Instituto de Ciências Matemáticas e de Computação

Estruturas de Dados Eficientes para Algoritmos Evolutivos Aplicados ao Projeto de Redes

Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

ESTRUTURAS DE DADOS EFICIENTES PARA ALGORITMOS EVOLUTIVOS APLICADOS AO PROJETO DE REDES

TELMA WOERLE DE LIMA
ALEXANDRE CLÁUDIO BOTAZZO DELBEM

Nº 301

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Outubro/2007

Sumário

1 Introdução **3**

2 Grafos e Árvores **4**

2.1 Percursos em Árvores e Grafos 5

2.2 Algoritmos de Árvore Geradora Mínima 7

3 Introdução a Projeto de Redes **7**

3.1 Árvore Geradora de Comunicação Ótima 8

3.2 Árvore Geradora Mínima Probabilística 9

3.3 Árvore Geradora Mínima com Restrição de Grau 10

3.4 Uma Árvore Máxima 10

3.5 Roteamento de veículos 10

4 Algoritmos Evolutivos **12**

5 Estruturas de Dados para AEs Aplicados a PPRs **17**

5.1 Vetor de Características 18

5.2 Número de Prüfer 19

5.3 Tendência de Ligação e Nó 20

5.4 Predecessores ou Codificação Determinante 21

5.5 Precedentes Diretos 23

5.6 Conjunto de Arestas 25

5.7 *Blob Code* 30

5.8 Chaves Aleatórias para Redes 31

5.9 Permutação Baseada em Árvore 33

5.10 *D-Based* 34

5.11 Ajuste Adaptativo das Ligações 35

5.12 Nó-Profundidade 35

5.12.1 Operador 1 36

5.12.2 Operador 2 37

5.12.3 Determinação dos nós p, r e a 37

5.13 Sub-Conjunto de Comprimento Fixo 38

5.14 *Dandelion Code* 39

6 Estrutura de Dados Proposta: NDDE **41**

6.1 Operadores de Recombinação 41

6.2 Melhorias nos Operadores 1 e 2 da NDE 46

6.2.1 Obtenção Eficiente dos Nós p, a e r 47

6.2.2 Análise de Complexidade 48

7 Resultados preliminares **51**

8 Considerações finais **55**

1 Introdução

Os algoritmos evolutivos (AEs) [31, 9] são ferramentas poderosas de otimização inspiradas na evolução natural e tem sido aplicadas a muitos problemas nas mais diversas áreas do conhecimento humano. Os AEs simulam a evolução de uma população de indivíduos que são possíveis soluções para um problema. No processo evolutivo as soluções que são mais adaptadas possuem maior probabilidade de sobreviver ou transmitir suas características para gerações futuras.

AEs tem sido adequados para complexos problemas de otimização envolvendo características que tornam difícil o uso de métodos clássicos de otimização [16]. Além disso, AEs não possuem algumas dificuldades principais dos algoritmos clássicos, tais como dependência de encontrar uma solução factível como solução inicial, tendência para encontrar soluções subótimas em vez de encontrar o ótimo global, perda de eficiência quando ocorrem mudanças no problema de otimização, ineficiência dos algoritmos para espaços de busca discretos e inadequação dos algoritmos para máquinas paralelas [16].

No entanto, AEs com codificações convencionais tem sido ineficientes para problemas de projeto de redes (PPRs) [14], em especial para grandes sistemas. Essas abordagens evolutivas geram muitos componentes desconexos ou grafos acíclicos quando aplicados para grandes sistemas, enquanto que em PPRs geralmente busca-se a produção de árvores (ou florestas) geradoras. Dependendo da codificação adotada, a produção de tais grafos pode consumir uma grande parte do tempo de execução, o qual reduz a eficiência dos AEs. Além disso, as redes produzidas podem ser muito diferentes de seus pais, reduzindo drasticamente a convergência dos AEs.

Por outro lado PPRs são importantes pois envolvem problemas do mundo real de diversas áreas de engenharia e ciências, tais como circuitos elétricos, roteamento de veículos, redes de computadores e árvores filogenéticas. Com o objetivo de melhorar a eficiência dos AEs para PPRs, pesquisas têm proposto novas codificações [45, 25, 11, 24, 26, 51, 37, 36], as quais têm produzido aumento significativo da eficiência dos AEs. Uma dessas codificações é a representação nó-profundidade [17] que tem como vantagem a capacidade de trabalhar com redes correspondendo a florestas.

Em geral, a aplicação de AEs utilizando a representação nó-profundidade tem apresentado resultados satisfatórios para problemas de projeto de redes. Este trabalho investiga a aplicação de AEs utilizando a representação nó-profundidade para problemas de projeto de redes, o aperfeiçoamento da representação nó-profundidade pelo desenvolvimento de novos operadores de reprodução, bem como a análise de complexidade desses operadores.

Este trabalho está organizado da seguinte forma: A Seção 2 introduz conceitos de teoria de grafos e árvores. A Seçao 3 aborda o PPR e seus principais tipos presentes na literatura. A Seção 4 traz os principais conceitos da teoria de AEs. A Seção 5 apresenta as representações de soluções de PPRs para AEs presentes na literatura. A Seção 6 descreve

a estrutura de dados NDDE uma nova representação de soluções para AEs aplicados a PRs, seus novos operadores e as modificações feitas na NDE. Os resultados iniciais de um AE utilizando a NDDE são apresentados na Seção 7. A Seção 8 traz as considerações finais deste trabalho.

2 Grafos e Árvores

Os conceitos de grafos e árvores são muito importante para a formalização de algumas definições na área de projeto de redes (PRs). Um grafo é uma estrutura composta por um conjunto finito e não vazio denominado nós ou vértices e um conjunto também finito de pares de nós ou vértices denominado conjunto de arestas ou ligações, sendo representado por $G = (V, E)$, onde V é o conjunto de nós e E o conjunto de arestas. A ordem de um grafo é representada por n, que é o número de nós do grafo ($n = |V|$). O tamanho do grafo é representado por m, que é o número de arestas ($m = |E|$) [28, 35, 67, 27].

Um grafo pode ter orientação ou não em suas arestas. O grafo que possui orientação nas arestas é dito grafo orientado, o conjunto das arestas é composto por pares ordenados e as arestas são chamadas de arcos. Nos grafos não orientados, ou seja, sem orientação nas arestas, é chamado somente de grafo e no conjunto de arestas os pares (x, y) e (y, x) representam a mesma aresta.

O grau de um nó do grafo é definido pelo número de arestas que incidem nele. Uma aresta é dita incidente em um nó se um de seus extremos é o nó. Para grafos orientados além do grau total do grafo, existe o grau de saída do nó, que é o número de arestas que possuem este nó como o primeiro nó no par ordenado; e o grau de entrada do nó, o número de arestas que possuem este nó como segundo nó no par ordenado. Um nó v_i é dito adjacente a um nó v_j se existe a aresta (v_i, v_j) [28].

Um caminho em um grafo de um nó v_i para um nó v_j é definido pela seqüência de nós $\{v_i, v_{i+1}, v_{i+2}, \ldots, v_{j-1}, v_j\}$ que possui as arestas $\{(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{j-1}, v_j)\}$. Um ciclo é um caminho no qual $v_i = v_j$. O tamanho de um caminho ou ciclo é determinado pelo número de arestas que compõem o caminho [67, 27].

Um grafo é dito conexo se para todos os pares de nós $v_i, v_j \in V$ existe um caminho entre eles. Uma componente de G é o maior subgrafo conexo de G. Um grafo é completo se todo par distinto de nós é unido por uma aresta, ou seja, existe uma aresta $e \in E$ para todo par distinto de nós. Em um grafo completo, o número de arestas é dado por $m = n(n-1)/2$.

Um dos conceitos de grafos mais importantes para PRs é o conceito de árvores. Uma árvore (T) é um grafo conexo que não contém ciclos. Uma árvore enraizada é uma árvore que contém um nó distinguível que recebe o nome de r chamado raiz de T. O número de arestas de uma árvore é $m = n - 1$ [28, 35, 27].

A profundidade de um nó na árvore é o tamanho do caminho único da raiz da árvore

até o nó. A profundidade da árvore é a maior profundidade de todos os seus nós. Um nó v_i é o pai ou raiz de um nó v_j, considerando a orientação de r até v_j, se existe a aresta (v_i, v_j), ou seja, no caminho de r até v_j, v_i é o nó diretamente ligado a v_j no caminho.

A árvore geradora de um grafo $G = (V, E)$ é a árvore $T = (W, S)$, onde $W = V$, ou seja, a árvore é um subgrafo de G que contém todos os vértices de G e não contém ciclos [67].

Os nós e as arestas de um grafo podem conter valores associados, os valores associados aos nós são geralmente o rótulo do nó e os valores associados as arestas são o seu custo. Uma árvore geradora é dita árvore geradora mínima se a soma dos custos associados às arestas é o menor possível dentre todas as árvores geradoras do grafo. Os nós da árvore que possuem grau um são chamados de nós folha.

Existem duas formas principais de representar um grafo a matriz de adjacências e a lista de adjacências. Na matriz de adjacências, o grafo é representado por uma matriz de n linhas por n colunas, onde cada linha e coluna é o índice de um vértice, cada aresta (v_i, v_j) é representada pela atribuição do valor 1 ou do custo da aresta na posição $a_{i,j}$ da matriz. A lista de adjacências consiste em construir uma lista encadeada [3] para cada nó contendo os nós adjacentes a ele.

2.1 Percursos em Árvores e Grafos

Os percursos determinam uma ordem em que todos os nós da árvore ou do grafo são percorridos, por exemplo, em um processo de busca. Os principais percursos em árvores são pré-ordem, pós-ordem, *top-down* e *bottom-up* [35, 67].

No percurso em pré-ordem de uma árvore $T = (W, S)$, os nós são percorridos a partir da raiz de T, em seguida é visitada a raiz do primeiro filho de r. O algoritmo segue visitando as raízes dos filhos da última raiz visitada até chegar no nó folha. Ao chegar neste tipo de nó, o processo retorna para o próximo filho do pai do nó folha. Após percorrer todos os filhos da raiz da sub-árvore terem sido visitados, o algoritmo retorna para visitar os filhos da raiz da sub-árvore acima, até que todos os nós tenham sido percorridos. O Algoritmo 2.1 descreve os passos do percurso em pré-ordem [35, 67].

Algoritmo 2.1 Percurso em Pré-ordem.

(1) Se a árvore for vazia pare;
(2) Senão visite a raiz;
(3) Percorra o filho mais a esquerda não visitado em pré-ordem;
(4) Se a árvore tiver filhos volte ao passo (3);

O algoritmo recursivo em pré-ordem tem complexidade de tempo $O(n)$ para uma árvore de n nós.

No percurso em pós-ordem de uma árvore $T = (W, S)$, os nós são percorridos a partir dos nós folha de cada sub-árvore da árvore sendo que a raiz da sub-árvore (árvore), só é

visitada depois que todos os seus filhos são visitados. O Algoritmo 2.2 descreve os passos do percurso em pós-ordem [35, 67].

Algoritmo 2.2 - Percurso em Pós-ordem.

(1) Se a árvore for vazia pare;
(2) Senão Percorra o filho mais a esquerda não visitado em pré-ordem;
(3) Se a árvore tiver filhos volte ao passo (3);
(4) Senão visite a raiz;

O algoritmo recursivo em pós-ordem também possui complexidade de tempo $O(n)$ para uma árvore de n nós.

No percurso *top-down*, os nós da árvore são visitados em ordem crescente de profundidade, todos os nós de uma mesma profundidade são visitados da esquerda para direita. No percurso *bottom-up*, os nós começam a ser percorridos pelas folhas e em ordem decrescente de profundidade, nós folha na mesma profundidade são percorridos da esquerda para a direita. Ambos os algoritmos podem ser implementados de forma recursiva e possuem complexidade de tempo $O(n)$ [67].

Os principais percursos em grafos (árvores ou não) são busca em profundidade e busca em largura [67]. O percurso de busca em profundidade de um grafo $G = (V, E)$ com n nós consiste em a partir de um nó aleatório marcá-lo como visitado e visitar o primeiro de seus adjacentes, marcando-o. O processo segue recursivamente, sempre visitando o primeiro adjacente não visitado do último nó visitado, se o ultimo nó visitado não possui nenhum adjacente retorna ao nó anterior e verifica se este possui ainda algum adjacente a ser visitado. O algoritmo repete este processo até que todos os nós tenham sido percorridos. O Algoritmo 2.3 descreve os passos para efetuar uma busca em profundidade [67].

Algoritmo 2.3 - Busca em Profundidade.

(1) Selecione o nó inicial r;
(2) Marque r;
(3) Adicione r a pilha;
(4) Enquanto a pilha for não vazia faça
    (5) $u \leftarrow$ copia o topo da pilha;
    (6) Se adjacentes $[u]$ contém v não marcado
        (7) Marque v e adicione v a pilha;
        (8) Retorna ao passo (4);
    (9) Senão remova u da pilha e retorna ao passo (4)

O percurso obtido por uma busca em largura em um grafo $G = (V, E)$, começa visitando um nó aleatório e, em seguida, percorre todos os seus adjacentes ainda não visitados. Recursivamente, percorre os adjacentes não visitados dos nós já visitados na ordem em que foram visitados até que todos os nós do grafo tenham sido visitados. Os passos para efetuar uma busca em largura estão descritos no Algoritmo 2.4 [67].

Ambos os algoritmos de busca em profundidade e busca em largura possuem complexidade de tempo $O(n + m)$, pois necessitam percorrer todos os vértices e, para cada nó n, verificar todos os seus vizinhos, isto é, percorrer as m arestas do grafo no total [67].

Algoritmo 2.4 - Busca em Largura.

(1) Selecione o nó inicial r;
(2) Marque r;
(3) Marque todos adjacentes a r e adicione a fila;
(4) Enquanto a fila for não vazia faça
  (5) $u \leftarrow$ copia o primeiro nó da fila;
  (6) Marque todos adjacentes a u não marcados e adicione a fila
  (9) Remova u da fila e retorna ao passo (4)

2.2 Algoritmos de Árvore Geradora Mínima

Os algoritmos de Prim e Kruskal [28, 67, 3] são utilizados para encontrar uma árvore geradora mínima $T = (V, E_T)$ de um grafo $G = (V, E)$ tal que a soma dos pesos de E_T seja mínima.

No algoritmo de Prim primeiramente escolhe um nó v aleatoriamente. Em seguida, seleciona a aresta de menor custo ligada a v. Adiciona o outro extremo da aresta a um conjunto de nós V_T. Até que sejam inseridos todos os nós em V_T, o algoritmo analisa as arestas dos nós já inseridos e seleciona a aresta de menor peso não considerada ainda. Após todos os nós terem sido inseridos, o resultado é uma árvore geradora mínima do grafo. A complexidade de tempo do algoritmo de Prim é $O(mlogn)$ [28].

Diferente do algoritmo de Prim, o algoritmo de Kruskal ordena as arestas do grafo de forma crescente e seleciona a aresta de menor peso do grafo. Até que sejam aceitas $n - 1$ arestas, o algoritmo segue testando as arestas em seqüencia, se a aresta não formar um ciclo adiciona a aresta a E_T e caso contrário, testa a próxima aresta. Ao terminar o processo iterativo, obtém-se a árvore geradora mínima do grafo. A complexidade de tempo do algoritmo de Kruskal é $O(mlogn)$ [28].

3 Introdução a Projeto de Redes

O problema de projeto de redes (PPRs) consiste em encontrar um sub-grafo de um grafo não direcionado com pesos, sujeito a uma restrição de orçamento na soma dos pesos das suas arestas, que minimiza o comprimento dos caminhos entre todos os pares de pontos [30].

A formulação matemática do problema é descrita a seguir. Um PPR consiste em dado um grafo $G = (V, E)$, uma função de pesos $L : E \rightarrow N$, um orçamento B e um critério de limiar C $(B, C \in N)$, existe um subgrafo de $G' = (V, E')$ de G com pesos tais que $\sum_{\{i,j\} \in E'} L(\{i, j\}) \leq B$ e o valor de critério $f(G') \leq C$, onde $f(G')$ é a soma dos comprimentos dos menores caminhos em G' entre todos os pares de nós [30].

Johnson et. al [30] mostraram que o PPR genérico é um problema $NP - completo$, assim como as suas derivações.

Os principais problemas conhecidos da área de PRs são árvore geradora de comu-

nicação ótima [44], árvore geradora de ótimo custo de ligação [44], árvore geradora de requerimento ótimo [44], problema de fonte única [44], sistema de distribuição elétrica [11, 19], árvore geradora mínima probabilística [1], árvore geradora mínima com restrição de grau [54], redes de telecomunicações [2, 49, 61], árvores de Steiner [29], roteamento de veículos [39, 68] entre outros.

Os algoritmos clássicos de árvore geradora mínima de Prim e Kruskal têm sido utilizados na construção de soluções para PRs. Apesar de eles escolherem as arestas de menor custo para compor a árvore esses algoritmos não são adequados para redes de larga-escala [8], obtendo para alguns pontos da rede caminhos muito longos, que ocasionam problemas de confiabilidade da rede, além de dificultarem o tratamento de restrições dos PPRs [8]. Devido a essas dificuldades são necessários o uso de metaheurísticas[1] ou relaxações nos problemas para que estes possam ser resolvidos em tempo de execução adequado. [49] apresenta a formulação matemática de PPRs para redes de comunicação, suas técnicas de modelagem e as principais metaheurísticas utilizadas na solução desses problemas, que serão estudadas durante o desenvolvimento do presente projeto de pesquisa para comparação entre o algoritmo proposto e os algoritmos já utilizados.

A seguir serão definidos os principais PPRs encontrados na literatura e outros problemas envolvendo a obtenção de árvores em grafos.

3.1 Árvore Geradora de Comunicação Ótima

O problema de árvore geradora de comunicação ótima (OCSTP do inglês, *Optimal Communication Spanning Tree Problem*) consiste em encontrar uma árvore geradora que conecta todos os nós e satisfaz as necessidades de comunicação entre os pontos para um custo total mínimo. O número e a posição dos nós da rede são dados a priori e o custo da rede é o custo da ligação dos nós. Como outros problemas de árvore geradora com restrição o OCSTP é NP-difícil [56].

Este problema é definido formalmente como segue. Dado um conjunto de n cidades, representado por um grafo não direcionado $G = (V, E)$, com $|V| = n$. O custo por unidade de ligação de comunicação entre cada par de cidades é representado pela matriz C com n linhas e n colunas, onde cada elemento $c_{i,j}$ é o custo da ligação entre as cidades i e j. O conjunto de demandas é também uma matriz R com n linhas e n colunas onde, seus elementos $r_{i,j}$ representam a largura de banda necessária entre todos os pares de cidades. Deve-se encontrar a árvore geradora conectando essas cidades que atende a demanda de

[1]Heurísticas são algoritmos ou métodos exploratórios para definição de problemas em que as soluções são encontradas por métodos de busca em que o progresso é obtido pela avaliação empírica (ou com base em conhecimento a priori de especialistas do problema) do resultado. Metaheurísticas são heurísticas que propõem métodos de solução genéricos para problemas, em geral de otimização. Podem incluir diferentes heurísticas de construção e busca local, com o objetivo de produzir um processo de busca mais eficiente na solução de problemas de elevada complexidade computacional. Exemplos de metaheurísticas são algoritmos evolutivos, busca tabu, *Simulated Annealing*, algoritmos de formigas e nuvens de partículas.

tráfico de forma mais econômica [44].

A expressão matemática da função objetivo do OCSTP pode ser vista na Equação 1.

$$min_T \sum_{s,d} \left(\sum_{(i,j) \in P_{s,d}(T)} R_{s,d} C_{i,j} \right), \tag{1}$$

onde, T é uma árvore que soluciona o problema com os custos definidos por $c_{i,j}$, e $P_{s,d}(T)$ é o caminho único entre os nós s, d em T. Os nós s, d são os pares de demanda e os nós i, j são os pares de nós no caminho $P_{s,d}(T)$ [44].

Heurísticas são aplicadas para resolver o OCSTP com o uso do relaxamento de restrições do problema [8]. Como uma alternativa as técnicas utilizando heurísticas e relaxamentos pesquisas utilizando AEs para o OCSTP têm sido desenvolvidas e aplicadas a redes com centenas de nós com resultados satisfatórios [61, 56, 57, 26, 32, 44].

3.2 Árvore Geradora Mínima Probabilística

O problema de árvore geradora mínima probabilística (PMSTP, do inglês *Probabilistic Mininum Spanning Tree Problem*) é uma variação probabilística do problema de árvore geradora mínima definido por Bertsimas [6]. É um modelo de PPRs mais próximo dos problemas do mundo real do que o de árvore geradora mínima MSTP (do inglês *Minimun Spanning Tree Problem*). Dado um grafo $G = (V, E)$, com um custo associado a cada aresta, o PMSTP assume que cada subconjunto de nós é ativado com uma probabilidade definida a priori. Para uma árvore T de G, o custo no tempo a é definido pela soma dos custos das arestas de T_a, a sub-árvore de T ativada no tempo a. A solução do PMSTP é encontrar a árvore geradora que tem o menor custo ativado esperado. Bertsimas [6] mostrou que, embora algumas instâncias do PMSTP possam ser resolvidas em tempo polinomial, o PSMTP é NP-Completo [1].

Formalmente PMSTP pode ser definido como segue. Dado um grafo conexo $G = (V, E)$, não necessariamente completo, a função de custo $c : E \rightarrow \mathbb{R}^+$, e a função de probabilidade $p : 2^{|V|} \rightarrow [0, 1]$, o objetivo é encontrar a árvore geradora T, que minimiza o custo ativo esperado, $E[L_T]$, onde:

$$E[L_T] = \sum_{S \subseteq V} p(S) L_{T(S)} \tag{2}$$

a soma é feita para todos os subconjuntos de V, $T(S)$ é a sub-árvore mínima (de acordo com a função c) para conectar todos os nós de S, $L_{T(S)}$ é o custo total das arestas em $T(S)$ e $p(S)$ é a probabilidade dos nós estarem ativos [1].

Segundo Abuali et. al [1] a utilização de AEs para PMSTP obtém melhores soluções do que algoritmos de busca gulosa.

3.3 Árvore Geradora Mínima com Restrição de Grau

O problema de árvore geradora mínima com restrição de grau (dc-MSTP, do inglês *Degree Constrained Minimum Spanning Tree*) é uma extensão do MSTP, para o qual o número de arestas incidentes a um nó não deve ultrapassar um limitante superior de grau [37]. A adição da restrição de grau torna o problema de árvore geradora mínima NP-Difícil [4].

