A GRASP Approach for The Minimum Spanning Tree Under Conflict Constraints

Bruno José da Silva Barros[1], **Rian Gabriel S. Pinheiro**[2], **Luiz Satoru Ochi**[1], **Geymerson S. Ramos**[2]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ – Brasil

[2]Instituto de Computação – Universidade Federal de Alagoas (UFAL)
Maceió – Alagoas –Brasil

bruno_barros@id.uff.br, rian@ic.ufal.br, satoru@ic.uff.br

Abstract. *Given an undirected graph with weights on the edges and a set of conflicting edge pairs, the Minimum Spanning Tree Under Conflict Constraints (MSTCC) consists in finding minimum spanning tree with at most one edge of each conflicting pair and minimum cost. The problem is $\mathcal{NP}$-hard and was introduces recently at literature. This paper proposes a new approach for solving MSTCC, our method is based on the GRASP metaheuristic coupled with adaptive memory programming. We test the proposed heuristic on literature's instances with thousands of conflicts. The results indicate that the proposed algorithm was able to find all known optimal solutions and outperform the best-known heuristics for the MSTCC.*

KEYWORDS. Spanning Tree Problem. Conflict Constraints. GRASP. Combinatorial Optimization. Smart Algorithms.

Resumo. *Dados um grafo não direcionado com pesos nas arestas e um conjunto de pares de arestas conflitantes, a Árvore de Geradora Mínima com Restrições de Conflito (*AGMRC*) consiste em encontrar árvore geradora com no máximo uma aresta de cada par conflitante e custo mínimo. O problema é $\mathcal{NP}$-difícil e foi introduzido recentemente na literatura. Este artigo propõe uma nova abordagem para a solução do* AGMRC*, o método é baseado na meta-heurística GRASP com uso de uma memória adaptativa. A heurística proposta foi avaliada nas instâncias da literatura com milhares de conflitos. Os resultados indicam que o algoritmo proposto foi capaz de encontrar todas as soluções ótimas conhecidas e superar as heurísticas da literatura para o* AGMRC.

PALAVRAS CHAVE. Árvore Geradora Mínima. Restrições de Conflitos. GRASP. Otimização Combinatória. Algoritmos Inteligentes.

1. Introdução

Como já é conhecido, o problema da Árvore Geradora Mínima, (AGM, *Spanning Tree Problem* em inglês) é um problema de muita importância na área de otimização combinatória e inteligência computacional. Encontrar uma AGM de um grafo é um problema que pode ser resolvido em tempo polinomial [Kruskal 1956]. Apesar disso, algumas variações deste problema são $\mathcal{NP}$-difíceis. Este trabalho foca na variante conhecida com o problema da Árvore Geradora Mínima sob Restrições de Conflitos (AGMRC, *Spanning Tree Problem under Coflict Constraints* em inglês). Recentemente, alguns problemas clássicos foram considerados sob uma visão que considera restrições de conflitos. Um exemplo que ganhou esta abordagem é o *Bin Packing* com restrições de conflitos, tratado em Capua et al. (2015).

Definição: Dados um grafo $G = (V, E)$, uma função de peso ω que atribui a cada aresta $e \in E$ um peso $\omega(e)$ e um grafo de conflitos $\hat{G} = (E, \hat{E})$ em que os vértices de $\hat{G}$ são as arestas de G e $\hat{E}$ representa os pares de arestas conflitantes. O problema da AGMRC consiste em encontrar uma árvore geradora T de G com custo mínimo, tal que, se duas arestas $(e_i, e_j) \in \hat{E}$, então $(e_i, e_j) \notin T$, isto quer dizer, T é livre de conflitos.

1.1. Trabalhos Relacionados

O AGMRC foi introduzido por Darmann et al. (2009, 2011) juntamente com sua demonstração de $\mathcal{NP}$-dificuldade e uma formulação matemática. Esses dois primeiros trabalhos tiveram um caráter bastante teórico, por exemplo, os autores mostraram que este problema é fortemente $\mathcal{NP}$-difícil mesmo que o grafo de conflitos seja um conjunto de caminhos de tamanho igual a dois. E que é polinomial se for um conjunto de caminhos de tamanho igual a um. Outras propriedades do problema foram estudadas por Zhang et al. (2011). Além de disponibilizarem um conjunto de instâncias para o problema, os autores mostraram que se o grafo de conflitos é uma coleção de cliques disjuntas, o problema pode ser resolvido em tempo polinomial. Além disso, provaram que se G é um grafo cacto e $\hat{G}$ é livre, então, é possível encontrar uma solução viável (ou provar não existir solução viável) em tempo polinomial, entretanto, a versão de otimização mantém-se $\mathcal{NP}$-difícil. Zhang et al. (2011) ainda propuseram: um algoritmo construtivo, um método de busca local, duas meta-heurísticas (sendo uma dela um *Tabu Search* e a outra um *Tabu Thresholding*), e uma formulação de programação linear inteira. Vale destacar que foram os primeiros algoritmos heurísticos para o AGMRC.

