Calvin Rodrigues da Costa

Problema da árvore de Steiner em Grafos para melhor tratar o roteamento *multicast*

São José dos Campos, SP

Calvin Rodrigues da Costa

Problema da árvore de Steiner em Grafos para melhor tratar o roteamento *multicast*

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência de Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof[a]. Dr[a]. Mariá Cristina Vasconcelos Nascimento Rosset

São José dos Campos, SP

Novembro de 2014

Calvin Rodrigues da Costa

Problema da árvore de Steiner em Grafos para melhor tratar o roteamento *multicast*

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em 28 de novembro de 2014:

Profª. Drª. Mariá Cristina Vasconcelos Nascimento Rosset
Orientadora

Prof. Dr. Álvaro Luiz Fazenda
Convidado 1

Prof. Dr. Valerio Rosset
Convidado 2

Coordenador de TCC II

São José dos Campos, SP
0 Novembro de 2014

Dedico este trabalho a minha família pelo apoio.
A meus professores, que nesses anos transmitiram tantos conhecimentos. Em especial, à Prof^{a}.
Dr^{a}. Mariá que me orientou com muito afinco e paciência neste projeto.
Aos meus amigos pela convivência e apoio durante o curso.
Por fim, a Deus por me permitir o presente de conhecer a todos.

“There’s no such thing as a painless lesson. They just don’t exist.
Sacrifices are necessary. You can’t gain anything without losing something first.
Although, if you can endure that pain and walk away from it, you’ll find you now have a heart
strong enough to overcome any obstacle - a heart made fullmetal.”
(Edward Elric - *FullMetal Alchemist*)

Resumo

Com o aumento do uso e do desenvolvimento de aplicações focadas na comunicação por meio de *multicast*, a necessidade por algoritmos mais eficientes para organizar o roteamento de dados em uma rede de computadores torna-se evidente. Muitas das aplicações requisitam uma determinada qualidade de serviço (QoS) acerca de determinados problemas como, por exemplo, o de tratamento de atraso de pacotes. Considerando que, em uma rede de computadores, é possível a existência de inúmeros dispositivos inclusos a ela, os algoritmos de roteamento devem ser de baixa complexidade. A baixa complexidade em um algoritmo de roteamento permite que, sem requisitar uma taxa de processamento ou tempo de execução muito elevados, obter em redes de computadores com muitos dispositivos uma estrutura de roteamento tal que reduza os efeitos do tráfego de grandes volumes de informação na rede. Em particular, o conceito do problema da árvore de Steiner em grafos é usado para a organização de diversos algoritmos de roteamento. Tendo isso em mente, neste trabalho, é proposto o uso do STPG, abordado pela meta-heurística GRASP, com intuito de melhor tratar casos nos quais há uma grande quantidade nós na rede. Esperamos, como resultado deste trabalho, um algoritmo de roteamento capaz de produzir uma árvore de Steiner a baixo custo, porém, com uma boa qualidade. Para avaliar a qualidade da meta-heurística proposta, pretendíamos realizar diversos experimentos envolvendo dados *benchmark* do problema de Steiner e com estudos de caso encontrados na literatura para roteamento *multicast*. Devido a um problema no código que não conseguiu ser resolvido até o presente momento, não foi possível avaliar a eficiência computacional da proposta. Entretanto, diversas inferências a respeito da corretude do algoritmo são apresentadas neste texto.

Palavras-chaves: *multicast*, Problema da Árvore de Steiner em Grafos, GRASP.

Abstract

With the rise of use and development on applications with focus on communications through multicast, the need for algorithms more efficient to organize the routing of data in a computer network becomes evident. Many applications require a determined quality of services (QoS) about determined problems like, for example, the packet delay treatment. Given that, in a computer network, it is possible the existence of many devices included to it, the routing algorithms must be of low complexity. The low complexity in a routing algorithm allows, without requiring a very high processing rate or execution time, get in computer networks with many devices a routing structure which reduces the effects of large volumes of information traffic on the network. In particular, the Steiner tree problem concept in graph is used to organize several routing algorithms. With this in mind, in this paper, it is proposed the use of STPG, approached by meta-heuristic GRASP, with the goal of better treat cases in which there is a lot nodes in the network. Is expected, as a result of this research, a routing algorithm capable of producing a Steiner tree at low cost, however with a good quality. To evaluate the quality of the proposed meta-heuristic, it is intended to realize several experiments involving benchmark data's of the Steiner problem and with case studies found in literature of multicast routing. Due to a problem in the code that could not be solved so far, was not possible to evaluate the computational efficiency of the goal. However, inferences about the correctness of the algorithm are presented in this paper.

Key-words: Multicast, Steiner tree problem in graphs, GRASP.

Lista de Figuras

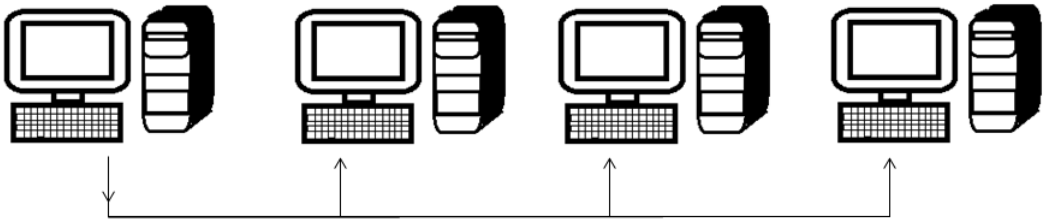
Lista de abreviaturas e siglas

CBT	Árvore Centralizada
CSTPG	Problema da Árvore de Steiner em Grafos Restrita
DCSP	*Delay-constrained Shortest Path*
DCSTPG	Problema da Árvore de Steiner em Grafos Restrita por Atraso
DMCT	Heurística *Delay-constrained Minimum Steiner Tree*
DSTPG	Problema da Árvore de Steiner em Grafos Dinâmica
GA	Algoritmo Genético
GRASP	*Greedy Randomized Adaptive Search Procedure*
KMB	Algoritmo Kou-Markowsky-Berman
MST	Árvore Geradora Mínima
PRIM_SG	Algoritmo de Prim adaptado à estratégia semi-gulosa
STG	Árvore de Steiner em Grafos
STPG	Problema da Árvore de Steiner em Grafos
TS	Busca Tabu
W_g	Função peso de um grafo G
a_e	Lista restrita de candidatos, arestas escolhidas
gM	Grupo *multicast*
p_n	Novo vértice pai
v_f	Vértice filho
v_p	Vértice pai

