LIII Simpósio Brasileiro de Pesquisa Operacional
João Pessoa - PB, 3 a 5 de novembro de 2021

COMPUTANDO A TREEWIDTH DE UM GRAFO: ALGORITMO GENÉTICO TENDENCIOSO DE CHAVES ALEATÓRIAS

Samuel Eduardo da Silva, Celso C. Ribeiro, Uéverton S. Souza
Universidade Federal Fluminense, Instituto de Computação
Niterói – RJ, Brasil
samueleduardo@id.uff.br,
{celso,ueverton}@ic.uff.br

RESUMO

Seja $G = (V, E)$ um grafo e $\mathcal{C}_\mathcal{G}$ a família dos grafos cordais G' tal que $V(G) = V(G')$ e $E(G) \subseteq E(G')$. A *treewidth* de um grafo G é o tamanho da menor clique de tamanho máximo de um grafo em $\mathcal{C}_\mathcal{G}$, menos um. A noção de *treewidth* visa mensurar o quão próximo de uma árvore um dado grafo G se encontra. Para qualquer permutação ρ dos vértices de um grafo G, determinar o tamanho da maior clique do supergrafo cordal (triangulação) de G que possui ρ como esquema de eliminação perfeita é uma tarefa que pode ser executada em tempo polinomial. Sendo assim, uma estratégia para computar a *treewidth* de um grafo G é encontrar o esquema de eliminação perfeita do grafo cordal $G' \in \mathcal{C}_G$ que possui a menor clique máxima. Neste artigo, baseado na estratégia de manipular permutações representando esquemas de eliminação perfeita de triangulações, é desenvolvido um algoritmo genético tendencioso de chaves aleatórias (BRKGA) para computar a *treewidth* de um grafo, onde utiliza-se heurísticas clássicas acopladas à técnicas de algoritmos genéticos. A solução proposta é competitiva com os métodos heurísticos da literatura.

PALAVRAS CHAVE. *treewidth*, decomposição em árvores, triangulações, grafos cordais, BRKGA.
Tópicos (MH – Metaheurísticas, TAG - Teoria e Algoritmos em Grafos)

ABSTRACT

Let $G = (V, E)$ be a graph and $\mathcal{C}_\mathcal{G}$ be the family of chordal graphs G' such that $V(G) = V(G')$ and $E(G) \subseteq E(G')$. The *treewidth* of a graph G is the size of the smallest maximum clique of a graph in $\mathcal{C}_\mathcal{G}$, minus one. The notion of treewidth aims to measure how close to a tree a given graph G is. For any permutation ρ of the vertices of a graph G, determining the size of the largest clique of the chordal supergraph (triangulation) of G that has ρ as a perfect elimination ordering is a task that can be performed in polynomial time. Therefore, a strategy to compute the treewidth of a graph G is to find the perfect elimination ordering of the chordal graph $G' \in \mathcal{C}_G$ that has the smallest maximum clique. In this article, based on the strategy of manipulating permutations representing perfect elimination ordering of triangulations, we developed a biased random-key genetic algorithm (BRKGA) to compute the treewidth of a graph, where we use classical heuristics coupled with genetic algorithm techniques. The proposed solution is competitive with the heuristic methods of the literature.

KEYWORDS. treewidth, tree decomposition, triangulation, chordal graphs, biased random-key genetic algorithms.
Paper topics (MH – Metaheuristics, TAG - Theory and Algorithms in Graphs)

1. Introdução

A *treewidth* de um grafo, também conhecida como *largura arbórea*, é um parâmetro importante em teoria dos grafos que visa mensurar o quão próximo de uma árvore um dado grafo G se encontra. Diversas classes de grafos possuem *treewidth* limitada por uma constante e muitos problemas *NP-difíceis* podem ser solucionados em tempo polinomial em classes de grafos que possuem *treewidth* constante (veja Cygan et al. [2015]; Gaspers et al. [2019]).

Um conceito importante para entender a noção de *treewidth* é o de *k-árvore*. Uma k-árvore é definida recursivamente da seguinte forma: um grafo completo[1] com $k+1$ vértices é uma *k-árvore*; uma k-árvore T com $n+1$ vértices onde $n \geq k+1$ pode ser construída a partir de uma k-árvore H com n vértices adicionando-se um novo vértice adjacente a exatamente k vértices que formam uma k-clique em H.

Um grafo G é uma *k-árvore parcial* se G é subgrafo de uma *k-árvore*. Por fim, dentre as diversas definições e caracterizações alternativas de *treewidth*, pode-se definir a *treewidth* de um grafo G como o menor k para o qual G é uma k-árvore parcial. Um grafo é dito cordal se não possui um ciclo de k vértices, para $k \geq 4$, como subgrafo induzido. É bem conhecido na literatura que as k-árvores são uma subclasse dos grafos cordais. Sendo assim, dado um grafo $G = (V, E)$ e denotando por $\mathcal{C}_\mathcal{G}$ a família dos grafos cordais G' tal que $V(G) = V(G')$ e $E(G) \subseteq E(G')$, a *treewidth* de um grafo G pode ser definida alternativamente como a cardinalidade da menor clique de tamanho máximo de um grafo em $\mathcal{C}_\mathcal{G}$, menos um.