Posteriormente, Samer and Urrutia (2013, 2015) propuseram um algoritmo *branch and cut* (B&C) que melhorara os resultados da literatura na época, porém os *gaps* obtidos ainda permaneceram altos para muitas instâncias. Bittencourt et al. (2016) apresentaram uma nova formulação matemática baseada em restrições *MTZ* (inicias dos criadores: Miller, Tucker e Zemlin). No trabalho de Barbosa and Delbem (2017), foi apresentado um algoritmo ILS (*Iterated Local Search*) e uma nova busca local. Apesar da melhora dos resultados da literatura, ainda existem instâncias com soluções viáveis que não se conhece o valor ótimo.

Recentemente, Carrabs et al. (2017) propuseram um algoritmo genético para o AGMRC. Em outro trabalho, Carrabs et al. (2018) propuseram novas equações válidas para o AGMRC e implementaram um novo algoritmo B&C, além disso geraram um novo conjunto de instâncias para o problema.

Muitas aplicações utilizam a AGM, como por exemplo: projetos de redes, transportes, comunicação, etc. Em muitas dessas aplicações, alguns conflitos podem surgir, impossibilitando que alguns pares de conexões sejam utilizados. Em Darmann et al. (2009), foi descrita uma aplicação bem específica do AGMRC, que é a instalação de um oleoduto com o objetivo de conectar vários países. A forma mais barata de construir o sistema é utilizando a árvore geradora mínima, mas por razões políticas, técnicas ou por questões internas das várias empresas, podem surgir muitos conflitos. Por exemplo, uma empresa pode não estar disposta a colaborar como outra. Nesse caso, o problema clássico da AGM não representa uma solução viável, já que é necessário respeitar os conflitos.

Neste trabalho é desenvolvida uma meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedure*) com memória adaptativa. Algoritmos similares foram utilizados com êxito na resolução de diversos problemas como: o Problema de Recobrimento de Rotas [Motta and Ochi 2009] e o Problema de Roteamento de Veículos com Múltiplas Origens [Neves and Ochi 2009], outro trabalho que também faz uso de memória adaptativa é Gonçalves et al. (2005). Entretanto estes trabalhos utilizam a memória apenas para calibrar o valor do α presente no GRASP, o que não acontece neste trabalho que utiliza a memória adaptativa para guiar o construtivo do GRASP. Com a técnica da memória adaptativa, a resolução se apoia na ideia de deixar o algoritmo aprender a resolver cada instância de teste, isto é, não engessar o algoritmo exageradamente, dessa forma, o algoritmo se torna flexível às características intrínsecas da instância.

A estrutura do trabalho é organizada da seguinte forma: na Seção 2 é apresentado o algoritmo proposto, a Seção 3 detalha os experimentos computacionais realizados, e por fim, a Seção 4 apresenta as conclusões e trabalhos futuros.

2. GRASP com Memória Adaptativa

A meta-heurística GRASP, proposta por Feo and Resende (1995), consiste em um método guloso aleatório seguido de uma busca local. Dentre as meta-heurísticas existentes, o GRASP tem se mostrado uma das mais competitivas. No entanto, assim como outros *frameworks* dedicados a resolver problemas $\mathcal{NP}$-difíceis, o método pode não conseguir resolver bem um problema específico caso não conte com adaptações para se adequar às características do problema que se pretende resolver. O GRASP possui várias adaptações conhecidas, muitas delas podem ser encontradas em Resende and Ribeiro (2016). Um dos mecanismos que é muito utilizado é o uso de memória.

No contexto de meta-heurísticas, vários tipos de memória podem ser utilizados. Na meta-heurística Busca Tabu por exemplo, utiliza-se uma memória de curto prazo para se evitar um retorno rápido às soluções já verificadas. Existem também meta-heurísticas que fazem uso de um conjunto elite de solução, como por exemplo o GRASP com reconexão de caminhos [Resende and Ribeiro 2016]. Outro tipo de memória, denominada memória adaptativa ou memória flexível, permite que a meta-heurística aprenda no decorrer de suas iterações. No caso do GRASP, o conhecimento desta memória de longo prazo é utilizado na fase de construção.

O Algoritmo 1 apresenta um pseudocódigo do GRASP proposto. No algoritmo, a variável T_b representa a melhor solução encontrada até o momento e a T' a solução corrente. Nesta proposta, optou-se por trabalhar com soluções inviáveis. Desta forma, é utilizada uma estratégia de penalização na função objetivo (*value*) dos conflitos que

aparecem na solução. Seja T uma árvore geradora de G, o valor da função objetivo (FO) para o AGMRC é dado por:

$$value(T) = \sum_{e \in T} \omega(e) + \Delta|(T \times T) \cap \hat{E}|$$

Onde Δ é a penalidade dos conflitos na FO e ω é função de custo das arestas. O Δ foi escolhido de forma a garantir que nenhuma solução livre de conflitos tivesse uma avaliação pior que alguma solução com conflitos. A memória adaptativa é representada pelo vetor ρ e é explicada na Seção 2.1. A criação da solução através do método construtivo é detalhada na Seção 2.2, já a busca local é apresentada na Seção 2.3. Por fim, os detalhes de implementação são abordados na Seção 2.4.

