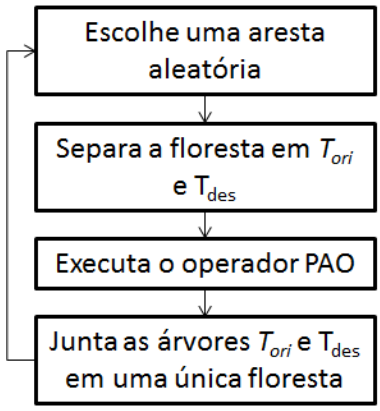


Figura 6.3: *Divisão e união da floresta.*

De acordo com a Figura 6.3 a divisão da floresta ocorre removendo de forma aleatória uma aresta que conecta dois vértices i e j da árvore. A árvore de origem T_{ori} será a subárvore podada com raiz j. A árvore de destino T_{des} será a árvore original sem a subárvore podada. Logo após a execução do operador PAO a árvore restante de T_{ori} é conectada em T_{des} pela aresta removida. A escolha da aresta de forma aleatória antes de cada execução do operador PAO faz com que qualquer efeito advindo desse processo seja atenuado.

A divisão e união da floresta ocorrem antes e depois de cada execução do operador PAO, respectivamente. Portanto, ela é realizada na mesma quantidade de vezes que o PAO é executado (conforme Equação 6-2). Como a divisão da floresta ocorre antes do operador PAO ser executado, a aresta removida é a mesma para todos os trabalhadores.

No final de cada geração do GA o melhor indivíduo obtido pela aplicação do PAO nos Trabalhadores é inserido na população, caso seja melhor do que a pior floresta

da população. Caso ele seja pior do que todas as florestas esse novo indivíduo é descartado. Esse processo de inserção do novo indivíduo obtido na população é considerado totalmente elitista.

6.3 Ferramentas Utilizadas

Inicialmente, foi esperado o uso do FPGA da família Stratix V, modelo 5SGXEA7N. Porém, devido a problemas de importação foi utilizado o FPGA da família Cyclone II, modelo EP2C70F896C6. Esse modelo trabalha em uma frequência de de 500MHz, possui uma memória SDRAM de 1MB, *display* LCD, conexões PCI *Express* e USB para comunicação com dispositivos externos. Possui 68416 células lógicas programáveis. Além disso, contém o processador Nios II e suporta a interface JTAG (*Joint Test Action Group*). Essa diferença entre dispositivos ocasionou na utilização de grafos com menor número de vértices.

Para programação no dispositivo FPGA foi utilizado o *software* Quartus versão 10.1. Esse *software* proprietário é produzido pela Altera, mesma empresa que desenvolveu a placa FPGA. O Quartus é uma ferramenta EDA (*Eletronic Design Automation*) no qual é possível realizar síntese, simulações, análises e programação em dispositivos FPGAs. Por meio dele o tempo de desenvolvimento de projetos em *hardware* configurável é reduzido devido a automatização de suas tarefas. Implementar um projeto de circuito envolve as seguintes etapas: descrição do projeto, compilação, simulação e programação do dispositivo FPGA.

Além do Quartus, foi utilizada a ferramenta SOPC *Builder* (*System on a Programmable Chip*) para geração de um SOPC completo. Essa ferramenta permite a automatização das conexões entre os componentes de *hardware*, criando assim sistemas de auto desempenho. O SOPC *Builder* possui uma vasta biblioteca de componentes como, por exemplo, controladores de memória, interfaces, periféricos, processadores (como o Nios II) e co-processadores. As interconexões entre os componentes são realizadas por meio do barramento Avalon.

O GA desenvolvido, aqui chamado de H-GA (*Hardware Genetic Algorithm*), para ser executado no processador Nios II foi implementado utilizando linguagem C. A execução no Nios II permite que o H-GA tenha desempenho superior em relação aos processadores de propósito geral, pricipalmente em relação ao tempo. A linguagem descritiva de *hardware* utilizada foi a BSV.

CAPÍTULO 7

Configurações e Conjunto de Dados

O H-GA foi utilizado para resolução do problema de árvore geradora mínima com restrição de grau (dc-MSTP) descrito na Seção 2.1.1. Dentre os testes realizados optou-se pelos parâmetros apresentados na Tabela 7.1 para a execução do algoritmo e obtenção dos resultados que serão apresentados e discutidos no Capítulo 8.

Tabela 7.1: *Parâmetros do H-GA.*

Número de Gerações	300
Tamanho da População	6
Operador de Seleção	Torneio de 3
Operador de Mutação	PAO

Testes empíricos resultaram que um número de 300 gerações é o suficiente para encontrar uma solução ótima. Menos que 300 gerações faz com que o espaço de busca não seja bem explorado, enquanto que mais que 300 gerações não leva a uma melhora de *fitness*.

Cada indivíduo da popução representa uma floresta codificada na representação NDDE. O tamanho da população foi determinado por meio de testes empíricos. Com uma população inferior a 6 indivíduos o H-GA apresentou convergência prematura para ótimos locais. Com uma população maior que 6 indivíduos o tempo de processamento aumentou, porém os valores de *fitness* não apresentaram melhora.

O melhor indivíduo obtido a partir das diversas aplicações do operador de mutação nos trabalhadores somente é inserido na população caso seja melhor do que o pior indivíduo da população. Ou seja, se o novo indivíduo A possuir um *fitness* melhor do que o indivíduo B, de pior *fitness* presente na população, A é inserido na população no lugar de B. Caso contrário, A é descartado. Esse processo é conhecido como elitismo total, ou seja, todos os indivíduos são preservados, sendo descartados somente se são piores que um novo indivíduo.