O fato dos grafos com *treewidth* no máximo k serem exatamente a classe dos subgrafos de uma k-árvore, implica em ser possível decompor os grafos com *treewidth* no máximo k através de um estrutura em formato de árvore denominada *árvore de decomposição*. Árvores de decomposição são amplamente utilizadas na confecção de algoritmos de programação dinâmica para problemas em grafos com *treewidth* constante (veja Cygan et al. [2015]). A seguir, apresenta-se a definição de árvores de decomposição e sua relação com a *treewidth*.

Uma *árvore de decomposição* de um grafo $G = (V, E)$ é uma árvore T, com nós $1, \ldots, \ell$, onde cada nó t possui uma bolsa X_t que é um subconjunto de V, satisfazendo as seguintes propriedades (Cygan et al. [2015]):

1. a união de todos os conjuntos (bolsas) X_i é igual a V (cada vértice está contido em pelo menos uma bolsa, podendo estar contido em mais de uma);
2. para cada aresta (v, w) do grafo, existe uma bolsa X_t que contém ambos v e w;
3. se ambos conjuntos X_i e X_j contém um vértice v, então todos os nós no caminho entre i e j na árvore T contém v em suas respectivas bolsas.

A *largura* de uma árvore de decomposição T é o tamanho de sua maior bolsa X_t menos um. A *treewidth*, $tw(G)$, de um grafo G pode ser definida alternativamente como a largura mínima de todas as possíveis árvores de decomposição de G. Notem que o decremento por um faz com que árvores e florestas não triviais tenham *treewidth* igual a um.

Para verificar que as diferentes definições de *treewidth* são equivalentes, basta observar que a partir de uma árvore de decomposição com largura k, pode-se obter um grafo cordal (e posteriormente uma k-árvore) transformando cada bolsa em uma clique (que terá tamanho no máximo $k+1$)[2]. Além disso, em um grafo cordal cujo tamanho da maior clique é $k+1$, pode-se remover

[1] Grafo completo é um grafo que contém aresta entre cada par de vértices.

[2] A partir de um grafo cordal G com clique máxima de tamanho $k+1$ obter um supergrafo de G que é uma k-árvore é uma tarefa simples.

sucessivamente vértices (simpliciais[3]) cuja vizinhança é uma clique de tamanho no máximo k. A partir desse esquema de eliminação perfeita pode-se construir uma decomposição em árvore onde cada bolsa é formada pela vinhança fechada de um vértice simplicial no momento de sua remoção/eliminação, maiores detalhes serão apresentados na Seção 2.

Decidir se um grafo tem *treewidth* de tamanho no máximo *k* é *NP-difícil*, como demonstrado em Robertson e Seymour [1986]; Arnborg et al. [1987]. Sendo assim, têm-se a motivação de encontrar soluções (i.e., supergrafos cordais ou árvores de decomposição) que, talvez não sejam ótimas (i.e., menor clique máxima ou largura), mas que sejam de boa qualidade.

Para qualquer permutação ρ dos vértices de um grafo G, determinar o tamanho da maior clique do supergrafo cordal minimal de G que possui ρ como esquema de eliminação perfeita é uma tarefa que pode ser executada em tempo polinomial, como pode-se verificar em detalhes mais adiante. Sendo assim, uma estratégia para computar a *treewidth* de um grafo G consiste em buscar o esquema de eliminação perfeita do grafo cordal $G' \in \mathcal{C}_G$ que possui a menor clique máxima. As principais heurísticas para resolução do problema são baseadas em tentar determinar essa permutação. Para isso, ao criar uma permutação, os vértices são escolhidos em ordem de acordo com algum critério. Segundo Gaspers et al. [2019], as mais comumente utilizadas são: *min-degree* e *fill-in*. A primeira escolhe sucessivamente vértices que têm o menor grau e a segunda escolhe vértices que tem sua vizinhança mais próxima de uma clique, ambos critérios podem ser implementados em tempo polinomial.

Com essas heurísticas pode-se criar facilmente permutações que codificam soluções viáveis para o problema. Este trabalho busca ir além dessas soluções. Para isso, foram então utilizados algoritmos genéticos, que são métodos baseados na analogia com o processo de seleção natural e evolução. Tais algoritmos pertencem à classe dos métodos populacionais, que consideram uma população de soluções e não apenas uma única solução. As permutações dos vértices são então utilizadas como indivíduos dessa população de soluções, que são melhoradas por meio dos processos de mutação e cruzamento.

A partir dos algoritmos genéticos, surge o algoritmo genético tendencioso de chaves aleatórias (do inglês, biased random-key genetic algorithm, ou BRKGA simplificando). Este trabalho apresenta um método para computar soluções aproximadas para o problema TREEWIDTH com base na heurística *fill-in*, onde obtém-se uma permutação inicial a ser injetada em uma população aleatória de soluções a partir do qual desenvolve-se o BRKGA. Portanto, o principal objetivo deste trabalho é discutir heurísticas para o problema, bem como propor uma meta-heurística para o mesmo.