Formalmente o dc-MSTP pode ser definido como segue. Dado $G = (V, E)$ um grafo não orientado. Seja, W a matriz de pesos, com $w_{i,j}$ o peso do par de nós i, j, $d(i)$ o grau do nó i e d a restrição de grau do problema, com $d > 2$. Seja T uma árvore geradora de G. A solução do dc-MSTP é dada por:

$$min_T \sum_{(i,j)\in T} w_{i,j} \tag{3}$$

sujeito a restrição

$$d(i) \leq d, \text{para todo } i \in T$$

Várias pesquisas utilizando AEs para dc-MSTP têm sido desenvolvidas nos últimos anos encontrando soluções melhores que algoritmos baseados em heurísticas. Além disso, o dc-MSTP tem sido utilizado para avaliar o desempenho de representações de PPRs em AEs [17, 54, 26, 38, 37, 69].

3.4 Uma Árvore Máxima

O OTP (do inglês, *One-Max Tree Problem*) foi proposto por [24] como um teste padrão para avaliar o desempenho de algoritmos de otimização para projeto topológico de árvores.

Uma árvore T_{opt} de um grafo $G(V, E)$ é escolhida aleatoriamente como sendo a árvore ótima. Então, para avaliação de uma nova árvore T_b, utiliza-se a distância $d_{opt,b}$ entre as duas árvores T_{opt} e T_b, que é dada por

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

onde, l_{ij}^{opt} é 1 se a aresta entre os nós i e j está em T_{opt} e 0, caso contrário. A qualidade de uma solução (aptidão de um indivíduo) T_i é definida como a distância $d_{opt,i}$ para a solução ótima T_{opt}.

3.5 Roteamento de veículos

O problema de roteamento de veículos (VRP do inglês, *Vehicle Routing Problem*) diz respeito a distribuição de produtos entre depósitos e usuários finais (clientes). Os modelos

apresentados a seguir podem ser utilizados não somente para a solução de problemas de distribuição ou coleta de produtos, mas também, para a solução de outros problemas do mundo real tais como, roteamento de ônibus escolar, limpeza de ruas, roteamento de vendedores e de unidades de manutenção [65].

A solução do VRP consiste na determinação de um conjunto de rotas, uma para cada veículo que começa e termina em seu próprio depósito, de forma que todas as requisições dos clientes são atendidas, todas as restrições operacionais satisfeitas e o custo de transporte global minimizado. No processo de otimização consideram-se os seguintes componentes: rede de estradas representada por um grafo $G = (V, E)$, onde os nós, V, representam junções das estradas, depósitos e clientes; as arestas, E, representam as estradas. A cada aresta (i, j) é associado um custo $c_{i,j}$. Aos veículos é associada sua capacidade de transporte [65].

Os principais tipos de VRPs são [65]:

- Problema de roteamento de veículos com restrição de capacidade e distância (CVRP do inglês, *Capacitaded Vehicle Routing Problem*): todos os clientes correspondem a entregas e as demandas são determinísticas, conhecidas previamente e não podem ser alteradas. Os veículos são idênticos e permanecem todos em um único depósito central e *somente a restrição de capacidade para os veículos é imposta.* O objetivo é *minimizar o custo total*, isto é, uma função de peso do número de rotas e seus comprimentos ou o tempo de viagem para atender todos os clientes;

- Problema de roteamento de veículos com janela de tempo (VRPTW, do inglês, *Vehicle Routing Problem with Time Window*): uma extensão do CVRP no qual *a cada cliente é associado um intervalo de tempo, chamado janela de tempo.* São dados no problema os instantes em que os veículos deixam o depósito, o tempo de viagem para cada arco e um tempo de serviço adicional para cada cliente. *O serviço em cada cliente deve começar dentro da janela de tempo e o veículo deve permanecer no cliente todo o tempo de serviço especificado.* Em caso do veículo chegar ao cliente antes do tempo de início especificado na janela de tempo, é permitido que ele espere no cliente até o tempo de início;

- Problema de roteamente de veículos com retirada (VRPB, do inglês, *Vehicle Routing Problem with Backhauls*): também é uma extensão do CVRP no qual *o conjunto de clientes é particionado em dois subconjuntos disjuntos.* O primeiro subconjunto contém os clientes que requerem uma quantidade de produtos para ser *entregue.* O segundo subconjunto contém os clientes que requerem que uma dada quantidade de produtos seja *retirada.* Esta classe de problema tem uma condição prévia que deve ser respeitada: em uma rota que atenda clientes do primeiro e do segundo subconjunto, todos os clientes do primeiro subconjunto devem ser atendidos antes que qualquer cliente do segundo subconjunto seja atendido;

- Problema de roteamente de veículos com entrega e retirada (VRPPD, do inglês, *Vehicle Routing Problem with Pickup and Delivery*): nesta classe de VRP *cada cliente é associado com duas demandas, uma de entrega e uma de retirada.* Às vezes, somente uma demanda é especificada indicando a diferença entre as demandas de retirada e entrega. Assume-se que em cada cliente a entrega é efetuada antes da retirada. O VRPPD generaliza o CVRP quando as demandas de retirada são nulas.

Diversos algoritmos de otimização têm sido aplicados para a solução de VRPs. Visto que esses problemas são NP-Difíceis [65] tem sido necessária a aplicação de uma série de relaxamentos de restrições para que os algoritmos possam encontrar uma solução para o problema em tempo polinomial.

Algumas metaheurísticas também têm sido aplicadas para VRPs tais como AEs [42], *simulated annealing*, busca tabu [68] e algoritmos de formigas [68]. As abordagens desenvolvidas com essas metaheurísticas têm encontrado soluções ótimas para problemas com algumas centenas de clientes. Dentre essas, a que tem mostrado melhor desempenho, é a busca tabu. Na maioria dos casos essas metaheurísticas demandam mais esforço computacional para solucionar o problema do que as técnicas convencionais de otimização [65].

Com este trabalho tem como objetivo analisar a solução de problemas do mundo real, para PPRs no mundo real deve levar em consideração, além de minimizar a soma do peso dos caminhos entre todos os pares de pontos outras três características [8]:

1. o custo da rede, ou seja, o custo para manter a rede em funcionamento;

2. a performance da rede, qual a quantia de trafego permitida pela rede;

3. a confiabilidade, qual a integridade ou qualidade do percurso da rede.

Essas características devem ser consideradas principalmente em projetos de redes de comunicação [49, 8].

4 Algoritmos Evolutivos

Os Algoritmos Evolutivos (AEs) envolvem um grupo importante de algoritmos de busca probabilística de soluções inspirado no processo de evolução biológica [23, 31]. Esses algoritmos têm sido aplicados principalmente a problemas de otimização global [31]. Os AEs também podem ser utilizados em outras classes de problemas envolvendo, por exemplo, sistemas classificadores [31], ou o projeto de objetos complexos tais como redes neurais, conjuntos de regras para executar tarefas e códigos Lisp [31].

AEs tem como base os conceitos da Teoria de Darwin sobre a evolução e seleção natural das espécies e seus mecanismos de reprodução [23]. Nos AEs, uma população de indivíduos (conjunto de soluções potenciais) precisam ser modelados computacionalmente.

A estrutura de dados utilizada para isso, é por demais importante. Diferentemente de outras técnicas de busca e/ou otimização, que constroem uma única solução por iteração ou consideram apenas uma solução incremental a cada iteração os AEs trabalham com conjuntos de soluções. A forma mais simples de representação de soluções é um *array* binário por indivíduo, chamado cromossomo. Cada posição do cromossomo representa um gene, isto é uma característica da solução do problema. Os valores possíveis que um gene pode assumir são seus alelos. Os novos indivíduos são obtidos pelos operadores de reprodução aplicados aos cromossomos alterando seus genes. O processo de seleção natural utiliza a função de aptidão (avaliação ou *fitness*) dos indivíduos para tomar a decisão sobre seleção dos mesmos.

Os principais AEs são: programação evolutiva (PE), estratégias evolutivas (EEs) e algoritmos genéticos (AGs). Pode-se dizer que os AEs tiveram seu início na década de 30, quando os sistemas evolutivos naturais começaram a ser analisados como algoritmos de exploração de múltiplos picos de uma função de *fitness*. Nas décadas de 60 e 70, houve a formalização e definição dos principais AEs. A partir da década de 80, houve um grande crescimento da aplicação de AEs, principalmente, em problemas de otimização. Desde então a utilização de AEs somente tem crescido atingindo diversas áreas da engenharia e ciências [31].

Um AE simples, chamado EV(μ, λ), contém uma população de indivíduos de tamanho fixo, cada um deles com comprimento fixo, genes com valores em ponto flutuante, uma função de *fitness* e o mecanismo de seleção descrito a seguir. A população inicial é gerada aleatoriamente e avaliada pela função de *fitness*. O EV simula a evolução da população pela repetição dos seguintes passos: 1) seleciona com distribuição aleatória uniforme λ indivíduos (os pais) da população atual (de tamanho μ), com $1 \leq \lambda \leq \mu$; 2) copia os λ pais selecionados e aplica o operador de mutação nas cópias alterando, em média, um gene pela soma ou subtração aleatória de um pequeno valor; 3) cada novo indivíduo compete diretamente pela sobrevivência com um membro da população atual selecionado aleatoriamente, ganha o indivíduo de melhor *fitness* [31]. O algoritmo EV é utilizado como base para a definição dos principais AEs.

A PE simula a evolução de uma população contendo algoritmos com o objetivo de desenvolver a inteligência artificial [23]. Cada indivíduo representa uma máquina de estados finitos, que processa uma seqüência de símbolos. Os indivíduos são avaliados por uma função de *payoff* de acordo com a saída da máquina. Os descendentes são obtidos pela mutação dos pais. Para a próxima geração sobrevivem somente os indivíduos de maior *payoff* entre os pais e os novos descendentes [23]. As diferenças de PE em comparação com EV são que, em PE, obrigatoriamente $\lambda = \mu$, ou seja, todos os indivíduos da população atual são copiados e modificados pelo operador de mutação e o processo de seleção é fortemente elitista, sobrevivendo para a próxima geração somente os indivíduos de melhor *fitness*. O processo de seleção utilizado em PE, denominado forte elitismo, permite ao

algoritmo convergir mais rápido para um *fitness* ótimo, porém diminui a diversidade dos indivíduos o que pode gerar a convergência para um ótimo local [31].

Nas EEs cada gene no cromossomo é uma dimensão do problema e possui representação de ponto flutuante. Cada novo indivíduo é obtido de cada pai por meio de um operador de mutação Gaussiana com média zero e um desvio padrão pré-selecionado para cada gene. Assim com na PE, a EE seleciona somente os indivíduos de melhor *fitness* para a próxima geração. No primeiro modelo de EE, denominado $(1+1)$-EE, um pai gerava um único novo indivíduo e ambos competiam diretamente pela sobrevivência. Porém, este modelo converge muito lentamente e, na maioria das vezes, para um ótimo local. Outros dois modelos de EE são utilizados atualmente (μ, λ)-EE e $(\mu + \lambda)$-EE, com λ podendo ser maior que μ, ou seja, um pai pode gerar mais um descendente. No primeiro modelo, os μ pais morrem e sobrevivem os λ filhos. No segundo modelo sobrevivem somente os λ melhores indivíduos. Uma outra modificação proposta pela EE é a evolução do *array* de desvio padrão para cada um dos indivíduos [23]. As principais diferenças entre EE e EV são: 1) na EE um pai pode gerar mais de um descendente; 2) EE considera a sobrevivência somente dos melhores indivíduos [31].

Os AGs são os AEs mais conhecidos e utilizados. As principais diferenças entre AGs e os demais AEs apresentados são: 1) a utilização do valor de *fitness* para definir quais dos indivíduos da população irão deixar descendentes; 2) a reprodução com o cruzamento dos pais (recombinação), além de mutação; 3) a utilização da representação binária nos cromossomos, pois esta é considerada uma forma de representação universal; 4) os AGs selecionam para a próxima geração como sobreviventes somente os novos indivíduos gerados. A seleção para reprodução com relação ao *fitness* é feita com base na proporção do *fitness* de cada indivíduo em relação ao *fitness* total (soma do *fitness* da população), assim indivíduos com *fitness* acima da média conseguem pelo menos um descendente para problemas de maximizar, em problemas de minimizar são os indivíduos abaixo da média que geram pelo menos um descendente.

Com base nos principais AEs apresentados nesta Seção, são definidas as principais componentes de um AE. Dependendo de escolhas específicas de tipo de seleção, número de descendentes gerados, operador de reprodução e representação dos indivíduos pode-se classificar um AE como um dos algoritmos apresentados. O Algoritmo 4.1 apresenta os principais passos de um AE genérico.

A seguir são descritos com mais detalhes os principais métodos de seleção de indivíduos para reproduzir ou para sobreviver para a próxima geração. Os principais métodos de seleção utilizados, em ordem de pressão seletiva do mais fraco para o mais forte, são [31]:

1. Seleção uniforme ou neutra: todos os indivíduos tem igual probabilidade de serem selecionados independente do seu valor de *fitness*. Em geral este método é utilizado para selecionar os indivíduos que irão reproduzir;

Algoritmo 4.1 Pseudo-código de um AE típico.

ALGORITMO AE
// inicializa o contador de tempo
(1) $t := 0$;
// inicializa uma população de indivíduos, em geral de forma aleatória
(2) INICIA_POPULACAO($P(t)$);
// avalia o grau de adequação dos indivíduos de P'
(3) AVALIA(P');
// testa o critério de término (por exemplo, um tempo t máximo)
(4) **ENQUANTO** critério não atingido **FACA**
  // seleciona uma sub-população para geração de descendentes
  (5)$P' :=$ SELECIONA_PAIS($P(t)$);
  // reproduz os pais selecionados aplicando operadores de recombinação e/ou mutação
  (6)REPRODUZ(P');
  // avalia o grau de adequação dos indivíduos de P'
  (7)AVALIA(P');
  // seleciona os sobreviventes entre os indivíduos de $P(t)$ e P'
  (8)$P(t+1) :=$ SOBREVIVENTES($P(t)$,P');
  // incrementa o contador de tempo
  (9)$t := t+1$;

2. Proporcional ao *fitness*: para cada indivíduo é assinalada uma probabilidade $\bar{p}_i$ proporcional ao seu *fitness* definida por f_i/f_{sum}, onde f_i é o *fitness* do indivíduo i e f_{sum} é a soma do *fitness* de todos os indivíduos. Os indivíduos são selecionados de acordo com a probabilidade $\bar{p}_i$, que é alterada dinâmicamente. No início do processo evolutivo esse método de seleção é mais elitista e, no final do processo, comporta-se como uma distribuição uniforme, pois os indivíduos tendem a ter valores de *fitness* próximos. Pode ser utilizado para selecionar os indivíduos para reproduzir ou sobreviver;

3. Classificação linear ou torneio de dois: dois indivíduos, selecionados aleatoriamente, competem entre si em um torneio; o vencedor do torneio é o indivíduo de melhor *fitness*. São realizados λ torneios para selecionar λ indivíduos. Pode ser utilizado para selecionar os indivíduos para reproduzir ou sobreviver. Segundo [31] o torneio de dois tem o mesmo comportamento do processo de seleção dos indivíduos por meio de uma classificação linear crescente dos indivíduos (ver a referência [31] para a demonstração matemática);

4. Classificação não linear ou torneio de $k > 2$: similar a seleção por torneio de dois, só que em vez de selecionar aleatoriamente dois indivíduos, seleciona-se $k > 2$ indivíduos que competem entre si; vence o indivíduo de melhor *fitness*. Quanto maior for k, maior será a pressão seletiva, conseqüentemente mais elitista será o processo. Ao contrário do torneio de dois, este método de seleção tem o comportamento de uma classificação não linear (ver a referência [31] para a demonstração matemática);

5. Trucamento: seleciona somente os indivíduos de melhor *fitness*, os demais são descartados. Este método é o de maior pressão seletiva nos indivíduos, em geral é

utilizado para definir os indivíduos que sobreviveram para a próxima geração. Uma combinação bastante comum é utilizar seleção uniforme para selecionar os indivíduos para reprodução e seleção por truncamento para determinar os indivíduos que sobreviverão.

Os métodos de reprodução dividem-se em mutação e recombinação. A mutação é o mecanismo de reprodução no qual um indivíduo reproduz, pela cópia de seu cromossomo gerando um filho, que então sofre uma variação de um ou mais gene(s). A quantidade de variação é controlada pelo número de genes que poderão ser modificados e o quanto serão alterados. No caso de representações em ponto flutuante para ter maior exploração do espaço, muta-se poucos genes com grandes alterações. Para representação binária deve-se aumentar o número de genes que serão alterados. Para efetuar pequenas alterações no indivíduo, muta-se poucos genes com pequenas alterações em cada gene. Em geral, para representações de ponto flutuante, utiliza-se a distribuição Gaussiana com média zero para mutar os indivíduos e desvio padrão igual para todos os genes ou com desvio padrão variando para cada um dos genes como no caso das EEs [31]. Em representações binárias, a mutação consiste em calcular o complemento do valor do gene.

A recombinação é o mecanismo de reprodução onde os novos indivíduos são compostos por genes copiados de partes de dois ou mais pais. Em AEs com cromossomos de comprimento fixo, a recombinação é tradicionalmente chamada de crossover[2], pois seleciona-se aleatoriamente um ou mais pontos, que divide os cromossomos pais em subseqüências que são trocadas entre eles para compor os novos indivíduos [31]. Por exemplo, suponha um cromossomo contendo valores de quatro variáveis inteiras (ou genes) de um problema, isto é, $\boxed{x_1}\boxed{x_2}\boxed{x_3}\boxed{x_4}$, onde um conjunto de valores para as variáveis x_1, x_2, x_3 e x_4 define uma solução para o problema. Considere os seguintes indivíduos (cromossomos): $\boxed{12}\boxed{34}\boxed{08}\boxed{71}$ e $\boxed{05}\boxed{17}\boxed{29}\boxed{66}$.

Aplicando-se o operador de crossover de um ponto a esses indivíduos, com o ponto de crossover entre o segundo e o terceiro genes, obtém-se os seguintes descendentes: $\boxed{12}\boxed{34}\boxed{29}\boxed{66}$ e $\boxed{05}\boxed{17}\boxed{08}\boxed{71}$.

Outros tipos de crossover de pontos são:

1. crossover de dois pontos: são escolhidos dois pontos de corte e troca-se a subseqüência entre eles para compor os novos indivíduos;

2. crossover de multipontos ou uniforme: define-se uma máscara aleatória que determina de qual dos pais será copiado o gene para compor o novo indivíduo.

Em representações de ponto flutuante, podem ser aplicados outros tipos de recombinação. Por exemplo, os genes do novo indivíduo podem ser obtidos da média aritmética dos genes dos pais, ou por outro tipo de distribuição ponderada entre os genes dos pais.

[2]Crossover é um tipo de recombinação de acordo com Jong [31]

5 Estruturas de Dados para AEs Aplicados a PPRs

As estruturas de dados para AEs aplicados a PPRs devem lidar com representações de árvores geradoras de grafos (ver Seção 2) [28, 27], pois as soluções de PPRs, em geral, envolvem árvores geradoras de um grafo que representa o problema.

Raidl e Julstrom [54] definiram um conjunto de critérios para avaliar uma representação de árvores para AEs:

- Espaço: as representações não devem utilizar quantidades excessivas de memória;
- Tempo: a complexidade de tempo para avaliar, recombinar e mutar as representações deve ser pequeno. O processo de avaliação em geral inclui a decodificação da representação, então este também deve ser eficiente;
- Factibilidade: todas as representações, principalmente as obtidas por crossover e mutação devem representar soluções factíveis do problema;
- Cobertura: a representação deve ser capaz de codificar todo o espaço de busca, ou seja, capaz de representar todas as árvores possíveis;
- Tendência: todas as soluções devem ser igualmente prováveis de serem representadas. Uma representação que possui uma tendência maior em codificar certos tipos de solução só é favorável se essas soluções estão próximas da solução ótima;
- Localidade: pequenas alterações na representação devem produzir pequenas alterações na árvore representada;
- Hereditariedade: as operações de recombinação devem preservar a maioria das arestas dos pais nos filhos. Considere E_{T1} e E_{T2} os conjuntos de arestas dos pais e $E_{T'1}$ e $E_{T'2}$ os conjuntos de arestas dos filhos, o ideal é que $E_{T1} \cup E_{T2}$ seja igual $E_{T'1} \cup E_{T'2}$, isto é, que todas as arestas presentes nos pais estejam nos filhos;
- Restrições: devem permitir a inclusão de restrições dos problemas com facilidade, sem a necessidade de grandes modificações;
- Hibridismo: os operadores da representação devem permitir a inserção de heurísticas do problema;
- Grafos esparsos: a representação deve codificar soluções em grafos não completos (com menos de $n(n-1)/2$ arestas) preservando as características anteriores.

Uma representação com as características mencionadas deveria ser eficiente em AEs aplicados a PPRs. A última propriedade em especial não tem sido avaliada na literatura,

visto que as representações existentes foram desenvolvidas para lidar somente com grafos completos.

Além das propriedades anteriores, recentemente Rothlauf [55] analisa três propriedades e a influência delas no funcionamento dos AEs para PPRs:

- Localidade: espera-se que pequenas alterações na codificação produzam pequenas alterações na árvore correspondente, definida como alta localidade. Uma representação com baixa localidade, ou seja, onde pequenas alterações na codificação produzem grandes alterações na árvore correspondente, faz com que os AEs utilizando-a assemelhem-se a processos de busca aleatória;

- Escalabilidade: a representações que tornam genes mais significativos que outros para a solução do problema diminuem a velocidade de convergência dos AEs. Assim, é desejável que as representações não possuam escalabilidade dos genes;

- Redundância: mais de uma representação codifica a mesma árvore. Se a redundância for sinônima (codificações similares representam árvores iguais) não altera a complexidade do problema, caso contrário a complexidade do problema é alterada, podendo diminuir ou aumentar.

5.1 Vetor de Características

A representação Vetor de Características (CV, do inglês, *Characteristic Vector*) é um *array* binário que indica se uma possível aresta é utilizada ou não em um grafo [60, 15]. Para um grafo com n nós, existem $m = n(n-1)/2$ possíveis arestas, assim o *array* da representação CV possui tamanho m. Todos as possíveis arestas são numeradas de $1 \ldots m$ e cada aresta é associada a uma posição do *array*. Um exemplo da CV da árvore com cinco nós da Figura 1 é apresentado na Tabela 1. A presença da aresta i é indicada pelo valor 1, na posição i do *array* e a ausência pelo valor 0.