Os dados utilizados para esse trabalho foram obtidos diretamente dos autores e tratam-se de casos de *benckmark* da literatura para avaliação de GAs aplicados ao dc-

MSTP. Esses dados estão dispostos em dois conjuntos de grafos denominados SHRD e M-Graphs.

O conjunto SHRD é composto por grafos completos de 15, 20, 25 e 30 vértices. Adicionalmente, foi gerado um grafo com 50 vértices que segue o mesmo processo de criação dos SHRD. Os pesos das arestas desse conjunto de grafos são compostos por números inteiros proporcionais ao rótulo dos vértices que compõem a aresta. Em outras palavras, quanto maior o rótulo dos vértices, maiores serão os valores dos pesos. Esse conjunto de grafos foi definido em [34] e segue o processo de construção descrito a seguir. O primeiro vértice é conectado a todos outros vértices por uma aresta de custo l. O segundo vértice é conectado aos outros vértices, exceto o vértice 1, por uma aresta de tamanho $2l$. A regra de construção dos pesos das arestas pode ser definida por, o vértice v_i, $i = 1 \ldots n$, onde n número de vértices, é conectado a todos os vértices v_k com $k > i$, por uma aresta de custo $i * l$. Para tornar o processo aleatório, o peso das arestas é perturbado por um número aleatório com distribuição uniforme entre 1 e 18, para $l = 20$. Essa perturbação diminui a probabilidade de surgir um grande número de soluções ótimas, sem afetar a complexidade do problema.

O conjunto M-Graphs é composto por grafos completos com 50, 100, 200, 300, 400 e 500 vértices. Para os grafos com 50, 100 e 200 vértices, o conjunto de teste possui 3 matrizes de adjacências com pesos diferentes, que serão diferenciados pelo código N1, N2, N3. O processo de construção dos pesos das arestas desses grafos foi proposto por [33] como segue. Para cada grafo gerado são definidos quatro parâmetros: n: número de vértices, f: número de vértices com grau elevado na árvore geradora mínima, ld: limite inferior do grau dos vértices com grau elevado e lu: limite superior do grau dos vértices com grau elevado. Primeiramente, são construídos f diferentes grafos do tipo estrela, com o grau de cada grafo escolhido de forma aleatória dentro do intervalo $[ld, lu]$. Esses grafos estrela são conectados pela adição de $f - 1$ arestas de forma aleatória. Como resultado, nesse momento obtém-se uma árvore. As árvores resultantes, até esse momento, podem conter menos vértices do que o número n de vértices do grafo. Então, para completar o grafo, são adicionados vértices e arestas de forma aleatória. Para as arestas que compõem a árvore geradora são atribuídos pesos de forma aleatória no intervalo entre $[0;0,1]$. Para as demais arestas os valores são obtidos de forma aleatória dentro do limite $(0,1;1,0]$ tendo 6 casas de precisão.

A implementação do operador PAO em FPGA desenvolvido em [25] só permite o uso de grafos com pesos nas arestas do tipo inteiro e com valor máximo de 255. No caso do conjunto de dados SHRD, os pesos das arestas excedem o valor máximo suportado pela implementação, assim foi aplicado um processo de normalização dos valores definido pela

Equação 7-1.

$$\text{Novo peso} = \lceil \left(\frac{\text{Peso}}{\text{Maior peso das arestas do grafo}} \right) * 255 \rceil \tag{7-1}$$

Para o conjunto de grafo M-Graphs foi necessário a conversão dos pesos para que os mesmos se tornassem valores inteiros. Essa conversão foi realizada por meio da Equação 7-2.

$$\text{Novo Peso} = \lceil \text{Peso} * 255 \rceil \tag{7-2}$$

Vale ressaltar, que apesar de terem sido analisadas outras configurações de parâmetros para o H-GA os resultados apresentados referem-se a testes preliminares do algoritmo. Os testes preliminares são devidos as limitações da implementação do operador PAO na placa FPGA, como a necessidade de divisão da floresta em duas e normalização dos pesos das arestas. Essas limitações do operador PAO podem ser resolvidas utilizando um novo modelo de FPGA.

CAPÍTULO 8

Resultados

Neste Capítulo serão apresentados os resultados da aplicação do H-GA para resolver o problema de árvore geradora mínima com restrição de grau (dc-MSTP). Os dados apresentados neste Capítulo referem-se aos valores no universo original dos dados. Para isso, ao final da execução do H-GA as soluções obtidas, tanto para a população inicial, quanto para a população final, tiveram seu *fitness* recalculado com as matrizes de pesos originais dos grafos.

O H-GA foi executado com cada um dos grafos, de ambos os conjuntos de teste, com as restrições de grau $d = 3, 4$ e 5. Devido ao caráter estocástico dos GAs, cada teste foi replicado 10 vezes e sua média será apresentada nos resultados.

8.1 Grafos SHRD

A seguir são apresentados e discutidos os resultados obtidos para o conjunto de grafos SHRD. Primeiramente, será discutida a capacidade de otimização do H-GA. Depois, os resultados serão comparados com os valores ótimos obtidos por um método determinístico e com o desempenho do GA em *software* que também faz uso da representação nó-profundidade-grau.

