Um algoritmo de recozimento simulado para o problema da árvore geradora mínima com restrição de grau

José F. Lopes[a], Iago A. Carvalho[a]

[a]*Departamento de Ciência da Computação, Universidade Federal de Alfenas*

Abstract

O problema da árvore geradora mínima com restrição de grau (DMST) é definido sob um grafo G = (V, E), onde o objetivo é encontrar uma árvore geradora de G com peso mínimo e cujos vértices tenham grau menor ou igual a P. Este trabalho propõe o uso de uma meta-heurística para resolução da DMST, um algoritmo de recozimento simulado. Esse algoritmo é inspirado no processo físico de recozimento de metais, no qual um material passa por perturbações em seu estado, visando, gradualmente, alcançar um equilíbrio térmico. Na implementação proposta, as soluções (i.e. as árvores geradoras) são representadas de maneira indireta pela estrutura de dados Node-depth phylogenetic-based encoding. Além do algoritmo de recozimento simulado, é apresentado um algoritmo de seleção clonal, que também utiliza o Node-depth phylogenetic-based encoding para representar as soluções. Além da estrutura de dados, dois operadores de mutação foram utilizados, o SPRN e o TBRN, com o intuito de espelhar as perturbações impostas ao estado no metal. Os dois operadores citados são aplicados de maneira probabilística e exclusiva. A biblioteca Optuna foi utilizada para otimizar os operadores citados. Além disso, a temperatura inicial e o fator de resfriamento do recozimento simulado foram otimizados seguindo a mesma abordagem. Para

avaliar a eficácia do método proposto, o comparamos com o melhor método da literatura, um algoritmo genético híbrido que apresenta estratégias de busca local. As instâncias utilizadas foram baseadas nas instâncias de Singh e Sundar. Em conclusão, a abordagem aqui proposta se mostrou eficaz, superando o algoritmo genético híbrido em todas as instâncias, e encontrando a árvore geradora mínima do grafo quando possível.
Keywords: Recozimento simulado, Problema da árvore geradora mínima, Seleção clonal, Otimização combinatória, Meta-heurística

1. Introdução

Seja $G = (V, E)$ um grafo conectado, onde V é um conjunto de vértices e E é um conjunto de arestas. Cada aresta $e \in E$ está associada a um peso $w_e > 0$. Uma árvore geradora de um grafo é um subgrafo conectado simples que contém todos os vértices em V e não possui ciclos. O problema da árvore geradora mínima (MST, do inglês *Minimum Spanning Tree*) consiste em computar uma árvore geradora cujo somatório dos pesos de suas arestas seja minimo. Este problema aparece naturalmente em redes de comunicação, design de circuitos e análise de clusters, dentre outras áreas. Além disso, ele pode ser utilizado sempre que houver o interesse de definir uma estrutura conectada de custo mínimo entre três ou mais pontos de interesses dispersos fisicamente. O MST é resolvido em tempo polinomial por algoritmos como o Kruskal ou o Prim [1].

Existem várias variantes do MST com uma ou mais características adicionais. Geralmente, são problemas NP-completos, como o problema da árvore mínima sujeita a restrições de grau [2], o problema da árvore mínima com

erro limitado e custo mínimo [3], e o problema da árvore mínima sujeita a restrições de folhas [4]. Suas estruturas complexas são geralmente abordadas por meio de técnicas sofisticadas, como algoritmos de geração de colunas [5], programação por restrições [6] ou métodos de *branch-and-bound* [7]. Além disso, soluções aproximadas para esses problemas podem ser obtidas rapidamente por meio de heurísticas e meta-heurísticas [8].

O foco deste trabalho é o problema da árvore geradora mínima com restrição de grau (DMST, do inglês *Degree-constrained Minimum Spanning Tree*). Este problema é definido em um grafo ponderado $G = (V, E)$, como definido anteriormente além de um parâmetro adicional P que restringe o grau máximo dos vértices da árvore. O objetivo é encontrar uma árvore geradora de custo mínimo em G cujos vértices tenham grau menor ou igual a P. Este problema é NP-difícil quando $P \geq 2$ [9].

Propomos duas técnicas para solucionar o problema, uma baseada no algoritmo de recozimento simulado [10] e outra baseada no algoritmo de seleção clonal [11]. Ambas as abordagens utilizam a estrutura de dados *Node-depth Phylogenetic-based Encoding* (NPE) [12] para representar as soluções.

O restante deste trabalho está estruturado da seguinte forma: na seção 2, apresentamos uma breve revisão da literatura; na seção 3, apresentamos o NPE, juntamente com os algoritmos de recozimento simulado e seleção clonal, além do processo de otimização de parâmetros feito; na seção 4, apresentamos os resultados desse trabalho; por fim, na seção 5, apresentamos nossas conclusões.

2. Revisão de Literatura

Diversas heurísticas ad hoc para o DCMST foram propostas na literatura. No trabalho [13] foram propostas heurísticas primais e duais para esse problema, enquanto [14] propuseram duas heurísticas primais gerais. Adicionalmente, o estudo [15] apresentou uma heurística baseada no algoritmo de Prim.