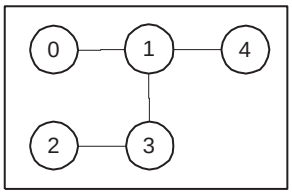


Figura 1: Exemplo de uma árvore com 5 nós.

1	0	0	0	0	1	1	1	0	0
0-1	0-2	0-3	0-4	1-2	1-3	1-4	2-3	2-4	3-4

Tabela 1: Representação por *array* de características para a árvore da Figura 1.

CV possui localidade alta, pois a alteração de uma posição do *array* consiste na inclusão ou remoção de uma aresta. A localidade de CV é independente da estrutura da árvore representada [57]. Se a CV representa uma árvore a inclusão de uma aresta forma um ciclo sendo necessário um mecanismo de correção, que remove uma aresta do ciclo formado.

Uma vez que CV utiliza a representação binária os operadores de crossover e mutação padrões podem ser aplicados. Porém, são necessários mecanismos de correção, pois, o principal problema de CV é a representação de soluções infactíveis, ou seja, que possuem ciclos ou são grafos desconexos [57]. A complexidade de espaço necessária para CV é $O(m)$, que para grafos completos é $O(n^2)$. Para decodificar a árvore representada é necessário percorrer todo o *array* de CV de comprimento m, então a complexidade de tempo é $O(m)$ e para grafos completos novamente é $O(n^2)$ [53].

5.2 Número de Prüfer

A representação por Número de Prüfer tem como base a propriedade da correspondência única entre uma *string* de comprimento $n-2$ e uma única árvore geradora [50]. Dada uma árvore T com n nós, o número de Prüfer (P) é um número de $n-2$ dígitos, onde esses dígitos são números entre 1 e n e são definidos pelo Algoritmo 5.1 de codificação [69, 1].

Algoritmo 5.1 - Codificação por número de Prüfer [69, 1].

(1) $P \leftarrow \emptyset$
(2) $\overline{P} \leftarrow \emptyset$
(3) Enquanto (houver mais de dois nós a serem considerados)
  (4) $i \leftarrow$ nó folha de menor rótulo; (ver Seção 2)
  (5) $j \leftarrow$ nó predecessor de i;
  (6) $P \leftarrow$ concatena (P, j);
  (7) $\overline{P} \leftarrow$ concatena $(\overline{P}, i)$, se $\notin P$;
  (8) Remova o nó i e a aresta (i, j) de T.

Considere a Figura 2, a conversão, passo a passo, para o número de Prüfer é realizada da seguinte maneira: inicialmente 2 é o nó folha de menor rótulo. O nó adjacente a 2 é adicionado a P ($P = 3$). O nó 2 é então removido junto com a aresta $(2, 3)$. Na próxima iteração, 4 é selecionado como o nó folha de menor rótulo, 3 é colocado em P ($P = 33$). O nó 4 é removido de T. A seguir, 3 é o nó folha de menor rótulo, como 1 é o nó adjacente a 3, este é adicionado a P ($P = 331$). O nó 3 é removido de T. No próximo passo, 5 é o nó folha de menor rótulo e 1 é nó adjacente a ele, portanto, $P = 3311$ (ver Tabela 2). Neste estágio, o grafo possui apenas dois nós e o algoritmo termina.

3	3	1	1

Tabela 2: Representação por número de Prüfer para a árvore da Figura 2 [69, 1].

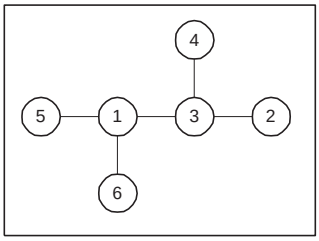


Figura 2: Grafo com 6 nós [69, 1].

O procedimento para a decodificação por número de Prüfer (ver Algoritmo 5.2) considera P como sendo o número de Prüfer e $\overline{P}$ o conjunto de todos nós não pertencentes a P. Esses nós são denominados elegíveis para posteriores considerações. Enquanto houver pelo menos um dígito em P, o nó de menor rótulo em $\overline{P}$ é atribuído a i, j recebe o dígito mais à esquerda de P, (i, j) é adicionado em T, i é removido de $\overline{P}$ e j de P. Caso não exista mais nenhum j em P, j é colocado em $\overline{P}$. Se não restar mais nenhum dígito em P, há dois nós remanescentes em $\overline{P}$ e a aresta correspondente a esses nós é adicionada à árvore. A complexidade de tempo do algoritmo de decodificação é $O(nlogn)$ [26].

Algoritmo 5.2 - Decodificação por número de Prüfer [26, 69, 1].

(1) Enquanto(restar pelo menos um dígito em P)
  (2) $i \leftarrow$ nó elegível com o menor rótulo.
  (3) $j \leftarrow$ dígito mais a esquerda de P.
  (4) Adicione a aresta (i, j) em T.
  (5) Remova i de $\overline{P}$ e j de P.
  (6) Se ($j \notin P$)
    (7) Coloque j em $\overline{P}$.
* Seja r e s os dois nós restantes: *
(8) Adicione a aresta (r, s) na árvore.

É fácil ver que uma das principais desvantagens do número de Prüfer é sua localidade relativamente baixa, uma vez que a mudança de somente um dígito de um número de Prüfer pode mudar drasticamente a árvore resultante [26]. Uma outra desvantagem do número de Prüfer ocorre para grafos esparsos. Neste caso, a geração de árvores válidas utilizando número de Prüfer torna-se improvável. Além disso, é importante salientar que há diversos PPRs que não correspondem a grafos completos.

Com a representação número de Prüfer, pode-se utilizar operadores de crossover padrão sem gerar soluções infactíveis para problemas de grafos completos. Para problemas como o de transporte, são necessárias algumas modificações no processo de decodificação [63].

5.3 Tendência de Ligação e Nó

A representação Tendência de Ligação e Nó (LNB, do inglês *Link and Node Biased*) foi desenvolvida por Palmer [44]. Esta representação permite que o processo de busca

do algoritmo genético dê preferência para nós que são mais importantes para solução do problema. Podem ser utilizadas duas formas de tendência para a representação: tendência de nó e tendência de ligação e nó [45].

Em uma NB (do inglês, *Node Biased*), uma árvore de quatro nós possui quatro tendências $[b_1 b_2 b_3 b_4]$, armazenadas em um *array* com quatro posições. As tendências são multiplicadas pelo fator P_2, que representa o quanto a tendência deve ser considerada, e pelo custo máximo das arestas C_{max}. Os valores do *array* de tendência são, então, somados a matriz de custos do grafo. Uma nova matriz de custo do grafo é obtida por: $C'_{i,j} = C_{i,j} + P_2(b_i + b_j)C_{max}$, que é a matriz tendenciosa de nós. Para decodificar a árvore representada pelo *array* de tendências aplica-se o algoritmo de Prim (ver Seção2) na matriz de custo tendenciosa. NB tem a desvantagem de não ser capaz de representar todas as árvore geradoras de um grafo [45, 44].

Para solucionar esse problema, foi proposta uma representação que considera tanto a tendência de nó como a de ligação. A representação possui um *array* de tendências para os n nós e para as m arestas do grafo. A fórmula de atualização da matriz de custo possui dois fatores de multiplicação P_1 que multiplica as tendências de ligação e P_2, que multiplica as tendências de nó, e a matriz de custos tendenciosa é obtida por, $C'_{i,j} = C_{i,j} + P_1 b_{i,j} C_{max} + P_2(b_i + b_j)C_{max}$, onde $b_{i,j}$ é a tendência associada a aresta (i, j); b_i, b_j é a tendência associada aos nós i e j, respectivamente. Com a alteração dos valores de P_1 e P_2 é possível dar maior prioridade as tendências de ligação ou as tendências de nó. Novamente, o algoritmo de Prim é aplicado sobre a nova matriz para obter a árvore geradora correspondente [45, 44].

Como a LNB utiliza o algoritmo de Prim para encontrar a árvore geradora a partir da matriz de custos modificada, a complexidade de tempo do algoritmo de decodificação é $O(m + nlogn)$, onde n é o número de nós e m o número de arestas. Substituindo o valor de $m = n(n-1)/2$, para um grafo completo, obtém-se que a complexidade de tempo é $O(n^2)$ [44]. O espaço utilizado para armazenar a árvore é de n posições para as tendências de nó e de $m = n(n-1)/2$, no caso de um grafo completo, posições para as tendências de ligação, então a complexidade de espaço total é $O(n^2)$. Para grafos não completos a complexidade de espaço é $O(m + n)$.

Gaube e Rothlauf [24] mostraram que a LNB dependendo do valor dos fatores P_1 e P_2, possui uma tendência em gerar alguns tipos de estruturas de árvores a partir de codificações geradas aleatoriamente.

5.4 Predecessores ou Codificação Determinante

A representação por Predecessores foi inicialmente proposta em [2] e recebeu o nome de codificação determinante pois tem como base os teoremas de Even [22] sobre como determinar o número de árvores geradoras de um grafo orientado com uma determinada

raiz a partir de sua matriz de adjacências. Na representação por predecessores uma árvore é codificada por armazenar para cada nó (a partir da raiz r), o seu predecessor, considerando a árvore orientada no sentido dos caminhos da raiz para as folhas [52]. Se $Pred[i] = j$, então j é o primeiro nó no caminho de i até r em T. Por convenção, assuma $Pred[r] = r$. Como exemplo, na Tabela 3 é mostrado um *array* representando a árvore da Figura 3.

Uma árvore enraizada é representada por uma única seqüência de n dígitos, onde os dígitos são números entre 1 e n. Além disso, qualquer árvore pode ser representada por n desses dígitos. Portanto, esta é uma representação que cobre todo o espaço de soluções [40, 52, 45, 2].

Uma matriz que mapeia os pares de nós em arestas, pode ser computada no início do AE e necessita de espaço $O(n^2)$. O processo de codificação/decodificação, a partir da lista de arestas pertencentes à árvore, pode ser realizado em tempo $O(n)$ [45, 2].

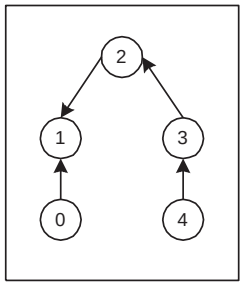


Figura 3: Grafo orientado com 5 nós.

1	1	1	2	3

Tabela 3: Representação por Predecessores para a árvore da Figura 3.

Um dos grandes problemas da representação por predecessores é a grande quantidade de soluções infactíveis que podem ser obtidas pela representação, ou contendo ciclos ou representando árvores desconexas. Os primeiros AEs que utilizaram a representação aplicaram mecanismos de correção após a aplicação do operadores de crossover e mutação padrões [45, 2]. Os AEs, mais recentes utilizam operadores de inicialização da população, recombinação e mutação específicos para evitar a geração de representações que não codificam árvores [40, 52].

Em [52] é proposto um algoritmo de inicialização que mantém dois conjuntos de nós: um para os nós na árvore e outro para os demais nós ainda não conectados a árvore. Aleatoriamente, seleciona nós de ambos os conjuntos um nó da árvore para conectar o novo nó aleatório escolhido do segundo conjunto, sempre que um nó é conectado a árvore ele é removido do conjunto de nós ainda não conectados a árvore. Antes de conectar o nós à árvore verifica se a aresta é válida. O processo se repete até que todos os nós

estejam na árvore. A operação de recombinação é uma operação de recombinação de arestas, inserindo, primeiramente, as arestas em comum em ambos os pais e, em seguida para os nós sem predecessor, ou seja, desconectados, escolhe aleatoriamente um de seus predecessores nos indivíduos pais, até todos os nós terem seus predecessores definidos. A operação de mutação consiste em remover uma aresta, obtendo duas componentes, e inserir uma nova aresta válida ligando as duas componentes.

Em [40] é utilizado um algoritmo de inicialização baseado no algoritmo de Prim, este algoritmo tem complexidade de tempo de $O(nlogn)$, onde n é o número de nós do grafo. Na operação de recombinação também é utilizado um algoritmo com base no algoritmo de Prim, porém o conjunto de arestas possui somente as arestas pertencentes aos indivíduos pais. Similar ao algoritmo de mutação proposto em [52], o algoritmo de mutação utilizado nesta abordagem também remove uma aresta e, em seguida, re-conecta as componentes pela aresta de menor custo que incide em um dos nós da aresta removida.

5.5 Precedentes Diretos

Em [11, 10] foi proposta uma representação que considera o espaço de soluções um conjunto de nós parcialmente ordenado (A, $\leq$), onde:

1. $a \leq a$;
2. $a \leq b$ e $b \leq a$ implica $a = b$;
3. $a \leq b$ e $b \leq c$ implica $a \leq c$;

para todo $a, b, c \in A$.

Um elemento a é denominado precedente direto de b em A se e somente se:

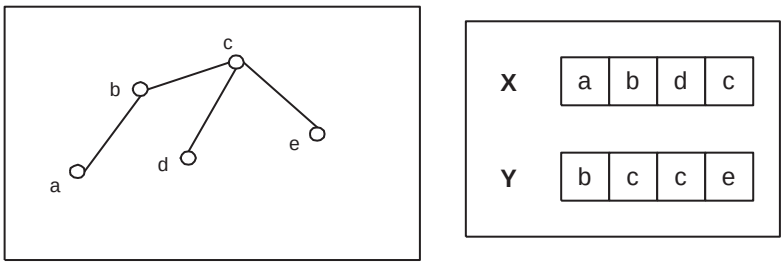
i. $a \neq b$;

ii. $a \leq b$;

iii. $\nexists\ c \in A$ tal que $a \leq c$ e $c \leq b$.

Esta relação entre a e b é denotada por $b \hookleftarrow a$. Similarmente, b é dito sucessor direto de a se e somente se a é um de seus precedentes direto.

Em uma árvore, os nós são parcialmente ordenados e com exceção da raiz todos têm um único precedente direto. Assim, com base nestes conceitos podemos definir a representação de uma árvore em dois *array*es (X, Y) de mesmo tamanho, onde em X serão armazenados todos os elementos, exceto a raiz e em Y, os precedentes diretos dos elementos em X (ver Figuras 4(a) e 4(b)). Desta maneira, $X \hookleftarrow Y$ representa uma árvore geradora. A operação de codificação pode ser realizada com o auxílio de uma busca em largura (método de busca e percurso em grafos [28, 3]). Para aplicar alterações na representação por precedência



(a) Exemplo de uma árvore com 5 nós. (b) Precedentes diretos.

Figura 4: Exemplo da representação por precedentes diretos.

é necessário verificar se esta é consistente, ou seja, se as regras (i-ii-iii) são obedecidas. Caso contrário, as alterações são ditas inconsistentes.

A operação de recombinação é efetuada por meio do intercâmbio de caminhos entre duas árvores geradoras. As principais etapas do processo para gerar as soluções derivadas de duas outras são mostradas no Algoritmo 5.3 com T_1 e T_2 as árvores pais na representação precedentes diretos.

Algoritmo 5.3 - Algoritmo de recombinação por intercâmbio de caminhos [11, 10].

(1) Selecione aleatoriamente dois nós a e b;
(2) Encontre os caminhos C_1 e C_2 entre a e b: C_1 em T_1 e C_2 em T_2;
(3) Se for possível, submeta C_2 a T_1 e C_1 a T_2. (ver Algoritmo 5.4)
(4) Caso contrário, volte ao passo 1.

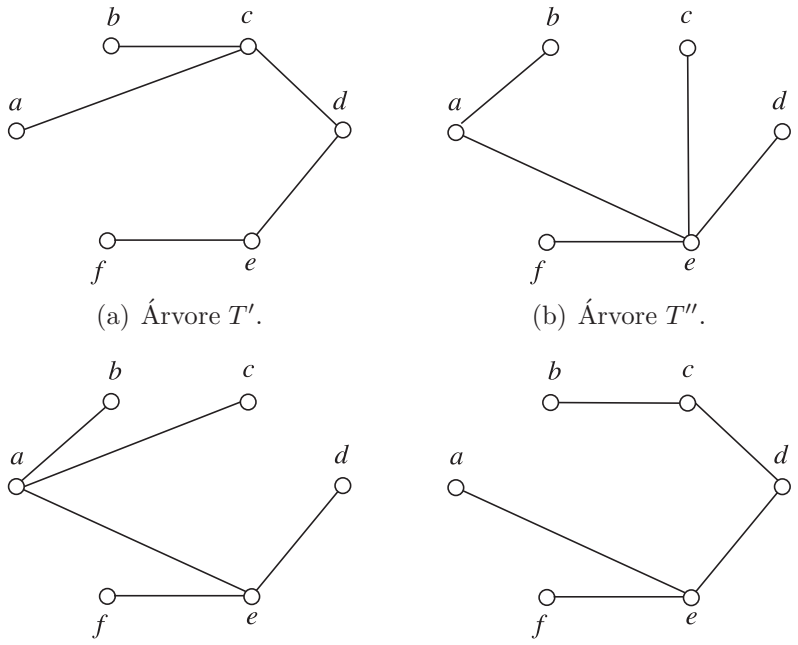
Para verificar se o caminho pode ser submetido a árvore, considere a o nó de menor rótulo do caminho C. Denote por $F(x)$ os sucessores diretos do nó x em C. Considere um conjunto L que conterá os elementos da árvore, inicialmente $L = F(a)$. Com essas definições o processo de submeter um caminho a árvore pode ser realizado pelo Algoritmo 5.4.

Algoritmo 5.4 - Algoritmo de submissão do caminho à árvore [11, 10].

(1) Altere T, modificando toda relação de precedência $x \hookleftarrow y$ de T para $x \hookleftarrow z$ de C, se e somente se, $x \in L$ e $x \hookleftarrow z$ é consistente em T.
(2) Faça $L = \cup F(x \in L)$.
(3) Se $L \neq \emptyset$, volte ao passo (1).

Considere como exemplo da operação de recombinação as árvore T' e T'' das Figuras 5(a) e 5(b), sua representação por precedentes diretos é: $T' = \{c \hookleftarrow a, b \hookleftarrow c, d \hookleftarrow c, e \hookleftarrow d, f \hookleftarrow e\}$ e $T'' = \{b \hookleftarrow a, e \hookleftarrow a, c \hookleftarrow e, d \hookleftarrow e, f \hookleftarrow e\}$. Os caminhos selecionados são $C' = \{b \hookleftarrow c, d \hookleftarrow c\}$ em T' e $C'' = \{b \hookleftarrow a, e \hookleftarrow a, d \hookleftarrow e\}$ em T''.

Submeta C' a T'', c é o nó de menor rótulo de C', e os sucessores de c, $F(c) = \{b, d\}$, ambos os precedentes são mudanças válidas em T'' aplicando as mudanças nos precedentes obtém-se como resultado a árvore $\{b \hookleftarrow c, e \hookleftarrow a, c \hookleftarrow e, d \hookleftarrow c, f \hookleftarrow e\}$. Similarmente, submetendo C'' para T', o elemento a é o nó de menor rótulo do caminho e os sucessores de a, $F(a) = \{b, e\}$. Como ambas as relações de precedência são mudanças válidas em



(a) Árvore T'.

(b) Árvore T''.

(c) Filho 1.

(d) Filho 2.

Figura 5: Exemplo do recombinação.

T' as atualizações resultam em $\{c \hookleftarrow a, b \hookleftarrow a, d \hookleftarrow c, e \hookleftarrow a, f \hookleftarrow e\}$ (ver Figura 5(c)). Somente o elemento e tem um sucessor em C'': $F(e) = d$, que é agora consistente, resultando na árvore $\{c \hookleftarrow a, b \hookleftarrow a, d \hookleftarrow e, e \hookleftarrow a, f \hookleftarrow e\}$ (ver Figura 5(d)).

A operação de recombinação proposta apresenta as seguintes vantagens [11, 10]:

- baixo custo computacional: alterar a precedência de um elemento envolve exclusivamente uma operação de substituição no *array* de precedentes diretos;
- convergência rápida: troca de caminhos durante a recombinação propaga grandes blocos de informação, reduzindo o espaço de busca significativamente;
- viabilidade: uma simples verificação de consistência durante o intercâmbio de caminhos garante a factibilidade da solução.

5.6 Conjunto de Arestas

A representação Conjunto de Arestas proposta em [51] consiste em representar uma árvore diretamente por meio de seus conjuntos de arestas. A Figura 6 apresenta o exemplo de uma árvore geradora com 12 nós e sua representação por conjunto de arestas.

Esta representação pode ser implementada de duas formas: por meio de um *array* ou de uma tabela *hash* onde as entradas são os pares de nós que representam a aresta. A implementação por tabela *hash* permite que as operações de inserção, remoção e busca das arestas de um indivíduo ocorram em tempo constante, portanto, mais eficiente em

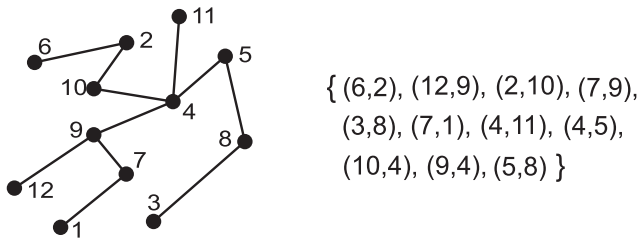


Figura 6: Exemplo de uma árvore com 12 nós e sua representação em conjunto de arestas.

tempo de computação. Ambas as formas de implementação possuem complexidade de espaço $O(n)$, onde n é o número de nós do grafo [54].

Para garantir a obtenção de somente soluções factíveis na geração aleatória de soluções e na aplicação das operações de recombinação e mutação, foram propostos algoritmos específicos para cada uma das operações. Para geração de soluções são propostos três algoritmos: o primeiro e o segundo são modificações nos algoritmos de árvore geradora mínima de Prim (ver Algoritmo 5.5) e Kruskal (ver Algoritmo 5.6), respectivamente, e o terceiro é o algoritmo de geração por caminhos aleatórios (ver Algoritmo 5.7) [54]. Considere para todos os algoritmos E o conjunto de arestas, V o conjunto de nós pertencentes ao grafo dado como entrada e E_T o conjunto de arestas que vão compor a árvore geradora da inicialização.

