Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

DCC059 - Teoria dos Grafos - Semestre 2016-3

Eduardo Melão
Iago Gomes de Lima Rosa
Jhuan Barbosa da Silva e Cedro
Mateus Teixeira Magalhães

Implementação de Heurísticas em Grafos para o Problema da Árvore de Steiner

Juiz de Fora

Dezembro de 2016

Eduardo Melão
Iago Gomes de Lima Rosa
Jhuan Barbosa da Silva e Cedro
Mateus Teixeira Magalhães

Implementação de Heurísticas em Grafos para o Problema da Árvore de Steiner

Relatório do trabalho final de Teoria dos Grafos, parte integrante da avaliação da disciplina.

Professor: Stênio Sã Rosário F. Soares

Juiz de Fora

Dezembro de 2016

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O problema da Árvore de Steiner consiste em, dado um grafo $G = (V, E)$ não-direcionado e um subconjunto de vértices de V (chamados de **nós terminais**), conectá-los aciclicamente e de modo que a soma dos pesos das arestas usadas para conectar os nós terminais seja a menor possível. A árvore resultante é chamada de **Árvore de Steiner**, os vértices intermediários usados para conectar os nós terminais são chamados de **nós de Steiner** e as arestas são chamadas de **arestas de Steiner**.

Abaixo temos a ilustração do problema. Em azul temos os nós terminais, em verde as arestas de Steiner e as pontas das arestas verdes que não são terminais são os nós de de Steiner.

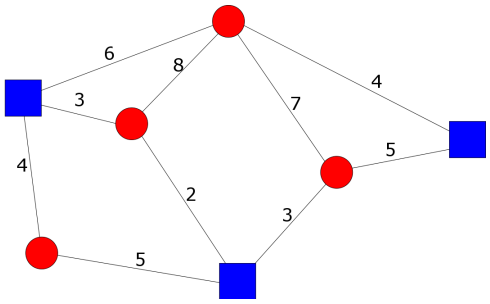

Figura 1 – Exemplo de grafo com nós terminais.

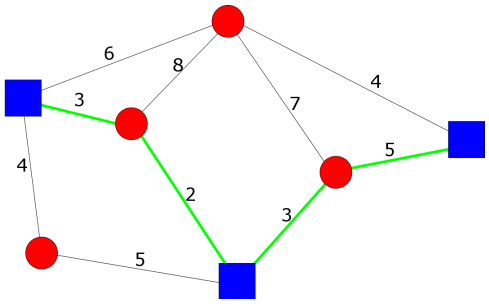

Figura 2 – Exemplo de árvore de Steiner para os nós terminais dados.

O problema da Árvore de Steiner é semelhante ao problema da árvore geradora mínima, já que ambos buscam uma árvore de peso mínimo que interliga vértices. A diferença é que, no problema da Árvore de Steiner, não necessariamente o conjunto de vértices contém todos os vértices do Grafo. Assim, a árvore geradora mínima é um caso particular da árvore de Steiner no qual todos os nós do grafo são nós terminais.

Podem existir várias árvores de Steiner diferentes para um determinado conjunto de vértices iniciais, sendo essas diferenças ocasionadas pelas diferentes formas de análise do problema que possivelmente podem existir. Em resumo, dado um conjunto de N pontos terminais, procura-se a árvore de menor peso que interliga todos.

Alguns exemplos de aplicação da árvore de Steiner: área de redes de computadores, distribuição de vídeo e conferências multimídia que utilizam comunicação *multicast* para transmissão de dados.

O presente trabalho consiste em estudar a árvore de Steiner, usando heurísticas para a solução e implementação com listas de adjacências e tabelas *hashing*.

2 ESTRUTURAS DE DADOS UTILIZADAS

Para o problema, é mostrado a seguir como foram implementadas as estruturas do grafo (vértices e arcos). O conjunto dessas estruturas forma um grafo que, por sua vez, deve ser representado em alguma outra estrutura adequada. Para este trabalho, foram escolhidas duas estruturas para essa representação: **lista de adjacências** e a **tabela *hash* de adjacências**. São detalhadas a seguir as principais informações sobre a estrutura implementada.

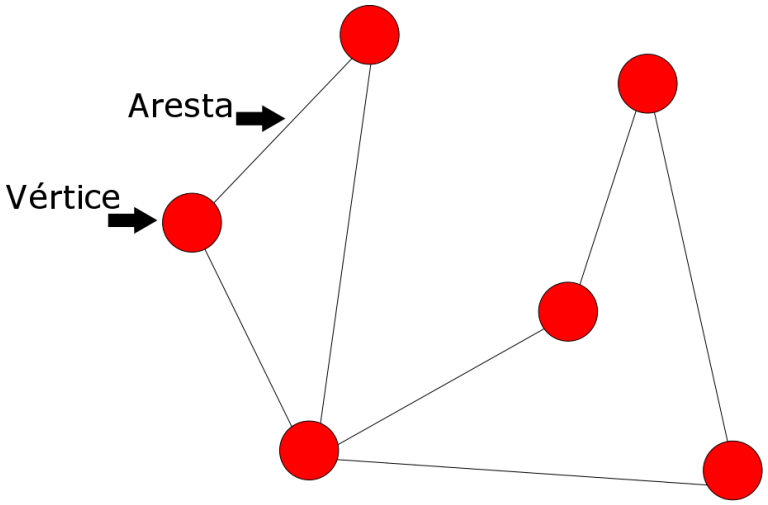

Figura 3 – Estrutura do grafo e seus componentes

2.1 VÉRTICE

Essa estrutura é composta principalmente por: `id` para sua identificação; `grau` (número total de arcos com origem no vertice); `idArvore` para identificação de sua componente conexa no grafo; `marcado` para o auxílio em várias funções como, por exemplo, a de busca em profundidade (no problema da Árvore de Steiner podemos utilizar este marcador para sinalizar os nós terminais); `nivel` para a possível identificação de seu nível em uma arvore e é auxiliar na heurística construtiva; conjunto de arcos cuja origem é este vértice.