Um grande número de algoritmos metaheurísticos foram propostos para o DCMST. O estudo inicial de [16] propôs um algoritmo genético (GA) baseado na codificação de números de Prüfer [17]. Posteriormente, os autores de [18] apresentaram um GA baseado em uma codificação de matriz multidimensional, enquanto [19] propuseram um algoritmo semelhante baseado na codificação de pesos. Duas variantes de GA com a codificação clássica de conjunto de arestas foram propostas por [20], sendo que uma delas implementou uma heurística dentro de seus operadores evolutivos. Finalmente, [21] propuseram um GA híbrido (HGA) que combina um GA de estado constante (steady state) e estratégias de busca local. Até onde sabemos, este algoritmo demonstrou ser a metaheurística mais eficiente da literatura para resolver o DCMST.

Apesar do volume de trabalhos da literatura, nenhum deles empregou o algoritmo de recozimento simulado para o problema. Além disso, nenhum aplica a estrutura de dados NPE para representar as soluções do problema. Portanto, neste trabalho apresentamos o uso da estrutura NPE, associada ao algoritmo de recozimento simulado e ao algoritmo de seleção clonal, para solucionar o DCMST, e contrastamos os dois métodos propostos ao algoritmo genético híbrido de [21].

3. Metodologia Utilizada

O método proposto utiliza uma estrutura de dados denominada *Node-depth Phylogenetic-based encoding* (NPE) [12] para representar as soluções, em particular, árvores geradoras. Esta estrutura é combinada com dois operadores de mutação: *Subtree Pruning and Regrafting Operator on NPE* (SPRN) e *Tree Bisection and Reconnection on NPE* (TBRN), os quais foram apresentados em conjunto com a estrutura de dados.

Para explorar o espaço de busca, dois algoritmos foram empregados separadamente: o algoritmo de recozimento simulado [10] e o algoritmo de seleção clonal [11]. Ambos os algoritmos utilizaram os operadores mencionados para gerar novas soluções.

3.1. Recozimento simulado

O recozimento simulado, uma metaheurística baseada no processo físico de resfriamento de metais [10], envolve perturbações no estado do metal em busca de um equilíbrio térmico. Ao aplicar essa ideia a problemas de otimização combinatória, a representação de uma solução substitui o metal, as perturbações referem-se a modificações no estado da solução, e a meta passa a ser a minimização da função objetivo associada ao problema.

Neste trabalho, a solução do problema é representada pelo NPE, utilizando seus operadores, para explorar o espaço de soluções. O NPE atua como a entidade que sofre as perturbações, refletindo o conceito do recozimento simulado na busca pela otimização do problema em questão.

O algoitmo 1 ilustra o pseudocódigo para o recozimento simulado.

Algoritmo 1 Pseudocódigo para o recozimento simulado

1: //Seja $x0$ a solução inicial
2: //Seja $temp$ a temperatura do sistema
3: //Seja cf o fator de resfriamento
4: //Seja f a função que avalia uma solução
5: $x\_atual \leftarrow x0$
6: $f\_atual \leftarrow f(x\_atual)$
7: $T \leftarrow temp$
8: $x\_melhor \leftarrow x\_atual$
9: $f\_melhor \leftarrow f\_atual$
10: **enquanto** $criterio\_de\_parada == FALSO$:
11: $x\_novo \leftarrow x\_atual.copia()$
12: // aplicar SPRN e/ou TBRN em x_novo
13: $f\_novo \leftarrow f(x\_novo)$
14: $p\_aceita \leftarrow exp(-(f\_novo - f\_atual)/T)$
15: **se** $aleatorio < p\_aceita$:
16: $x\_atual \leftarrow x\_novo$
17: $f\_atual \leftarrow f\_novo$
18: **fimse**
19: **se** $f\_atual < f\_melhor$:
20: $x\_melhor \leftarrow x\_atual$
21: $f\_melhor \leftarrow f\_atual$
22: **fimse**
23: $T \leftarrow T * cf$
24: **fimenquanto**
25: **retorna** (x_melhor, f_melhor)

O procedimento começa definindo a solução inicial, denotada por x_0, como a solução atual. Em seguida, na linha 6, a solução é avaliada. Como x_0 é a única solução até o momento, é considerada a melhor solução encontrada. Dentro da estrutura de repetição, a solução atual, denotada por x_atual, é copiada e modificada utilizando os operadores SPRN e TBRN. A nova solução é gerada através dessas alterações e, então, avaliada na linha 13. Na linha 14, é calculada a probabilidade de se aceitar a nova solução, mesmo que sua avaliação seja pior do que a de x_atual. Essa probabilidade visa escapar de mínimos locais. O procedimento continua verificando se a solução atual é a melhor encontrada até o momento e atualiza x_melhor, se necessário. Por fim, o procedimento retorna uma tupla contendo a melhor solução encontrada, juntamente com sua avaliação.