A Seção 2 deste trabalho apresenta algumas definições auxiliares. A Seção 3 apresenta o algoritmo genético tendencioso de chaves aleatórios proposto. Na Seção 4, tem-se os experimentos realizados com a solução proposta. Por fim, na Seção 5, são apresentadas as considerações finais.

2. Preliminares

Para um entendimento do cálculo da *treewidth* algumas definições e teoremas são necessários. O primeiro é o de grafo cordal:

Definição 1 *Um grafo simples G é um grafo cordal, se todos seus ciclos induzidos tem exatamente três vértices. Analogamente, todos os ciclos de um grafo simples G com quatro ou mais vértices têm uma corda, que é uma aresta que não é parte do ciclo, mas conecta dois vértices do ciclo.*

[3]Vértices simpliciais são vértices cuja vizinhança induz uma clique.

No contexto deste trabalho, grafos cordais são importantes pois, como observado por Patil [1986], uma de suas subclasses são as k-árvores, que foram definidas anteriormente. Grafos cordais podem ser caracterizados a partir de esquemas de eliminação perfeita e vértices simpliciais.

Definição 2 *Um vértice v de um grafo G é* simplicial *se sua vizinhança induz uma clique. Uma ordem $\rho = v_1, v_2, \ldots, v_n$ dos vértices de um grafo G é um* esquema (ou ordem) de eliminação perfeita *de G se para todo $1 \leq i \leq n$, v_i é um vértice simplicial em $G[X]$ onde $X = \{v_i, \ldots, v_n\}$.*

Teorema 3 *Dirac [1961] Um grafo cordal ou é um grafo completo ou possui pelo menos dois vértices simpliciais não adjacentes.*

Fulkerson e Gross [1965] apresentaram um algoritmo para reconhecer grafos cordais, que se baseia no Teorema 3. O algoritmo tem os seguintes passos: (1) procure um vértice simplicial e remova-o do grafo; (2) repita o próximo passo até não haver mais vértices simpliciais; (3) se não houver vértices simpliciais, e o grafo não for estiver vazio, pelo Teorema 3, pode-se concluir que o grafo não é cordal. Se estiver vazio, então é cordal.

Se o grafo for cordal, sempre haverão vértices simpliciais a serem removidos até o grafo se tornar vazio, consequentemente, pela Definição 2, tem-se um esquema de eliminação perfeita. Portanto, é possível sempre encontrar uma ordem de eliminação perfeita para grafos cordais. Logo, tem-se os seguintes teoremas:

Teorema 4 *(Fulkerson e Gross [1965]) Um grafo é cordal se e somente se ele tem um esquema de eliminação perfeita.*

Teorema 5 *(Dirac [1961]) Dado um esquema de eliminação perfeita $f = v_1, \ldots, v_n$ de um grafo cordal G, a* treewidth *de G é dado por $\max(|N(v_i) \cap \{v_i, \ldots, v_n\}|)$ onde $N(v_i)$ indica os vizinhos de v_i em G.*

Com os dois teoremas acima, consegue-se garantir que sempre é possível obter um esquema de eliminação de um grafo cordal e consequentemente o valor da *treewidth* deste grafo. Além disso, o algoritmo descrito fornece um esquema de eliminação perfeita. Também, é possível transformar qualquer grafo G em cordal através da adição de arestas e o grafo resultante é chamado de uma *triangulação* de G. O processo para isto é descrito nos seguintes passos: (1) crie uma cópia do grafo original, denominada C; (2) escolha qualquer vértice x de C, adicione arestas entre os vértices da vizinhança de x até estes formarem uma clique; (3) remova x de C; (4) adicione ao grafo original todas as arestas adicionadas no passo anterior; (5) repita o passo 2, se ainda houver vértices em C. Ao final do processo, o grafo resultante é dito uma triangulação de G, e por construção, um grafo cordal. A ordem em que os vértices foram sendo removidos da cópia do grafo original fornecem um esquema de eliminação perfeita. Logo, para calcular sua *treewidth*, basta aplicar o Teorema 5, retornando o valor da maior vizinhança encontrada durante o processo. No entanto, encontrar o esquema de eliminação perfeita que resulte na triangulação com menor *treewidth* é NP-difícil, uma vez que equivale a computar a *treewidth* do grafo original, veja também Yannakakis [1981].

Com esse embasamento teórico, pode-se definir o problema de encontrar a *treewidth*:

TREEWIDTH
Instância: Um grafo $G = (V, E)$.
Objetivo: Determinar o menor k para o qual G é uma k-árvore parcial.

Alternativamente, pode-se reformular o problema da seguinte forma:

TREEWIDTH
Instância: Um grafo $G = (V, E)$.
Objetivo: Determinar uma permutação dos vértices de G representando um esquema de eliminação perfeita da triangulação de G com menor clique máxima (*treewidth*).

Retornando à noção de decomposição em árvore, a seguir é apresentado um teorema importante.

Teorema 6 *(Bodlaender e Möhring [1993]) Seja T uma decomposição em árvore de $G = (V, E)$, e $K \subseteq V$ uma clique de G. Então existe um nó t de T tal que $K \subseteq X_t$.*

Dado um esquema de eliminação perfeita $\rho = 1, 2, \ldots, \ell$ de um grafo G, com base no Teorema 6, consegue-se construir uma decomposição em árvore T de G onde cada bolsa da decomposição é formada pela vinhança fechada de um vértice simplicial no momento de sua remoção/eliminação. Dessa forma, o pai de um nó i em T será o menor j em X_i, ou seja, o próximo vizinho de i a ser removido de acordo com ρ. Nesse caso, o nó ℓ será a raiz de T.