2.2 ARCO

O arco é uma estrutura para representar as ligações entre pares de vértices. Essa estrutura composta principalmente por: `id`, para identificação da arco; referências para os pares de vértices que o compõe; `marcado` para auxiliar em funções que identificam conexidade, inclusive no próprio problema; `peso` para armazenar o custo do caminho entre os vértices.

Em grafos não direcionados (identificados pela variável `direcionado`), não há restrição de sentido na conexão entre dois nós. Nesses casos, chamamos as conexões de arestas (equivalente a dois arcos iguais com extremidades invertidas) e cada arco possui uma referência para o arco de direção oposta (`dual`).

2.3 GRAFO

A implementação da estrutura do grafo foi feita de duas formas: usando lista encadeada de adjacências e usando tabela *hash* de adjacências. Para isso, foram criadas a classe abstrata `Grafo` e as classes herdeiras instanciáveis `GrafoLista` e `GrafoHash`. O mesmo foi feito com as classes `No` (abstrata), `NoLista` e `NoHash` (herdeiras). Como a diferença entre as classes herdeiras está na estruturação dos vértices e arcos de `Grafo`, foram criados para cada uma das classes herdeiras ditas acima iteradores (`it`) para o conjuntos de `NoLista`, `NoHash` e `Arco`, de acorco com a necessidade da classe. Para auxiliar com as iterações, foram criadas as seguintes funções abstratas:

- `itInicio()`: Atribui ao iterador o primeiro elemento do conjunto;
- `itProx()`: Atualiza o iterador para o próximo elemento do conjunto;
- `itEhFim()`: Verifica se o iterador corresponde ao último elemento do conjunto;
- `pushIt()`: Empilha iterador atual em `itPilha`;
- `popIt()`: Desempilha iterador de `itPilha`;
- `getIt()`: Retorna o iterador.

Com as funções de iteração, a implementações de todas as funcionalidades não estruturais puderam ser implementadas unicamente em `Grafo`. Com o multi-uso destas funções viu-se a necessidade implementar uma pilha de iteradores (`itPilha`), para os casos em que mais de um iterador é necessário.

A Figura 6 mostra um diagrama de classes simplificado que ajuda a entender a estruturação principal do trabalho.

2.3.1 **Lista Encadeada de Adjacências**

Um grafo representado por uma lista de adjacência convencional possui uma lista encadeada de vértices e cada vértice também possui uma lista encadeada de arcos, que representam os seus vértices adjacentes. A Figura 4 ilustra a representação do grafo por lista de adjacência.

A classe `GrafoLista` possui uma lista encadeada de `NoLista` e esta por sua vez possui uma lista encadeada de arcos, como mostra o diagrama da Figura 6

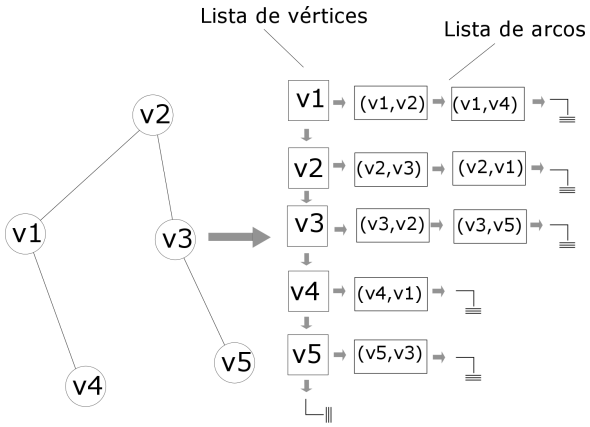

Figura 4 – Representação lista de adjacências

2.3.2 **Tabela Hash de Adjacências**

A estruturação do grafo com o uso tabelas *hash* (ou tabelas de dispersão)[3] é uma abordagem que visa reduzir o tempo de busca e, consequentemente, inserção e remoção de objetos na estrutura do grafo. Com isso, o grafo possui em sua estrutura uma tabela *hash* contendo todos os vértices do grafo e cada vértice contém também uma tabela *hash* contendo todos os seus arcos.

É importante ressaltar que essa estruturação aumenta o gasto com memória, uma vez que o tabela possui tamanho fixo, baseado no fator `ordem` definido na criação do grafo, e que a tabela necessita de um tamanho ligeiramente maior para reduzir o número de colisões.

Para auxiliar com as tabelas *hash*, foi criada a classe `THash<T>`, que implementa as funções de inserção, remoção, busca e iterações na tabela de um tipo de dados genérico `T`. Para um objeto desta classe ainda é possível definir as funções `funcaoHash` e `funcaoReHash` a serem utilizadas. A classe `GrafoHash` possui uma tabela de `NoHash` e esta por sua vez possui uma tabela de arcos, como mostra o diagrama da Figura 6