3.1.1. Função objetivo

A função objetivo utilizada no recozimento simulado permite soluções inviáveis, mas a penaliza. O método utilizado é uma modificação do 3, onde há a existência de uma lista graus, e a solução é penalizada sempre que um vértice tiver seu grau superior ao parâmetro restritivo.

3.2. Seleção clonal

O algoritmo de seleção clonal (CLONALG) [11] é inspirado no sistema imunológico, e se baseia na ideia de que cada ser vivo é exposto a um mesmo antígeno repetidas vezes, sempre formando anticorpos capazes de reagir contra ele. Simulando essa característica em um sistema computacional, é perceptível a capacidade de se gerar uma certa memória em relação a experiências passadas.

Associando essa ideia a conceitos de algoritmos evolucionários, a implementação do CLONALG conta com uma população de anticorpos que, a cada iteração, são clonados e modificados. Após estas alterações, os clones mais bem preparados para lidar com os antígenos são selecionados para fazerem parte da população.

Em problemas de otimização combinatória, a população de anticorpos do CLONALG representa possíveis soluções para o problema em questão. Os clones são modificados por meio de perturbações em sua estrutura, visando explorar o espaço de soluções. A seleção dos melhores clones mutados é realizada com base em uma função objetivo, que avalia a qualidade das soluções em relação a um problema.

O algoritmo 2 apresenta o pseudo-código do CLONALG. Ele recebe dois parâmetros como entrada: o tamanho do conjunto de soluções $tp \in \mathbb{N}_{>0}$; e a escala de clones $cs \in \mathbb{N}_{>0}$, que determina o número de cópias geradas a cada aplicação do operador clonal. A linha 3 cria a população inicial do algoritmo, que é avaliada na linha 4. O *loop* entre as linhas 5 e 10 é executado até que um critério de parada seja atendido. Cada iteração deste *loop* corresponde a uma geração do CLONALG. Primeiro, a linha 6 clona a população atual utilizando o operador de clonagem. Então, a linha 7 aplica o operador de hipermutação (em nosso caso, o SPRN ou o TBRN) e a linha 8 avalia o novo conjunto de soluções criado. Finalmente, a linha 9 seleciona um subconjunto de clones que serão persistidos para a próxima geração. O algoritmo para e retorna a população resultante, contendo a melhor solução encontrada, na linha 11.

Algoritmo 2 CLONALG

1: // Seja tp o tamanho da população
2: // Seja ec a escala de clones, i.e., quantos clones serão gerados
3: $P \leftarrow inicializa\_populacao(tp)$
4: $avalia(P)$
5: **enquanto** $parada == FALSE$:
6: $\quad P' \leftarrow clone(P, ec)$
7: $\quad hipermutacao(P')$
8: $\quad avalia(P')$
9: $\quad P \leftarrow seleciona(P', tp)$
10: **fimenquanto**
11: **retorna** P

O algoritmo proposto é semelhante a implementação de [22] para o problema da árvore geradora mínima para calibração com erro mínimo. Na abordagem proposta, a população de anticorpos é um conjunto de árvores geradoras codificadas como NPE. A busca em profundidade foi utilizada para gerar os NPEs da população inicial, com uma leve modificação, que consiste em manter uma lista com o grau dos vértices, visando não gerar soluções inviáveis para o problema.

O processo de clonagem gera uma matriz a partir da população, onde cada linha armazena os clones de um indivíduo. A mutação ocorre aplicando os operadores SPRN e TBRN nos clones da matriz. Por fim, os clones são avaliados e selecionados para a próxima iteração.

3.2.1. Função objetivo

A função objetivo utilizada no CLONALG não permite soluções inviáveis. Para isso, os operadores SPRN e TBRN foram modificados da seguinte forma: há a existência de uma lista de grau dos vértices, sujeitando a escolha do vértice a ao seu grau. Caso o grau de a seja igual ao parâmetro restritivo, outro vértice será escolhido. Cabe lembrar que essa modificação foi feita apenas para o algoritmo de seleção clonal, enquanto que os operadores SPRN e TBRN referentes ao recozimento simulado podem gerar soluções inviáveis.

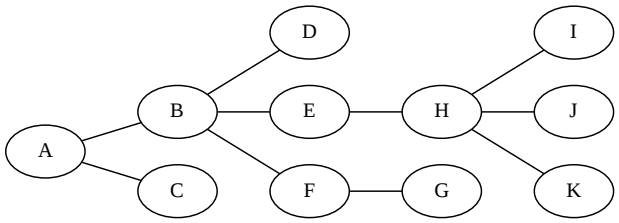
3.3. Node-depth Phylogenetic-based encoding

O *Node-depth Phylogenetic-based encoding* (NPE) é uma representação indireta de árvores geradoras, comumente utilizada em algoritmos evolucionários. Essa representação utiliza duas listas: uma para armazenar os vértices da árvore e outra para armazenar as profundidades de cada vértice. Seja N a lista de vértices e D a lista de profundidade. O vértice em N_i está na profundidade D_i. A Figura 3 mostra o exemplo de uma árvore geradora e um possível NPE correspondente.