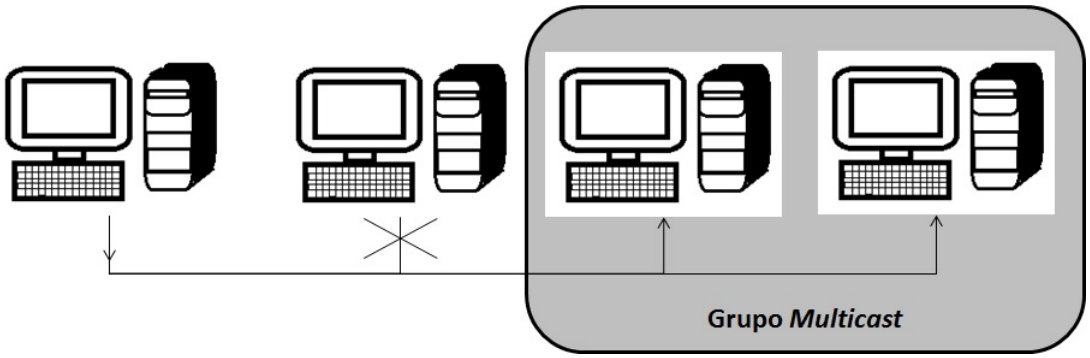
Sumário

1 Introdução

Em redes de computadores, a comunicação efetuada entre os aparelhos pertencentes à rede ocorre com o envio de informações que trafegam por vias existentes para essa transmissão. A metodologia de envio das informações em uma rede de computadores varia de acordo com o objetivo a ser alcançado. Para envios de informação em massa, há os métodos de *broadcast*, por meio dos quais informações são transmitidas a toda a rede como representado na Figura 1(a), e de *multicast*, Figura 1(b), por meio dos quais as transmissões são distribuídas a partir de um ou mais fornecedores para um determinado número de destinos (TANENBAUM, 2011).



(a) *Operação de broadcast*



(b) *Operação de multicast*

Figura 1 – Figuras representando operação de *broadcast* e *multicast*.

Em particular, a operação de *multicast*, aplicada a uma rede de computadores, está sujeita a problemas como perdas ou atrasos de pacotes que, por vezes, estão relacionados ao congestionamento da própria rede. Portanto, para um melhor funcionamento da transmissão de dados, são necessários métodos que atenuem tais problemas. Segundo Oliveira e Pardalos (2005), os atuais focos de melhora para o roteamento *multicast* tratam sobre minimizar o custo total da árvore de roteamento, o atraso de envio e de recebimento de dados e o próprio congestionamento da rede.

Seja um grafo $G = (V(G), E(G))$, que represente uma rede, em que $V(G)$ representa o conjunto de vértices (nós) e $E(G)$, o grupo de arestas. Nesse grafo, um grupo *multicast* é o conjunto de vértices que participa da operação de *multicast* como apresentado na Figura 1(b).

O tamanho de um grupo *multicast* geralmente é inferior ao tamanho da rede na qual o grupo se encontra. Dessa forma, dado um grupo *multicast* X, com $X \subseteq V(G)$, o problema da árvore de Steiner em grafos (STPG) busca uma árvore com peso mínimo $T_G = (V(T_G), E(T_G))$ que contenha todos os nós de X (MACULAN, 1987).

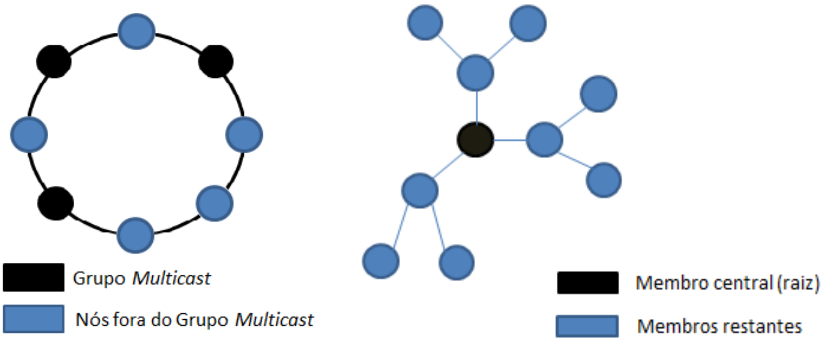
Para lidar com diferentes aplicações, o problema da árvore de Steiner em grafos possui diversas variantes. Dentre elas, podemos citar a árvore de Steiner com coleta de prêmios (BIENSTOCK et al., 1993), a com restrições (BEZENŠEK; ROBIČ, 2014) e a dinâmica (IMASE; MAXMAN, 1991) sendo que, as variações interessantes a este trabalho são: a STPG dinâmica (DSTPG) e a STPG restrita (CSTPG) que são encontradas na literatura, relacionadas ao problema de roteamento *multicast*, em (AHARONI; COHEN, 1998) e (KOMPELLA; PASQUALE; POLYZOS, 1993a; KOMPELLA; PASQUALE; POLYZOS, 1993b; SKORIN-KAPOV; KOS, 2003; SKORIN-KAPOV; KOS, 2006).

Na literatura, diversos algoritmos para encontrar a árvore de Steiner em grafos foram propostos. Entretanto, poucas dessas estratégias são distribuídas, pois se utilizam da premissa do conhecimento do grafo como um todo. Neste trabalho, o problema da árvore de Steiner em grafos restrita será abordado de maneira paralela (baseada em vizinhança) pela meta-heurística *Greedy Randomized Adaptive Search Procedure* (GRASP) (FEO; RESENDE, 1989) para realizar o roteamento *multicast*. A meta-heurística GRASP, em sua forma original, é uma estratégia iterativa composta por duas fases: uma de construção na qual é produzida uma solução inicial, e uma fase de busca local que, a partir da solução inicial, tenta obter uma solução de melhor qualidade segundo a métrica definida. O modelo computacional desenvolvido utiliza o algoritmo de Prim (PRIM, 1957) na fase de construção de soluções modificado para ser semi-guloso e uma estratégia de busca local baseada em troca dos vértices pertencentes à STG.