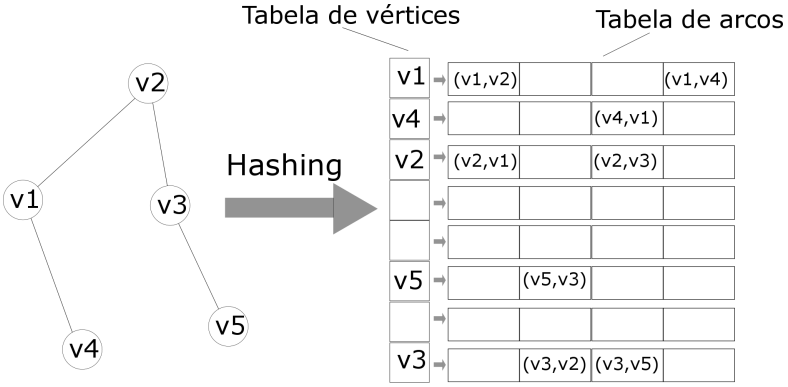

Figura 5 – Representação da tabela hashing

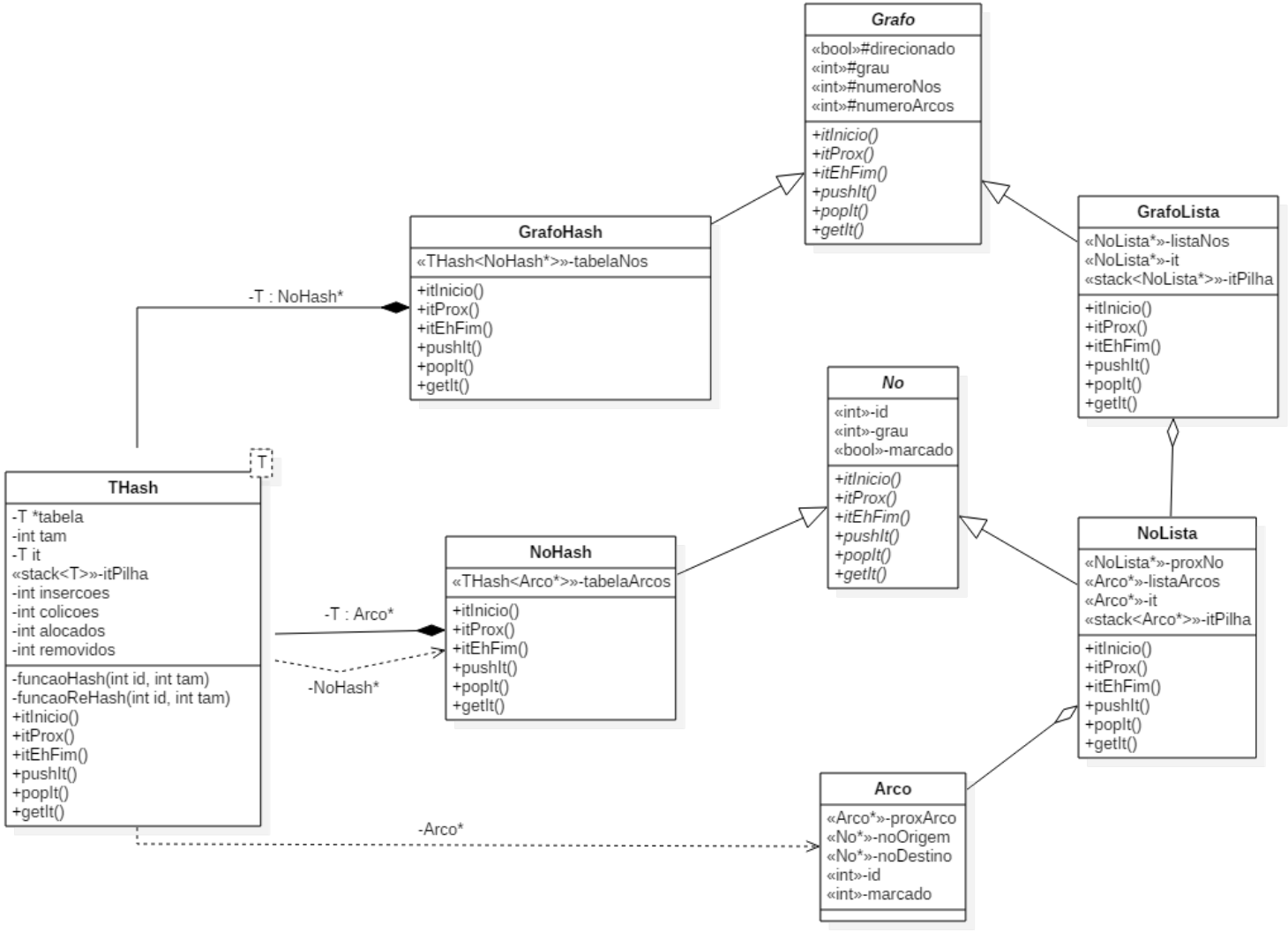

Figura 6 – Diagrama de classes simplificado.

2.3.3 FUNCIONALIDADES

Como parte obrigatória da primeira parte deste trabalho, a classe `Grafo` possui as seguintes funcionalidades implementadas:

- Inserir e excluir vértice e arco;
- Obter grau do grafo;
- Obter grau de um vértice;
- Obter sequencia de grau;
- Verificar se é k-regular;
- Verificar se é completo;
- Verificar se dois nos são adjacentes;
- Percurso em profundidade;
- Percurso em largura;
- Verificar se é conexo;
- Verificar se dois vértices estão na mesma componente conexa;
- Verificar se um vértice é articulação;
- Verificar se um arco é ponte;
- Obter vizinhança de um vértice;
- Obter fechamento transitivo;
- Obter ordenação topológica;
- Obter caminho minimo entre dois vértices;
- Algoritmo de Djikstra;
- Algoritmo de Floyd;
- Obter subgrafo induzido por conjunto de vértices;
- Listar componentes conexas;
- Efetuar produto cartesiano com outro grafo;
- Algoritmo de Prim;
- Algoritmo de Kruskal;
- Verificar se é k-conexo;
- Verificar se é euleriano.