O algoritmo de Prim modificado para geração de soluções na representação conjunto de arestas altera o processo de escolha da nova aresta a ser inserida na árvore escolhendo aleatoriamente uma das arestas possíveis em vez de escolher a aresta de menor peso como no algoritmo de Prim para árvore geradora mínima. Inicialmente escolhe um nó aleatório, adiciona ao conjunto de nós conexos C e constrói o conjunto A de todas as arestas válidas, ou seja, que geram uma árvore parcial conexa. Escolhe aleatoriamente uma aresta pertencente a A, verifica se esta aresta não forma ciclo e adiciona a aresta na árvore, o novo nó é incluído em C e as novas arestas são incluídas em A, repete-se esse processo até que todos os nós tenham sido incluídos, (ver o Algoritmo 5.5). A complexidade de tempo do algoritmo é $O(m)$ em geral. Para grafos completos, a complexidade de tempo e espaço é $O(n)$ pois é necessário manter somente o controle de nós já inseridos na árvore [54].

O processo de geração das soluções utilizando o algoritmo de Prim modificado é simples e eficiente, permitindo também a inserção de qualquer tipo de restrição do problema facilmente. O problema desse algoritmo é a tendência maior para alguns tipos de estrutura, as arestas ligadas ao nó inicial tem uma probabilidade maior de serem escolhidas, assim o algoritmo tem um tendência maior em gerar árvores estrela [54].

O algoritmo de Kruskal modificado (ver Algoritmo 5.6) para geração de soluções na representação conjunto de arestas em vez de selecionar uma aresta para inclusão na árvore pelo seu peso, seleciona uma aresta aleatoriamente e, então, verifica se esta forma um ciclo; caso contrário, a aresta é incluída na árvore. O processo repete-se até que $n - 1$ arestas

Algoritmo 5.5 - Geração das Soluções por Conjunto de Arestas com Algoritmo de Prim Modificado [54].

(1) $E_T \leftarrow O$;
(2) Escolhe um nó aleatório inicial $s \in V$;
(3) $C \leftarrow \{s\}$; /* conjunto de nós conexos */
(4) $A \leftarrow \{e \in E \mid e = (s, v), v \in V\}$; /* conjunto de arestas que podem ser escolhidas */
(5) Enquanto $C <> V$
    (6) Escolha uma aresta $(u, v) \in A, u \in C$ aleatoriamente;
    (7) $A \leftarrow A - \{(u, v)\}$;
    (8) Se $(v \notin C)$
        (9) /* Conecta v a árvore parcial */
        (10) $E_T \leftarrow E_T \cup \{(u, v)\}$;
        (11) $C \leftarrow C \cup \{v\}$;
        (12) $A \leftarrow A \cup \{e \in E \mid e = (v, w), \wedge w \notin C\}$
(13) Retorne E_T

tenham sido adicionadas a árvore. A complexidade de tempo para verificar se uma aresta forma ou não ciclo é $O(m)$. O passo do algoritmo proposto que possui maior complexidade é o que verifica se uma aresta forma ou não ciclo, então, a complexidade de tempo do algoritmo de Kruskal modificado é $O(m)$ [51, 54].

Algoritmo 5.6 - Geração das Soluções por Conjunto de Arestas com Algoritmo de Kruskal Modificado [51, 54].

(1) $E_T \leftarrow O$;
(2) $A \leftarrow E$;
(3) Enquanto $|E_T| < |V| - 1$
    (4) Escolha aleatoriamente um aresta $(u, v) \in A$
    (5) $A \leftarrow A - \{(u, v)\}$;
    (6) Se u e v não são conectados em E_T
        (7) $E_T \leftarrow E_T \cup \{(u, v)\}$;
(8) Retorne E_T;

Assim como o algoritmo de Prim modificado, o algoritmo de Kruskal modificado apresenta uma tendência em formar certos tipos de estrutura, porém menos tendencioso que o algoritmo de Prim. A inclusão de restrições do problema também pode ser feita com o algoritmo de Kruskal, no entanto, o algoritmo de Prim fornece mais flexibilidade para esse fim [54].

No algoritmo de geração das soluções em conjunto de arestas por caminhos aleatórios, utiliza-se o processo definido por Broder [7] de caminhos aleatórios em G, o grafo do problema. Uma partícula começa de um nó arbitrário em G. A cada passo, a partícula move-se aleatoriamente escolhendo uma aresta adjacente de um de seus vizinhos. Quando a partícula visita um nó não incluído na árvore, adiciona a aresta do caminho na árvore, até que todos os nós tenham sido visitados, completando a árvore geradora [54]. O Algoritmo 5.7 apresenta o pseudo-código deste processo.

Segundo Broder [7], este algoritmo retorna árvores geradas com probabilidades uniformes. O pior caso de complexidade de tempo deste algoritmo não possui um limite superior. Experimentos mostram que, para a maioria dos grafos, o algoritmo proposto

Algoritmo 5.7 - Geração das Soluções por Conjunto de Arestas com Algoritmo de Caminhos Aleatórios [54].

(1) $E_T \leftarrow O$;
(2) $v_0 \leftarrow$ um nó aleatório de G;
(3) Marque v_0 como visitado;
(4) Enquanto $|E_T| < n - 1$
    (5) $v_1 \leftarrow$ um vizinho aleatório de v_0;
    (6) Se v_1 não visitado
        (7) $E_T \leftarrow E_T \cup \{(v_0, v_1)\}$;
        (8) Marque v_1 como visitado;
    (9) $v_0 \leftarrow v_1$;
(10) Retorne E_T

possui tempo de execução $nlogn$ e para alguns casos especiais é n^3 [54]. Com relação a verificação de restrições do problema este método permite o mesmo processo que o algoritmo de Prim.

O operador de recombinação tem como objetivo garantir uma boa hereditariedade, ou seja, deve construir descendentes que, juntos, possuam a maioria ou todas as arestas dos pais. Uma forma de obter o resultado esperado, é aplicar um dos algoritmos propostos nesta Seção para geração de soluções no grafo $G' = (V, E_{T1} \cup E_{T2})$, onde E_{T1} e E_{T2} são as arestas dos pais. Porém quando o problema possui restrições, nem sempre é possível obter soluções factíveis utilizando somente as arestas dos pais, assim é necessário um outro algoritmo de recombinação [54].

O algoritmo proposto inclui as arestas presentes em ambos os pais (E_{T1} e E_{T2}) na nova árvore (E_T), sempre obedecendo a restrição do problema. Em seguida, insere as arestas do conjunto F, pertencentes à disjunção das arestas de seus pais. Se após a inclusão desses conjuntos, uma árvore geradora válida não foi formada, então, determina cada componente desconexa U_k e seleciona arestas aleatoriamente para conectar cada componente à árvore que está sendo formada. O Algoritmo 5.8 descreve os passos da aplicação do operador de recombinação. Para ilustrar o funcionamento do operador, considere a Figura 7.

O operador de mutação provê alta localidade, isto é, uma pequena alteração no cromossomo provoca uma pequena mudança na árvore. O operador de mutação consiste em escolher aleatoriamente uma aresta a ser inserida e remover uma aresta pertencente ao ciclo formado pela inclusão da nova aresta. Inicialmente, uma aresta (i, j) é escolhida para inserção. O próximo passo é determinar o caminho (L) conectando os nós i a j. Em seguida, uma aresta (a, b) é escolhida para ser removida da árvore. Por último, (i, j) é inserido e (a, b) é removido da solução. Os passos do processo de mutação são descritos pelo Algoritmo 5.9. Um exemplo de funcionamento deste operador é ilustrado na Figura 8.

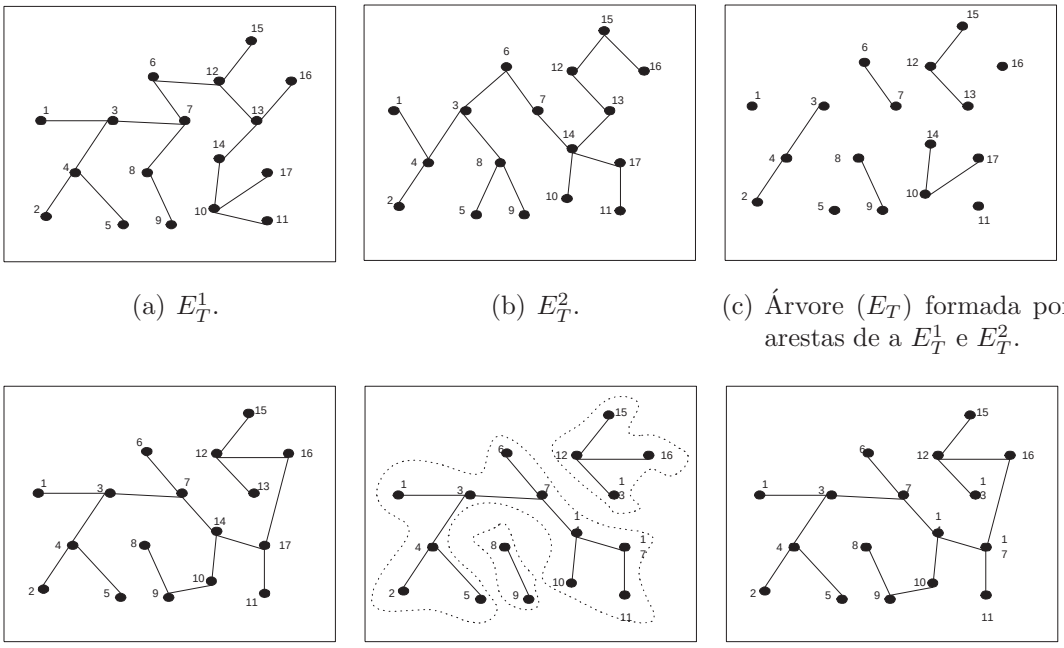
Os operadores de mutação e recombinação podem ser implementados em tempo $O(n)$ [51]. Buscando melhorar o desempenho do algoritmo, foi proposta em [51] a utilização de heurísticas sobre a inicialização e os operadores de mutação e recombinação. Rothlauf et al. [58, 66] mostraram que a representação de conjunto de arestas com inicialização e ope-

Algoritmo 5.8 - *Crossover* por Conjunto de Arestas [51].

(1) $E_T \leftarrow E_{T1} \cap E_{T2}$
(2) $F \leftarrow E_{T1} \cup E_{T2} \backslash E_T$;
(3) Para (todas arestas $(i,j) \in F$ em ordem aleatória)
    (4) Se $(i,j) \notin E_T$)
        (5) $E_T \leftarrow E_T \cup \{(i,j)\}$;
        (6) Se ($|E_T| = |V| - 1$)
            (7) Retorne E_T;
/* determine todas componentes desconexas: U_k */
(8) $U \leftarrow \{U_k\}$, $\forall i,j \in V, i \neq j$:
(9) $i \in U_k \wedge$ conectado$(i,j,E_T) \longrightarrow j \in U_k$,
(10) $i \in U_k \wedge$ **not** (conectado $(i,j,E_T)) \longrightarrow j \notin U_k$,
(11) $\bigcup_k U_k = V$;
/* conecte as componentes aleatoriamente */
(12) Para (todo $U_k \in U \backslash U_1$ em ordem aleatória)
    (13) Escolha $i \in U_1|$ aleatoriamente;
    (14) Escolha $j \in U_k|$ aleatoriamente;
    (15) $E_T \leftarrow E_T \cup (i,j)$;
    (16) $U_1 \leftarrow U_1 \cup U_k$;
Retorne E_T;

Algoritmo 5.9 - Mutação por Conjunto de Arestas [51, 54].

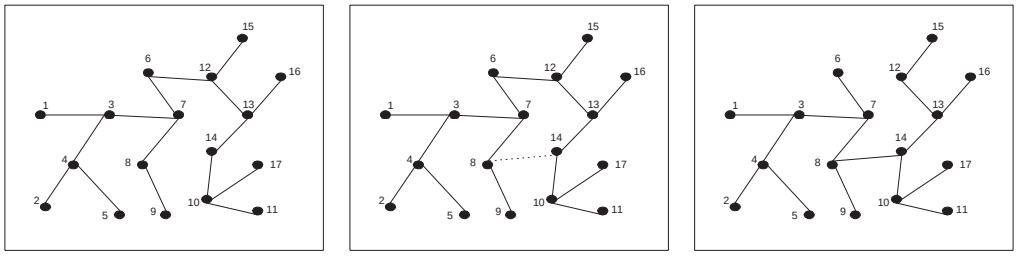
(1) /* Escolha da aresta *(i, j)* para inserção: */
(2) Escolha $i \in V$ aleatoriamente;
(3) Escolha $j \in V \backslash \{i\}|$ grau$(j) < d$ e $(i,j) \notin E_T$ aleatoriamente;
/* Escolha da aresta (a,b) para ser apagada: */
(4) $L \leftarrow \{(k,l) \in E_T | (k,l)$ compõe o caminho de i a $j\}$;
(5) $(a,b) \leftarrow (a,b) \in L | a = i \vee b = i$;
(6) $E_T \leftarrow E_T \cup \{(i,j)\} \backslash \{(a,b)\}$;
(7) Retorne E_T;



(a) E_T^1. (b) E_T^2. (c) Árvore (E_T) formada por arestas de a E_T^1 e E_T^2.

(d) Inclusão de arestas em E_T pertencentes um dos pais. (e) Determinação das componentes. (f) Árvore resultante.

Figura 7: Ilustração do funcionamento do operador de recombinação por Conjunto de Arestas.



(a) E_T. (b) Uma aresta é escolhida para ser inserida em E_T. (c) Árvore resultante.

Figura 8: Ilustração do operador de mutação por Conjunto de Arestas.

radores de recombinação e mutação utilizando heurísticas é tendenciosa. A representação possui maior probabilidade de codificar árvores próximas a árvore geradora mínima e deve ser utilizada somente quando a solução do problema for parecida com a árvore geradora mínima do grafo.

5.7 *Blob Code*

Blob code utiliza uma *string* de Prüfer, como a representação por número de Prüfer, para representar uma árvore geradora de um grafo. Porém, o algoritmo de decodificação da representação *Blob code* aumenta a localidade e a hereditariedade da *string* de Prüfer em relação ao processo de decodificação da representação por número de Prüfer [34, 33, 32]. Essa melhoria é observada em AEs com operadores de crossover e de mutação padrões.

O algoritmo de decodificação de *strings* de Prüfer em árvores geradoras únicas, chamado de *Blob code*, foi proposto em [48] juntamente com outras formas de decodificação diferentes do mapeamento proposto por Prüfer [50]. Considere os nós do grafo numerados de $0-(n-1)$. No algoritmo de decodificação o *blob* é o conjunto de nós do grafo exceto o nó zero. Cada iteração do algoritmo remove um nó do *blob* e armazena um arco (aresta com orientação) na árvore geradora. Ignorando a orientação das arestas obtém-se o árvore geradora não direcionada da *string* de Prüfer [34, 32, 48].

No algoritmo de decodificação são utilizadas duas funções **sucessor**(v) e **path**(v). A primeira retorna w se $(v \rightarrow w)$ está entre as arestas presentes atualmente na árvore geradora. A segunda retorna $TRUE$ se o caminho orientado de v até o nó zero tem intersecção com o *blob*; e caso contrário retorna $FALSE$. O Algoritmo 5.10 apresenta o algoritmo de decodificação *Blob code*, onde $a_1a_2\ldots a_{n-2}$ é a *string* de Prüfer e T é a árvore geradora orientada [34]. Um arco é representado no algoritmo por $(i \rightarrow j)$.

Algoritmo 5.10 Algoritmo de Decodificação *Blob Code*.

(1) blob $\leftarrow \{1, 2, \ldots, n-1\}$;
(2) $T \leftarrow \{(blob \rightarrow 0)\}$;
(3) Para i de 1 até $n-2$
    (4) blob $\leftarrow$ *blob* $-\{i\}$;
    (5) Se **path**(a_i)
        (6) $T \leftarrow T \cup \{(i \rightarrow a_i)\}$;
    (7) Senão
        (8) $T \longleftarrow T \cup \{(i \rightarrow$ **sucessor**(blob))$\}$;
        (9) $T \leftarrow T - \{$(blob$\rightarrow$**sucessor**(blob))$\}$;
        (10) $T \leftarrow T \cup \{$(blob $\rightarrow a_i)\}$;
(11) blob $\leftarrow (n-1)$ em todas as arestas;
Retorne T;

A complexidade de tempo do algoritmo *Blob code* é, no pior caso, $O(n^2)$. Porém, segundo Julstrom [34], na média este algoritmo parece ser muito mais rápido. Como a *string* de Prüfer representa uma única árvore utilizando *Blob code*, os operadores de crossover e mutação padrões podem ser aplicados sem gerar soluções infactíveis. Julstrom [34] recomenda a utilização do operador de crossover de dois pontos e o operador de mutação padrão, no qual com uma probabilidade de mutação pequena, cada símbolo é substituído por um valor aleatório no domínio dos nós. Ambos os operadores tem complexidade de tempo $O(n)$.

5.8 Chaves Aleatórias para Redes

A representação Chaves Aleatórias para Rede (NetKeys, do inglês, *Network Random Keys*) utiliza NetKeys como uma alternativa aos problemas de CV (ver Seção 5.1) [57, 59].

NetKeys permite que as arestas que formam a árvore sejam priorizadas, as variáveis são números reais entre 0 e 1 e podem ser interpretadas como a importância da aresta, assim é possível distinguir quais arestas são mais ou menos importantes. Como NetKeys

armazenam a importância das arestas, não ocorrem problemas de redundância, sub- ou sobre- representação de soluções, ou seja, um tipo de árvore ser menos representado (sub) ou mais (sobre) do que outro, e a representação não necessita de operadores especiais [57].

A representação NetKeys consiste de um *array* de números reais de comprimento $m = n(n-1)/2$, onde cada posição representa uma das arestas do grafo. Para qualquer seqüência de chaves $r = r_0, \ldots, r_{m-1}$, a permutação σr de r é definida como a seqüência com elementos $\sigma r_i = r_{\sigma(i)}$. A permutação σ, de σr, é a ordem decrescente da seqüência de chaves, representada por r^s, com a ordenação $(\sigma(0), \ldots, \sigma(m-1))$, isto é, uma permutação de uma seqüência de chaves é a lista em ordem decrescente das chaves [57].

Uma árvore representada por NetKeys pode ser decodificada pelo Algoritmo 5.11.

Algoritmo 5.11 Pseudo-código de Decodificação da Representação NetKeys.

(1) Seja $i = 0$, G um grafo vazio com n nós e r^s a permutação de comprimento m de r. Todas as arestas de G são numeradas de $1, \ldots, m$;
(2) Seja j o número da $i-$ésima posição da permutação r^s;
(3) Se a inserção da aresta com número j em G não forma ciclo, então insere a aresta j em G;
(4) Pare, se existem $(n-1)$ arestas em G;
(5) Incremente i e retorne ao passo (2).

Como exemplo da utilização de NetKeys, considere a árvore da Figura 9 obtida a partir da seqüencia de chaves da Tabela 4, da permutação $r^s = 8 \rightarrow 1 \rightarrow 6 \rightarrow 2 \rightarrow 5 \rightarrow 10 \rightarrow 7 \rightarrow 3 \rightarrow 9 \rightarrow 4$ e da aplicação do Algoritmo 5.11.

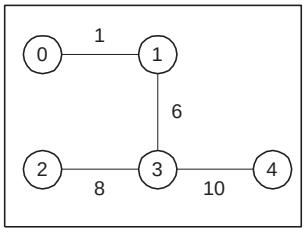


Figura 9: Exemplo de uma árvore com 5 nós.

posição	1	2	3	4	5	6	7	8	9	10
valor	0.82	0.69	0.31	0.12	0.47	0.76	0.38	0.97	0.21	0.45
aresta	0-1	0-2	0-3	0-4	1-2	1-3	1-4	2-3	2-4	3-4

Tabela 4: Representação por *NetKeys* para a árvore da Figura 9.

A representação NetKeys possui alta localidade e hereditariedade, pode ser utilizada com operadores de crossover e mutação padrões sem obter soluções infactíveis e o processo de decodificação tem complexidade de tempo $O(mlog(m))$, onde $m = n(n-1)/2$ que é o tempo necessário para construir a lista de permutação ordenada [57, 59].

5.9 Permutação Baseada em Árvore

Na representação Permutação Baseada em Árvore (PBA) utiliza uma matriz de 2 linhas com n colunas, para uma árvore de n nós. A primeira dimensão da matriz representa a ordem dos nós nas árvores, ou seja uma permutação dos valores de 1 até n. A segunda dimensão representa o grau de cada nó. Cada posição da segunda linha, pode assumir valores entre 1 e b, sendo b o maior grau da árvore [70].

A Algoritmo 5.12 apresenta os passos para codificação de uma árvore na representação permutação baseada em árvore.

Algoritmo 5.12 Algoritmo de codificação da representação permutação baseada em árvore [70].

(1) Seleciona o nó 1 (nó raiz) como o nó atual e insira esse nó no primeiro digito da primeira dimensão da permutação e o valor do seu grau como primeiro digito na dimensão de grau;
(2) Verifique todos os nós sucessores do nó atual da esquerda para direita. Se existem nós sucessores, pegue o sucessor mais a esquerda para ser o nó atual e vá para o passo (3), caso contrário vá para o passo (4);
(3) Coloque o nó atual na dimensão de nós e seu grau na dimensão de graus e retorne ao passo (2);
(4) Remova o nó atual e suas arestas adjacentes da árvore, faça seu nó predecessor ser o nó atual;
(5) Se todos os nós foram verificados, pare, caso contrário retorne ao passo (2).

A Figura 10 apresenta um exemplo da representação permutação baseada em árvore. No caso da população inicial de um AG, as representações podem ser obtidas de forma aleatória. No entanto, para manter a conectividade entre os nós, as posições na dimensão de graus precisa satisfazer algumas condições. O valor do grau de cada nó deve ser maior que 1 e menor b. O valor de b deve garantir que a soma do valor total dos graus é $2(n-1)$ [70].

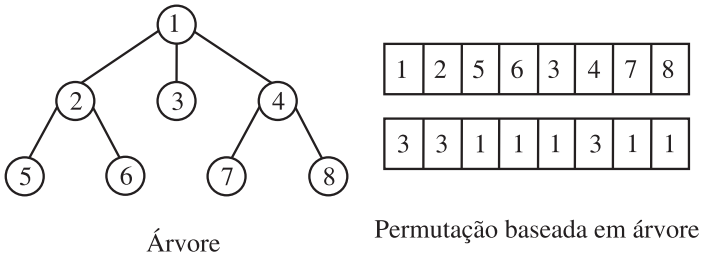


Figura 10: Exemplo de uma árvore com 8 nós e a representação permutação baseada em árvore.