Uma maneira simples e eficaz de se gerar o NPE de um grafo é utilizando um algoritmo de busca em profundidade [12]. Esta técnica foi empregada nesse trabalho para a geração de soluções iniciais. No exemplo da Figura 3, a busca se inicia no vértice A.

3.3.1. Processo de decodificação

Por se tratar de uma representação indireta, é necessário que haja um processo de decodificação para que seja possível encontrar quais são as arestas pertences a árvore geradora. Dito isso, é possível decodificar o NPE em um



(a)

N = [A B D F G E H I K J C]
D = [0 1 2 2 3 2 3 4 4 4 1]

(b)

Figura 1: Árvore geradora (a) e NPE correspondente (b)

conjunto de arestas, seguindo a simples ideia de que cada vértice possui uma aresta para o vértice anterior mais próximo que possua profundidade uma unidade inferior a sua. Em outras palavras, com x_i sendo a posição de x em N e y_i a posição de y, a aresta x, y existe para todo $y_i > x_i$ desde que $D_{yi} = D_{xi} + 1$.

O algoritmo 3 apresenta a decodificação de um NPE para um conjunto de arestas. O algoritmo começa verificando a existência do NPE ao verificar o tamanho da lista N. Em seguida, inicia a criação da estrutura $Arvore$, que será responsável por armazenar o conjunto de arestas. Atribui ao primeiro elemento da lista $raizes$ o vértice da primeira posição da lista N. A interpretação para a lista raízes é feita da seguinte forma: o vértice armazenado na posição x é o último vértice encontrado na profundidade x até o momento da execução. Ao percorrer a lista N e encontrar um vértice na profundidade d (com $d > 0$), saberemos que ele possui uma aresta conectando-o ao

vértice em $raizes_{d-1}$. O algoritmo, em seguida, percorre a lista N a partir da posição 1, usando a variável de contador j. Para cada vértice n_j encontrado, uma aresta $(raizes_{dj-1}, n_j)$ é adicionada à estrutura *Arvore*, e o vértice n_j é marcado como o último vértice encontrado na profundidade d_j (ou seja, $raizes_{d_j} \leftarrow n_j$). Por fim, *Arvore* será o conjunto de todas as arestas da árvore codificada como NPE.

Algoritmo 3 Decodificação do NPE

```
// Seja S um NPE
// Seja N a lista de vértices de S
// Seja |N| o número de vértices em N, i.e., o tamanho de N
se |N| == 0:
    retorna NULO
fimse
Arvore ← {}
raizes_0 ← N_0
j ← 1
enquanto j < |N|:
    k ← raizes_{d_j - 1}
    Arvore ← Arvore ∪ (k, n_j)
    raizes_{d_j} ← n_j
    j ← j + 1
fimenquanto
retorna Arvore
```

Posteriormente, um algoritmo baseado no processo de decodificação será apresentado para avaliar as soluções geradas.

3.3.2. Subtree Pruning and Regrafting Operator on NPE

O operador *Subtree Pruning and Regrafting Operator on NPE* (SPRN) [12] modifica a árvore geradora ao realocar uma de suas subárvores. Para tanto, se faz necessária a escolha de dois vértices, p e a, onde p é a raíz da

subárvore e a é o vértice para onde a subárvore será realocada.

O algoritmo 4 ilustra o pseudocódigo para o operador. A escolha do vértice p deve ser feita aleatoriamente, respeitando a regra de que p não é a raiz da árvore geradora, ou seja, não está na primeira posição da lista N. Uma vez definido o vértice p e seu índice em N, o algoritmo prossegue determinando a subárvore que possui p como raiz, denotando o intervalo que contém os vértices dessa subárvore como rp. Em seguida, o vértice a é selecionado, obedecendo à regra de que a não deve pertencer ao intervalo rp e a e p são adjacentes no grafo.

Para representar o NPE modificado, são criadas novas listas, N' e D'. Inicialmente, os vértices do intervalo $[0, ia]$ são adicionados a N', excluindo aqueles que estão contidos em rp. Da mesma forma, as profundidades dos vértices agora presentes em N' são copiadas para D'. A modificação ocorre sequencialmente, percorrendo os vértices de rp, adicionando-os a N' e recalculando suas profundidades em relação à profundidade do vértice a, que são então adicionadas a D'. Por fim, os vértices que não foram envolvidos na modificação, juntamente com suas profundidades, são copiados para N' e D', respectivamente.