1.1 Contextualização e Motivação

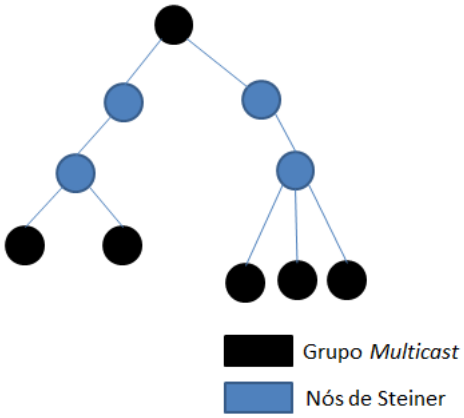
Na literatura, há diversos algoritmos que objetivam criar a estrutura para o roteamento *multicast*. Esses algoritmos utilizam algumas abordagens, dentre as quais destacamos, por serem as mais citadas na literatura (OLIVEIRA; PARDALOS, 2005), as estratégias de roteamento baseadas nos seguintes conceitos: na estrutura de um com grafo de anel (BALDI; OFEK; YENER, 1997), na estrutura de uma árvore centralizada (CBT) (WALL, 1980) e na estrutura conhecida como a árvore de Steiner em grafos (HWANG; RICHARDS; WINTER, 1992).

O roteamento baseado em grafo de anel (Figura 2(a)) segue o princípio de conectar cada membro do grupo *multicast* formando uma estrutura de grafo de anel. Estruturar o grupo *multicast* em forma de anel permite uma razoável margem de confiabilidade já que, caso um único membro se torne inativo, não ocorrerá perda de conexão entre os demais participantes ativos, devido à existência de duas rotas de comunicação entre os nós (OLIVEIRA; PARDALOS,



(a) *Estrutura para roteamento baseada em grafo de anel*

(b) *Estrutura para roteamento baseada em uma árvore centralizada*



(c) *Estrutura para roteamento baseada em uma árvore de Steiner*

Figura 2 – Exemplos de três tipos de estruturas altamente utilizadas por algoritmos de roteamento da literatura.

2005). Apesar disso, o uso de apenas um canal de comunicação[1] para todo o grupo *multicast*, como ocorre no roteamento baseado em grafo de anel, favorece o congestionamento da rede.

A estratégia de roteamento baseada na árvore centralizada (CBT) organiza os participantes do grupo *multicast* escolhendo um de seus membros para ser o centro da árvore[2]. O critério de escolha de quem será o membro central é variável, podendo ser, por exemplo, o nó mais próximo dos demais membros (Figura 2(b)). Assim, a estrutura da CBT tem como um de seus intuitos proporcionar uma boa escalabilidade devido ao uso de apenas uma árvore de roteamento para cada grupo *multicast*. Como consequência, reduz-se o consumo de memória em comparação com abordagens que utilizam uma árvore de roteamento para cada membro de um grupo *multicast* (BALLARDIE; FRANCIS; CROWCROFT, 1993). Assim como ocorre para todo algoritmo de roteamento *multicast* baseado em árvores, o congestionamento da rede afeta negativamente a estrutura de roteamento baseado em CBT, pois seu vértice central representa um gargalo na estrutura, assim como um ponto único de falha.

Por fim, as abordagens de roteamento baseadas na árvore de Steiner em grafos (STPG)

[1] Rota que conecta membros de um grupo *multicast* e que é utilizada para a comunicação.

[2] O membro do grupo *multicast* definido como centro será a raiz para as sub-árvores, A_{sub}, na árvore A.

(Figura 2(c)), alvo de estudo deste TCC, tem como principal característica a composição de uma árvore de peso mínimo a partir de um grupo *multicast*. O problema de estruturar o grupo como uma árvore de peso mínimo é NP-completo (KARP, 1972) e, portanto, para lidar com instâncias de larga escala, diversas heurísticas são encontradas na literatura (OLIVEIRA; PARDALOS, 2005; BEZENŠEK; ROBIČ, 2014).

No roteamento *multicast* com o conceito do problema da árvore de Steiner em grafos, embora sujeito a ficar desconexo, como qualquer estrutura de roteamento *multicast* baseada em árvores, cada membro do grupo pode obter sua própria rota de *multicast*. Então, caso um membro fique inativo, nem todos serão prejudicados e, como as rotas *multicast* de cada membro podem ser diferentes, a ocorrência de congestão na rede tende a ser reduzida. Devido a isso, o STPG configura-se como uma opção eficiente para um roteamento *multicast*.

Diversos trabalhados da literatura abordam o problema da STPG. Do ponto de vista distribuído e paralelo, Bezenšek e Robič (2014) destacam os trabalhos que consideram essa abordagem em seu *survey*. Eles destacam que muito esforço tem sido dedicado para determinar limitantes para o problema, por meio de métodos aproximativos. Como consequência, métodos heurísticos não aproximativos têm recebido menos atenção, sendo que, dos destacados pelos autores, o mais recente é o de 2001. Eles mencionam que esses algoritmos, entretanto, têm um papel importante para diversas heurísticas de busca local para evitar ótimos locais.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste TCC é desenvolver um algoritmo baseado em vizinhanças para encontrar uma árvore de Steiner. Esse objetivo vai de encontro com a aplicação aqui visada, a de roteamento *multicast*, em casos nos quais os nós da rede são distribuídos, como, por exemplo, redes de sensores e atuadores. Tem-se em mente que a NP-completude do STPG limita o bom desempenho de estratégias heurísticas de roteamento que se baseiem em tal estrutura. A imposição de um conhecimento local pode afetar ainda mais a qualidade das soluções encontradas.

1.2.2 Objetivos Específicos

Para atingir os objetivos gerais deste projeto, os seguintes objetivos específicos foram seguidos:

- Adaptação do algoritmo de Prim (algoritmo visto na disciplina de Teoria de Grafos e caracterizada por encontrar a árvore geradora mínima de um grafo conexo) para ser uma heurística construtiva semi-gulosa do STPG;

- Identificação das estruturas de dados adequadas para a implementação da heurística, desenvolvida em linguagem C;
- Obtenção de uma árvore com os nós do grupo *multicast* por meio da heurística construtiva implementada;
- Validação do algoritmo utilizado tanto na fase construtiva quanto na de Busca Local para a manipulação da árvore;
- Refinamento da árvore da fase construtiva por meio de um algoritmo de poda;
- Um segundo refinamento da árvore obtida na fase construtiva por meio de uma estratégia de Busca Local com realocação de vértices;
- Estudo e aplicação de modelos (instâncias) *multicast* para validar a heurística distribuída proposta;
- Validação da eficiência da proposta comparando seus resultados com os de algoritmos estado da arte para aplicações, usando instâncias presentes na biblioteca Steinlib (KOCH; MARTIN; Voß, 2000).