Suponha que a dimensão de nós para uma codificação P é representada como $P_1(k)$, $k = 1, 2, \ldots, n$ e a dimensão de graus por $P_2(k)$. O processo de decodificação está descrito no Algoritmo 5.13. Qualquer árvore pode ser codificada por essa representação, cada representação codifica uma única árvore e diferentes cromossomos podem representar a mesma árvore [70].

Uma característica da representação é manter a estrutura da árvore. A representação possui alta localidade. Com o objetivo de gerar somente soluções factíveis a representação

Algoritmo 5.13 Algoritmo de decodificação da representação permutação baseada em árvore [70].

(1) Inicialize $k \leftarrow 1$ e $j \leftarrow 2$;
(2) Selecione o nó $r = P_1(k)$ e o nó $s = P_1(j)$, adicione a aresta de r para s na árvore;
(3) Faça $P_2(k) \leftarrow P_2(k) - 1$ e $P_2(j) \leftarrow P_2(j) - 1$;
(4) Se $P_2(k) = 0$, então $k \leftarrow k + 1$; caso contrário vá para o passo (6);
(5) Se $j = n$, pare, caso contrário retorne ao passo (4);
(6) Se $P_2(j) \geqslant 1$, então $k \leftarrow j$, $j \leftarrow j + 1$, retorne ao passo (2); caso contrário $j \leftarrow j + 1$, retorne ao passo (2).

utiliza operadores de recombinação e mutação específicos.

5.10 *D-Based*

Esta representação consiste de uma *string* de números inteiros representando os nós de comprimento $d * n$, onde n é o número de nós e d o número de vezes que o nó aparece na *string* [62].

Por exemplo, considere a *string* $(1, 3, 1, 2, 4, 5, 2, 3, 4, 5)$ de comprimento $2n$, onde $n = 5$. Conectando-se cada par de nós adjacentes na *string* obtém-se o grafo correspondente $(1, 3)$, $(3, 1)$, $(1, 2)$, $(2, 4)$ e assim por diante.

Para decodificar uma árvore geradora, desse grafo, utiliza-se a seguinte regra de construção de árvores. Considere os conjuntos $SelNode\{\emptyset\}$ e $EdgeSet\{\emptyset\}$. Comece com par de nós mais à esquerda na *string*, $(1, 3)$ no exemplo, conecte-os e atualize os conjuntos $SelNode\{1, 3\}$ e $EdgeSet\{(1, 3)\}$. A seguir considere o nó 1 da *string* e verifique o par $(3, 1)$, como este par representa uma aresta que já existe na árvore, considere o nó seguinte 2 e o par $(1, 2)$. Como $2 \notin SelNode$, então conecte-o à árvore, isto é, faça $SelNode\{1, 3, 2\}$ e $EdgeSet\{(1, 3), (1, 2)\}$. Considere o nó seguinte 4 na *string* e o par $(2, 4)$. Novamente $4 \notin SelNode$, então, conecte o novo nó a árvore: $SelNode\{1, 3, 2, 4\}$ e $EdgeSet\{(1, 3), (1, 2), (2, 4)\}$. O próximo nó é 5, e $5 \notin SelNode$. Assim, adicione 5 em $SelNode\{1, 3, 2, 4, 5\}$ e o par $(4, 5)$ em $EdgeSet\{(1, 3), (1, 2), (2, 4), (4, 5)\}$, obtendo a árvore completa. A construção de árvore termina se o número de arestas selecionas atinge $n - 1$ [62].

O grau máximo de um nó é o número de vezes em que o nó aparece na *string*, portanto o comprimento da representação varia de acordo com o grau máximo desejado nas soluções. O melhor operador de recombinação para a representação é a recombinação baseada na distância de arestas segundo Soak, Corne e Byung-Ha [62]. Dados os dois pais, este operador compara as arestas nas posições correspondentes e escolhe para o filho a aresta de menor custo. O operador de mutação utilizado troca aleatoriamente duas posições da *string* [62].

5.11 Ajuste Adaptativo das Ligações

A representação Ajuste Adaptativo das Ligações (AAL) consiste em um *array* de tamanho m, onde $m = n(n-1)/2$ e n o número de nós do grafo. Cada posição do *array* corresponde a uma aresta do grafo e contém um valor que estima/indica a contribuição da aresta para a melhor solução do problema [61].

No processo de obtenção da árvore, constrói-se uma lista ordenada de forma ascendente do *array* e percorre-se a ordem resultante adicionando as arestas até obter uma árvore válida com $n - 1$ arestas. As arestas que formam ciclos são ignoradas. De acordo com o valor de aptidão da árvore obtida são aplicados dois processos adaptativos das ligações no *array*. Se a árvore obtida é melhor que a melhor árvore obtida até o momento atual, aplica-se um processo chamado α caso contrário, aplica-se um processo chamado β.

O processo α consiste em aumentar o valor das posições do *array* que estão presentes na nova melhor árvore obtida. Com isso, objetiva-se o direcionamento da busca para longe do caminho de busca da melhor árvore obtida, pois causa a perda de prioridade das arestas pertencentes a melhor árvore.

O processo β escolhe uma porção aleatória θ de arestas para ganhar prioridade. O efeito obtido com a aplicação do processo β é aumentar gradualmente a prioridade de arestas que aparecem freqüentemente em boas soluções.

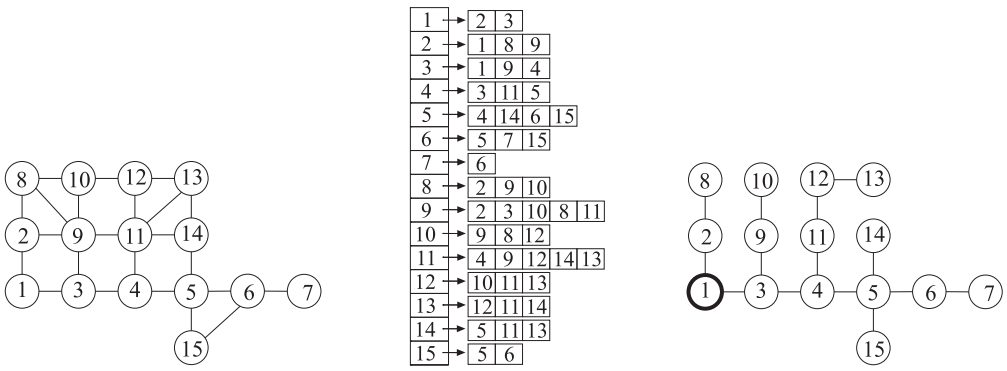
O AE que utiliza a AAL aplica operadores de crossover uniforme e mutação de troca e após a avaliação aplica os processos α e β [61]. O AE proposto foi aplicado para os problemas de árvore geradora de comunicação ótima [61] e árvore geradora mínima quadrática [61]. Segundo, Soak, Corne e Ahn [61] os resultados mostraram que o algoritmo proposto é relativamente estável e insensível a estrutura de dados, para o problema de árvore geradora de comunicação ótima (ver Seção3), o AE não apresentou no geral resultados superiores aos AEs anteriores para o problema, porém para o problema de árvore geradora mínima quadrática, obteve resultado melhor na maioria dos casos testados.

5.12 Nó-Profundidade

A representação nó-profundidade (NDE, do inglês, *Node-Depth Encoding*) é baseada nos conceitos de caminhos e de profundidade do nó em um grafo (árvore) [18]. A representação consiste de pares contendo um nó e sua profundidade (n_i, d_i). Uma árvore pode ser representada por dois *array*es, um para os nós e outro para as profundidades, de tamanho n. A ordem dos nós no *array* pode ser determinada por uma busca em profundidade [28] ou um percurso em pré-ordem em árvores [67, 3].

A Figura 11 apresenta um grafo (ver Figura 11(a)), sua lista de adjacências (ver Figura 11(b)), uma árvore geradora do grafo (ver Figura 11(c)) e sua NDE (ver Figura 11(d)) obtida por meio de uma busca em profundidade no grafo.

É importante ressaltar que esta representação além de codificar árvores geradoras pode



(a) Grafo G_1. (b) Lista de Adjacências. (c) Árvore Geradora.

$$\begin{bmatrix} \text{profundidade} \\ \text{nó} \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 4 & 5 & 3 & 4 & 4 & 5 & 4 \\ 1 & 2 & 8 & 3 & 9 & 10 & 4 & 11 & 12 & 13 & 5 & 14 & 6 & 7 & 15 \end{bmatrix}$$

(d) Representação Nó-Profundidade.

Figura 11: Exemplo da representação nó-profundidade.

representar florestas geradoras. Neste caso, cada árvore é representada por uma NDE, a floresta é representada pelo conjunto de NDEs [18] por meio de um *array* de endereços com ponteiros para cada NDE. Deve-se observar, que a maioria das representações trabalham somente com árvores geradoras [51, 37, 25, 1].

Com o objetivo de gerar novas soluções eficientemente e somente soluções factíveis, foram propostos dois operadores para gerar uma nova solução a partir de uma existente. Os operadores são similares e foram denominados Operador 1 e Operador 2.

Esses operadores produzem novas florestas geradoras F' de um grafo G quando aplicadas a outra floresta F de G [17]. Ambos os operadores transferem uma sub-árvore de uma árvore T_{origem} para $T_{destino}$. Enquanto que no operador 1 a raiz da sub-árvore podada é a mesma em T_{origem} e $T_{destino}$; no operador 2, um novo nó (diferente da raiz) é escolhido para ser a nova raiz da sub-árvore em $T_{destino}$ [18].

O operador 1 necessita a consideração de dois nós em especial: o nó p, que indica a raiz da sub-árvore a ser transferida e o nó adjacente a, que não pertence a T_{origem}, mas é adjacente a p onde a sub-árvore podada será conectada. O operador 2 requer além desses dois nós, o nó r que será a nova raiz da sub-árvore podada.

5.12.1 Operador 1

Considere que os nós p e a foram previamente escolhidos, que a implementação da representação utiliza *array*es e que são conhecidos os índices de p (i_p) e a (i_a) nos *array*es T_{origem} e $T_{destino}$, respectivamente.

O operador 1 pode ser descrito pelos passos do Algoritmo 5.14.

Para efetuar a operação do passo (1) do algoritmo, a NDE é percorrida à partir do índice do nó p até encontrar o índice i_x do primeiro nó de profundidade menor ou igual a

Algoritmo 5.14 - Operador 1 [17]

(1) Determine o intervalo (i_p, i_l) em T_{origem}, correspondente aos nós da sub-árvore enraizada em p;
(2) Copie os dados de (i_p, i_l) para um *array* temporário T_{tmp} contendo a sub-árvore podada e atualiza os valores de profundidade $d_i = d_i - d_p + d_a + 1$;
(3) Construa a nova representação $T'_{destino}$ contendo os dados de $T_{destino}$ e T_{tmp}, inserindo T_{tmp} em $T'_{destino}$ a partir de $i_a + 1$;
(4) Construa a nova representação T'_{origem} removendo os dados de T_{tmp} de T_{origem};
(5) Atualize a estrutura da nova floresta F' com as árvores não alteradas de F e as novas árvores obtidas.

profundidade de p fazendo $i_l = i_x - 1$.

5.12.2 Operador 2

O operador 2 necessita a consideração de três nós para aplicação do algoritmo: o nó a ser podado (p), a nova raiz (r) e o nó adjacente (a). Os nós p e r estão em T_{origem} e o nó a em $T_{destino}$. As diferenças entre os operadores 1 e 2 estão nos passos 2 e 3, ou seja, somente a formação e armazenamento das sub-árvores podadas em *array*es temporários são diferentes [17].

Para o operador 2, o procedimento de cópia da sub-árvore podada pode ser definido por dois passos. O primeiro passo é similar ao passo 2 do operador 1, com diferença no índice. Ao invés de ser i_p, considera-se i_r. O *array* retornado por esse procedimento é denominado T_{tmp1}.

O segundo passo considera os nós do caminho de r a p (isto é, $r_0, r_1, r_2, \ldots, r_n$, onde $r_0 = r$ e $r_n = p$) como raízes de sub-árvores, ou seja os nós com profundidades entre as profundidades de r e p. A sub-árvore enraizada por r_1 contém a sub-árvore enraizada por r_0. A sub-árvore enraizada por r_2 contém a sub-árvore enraizada por r_1 e assim por diante. O algoritmo para o segundo passo faz a cópia das sub-árvores enraizadas por r_i, onde $i \in \{1, \ldots, n\}$, sem copiar a sub-árvore enraizada por r_{i-1} e armazena as sub-árvores em um *array* temporário T_{tmp2}.

No passo 3 do operador 2, ao invés de $T'_{destino}$ ser criado a partir de $T_{destino}$ e T_{tmp}, como no passo 3 do operador 1, $T'_{destino}$ é construído utilizando os *array*es $T_{destino}$, T_{tmp1} e T_{tmp2}, inserindo primeiro T_{tmp1} a partir do índice $i_a + 1$ e em seguida T_{tmp2}.

Deve-se observar que esses operadores sempre produzem novas soluções factíveis sem a necessidade de algoritmos de reparo ou de verificação de ciclos.

5.12.3 Determinação dos nós p, r e a

O algoritmo para determinação eficiente dos nós p, r e a apropriados utiliza duas estruturas auxiliares a matriz Π_x e o *array* π . A matriz Π_x contém a identificação do nó, sua posição, sua floresta e sua árvore, sempre que um nó tem sua posição alterada uma nova coluna com as informações atualizadas é inserida na matriz. O *array* π armazena em cada posição o ancestral da floresta representada naquela posição [18]. O passos para

a determinação dos nós p, r e a é descrito pelo Algoritmo 5.15.

Algoritmo 5.15 - Determinação dos nós p, r e a [17]

(1) Escolha aleatoriamente um índice de T_{origem} diferente da raiz e chame esse nó de p;
(2) Se for operador 2 escolha aleatoriamente um índice no intervalo de (i_p, i_l) e chame de r;
(3) Escolha aleatoriamente um nó da lista de adjacências de p no caso do operador 1 e de r no caso do operador 2 e chame de a.
(4) Verifique se $a \notin T_{origem}$ utilizando a matrix Π_a e o *array* π, em caso afirmativo retorne a, seu índice i_a e chame sua árvore de $T_{destino}$; caso contrário retorne ao passo (3);

Além de possibilitar a representação de florestas geradoras, a NDE é adequada para lidar com problemas de grafos não completos, sem ocorrer custo adicional para isso. A complexidade de espaço e tempo da NDE é $O(|T_{origem}| + |T_{destino}| + f)$, onde f é o número de árvores na floresta.

5.13 Sub-Conjunto de Comprimento Fixo

A representação Sub-Conjunto de Comprimento Fixo (SCCF) foi desenvolvida para problemas onde existe um número mínimo de nós folha, chamado de l. Para este tipo de problema observa-se que, se o número folhas na árvore for maior que o número mínimo, o valor de custo da árvore aumentará, por isso fixa-se o número de folhas nas árvores como o número mínimo de folhas exigido pelo problema [33].

A representação consiste em distinguir os nós que são internos e os nós que são folhas. O espaço de busca é restringido para conjuntos de árvores geradoras com o número mínimo de folhas necessário ao problema. A árvore representada pela codificação pode ser obtida em dois passos. O primeiro passo obtém a árvore geradora mínima com os nós internos. O segundo passo conecta cada nó folha ao nó interno mais próximo, ou seja, de menor custo. Se um nó interno não possuir nenhum nó folha com aresta de menor custo relacionado para conectá-los, este nó tornar-se-á um nó folha [33].

Duas formas podem ser utilizadas para representar as árvores na SCCF: uma lista dos vértices interiores ou uma *string* binária de comprimento n, onde os nós internos são representados por 1 e os nós folha por 0. A representação é chamada de sub-conjunto de comprimento fixo pois representa somente conjuntos de comprimento fixo $n-l$. Este tipo de representação, onde somente parte da solução do problema é representada efetivamente, é utilizada em AEs detecção de *outlier*, ou seja, amostras anômalas [13], problemas em química [41] e no problema de $p-$*médio* [43, 12, 21].

No processo de decodificação da representação, o primeiro passo aplica o algoritmo de Prim para encontrar a árvore geradora mínima dos nós internos que, na implementação utilizada em [33] tem complexidade de tempo $O((n-l)^2)$. O segundo passo, encontra o nó mais próximo a cada nó folha com complexidade de tempo $O(nl)$. O tempo total para decodificação e para, no caso dos AEs, avaliação das árvores representadas é, portanto $O(n^2)$ [33]. Para a operação de recombinação é utilizado o operador proposto por Rad-

cliffe, chamado de *Random Respectful Recombination*, segundo o qual dados dois pais com representações de nós internos, seleciona-se o conjunto de nós internos da intersecção das codificações de ambos os pais, e escolhe-se aleatoriamente nós do conjunto de nós internos disjuntos das codificações dos pais até completar $(n - l)$ nós.

5.14 *Dandelion Code*

Dandelion code utiliza um dos três mapeamentos de *string* de Prüfer propostos por Piccioto [48], *Dandelion Code, Blob Code e Happy Code.* A representação utiliza uma *string* de Prüfer que tem comprimento $n - 2$, onde n é o número de nós do grafo. Esta representação possui as principais propriedades necessárias para uma representação segundo os critérios apresentados em [54]. Além disso, é mais eficiente que a representação por número de Prüfer [64, 46, 47]. A Figura 12 apresenta uma árvore de 25 nós. A Tabela 5 mostra a codificação em *Dandelion Code* desta árvore. O algoritmo para decodificação do *Dandelion Code* é descrito no Algoritmo 5.16. A Algoritmo 5.17 apresenta os passos para codificação.

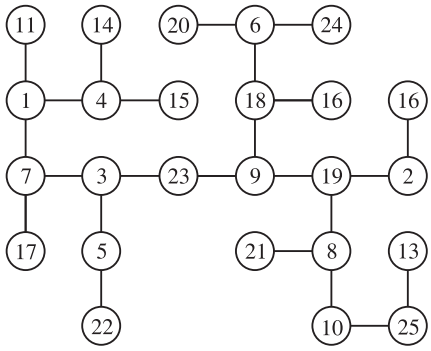


Figura 12: Exemplo de uma árvore com 25 nós.

Nós, i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Mapeamento, d_i	19	7	1	3	18	3	23	19	10	1	2	25	4	4	18	7	9	8	6	8	5	9	6

Tabela 5: Representação *Dandelion Code* da Figura 12, a codificação é representada na linha de mapeamento [46, 47].

Os passos do algoritmo de codificação de uma árvore são exatamente os passos inversos do algoritmo de decodificação:

Considere a Figura 12 para um exemplo dos algoritmos de decodificação e codificação. No primeiro passo do algoritmo de decodificação, considere a função ϕ_D que mapeia i para d_i na Tabela 5. No segundo passo, identificam-se os ciclos de ϕ_D presente na representação, no exemplo, são $(3, 7)$, um dois-ciclo, $(8, 23, 9, 19)$, um quatro-ciclo e o um-ciclo (10). Utilizando a notação definida no algoritmo têm-se $Z_1 = (7, 3)$, $Z_2 = (23, 9, 19, 8)$ e $Z_3 = (10)$.

Algoritmo 5.16 Algoritmo de Decodificação *Dandelion Code* [46, 47].

(1) Define a função $\phi_D : [2, n-1] \to [1, n]$ tal que $\phi_D(i) = d_i$ para cada $i \in [2, n-1]$ os rótulos do nós;
(2) Sejam $Z_1, Z_2, \ldots, Z_t$ os ciclos associados com a função ϕ_D e m_i o elemento de maior rótulo no ciclo Z_i. Sem perda de generalidade, assuma que os ciclos são armazenados de forma que m_i seja o elemento mais a direita de Z_i, e que $m_i > m_j$ se $i < j$;
(3) Forme uma lista simples π dos elementos em $Z_1, Z_2, \ldots, Z_t$ na ordem em que ocorrem em sua lista de ciclo, do primeiro elemento de Z_1 até o último elemento de Z_t;
(4) Para construir a árvore T correspondente a D pegue o conjunto de nós isolados, crie um caminho do nó 1 ao nó n seguindo a lista π da esquerda para direita. Então, crie a aresta (i, d_i) para todo $i \in [2, n-1]$ não presente em π.

Algoritmo 5.17 Algoritmo de Decodificação Dandelion Code [46, 47].

(1) Encontre um caminho único de 1 para n em T, e seja π a lista dos nós intermediários do caminho;
(2) Recupere os ciclos $\{Z_i, \}$ reescrevendo π em uma lista da esquerda para a direita. Um ciclo é fechado imediatamente a direita de cada nó de rótulo mínimo da direita para esquerda, ou seja, cada elemento de rótulo menor que todos os elementos a sua direita;
(3) A *string* de Dandelion correspondente a T é a única *string* $D = (d_2, d_3, \ldots, d_n - 1)$ que tem: a) os ciclos da função $\phi_D(i) = d_i$ são precisamente $\{Z_i\}$; b) para cada $i \in [2, n-1]$ não pertencente a π, o primeiro nó no caminho do vértice i para o vértice n na árvore T é $d_i = succ(i)$, onde o vértice n é considerado como a raiz de T.

No terceiro passo, a lista π é encontrada como $(7, 3, 23, 9, 19, 8, 10)$. No passo 4, a árvore única T correspondente a Dandelion é a árvore que contém o caminho $1 - 7 - 3 - 23 - 9 - 19 - 8 - 10 - 25$ e as 16 arestas $\{(i, d_i) : i \in [2, 24] \setminus \pi\}$ (ver Figura 12).

Para codificar uma árvore, o primeiro passo do algoritmo, encontra o caminho único de 1 até n, considerando os nós rotulados de 1 a n, no exemplo $n = 25$. Considera-se o nó n a raiz da árvore T. O sucessor de um nó $succ(i)$, com $i \in [1, 24]$, é definido como o primeiro vértice no caminho único do nó i até o nó raiz (ver Tabela 6).

i	$succ(i)$	i	$succ(i)$	i	$succ(i)$	i	$succ(i)$
1	7	7	3	13	25	19	8
2	19	8	10	14	4	20	6
3	23	9	19	15	4	21	8
4	1	10	25	16	18	22	5
5	3	11	1	17	7	23	9
6	18	12	2	18	9	24	6

Tabela 6: Mapeamento da função $succ(i)$ [46, 47].