Com o objetivo de analisar a capacidade de otimização do método proposto, a Tabela 8.1 apresenta os valores médios, de todas as execuções, do *fitness* das populações iniciais para cada grafo SHRD.

A coluna *Média dos Melhores Indivíduos* apresenta a média dos melhores valores de *fitness* encontrados nas 10 execuções do H-GA para as populações iniciais obtidas. Já a coluna *Média da Média da População* apresenta a média da média dos valores de *fitness* da população inicial de cada execução. Observa-se, dos dados apresentados na Tabela 8.1 que os indivíduos gerados nas populações iniciais possuem diversidade entre si, visto que a média dos melhores indivíduos é diferente da média da população. No entanto, essa diversidade pode ser considerada pequena, pois a diferença, entre os valores de *fitness* do melhor indivíduo e da média da população, não é expressiva, em torno de 3%.

Tabela 8.1: *Fitness das populações iniciais.*

Vértices	Grau	Média dos Melhores Indivíduos	Média da Média da População
SHRD 15	3	1106,70	1277,31
	4	1176,20	1355,86
	5	1168,80	1406,61
SHRD 20	3	2267,40	2441,46
	4	2246,50	2508,35
	5	2351,30	2576,70
SHRD 25	3	3123,00	3485,98
	4	3208,90	3639,33
	5	3241,90	3600,83
SHRD 30	3	5292,60	5849,35
	4	6156,20	6544,25
	5	6745,40	7103,63
SHRD 50	3	15629,60	16586,16
	4	16375,60	17462,61
	5	16943,20	18055,38

A Tabela 8.2 apresenta as médias dos valores de *fitness* das populações finais obtidas após a execução H-GA. Novamente, os dados dizem respeito a melhor solução obtida e a média da população.

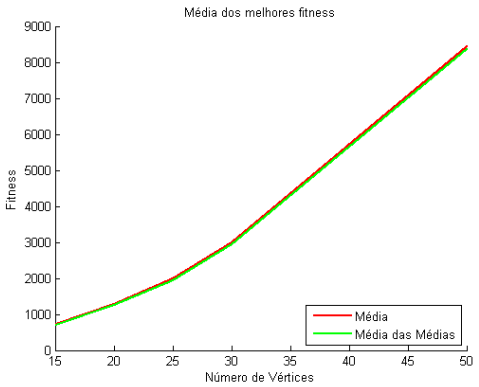
Tabela 8.2: *Fitness das populações finais.*

Vértices	Grau	Média dos Melhores Indivíduos	Média da Média da População	Melhora (%)
SHRD 15	3	692,10	719,08	43,70
	4	669,30	700,91	48,31
	5	585,50	619,44	55,96
SHRD 20	3	1267,30	1288,23	47,24
	4	1113,70	1145,28	54,34
	5	941,80	979,41	61,99
SHRD 25	3	1950,80	1988,96	42,94
	4	1602,80	1650,91	54,64
	5	1454,90	1510,59	58,05
SHRD 30	3	2950,50	2991,46	48,86
	4	2431,30	2519,95	61,49
	5	2270,60	2355,58	66,84
SHRD 50	3	8363,60	8437,68	49,13
	4	6651,60	6824,26	60,92
	5	5825,10	6051,43	66,48

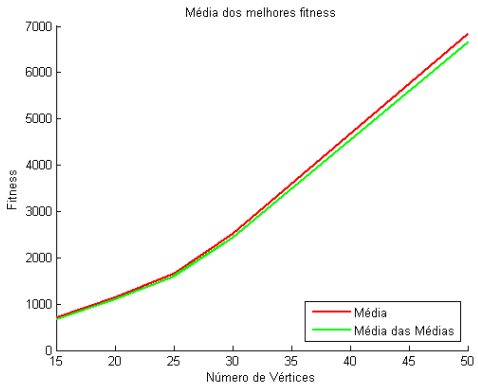
Na Tabela 8.2 além dos valores de *fitness*, também é apresentada a taxa de melhora obtida pelo H-GA. A melhora é calculada entre as médias dos valores de *fitness*

do melhor indivíduo das populações inicial e final, em valores percentuais. Observa-se que, o GA diminuiu em média 50% os valores obtidos na população final em relação a inicial. A maior melhora ocorreu para o caso SHRD 30 com restrição de grau 5, com um ganho de $66,84\%$. Por outro lado, o pior ganho foi de $43,7\%$ para o grafo SHRD 15 com restrição de grau 3. Esses resultados mostram que o H-GA tem capacidade otimização, conseguindo otimizar o *fitness* das soluções da população inicial, e assim, ter soluções melhores na população final.

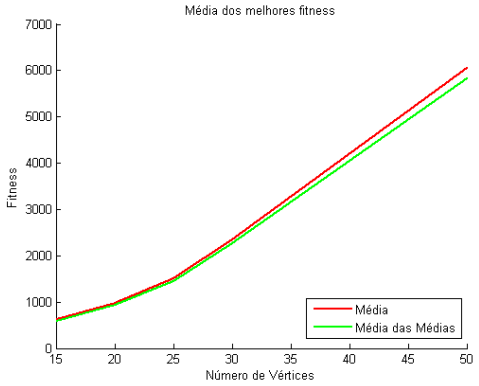
A Figura 8.1 ilustra os gráficos dos *fitness* das populações finais conforme a Tabela 8.2. Destaca-se da Figura 8.1, que os valores de *fitness* do melhor indivíduo e da média da população são muito próximos. Isso indica que, provavelmente, o H-GA convergiu em seu resultado. Esse é mais um indicativo de que o algoritmo desenvolvido possui capacidade de otimização.