As funcionalidades listadas acima podem ser testadas pelo usuário através da instanciação da classe `MenuTrabalho`.

3 ABORDAGENS ALGORÍTMICAS USADAS NA SOLUÇÃO

O problema da Árvore de Steiner se enquadra na categoria de problemas *NP*, sendo ele um problema *NP-Completo*. Para esses problemas, não se conhece algoritmo de custo polinomial para a solução, ou seja, o custo para se encontrar a solução exata possui, muitas vezes, proporção fatorial em relação à entrada. Para o problema da Árvore de Steiner, assim como os demais problemas da classe *NP*, é necessário o uso de abordagens heurísticas para encontrar soluções aproximadas com custo computacional viável (ordem de complexidade de tempo polinomial).

Foi proposto para este trabalho uma abordagem heurística gulosa construtiva para solução do problema. A abordagem gulosa construtiva supõe um conjunto solução para o problema, um conjunto de candidatos à solução e uma função de ranqueamento dos candidatos, denominada função critério. Por essa abordagem temos o melhor candidato em cada iteração inserido na solução, o conjunto de candidatos é atualizados o mesmo é novamente ranqueados (ordenados) de acordo com a função critério. O algoritmo então constrói a solução com base em uma visão local, analisando sempre e apenas os melhores candidatos na iteração atual, por isso o nome “guloso” ou “míope” para essa abordagem.

3.1 ALGORITMO GULOSO

A primeira abordagem algorítmica utilizada foi a implementação do Algoritmo Guloso Construtivo para o problema. Essa implementação foi dividida em duas etapas: a construção e a poda.

3.1.1 **Etapa de construção**

Na construção, as arestas candidatas a constituir a solução são, inicialmente, aquelas que incidem em nós terminais. A cada iteração, as candidatas são ordenadas respeitando a função critério e coloca-se na solução a melhor aresta candidata. Se um novo nó tiver sido conectado pela aresta que acabou de entrar na solução, são adicionadas às candidatas as arestas adjacentes a este nó. Essa etapa do algoritmo termina quando todos os nós terminais estiverem na mesma componente conexa.

3.1.1.1 Função Critério

Os algoritmos utilizados para a solução do problema têm como resultado a soma dos pesos do conjunto solução de arestas que compõem a árvore, que é o conjunto formado pelas arestas que compõem a Árvore de Steiner. Partindo da heurística construtiva descrita acima e tendo em conhecimento de que o problema visa, também, uma árvore de peso mínimo a função critério deve priorizar as arestas de menor peso. Com isso, obtemos o primeira parcela da função critério (3.4).

Objetivando também conectar todos os terminais o quanto antes, a segunda parcela da função critério soma um *bônus* associado ao potencial que o nó destino deste arco possui de ser caminho para um nó terminal. O potencial de um nó é definido em dois estágios no início da execução do algoritmo. No primeiro estágio, o potencial de cada nó é definido como `0` se não for nó terminal e `1` se for nó terminal. No segundo estágio, o potencial é atualizado como o somatório de seu potencial com os potenciais que os vizinhos possuíam antes da atualização. Esta atualização é a feita n vezes para que não haja muitos nós com potencial nulo e, assim, o acréscimo do bônus seja efetivo.

$$n = arredondarParaBaixo\left(\frac{numeroNos}{numeroTerminais}\right) \tag{3.1}$$

A Figuras 7 e 8 exemplificam o pré-processamento do potencial dos vértices. Em azul temos os nós terminais.

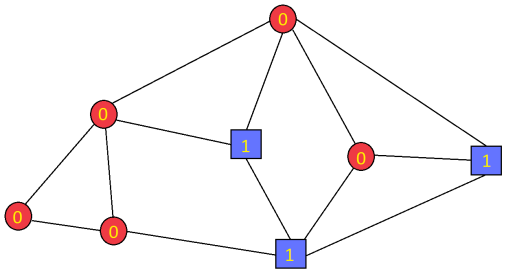

Figura 7 – Inicialização dos potenciais dos nós de um grafo (estágio 1)

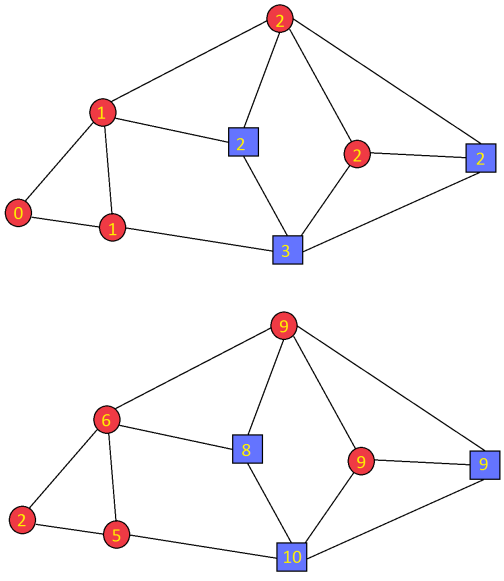

Figura 8 – Inicialização dos potenciais dos nós de um grafo (estágio 2 - $n = 2$).

Contudo, à medida que cada vez mais vértices já estão contidos na solução, o acréscimo de bônus devido ao potencial dos vértices torna-se defasado, já que de nada serve o potencial de levar a vértices que já foram alcançados. Por isso, o *bônus* deve decair à medida que a solução cresce, sendo assim multiplicado pelo fator de decaimento γ. Este decaimento pode ser mais ou menos brusco de acordo com o valor atribuído à variável λ.