Da Tabela 6 obtém-se $\pi = (7, 3, 23, 9, 19, 8, 10)$. No passo 2, recuperam-se a partir de π os ciclos Z_i. As separações são inseridas sempre que o elemento possui rótulo menor que o elemento à direita. Assim, obtém-se os ciclos $Z_1 = (7, 3)$, $Z_2 = (23, 9, 19, 8)$ e $Z_3 = (10)$.

O terceiro passo determina a representação *Dandelion* da árvore T por meio da construção do diagrama de mapeamento para ϕ_D com a inserção inicial dos ciclos Z_i e, para os demais nós, o diagrama de mapeamento é preenchido com o resultado da função $succ(i)$ para $i \in [2, n-1] \setminus \pi$. A representação da árvore correspondente é $D = (d_2, d_3, \ldots, d_{24}) = (19, 7, 1, 3, 18, 3, 23, 19, 10, 1, 2, 25, 4, 4, 18, 7, 9, 8, 5, 9, 6)$ [46, 47].

Assim como a representação de número de Prüfer, *Dandelion Code* utiliza operadores de crossover e mutação padrões, não necessitando de algoritmos de correção. A eficiência da representação é definida pela operação de maior custo, as operações padrões de crossover, mutação e o algoritmo de decodificação são $O(n)$. Assim, a representação possui complexidade de tempo $O(n)$. *Dandelion code* apresenta os melhores valores de localidade e hereditariedade das representações que utilizam *string* de Prüfer [64, 46, 47].

A Tabela 7 apresenta um resumo das principais representações de PPRs para AEs apresentadas no relatório técnico de Raidl [53] e da NDE de acordo com os critérios propostos por este relatório técnico.

Representação	Espaço	Tempo [1]	Factibilidade [2]		Localidade/ Hereditariedade
			Completo	Esparso	
CV	$O(m)$	$O(m)$	Sim	Não	Alta
Predecessores	$O(n)$	$O(n^2)$	Sim	Não	Alta
Número de Prüfer	$O(n)$	$O(n \log n)$	Sim	Não	Baixa
Blob Code	$O(n)$	$O(n^2)$	Sim	Não	Média
LNB	$O(m+n)$	$O(m+nlogn)$	Sim	Sim	Média
NetKeys	$O(m)$	$O(m \log m)$	Sim	Sim	Alta
Conjunto de Arestas	$O(n)$	$O(n)$	Sim	Sim	Alta
Nó-Profundidade	$O(\|T_{origem}\|+ \|T_{destino}\|+f)$	$O(\|T_{origem}\|+ \|T_{destino}\|+f)$	Sim	Sim	Alta
Precedentes Diretos	$O(n)$	$O(n+m)$	Sim	Sim	Alta

Tabela 7: Complexidade de Espaço e Tempo, Factibilidade, Localidade e Hereditariedade baseada em [53].

[1]Complexidade de tempo necessária para obter somente soluções factíveis em grafos completos.
[2]Capacidade de gerar somente soluções válidas para grafos completos e esparsos.

6 Estrutura de Dados Proposta: NDDE

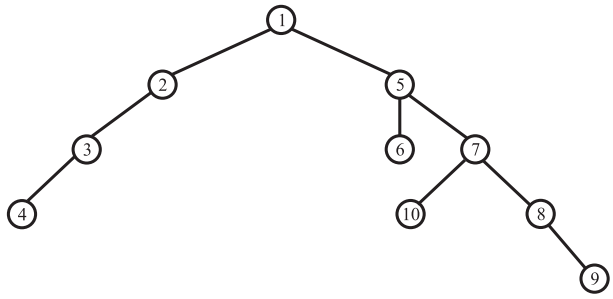
A nova estrutura de dados proposta chamada de representação nó profundidade e grau, (NDDE, do inglês, *Node-Depht-Degree Encoding*), consiste em uma melhoria da representação NDE [17] (ver Seção 5.12) por meio do desenvolvimento de operadores de recombinação e melhoria dos operadores existentes. Com isso objetiva-se a redução da complexidade de tempo da representação para permitir a solução de PPRs de larga-escala em tempo real. A maioria das representações existentes, devido a complexidade de tempo relativamente alta, não conseguem trabalhar eficientemente em tempo real para PPRs de larga-escala.

6.1 Operadores de Recombinação

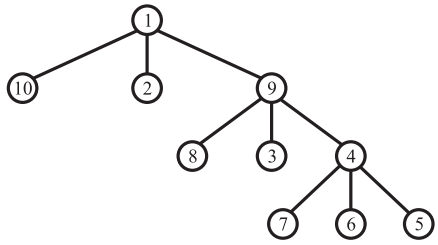
O primeiro operador de recombinação proposto (DRO, do inglês *Depths Recombination Operator*) recombina informações de duas florestas F_1 e F_2 escolhendo um ponto de crosso-

ver em suas NDEs(ver Figuras 13 e 14). Para facilitar o entendimento, considere florestas com uma única árvore. Aplica-se a operação de crossover de um ponto para o *array* de profundidades obtendo dois novos *array*es d^1 (ver Figura 16(c)) e d^2 (ver Figura 16(d)), como no operador de crossover padrão. O *array* de nós é copiado para o filho exatamente como está no pai.

Pode ocorrer o caso onde d_i^1 (d_i^2) possui profundidade maior do que d_{i-1}^1+1 (d_{i-1}^2+1), com o ponto de crossover entre os índices $i-1$ e i de d^1 (ou d^2). Isso indica uma sub-árvore desconectada do restante da árvore. Assim, um procedimento de correção no *array* é efetuado pelo re-cálculo das profundidades a partir do ponto de crossover. Este procedimento tem um custo de tempo $O(|T|)$, no pior caso, onde $|T|$ é o tamanho da árvore T. Como resultado duas novas topologias são obtidas. É importante ressaltar que qualquer *array* de profundidades da NDE corresponde a uma topologia de árvore válida independente do *array* de nós da NDE.



(a) Primeiro pai.



(b) Segundo pai.

profundidade	0	1	2	3	1	2	2	3	4	3
nó	1	2	3	4	5	6	7	8	9	10

(c) NDE do primeiro pai.

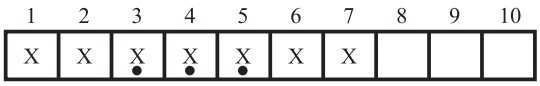
profundidade	0	1	1	1	2	2	2	3	3	3
nó	1	10	2	9	8	3	4	7	6	5

(d) NDE do segundo pai.

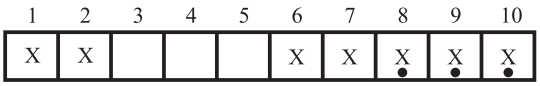
Figura 13: Pais para os operadores DRO e NDRO.

profundidade	0	1	2	3	1	2	2	3	4	3
nó	1	2	3	4	5	6	7	8	9	10
profundidade	0	1	1	1	2	2	2	3	3	3
nó	1	10	2	9	8	3	4	7	6	5

Figura 14: Crossover de um ponto (DRO e NDRO). A linha tracejada representa o ponto de crossover.



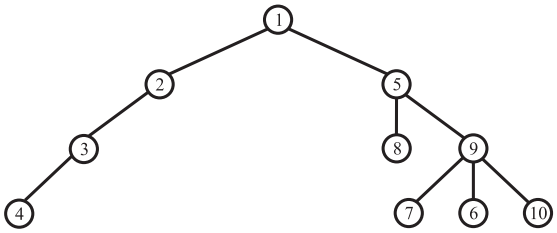
(a) Array A_1.



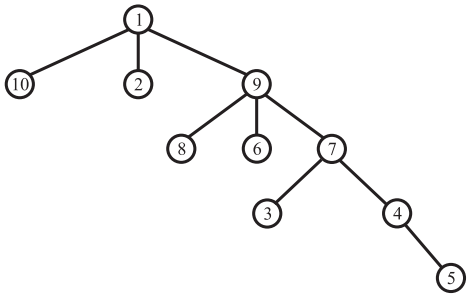
(b) Array A_2

Figura 15: Vetores auxiliares.

O segundo operador de recombinação proposto (NDRO, do inglês, *Node Depth Recombination Operator*) pode ser obtido pelo operador de crossover de um ponto (ver Seção 4)



(a) Primeiro descendente.



(b) Segundo descendente.

profundidade	0	1	2	3	1	2	2	3	3	3
nó	1	2	3	4	5	8	9	7	6	10

(c) NDE do primeiro descendente.

profundidade	0	1	1	1	2	2	2	3	4	3
nó	1	10	2	9	8	6	7	3	4	5

(d) NDE do segundo descendente.

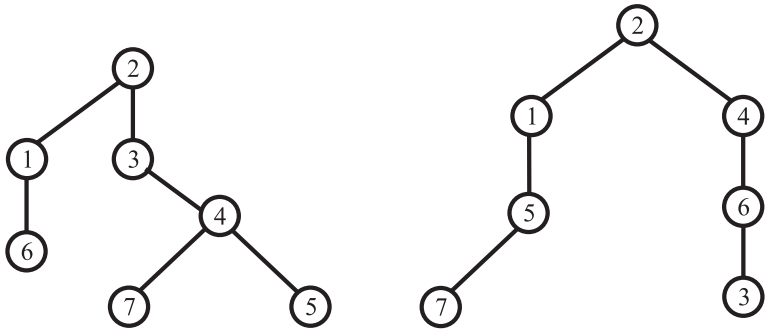
Figura 16: Descendência resultante do segundo crossover de um ponto.

aplicado ao *array* de nós da NDE e o resultado combinado com o resultado de DRO, ou seja, aplica-se o crossover de um ponto para os *arrays* de profundidade e de nó.

Este processo pode produzir dois novos filhos com nós repetidos, isto é, correspondendo a representações inválidas. Uma operação de correção pode ser obtida utilizando dois *array*es A_1 (ver Figura 15(a)) e A_2 (ver Figura 15(b)) para auxilair a cópia dos nós dos pais para os filhos. Em A_1, são marcados com um X os nós escolhidos para o primeiro filho; em A_2, marcam-se com um X os nós que compõem o segundo filho. Caso um nó já esteja marcado, ele é marcado também com um $\bullet$. Os nós que são repetidos (marcados por X e $\bullet$) em um indivíduo são os mesmos nós que estão faltando no outro indivíduo. Assim, incluindo os nós repetidos em um indivíduo nas posições vazias do outro indivíduo, obtém-se filhos válidos.

Os operadores DRO e NDRO tem a propriedade de parcialmente preservar integralmente a topologia de seus pais (ver Figuras 16(a) e 16(b)). DRO, requer tempo de execução $O(|T|)$, porém NDRO requer tempo $O(n)$, pois necessita verificar todos os nós para verificar se não existe nenhum nó repetido. Esses operadores podem significantemente colaborar em processos de otimização, reduzindo o tempo para convergência e aumentando a capacidade do algoritmo em evitar a convergência prematura para ótimos locais. DRO e NDRO possuem propriedades importantes que podem afetar positivamente a convergência de um AE ou mesmo de outra metaheurística para PPRs. Algumas características desses operadores precisão sem melhoradas para trabalhar eficientemente para grafos esparsos, evitando a geração de soluções infactíveis. Esta é uma motivação para desenvolver um operador de recombinação mais eficiente para ser aplicado em grafos esparsos.

A terceira proposta de um operador de recombinação é baseada no histórico evolutivo resultante da aplicação dos operadores 1 e 2 da NDE, chamado de EHO (do inglês, *Evolutionary History Operator*). Este histórico é armazenado na matriz Π_x da qual pode-se recuperar, na estrutura de dados, a localização dos nós de cada indivíduo. Outro *array*,



(a) Primeiro pai. (b) Segundo pai.

Figura 17: Exemplo dos pais para EHO.

chamado π, armazena o índice do ancestral de cada indivíduo. Essas estruturas já são utilizadas pela NDE (ver referências [17] e [18]).

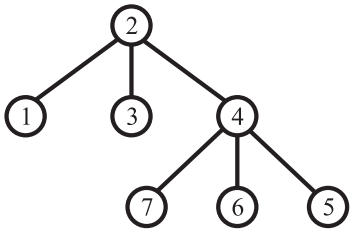
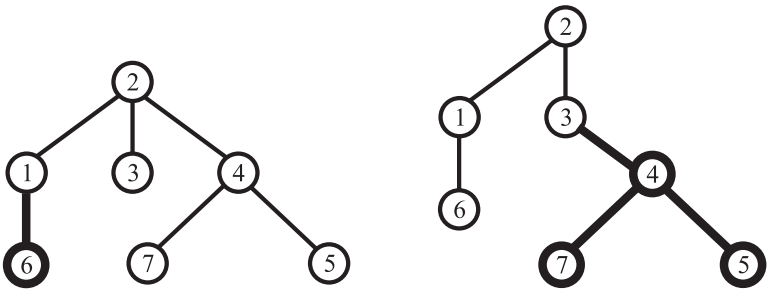


Figura 18: Ancestral comum entre os pais.



(a) Primeira modificação. (b) Segunda modificação.

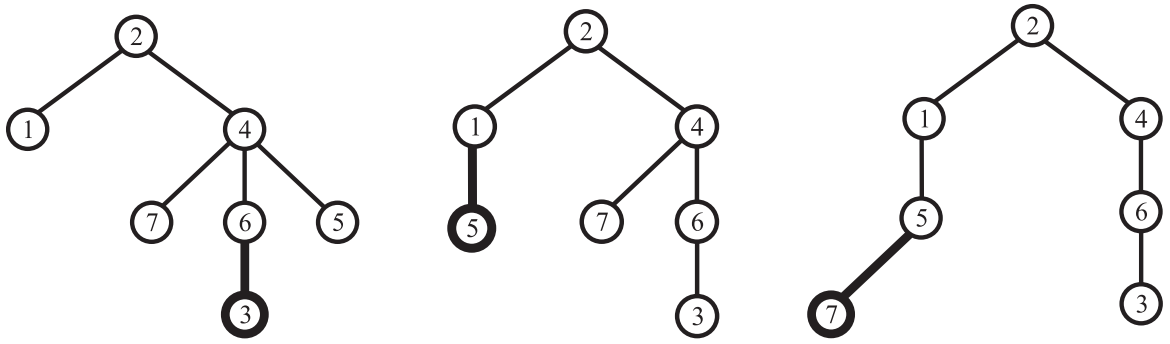
Figura 19: Modificações (arestas em negrito) aplicados para o ancestral comum (ver Figura 18) que deram origem ao primeiro pai.

Este operador consiste em:

1. Selecionar dois indivíduos I_1 (ver Figura 17(a)) e I_2 (ver Figura 17(b)) para recombinação;

2. Determinar o ancestral comum entre I_1 e I_2 utilizando o *array* π (ver Figura 18);

3. Determinar o conjunto de modificações de cada indivíduo a partir do ancestral comum (localizando a posição dos nós na matriz Π para cada um dos ancestrais) até obter o indivíduo atual. A Figura 19 apresenta as modificações para obter I_1 e a Figura 20, para obter I_2;



(a) Primeira modificação. (b) Segunda modificação. (c) Terceira modificação.

Figura 20: Modificações (arestas em negrito) aplicados para o ancestral comum (ver Figura 18) que deram origem ao segundo pai.

4. Selecionar aleatoriamente do conjunto de modificações dos dois indivíduos um subconjunto para ser aplicado ao ancestral comum;

5. Aplicar o conjunto de modificações ao ancestral comum obtendo o novo indivíduo. A Figura 21 ilustra um novo indivíduo obtido pela aplicação das modificações correspondentes as Figuras 19(a), 20(a) e 20(b).

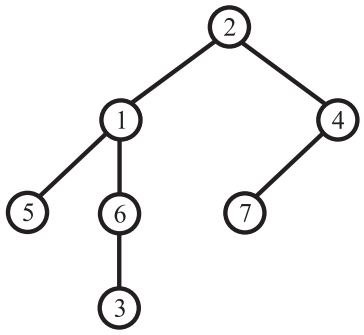


Figura 21: Novo indivíduo depois da aplicação de um subconjunto de modificações ao ancestral comum.

Uma das principais características desse operador é que parte das informações comuns entre os pais é totalmente preservada. Outra característica é que este operador pode ser aplicado para problemas de grafos esparsos sem a necessidade de mecanismos de correção de soluções infactíveis.

A complexidade de tempo para EHO é um pouco complexa necessitando de um estudo mais detalhado do tempo necessário para executar as operações dos passos 2 e 3. A operação do passo 2 pode ser executada no pior caso em tempo $O(g)$, onde g é o número de indivíduos gerados. No pior caso o ancestral comum é o primeiro indivíduo gerado. Para a operação do passo 3 a recuperação da posição de um nó em um indivíduo ocorre em tempo $O(log|\Pi_x|)$, onde $|\Pi_x|$ é o número de colunas da matriz Π_x do nó x. O ancestral em Π_x pode ser localizado por meio de uma busca binária daí a função log na complexidade do

passo 3 (ver as referências [17, 19]). Como cada nó do indivíduo requer esta operação para cada ancestral recuperado, considerando A o maior número de indivíduos até o ancestral comum e n o número de nós, a complexidade de tempo total é $O(nAlog|\Pi_x|)$. Assim, a complexidade de EHO é $O(nAlog|\Pi_x| + g)$. Limitando por uma constante o número de indivíduos gerados a partir de um ancestral comum, por exemplo por um processo de reinicialização da população como o descrito na Seção 6.2.2, g e A também ficam limitados por uma constante. Assim, a complexidade do EHO resulta em $O(nlog|\Pi_x|)$.

6.2 Melhorias nos Operadores 1 e 2 da NDE

As melhorias nos operadores 1 e 2, deram origem aos operadores chamados de PAO (do inglês, *Preserve Ancestor Operator*) e CAO (do inglês, *Change Ancestor Operador*) que estão propostos em [20]. Primeiramente, foi proposta uma modificação da representação NDE (ver Seção 5.12) acrescentando além dos *array*es nó e profundidade um *array* contendo o grau de cada nó. Assim, um nó passa a ser representado pela trinca ordenada *(nó,profundidade,grau)* e a estrutura de dados recebe o nome de nó-profundidade-grau (NDDE, do inglês *Node-Depth-Degree Encoding*).

Na NDDE, os operadores PAO e CAO, correspondem respectivamente aos operadores 1 e 2 da NDE, transferindo sub-árvores de uma árvore T_{origem} para uma árvore $T_{destino}$. No operador PAO, a raiz da sub-árvore transferida não é alterada. No operador CAO a sub-árvore transferida passa a ter uma nova raiz. Assim, PAO provoca pequenas mudanças nas árvores e CAO provoca grandes e complexas mudanças nas novas árvores geradas, principalmente em $T_{destino}$.

A seguir são apresentados os algoritmos dos operadores PAO e CAO com as melhorias propostas em [20]. Para definição do algoritmo para o operador PAO, considere que o nó p e seu índice i_p em T_{origem} e o nó a e seu índice i_a em $T_{destino}$ já foram adequadamente escolhidos. O Algoritmo 6.1 apresenta os passos para aplicação do operador PAO.

Algoritmo 6.1 - Operador PAO

(1) Determine o intervalo (i_p, i_l) em T_{origem}, correspondente aos nós da sub-árvore enraizada em p. Uma vez que i_p já é conhecido só é necessário encontrar i_l. Os nós comprendidos no intervalo correspondem a p e aos nós x consecutivos tal que $d_x > d_p$;
(2) Copie os dados de (i_p, i_l) para um *array* temporário T_{tmp} contendo a sub-árvore podada e atualiza os valores de profundidade $d_x = d_x - d_p + d_a + 1$;
(3) Construa a nova representação $T'_{destino}$ contendo os dados de $T_{destino}$ e T_{tmp}, inserindo T_{tmp} em $T'_{destino}$ a partir de $i_a + 1$. O grau de a deve ser incrementado de um;
(4) Construa a nova representação T'_{origem} removendo os dados de T_{tmp} de T_{origem}. O grau do ancestral de p deve ser diminuído de um;
(5) Copie a floresta da estrutura de dados F e gere F' alterando os ponteiros das árvores T_{origem} e $T_{destino}$ para os ponteiros das árvores T'_{origem} e $T'_{destino}$ respectivamente. Calcule o número de nós adjacentes a T'_{origem} e $T'_{destino}$ e armazene na estrutura F';
(6) Atualize as matrizes Pi_x para cada x em T'_{origem} e $T'_{destino}$.

Para aplicação de CAO, são necessários três nós como entrada: o nó corte p, a nova raiz

r e o nó adjacente a com seus respectivos índices i_p, i_r e i_a, sendo que a deve ser adjacente a r em $T_{destino}$ e p e r estão em T_de. Considere que esses nós já foram previamente encontrados para o Algoritmo 6.2, que descreve os passos para aplicação de CAO, ser utilizado.

Algoritmo 6.2 - Operador CAO

(1) Determine o intervalo (i_r, i_l) em T_{origem}, correspondente aos nós da sub-árvore enraizada em r. Os nós comprendidos no intervalo correspondem a r e aos nós x consecutivos tal que $d_x > d_r$. Este passo é similar ao passo (1) de PAO;
(2) Copie os dados de (i_r, i_l) para um *array* temporário T_{tmp1} contendo a sub-árvore de r e atualiza os valores de profundidade $d_x = d_x - d_r + d_a + 1$;
Considere os nós no caminho de r para p em T_{origem} como $r_0, r_1, \ldots, r_n$, $r = r_0$ e $p = r_n$. Construa T_{tmp2} copiando as sub-árvores enraizadas em r_i para $1 \leq i \leq n$ sem a sub-árvore enraizada em r_{i-1} e atualize a profundidade dos nós por $d_x = d_x - d_{r_i} + i + d_a + 1$. O grau de r deve ser incremento por um e o grau de p deve ser decrementado por um;
(3) Construa a nova representação $T'_{destino}$ contendo os dados de $T_{destino}$ e $[T_{tmp1}, T_{tmp2}]$, inserindo o *array* $[T_{tmp1}, T_{tmp2}]$ em $T'_{destino}$ a partir de $i_a + 1$. O grau de a deve ser incrementado de um;
(4) Construa a nova representação T'_{origem} removendo os dados de T_{tmp1} e T_{tmp2} de T_{origem}. O grau do ancestral de p deve ser diminuído de um;
(5) Copie a floresta da estrutura de dados F e gere F' alterando os ponteiros das árvores T_{origem} e $T_{destino}$ para os ponteiros das árvores T'_{origem} e $T'_{destino}$ respectivamente. Calcule o número de nós adjacentes a T'_{origem} e $T'_{destino}$ e armazene na estrutura F';
(6) Atualize as matrizes Pi_x para cada x em T'_{origem} e $T'_{destino}$.