(a) *Fitness da população final para restrição de grau igual a 3.*



(b) *Fitness da população final para restrição de grau igual a 4.*



(c) *Fitness da população final para restrição de grau igual a 5.*

Figura 8.1: *Fitness das populações finais para todos os conjuntos de vértices.*

Em [51] são apresentados os valores ótimos para o dc-MSTP com os grafos

SHRD de 15, 20, 25 e 30 vértices obtidos por um algoritmo exato de *branch and cut*. Além da implementação em FPGA, também foi aplicada para os casos de teste uma versão em *software* de um algoritmo genético com a representação nó-profundidade-grau, chamado EANDD. Para a realização dos testes foram utilizadas as mesmas configurações da implementação em FPGA. Além disso, foi garantido um número similar de avaliações em ambas as implementações. O EANDD foi executado utilizando um computador com processador Intel Core i7, com 32Gb de memória RAM. A Tabela 8.3 apresenta o *fitness* da solução ótima [51], e as médias de *fitness* do melhor indivíduo e de tempo de execução de ambas as implementações analisadas. Para o grafo SHRD de 50 vértices não existe valor para a solução ótima, pois o mesmo foi proposto neste trabalho e não foi possível desenvolver um método para obter a solução ótima.

Tabela 8.3: *Comparação dos fitness obtidos na implementação em software e hardware.*

Vértices	Grau	Ótimo	H-GA		EANDD	
			Fitness	Tempo (s)	*Fitness*	Tempo (s)
SHRD 15	3	582	692,10	0,0000896	618,48	0,1894
	4	430	669,30	0,0000888	458,12	0,2302
	5	339	585,50	0,0000875	349,44	0,1942
SHRD 20	3	1088	1267,30	0,0000800	1147,36	0,2178
	4	802	1113,70	0,0000807	851,26	0,2036
	5	627	941,80	0,0000788	674,94	0,2678
SHRD 25	3	1745	1950,80	0,0000778	1851	0,2558
	4	1276	1602,80	0,0000793	1376,14	0,2762
	5	999	1454,90	0,0000758	1076,54	0,2696
SHRD 30	3	2592	2950,50	0,0000854	2709,86	0,4082
	4	1905	2431,30	0,0000869	2033,88	0,4068
	5	1504	2270,60	0,0000857	1613,74	0,3148
SHRD 50	3	-	8363,60	0,0000838	7990,46	0,4120
	4	-	6651,60	0,0000883	5985,86	0,4390
	5	-	5825,10	0,0000880	4761,30	0,3990

De acordo com a Tabela 8.3, é possível observar que os valores de *fitness* obtidos via *software* são melhores do que os obtidos via *hardware*. Uma alternativa possível para a melhora dos resultados via *hardware* seria uma heurística para a escolha dos vértices p e a, utilizados no operador PAO. No entanto, a implementação em *hardware* necessita de um tempo de execução expressivamente inferior ao obtido pela implementação em *software*. Além disso, o tempo de execução mantém-se, praticamente, constante com o aumento no tamanho dos grafos, o que não acontece com a implementação em *software*. Esse resultado advém, principalmente, da capacidade de paralelismo dos FPGAs, o que

indica a aplicação do mesmo para solução de problemas em tempo real. Assim, mesmo que sejam necessárias mais avaliações no H-GA para obter resultados equivalentes ao EANDD, o tempo de execução não seria um problema.

8.2 Grafos M-Graphs

A seguir serão apresentados os resultados obtidos utilizando o conjunto de grafos M-Graphs. Novamente é discutida a capacidade de otimização do H-GA e é realizada a comparação entre os resultados obtidos com uma implementação em *software*.

Para analisar a capacidade de otimização, a Tabela 8.4 apresenta os valores médios, de todas as execuções, do *fitness* das populações iniciais para cada grafo M-Graph.

Com os dados apresentados na Tabela 8.4 é possível observar que os indivíduos gerados possuem diversidade entre si, dado que a média dos melhores indivíduos difere da média da população. Porém, essa diversidade é pequena, em torno de 3%, assim como aconteceu para os grafos SHRD.

Após a execução do H-GA foram obtidos os valores de *fitness* das populações finais. A Tabela 8.5 apresenta esses valores, expressos pela média das melhores soluções obtidas em cada execução e a média da população.

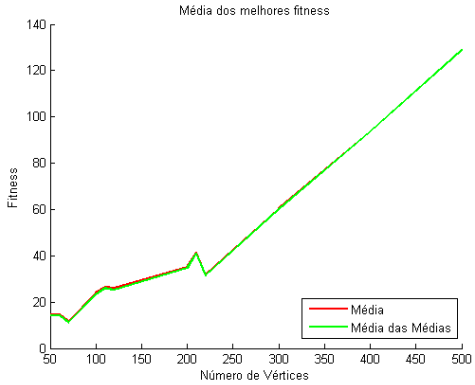
Tabela 8.5: *Fitness das populações finais.*