Esta decomposição em árvore gerada pelo esquema de eliminação perfeita pode ser convertida no que chama-se *decomposição em árvore legal (nice)*. O conceito de decomposição em árvore legal é introduzido por Kloks [1994] e surge para limitar a estrutura de uma decomposição em árvore em um conjunto pequeno de possíveis transições entre nós/bolsas. A decomposição em árvore ótima mantém o valor de largura da decomposição original, mas permite apenas quatro tipos de transições entre nós. As quatro transições são: nós que introduzem um vértice - nó com apenas um filho, onde sua bolsa contém apenas um vértice a mais que a bolsa do filho; nós que esquecem um vértice - nó com apenas um filho, onde sua bolsa contém apenas um vértice a menos que a bolsa do filho; nós de junção - nós com dois filhos tendo bolsas com o mesmo conteúdo que o pai; e nós folha. A motivação desse tipo de estrutura é que com ela consegue-se desenvolver algoritmos de programação dinâmica para problemas em grafos, veja, por exemplo Duarte et al. [2021]).

Para maiores detalhes sobre decomposições em árvores, *treewidth*, e conceitos envolvendo parâmetros de largura e complexidade parametrizada, convida-se o leitor a consultar Cygan et al. [2015]. Sobre demais notações e conceitos da Teoria dos Grafos, veja Diestel et al. [2010]. A próxima seção apresenta o algoritmo genético tendencioso de chaves aleatórias proposto.

3. Algoritmo genético tendencioso de chaves aleatórias

Algoritmos genéticos com chaves aleatórias foram inicialmente introduzidos por Bean [1994] para problemas de otimização em que suas soluções podem ser representadas por vetores de permutações. Soluções são representadas como vetores de números reais no intervalo (0,1], gerados randomicamente e associados a alguma estrutura da entrada do problema. No contexto deste artigo, sempre representam uma permutação dos vértices do grafo e cada permutação codificada é chamada de **indivíduo** e cada chave é chamada de **gene**. Um algoritmo determinístico, chamado decodificador, recebe como entrada um vetor e o associa com uma solução. Dois indivíduos pais são selecionados randomicamente de uma população e é feita então a operação de cruzamento. Este algoritmo é denominado RKGA (do inglês, *Random-key genetic algorithm*).

Gonçalves e Resende [2011] introduziram o algoritmo genético tendencioso de chaves aleatórias, ou BRKGA (do inglês, *Biased random-key genetic algorithm*), que difere do RKGA na forma com que os pais são selecionados na operação de cruzamento. No BRKGA, cada elemento é gerado combinando um elemento selecionado randomicamente do conjunto de soluções mais aptas

da população corrente, enquanto o outro é selecionado do restante da população. A seleção é dita tendenciosa pois um pai é sempre selecionado dos melhores indivíduos e tem probabilidade maior de passar seus genes para a nova geração.

Cada indivíduo da população inicial é gerado de forma aleatória, ou seja, os indivíduos correspondem a permutações aleatórias do vetor de vértices do grafo. Os indivíduos associados às melhores soluções formam o conjunto *Elite*, enquanto os demais forman o conjunto *Resto*. O tamanho da população é dado por $\mid Elite \mid + \mid Resto \mid$. A Figura 1 de Gonçalves e Resende [2011] ilustra como a população da geração seguinte do BRKGA é formada:

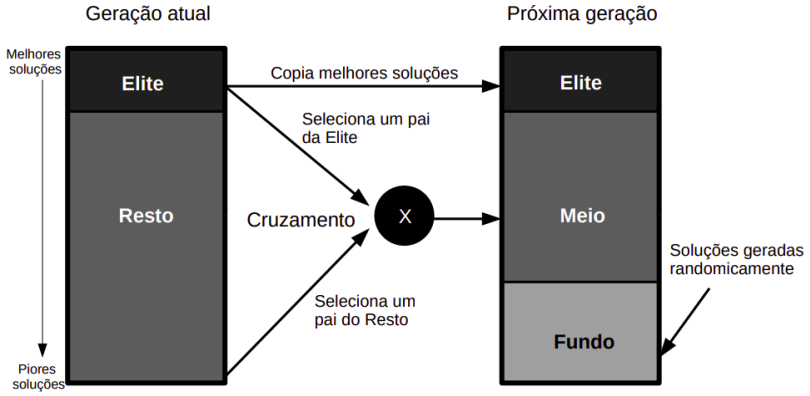


Figura 1: Evolução da população entre gerações no BRKGA.

O procedimento de cruzamento é feito como proposto por Spears e De Jong [1995]: para cada gene do filho, há uma probabilidade maior desse gene ser herdado de um gene do pai pertencente ao conjunto *Elite*.