1.3 Organização do Texto

O restante deste TCC está organizado da seguinte forma: na Seção 2, apresentamos a revisão bibliográfica onde apresentamos os principais conceitos do problema da árvore de Steiner em grafos, assim como, as heurísticas, meta-heurísticas com alguns exemplos da aplicação de paralelismo para resolver este problema; a Seção 3 apresenta o método de solução da STPG proposta neste TCC; a Seção 4, têm-se as considerações finais em relação ao trabalho realizado; e, por fim, na Seção 4.1 o que foi aprendido durante o desenvolvimento deste trabalho.

2 Problema da Árvore de Steiner em Grafos

A árvore de Steiner, como já foi dito anteriormente, é aquele subgrafo acíclico de um dado grafo G composto por ao menos o conjunto $V_s \subseteq V$ de vértices e com as arestas cuja soma produza a árvore de peso mínimo com os vértices de V_s. Mais formalmente, podemos definir a árvore de Steiner como sendo aquele grafo induzido por $V_s \cup V_{ns} \subseteq V$, sendo V_{ns} os nós de Steiner escolhidos de forma que:

$$\min G[V_s \cup V_{ns}]$$

sujeito a:

$G[V_s \cup V_{ns}]$ é conexo e acíclico

$V_{ns} \subset V \backslash V_s$

A árvore de Steiner pode, portanto, englobar vértices não pertencentes a V_s, chamados nós de Steiner, que são selecionados apenas caso permitam uma redução no peso total da árvore (MACULAN, 1987).

Como apresentado na Seção 1, o STPG possui diversas variações. A variação do STPG dinâmico opera com ambientes altamente dinâmicos como redes de comunicação utilizando-se, por exemplo, de memória cache para manter as informações necessárias ao seu funcionamento. Esta variante encontra-se em trabalhos como de (AHARONI; COHEN, 1998) para o problema de roteamento *multicast*.

Oliveira e Pardalos (2005) apontam, em seu *survey*, o problema da árvore de Steiner em grafos restrita por atraso (DCSTPG) como um dos problemas mais estudados em roteamento *multicast*, tanto em otimização quanto na aplicação em algoritmos. Relativos a esta variante, são encontrados trabalhos como de Kompella, Pasquale e Polyzos (1993a), Feng e Yum (1999), Bastos e Ribeiro (2002) que a aplicam ao problema de roteamento *multicast*.

Segundo (BEZENŠEK; ROBIČ, 2014), tratando-se do STPG distribuído, três tipos de algoritmos são encontrados:

- Baseados no menor caminho[1]: que usam estratégias baseadas na busca pelo menor caminho e na árvore geradora mínima. Essa estratégia é baseada em vizinhança.
- Baseados na Distância Média: é uma estratégia construtiva aglomerativa que une subárvores de único vértice por meio de um caminho que cruze um nó de Steiner e possua

[1] Um caminho é um grafo simples no qual dois vértices serão adjacentes se forem consecutivos e não-adjacentes, caso contrário.

uma distância média mínima para cada sub-árvore.

- Baseados em distância: que usa uma métrica de distância entre nós para então determinar uma árvore geradora mínima.

Como pôde-se notar, o problema de encontrar árvores geradoras mínimas (MST) e a árvore de Steiner estão relacionados. Por esse motivo, na próxima seção, esse problema é discutido.

2.1 Árvore Geradora Mínima

Segundo Sedgewick (2002), uma árvore geradora mínima (MST) de um grafo G conexo, que possua peso em suas arestas, é uma de suas árvores induzidas e cujo peso total de suas arestas seja mínimo. Para uma MST T', o custo é calculado a partir da somatória dos pesos de todas arestas. Neste TCC, o peso de uma aresta será denotado pela função $W : E(G) \rightarrow \Re$.

$$\min \sum_{e \in E(T')} W(e)$$

sujeito a

$V(T') = V(G)$

$T' \subseteq G$

Esse problema pode ser resolvido em tempo polinomial e os métodos mais conhecidos para esse fim são o algoritmo de *Boruvka-Kruskal* e de *Jarnik-Prim*. Resumidamente, o algoritmo de *Boruvka-Kruskal* parte de um subgrafo gerador vazio e encontra uma sequência de florestas vizinhas, que resulta em uma árvore ótima. Essa sequência é construída adicionando arestas, uma por vez, de modo que a aresta adicionada em cada iteração é aquela com peso mínimo, desde que o subgrafo resultante ainda seja uma floresta. Um pseudocódigo desse método é apresentado no Algoritmo 1.

Algoritmo 1 Algoritmo de Boruvka-Kruskal

ENTRADA: um grafo conexo com peso (G, W);
SAÍDA: uma árvore ótima $T = (V, F)$ de G e sua função peso $W(F)$;
1 Faça $F := \emptyset$ e $W(F) := 0$ (F corresponde ao conjunto de arestas da floresta atual);
2 **Enquanto** houver uma aresta $e \in E \backslash F$ tal que $F \cup \{e\}$ é o conjunto de arestas de uma floresta **faça**
5 Escolha uma aresta e de custo mínimo;
6 Substitua F por $F \cup \{e\}$ e $W(F)$ por $W(F) + W(e)$;
7 **fim enquanto**
8 RETORNE $((V, F), W(F))$;

O algoritmo de Jarnik-Prim, por outro lado, funciona de maneira mais local. Um pseudocódigo desse método é apresentado no Algoritmo 2.

Algoritmo 2 Algoritmo de Jarnik-Prim

ENTRADA: um grafo conexo com peso (G, W);
SAÍDA: uma árvore ótima T de G com função predecessor p, e sua função peso $W(T)$, c : função custo provisionado;

```
1   Faça p(v) := ∅ e c(v) := ∞ ∀ v ∈ V e W(T) := 0;
2   Escolha um vértice r como raiz;
3   Substitua c(r) por 0;
4   Enquanto houver um vértice não colorido faça
5       Escolha um vértice u de custo mínimo c(u);
6       Pinte u;
7       Para cada vértice v não colorido tal que w(uv) < c(v) faça
8           Substitua p(v) por u e c(v) por W(uv);
9       fim para
10      Substitua W(T) por W(T) + c(u);
11  fim enquanto
12  RETORNE (p, W(T));
```