Vértices	Grau	Média dos Melhores Indivíduos	Média da Média da População	Melhora (%)
M-Graph 50 (N1)	3	14,150350	14,594200	58,35
	4	15,345000	15,967891	55,21
	5	15,743500	16,257724	55,15
M-Graph 50 (N2)	3	14,274801	14,733051	58,44
	4	15,121450	15,700350	56,59
	5	15,280350	15,964050	56,22
M-Graph 50 (N3)	3	11,147400	11,582700	61,23
	4	13,059150	13,454791	55,29
	5	13,672650	14,153225	53,01
M-Graph 100 (N1)	3	23,261798	23,924072	64,30
	4	29,086349	29,900223	55,82
	5	34,191200	36,771515	46,56
M-Graph 100 (N2)	3	25,870801	26,497435	62,68
	4	30,821202	31,472759	55,92
	5	32,100001	32,884784	53,75
M-Graph 100 (N3)	3	25,269051	25,889934	63,23
	4	30,259102	31,251935	55,99
	5	32,141703	33,043402	53,66
M-Graph 200 (N1)	3	34,720905	35,165537	73,44
	4	49,394902	50,398387	62,41
	5	53,735303	54,405376	59,44
M-Graph 200 (N2)	3	41,054257	41,470089	70,13
	4	53,776660	54,759043	61,10
	5	58,157404	59,203274	58,28
M-Graph 200 (N3)	3	31,452245	31,754544	74,57
	4	45,365646	46,139745	63,49
	5	50,372746	51,372021	59,33
M-Graph 300	3	60,105370	60,273181	69,33
	4	73,083044	73,660283	62,45
	5	77,020954	78,126577	60,05
M-Graph 400	3	93,564608	93,756509	65,94
	4	106,776948	107,431808	60,91
	5	115,085364	115,990491	57,97
M-Graph 500	3	128,783529	128,880621	62,85
	4	137,424813	137,956787	60,58
	5	146,288269	146,835473	58,12

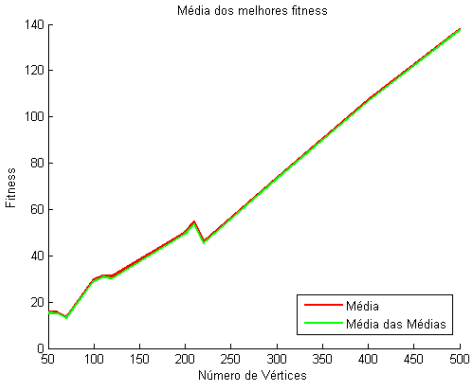
A Tabela 8.5 apresenta também a melhora obtida entre a média dos valores de *fitness* dos melhores indivíduos das populações finais e iniciais. Observa-se que o H-GA obteve, em média, uma melhora de 60% dos valores de *fitness*. A maior melhora ocorreu para o caso M-Graph 200 (N3) com restrição de grau 3, com um ganho de $74,57\%$. O pior ganho foi para o caso M-Graph 100 (N1) com restrição de grau 5, de $46,56\%$. Novamente

os resultados mostram que o H-GA tem capacidade otimização, conseguindo otimizar o *fitness* das soluções da população inicial, e assim, ter soluções melhores na população final.

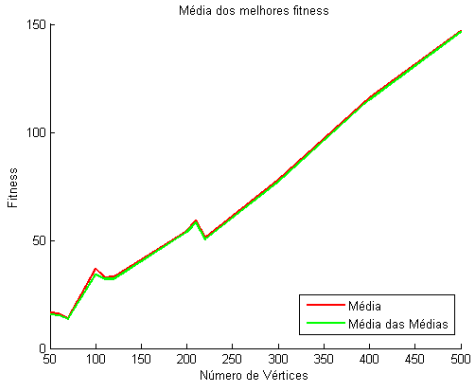
A Figura 8.2 apresenta os gráficos dos *fitness* finais da população, com base nos dados apresentados na Tabela 8.5.



(a) *Fitness da população final para restrição de grau igual a 3.*



(b) *Fitness da população final para restrição de grau igual a 4.*



(c) *Fitness da população final para restrição de grau igual a 5.*

Figura 8.2: *Fitness das populações finais para todos os conjuntos de vértices.*

Conforme ilustra a Figura 8.2, os valores de *fitness* dos melhores indivíduos e da média da população são muitos próximos. Isso indica que o H-GA provavelmente convergiu em seu resultado, confirmando, assim, a sua capacidade de otimização.

Assim como os grafos SHRD, foi aplicado o EANDD nos grafos M-Graphs com as mesmas configurações do H-GA. A Tabela 8.6 apresenta o comparativo entre as médias dos melhores *fitness* e os tempos de execução obtidos pelos EANDD e H-GA. Além disso, são apresentadas as soluções ótimas obtidas em [51]. Os resultados apresentados em [51] são as obtidas por um algoritmo exato de *branch and cut*. Em [51] foi apresentado

apenas os resultados para a restrição de grau igual a 5. Portanto, a Tabela 8.6 apresenta os resultados apenas para essa restrição.

Tabela 8.6: *Comparação dos fitness obtidos na implementação em software e hardware.*

Vértices	Ótimo	H-GA		EANDD	
		Fitness	Tempo (s)	*Fitness*	Tempo (s)
M-Graph 50 (N1)	6,60	15,74	0,00008877	8,92	0,3456
M-Graph 50 (N2)	5,78	15,28	0,00008873	8,22	0,3796
M-Graph 50 (N3)	5,50	13,67	0,00008882	6,44	0,4924
M-Graph 100 (N1)	11,08	34,19	0,0001368	14,13	1,0826
M-Graph 100 (N2)	11,33	32,10	0,0001366	16,01	1,1544
M-Graph 100 (N3)	10,19	32,14	0,0001365	15,02	1,0302
M-Graph 200 (N1)	18,33	53,73	0,0001286	23,06	2,9034
M-Graph 200 (N2)	19,16	58,15	0,0001283	26,01	3,1752
M-Graph 200 (N3)	16,13	50,37	0,0001281	17,83	2,4270
M-Graph 300	40,69	77,02	0,0001346	50,83	4,1314
M-Graph 400	54,69	115,08	0,0001574	78,27	6,5264
M-Graph 500	79,34	146,28	0,0001624	108,27	10,0430

Conforme mostra a Tabela 8.6, mais uma vez os resultados obtidos via *software* foram melhores do que os obtidos via *hardware* para os valores de *fitness*. Porém, o tempo de execução utilizado pelo H-GA é notadamente inferior ao EANDD. Esse resultado ilustra os benefícios do paralelismo do FPGA.