Como ilustrado na Figura 1, os indivíduos do conjunto *Elite* são copiados para a população da próxima geração. Os elementos do conjunto *Resto* são trocados por novos mutantes criados, que formam o conjunto *Fundo* na nova geração. Dessa forma, $\mid Meio \mid = \mid Resto \setminus Fundo \mid$ soluções filhas são criadas. Como qualquer permutação dos vértices é uma solução viável para o problema TREEWIDTH , então todo cruzamento também irá gerar uma solução viável.

Para a implementação do BRKGA, foi utilizado a biblioteca *C++ brkgaAPI framework* desenvolvida por Toso e Resende [2015]. Essa biblioteca conta com um *framework* para sua utilização onde basta apenas a implementação de um **decodificador** para os indivíduos. Outra aplicação utilizando esse *framework* pode ser vista em Brandão et al. [2017].

Segundo Gonçalves et al. [2014], o *framework* do BRKGA possui os seguintes parâmetros: (a) tamanho da população ($p = |Elite| + |Resto|$); (b) fração pe de uma população correspondente ao conjunto $Elite$; (c) fração pm da população correspondente ao conjunto de mutantes $Fundo$; (d) probabilidade $rhoe$ que o filho tem de seus genes herdarem do pai do conjunto $Elite$; (e) número k de gerações sem melhora na solução até um *restart* ser feito.

O parâmetro *restart* citado anteriormente determina o seguinte procedimento: se uma população não consegue gerar indivíduos mais aptos após um certo número de gerações, então executa-se um reinício que mantêm os indivíduos mais aptos e completa-se a população com novos indivíduos gerados randomicamente. Para a escolha destes parâmetros, foi utilizando uma técnica de refinamento (*tuning*), que será discutido com detalhes na próxima seção.

Como forma de obter melhores resultados, foi utilizada a melhor heurística construtiva da literatura para o problema TREEWIDTH de modo a ser codificada em chaves aleatórias e injetada na

população do BRKGA. A heurística construtiva escolhida é a *fill-in*, pois obtém melhores resultados que a *min-degree*, como observado por Gaspers et al. [2019]. Ela executa o processo de triangulação escolhendo prioritariamente os vértices que têm vizinhança mais próxima de uma clique, ou seja, aqueles que demandam menos arestas a serem inseridas para que se obtenha uma clique. Foi implementado um algoritmo para injeção de indivíduos codificados como chaves aleatórias na população do BRKGA omitido neste artigo por questões de espaço.

Como dito anteriormente, o *framework* necessita apenas da implementação de um decodificador. O Algoritmo 1 mostra o funcionamento do decodificador proposto. Ele recebe como entrada um grafo G e o vetor ρ que armazena uma permutação dos vértices do grafo G. Inicialia-se então a variável tw com valor 0, na linha 1 e copia-se o grafo G para o grafo G', na linha 2. A linha 3 itera sobre todo vértice x pertencente a ρ. Na linha 4, atualiza-se o valor de tw com o máximo entre tw e o valor do grau do vértice x em G', dado pela função $deg_{G'}(x)$. Note-se que como estão sendo removidos sucessivamente vértices em G', então $deg_{G'}(x)$ retorna o grau do vértice x no momento corrente. As linhas 5 e 6 iteram sobre todos os pares de vizinhos a e b do vértice x. Na linha 7 é verificado se não há aresta entre a e b. Caso não houver, esta é adicionada ao grafo G' e desta forma toda a vizinhança de x é transformada em uma clique. Na linha 12, o vértice x é removido de G. Por fim, na linha 14, é retornado o valor tw computado da *treewidth* de ρ.

Algoritmo 1 Decodificador(ρ, G)

```
1:  tw ← 0
2:  G' ← G
3:  for all x ∈ ρ do
4:      tw ← max{tw, deg_G'(x)}
5:      for all vizinho a de x do
6:          for all vizinho b de x do
7:              if não existe aresta entre a e b em G' then
8:                  cria aresta entre a e b em G'
9:              end if
10:         end for
11:     end for
12:     remove x de G'
13: end for
14: retorna tw
```

4. Experimentos computacionais

Esta seção descreve os experimentos computacionais realizados com o BRKGA. As instâncias para os experimentos foram todas retiradas do desafio PACE 2017 de Dell et al. [2018]. Para os experimentos foi utilizado um computador Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16GiB de RAM, com sistema operacional Ubuntu 18.04 kernel 4.15.0-112-generic. O algoritmo foi implementado na linguagem *C++*.

4.1. Ajuste dos parâmetros

Conforme discutido na seção anterior, o BRKGA possui parâmetros a serem ajustados. Para isso, foi utilizada o IRACE de Alfaro-Fernández et al. [2017], para ajuste (*tuning*) de parâmetros. Foram então buscados valores apropriados para os parâmetros p (tamanho da população), pe (fração da população correspondente ao conjunto $Elite$), pm (fração da população correspondente ao conjunto de mutantes), $rhoe$ (probabilidade de o filho herdar um gene do pai $Elite$) e o número k de gerações

consecutivas sem melhoria na melhor solução encontrada até que um $restart$ seja feito. Foi dado um limite de 1000 execuções para o IRACE sobre 20 instâncias de tamanhos variados. A variação de valores possíveis a serem considerados para cada parâmetro seguiu as recomendações de Gonçalves e Resende [2011]. A Tabela 1 mostra os parâmetros considerados.