A seguir, serão descritas as atividade executadas em cada linha do algoritmo proposto. Primeiramente, as linhas 2 e 3 calculam as variáveis que são utilizadas nos pesos da memória adaptativa. A linha 4 produz uma solução inicial completamente aleatória (algoritmo de Kruskal). A memória adaptativa é inicializada na linha 7 com um vetor unitário de dimensão m. A linha 8 representa a condição de parada. O *loop* principal se inicia com a criação de uma solução gulosa aleatória na linha 9, em seguida a busca local é executada na linha 10 e a melhor solução atualizada na linha 12. Por fim, os pesos da memória adaptativa são calculados na linha 13 e a memória atualizada na linha 16.

Algoritmo 1 GRASP com memória adaptativa

1: **procedure** GRASPMA(grafo $G = (V, E)$, grafo de conflitos $\hat{G} = (E, \hat{E})$)
2: $\beta \leftarrow m/10$ ▷ β e ϵ, variáveis auxiliares da memória adaptativa
3: $\epsilon \leftarrow (|\hat{E}|/n)/5$
4: $T' \leftarrow$`RandomSolution`$(G, \hat{G})$
5: $T' \leftarrow$ `localSearch`$(T', G, \hat{G})$
6: $T_b \leftarrow T'$
7: $\rho[e] \leftarrow 1 \quad \forall e \in E$ ▷ Memória adaptativa, ver Seção 2.1
8: **while** Stop condition **do**
9: $T' \leftarrow$ `RandomGreedySolution`$(G, \hat{G}, \rho)$ ▷ Seção 2.2
10: $T' \leftarrow$ `localSearch`$(T', G, \hat{G})$ ▷ Seção 2.3
11: **if** $value(T') < value(T_b)$ **then**
12: $T_b \leftarrow T'$
13: $\beta \leftarrow \beta + \epsilon$
14: $\rho[e] \leftarrow 1 \quad \forall e \in E$
15: **end if**
16: $update(\rho, \beta)$ ▷ Atualização da memória adaptativa
17: **end while**
18: **return** T_b
19: **end procedure**

2.1. Memória adaptativa

A memória adaptativa tem como objetivo guardar informação sobre cada elemento que pode compor uma solução. Com essas informações em mãos, o algoritmo pode a cada

iteração construir uma nova solução com base no conhecimento prévio. No contexto do AGMRC, cada solução S corresponde a um conjunto de arestas, assim, a memória adaptativa irá associar um peso a cada aresta de acordo com sua utilização nas iterações passadas do algoritmo. Em outras palavras, arestas que fizeram parte de boas soluções terão pesos maiores que arestas que só apareceram em soluções ruins.

No Algoritmo 1, a memória adaptativa é representada pelo vetor ρ. Para calcular os pesos atribuídos a ρ são utilizadas variáveis auxiliares β e ϵ. Tai variáveis são utilizadas para controlar a reconstrução da nova solução a cada iteração, e são calculadas com os dados de entrada e atualizados durante a execução. A variável β representa os pesos atribuídos às arestas presentes em uma nova melhor solução, enquanto que ϵ é a taxa de crescimento de β durante o algoritmo.

Durante a execução do GRASP, o vetor ρ precisa ser atualizado (função *update* no Algoritmo 1), além dele, a variável β também sofre atualização durante a execução. A atualização funciona da seguinte forma, inicialmente $\rho[i] \leftarrow 1$, para todo $e_i \in E$. Em seguida, no decorrer do algoritmo, as arestas que aparecem na solução corrente passam a ter mais chances de aparecerem nas soluções futuras. Uma solução simples seria incrementar em uma unidade o peso de todas presentes nessa solução, ou seja, se $e \in T'$ então $\rho[e] \leftarrow \rho[e] + 1$. Entretanto, essa atualização não consegue ser eficiente em todas as instâncias, pois se o GRASP passar muitas iterações preso em um ótimo local e depois encontrar uma solução melhor, o vetor ρ fará com que as próximas iterações produzam soluções semelhantes à solução antiga e não à nova.

Para tratar deste problema — já que o objetivo é obter soluções mais semelhantes a T_b —, toda vez que T_b for atualizado, todas as entradas de ρ serão fixadas com o valor 1, com exceção das entradas referentes às arestas que aparecem em T_b que receberão o valor β. A medida que T_b é atualizado, o fator de atualização de pesos β também se torna mais agressivo como mostrado na linha 13 do pseudocódigo do GRASP. Vale lembrar que essa atualização mais intensa só é ativada quando T_b é atualizada.