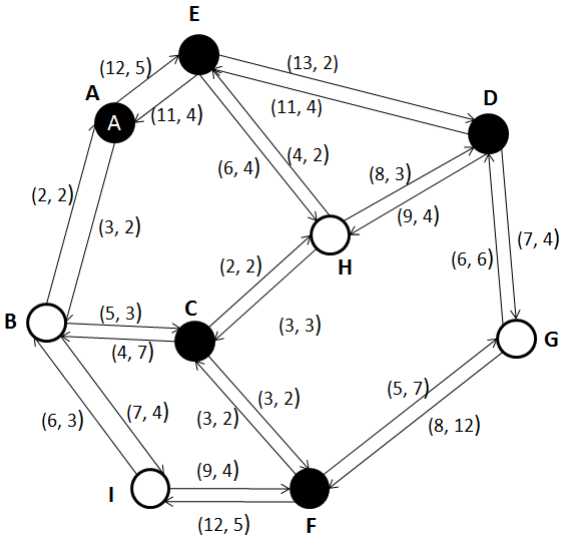
2.2 Heurísticas

Em seu *survey*, Oliveira e Pardalos (2005) referenciam algumas heurísticas para a construção de uma árvore geradora mínima (MST) a fim de encontrar uma solução heurística para o STGP (FENG; YUM, 1999). A heurística de Feng e Yum (1999), Figura 3, aplica algoritmo proposto por Dijkstra (1959) e gerencia os caminhos num intervalo de restrição de atraso específico. Os parâmetros de entrada da heurística são um grafo direcionado $G_d = (V(G_d), E(G_d))$, uma matriz de atrasos $D_{n\times n}$, uma matriz de custos $C_{n\times n}$, a restrição Δ e, para o problema que focado neste TCC, o conjunto V_s como definido na Seção 2. O retorno dessa heurística é uma CSTPG, $T_{final} = (V(T_{final})), E(T_{final}))$, do vértice v'_0 a todos os vértices $v' \in V_s$.

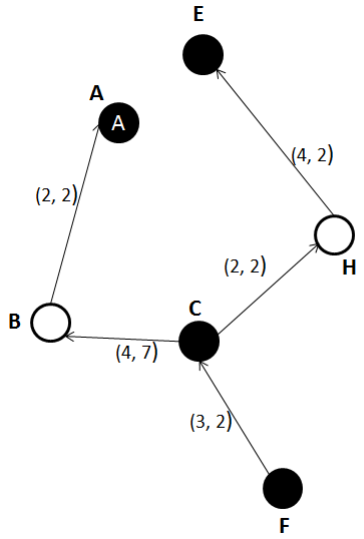
O primeiro passo da heurística é executar o algoritmo chamado, por Feng e Yum (1999), como *Delay-constrained Shortest Path* (DCSP). O DCSP calcula a árvore de caminho com atraso mínimo, $T_1 = (V(T_1), E(T_1))$ usando, como referência, $D_{n\times n}$, $C_{n\times n}$ e Δ (Figura 3(b)). Caso $V_1 = V_s$, então $T_{final} \leftarrow T_1$ e retorna-se T_{final}.

Caso T_{final} não seja obtida durante o primeiro passo, aplica-se o algoritmo de Dijkstra (1959) que, com base em $D_{n\times n}$, calcula a árvore de caminho com atraso mínimo $T_2 = (V(T_2), E(T_2))$ tal que $V_s \backslash V(T_1) \subseteq V(T_2)$ (Figura 3(c)). Ao obter T_1 e T_2, o algoritmo combina-os para conseguir a DCSTP T_{final}.

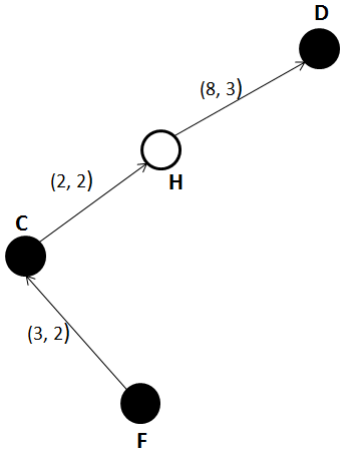
Ao combinar T_1 e T_2 a fim de obter a DCSTP T_{final}, Figura 3(d), pode ser que sejam formados ciclos. Com a intenção de evitar ciclos em T_{final}, a heurística de Feng e Yum (1999) verifica o grau de arcos incidentes a cada nó em T_{final}. Caso um nó possua dois arcos incidentes



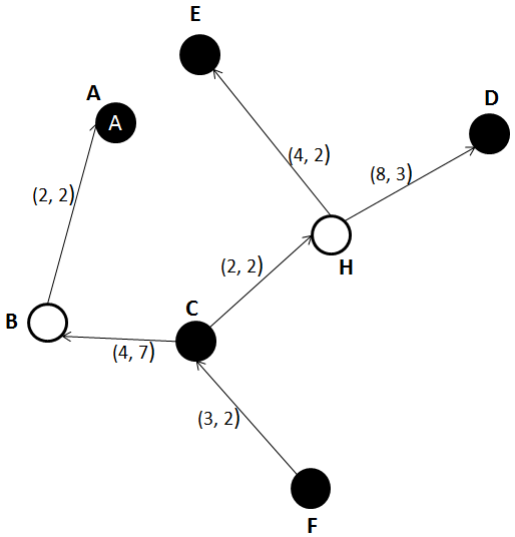
(a) *Exemplo de rede com* $v_0' = F$, $V_s = \{A, E, C, D\}$ *e* $\Delta = 12$ *(FENG; YUM, 1999)*



(b) *Árvore* T_1 *obtida utilizando o algoritmo DCSP. (FENG; YUM, 1999)*



(c) *Árvore* T_2 *obtida utilizando o algoritmo de Dijkstra (1959).(FENG; YUM, 1999)*



(d) *Ávore* T_{Final} *gerada pela combinação de* T_1 *e* T_2*. (FENG; YUM, 1999)*

Figura 3 – Imagem adaptada do exemplo de Feng e Yum (1999).

a ele, considera-se a existência de um ciclo e, dos dois arcos incidentes ao nó, será mantido o que fornecer o menor custo e que cumprir as restrições impostas. Para maiores detalhes sobre esta operação, incluindo a eliminação do arco que não for escolhido, Feng e Yum (1999) apresentam em seu trabalho um exemplo no qual este problema é tratado.

Uma das vantagens do algoritmo proposto por Feng e Yum (1999) é o seu custo polinomial. Embora com tempo polinomial, assim como o algoritmo proposto em (KOU; MARKOWSKY; BERMAN, 1981), o algoritmo de Feng e Yum (1999) segue uma estratégia gulosa e, em consequência, baseia-se na escolha de ótimos locais a fim de tentar obter um ótimo global.