Algoritmo 4 Operador SPRN

1: // Seja S um NPE
2: // Seja N a lista de vértices de S
3: // Seja $|N|$ o número de vértices em N, i.e., o tamanho de N
4: // Seja D a lista das profundidades de S
5: // Seja ip o indice do vértice p em N, com $0 < ip < |N|$
6: $fp \leftarrow ip + 1$
7: **enquanto** $(fp < |N|)$ && $(D_{fp} > D_{ip})$:
8: $\quad fp \leftarrow fp + 1$
9: **fimenquanto**
10: // Seja rp o intervalo $[ip, fp)$
11: // Seja ia o indice do vértice a em N, com $ia \notin rp$
12: $N' \leftarrow$ { todo vértice x, tal que $x_i \in [0, ia]$ e $x_i \notin rp$, onde x_i é o indice de x em N }
13: $D' \leftarrow$ { toda profundidade y, tal que $y_i \in [0, ia]$ e $y_i \notin rp$, onde y_i é o indice de y em D }
14: **para cada** $i \in rp$:
15: $\quad N' \leftarrow N' \cup N_i$
16: $\quad D' \leftarrow D' \cup D_i - D_{ip} + D_{ia} + 1$ // recalculando a profundidade do vértice em n_i
17: **fimpara**
18: $N' \leftarrow N' \cup$ { todo vértice x, tal que $x_i \in [ia + 1, |N|)$ e $x_i \notin rp$, onde x_i é o indice de x em N}
19: $D' \leftarrow D' \cup$ { toda profundidade y, tal que $y_i \in [ia + 1, |N|)$ e $y_i \notin rp$, onde y_i é o indice de y em D}
20: **retorna** (N', D')

A imagem 2a ilustra uma solução antes da aplicação do SPRN, enquanto a imagem 2b ilustra um exemplo da aplicação do SPRN na solução.

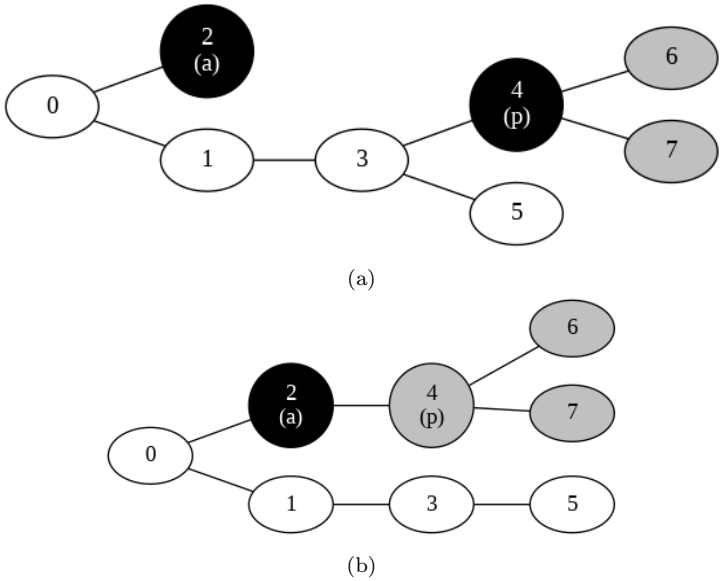


Figura 2: Solução pré-SPRN (a) e Solução pós-SPRN (b)

3.3.3. Tree Bisection and Reconnection on NPE

O operador *Tree Bisection and Reconnection on NPE* (TBRN) [12], ilustrado pelo pseudocódigo 5, modifica a árvore ao escolher uma subárvore qualquer, alterar sua raiz e, em seguida, realocá-la para outra parte da árvore. A diferença entre SPRN e TBRN está na alteração que ocorre na subárvore antes de sua realocação.

Para executar o operador TBRN, é necessário determinar três vértices: p, r e a. O vértice p corresponde à raiz da subárvore que será modificada. O vértice r é escolhido como um vértice contido na subárvore com raiz em p. Por fim, o vértice a é selecionado como um vértice adjacente a r que está localizado fora da subárvore onde p é a raiz.

Algoritmo 5 Operador TBRN

1: // Seja N a lista de vértices do NPE, com $|N|$ sendo sua cardinalidade
2: // Seja D a lista de profundidades do NPE
3: // Seja ip o indice do vértice p em N, com $0 < ip < |N|$
4: $fp \leftarrow ip + 1$
5: **enquanto** $(fp < |N|)$ && $(D_{fp} > D_{ip})$
6: $\quad fp \leftarrow fp + 1$
7: **fimenquanto**
8: $rp \leftarrow [ip, rp)$
9: // Seja ir o indice do vértice r em N, com $ir \in rp$
10: $fr \leftarrow ir + 1$
11: **enquanto** $(fr < |N|)$ && $(D_{fr} > D_{ir})$
12: $\quad fr \leftarrow fr + 1$
13: **fimenquanto**
14: $rr \leftarrow [ir, rr)$
15: $Ntmp \leftarrow \{N_i \mid \forall i \in rr\}$
16: $Dtmp \leftarrow \{D_i - D_r + D_a + 1 \mid \forall i \in rr\}$
17: // Seja ia o indice do vértice a em N, tal que a é adjacente de r e $ia \notin rp$
18: $N' \leftarrow$ { todo vértice x, tal que $x_i \in [0, ia]$ e $x_i \notin rp$, onde x_i é o indice de x em N }
19: $D' \leftarrow$ { toda profundidade y, tal que $y_i \in [0, ia]$ e $y_i \notin rp$, onde y_i é o indice de y em D }
20: $ix \leftarrow ir$
21: **enquanto** $ix \neq ip$:
22: $\quad$ // Seja i o indice do vértice imediatamente anterior a x que possui profundidade igual a $D_{ix} - 1$
23: $\quad fx \leftarrow i + 1$
24: $\quad$ **enquanto** $(fx < |N|)$ && $(D_{fx} > D_i)$
25: $\quad\quad fx \leftarrow fx + 1$
26: $\quad$ **fimenquanto**
27: $\quad$ // Seja aux_{ix} o indice de x em $Ntmp$
28: $\quad Ntmp \leftarrow Ntmp \cup \{N_j \mid \forall j \in (i, fx]\}$
29: $\quad Dtmp \leftarrow Dtmp \cup \{D_j - D_i + Dtmp_{aux\_ix} + 1 \mid \forall j \in (i, fx]\}$
30: $\quad ix \leftarrow i$
31: **fimenquanto**
32: $N' \leftarrow N' \cup Ntmp \cup \{N_i \mid \forall i \in (ia + 1, |N|)\}$
33: $D' \leftarrow D' \cup Dtmp \cup \{D_i \mid \forall i \in (ia + 1, |N|)\}$
34: **retorna** (N', D')