2.2. Método construtivo

A fase de construção do GRASP é um processo iterativo em que, a cada iteração, os elementos que não pertencem à solução parcial são avaliados por uma função gulosa, que estima o ganho de sua inclusão na solução parcial. Eles são ordenados por seu valor estimado em uma lista chamada lista restrita de candidatos (RCL) e um deles é escolhido aleatoriamente e incluído na solução. O tamanho da RCL é limitado por um parâmetro $\alpha \in [0, 1]$. Este processo para quando uma solução viável é obtida. Com o parâmetro $\alpha = 0$, o método corresponde a uma implementação de um algoritmo puramente guloso, assim, o elemento de menor critério guloso sempre será selecionado em qualquer iteração. Por outro lado, o ajuste $\alpha = 1$ leva a um algoritmo completamente aleatório, uma vez que qualquer novo elemento pode ser adicionado com igual probabilidade em qualquer iteração.

Normalmente, todos candidatos na RCL possuem a mesma probabilidade de ser escolhidos. No entanto, qualquer função de distribuição de probabilidade pode ser utilizada para enviesar a escolha dos candidatos. Bresina (1996) propôs um método de enviesamento baseado no ordenamento dos candidatos de acordo com o critério guloso, em outras palavras, o i-ésimo melhor candidato teria como enviesamento uma função de

i. Posteriormente, Binato et al. (2000) aplicaram o método de Bresina (1996) na etapa construtiva do GRASP. Outros trabalhos, como Expósito et al. (2016), também aplicaram a mesma estratégia com sucesso. Neste trabalho, diferente da estratégia de Bresina (1996), o enviesamento não será calculado em função do ordenamento, e sim, em função da memória adaptativa. Com relação ao tamanho da RCL, será utilizado um valor $\alpha = 1$, ou seja, todos os elementos farão parte da RCL, no entanto, tal estratégia não será completamente aleatória por conta do enviesamento utilizado.

Para calcular o enviesamento, o método construtivo aleatório do GRASP proposto utiliza o conhecimento da memória adaptativa para gerar uma nova solução a cada iteração. A memória adaptativa implementa uma função de enviesamento para escolha das arestas. Seja e_j uma aresta do grafo G, e $\rho(e_j)$ o peso da memória contabilizando à aresta e_j. A probabilidade de e_j aparecer na solução é dada por:

$$P(e_j) = \frac{\rho(e_j)}{\sum_{i=1}^{|E|} \rho(e_i)}.$$

Com as probabilidades calculadas, a construção de uma nova solução é iniciada. Para isso, foi utiliza uma variação aleatória do algoritmo de *Kruskal*, em que as arestas são escolhidas em um método de roleta de acordo com suas probabilidades. Note que uma aresta já escolhida não precisa ter $P(e) = 0$, basta ignorar caso alguma seja selecionada na roleta. Também vale destacar que a primeira solução construída na linha 4 do Algoritmo 1 é uma exceção à construção gulosa aleatória. Essa solução é construída de forma totalmente aleatória.

2.3. Busca Local

A busca local (BL) utilizada neste trabalho foi proposta por Barbosa and Delbem (2017) e é denominada BL4ex. Ela consiste em trocar as arestas da solução corrente duas a duas por outras arestas que não estão na solução e que reconectam a árvore sem formar ciclos. Dada uma árvore geradora T de G, a estrutura de vizinhança é definida como:

$$N(T) = \{T \setminus \{e_i, e_j\} \cup \{e_k, e_l\} \mid e_i, e_j \in T \text{ e } e_k, e_l \in E \setminus T\}.$$

Outra vizinhança, proposta por Zhang et al. (2011), foi testada, porém, o seu uso, a não ser nas instâncias menores, faz com que a heurística fique presa facilmente em ótimos locais. Dessa forma, optou-se por trabalhar apenas com a BL4ex, que apesar de ser mais pesada, apresenta resultados muito melhores. Além disso, na prática, é possível implementá-la de forma eficiente. A BL4ex tem complexidade $O(n^2m^2)$.

2.4. Detalhes de Implementação

Como a BL4ex é um pouco pesada, algumas técnicas se fazem necessárias para fazer com que seja possível acelerar o algoritmo.

Em Barbosa and Delbem (2017), os autores explicam que par ira acelerar a execução da BL4ex, antes de testar a combinação das arestas para reconectar a árvore, eles dividem as arestas em três grupos, essa parte do código será chamada de fase de separação. Ao remover duas arestas da solução corrente T, serão formadas três componentes conexas V_A, V_B e V_C. Com isso, considerando o conjunto das arestas que não

produzem ciclo, pode-se particionar este conjunto nas partes E_{AB}, E_{AC} e E_{BC}, de acordo com as componentes conexas que as arestas conectam. Assim, as arestas de E_{AB} conectam V_A a V_B; as arestas de E_{AC} conectam V_A a V_C e E_{BC} conectam V_B a V_C. Tal particionamento realmente faz muita diferença, porém o detalhe que mais impactou no tempo computacional foi a exclusão das arestas com custo maior do que a soma dos custos das duas arestas já retiradas da solução. E para fazer isso, não é necessário um novo pré-processamento, basta excluir as arestas na fase de separação.