Com isso, resultado parcial do *bônus* é um valor entre 0 e 1, portanto o multiplicamos por um fator ρ, que ajusta a grandeza do *bônus*.

$$\gamma = \cdot \left(1 - \frac{numeroNosSolucao}{numeroNosGrafo}\right)^{\lambda} \tag{3.2}$$

$$bônus(aresta) = \rho \cdot \gamma \cdot \left(\frac{potencialNoDestino(aresta)}{maiorPotencialGrafo}\right) \tag{3.3}$$

Obtemos, enfim, a função critério dada pela Equação 3.4 Os candidatos ao conjunto solução são arestas ranqueadas de acordo com a função critério, dada pela Equação 3.4.

$$F(aresta) = \frac{MédiaDosPesos}{Peso(aresta)} + bônus(aresta) \tag{3.4}$$

Após alguns testes preliminares, foram adotados os valores de $\lambda = 2$ e $\rho = 4$. Vale ressaltar que o valor γ não é constante.

3.1.2 **Etapa de poda**

A segunda etapa do algoritmo é necessária para remover da solução ramificações da árvore que não ligam nós terminais e são desnecessárias para a solução. Na poda, procura-se remover as arestas que conectam nós que não são terminais e possuem grau 1. Este processo é repetido até que não existam mais ramificações desnecessárias na árvore. As figuras 10 e 11 ilustram a etapa de poda para o exemplo mostrado na figura 9.

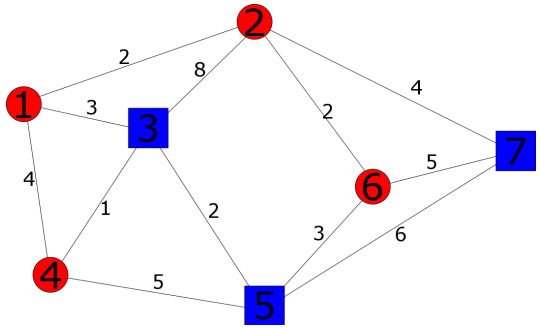

Figura 9 – Grafo exemplo.

O Algoritmo 1 descreve o pseudocódigo da heurística descrito acima.

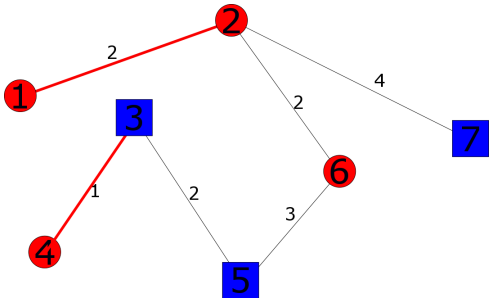

Figura 10 – Solução do algoritmo guloso para o grafo exemplo antes da poda.

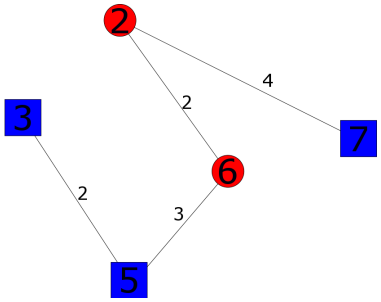

Figura 11 – Solução do algoritmo guloso para o grafo exemplo após a poda.

Algoritmo 1: ALGORITMO GULOSO

Entrada: Grafo **G=(V,E)**, Subconjunto de **V** com os *nós Terminais*

Saída: Soma dos pesos do conjunto de arestas da Árvore de Steiner

```
1  início
2      soluçãoArestas ← ∅;
3      //Parte de Construtiva:
4      soluçãoNós ← nósTerminais;
5      candidatos ← Arestas adjacentes aos nós terminais;
6      enquanto nós terminais não estiverem na mesma componente conexa faça
7          ordena candidatos de acordo com a função critério;
8          soluçãoArestas ← soluçãoArestas ∪ Melhor candidato;
9          se nó destino do melhor candidato não está na solução de nós então
10             soluçãoNós ← soluçãoNós ∪ nó destino do melhor candidato;
11             candidatos ← candidatos ∪ Arestas adjacentes ao nó que foi adicionado à
                soluçãoNós;
12         fim
13         remove o melhor candidato dos candidatos;
14     fim
15     //Parte de Poda:
16     ordena arestas da solução em ordem crescente de acordo com o menor grau de
        seus nós e colocando em último arestas que ligam nós terminais que têm grau 1;
17     enquanto topo da solução é um galho que pode ser removido faça
18         remove galho;
19         reordena solução;
20     fim
21     retorna somaDosPesos(soluçãoArestas)
22 fim
```

3.2 ALGORITMO GULOSO RANDOMIZADO

O Algoritmo Guloso Randomizado é a execução do algoritmo guloso passando como parâmetro uma constante α e um número de iterações. A constante α seleciona a faixa dos melhores candidatos dentre os quais algum será inserido na solução de forma randômica. Por exemplo, para um $\alpha = 0.5$, será escolhido de forma randômica um candidato dentre os 50% melhores classificados a entrar na solução. O algoritmo é iterado um determinado número vezes e retorna o melhor resultado obtido. Cabe ressaltar que o algoritmo guloso foi implementado usando o algoritmo guloso randomizado, com parâmetro $\alpha = 0$. O Algoritmo 2 descreve o funcionamento da Heurística Gulosa Randomizada.

3.3 ALGORITMO GULOSO RANDOMIZADO REATIVO

O Algoritmo Guloso Randomizado Reativo recebe como parâmetro um vetor `alphas`, um número total de iterações e o número de iterações por bloco. O vetor `alphas` contém diversos valores de α a serem testadas pelo algoritmo guloso randomizado. A cada bloco de iterações, o método de escolha de um valor α é modificado, de forma a se adaptar às melhores soluções obtidas.