Tabela 1: Parâmetros considerados pelo IRACE para o BRKGA do problema TREEWIDTH.

Parâmetro	Valores	BRKGA
p	100, 150, 200	200
pe	0.10, 0.11, ..., 0.25	0.15
pm	0.10, 0.11, ..., 0.30	0.10
$rhoe$	0.50, 0.51, ..., 0.80	0.67
k	50, ,100	100

Os valores na coluna "BRKGA" foram aqueles selecionados pelo IRACE e usados na implementação dos algoritmos.

4.2. Versões do BRKGA

Conforme apresentado anteriormente, foi implementado um algoritmo para injetar soluções codificadas como chaves aleatórias na população inicial do BRKGA. Foram testadas duas versões do BRKGA com configurações de execução diferentes, sendo elas:

(1) 500 gerações ou 15 minutos, o que ocorrer primeiro, para gerar o indivíduo mais apto a partir de uma população totalmente aleatória (fase 1) e mais 500 gerações ou 15 minutos, o que ocorrer primeiro, injetando o indivíduo gerado na fase 1 e mais um obtido pela heurística construtiva como parte da população inicial (fase 2).

(2) 500 gerações ou 15 minutos, o que ocorrer primeiro, injetando a solução da heurística construtiva na população inicial (fase 1) e mais 500 gerações ou 15 minutos, o que ocorrer, primeiro injetando os dois indivíduos mais aptos da fase 1 na população inicial (fase 2).

Para cada versão do BRKGA para o problema TREEWIDTH, foram testadas as instâncias 2 e 13 do *dataset ex* do PACE 2017. Cada uma foi executada 10 vezes, variando-se a semente do algoritmo aleatório do BRKGA de 1 a 10.

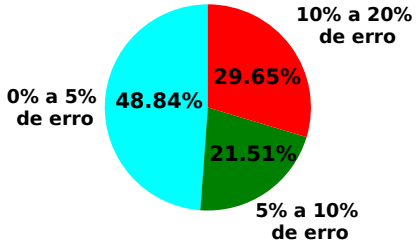
Para a instância 2, a versão 1 obteve melhores resultados e encontrou o valor ótimo em uma das execuções. Para a instância 13, em todas as execuções da versão 1 o BRKGA encontrou a solução ótima e da versão 2 em apenas quatro das dez execuções. A versão 1 foi então utilizada para o problema da TREEWIDTH no restante do trabalho e em todos os demais experimentos.

4.3. BRKGA vs heurísticas construtivas

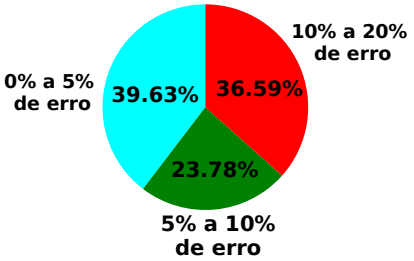
Além de avaliarem as principais heurísticas construtivas para o problema TREEWIDTH, *min-degree* e *fill-in*, Gaspers et al. [2019] também propuseram uma rotina para melhorar seus desempenhos. As heurísticas aprimoradas foram denominadas *turbo-fill-in* e *turbo-min-degree*. Assim como em Gaspers et al. [2019], foram testadas todas as 200 instâncias do benchmark *ex* do PACE 2017, onde o BRKGA foi executado uma vez com semente aleatória e, em todas, o BRKGA obtém resultados iguais ou melhores que as soluções obtidas pelas heurísticas construtivas *turbo-fill-in* e *turbo-min-degree* relatados em Gaspers et al. [2019].

Foi feito um estudo comparativo com relação ao erro relativo percentual entre o valor da solução obtida por cada algoritmo e o valor ótimo de cada instância. Seja opt o valor ótimo para uma instância e x o valor obtido por um algoritmo. O erro relativo é dado por $erro = \frac{x-opt}{opt} \times$

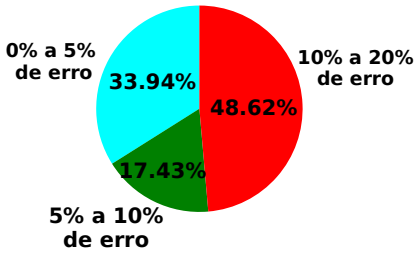
100%. Foram considerados os intervalos de faixas de erro de [0%, 5%], (5%, 10%], (10%, 20%]. Desta forma, foram obtidos os gráficos da Figura 2. Neles, apresenta-se o percentual de instâncias que obtiveram erros relativos em cada faixa, para cada algoritmo. Todos os algoritmos obtiveram soluções com erro percentual relativo inferior ou igual a 20% para todas as instâncias.



(a) BRKGA



(b) *turbo-fill-in*



(c) *turbo-min-degree*

Figura 2: Percentual de erro relativo para cada algoritmo em relação aos valores ótimos.