Para calcular a função objetivo (FO) é utilizada a forma incremental descrita por Zhang et al. (2011). Com isso, durante a verificação das novas arestas não é preciso calcular o valor da FO, basta computar o acréscimo das duas novas arestas e só no final computar o valor completo da FO. Para fazer isso rapidamente, é necessário manter uma estrutura auxiliar que guarda as informações da solução corrente. A estrutura precisa manter o número de conflitos de cada aresta na solução corrente, atualizando-as a cada modificação. Assim, a cada iteração da BL4ex é possível verificar em $O(1)$ se uma aresta causa conflitos na solução e o custo de adicioná-la na solução.

Ao reconstruir a solução durante o processo de busca local, é necessária a garantia de que as soluções sejam acíclicas. Existe a possibilidade de utilizar a estrutura *union-find* — normalmente implementada no algoritmo de Kruskal. Uma explicação do funcionamento desta estrutura pode ser encontrada em Cormen et al. (2002). No entanto, mesmo a estrutura *union-find* sendo muito rápida, em instâncias de grande porte essa estratégia se torna lenta, já que a cada iteração tudo precisa ser reconstruído, tornando assim, todo o processo lento. Isso justifica utilização da estratégia abordada em Barbosa and Delbem (2017). Trata-se de: definir dois atributos para cada vértice; a partir de um vértice inicial, fazer uma busca em profundidade (DFS, *deep first search*); e marcar o tempo que cada vértice foi alcançado juntamente com o tempo que a busca fechou o vértice.

Com isso, considere que a aresta $e = (u, v)$ foi removida da solução e seja $d(x)$ o tempo de descoberta do vértice x e $f(x)$ o seu tempo de fechamento. Para saber se x é descendente de u, basta verificar se $d(x) > d(u)$ e $f(x) < f(u)$. Por fim, a aresta $e = (x, y)$ reconectará a solução se x e y não possuírem descendentes em comum, ou seja, um será descendente de u e o outro não. Analogamente, esta verificação é estendida para a remoção de duas arestas.

Na solução proposta, a verificação de ciclo foi aprimorada. Foi utilizada a DFS depois de ser retirada uma aresta da solução. Em seguida, é executada a DFS a partir de cada um dos extremos da aresta retirada, marcando cada vértice descoberto com uma *flag* referente ao vértice em que a busca se iniciou. Assim, para verificar se $e = (x, y)$ reconecta a solução basta testar se $flag(x) \neq flag(y)$.

3. Testes Computacionais

O GRASP foi implementado na linguagem de programação C e executado em um computador com sistema operacional Ubuntu 18.04.02, com arquitetura de 64 bits, 16 GB de RAM e processador Intel Core i7-4770 3.40GHz.

Para realizar os testes e comparar com outros algoritmos da literatura, foram utilizadas as instâncias geradas por Zhang et al. (2011). Tais instâncias são divididas em dois grupos denominados `type1` e `type2`. As instâncias do grupo `type1` são difíceis de serem resolvidas, neste grupo, várias instâncias se mostraram inviáveis, outras apresentam

soluções viáveis mas não se conhece o valor ótimo. Ainda neste grupo, apenas 8 instâncias possuem otimalidade comprovada, por outro lado, existe um subconjunto que não se sabe se existe solução viável. Em contrapartida, o grupo `type2` é formado por instâncias fáceis com todos os ótimos conhecidos. Para essas instâncias, as meta-heurísticas não têm dificuldade em resolvê-las. Por este motivo, os resultados apresentados aqui foram restringidos apenas para instâncias difíceis (`type1`) com viabilidade comprovada. Teste computacionais nas instâncias fáceis não acrescentam muitas informações, da mesma forma que a execução de testes em instâncias em que não é atingida uma solução viável.

A Tabela 1 mostra as instâncias `type1` utilizadas neste trabalho. A coluna `Instância` identifica a instância, a colunas $|V|$, $|E|$ e $|\hat{E}|$ apresentam o número de vértices, o número de arestas e o número de conflitos, respectivamente. A coluna L_i apresenta os limites inferiores [obtidos pelos B&C de Samer and Urrutia (2013, 2015) e de Bittencourt et al. (2016)] e os limites superiores obtidos pelas heurísticas testadas são encontrados na coluna L_s. Note que, quando o valor dos limites são idênticos, significa que o valor é ótimo.