A abordagem reativa consiste em associar ao vetor `alphas` uma distribuição de probabilidade que indicará os valores de tem mais chances de serem utilizados para encontrar a melhor solução do algoritmo randomizado. A cada bloco de iterações, a distribuição de probabilidade de escolha de cada α é calculada a partir da soma e padronização das soluções encontradas para cada α_i em `alphas`, conforme a Equação 3.5 [1]. O fator δ determina o quanto a distribuição de probabilidade será alterada a cada bloco de iterações. Quando maior o valor de δ, mais os resultados do ultimo bloco influenciarão na próxima distribuição. O pseudocódigo é descrito no Algoritmo 4.

$$q_i = \left(\frac{melhorSoluçãoObtida}{SomaSolucoes(\alpha_i)} \right)^{\delta} \tag{3.5}$$

O Algoritmo 4 foi implementado com um vetor `alphas` de tamanho 10 (`alphas` = $\{5\%, 10\%, 15\%, ..., 50\%\}$), blocos de 100 iterações e um total de 1000 iterações. Para o primeiro bloco de iterações, todos os $\alpha_i \in$ `alphas` foram utilizados igualmente.

3.4 ALGORITMO GULOSO RANDOMIZADO ADAPTADO

Outra abordagem implementada a partir do Algoritmo Guloso Randomizado consiste em, após cada bloco de iterações, atualizar o vetor `alphas` com valores de α que sejam próximos aos melhores resultados obtidos nas iterações do bloco, ou seja, é feita um refinamento dos melhores valores de α encontrados. Esse refinamento é feito selecionando

Algoritmo 2: Algoritmo Guloso Randomizado

Entrada: Grafo **G=(V,E)**, Subconjunto de **V** com os *nós Terminais*, parâmetro α para randomização, número de iterações
Saída: Menor soma dos pesos do conjunto de arestas da Árvore de Steiner

```
1  início
2      menorSomaDePesos ← ∞;
3      para cada iteração faça
4          soluçãoArestas ← ∅;
5          //Parte de Construção:
6          soluçãoNós ← nósTerminais;
7          candidatos ← Arestas adjacentes aos nós terminais;
8          enquanto nós terminais não estiverem na mesma componente conexa faça
9              ordena candidatos de acordo com a função critério;
10             soluçãoArestas ← soluçãoArestas ∪ Escolha randômica entre
                α × G− > numeroNos dos melhores candidatos;
11             se nó destino da aresta escolhida não está na solução de nós então
12                 soluçãoNós ← soluçãoNós ∪ nó destino da aresta escolhida;
13                 candidatos ← candidatos ∪ Arestas adjacentes ao nó que foi
                    adicionado à soluçãoNós;
14             fim
15             remove a aresta escolhida dos candidatos;
16         fim
17         //Parte de Poda:
18         ordena arestas da solução em ordem crescente de acordo com o menor grau
            de seus nós e colocando em último arestas que ligam nós terminais que têm
            grau 1;
19         enquanto topo da solução é um galho que pode ser removido faça
20             remove galho;
21             reordena solução;
22         fim
23         se menorSomaDePesos > somaDePesos(soluçãoArestas) então
24             menorSomaDePesos ← somaDePesos(soluçãoArestas);
25         fim
26     fim
27     retorna menorSomaDePesos
28 fim
```

os melhores valores de α a cada bloco de iterações (usando o mesmo critério de α_i com menor q_i - 3.5) e seus respectivos vizinhos. O pseudocódigo é descrito no Algoritmo 4.

O Algoritmo 4 foi implementado com um vetor `alphas` de tamanho 12 (inicialmente como `alphas` $= \{5\%, 10\%, 15\%, ..., 60\%\}$), blocos de 120 iterações e um total de 1200 iterações.

Algoritmo 3: Algoritmo Guloso Randomizado Reativo

Entrada: Grafo **G=(V,E)**, Subconjunto de **V** com os *nós Terminais*, vetor de parâmetros *alphas* para randomização, número de iterações, tamanho do bloco de iterações

Saída: Menor soma dos pesos do conjunto de arestas da Árvore de Steiner

```
1  início
2      δ ← 1.0;
3      melhorSolução ← ∞;
4      somaSoluções_i ← 0, para cada α_i em alphas;
5      probabilidades ← distribuição uniforme;
6      para cada iteração faça
7          α ← Escolha aleatória de α_i em alphas com base em probabilidades
8          solução ← algoritmoGulosoRandomizado(G, V, α, número de iterações
             para o guloso)
9          se solução > melhorSolução então
10             melhorSolução ← solução;
11         fim
12         somaSoluções_i ← somaSolucoes_i + solução
13         se mod(iteração, tamanho do bloco de iteracoes então
14             somaSoluções_i ← (somaSoluções_i)^δ, para cada i;
15             probabilidades ← atualizaProbabilidades(somaSoluções)
16         fim
17     fim
18     retorna melhorSolução
19 fim
```

Algoritmo 4: Algoritmo Guloso Randomizado Adaptado

Entrada: Grafo **G=(V,E)**, Subconjunto de **V** com os *nós Terminais*, numero de alphas para randomização, valor *prec* de intervalo entre alphas, número de iterações, tamanho do bloco de iterações