Os algoritmos *turbo-fill-in* e *turbo-min-degree* são aprimoramentos das heurísticas *fill-in* e *min-degree*, mas que não aumentam suas complexidades de tempo significantemente. Como o BRKGA, para o problema TREEWIDTH, utiliza a heurística construtiva *fill-in* para gerar indivíduos a serem codificados por chaves aleatórias e inseridos na população inicial, o tempo de execução total do BRKGA é necessariamente superior ao tempo para gerar esses indivíduos. Portanto, os tempos de execução do BRKGA são naturalmente superiores aos das heurísticas construtivas *turbo-fill-in* e *turbo-min-degree*.

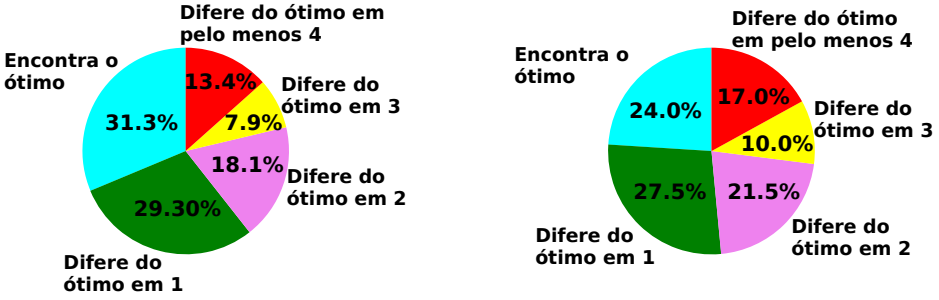
4.4. BRKGA vs algoritmo vice-campeão do PACE 2017

No desafio heurístico do PACE 2017 (veja Dell et al. [2018]) foi definido os vencedores que obtiveram os melhores resultados nas 200 instâncias do benchmark heurístico, *he*. O algoritmo campeão de Tamaki et al. utiliza o algoritmo exato de Tamaki [2019] como sub-rotina. Como as instâncias do *benchmark* exato (*ex*) utilizado por Gaspers et al. [2019] são relativamente pequenas, durante a execução do algoritmo campeão, a sub-rotina exata é executada para praticamente todas as instâncias. Desta forma, para este *benchmark*, o algoritmo de Tamaki et al. obtém soluções ótimas para todas as instâncias. Sendo assim, comparar a qualidade das soluções do BRKGA com o algoritmo de Tamaki et al. é análogo a comparar com um algoritmo exato para TREEWIDTH. Vale apontar que em instâncias em que essa sub-rotina exata não consegue finalizar sua execução em tempo viável, o algoritmo de Tamaki et al. retorna uma solução gulosa. Ao ser executado para cada instância, com tempo limite igual ao demandado pelo BRKGA em cada instância, o algoritmo de Tamaki et al. encontra uma solução melhor ou igual do que a obtida pelo BRKGA para todas as instâncias. Vale

ressaltar que o método exato desenvolvido por Tamaki [2019] e utilizado como sub-rotina no PACE 2017 possui um desempenho excelente, tendo sido premiado como melhor trabalho do European Symposium on Algorithms (ESA) de 2017 (Best Paper Award Track B - ESA 2017), veja Tamaki [2017]. A implementação utilizada nos experimentos apresentados nestes experimentos foi disponibilizada pelo autor no repositório GitHub.[4].

O vice-campeão do desafio foi o algoritmo de Strasser Hamann e Strasser [2018]. Vale ressaltar que conforme Dell et al. [2018], os dois primeiros colocados do desafio obtiveram desempenhos semelhantes. Para obter os resultados do algoritmo de Strasser foi utilizada a implementação disponibilizada pelo autor para o desafio no repositório GitHub.[5]

Para a comparação entre o BRKGA e o método de Strasser, cada instância foi executada cinco vezes pelo BRKGA, com sementes diferentes. Primeiramente, foi executado o BRKGA e o tempo médio das cinco execuções de cada instância do BRKGA foi utilizado como tempo limite para a execução do algoritmo de Strasser, que para quando atinge esse tempo e que, por ser um método randomizado, foi executado cinco vezes para cada instância, com sementes diferentes. A Figura 3 sumariza os resultados.



(a) BRKGA (b) Algoritmo de Strasser

Figura 3: Comparativo entre o BRKGA e o Algoritmo de Strasser com os valores da solução ótima.

Em seguida, foi efetuada uma comparação direta entre as cinco execuções de cada instância, comparando: a) o melhor execução de cada instância; b) o valor médio das soluções das cinco execuções de cada instância; e c) o valor da mediana das soluções das cinco execuções de cada instância. A Figura 4 resume estes resultados.

5. Considerações finais

Neste artigo foi discutido computações de maneira heurística para o problema da TREEWIDTH. Foi desenvolvido um algoritmo genético tendencioso de chaves aleatórias que conseguiu ser competitivo, superando as heurísticas conhecidas da literatura (Gaspers et al. [2019]) em qualidade de solução e obtendo resultados melhores que o segundo colocado no desafio PACE 2017 para a maioria das instâncias, considerando tanto melhor execução, média das execuções e mediana das cinco execuções por instância. Além disso, o BRKGA mostra-se competitivo com métodos exatos, conseguindo chegar ao ótimo ou ao ótimo + 1 em mais de 50% das instâncias com pelo menos cinco execuções por instância.