Tabela 1. Instâncias `type1`

Instância	$\|V\|$	$\|E\|$	$\|\hat{E}\|$	L_i	L_s
I_1	50	200	199	**708**	**708**
I_2	50	200	398	**770**	**770**
I_3	50	200	597	**917**	**917**
I_4	50	200	995	**1324**	**1324**
I_5	100	300	448	**4041**	**4041**
I_6	100	300	897	**5658**	**5658**
I_7	100	500	1247	**4275**	**4275**
I_8	100	500	2495	**5997**	**5997**
I_9	100	500	3741	6707,8	7523
I_{10}	200	600	1797	13264	13815
I_{11}	200	800	3196	20941,5	21480

A seguir, serão apresentados os resultados dos experimentos computacionais. Foram realizadas comparações do GRASP proposto com os métodos da literatura: ILS e MEGA (Multiethnic Genetic Algorithm).

3.1. GRASP × ILS

Nesta seção, o método proposto é comparado com o ILS de Barbosa and Delbem (2017). Ao conhecimento dos autores, essa foi a proposta algorítmica com melhores resultados na literatura. Os autores de Barbosa and Delbem (2017), gentilmente, forneceram o código fonte com duas versões do ILS, no entanto, apenas a melhor versão será comparada.

Os dois algoritmos foram executados na mesma máquina e utilizando os mesmos critérios de parada. Cada algoritmo foi executado 30 vezes com limite de $m = |E|$ iterações. Entretanto, não foi possível obter os mesmos resultados divulgados pelos autores, já eles utilizaram sementes aleatórias na geração dos números aleatórios. Para facilitar a reprodução dos resultados, em ambos os algoritmos foram executados testes com as sementes inteiras de 1 até 30, uma para cada execução de cada instância.

Nos testes computacionais, o ILS não alcançou o melhor valor conhecido na instância I_{10}, mas nos resultados apresentados por Barbosa and Delbem (2017) esse valor

Tabela 2. Comparação do ILS com o GRASP

Instância	ILS					GRASP				
	Média	Best	GAP	#best	Tempo (s)	Média	Best	GAP	#best	Tempo (s)
I_1	**708**	**708***	0,00%	30	0,367	**708**	**708***	0,00%	30	0,182
I_2	**770**	**770***	0,00%	30	0,372	**770**	**770***	0,00%	30	0,186
I_3	919,3	**917***	0,00%	16	0,363	**917**	**917***	0,00%	30	0,183
I_4	1326,5	**1324***	0,00%	22	0,353	**1324**	**1324***	0,00%	30	0,179
I_5	4041,4	**4041***	0,00%	27	2,544	**4041**	**4041***	0,00%	30	1,723
I_6	5672,2	**5658***	0,00%	2	1,89	5706,2	**5658***	0,00%	5	1,533
I_7	4275,4	**4275***	0,00%	24	6,605	**4275**	**4275***	0,00%	30	4,321
I_8	6026	**5997***	0,00%	2	6,266	6043,9	**5997***	0,00%	3	3,946
I_9	7550,3	**7523**	0,00%	7	5,611	7631,5	**7523**	0,00%	7	3,784
I_{10}	13924,7	13820	0,04 %	0	22,338	14144,2	**13815**	0,00%	1	27,28
I_{11}	21571,8	**21480**	0,00%	1	46,964	21721,4	**21480**	0,00%	2	44,166

é alcançado. Para cada algoritmo, a coluna `Média` equivale à média dos custos obtidos nas 30 execuções, `Best` ao melhor valor, `GAP` ao *gap* da melhor solução para o melhor valor conhecido, `#best` à quantidade de vezes que o melhor valor conhecido foi obtido e `Tempo` ao tempo médio em segundos. O valores em negrito são os melhores valores já encontrados, e os marcados com (*) são valores ótimos.

O *gap* é calculado como:

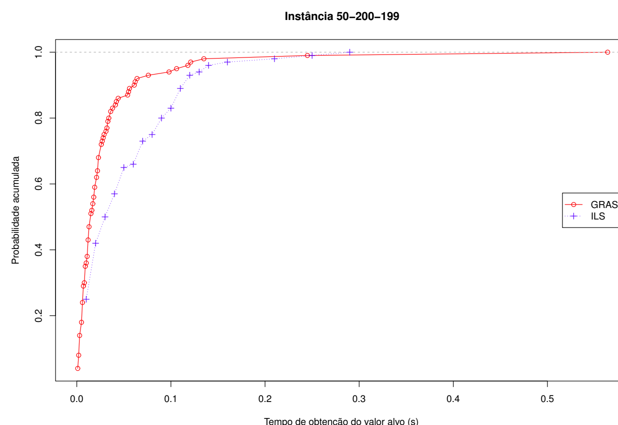
$$gap = 100 \times \frac{value(x) - value(L_s)}{value(L_s)}$$

De acordo com estes resultados, fica claro que o GRASP consegue chegar mais vezes nas melhores soluções, inclusive, em 6 instâncias encontra o ótimo nas 30 execuções enquanto o ILS faz isso apenas em 2 instâncias. Vale destacar que em uma instância o ILS não chega no melhor valor conhecido. Com relação ao tempo médio de cada algoritmo, os resultados mostram que o GRASP executa mais rapidamente que o ILS, com exceção em apenas uma instância.