Saída: Menor soma dos pesos do conjunto de arestas da Árvore de Steiner

```
1  início
2      melhorSolução ← ∞;
3      somaSoluções_i ← 0, para cada α_i em alphas;
4      para cada α_i, i = 1, ..., numero de alphas faça
5          α_i ← i · prec;
6      fim
7      para cada iteração faça
8          α ← alphas[mod(iteração, quantidade de alphas)]
9          solução ← algoritmoGulosoRandomizado(G, V, α, número de iterações
             para o guloso)
10         se solução > melhorSolução então
11             melhorSolução ← solução;
12         fim
13         somaSoluções_i ← somaSolucoes_i + solução
14         se mod(iteração, tamanho do bloco de iteracoes então
15             alphas ← 1/3 dos melhores alphas
16             melhoresVizinhos ← ∅ prec ← prec/4 para cada alpha_i em alphas
                 faça
17                 melhoresVizinhos ←
                     melhoresVizinhos ∪ {(alpha_i − prec), (alpha_i + prec)}
18             fim
19             alphas ← alphas ∪ melhoresVizinhos
20             somaSoluções_i ← 0, para cada α_i em alphas;
21         fim
22     fim
23     retorna melhorSolução
24 fim
```

4 RESULTADOS

Para gerar os resultados foram usadas instâncias da *SteinLib Testdata* [2], que disponibiliza várias instâncias bem como a melhor solução conhecida para cada uma delas. Para o problema da árvore de Steiner, a solução é o valor dado pela da soma dos pesos das arestas que compõem a árvore.

Dentre as instâncias foram escolhidas as 3 maiores, 3 intermediárias e 4 instâncias pequenas. Alguns dados relativos à cada instância estão na Tabela 1.

Nome da instância	Número de nós	Número de arestas	Pesos das arestas	Número de nós terminais
cc6-2p	64	192	entre 100 e 210	12
cc3-4u	64	288	entre 1 e 2	8
cc3-4p	64	288	entre 100 e 210	8
hc6p	64	192	entre 100 e 110	32
cc10-2u	1024	5120	entre 1 e 2	135
hc10u	1024	5120	igual a 1	512
hc10p	1024	5120	entre 100 e 110	512
hc12p	4096	24576	entre 100 e 110	2048
cc12-2p	4096	24574	entre 100 e 210	473
cc12-2u	4096	24574	entre 1 e 2	473

Tabela 1 – Dados referentes a cada instância.

Todos os algoritmos foram executados 30 vezes para o registro da solução e tempo de execução. Nos testes do algoritmo guloso randomizado adotou-se o valor $\alpha = 25\%$.

A Tabela 2 mostra, para cada instância, a melhor solução obtida dentre todas as execuções de todas as abordagens e o desvio percentual da solução média de cada abordagem em relação à esta.

Nome da instância	Melhor solução obtida	Desvio percentual da solução média em relação à melhor solução obtida			
		Guloso	Guloso Randomizado	Guloso Randomizado Adaptado	Guloso Randomizado Reativo
cc6-2p	3285	12.938%	6.344%	2.544%	1.532%
cc3-4u	23	26.087%	8.696%	2.899%	3.043%
cc3-4p	2349	22.095%	11.679%	3.119%	3.472%
hc6p	4629	16.224%	5.651%	3.155%	3.046%
cc10-2u	443	1.580%	7.028%	2.378%	0.858%
hc10u	803	4.608%	2.283%	1.042%	0.693%
hc10p	81943	2.385%	1.303%	0.775%	0.317%
hc12p	331212	1.130%	0.638%	0.473%	0.235%
cc12-2p	166456	3.305%	6.568%	1.276%	0.511%
cc12-2u	1579	2.153%	8.472%	1.305%	0.477%

Tabela 2 – Comparação das abordagens com base nos melhores resultados obtidos.

Já a Tabela 3 mostra, para cada instância, a melhor solução presente no *benchmark* [2] e o desvio percentual da solução média de cada abordagem em relação à esta.

Nome da instância	Melhor solução no *benchmark*	Desvio percentual da solução média em relação à melhor solução no *benchmark*			
		Guloso	Guloso Randomizado	Guloso Randomizado Adaptado	Guloso Randomizado Reativo
cc6-2p	3171	16.998%	10.167%	6.230%	5.182%
cc3-4u	23	26.087%	8.696%	2.899%	3.043%
cc3-4p	2338	22.669%	12.204%	3.604%	3.959%
hc6p	4003	34.399%	22.173%	19.286%	19.161%
cc10-2u	342	31.579%	38.635%	32.612%	30.643%
hc10u	575	46.087%	42.841%	41.107%	40.620%
hc10p	59797	40.303%	38.821%	38.097%	37.470%
hc12p	236949	41.361%	40.674%	40.444%	40.110%
cc12-2p	121106	41.989%	46.474%	39.200%	38.149%
cc12-2u	1172	37.628%	46.141%	36.485%	35.370%

Tabela 3 – Comparação das abordagens com base nos melhores resultados do *benchmark* [2].

E, por fim a Tabela 4 mostra o desempenho obtido para máquina com processador core i7-2600 com $3.4GHz$.

Nome da instância	Tempo médio de execução (s)			
	Guloso	Guloso Randomizado	Guloso Randomizado Adaptado	Guloso Randomizado Reativo
cc6-2p	1.60E-03	2.18E-02	6.12E-01	5.77E-01
cc3-4u	1.23E-03	2.56E-02	7.77E-01	7.76E-01
cc3-4p	1.10E-03	2.50E-02	7.75E-01	7.78E-01
hc6p	8.00E-04	1.95E-02	6.99E-01	6.20E-01
cc10-2u	3.71E-01	1.02E+01	4.27E+02	3.81E+02
hc10u	4.65E-01	1.38E+01	5.46E+02	4.51E+02
hc10p	4.14E-01	1.47E+01	7.17E+02	5.85E+02
hc12p	1.67E+01	5.35E+02	1.97E+04	1.76E+04
cc12-2p	8.21E+00	2.36E+02	8.77E+03	7.75E+03
cc12-2u	8.29E+00	2.40E+02	9.24E+03	8.10E+03

Tabela 4 – Tempo médio de execução das abordagens.