[4]https://github.com/TCS-Meiji/PACE2017-TrackA
[5]https://github.com/kit-algo/flow-cutter-pace17

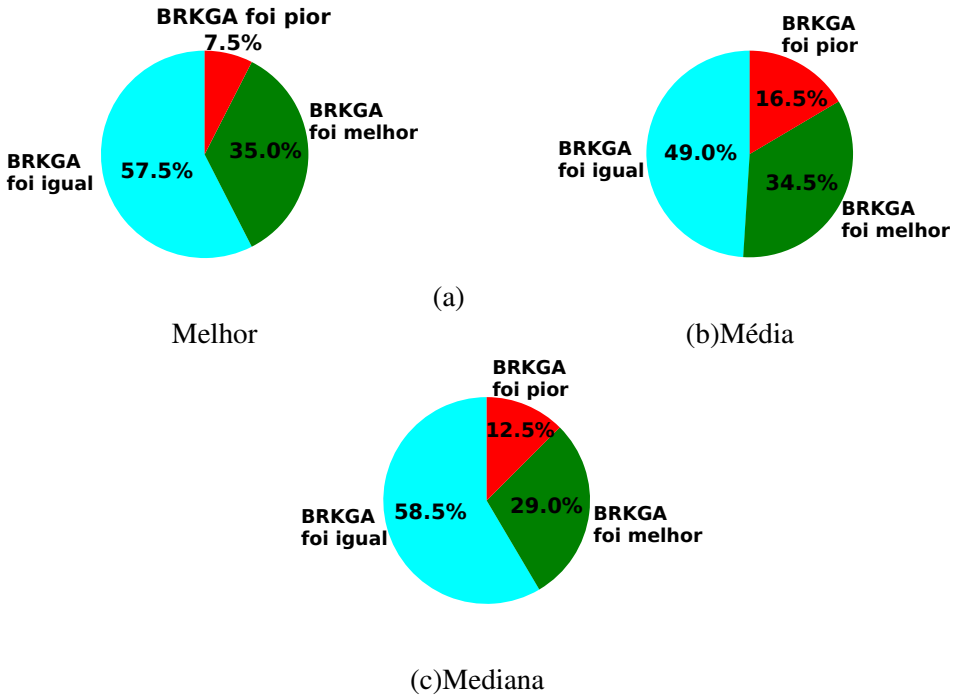


Figura 4: Comparativo entre o BRKGA e o algoritmo de Strasser.

Referências

Alfaro-Fernández, P., Ruiz, R., Pagnozzi, F., e Stützle, T. (2017). Exploring automatic algorithm design for the hybrid flowshop problem. In *12th Metaheuristics International Conference*, p. 201–203.

Arnborg, S., Corneil, D. G., e Proskurowski, A. (1987). Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284.

Bean, J. C. (1994). Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6(2):154–160.

Bodlaender, H. L. e Möhring, R. H. (1993). The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, 6(2):181–188.

Brandão, J. S., Noronha, T. F., Resende, M. G., e Ribeiro, C. C. (2017). A biased random-key genetic algorithm for scheduling heterogeneous multi-round systems. *International Transactions in Operational Research*, 24(5):1061–1077.

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., e Saurabh, S. (2015). *Parameterized algorithms*, volume 5. Springer.

Dell, H., Komusiewicz, C., Talmon, N., e Weller, M. (2018). The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Diestel, R., Schrijver, A., e Seymour, P. (2010). Graph theory. *Oberwolfach Reports*, 7(1):521–580.

Dirac, G. A. (1961). On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 25, p. 71–76. Springer.

Duarte, G. L., Eto, H., Hanaka, T., Kobayashi, Y., Kobayashi, Y., Lokshtanov, D., Pedrosa, L. L., Schouery, R. C., e Souza, U. S. (2021). Computing the largest bond and the maximum connected cut of a graph. *Algorithmica*, 83(5):1421–1458.

Fulkerson, D. e Gross, O. (1965). Incidence matrices and interval graphs. *Pacific journal of mathematics*, 15(3):835–855.

Gaspers, S., Gudmundsson, J., Jones, M., Mestre, J., e Rümmele, S. (2019). Turbocharging treewidth heuristics. *Algorithmica*, 81(2):439–475.

Gonçalves, J. F. e Resende, M. G. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525.

Gonçalves, J. F., Resende, M. G., e Toso, R. F. (2014). An experimental comparison of biased and unbiased random-key genetic algorithms. *Pesquisa Operacional*, 34(2):143–164.

Hamann, M. e Strasser, B. (2018). Graph bisection with pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23:1–34.

Kloks, T. (1994). *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media.

Patil, H. (1986). On the structure of k-trees. *Journal of Combinatorics, Information & System Sciences*, 11(2 4):57–64.

Robertson, N. e Seymour, P. D. (1986). Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322.

Spears, W. M. e De Jong, K. D. (1995). On the virtues of parameterized uniform crossover. Technical report, Naval Research Lab Washington DC.

Tamaki, H. (2017). Positive-Instance Driven Dynamic Programming for Treewidth. In Pruhs, K. e Sohler, C., editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, p. 68:1–68:13, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-049-1. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7880`.

Tamaki, H. (2019). Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311.

Toso, R. F. e Resende, M. G. (2015). A c++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*, 30(1):81–93.

Yannakakis, M. (1981). Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79.