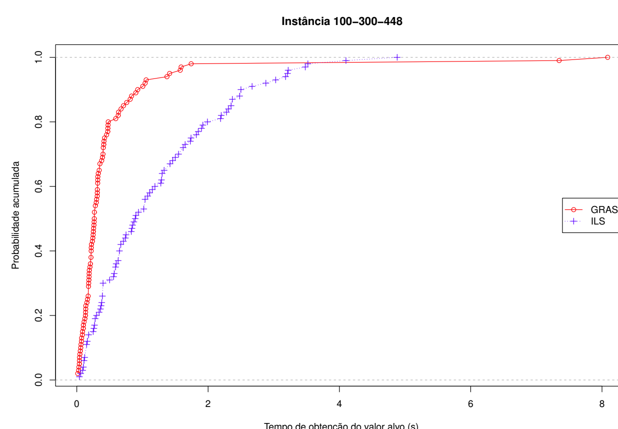
Outros experimentos foram realizados por meio de gráficos `TTTPlot` [Aiex et al. 2007], com estes gráficos, é possível verificar a probabilidade que cada algoritmo possui de alcançar alvos em função do tempo de execução. Neste trabalho, os alvos considerados foram os limites superiores. Dessa forma, cada algoritmo foi executado 100 vezes nas instâncias I_1 e I_5 com critério de parar ao alcançar o alvo. O resultado pode ser verificado na Figura 1. Pode-se observar que o GRASP apresenta uma convergência mais rápida aos alvos que o ILS.

3.2. GRASP × MEGA

Carrabs et al. (2017) desenvolveram um algoritmo genético denominado MEGA. Como o código-fonte não foi disponibilizado, não foi possível executar os testes sob as mesmas condições. A condição de parada utilizado no MEGA é bem diferente, já que o algoritmo utiliza mais de uma busca local e o número de iterações do MEGA pode variar de acordo coma a busca local utilizada. Além disso, o número de iterações é decidido em tempo de execução, causando uma grande variabilidade nos tempos computacionais em cada execução. Isso causa uma grande diferença nos testes do MEGA e do GRASP. O MEGA foi executado apenas 5 vezes, por isso a coluna #best foi removida nesta comparação.



(a) Instância I_1



(b) Instância I_5

Figura 1. Gráfico *Time to Target* **de duas Instâncias**

Mas para minimizar a diferença entre os testes, a coluna Pior foi adicionada no GRASP, essa coluna informa qual o pior valor da FO nas 30 execuções do GRASP, essa coluna serve de comparação com a melhor execução do MEGA.

Sendo assim, os valores obtidos pelo GRASP na comparação anterior, são comparados com os valores reportados no artigo do MEGA. Mesmo com essas diferenças é possível notar que o GRASP tem melhor performance como mostra a Tabela 3. Neste experimento, o MEGA foi executado apenas 5 vezes enquanto o GRASP foi executado 30 vezes. Os *hardwares* utilizados possuem configurações bem semelhantes. Considerando o tempo médio de execução de cada algoritmo, o GRASP executa por mais tempo em duas instâncias. O MEGA não chega nos melhores valores da literatura em várias instâncias e nas instâncias I_6 e I_{10} a ausência da média indica que provavelmente os algoritmo só encontrou solução viável em uma das 5 execuções, isso já mostra uma grande deficiência em relação ao GRASP que em 30 execuções nunca retornou uma solução inviável. Os valores médios também são bem melhores na coluna do GRASP. Outro fator em favor do GRASP está na coluna `Pior` do GRASP, nela é possível notar que a pior execução do

Tabela 3. Comparação do MEGA com o GRASP

Instância	MEGA				GRASP				
	Média	Best	GAP	Tempo (s)	Média	Best	GAP	Tempo (s)	Pior
I_1	708	**708***	0,00%	0,71	708	**708***	0,00%	0,182	708
I_2	770	**770***	0,00%	0,68	770	**770***	0,00%	0,186	770
I_3	917	**917***	0,00%	0,63	917	**917***	0,00%	0,183	917
I_4	1365,4	1336	0,91%	0,66	1324	**1324***	0,00%	0,179	1324
I_5	4099,2	4088	1,16%	2,39	4041	**4041***	0,00%	1,723	4041
I_6	-	6095	7,72%	1,81	5706,2	**5658***	0,00%	1,533	6091
I_7	4291,2	**4275***	0,00%	5,18	4275	**4275***	0,00%	4,321	4275
I_8	6325	6199	3,37%	5,12	6043,9	**5997***	0,00%	3,946	6131
I_9	7788	7665	1,89%	3,72	7631,5	**7523**	0,00%	3,784	8026
I_{10}	-	15029	8,79%	12,23	14144,2	**13815**	0,00%	27,28	14892
I_{11}	22350,8	22110	2,93%	23,42	21721,4	**21480**	0,00%	44,166	23220

GRASP é melhor ou igual à melhor execução do MEGA na maioria das instâncias, apenas em duas instâncias o *best* do MEGA é melhor que o pior valor encontrado pelo GRASP considerando as 30 execuções de cada instância. Com isso, fica evidente a superioridade do GRASP em relação ao MEGA. Acredita-se que o algoritmo genético (MEGA) teve um baixo desempenho pelo fato de que a combinação de duas boas soluções no MEGA, provavelmente, pode gerar filhos ruins.