A modificação realizada pelo operador TBRN envolve a redefinição do vértice r como a nova raiz da subárvore e, em seguida, a realocação dessa subárvore através da adição da aresta (a, r) à árvore.

O vértice p é selecionado aleatoriamente, contanto que não esteja na posição inicial da lista N. O vértice r também é selecionado aleatoriamente, desde que pertença à subárvore com p como raiz. Da mesma forma, o vértice a é escolhido aleatoriamente, com as seguintes restrições: (1) não está na subárvore com p como raiz e (2) é adjacente ao vértice r.

A imagem 3a ilustra uma solução antes da aplicação do TBRN, enquanto a imagem 3b ilustra um exemplo da aplicação do TBRN na solução.

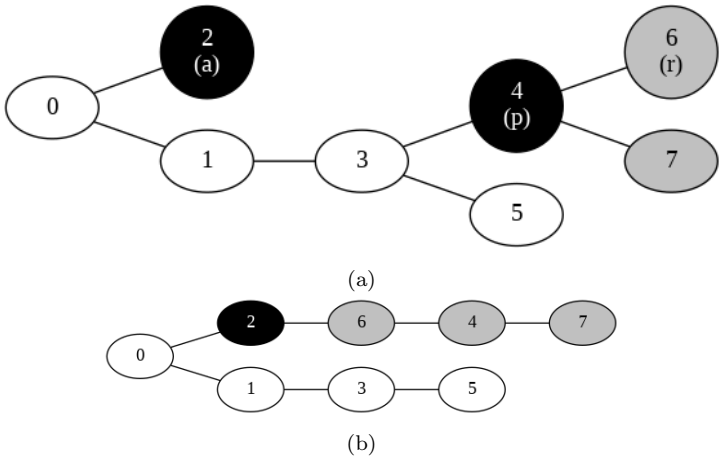


Figura 3: Solução pré-TBRN (a) e Solução pós-TBRN (b)

4. Análise dos Resultados

Os experimentos foram executados em um computador com 8GB de RAM, em um único núcleo de um processador Intel Core i5-7200U de 2.5GHz, utilizando a linguagem de programação Python 3 no Windows 10. Os algorit-

mos propostos foram comparado com um algoritmo genético híbrido (HGA, do inglês *Hybrid Genetic Algorithm*) proposto por [21] para este mesmo problema.

4.1. Instâncias

Duas classes de instâncias foram utilizadas, ambas baseadas nas instâncias utilizadas por [21]. Para cada conjunto de instâncias e vértices, geramos 10 diferentes instâncias aleatórias.

- **CRD:** instâncias nas quais os grafos possuem de 50 a 100 vértices distribuídos de maneira uniforme em um plano euclidiano de duas dimensões, sendo que o peso das arestas é a distância euclidiana entre as coordenadas dos vértices.
- **SYM:** instâncias nas quais os grafos possuem de 50 a 70 vértices. Os vértices são distribuídos em planos euclidianos de 3 a 5 dimensões e, de maneira análoga ao CRD, a distância euclidiana entre as coordenadas dos vértices é igual ao peso das arestas.

4.2. Otimização de parâmetros

Alguns dos parâmetros referentes aos algoritmos propostos passaram por um processo de otimização. Para tal, foi utilizada a biblioteca Optuna [23], escrita na linguagem Python 3. No processo, o algoritmo TPE (Tree-structured Parzen Estimator) foi utilizado. Cinco instâncias de cada grupo de instância foram geradas, totalizando dez, e a avaliação de cada conjunto de parâmetro foi feita a partir do valor médio encontrado; cem combinações foram testadas para cada algoritmo, e a combinação com melhor média foi

selecionada para os próximos experimentos. As tabelas 1 e 2 ilustram quais os parâmetros que foram otimizados, juntamente com o valor final da otimização.