6.2.1 Obtenção Eficiente dos Nós p, a e r

Para PAO, os nós p e a podem ser obtidos pelos seguintes passos, considerando que já foi escolhida uma floresta:

1. Escolha aleatoriamente uma árvore T_{origem} na floresta que tenha pelo menos uma aresta incidente a um nó em outra árvore. Se tal árvore não existe, G é a própria floresta;

2. Escolha aleatoriamente um nó em T_{origem}, para chamar de p, diferente da raiz, tal que o grau em T, $deg_T(p)$, é menor que seu grau em G, $deg_G(p)$;

3. Seja $N_T(p)$ o conjunto de nós adjacentes de p que estão em T_{origem} e $N_G(p)$ o conjunto de nós adjacentes a p em G. Selecione aleatoriamente um nó a de $N_G(p)/N_T(p)$;

4. Determine o índice de a na estrutura da floresta utilizando a matriz Π_a e o *array* π definidos para a NDE (ver Seção 5.12).

Para CAO, o procedimento de escolha dos nós p, r e a é o seguinte:

1. Escolha aleatoriamente uma árvore T_{origem} na floresta que tenha pelo menos uma aresta incidente a um nó em outra árvore. Se tal árvore não existe G é a própria floresta;

2. Escolha aleatoriamente um nó em T_{origem}, para chamar de r, diferente da raiz, tal que $deg_T(r) < deg_G(r)$;

3. (i) Selecione aleatoriamente um nó p de T_{origem}, diferente da raiz, tal que $d_p < d_r$, $i_p = max\{i_k | d_k = d_p; 0 < i_k < i_r\}$. Se este nó não existe, faça $p = r$;

   (ii) Seja $N_T(r)$ o conjunto de nós adjacentes de r que estão em T_{origem} e $N_G(r)$ o conjunto de nós adjacentes a r em G. Selecione aleatoriamente um nó a de $N_G(r)/N_T(r)$;

4. Determine o índice de a na estrutura da floresta utilizando a matriz Π_a e o *array* π definidos para a NDE (ver Seção 5.12).

6.2.2 Análise de Complexidade

A complexidade de tempo para obter uma nova floresta na NDDE é avaliada segundo dois limitantes: s_{op} complexidade para aplicar PAO e CAO e s_{tripla} a complexidade para encontrar os nós p, r e a, ambos como função de $|T_{origem}|$ e $|T_{destino}|$ o tamanho das árvores [20].

Analisando os passos dos Algoritmos 6.1 e 6.2 de PAO e CAO, respectivamente, tem-se claramente que os passos 1, 2 e 4 de ambos podem ser efetuados percorrendo a árvore T_{origem}, com tempo de execução $O(|T_{origem}|)$. Criar $T'_{destino}$ requer encontrar o ponto de inserção das sub-árvores em $T_{destino}$ e copiar os nós de ambos os *array*es para $T'_{destino}$. Novamente, percorrendo as árvores é possível efetuar essa tarefa com tempo de execução $O(|T_{origem}| + |T_{destino}|)$.

O passo 5 dos algoritmos de PAO e CAO, de copiar a estrutura da floresta (*array* de ponteiros para suas árvores) requer um tempo $O(t)$, com t o número de árvores da floresta. O número de adjacentes das árvores é calculado percorrendo as árvores, somando os seus graus e subtraindo o resultado do tamanho da árvore multiplicado por dois menos um. Este cálculo requer $O(|T_{origem}| + |T_{destino}|)$. Assim, o passo 5 total é $O(|T_{origem}| + |T_{destino}| + t)$.

Finalmente, o passo 6 requer novamente percorrer as árvores inserindo a coluna com índices de x em F' em Π_x. Os índices são previamente determinados e inclusão da coluna é efetuada em tempo $O(1)$. Assim, a complexidade do passo é $O(|T_{origem}| + |T_{destino}|)$, uma vez que o número de atualizações nas matrizes é limitado a $O(|T_{origem}| + |T_{destino}|)$.

Somando o tempo de todos os passos e chamando $O(|T_{origem}| + |T_{destino}|)$ de s, a complexidade é $s_{op} = O(s + t)$ [20].

Resta analisar a complexidade para encontrar os nós p, r e a. Para isso, analisa-se a complexidade dos passos definidos na Seção 6.2.1. O passo 1 pode, no pior caso, percorrer todos os ponteiros das árvores, assim o custo é $O(t)$.

O passo 2 pode, no pior caso, investigar todos os nós de T_{origem} para encontrar p, com custo de $O(|T_{origem}|)$.

O passo 3(i) para encontrar p em CAO pode ser implementado da seguinte forma:

1. Cria um *array* circular L_D de zeros com tamanho $i_r - 1$;
2. Faça $p = 0$;
3. Percorra a NDDE da posição 1 até i_r marcando para cada posição i, $L_D[d_i] = i$ se $d_i < d_r$;
4. Escolha aleatoriamente j entre 1 e $i_r - 1$;
5. Percorra L_D até encontrar um $L_D[k] \neq 0$ fazendo neste caso $p = L_D[k]$, ou percorra todo o *array*;
6. Se $p == 0$, faça $p = r$.

Assim o passo 3(i) possui complexidade $O(|T_{origem}|)$.

A implementação para o passo 3 para PAO e o passo 3(ii) de CAO utiliza três *array*es para cada nó (para o passo 3(ii) de CAO substituir p por r):

1. $L_G(p)$, contém os adjacentes de $N_G(p)$;
2. L_O é um *array* circular de comprimento $deg_G(p)$, tal que seus elementos são uma permutação de $1 \ldots deg_G(p)$;
3. L_T é um *array* binário de comprimento n.

Para encontrar o nó a, utilizam-se os seguintes passos:

1. Percorra a NDDE de T_{origem} marcando em L_T cada nó x da NDDE que está em $L_G(p)$;
2. Selecione aleatoriamente um inteiro i no intervalo $1 \ldots deg_G(p)$;
3. Se $L_T[L_G[L_O[i]]]$ é não marcado, chame $L_G[L_O[i]]$ de a e vá para o passo 4; senão continue até percorrer todo o *array* L_T;
4. Desmarque os nós da vizinhança de p em L_T utilizando a NDDE.

A complexidade do procedimento acima, depende somente dos passos para montar L_T, se considerarmos que L_G e L_O foram construídos em um processo *off-line* uma vez só antes dos operadores. L_T é inicializado somente uma vez. A operação de verificar se um nó é adjacente de p em G é efetuado em tempo constante utilizando L_G. Somente $|T_{origem}|$ verificações são necessárias, assim o passo 1 requer tempo $O(|T_{origem}|)$. O passo 2 é constante e o passo 3, como existem no máximo $|T_{origem}|$ nós marcados, pode ser

efetuado no máximo $|T_{origem}| + 1$ vezes. Desmarcar L_T também custa $|T_{origem}|$. Assim encontrar a requer $O(|T_{origem}|)$.

Portanto, os passos 1, 2 e 3 necessitam de tempo $O(|T_{origem}| + t)$. Como $|T_{origem}|$ é limitado por $|T_{origem}| + |T_{destino}|$, essas operações são também $O(s + t)$.

Para executar o passo 4, é necessário uma busca na matriz Π_a. O tempo desta busca é limitado pelo número de colunas em Π_a e pelo tamanho de L_h, a lista encadeada de ancestrais de F' em π o qual é limitado pelo número g de florestas geradas depois da inicialização de Π_x. Restringindo $g = O(\overline{s})$, o processo de busca é feito em $O(\overline{s})$.

Com o objetivo de restringir o número de colunas nas matrizes Π_x e o tamanho de L_h, a cada $k\overline{s}$ indivíduos gerados, essas estruturas de dados são reinicializadas, mantendo somente os b indivíduos presentes na população, onde $b < k\overline{s}$. O custo para a reinicialização de todas as matrizes Π_x é bn. Este custo é amortizado por $k\overline{s}$ indivíduos gerados, sem que o processo de reinicialização seja efetuado. Assim, o custo amortizado obtido é $O(\frac{bn}{k\overline{s}})$, uma vez que $\frac{b}{k}$ é uma constante, o custo amortizado final é $O(\frac{n}{\overline{s}})$. Portanto, o custo amortizado de encontrar a é $O(\overline{s} + \frac{n}{\overline{s}})$.

Em média, o tempo de execução para encontrar os nós p, r e a é $\overline{s_{triple}} = O(\overline{s} + t + \frac{n}{\overline{s}})$.

Se os n nós de um grafo são uniformemente distribuídos nas t árvores da floresta, então, o tamanho das árvores é $O(\frac{n}{t})$. Como, $\overline{s}$ é a complexidade de qualquer $s = O(|T_{origem}| + |T_{destino}|)$, então, $\overline{s} = O(\frac{n}{t})$. Considere $t = \sqrt{n}$, segue que $\overline{s} = O(\frac{n}{\sqrt{n}}) = O(\sqrt{n})$ [20].

Porém, em geral, os n nós de um grafo não estão uniformemente distribuídos em t árvores. No entanto, pode-se demonstrar que, mesmo neste caso, $\overline{s} = O(\sqrt{n})$, depois da aplicação de algumas operações da NDDE, mesmo que em alguns casos $s > O(\frac{n}{t})$.

A Equação 4 calcula o tamanho médio de s, onde Ω é o conjunto de todos os pares i, j de árvores na floresta, com $i \neq j$.

$$\sum_{(i,j)\in\Omega i\neq j} \frac{|T_i| + |T_j|}{|\Omega|} \tag{4}$$

Fixando o valor de i na soma da Equação 4, obtém-se $t-1$ termos na forma $|T_i|+|T_j|$, resultando na soma:

$$(t-1)|T_i| + \sum_j |T_j| - |T_i|.$$

Somando para todo T_i tem-se:

$$\sum_{(i,j)\in\Omega i\neq j} |T_i| + |T_j| = (t-1)\sum_i |T_i| + \sum_j |T_j| - |T_i|.$$

Dividindo por $|\Omega|$,

$$\sum_{(i,j)\in\Omega i\neq j} \frac{|T_i|+|T_j|}{|\Omega|} = \frac{(t-1)n+tn-n}{t(t-1)} = \frac{(t-1)n+n(t-1)}{t(t-1)} = \frac{2n}{t}.$$

Assim, $\overline{s} = O(\frac{n}{t})$, que resulta em $\overline{s} = O(\sqrt{n})$, quando $t = \lceil\sqrt{n}\rceil$. Quando, nenhum par de T_i e T_j possue nós adjacentes entre si, nenhuma nova árvore é produzida e, nesse caso, $\overline{s}$ da Equação 4 sobre-estima a complexidade real, pois, considera-se $|T_i|+|T_j| = 0$.

Uma vez que, $\overline{s} = O(\sqrt{n})$, então, $\overline{s_{triple}}$ é $O(\sqrt{n})$. Além disso, desde que $s_{op} = s+t$, $\overline{s_{op}} \leq \overline{s}+t$. Conclui-se que $\overline{s_{op}}$ é, também, $O(\sqrt{n})$.

Segundo Delbem e Telles [20] é possível sempre trabalhar como se houvessem $\lceil\sqrt{n}\rceil$ árvores na floresta e limitar $\overline{s}$ a $k\lceil n\rceil$).

Com os procedimentos definidos em [20] para problemas com mais ou menos $O(\lceil\sqrt{n}\rceil)$ árvores é possível resolver qualquer problema representado por florestas geradoras com tempo de execução $O(\sqrt{n})$ [20].

7 Resultados preliminares

O AE desenvolvido utilizando a NDDE, com as melhorias propostas para os operadores PAO e CAO, chamado de EAND, foi aplicado para o problema de árvore geradora mínima com restrição de grau descrito na Seção 3. Os resultados obtidos, são comparados com testes efetuados, anteriormente, para EAN, um AE utilizando a NDE, GAES, um AE utilizando a representação conjunto de arestas, e GAPE, um AE utilizando a representação de número de Prüfer.

O conjunto de dados de teste utilizados consiste de 10 grafos completos, com pesos obtidos aleatoriamente entre $1\ldots n$, com $n = 15$, 20, 30, 50, 100, 200, 300, 400, 500, 1000. E os testes foram efetuados com as restrições de grau entre $3\ldots 7$. Para cada caso de teste são efetuadas 20 execuções com 10.000 avaliações. EAN, GAES e GAPE utilizam seleção por torneio para escolher os indivíduos para reprodução e selecionam para sobreviver os indivíduos de melhor *fitness*. EAND utiliza torneio para selecionar os indivíduos para reprodução e os indivíduos que sobreviveram para a próxima geração. EAN e EAND possuem população de tamanho 10 e GAES e GAPE população de tamanho 100.

As Tabelas 8, 9, 10, 11 e 12, apresentam os resultados obtidos para cada caso de teste proposto. Cada tabela é referente a uma restrição. Em cada tabela as duas primeiras colunas são a identificação do grafo e o número de nós, as demais oito colunas, são divididas em duas para cada algoritmo testado, uma para o custo total da árvore e uma para o tempo de execução. Para cada grafo existem três linhas que representam o pior resultado, o melhor resultado e o resultado médio obtido nas 20 execuções.

Os resultados obtidos do tempo de execução de EAND (ver Tabelas 8, 9, 10, 11 e 12), em comparação com GAES e GAPE, comprova a ordem de complexidade de $O(\sqrt{n})$ mostrada para EAND. A comparação, dos resultados de tempo de execução, entre EAND

G	V	EAND		EAN		GAES		GAPE	
		Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)
1	15	35,00	0,09	23,00	0,12	23,21	0,31	45,03	0,61
		23,00	0,08	23,00	0,09	23,00	0,20	26,35	0,58
		26,30	0,09	23,00	0,11	23,11	0,22	34,49	0,59
2	20	68,00	0,11	35,00	0,15	38,69	0,30	91,72	0,67
		42,00	0,09	35,00	0,13	37,00	0,27	62,64	0,65
		52,45	0,10	35,00	0,14	37,58	0,28	75,48	0,66
3	25	102,00	0,12	44,40	0,15	46,17	0,36	151,32	0,78
		49,00	0,10	41,00	0,13	43,20	0,33	89,85	0,74
		62,65	0,11	42,47	0,14	44,66	0,34	111,88	0,76
4	30	117,00	0,13	56,00	0,17	59,80	0,43	197,52	0,83
		63,00	0,10	51,60	0,15	53,41	0,39	127,17	0,79
		87,25	0,11	53,15	0,16	55,67	0,41	158,06	0,81
5	50	337,00	0,17	117,80	0,24	155,15	0,79	635,46	1,20
		141,00	0,13	107,40	0,21	129,05	0,61	516,47	1,17
		235,10	0,15	113,84	0,22	142,03	0,67	570,19	1,18
6	100	1.164,00	0,26	575,00	0,37	860,29	1,58	3.515,09	2,96
		893,00	0,22	452,20	0,34	694,89	1,32	2.925,64	2,58
		1.019,15	0,25	503,74	0,36	760,75	1,42	3.243,70	2,72
7	200	6.443,00	0,49	3.249,60	0,69	5.001,92	3,90	16.282,78	11,45
		5.032,00	0,40	2.886,60	0,67	3.951,34	2,98	14.578,44	9,51
		5.479,70	0,45	3.044,78	0,68	4.628,76	3,30	15.541,64	10,47
8	300	17.307,00	0,73	10.079,10	1,01	14.328,55	7,07	38.956,27	27,79
		13.520,00	0,61	8.414,50	0,96	12.170,92	4,47	34.989,71	23,00
		15.339,60	0,68	9.233,88	0,98	13.462,37	5,39	37.005,29	25,22
9	400	34.095,00	0,97	21.780,40	1,34	32.286,71	9,78	70.785,82	49,18
		29.448,00	0,86	18.735,30	1,28	24.502,66	6,40	64.771,18	43,15
		31.365,45	0,92	20.422,26	1,30	28.041,07	7,43	68.109,30	46,07
10	500	60.112,00	1,23	40.708,40	1,68	54.124,20	14,14	113.803,67	82,69
		53.406,00	1,07	36.552,60	1,60	47.049,52	8,27	105.498,67	69,54
		56.518,75	1,16	38.299,35	1,62	50.619,70	9,82	109.249,88	77,33
11	1000	333.371,00	2,77	246.958,59	3,43	295.150,15	40,39	467.969,19	389,41
		299.644,00	2,26	230.556,30	3,22	258.136,39	19,76	440.836,94	328,50
		312.156,65	2,53	237.984,89	3,28	270.824,83	25,53	455.870,07	360,74

Tabela 8: Resultados obtidos para restrição de grau 3.

G	V	EAND		EAN		GAES		GAPE	
		Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)
1	15	27,00	0,09	23,00	0,12	23,34	0,23	52,76	0,61
		23,00	0,07	23,00	0,09	23,01	0,19	27,42	0,58
		23,70	0,08	23,00	0,11	23,12	0,21	37,69	0,59
2	20	55,00	0,10	36,00	0,14	36,19	0,30	86,93	0,68
		35,00	0,08	35,00	0,12	35,11	0,28	59,60	0,65
		42,25	0,09	35,38	0,13	35,47	0,29	70,60	0,66
3	25	63,00	0,12	42,90	0,15	45,53	0,36	137,07	0,79
		43,00	0,09	41,00	0,12	41,38	0,32	83,04	0,75
		54,20	0,10	41,78	0,14	42,91	0,34	114,19	0,77
4	30	88,00	0,12	53,00	0,20	57,08	0,40	177,27	0,84
		55,00	0,09	50,60	0,17	51,20	0,37	140,45	0,81
		69,75	0,11	51,83	0,19	54,01	0,39	159,58	0,83
5	50	246,00	0,15	122,10	0,23	152,33	0,75	628,50	1,22
		146,00	0,12	108,40	0,20	114,62	0,62	459,03	1,17
		187,15	0,14	113,33	0,22	132,19	0,67	545,25	1,19
6	100	1.003,00	0,25	520,90	0,37	864,40	1,56	3.464,34	3,29
		742,00	0,21	479,20	0,35	607,38	1,30	2.726,15	2,73
		840,15	0,23	498,95	0,36	734,25	1,42	3.198,92	2,95
7	200	6.028,00	0,42	3.265,10	0,71	5.285,35	3,79	16.391,57	13,28
		4.655,00	0,36	2.724,10	0,68	4.105,78	2,83	14.657,64	10,41
		5.203,20	0,39	3.064,18	0,69	4.557,91	3,15	15.477,69	11,70
8	300	16.914,00	0,61	9.638,40	1,02	14.519,09	6,59	39.760,38	29,95
		14.019,00	0,51	8.626,40	0,98	11.489,88	4,47	36.348,46	24,40
		15.585,15	0,57	9.168,80	0,99	13.169,55	5,29	37.680,52	27,58
9	400	37.675,00	0,83	21.055,20	1,34	31.325,54	8,84	72.807,30	59,24
		30.408,00	0,65	19.037,60	1,28	26.013,92	6,40	65.658,85	47,50
		34.147,50	0,73	20.100,43	1,30	27.951,94	7,11	69.437,25	52,31
10	500	64.476,00	1,01	39.176,50	1,68	53.225,75	13,17	114.808,64	97,14
		56.799,00	0,77	36.326,30	1,60	46.316,74	8,05	105.809,34	80,91
		60.684,30	0,90	37.544,59	1,62	49.380,22	9,65	109.286,27	87,02
11	1000	346.554,00	2,01	247.202,30	3,47	295.150,15	40,39	469.844,22	429,71
		321.351,00	1,63	233.390,20	3,26	258.136,39	19,76	448.828,88	389,28
		328.535,45	1,86	239.896,50	3,33	270.824,83	25,53	457.197,96	407,94

Tabela 9: Resultados obtidos para restrição de grau 4.

G	V	EAND		EAN		GAES		GAPE	
		Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)
1	15	27,00	0,09	23,00	0,13	23,35	0,24	41,87	0,61
		23,00	0,07	23,00	0,11	23,04	0,20	27,92	0,58
		23,70	0,08	23,00	0,12	23,20	0,22	32,56	0,59
2	20	46,00	0,10	36,00	0,15	36,33	0,31	93,32	0,68
		35,00	0,08	35,00	0,13	35,38	0,28	59,01	0,65
		39,05	0,09	35,40	0,14	35,81	0,29	71,61	0,66
3	25	68,00	0,11	42,80	0,16	42,82	0,35	132,39	0,75
		44,00	0,09	41,50	0,13	41,09	0,32	89,85	0,73
		52,65	0,10	42,01	0,14	41,76	0,34	109,88	0,74
4	30	81,00	0,12	52,90	0,19	58,43	0,42	187,07	0,83
		54,00	0,09	51,60	0,15	50,82	0,36	129,75	0,80
		64,75	0,11	52,14	0,17	54,51	0,39	161,78	0,82
5	50	259,00	0,15	120,90	0,23	150,80	0,74	645,63	1,24
		140,00	0,13	103,80	0,20	124,00	0,61	471,64	1,19
		184,75	0,14	111,79	0,22	140,24	0,66	570,83	1,21
6	100	975,00	0,25	511,70	0,38	958,98	1,49	3.403,53	3,34
		621,00	0,21	453,90	0,35	685,05	1,34	2.867,18	2,74
		843,05	0,23	486,73	0,36	763,17	1,40	3.164,97	2,96
7	200	6.319,00	0,42	3.214,50	0,70	5.111,18	4,14	16.315,44	13,24
		4.386,00	0,34	2.586,40	0,67	4.031,59	2,87	14.246,45	10,88
		5.098,30	0,37	2.975,40	0,68	4.499,77	3,23	15.606,71	12,06
8	300	17.431,00	0,57	9.690,90	1,02	15.276,62	6,62	38.998,08	30,96
		14.058,00	0,47	8.500,90	0,98	11.922,60	4,62	35.696,13	24,64
		15.419,80	0,53	9.189,22	0,99	13.182,79	5,24	37.175,68	28,01
9	400	37.553,00	0,76	21.657,90	1,36	31.873,93	9,61	70.626,33	56,79
		30.962,00	0,59	19.385,30	1,29	26.294,55	6,84	66.681,87	48,71
		33.686,40	0,70	20.372,73	1,31	28.426,04	7,69	68.588,84	53,57
10	500	67.287,00	0,94	38.890,40	1,69	53.372,19	13,81	113.111,00	96,34
		55.801,00	0,76	35.610,60	1,62	45.530,62	8,45	105.344,65	80,68
		61.589,15	0,85	37.774,14	1,63	49.138,57	10,51	109.091,32	87,04
11	1000	351.029,00	1,94	245.539,50	3,42	279.349,31	35,61	469.361,47	448,39
		326.209,00	1,40	236.046,20	3,25	262.105,30	18,88	446.833,91	388,65
		337.237,35	1,67	240.807,29	3,32	269.933,20	25,80	458.721,03	418,14

Tabela 10: Resultados obtidos para restrição de grau 5.

e EAN, apesar de não apresentar, exatamente, uma redução de tempo da $O(\sqrt{n})$, mostra que EAND possui um tempo de execução menor. Neste caso, o resultado teórico da complexidade de tempo não é verificado na prática, pois, apesar de teoricamente não garantir uma complexidade de tempo $O(\sqrt{n})$ para NDE, esta tem um desempenho, na prática, próximo a $O(\sqrt{n})$.