Tabela 8.4: *Fitness das populações iniciais.*

Vértices	Grau	Média dos Melhores Indivíduos	Média da Média da População
M-Graph 50 (N1)	3	33,532100	35,037050
	4	33,928099	35,648959
	5	34,600600	36,252676
M-Graph 50 (N2)	3	33,982001	35,448068
	4	33,943301	36,165926
	5	34,729101	36,463159
M-Graph 50 (N3)	3	27,685900	29,872291
	4	28,165500	30,091941
	5	27,936349	30,122167
M-Graph 100 (N1)	3	64,198653	67,014791
	4	65,028504	67,673571
	5	65,814508	68,811055
M-Graph 100 (N2)	3	69,157454	70,996711
	4	68,467255	71,403382
	5	68,409600	71,095687
M-Graph 100 (N3)	3	67,668103	70,415674
	4	68,563404	71,005250
	5	68,883507	71,307852
M-Graph 200 (N1)	3	128,474296	132,410081
	4	129,768359	134,072981
	5	131,065349	134,121704
M-Graph 200 (N2)	3	134,854394	138,821591
	4	137,006361	140,766279
	5	138,144505	141,913594
M-Graph 200 (N3)	3	120,946440	124,859637
	4	122,989155	126,366828
	5	123,155056	126,323445
M-Graph 300	3	190,947602	196,509388
	4	190,938630	196,159221
	5	191,694432	195,583767
M-Graph 400	3	269,505777	275,242086
	4	268,730923	274,809280
	5	270,299383	275,982394
M-Graph 500	3	341,528491	346,925732
	4	345,260962	349,965060
	5	344,672815	350,602059

CAPÍTULO 9

Conclusão

Este trabalho propõe o desenvolvimento de um algoritmo genético (GA) em *hardware* com a representação nó-profundidade-grau para problemas de projeto de redes (PPRs). Primeiramente, foi introduzida uma revisão sobre PPRs. A seguir, foram apresentados conceitos sobre diferentes EAs, em especial o GA como método proposto para ser utilizado na resolução do PPR da árvore geradora mínima com restrição de grau (dc-MSTP).

O GA desenvolvido utilizou da representação NDDE, objetivando a redução da complexidade de tempo para $O(\sqrt{n})$. Para se obter soluções factíveis, utilizou-se o PAO como operador de mutação. O GA foi implementado na arquitetura de *hardware* utilizando o processador Nios II de uma placa FPGA. Para um melhor entendimento dos benefícios da utilização dessa arquitetura, foram introduzidos conceitos sobre *hardware* reconfigurável. O conjunto de testes utilizado é composto de dois tipos de grafos que representam florestas de *benckmark*. Os resultados obtidos com a aplicação do GA foram apresentados e discutidos. Para ambos os tipos de grafos, observou-se uma melhora no *fitness*. Essa melhora se mostrou na redução nos pesos das florestas utilizadas para o problema. Observa-se que o algoritmo é capaz de resolver o dc-MST em um tempo de processamento limitado por uma constante $O(1)$. Além disso, esse tempo de processamento é inferior ao tempo necessário pelo EANDD, implementação em *software*. Isso é possível pela propriedade de paralelização do FPGA. Assim, a utilização de EAs em *hardware* se mostra viável para soluções de PPRs.

Como trabalhos futuros, sugere-se a alteração do módulo HP-RNP para aceitação de florestas com apenas uma árvore, assim o operador PAO seria aplicado de forma mais adequada para PPRs de *benckmark* que possuem essa característica. Tal alteração pode ocasionar melhora do desempenho do GA uma vez que, a cada aplicação do operador PAO, não será necessário a divisão e, posteriormente, a conexão da floresta. Visto que pode ser exatamente essa aresta que precisa ser alterada na solução para obter uma melhora do indivíduo e a mesma não será permitida devido ao processo de divisão.

Também sugere-se outra alteração na arquitetura com objetivo de permitir a utilização de valores de pesos maiores para as arestas. Essa solução pode ser viabilizada

por meio de estruturas de 64 bits, por exemplo. Além disso, com o objetivo de melhorar os resultados, outra proposta seria a implementação de algoritmos eficientes para escolha dos vértices p e a utilizados no operador PAO. Também, sugere-se a alteração do operador de seleção do H-GA para que seja permitido a inserção de indivíduos ruins. Outras sugestões são o uso de operadores de cruzamento, como o *Evolutionary History Recombitation* (EHR) e o desenvolvimento de outros modelos de algoritmos evolutivos [14]. Como última sugestão, também com o de objetivo melhorar a capacidade de otimização do H-GA, propõem-se uma análise do gerador de número aleatórios utilizado no HP-RNP, de forma a garantir a variabilidade na escolha dos vértices p e a.