Temperatura	Fator de resfriamento	Probabilidade de aplicação do TBRN
13.3634	0.9860	0.1005

Tabela 1: Otimização de parâmetros do recozimento simulado

População	Número de clones	Probabilidade de aplicação do TBRN
16	80	0.2956

Tabela 2: Otimização de parâmetros da seleção clonal

Neste trabalho, os operadores TBRN e SPRN são aplicados de maneira probabilística e exclusiva, o que explica a necessidade de se apresentar um parâmetro referente a probabilidade de aplicação do TBRN, uma vez que a probabilidade de aplicação do SPRN será o complemento.

4.3. Resultados

As tabelas a seguir mostram os resultados obtidos para os seguintes graus de restrição: 2, 3, 4 e 5. A primeira coluna de cada tabela é referente ao número de vértices, e as demais colunas mostram o valor médio obtido pela execução dos algoritmos. O grau de restrição e o conjunto de instâncias estão na descrição de cada tabela.

Número de vértices	Recozimento	Clonal	GA
50	2062.726	3681.139	3021.948
55	2150.507	3937.469	3277.603
60	2368.136	4379.869	3683.066
65	2641.387	4720.012	4034.560
70	2775.588	5167.172	4414.228

Tabela 3: Conjunto SYM; Grau de restrição = 2

Número de vértices	Recozimento	Clonal	GA
50	1471.945	3577.538	2262.14
55	1557.505	3763.246	2450.074
60	1661.131	4166.083	2757.816
65	1728.312	4531.196	3031.430
70	1842.315	4968.669	3336.920

Tabela 4: Conjunto SYM; Grau de restrição = 3

Número de vértices	Recozimento	Clonal	GA
50	1460.674	3681.139	3021.948
55	1537.277	3937.469	3277.603
60	1645.840	4379.869	3683.066
65	1711.141	4720.012	4034.560
70	1818.842	5167.172	4414.228

Tabela 5: Conjunto SYM; Grau de restrição = 4

Número de vértices	Recozimento	Clonal	GA
50	1459.621	3539.612	2088.624
55	1538.432	3676.52	2246.217
60	1651.010	4092.290	2561.095
65	1718.313	4362.485	2766.413
70	1827.450	4757.299	3113.436

Tabela 6: Conjunto SYM; Grau de restrição = 5

Número de vértices	Recozimento	Clonal	GA
50	825.765	2534.315	1620.961
60	1003.690	2934.606	2151.420
70	1194.473	3614.403	2647.3
80	1398.120	3998.892	3160.283
90	1661.934	4530.156	3605.949
100	1893.011	5012.312	4132.985

Tabela 7: Conjunto CRD; Grau de restrição = 2

Número de vértices	Recozimento	Clonal	GA
50	503.018	2327.099	950.392
60	559.342	2743.681	1256.782
70	581.359	3315.115	1590.564
80	634.312	3795.926	1888.652
90	684.578	4296.189	2275.759
100	768.865	4814.603	2683.053

Tabela 8: Conjunto CRD; Grau de restrição = 3

Número de vértices	Recozimento	Clonal	GA
50	500.338	2226.792	958.579
60	549.196	2728.384	1344.069
70	589.436	3224.182	1670.362
80	640.087	3702.376	2083.267
90	690.605	4139.379	2547.673
100	751.455	4656.827	2926.824

Tabela 9: Conjunto CRD; Grau de restrição = 4

Número de vértices	Recozimento	Clonal	GA
50	499	2196.233	989.497
60	544.943	2704.532	1223.689
70	583.149	3125.737	1524.840
80	636.447	3635.776	1878.766
90	676.986	4117.952	2250.276
100	739.149	4638.639	2664.626

Tabela 10: Conjunto CRD; Grau de restrição = 5

A possibilidade de se gerar soluções inviáveis no recozimento simulado se mostrou fundamental para a melhoria dos resultados, o que pode ser explicado da seguinte forma: se uma solução ótima s* compartilha uma subestrutura com uma solução inviável s, o algoritmo de recozimento simulado s e, então, modificá-la, com uma probabilidade de se chegar a s*, ou algo próximo (uma solução viável que também compartilha uma subestrutura com s*); impondo ao recozimento simulado as mesmas modificações impostas ao clonal, os resultados de ambos foram muito parecidos. Por se tratar de uma meta-heurística populacional, o algoritmo de seleção clonal, sem as citadas mudanças, teve suas populações com diversas soluções inviáveis.

Com base nas tabelas 3 à 10, pode-se observar que o recozimento simulado encontrou soluções melhores que os outros algoritmos em ambos os conjuntos de instâncias. Embora o algoritmo de seleção clonal utilize os mesmos operadores que o recozimento, seu resultado foi o pior entre os algoritmos, o que pode ser explicado pela não possibilidade de se gerar soluções inviáveis, levando a possível prisão em mínimos locais.

5. Conclusão

Neste trabalho, estudamos o uso da estrura de dados NPE, juntamente com o SPRN e TBRN, para representar e modificar árvores geradoras de um grafo. Além disso, a empregamos nas meta-heurísticas, recozimento simulado e seleção clonal, para solucionar o problema da árvore geradora mínima com restrição de grau. Dentre os algoritmos propostos para o problema, o recozimento simulado encontrou melhores soluções, superando, inclusive, o algoritmo genético com estratégias de busca local, enquanto o algoritmo de seleção clonal não apresentou resultados significativos.