A Figura 22 apresenta um gráfico comparativo do tempo de execução de cada um dos algoritmos. Observa-se que EAND possui o melhor desempenho de todos os AEs testados e para os casos maiores de 500 e 1000 nós tem um desempenho 10 vezes melhor do que GAES, que utiliza a representação conjunto de arestas que é uma das melhores representações presentes na literatura.

Comparando os valores de custo EAND obteve um desempenho pior do que EAN e GAES. Um dos motivos para esse desempenho deve-se a EAND não selecionar para a próxima geração somente os indivíduos de melhor *fitness*. Outro motivo é que o algoritmo desenvolvido ainda necessita do ajuste de alguns parâmetros, tais como a taxa de aplicação de cada um dos operadores propostos em [20] e o número de indivíduos gerados sem a aplicação da operação de reinicialização das estruturas de dados Π_x e π.

Os próximos experimentos deste projeto serão realizados de acordo com os critérios propostos em [5]. Bartz-Beielstein [5] discute sobre o planejamento de experimentos e métodos para validação dos experimentos realizados, voltado para experimentos da área de computação evolutiva. O objetivo é fornecer aos pesquisadores de computação evolutiva bons critérios para validação dos métodos desenvolvidos. Assim, espera-se que com a aplicação do conhecimento presente em [5] desenvolver melhores experimentos para uma

G	V	EAND		EAN		GAES		GAPE	
		Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)
1	15	31,00	0,09	23,00	0,12	23,36	0,23	42,53	0,60
		23,00	0,07	23,00	0,10	23,00	0,21	27,92	0,58
		24,05	0,08	23,00	0,11	23,14	0,23	35,54	0,59
2	20	55,00	0,10	35,80	0,14	37,05	0,28	96,93	0,70
		35,00	0,08	35,00	0,12	35,01	0,26	56,49	0,67
		40,45	0,09	35,44	0,13	35,78	0,27	70,87	0,69
3	25	70,00	0,11	43,90	0,14	42,40	0,35	154,17	0,75
		43,00	0,09	41,60	0,12	41,17	0,32	90,80	0,72
		50,45	0,10	42,15	0,13	41,61	0,33	110,26	0,74
4	30	94,00	0,11	56,00	0,19	59,17	0,42	187,97	0,83
		55,00	0,09	51,00	0,15	51,87	0,37	127,66	0,80
		67,00	0,10	52,60	0,17	53,89	0,39	152,32	0,82
5	50	236,00	0,15	119,60	0,23	155,44	0,71	637,01	1,23
		145,00	0,12	106,00	0,21	128,24	0,62	471,05	1,21
		188,05	0,14	115,33	0,22	139,90	0,66	560,73	1,22
6	100	969,00	0,30	522,10	0,37	863,32	1,74	3.365,43	3,41
		724,00	0,20	454,90	0,34	664,98	1,29	2.872,99	2,76
		817,35	0,23	499,05	0,36	761,17	1,41	3.167,00	2,98
7	200	5.768,00	0,41	3.262,80	0,71	4.991,93	4,13	16.342,30	13,50
		4.545,00	0,33	2.745,00	0,66	3.972,39	2,92	13.674,18	10,95
		5.045,50	0,37	3.003,41	0,69	4.531,58	3,35	15.428,49	12,25
8	300	17.294,00	0,62	9.699,70	1,02	14.366,56	6,13	40.576,92	31,48
		14.185,00	0,45	8.558,90	0,98	11.964,23	4,41	35.272,21	26,22
		15.639,00	0,53	9.121,88	0,99	13.256,71	4,99	37.151,14	28,76
9	400	37.162,00	0,78	21.635,60	1,36	29.341,20	11,29	73.830,73	58,24
		31.548,00	0,62	19.191,20	1,28	25.223,23	6,10	66.509,75	48,87
		33.856,45	0,69	20.176,10	1,31	27.916,51	7,66	69.441,18	53,55
10	500	66.914,00	0,94	39.748,30	1,71	54.622,36	14,12	112.514,41	93,34
		56.478,00	0,74	35.559,40	1,62	46.404,12	8,13	103.595,68	78,53
		61.457,80	0,84	37.726,13	1,64	50.485,28	9,74	108.832,31	87,89
11	1000	350.946,00	1,80	243.453,59	3,45	283.804,61	45,10	468.680,00	470,10
		318.993,00	1,47	233.179,50	3,26	259.056,07	20,44	445.181,03	397,59
		338.174,75	1,60	239.003,87	3,32	270.488,32	28,48	458.341,28	423,29

Tabela 11: Resultados obtidos para restrição de grau 6.

G	V	EAND		EAN		GAES		GAPE	
		Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)	Custo	Tempo(s)
1	15	37,00	0,09	23,00	0,14	23,16	0,24	42,53	0,60
		23,00	0,08	23,00	0,10	23,00	0,20	27,92	0,58
		25,10	0,08	23,00	0,12	23,08	0,22	35,87	0,59
2	20	56,00	0,10	35,50	0,17	35,66	0,34	96,93	0,68
		36,00	0,07	35,00	0,13	35,04	0,25	54,98	0,65
		41,80	0,09	35,18	0,15	35,33	0,28	73,21	0,67
3	25	66,00	0,11	42,00	0,15	43,82	0,36	154,17	0,75
		41,00	0,09	41,00	0,13	41,67	0,32	91,84	0,72
		50,20	0,10	41,36	0,14	42,43	0,34	109,66	0,74
4	30	91,00	0,12	52,70	0,17	57,21	0,40	205,80	0,83
		55,00	0,09	51,60	0,15	51,91	0,37	125,45	0,80
		71,50	0,11	52,07	0,17	53,71	0,39	161,55	0,82
5	50	232,00	0,15	119,90	0,22	156,36	0,71	629,27	1,22
		139,00	0,12	105,70	0,20	126,52	0,61	522,08	1,18
		181,35	0,14	114,82	0,22	142,79	0,65	572,93	1,20
6	100	1.003,00	0,24	510,60	0,38	844,77	1,64	3.437,23	3,32
		720,00	0,20	470,00	0,35	652,18	1,29	2.838,73	2,72
		828,65	0,22	493,37	0,36	748,84	1,39	3.202,90	2,97
7	200	6.003,00	0,43	3.221,00	0,70	5.068,24	3,81	16.758,14	13,49
		4.783,00	0,34	2.857,20	0,67	3.790,56	2,90	15.030,52	10,50
		5.214,80	0,37	3.024,61	0,68	4.532,52	3,17	15.648,25	12,21
8	300	16.730,00	0,58	9.873,10	1,02	14.628,12	6,53	38.743,41	31,40
		14.152,00	0,49	8.769,90	0,97	11.939,42	4,50	35.826,66	24,80
		15.479,00	0,54	9.201,38	0,98	13.246,79	5,14	37.331,27	28,40
9	400	36.746,00	0,77	21.036,60	1,36	31.018,64	10,42	69.988,63	58,40
		29.696,00	0,61	18.751,10	1,29	24.373,57	6,23	66.918,02	47,06
		33.583,05	0,68	20.098,86	1,31	27.885,44	7,48	68.709,98	52,93
10	500	66.873,00	0,94	39.079,40	1,70	53.028,85	11,20	111.975,95	98,98
		55.492,00	0,71	35.767,10	1,61	44.562,07	8,19	104.612,80	81,37
		61.827,70	0,83	37.642,46	1,64	49.635,46	9,25	108.795,91	89,60
11	1000	345.141,00	1,84	247.993,30	3,51	283.297,62	37,41	469.527,78	450,44
		324.743,00	1,32	233.080,00	3,29	257.058,89	20,79	450.166,94	404,10
		335.818,65	1,54	239.816,86	3,36	271.043,89	26,25	459.123,32	427,06

Tabela 12: Resultados obtidos para restrição de grau 7.

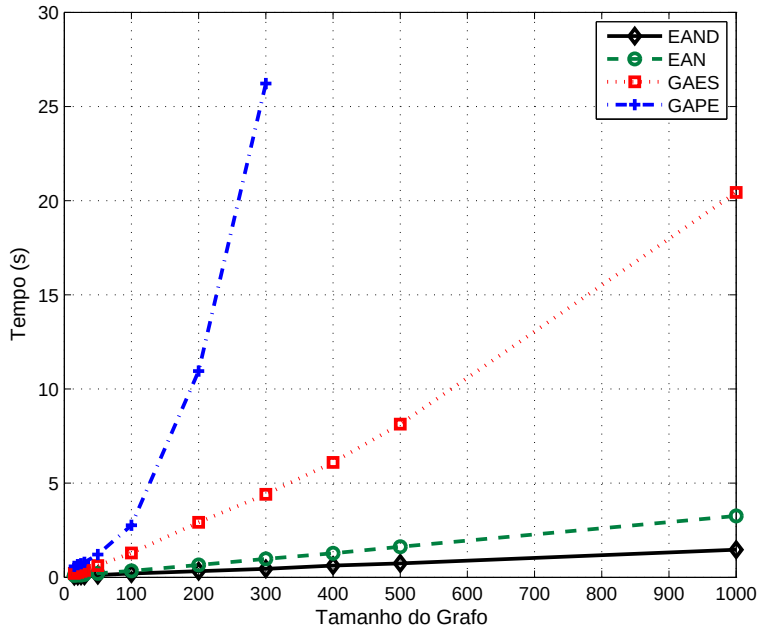


Figura 22: Tempo de Execução de EAND, EAN, GAES e GAPE.

validação confiável dos resultados deste projeto de pesquisa.

8 Considerações finais

Este trabalho investiga a aplicação da NDE para PPRs e o desenvolvimento de melhorias na NDE com o objetivo de diminuir o tempo de convergência dos AEs que utilizam a NDE.

As melhorias na NDE consistem no desenvolvimento dos operadores de recombinação apresentados neste trabalho e nas alterações propostas em [20] para os operadores PAO e CAO. A aplicação dessas melhorias resultou na representação chamada NDDE.

Os resultados apresentados neste trabalho mostram que o AE com a NDDE, mesmo sem os operadores de recombinação, apresenta uma grande melhora no tempo de computação em comparação com AEs utilizando as representações de Prüfer e Conjunto de Arestas, considerada uma das melhores representações pela literatura.

Referências

[1] F. N. Abuali, D. A. Schoenefeld, and R. L. Wainwright. Designing telecommunications networks using genetic algorithms and probabilistic minimum spanning trees. In *SAC '94: Proceedings of the 1994 ACM symposium on Applied computing*, pages 242–246, New York, NY, USA, 1994. ACM Press.

[2] F. N. Abuali, R. L. Wainwright, and D. A. Schoenefeld. Determinant factorization: A new encoding scheme for spanning trees applied to the probabilistic minimum spanning tree problem. In L. J. Eshelman, editor, *Proc. of the Sixth Int. Conf. on Genetic Algorithms*, pages 470–477, San Francisco, CA, 1995. Morgan Kaufmann.

[3] M. J. Atallah and S. Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1998. Produced By-Suzanne Lassandro.

[4] G. Ausiello, P. Crescenzi, V. Kann, Marchetti-sp, G. Gambosi, and A. M. Spaccamela. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, Heidlberg, 2003.

[5] T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation: The New Experimentalism (Natural Computing Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] D. J. Bertsimas. The probabilistic minimum spanning tree problem. *Networks*, 20(3):245–275, 1990.

[7] A. Z. Broder. Generating random spanning trees. In *FOCS*, pages 442–447. IEEE, 1989.

[8] R. S. Cahn. *Wide area network design: concepts and tools for optimization*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[9] A. P. L. F. Carvalho, A. C. B. Delbem, E. V. Simões, G. Telles, and R. Romero. Computação bioinspirada. In *Anais XXIII Jornada de AtualizaÃ§Ã£o em Informatica, JAI4*, pages 2248 – 2298, 2004.

[10] P. M. S. Carvalho, L. A. F. M. Ferreira, and L. M. F. Barruncho. Solving radial topology constrained problems with evolutionary algorithms. In *SEAL'98: Selected papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning*, pages 58–65, London, UK, 1999. Springer-Verlag.

[11] P. M. S. Carvalho, L. A. F. M. Ferreira, and L. M. F. Barruncho. On spanning-tree recombination in evolutionary large-scale network problems - application to electrical distribution planning. *IEEE Trans. Evolutionary Computation*, 5(6):623–630, 2001.

[12] E. S. Correa, M. T. A. Steiner, A. A. Freitas, and C. Carnieri. A genetic algorithm for the P-median problem. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 1268–1275, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[13] K. D. Crawford, D. J. Vasicek, and R. L. Wainwright. Detecting multiple outliers in regression data using genetic algorithms. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 351–356, New York, NY, USA, 1995. ACM Press.

[14] P. Crescenzi and V. Kann. *A compendium of NP optimization problems*. http://www.nada.kth.se/viggo/problemlist/compendium.html, 2005.

[15] L. Davis, D. Orvosh, A. Cox, and Y. Qiu. A genetic algorithm for survivable network design. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 408–415, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[16] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[17] A. Delbem, A. de Carvalho, C. A. Policastro, A. K. Pinto, K. Honda, and A. C. Garcia. Node-depth encoding for evolutionary algorithms applied to network design. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 678–687, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag. GECCO-2004 A joint meeting of the thirteenth international conference on genetic algorithms (ICGA-2004) and the ninth annual genetic programming conference (GP-2004).

[18] A. C. B. Delbem and A. de Carvalho. A forest encoding for evolutionary algorithms applied to design problems. *Genetic Algorithm and Evolutionary Computation Conference 20003, Lecture Notes in Computer Science*, 2723:634–635, 2003.

[19] A. C. B. Delbem, A. de Carvalho, and N. G. Bretas. Optimal energy restoration in radial distribution systems using a genetic approach and graph chain representation. *Electric Power Systems Researchs*, 67/3:197–205, 2003.

[20] A. C. B. Delbem and G. P. Telles. Efficient forest data structure for network design. Technical report, University of Sao Paulo, 2007.

[21] V. Estivill-Castro and R. Torres-Velázquez. Hybrid genetic algorithm for solving the p-median problem. In *SEAL'98: Selected papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning*, pages 19–25, London, UK, 1999. Springer-Verlag.

[22] S. Even. *Graph Algortithms*. Computer Science Press, 1979.

[23] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 2006.

[24] T. Gaube and F. Rothlauf. The link and node biased encoding revisited: Bias and adjustment of parameters. In *Proceedings of the EvoWorkshops on Applications of Evolutionary Computing*, pages 1–10, London, UK, 2001. Springer-Verlag.

[25] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. Wiley Series in Engineering Design and Automation. John Wiley & Sons (Sd), New York, USA, 1997.

[26] J. Gottlieb, B. A. Julstrom, G. R. Raidl, , and F. Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 343–350, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[27] J. Gross and J. Yellen. *Graph theory and its applications*. CRC Press, Inc., Boca Raton, FL, USA, 1999.

[28] J. L. Gross and J. Yellen. *Handbook of Graph Theory*. CRC Press, 2004.

[29] A. T. Haghighat, K. Faez, M. Dehghan, A. Mowlaei, and Y. Ghahremani. A genetic algorithm for steiner tree optimization with multiple constraints using prüfer number. In *EurAsia-ICT '02: Proceedings of the First EurAsian Conference on Information and Communication Technology*, pages 272–280, London, UK, 2002. Springer-Verlag.

[30] D. S. Johnson, J. K. Lenstra, and A. H. G. R. Kan. The complexity of the network design problem. *Networks*, 9:279–285, 1978.

[31] K. A. D. Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[32] B. A. Julstrom. The blob code: A better string coding of spanning trees for evolutionary search. In F. Rothlauf, editor, *Representations and Operators for Network Problems (ROPNET 2001)*, pages 256–261, San Francisco, California, USA, 7 July 2001.

[33] B. A. Julstrom. Codings and operators in two genetic algorithms for the leaf-constrained minimum spanning tree problem. *International Journal of Applied Mathematics and Computer Science*, 14(3):385–396, 2004.

[34] B. A. Julstrom. The blob code is competitive with edge-sets in genetic algorithms for the minimum routing cost spanning tree problem. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 585–590, New York, NY, USA, 2005. ACM Press.

[35] D. Jungnickel. *Graphs, Networks and Algorithms (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[36] A. Kershenbaum. *Telecommunications network design algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1993.

[37] J. D. Knowles and D. Corne. A new evolutionary approach to the degree-constrained minimum spanning tree problem. *IEEE Trans. Evolutionary Computation*, 4(2):125–134, 2000.

[38] M. Krishnamoorthy, A. T. Ernst, and Y. M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics*, 7(6):587–611, 2001.

[39] G. L. Libralao, F. C. Pereira, T. W. Lima, and A. C. B. Delbem. Node-depth encoding for evolutionary algorithms applied to multi-vehicle routing problem. In *IEA/AIE'2005: Proceedings of the 18th international conference on Innovations in Applied Artificial Intelligence*, pages 557–559, London, UK, 2005. Springer-Verlag.

[40] L. Lin and M. Gen. Node-Based Genetic Algorithm for Communication Spanning Tree Problem. *IEICE Trans Commun*, E89-B(4):1091–1098, 2006.

[41] C. B. Lucasius and G. Kateman. Towards solving subset selection problems with the aid of the genetic algorithm. In *Parallel Problem Solving from Nature 2*, pages 239–247, Amsterdam, 1992. North-Holland.

[42] P. Machado, J. Tavares, F. B. Pereira, and E. Costa. Vehicle routing problem: Doing it the evolutionary way. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO*

2002: Proceedings of the Genetic and Evolutionary Computation Conference, page 690, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[43] A. O., E. E., and D. Z. An efficient genetic algorithm for the p-median problem. *Annals of Operations Research*, 122:21–42(22), September 2003.

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

[45] C. C. Palmer and A. Kershenbaum. An approach to a problem in network design using genetic algorithms. *Networks*, 26:151–163, 1995.

[46] T. Paulden and D. Smith. From the dandelion code to the rainbow code: a class of bijective spanning tree representations with linear complexity and bounded locality. *IEEE Transactions Evolutionary Computation*, 10(2):108–123, April 2006.

[47] T. Paulden and D. Smith. Recent advances in the study of the dandelion code, happy code, and blob code spanning tree representations. In *CEC 2006. IEEE Congress on Evolutionary Computation, 2006*, pages 2111–2118, 2006.

[48] S. Picciotto. *How to Encode a Tree.* PhD thesis, University of California, 1999.

[49] M. Pióro and D. Medhi. *Routing, Flow, and Capacity Design in Communication and Computer Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[50] H. Prüfer. Neuer beweis eines satzes uber permutationen. *Arch. Math. Phys.*, 1918.

[51] G. Raidl. An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 104–111, La Jolla Marriott Hotel La Jolla, California, USA, 6-9 2000. IEEE Press.

[52] G. Raidl and C. Drexel. A predecessor coding in an evolutionary algorithm for the capacitated minimum spanning tree problem. In C. Armstrong, editor, *Proceeding of 2000 Genetic and Evolutionary Computation Conference*, pages 309–316, 2000.

[53] G. Raidl and et al. Edge-sets: An effective evolutionary coding of spanning trees.

[54] G. R. Raidl and B. A. Julstrom. Edge sets: an effective evolutionary coding of spanning trees. *IEEE Trans. Evolutionary Computation*, 7(3):225–239, 2003.

[55] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms.* Springer-Verlag, 2006.

[56] F. Rothlauf, J. Gerstacker, and A. Heinzl. On the optimal communication spanning tree problem, 2003.

[57] F. Rothlauf, D. E. Goldberg, and A. Heinzl. Network random keys: A tree representation scheme for genetic and evolutionary algorithms. *Evolutionary Computation*, 10(1):75–97, 2002.

[58] F. Rothlauf and C. Tzschoppe. On the bias and performance of the edge-set encoding. Technical report, Fakultät für Betriebswirtschaftslehre - Uni-Mannheim, 2005.

[59] B. Schindler, F. Rothlauf, and H.-J. Pesch. Evolution strategies, network random keys, and the one-max tree problem. In *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002*, pages 143–152, London, UK, 2002. Springer-Verlag.

[60] M. Sinclair. Minimum cost topology optimisation of the cost 239 european optical network. In *Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms*, AlÃ¨s, France, 1995.

[61] S.-M. Soak, D. Corne, and B.-H. Ahn. A new evolutionary algorithm for spanning-tree based communication network design. *IEICE Trans Commun*, E88-B(10):4090–4093, 2005.

[62] S.-M. Soak, D. Corne, and A. Byung-Ha. A new encoding for the degree constrained minimum spanning tree problem. In *KES*, pages 952–958, 2004.

[63] A. Syarif and M. Gen. Solving exclusionary side constrained transportation problem by using a hybrid spanning tree-based genetic algorithm. *Journal of Intelligent Manufacturing*, 14:389–399(11), 2003.

[64] E. Thompson, T. Paulden, and D. K. Smith. The dandelion code: A new coding of spanning trees for genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 11(1):91–100, February 2007.

[65] P. Toth and D. Vigo, editors. *The vehicle routing problem.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[66] C. Tzschoppe, F. Rothlauf, and H.-J. Pesch. The edge-set encoding revisited: On the bias of a direct representation for trees. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 258–270, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[67] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[68] S. Voss, I. H. Osman, and C. Roucairol, editors. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[69] G. Zhou and M. Gen. A note on genetic algorithms for degree-constrained spanning tree problems. *Networks (USA)*, 30(2):91–95, 1997.

[70] G. Zhou and M. Gen. A genetic algorithm approach on tree-like telecommunication network design problem. *J. of the Operational Research Society*, 54(3):248–254, 2003.