5 CONCLUSÃO

Pode-se concluir, a partir dos testes, que a heurística gulosa construtiva proposta, por si só, não é uma boa alternativa para obter a solução do problema proposto, sendo necessário o uso da abordagem randomizada e reativa para melhorar o resultado obtido pelo algoritmo guloso.

A heurística proposta consegue obter resultados relativamente próximos da melhor solução para algumas instâncias, porém também obtém resultados relativamente ruins para outras. A nova abordagem proposta, o Algoritmo Guloso Randomizado Adaptado, mostrou-se melhor que o Algoritmo Guloso e o Algoritmo Guloso Randomizado, por ter flexibilidade na escolha de α, mas foi inferior ao Algoritmo Guloso Randomizado Reativo.

O tempo computacional é um fator que pesa muito na execução das abordagens randomizada adaptada e reativa, devido ao fato das mesmas necessitarem de várias execuções do algoritmo guloso randomizado. Para o teste das maiores instâncias, várias horas são demandadas para executar cada abordagem randomizada adaptada ou reativa.

Não houve ganho de desempenho da estrutura com tabela *hash* de adjacências em relação à estrutura de lista encadeada de adjacência, como pode-se comprovar observando os resultados no Anexo A, provavelmente por error na implementação da estrutura. Todos os resultados obtidos no capítulo 4 foram obtidos usando a estrutura de lista encadeada de adjacências. Porém, não houve significativa influência o desempenho da solução do problema. Isso se deve, principalmente, por que a heurística usada para o problema da Árvore de Steiner não faz uso de muitas buscas, com as quais o uso *hashing* tenderia a ser mais eficiente em termos de desempenho.

REFERÊNCIAS

[1] PRAIS, Marcelo; RIBEIRO, Celso C. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. **INFORMS Journal on Computing**, v. 12, n. 3, p. 164-176, 2000.

[2] SteinLib Testdata Collection. SteinLib Testdata Library. Disponível em: <http://steinlib.zib.de>. Acesso em: 4 dez. 2016.

[3] Wikipédia. Tabela de dispersão. Disponível em: <https://pt.wikipedia.org/wiki/Tabela_de_dispersão>. Acesso em: 10 dez. 2016.

ANEXO A – Comparação do desempenho entre os Grafos com estruturas de tabela *hashing* e lista encadeada

Numero de Nos	Teste Busca		Teste Inserção		Teste Remoção	
	Hash	**Lista**	**Hash**	**Lista**	**Hash**	**Lista**
40	2.313E-05	4.133E-06	8.925E-07	6.333E-08	2.423E-05	3.333E-06
80	4.597E-05	8.767E-06	6.883E-07	6.750E-08	4.670E-05	5.167E-06
120	1.117E-04	1.127E-05	9.078E-07	7.944E-08	1.141E-04	9.733E-06
160	2.598E-04	1.367E-05	1.178E-06	6.646E-08	2.662E-04	1.410E-05
200	2.949E-04	1.310E-05	1.503E-06	6.033E-08	3.118E-04	1.593E-05
240	3.281E-04	1.640E-05	1.387E-06	5.931E-08	3.379E-04	1.580E-05
280	5.679E-04	1.570E-05	2.038E-06	4.405E-08	5.962E-04	2.173E-05
320	5.454E-04	1.503E-05	1.780E-06	5.250E-08	5.459E-04	1.567E-05
360	6.665E-04	1.673E-05	1.901E-06	7.074E-08	6.704E-04	1.707E-05
400	1.267E-03	2.123E-05	3.039E-06	6.567E-08	1.218E-03	2.007E-05
440	1.089E-03	2.047E-05	2.485E-06	6.712E-08	1.088E-03	2.043E-05
480	1.722E-03	2.997E-05	3.623E-06	6.639E-08	1.730E-03	3.180E-05
520	2.183E-03	3.520E-05	4.182E-06	5.609E-08	2.129E-03	3.520E-05
560	2.425E-03	3.863E-05	4.331E-06	6.196E-08	2.419E-03	3.600E-05
600	1.818E-03	2.567E-05	3.081E-06	4.767E-08	1.823E-03	2.873E-05
640	2.057E-03	2.723E-05	3.318E-06	6.115E-08	2.067E-03	2.870E-05
680	2.299E-03	2.943E-05	3.382E-06	4.461E-08	2.305E-03	3.040E-05
720	2.709E-03	3.093E-05	3.768E-06	6.060E-08	2.714E-03	3.217E-05
760	4.680E-03	3.287E-05	6.143E-06	4.425E-08	4.669E-03	3.823E-05
800	5.229E-03	3.450E-05	6.523E-06	6.238E-08	5.229E-03	3.573E-05
840	3.484E-03	4.327E-05	4.147E-06	4.389E-08	3.485E-03	4.633E-05
880	6.298E-03	4.043E-05	7.163E-06	4.693E-08	6.302E-03	3.843E-05
920	6.812E-03	4.273E-05	7.412E-06	4.580E-08	6.814E-03	4.037E-05
960	4.542E-03	5.093E-05	4.728E-06	4.830E-08	4.550E-03	5.837E-05

Tabela 5 – Desempenho das estruturas de Grafo (tempo em segundos).