Oliveira e Pardalos (2005) citam estudos de Doar e Leslie (1993) sobre a heurística proposta por Kou, Markowsky e Berman (1981) em redes reais. O algoritmo Kou-Markowsky-Berman (KMB), proposto por Kou, Markowsky e Berman (1981), aplica a heurística de Rede de Distância, de complexidade $O(|S||V(G)|^2)$, com $S \in V(G)$ e S como conjunto dos nós de Steiner e segue os seguintes passos:

1. Produzir um grafo completo G_1 (Figura 4(b)) a partir de G (Figura 4(a)) e S ;
2. Encontrar a Árvore geradora mínima (MST) T_1 de G_1 (Figura 4(c));
3. Obter o subgrafo G_s de G com a substituição de cada aresta pertencente a T_1 por seu respectivo menor caminho (Figura 4(d));
4. Encontrar a MST T_s de G_s (Figura 4(e));
5. Construir a árvore de Steiner T_h de T_s por meio da remoção de arestas, quando necessário, para que todos os vértices folha sejam nós de Steiner (Figura 4(f)).

O KMB possui tempo polinomial e, de acordo com Doar e Leslie (1993), pode obter resultados bons em redes reais. Porém, a CSTPG obtida pelo KMB pode possuir uma grande discrepância de custo se comparada com do valor ótimo.

Por fim, Martins et al. (2000) sugere o uso do algoritmo de Kruskal (1956) com implementação de uma construção aleatória da solução. O pseudocódigo, apresentado no Algoritmo 3, representa a ideia geral do algoritmo de Kruskal (1956) aplicado por Martins et al. (2000).

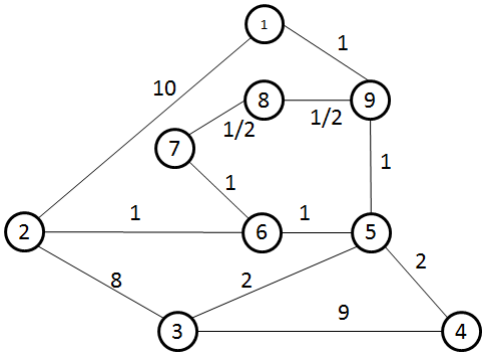
Algoritmo 3 Aleatorio_Kruskal(G, V_s, α)

```
1    r ← 0;
2    A ← {};
3    RCL ← Selecao(G, V_s, α);
4    Enquanto RCL ≠{} faça
5        r ← random() mod |RCL|;
6        A ← A ∪ RCL[r];
7        RCL ← Selecao(G, V_s, α);
8    retorna A;
fim Aleatorio_Kruskal
```

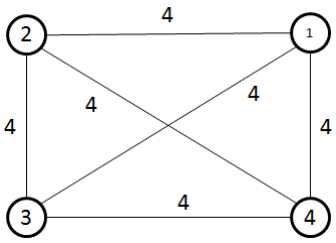
O algoritmo seleciona uma lista restrita de candidatos (RCL) com arestas $e = (i, j) \in E(G)$, então, seleciona aleatoriamente um dos membros pertencentes à RCL para adicionar a A. O processo se repete até que não existam mais candidatos a serem selecionados.

A seleção da lista de candidatos, chamada $Selecao(G, V_s, \alpha)$, é proposta de Martins et al. (2000). O parâmetro α[2] define o grau de aleatoriedade do algoritmo e é aplicado por $Selecao$ para selecionar uma lista composta por arestas e tais que $w_{ij} \leq w_{min} + \alpha(w_{max} - w_{min})$ sendo

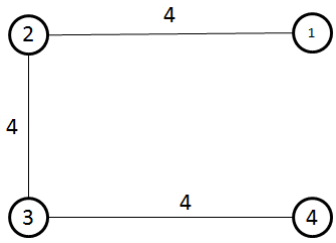
[2] Seus valores pertencem ao intervalo $0 \leq \alpha \leq 1$



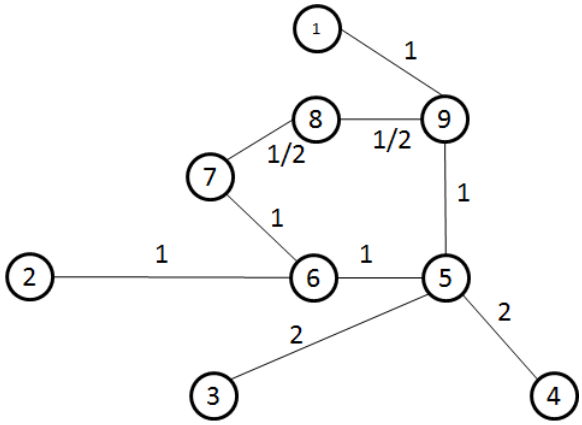
(a) *Grafo G.*



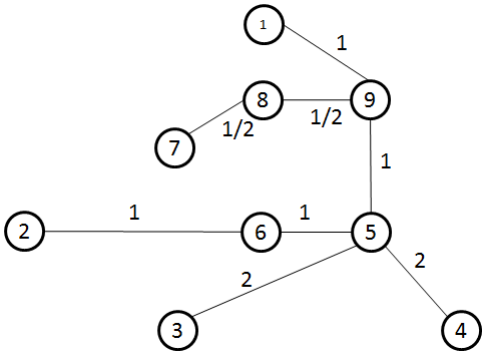
(b) *Grafo G_1.*



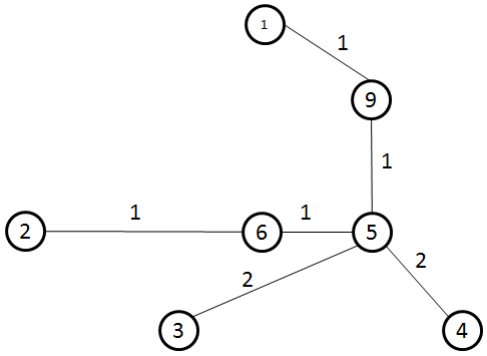
(c) *Árvore geradota mínima, T_1, de G_1.*



(d) *Grafo G_s.*



(e) *Árvore geradota mínima, T_s, de G_s.*



(f) *Árvore de Steiner, T_h, obtida de T_s.*

Figura 4 – Execução do algoritmo KMB com $S = \{v_1,\ v_2,\ v_3,\ v_4\}$. Imagens adaptadas de (KOU; MARKOWSKY; BERMAN, 1981).

w_{min} e w_{max}, respectivamente, o peso mínimo e o peso máximo das arestas que ainda não foram selecionadas.