4. Conclusões e Trabalhos Futuros

Neste trabalho foi desenvolvido uma meta-heurística GRASP com memória adaptativa para tratar o problema da Árvore Geradora Mínima sob Restrições de Conflitos. Foi utilizada uma busca local de tamanho $O(n^2m^2)$. Para isso, formas eficientes de implementá-la foram estudadas e bons resultados foram obtidos. Foi feita uma comparação do GRASP com algoritmos estado da arte: ILS e MEGA, utilizando instâncias da literatura. Nos testes realizado, foi possível constatar que o GRASP teve um desempenho superior aos demais tanto em termos de qualidade das soluções e tempo computacional.

Como trabalhos futuros, serão buscadas novas estruturas de vizinhança e novas formas de acelerar a busca local. Além disso, serão estudadas formas mais eficientes de atualizar os pesos/memória adaptativa além do uso de mineração de dados na fase construtiva.

Referências

Aiex, R. M., Resende, M. G. C., and Ribeiro, C. C. (2007). Ttt plots: a perl program to create time-to-target plots. *Optimization Letters*, 1(4):355–366.

Barbosa, M. A. L. and Delbem, A. C. B. (2017). Uma busca local iterada para o problema da árvore geradora mínima sob restrições de conflitos. *Anais do XLVIII Simpósio Brasileiro de Pesquisa Operacional*.

Binato, S., J. Hery, W., and M. Loewenstern, D. (2000). A grasp for job shop scheduling. *Essays and Surveys on Metaheuristics*, 15.

Bittencourt, Y. B., , Campêlo, M., and Dias, F. C. S. (2016). Formulaçao mtz para o problema da Árvore geradora mınima sob restriçoes de conflito. *Anais do XLVIII Simpósio de Pesquisa Operacional*.

Bresina, J. L. (1996). Heuristic-biased stochastic sampling. In *AAAI/IAAI, Vol. 1*, pages 271–278.

Capua, R., Frota, Y., Vidal, T., and Ochi, L. S. (2015). Um algoritmo heurıstico para o problema de bin packing com conflitos. *Anais do XLVII Simpósio Brasileiro de Pesquisa Operacional*.

Carrabs, F., Cerrone, C., and Pentangelo, R. (2017). A multiethnic genetic approach for the minimum conflict weighted spanning tree problem. *Networks*.

Carrabs, F., Cerulli, R., Pentangelo, R., and Raiconi, A. (2018). Minimum spanning tree with conflicting edge pairs: a branch-and-cut approach. *Annals of Operations Research*, pages 1–14.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2002). *Algoritmos: Teoria e Prática*. Elsevier, 2 edition. ISBN 85-352-0926-3.

Darmann, A., Pferschy, U., and Schauer, J. (2009). Determining a minimum spanning tree with disjunctive constraints. In *International Conference on Algorithmic DecisionTheory*, pages 414–423. Springer.

Darmann, A., Pferschy, U., Schauer, J., and Woeginger, G. J. (2011). Paths, trees and matchings under disjunctive constraints. *Discrete Applied Mathematics*, 159(16):1726–1735.

Expósito, A., Brito, J., and Moreno, J. A. (2016). A heuristic-biased grasp for the team orienteering problem. In *Conference of the Spanish Association for Artificial Intelligence*, pages 428–437. Springer.

Feo, T. A. and Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.

Gonçalves, L. B., Ochi, L. S., and Martins, S. L. (2005). A grasp with adaptive memory for a period vehicle routing problem. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 1, pages 721–727. IEEE.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

Motta, L. C. and Ochi, L. S. (2009). Metaheurísticas com memória adaptativa para o problema de recobrimento de rotas. In *IX Congresso Brasileiro de Redes Neurais*.

Neves, T. A. and Ochi, L. S. (2009). Grasp com memória adaptativa na solução de um problema de roteamento de veículos com múltiplas origens. *Anais do XXXVIII Simpósio Brasileiro de Pesquisa Operacional*.

Resende, M. G. and Ribeiro, C. C. (2016). *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. Springer-Verlag New York, 1 edition.

Samer, P. and Urrutia, S. (2013). Um algoritmo de branch and cut para árvores geradoras mınimas sob restriçoes de conflito. *Anais do XLV Simpósio Brasileiro de Pesquisa Operacional*.

Samer, P. and Urrutia, S. (2015). A branch and cut algorithm for minimum spanning trees under conflict constraints. *Optimization Letters*, 9(1):41–55.

Zhang, R., Kabadi, S. N., and Punnen, A. P. (2011). The minimum spanning tree problem with conflict constraints and its variations. *Discrete Optimization*, 8(2):191–205.