Trabalhos futuros podem empregar os métodos em novos problemas relacionados a árvores geradoras, além de adaptar o algoritmo de seleção clonal para trabalhar com soluções inviáveis. Além disso, empregar o NPE em outras meta-heurísticas, como *Iterated local search* [24], pode proporcionar uma investigação mais ampla acerca da estrutura de dados.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Minimum spanning trees, in: Introduction to Algorithms, MIT Press Cambridge, 2009, pp. 624–632.

[2] A. M. de Almeida, P. Martins, M. C. de Souza, Min-degree constrained minimum spanning tree problem: complexity, properties, and formulations, International Transactions in Operational Research 19 (3) (2012) 323–352.

[3] H. Akcan, On the complexity of energy efficient pairwise calibration in embedded sensors, Applied Soft Computing 13 (4) (2013) 1766–1773.

[4] N. Deo, P. Micikevicius, A heuristic for a leaf constrained minimum spanning tree problem, Congressus Numerantium (1999) 61–72.

[5] L. Gouveia, A. Paias, D. Sharma, Modeling and solving the rooted distance-constrained minimum spanning tree problem, Computers & Operations Research 35 (2) (2008) 600–613.

[6] T. F. Noronha, C. C. Ribeiro, A. C. Santos, Solving diameter-constrained minimum spanning tree problems by constraint programming, International Transactions in Operational Research 17 (5) (2010) 653–665.

[7] I. A. Carvalho, M. A. Ribeiro, An exact approach for the minimum-cost bounded-error calibration tree problem, Annals of Operations Research 287 (1) (2020) 109–126.

[8] A. E. Ezugwu, A. K. Shukla, R. Nath, A. A. Akinyelu, J. O. Agushaka, H. Chiroma, P. K. Muhuri, Metaheuristics: a comprehensive overview and classification along with bibliometric analysis, Artificial Intelligence Review 54 (2021) 4237–4316.

[9] M. R. Garey, D. S. Johnson, Computers and intractability, Vol. 174, freeman San Francisco, 1979.

[10] D. Henderson, S. H. Jacobson, A. W. Johnson, The theory and practice of simulated annealing, Handbook of metaheuristics (2003) 287–319.

[11] L. N. De Castro, F. J. Von Zuben, The clonal selection algorithm with engineering applications, in: Proceedings of GECCO, Vol. 2000, 2000, pp. 36–39.

[12] T. W. de Lima, A. C. B. Delbem, A. da Silva Soares, F. M. Federson, J. B. A. L. Junior, J. Van Baalen, Node-depth phylogenetic-based encoding, a spanning-tree representation for evolutionary algorithms. part i: Proposal and properties analysis, Swarm and Evolutionary Computation 31 (2016) 1–10.

[13] S. C. Narula, C. A. Ho, Degree-constrained minimum spanning tree, Computers & Operations Research 7 (4) (1980) 239–249.

[14] M. Savelsbergh, T. Volgenant, Edge exchanges in the degree-constrained minimum spanning tree problem, Computers & Operations Research 12 (4) (1985) 341–348.

[15] B. Boldon, N. Deo, N. Kumar, Minimum-weight degree-constrained spanning tree problem: Heuristics and implementation on an simd parallel machine, Parallel Computing 22 (3) (1996) 369–382.

[16] G. Zhou, M. Gen, A note on genetic algorithms for degree-constrained spanning tree problems, Networks: an International Journal 30 (2) (1997) 91–95.

[17] F. Rothlauf, D. Goldberg, Tree network design with genetic algorithms–an investigation in the locality of the pruefernumber encoding, in: Late breaking papers at the genetic and evolutionary computation conference, 1999, pp. 238–244.

[18] J. Knowles, D. Corne, A new evolutionary approach to the degree-constrained minimum spanning tree problem, IEEE Transactions on Evolutionary computation 4 (2) (2000) 125–134.

[19] G. R. Raidl, B. A. Julstrom, A weighted coding in a genetic algorithm for the degree-constrained minimum spanning tree problem, in: Proceedings of the 2000 ACM symposium on Applied computing-Volume 1, 2000, pp. 440–445.

[20] G. R. Raidl, B. A. Julstrom, Edge sets: an effective evolutionary coding of spanning trees, IEEE Transactions on evolutionary computation 7 (3) (2003) 225–239.

[21] K. Singh, S. Sundar, A hybrid genetic algorithm for the degree-constrained minimum spanning tree problem, Soft Computing 24 (3) (2020) 2169–2186.

[22] I. A. Carvalho, M. A. Ribeiro, A node-depth phylogenetic-based artificial immune system for multi-objective network design problems, Swarm and Evolutionary Computation 50 (2019) 100491. doi:10.1016/j.swevo.2019.01.007.

[23] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, 2019, pp. 2623–2631.

[24] H. R. Lourenço, O. C. Martin, T. Stützle, Iterated local search, in: Handbook of metaheuristics, Springer, 2003, pp. 320–353.