O algoritmo original de (KRUSKAL, 1956) funciona de maneira gulosa e, portanto, pode obter soluções que sejam mais custosas que o ótimo. Entretanto, possui complexidade de tempo $O(\log V(G))$ e, com a alteração proposta por (MARTINS et al., 2000), suas soluções para um mesmo grafo G poderão ser mais diversificadas de acordo com o parâmetro α.

2.3 Meta-Heurísticas

Segundo Glover e Kochenberger (2003), uma meta-heurística é um método de solução no qual são organizados procedimentos mais refinados por meio dos quais é possível obter uma solução mais robusta para determinado problema. Das meta-heurísticas existentes, a meta-heurística de Algoritmos Genéticos (GA) é citada no *survey* de Bezenšek e Robič (2014). Um trabalho que aplica essa meta-heurística ao STP em grafos é o apresentado por Esbensen (1995)[3]. Antes do GA, Esbensen (1995) utiliza técnicas para redução de grafos. Esbensen (1995) aplica o conceito de genótipos[4] para especificar conjuntos de vértices enquanto, um decodificador, registra o fenótipo[5] correspondente por meio da heurística da Rede de Distância.

Esbensen (1995) verifica o quanto uma população $P = p_0, p_1, ..., p_{|P|-1}$ é adequada, F, aplicando a Equação 2.1. Na fase de cruzamento entre dois genótipos α e β, β é reordenado, antes do cruzamento, para que fique homólogo a α. O resultado obtido, no cruzamento, é passado adiante para avaliação. Por fim, há a fase de mutação, nela o operador inverte os *bits* pertencentes ao genótipo com base em uma probabilidade, p_{mut}, predefinida de mutação.

$$F(p_i) = \frac{2i}{|P| - 1}, i = 0, 1, ..., |P| - 1. \tag{2.1}$$

O GA apresentado no trabalho de Esbensen (1995) possui complexidade polinomial, obtendo soluções de alta qualidade com uso de uma quantia de tempo moderada, para os casos que selecionou de Beasley (2014), e consegue tratar casos em que o grafo possui quantidades [6] elevadas de vértices e arestas. Em contrapartida, o GA precisa manter registros das populações existentes e consequentemente é sujeito a um uso de memória elevado.

Bezenšek e Robič (2014) abordam, em seu *survey*, sobre a meta-heurística GRASP (FEO; RESENDE, 1995) que é uma meta-heurística iterativa caracterizada pela independência entre cada uma de suas iterações que, cada uma, possui duas fases distintas: a primeira de construção e a segunda de busca local como dito na Seção 1.

Na fase de construção do GRASP estratégias como as heurísticas apresentadas na Seção 2.2 são aplicáveis. O mesmo ocorre à busca local que, a partir da solução obtida pela fase construtiva, tentará obter uma melhora na solução inicial. Conforme é apresentado em forma de pseudocódigo no Algoritmo 4, a execução do GRASP inicia com a chamada de $Ler\_Entrada()$ que pega o grafo G a ser processado.

Com o grafo G, o GRASP realiza iterações nas quais sua fase de construção seleciona uma lista restrita de candidatos (RCL). Então, aplica a $Semente$ recebida em uma função

[3] O trabalho de Esbensen (1995) aborda o STPG sem aplicação ao roteamento *multicast*.
[4] O genótipo é a representação ou código genético de um indivíduo (ESBENSEN, 1995).
[5] O fenótipo é a aparência física do indivíduo (ESBENSEN, 1995).
[6] Foram testados grafos aleatórios com mais de 2500 vértices e 62.500 arestas (ESBENSEN, 1995).

Algoritmo 4 GRASP (RESENDE; RIBEIRO, 2002)

```
1   Ler_Entrada();
2   Para k = 1, ..., Max_Iteracoes faça
3       Solucao ← Construcao_Gulosa_Aleatoria(Semente);
4       Solucao ← Busca_Local(Solucao);
5       Atualiza_Solucao(Solucao, Melhor_Solucao);
6   fim;
7   retorna Melhor_Solucao;
fim GRASP
```

aleatória a fim de selecionar um dos candidatos na RCL. O processamento interno da fase de construção ocorre até que a RCL seja vazia, caracterizando o fim de montagem da $Solucao$.

Concluída a fase de construção, inicia-se a fase de busca local. A Busca Local utilizará a vizinhança de $Solucao$ para obter um ótimo local. Ao término de cada iteração do GRASP, é verificado se a solução obtida é melhor que a registrada em $Melhor\_Solucao$ e, caso seja, $Melhor\_Solucao$ recebe $Solucao$.

Ao atingir a quantidade máxima de iterações, $Max\_Iteracoes$, $Melhor\_Solucao$ é retornada como ótimo global. A vantagem, desta meta-heurística, é possuir uma estrutura simples e com poucos parâmetros. Entretanto, o GRASP não possui implementação de memória e, por consequência, não utiliza de soluções anteriores para melhorar a obtenção de soluções ótimas.

Martins et al. (2000) apresentam um GRASP adaptado ao paralelismo no qual utilizam o Algoritmo 3 como fase de construção e, como fase de busca local, uma estratégia híbrida. Nesse algoritmo, a busca local é híbrida pois mistura os conceitos de Vizinhança Baseada em Nós e de Vizinhança Baseada em Caminhos.

No conceito de Vizinhança Baseada em Nós, os vizinhos da solução obtida são definidos por todos os conjuntos, S , de vértices que sejam de nós de Steiner obtidos por meio da adição de um vértice não terminal a S ou pela remoção de um vértice do conjunto S. Já no conceito de Vizinhança Baseada em Caminho, *key-node* é um nó de Steiner que possua no mínimo grau 3 e *key-path* é um caminho na árvore de Steiner T_{STP} que todos seus vértices intermediários são nós de Steiner e que seus extremos sejam nós-chave ou vértices terminais. A Vizinhança Baseada em Caminho realiza trocas de caminhos-chave por caminhos mais curtos e, então, verifica se houve melhora na solução e atualiza as referências aos caminhos-chave e nós-chave (MARTINS et al., 2000).

A meta-heurística GRASP apresentada em (MARTINS et al., 2000) executa da seguinte forma:

1. Registra o menor custo atual e constrói um grafo de distância, $G' = (V_s, E')$, com $V_s \subset V(G)$ e $E' \subset E(G)$;

2. Define pesos $w'_{ij} \forall (i,j) \in E'$ para, então, iniciar suas iterações;

3. A cada nova iteração, seu primeiro passo é obter uma MST, T , a partir de G';

4. Verifica-se a ocorrência de T em iterações anteriores. Caso não tenha sido avaliada anteriormente, T é submetida à fase de busca local baseada em Caminho que retornará T_p;

5. Verifica-se a ocorrência da solução T_p em iterações anteriores e se o custo da mesma, para um parâmetro de restrição λ, é inferior a $(1 + \lambda)$ vezes o custo mínimo atual;

6. Após verificada, submete-se T_p à busca local baseada em Nós que deve retornar uma solução T_n que, caso possua um custo inferior ao menor atual, será a nova solução ótima.