Referências Bibliográficas

[1] ABUALI, F. N.; WAINWRIGHT, R. L.; SCHOENEFELD, D. A. **Determinant factorization: A new encoding scheme for spanning trees applied to the probabilistic minimum spanning tree problem**. In: *Proceedings of the Sixth International Conference on Genetic Algorithms*, p. 470–477, 1995.

[2] AUSIELLO, G. **Complexity and approximation: Combinatorial optimization problems and their approximability properties**. Springer, 1999.

[3] BÄCK, T. **Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms**, volume 996. Oxford university press Oxford, 1996.

[4] BÄCK, T.; FOGEL, D. B.; MICHALEWICZ, Z. **Evolutionary computation 1: basic algorithms and operators**, volume 2. IOP Publishing Ltd, 2000.

[5] BROCK, B.; JR., W. A. H.; YOUNG, W. D. **Introduction to a formally defined hardware description language**. In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, p. 3–35, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[6] BROWN, S. D.; VRANESIC, Z. G. **Fundamentals of digital logic with VHDL design**, volume 6. McGraw-Hill New York, 2000.

[7] COMPTON, K.; HAUCK, S. **An introduction to reconfigurable computing**. *IEEE Computer, Apr*, 2000.

[8] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos Teoria e Prática - Tradução da 2ª Edição Americana**. Elsevier, 2ª edição edition, 2002.

[9] D. S. JOHNSON, J. K. LENSTRA, A. H. G. R. K. **The complexity of the network design problem**. *Networks*, 9:279–285, 1978.

[10] DA COSTA, L. A. A. F. **Algoritmos Evolucionários em Optimização Uni e Multi-objectivo**. PhD thesis, Universidade do Minho, Março 2003. Braga, Portugal.

[11] DA COSTA FILHO, P. A.; POPPI, R. J. **Algoritmo genético em química**. *Química Nova*, 22(3):405, 1999.

[12] DAVE, N. **Designing a reorder buffer in bluespec**. In: *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, p. 93–102. IEEE, 2004.

[13] DAVIS, L.; ORVOSH, D.; COX, A.; QIU, Y. **A genetic algorithm for survivable network design**. In: *Proceedings of the 5th International Conference on Genetic Algorithms*, p. 408–415, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[14] DE LIMA, T. W. **Estruturas de Dados Eficientes para Algoritmos Evolutivos Aplicados a Projeto de Redes**. PhD thesis, Universidade de São Paulo - USP São Carlos, Abril 2009.

[15] DE LIMA, T. W.; ROTHLAUF, F.; DELBEM, A. C. B. **The node-depth encoding: Analysis and application to the bounded-diameter minimum spanning tree problem**. *ACM*, p. 969–976, July 2008. Atlanta, Georgia, EUA.

[16] DELBEM, A. **Aumento de eficiência de soluções computacionais para problemas complexos e metodologia de modelagem de sistemas**. *USP-São Carlos. Monografia de Livre-Docência. Apêndice E*, 2009.

[17] DELBEM, A.; DE CARVALHO, A.; POLICASTRO, C. A.; PINTO, A. K.; HONDA, K.; GARCIA, A. C. **Node-depth encoding for evolutionary algorithms applied to network design**. In: *Genetic and Evolutionary Computation–GECCO 2004*, p. 678–687. Springer, 2004.

[18] DELBEM, A. C. B.; DE CARVALHO, A. C.; BRETAS, N. G. **Main chain representation for evolutionary algorithms applied to distribution system reconfiguration**. *Power Systems, IEEE Transactions on*, 20(1):425–436, 2005.

[19] DELBEM, A. C.; DE LIMA, T. W.; TELLES, G. P. **Efficient forest data structure for evolutionary algorithms applied to network design**. 2012.

[20] DIESTEL, R. **Graph Theory {Graduate Texts in Mathematics; 173}**. Springer-Verlag Berlin and Heidelberg GmbH & Company KG, 2000.

[21] EISNER, J. **State-of-the-art algorithms for minimum spanning trees-a tutorial discussion**. 1997.

[22] EVEN, S. **Graph algorithms**. Computer Science Press, 1979.

[23] FOGEL, D. B. **Evolutionary computation: toward a new philosophy of machine intelligence**, volume 1. Wiley, 2006.

[24] GABRIEL, P. H. R.; DELBEM, A. C. B. **Fundamentos de algoritmos evolutivos**, 2008.

[25] GOIS, M. M. **Representação nó-profundidade em fpga para algoritmos evolutivos aplicados ao projeto de redes de larga-escala**. Master's thesis, Universidade de São Paulo - USP São Carlos, Outubro 2011.

[26] GOLDBERG, D. E.; HOLLAND, J. H. **Genetic algorithms and machine learning**. *Machine learning*, 3(2):95–99, 1988.

[27] GOLDBERG, D. E. **The Design of Innovation: Lessons from and for Competent Genetic Algorithms**, volume 7. Springer, 2002.

[28] GOTTLIEB, J.; JULSTROM, B. A.; RAIDL, G. R. **Prüfer numbers: A poor representation of spanning trees for evolutionary search**. In: *In*, p. 343–350. Morgan Kaufmann, 2001.

[29] GROSS, J. L.; YELLEN, J. **Handbook of Graph Theory**. CRC Press, 2004.

[30] HOLLAND, J. H. **Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.** U Michigan Press, 1975.

[31] JONG, K. A. D. **Evolutionary computation: a unified approach**, volume 262041944. MIT press Cambridge, 2006.

[32] JULSTROM, B. A. **The blob code: A better string coding of spanning trees for evolutionary search**. In: *Genetic and Evolutionary Computation Conference Workshop Program*, p. 256–261, 2001.

[33] KNOWLES, J.; CORNE, D.; OATES, M. **A new evolutionary approach to the degree constrained minimum spanning tree problem**. *IEEE Trans. Evolutionary Computation*, 4(2):125–134, 2000.

[34] KRISHNAMOORTHY, M.; ERNST, A. T.; SHARAIHA, Y. M. **Comparison of algorithms for the degree constrained minimum spanning tree**. *Journal of heuristics*, 7(6):587–611, 2001.

[35] KRUSKAL, J. B. **On the shortest spanning subtree of a graph and the traveling salesman problem**. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[36] LIDEN, R. **Algoritmos Genéticos, Uma importante ferramenta da Inteligência Computacional**. Brasport, 2ª edição edition, 2006.

[37] LIN, C.-C. **Implementation of H. 264 Decoder in Bluespec SystemVerilog**. PhD thesis, Massachusetts Institute of Technology, 2007.

[38] MEDEIROS, S. C. D. **Inversão de parâmetros em dados sísmicos por algoritmo genéticos**. Master's thesis, Universidade Federal do Rio de Janeiro, 2005.

[39] MICHALEWICZ, Z. **Genetic algorithms + data structures = evolution programs**. Springer, 1996.

[40] NARULA, S. C.; HO, C. A. **Degree-constrained minimum spanning tree**. *Computers & Operations Research*, 7(4):239–249, 1980.

[41] NIKHIL, R. **Bluespec system verilog: efficient, correct rtl from high level specifications**. In: *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, p. 69–70. IEEE, 2004.

[42] NIKHIL, R. S.; CZECK, K. R. **Bsv by example**. *CreateSpace, Dec*, 2010.

[43] PACHECO, M. A. C. **Algoritmos genéticos: princípios e aplicações**. *ICA: Laboratório de Inteligência Computacional Aplicada. Departamento de Engenharia Elétrica. Pontifícia Universidade Católica do Rio de Janeiro. Fonte desconhecida*, 1999.

[44] PALMER, C. C. **An approach to a problem in network design using genetic algorithms**. PhD thesis, Brooklyn, New York, USA, 1994.

[45] PALMER, C. C.; KERSHENBAUM, A. **An approach to a problem in network design using genetic algorithms**. *Networks*, 26(3):151–163, 1995.

[46] PALNITKAR, S. **Verilog HDL: a guide to digital design and synthesis**. Prentice Hall, 2003.

[47] PILA, A. D. **História e terminologia a respeito da computação evolutiva**. *Revista de Ciências Exatas e Tecnologia*, 2006. Faculdade Comunitária de Santa Bárbara.

[48] PIÓRO, M.; MEDHI, D. **Routing, flow, and capacity design in communication and computer networks**. Elsevier, 2004.

[49] PRIM, R. C. **Shortest connection networks and some generalizations**. *Journal of Bell Systems and Technology*, 36:1398–1401, 1957.

[50] RAIDL, G. R. **An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem**. In: *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, p. 104–111. IEEE, 2000.

[51] RAIDL, G. R.; JULSTROM, B. A. **Edge sets: an effective evolutionary coding of spanning trees**. *Evolutionary Computation, IEEE Transactions on*, 7(3):225–239, 2003.

[52] RECHENBERG, I. **Cybernetic solution path of an experimental problem**. 1965.

[53] RIDLEY, M. **Evolution**. Pheonix, 2001.

[54] ROTHLAUF, F. **Representations for genetic and evolutionary algorithms**. Springer, 2006.

[55] ROTHLAUF, F.; GOLDBERG, D. E.; HEINZL, A. **Network random keys: A tree representation scheme for genetic and evolutionary algorithms**. *Evolutionary Computation*, 10(1):75–97, 2002.

[56] SCHINDLER, B.; ROTHLAUF, F.; PESCH, H.-J. **Evolution strategies, network random keys, and the one-max tree problem**. *UK: Springer-Verlag*, p. 143–152, 2002. Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002.

[57] SCHINDLER, B.; ROTHLAUF, F.; PESCH, H.-J. **Evolution strategies, network random keys, and the one-max tree problem**. In: *Applications of Evolutionary Computing*, p. 143–152. Springer, 2002.

[58] SCHWEFEL, H.-P. **Kybernetische evolution als strategie der experimentellen forschung in der strömungstechnik**. *Master's thesis, Technical University of Berlin*, 1965.

[59] TANOMARU, J. **Motivação, fundamentos e aplicações de algoritmos genéticos**. II Congresso Brasileiro de Redes Neurais, III Escola de Redes Neurais, 1995.

[60] TOTH, P.; VIGO, D. **The vehicle routing problem**, volume 9. Siam, 2002.

[61] ZBIGNIEW, M.; FOGEL, D. B. **How to solve it: Modern Heuristics**. Springer New York, 2004.

[62] ZHOU, G.; GEN, M. **A note on genetic algorithm for degree-constrained spanning tree problems.** *Networks (USA)*, 30(2):91–95, 1997.

[63] ZUBEN, F. J. V. **Computação evolutiva: uma abordagem pragmática**. *Anais da I Jornada de Estudos em Computação de Piracicaba e Região (1a JECOMP)*, 1:25–45, 2000.