No algoritmo proposto por Martins et al. (2000), durante a paralelização, as informações são atribuídas a um processo inicial, o processo mestre. O processo mestre realiza a divisão das iterações a serem executadas por cada processo escravo. Assim como no trabalho de Bastos e Ribeiro (2002), que será abordado mais adiante, uma semente distinta é passada para cada processo com objetivo de evitar repetições nas soluções iniciais.

O uso do algoritmo de Kruskal adaptado por Martins et al. (2000) permite um grau de variabilidade, da solução inicial obtida, entre cada uma das iterações a serem realizadas. Por meio dessas variações, o algoritmo de Martins et al. (2000) amplia as opções de solução avaliadas durante da fase de busca local e, em consequência, possui maiores chances de encontrar uma solução ótima. Uma desvantagem do algoritmo apresentado em (MARTINS et al., 2000) é sua fase de busca local que, para cada solução avaliada, necessita de até duas operações distintas com considerável custo computacional.

Um trabalho mais recente com a meta-heurística GRASP é o de Skorin-Kapov e Kos (2006). Nele, a meta-heurística GRASP é aplicada, com o CSTPG, para o problema de roteamento *multicast* e, para as instâncias do conjunto B obtidas em (KOCH; MARTIN; Voß, 2000), superou a meta-heurística de Busca Tabu (GLOVER, 1989; GLOVER, 1990) adaptada para CSTPG e o algoritmo proposto por Kompella, Pasquale e Polyzos (1993b). Em sua fase de construção, o algoritmo de Skorin-Kapov e Kos (2006) aplica uma heurística gulosa[7] tal como as apresentadas na Seção 2.2 enquanto que, na fase, de busca local no algoritmo de Skorin-Kapov e Kos (2006) utiliza uma variação da meta-heurística de Busca Tabu para CSTPG.

A Busca Tabu em (SKORIN-KAPOV; KOS, 2006) é a mesma proposta por Skorin-Kapov e Kos (2003). Nela, durante suas iterações, adiciona[8] as soluções obtidas na Lista Tabu, seu número de iterações é limitado por uma variável pré-definida e suas soluções são restritas por uma variável, Δ, de atraso.

Com base na solução, A_0, recebida da fase de construção, a Busca Tabu utilizada por Skorin-Kapov e Kos (2003) registra o custo da solução recebida e verifica sua vizinhança. A

[7] Entretanto, assim como (MARTINS et al., 2000), a heurística gulosa aplicada é adaptada para possuir algum grau de aleatoriedade.

[8] Esta operação inclui os casos em que não há uma vizinhança factível.

verificação, $(I - A)$, compara a vizinhança de A_0 aos vértices que estão na Lista Tabu. Os vértices, da vizinhança de A_0, que não estiverem na Lista Tabu são adicionados à árvore estruturada, $A_{vizinha}$.

A árvore $A_{vizinha}$ é estruturada a partir de A_0 e, depois de alterada, tem seu resultado comparado com a restrição Δ. Caso a mudança resulte em um custo inferior ao mínimo atual, A_{atual}, então $A_{atual} \leftarrow A_{vizinha}$. Se o custo de $A_{vizinha} > \Delta$ ou se o grafo for desconexo, então o custo referente à $A_{vizinha}$ será definido como infinito. Este processo repete até que todos os vértices $v \in V(G) \backslash V_s$ sejam verificados.

Terminado o processo anterior, é verificado se houve algum vizinho factível, custo $\neq \infty$, e se custo de $A_{atual} < custo\ de\ A_0$. Se constatado que o custo de $A_{atual} < custo\ de\ A_0$, então $A_0 \leftarrow A_{atual}$ e retorna-se a $(I - A)$ até que se atinja o número máximo de execuções.

Uma Busca Tabu paralela é proposta, no trabalho de Bastos e Ribeiro (2002), com uma adaptação ao STPG utilizando a estratégia de *path-relinking*[9]. O algoritmo inicia com a construção de sua solução inicial para, então, começar suas iterações. Em cada iteração é verificado se S [10]$\in R$ [11] e, caso seja verdade, o valor das repetições de S é incrementado e, também, do domínio tabu [12]. Caso seja alcançado o valor máximo de repetições de S , então será realizada uma diversificação. O processo de diversificação é uma nova execução do algoritmo de construção da solução inicial, ele também ocorre caso a solução obtida não seja aceita como de elite.

A *path-relinking* implementada no algoritmo de Bastos e Ribeiro (2002), gera novas soluções por meio das soluções de elite obtidas. Sua execução ocorre para cada par existente no conjunto de soluções de elite e, em seu início, calcula a diferença simétrica para cada par. Com estas informações, são selecionados os movimentos a serem aplicados para alcançar uma solução de custo mínimo (BASTOS; RIBEIRO, 2002).

O paralelismo, tanto na fase da Busca Tabu Reativa quanto na de *path-relinking*, é realizado dividindo as iterações entre processos escravos que devem manter o processo mestre atualizado. Cada processo escravo utiliza um parâmetro distinto, esta abordagem é aplicada para evitar que utilizem soluções iniciais semelhantes.

De acordo com Bastos e Ribeiro (2002), as soluções obtidas superam as fornecidas por outros algoritmos de Busca Tabu assim como os algoritmos de GRASP apresentados por Martins (1999), Martins et al. (2000) [13]. Entretanto, um ponto negativo na implementação refere-se à necessidade de manter a sincronização entre os processos para evitar falhas, na solução, devido informações desatualizadas.

---

9 A estratégia de *path-relinking* executa com base em um grupo de soluções de elite obtidas na fase de busca.
10 Conjunto dos vértices $v \in V(G)$ que são nós de Steiner presentes na solução atual.
11 Lista de soluções já visitadas.
12 Determina o número de iterações nas quais a operação é considerada proibida (BASTOS; RIBEIRO, 2002).
13 Tanto Bastos e Ribeiro (2002) quanto Martins et al. (2000) utilizam casos de teste fornecidos em Beasley (2014)

2.4 Paralelismo e Programação Distribuída aplicados ao roteamento *multicast*

Nessa seção, é realizada uma análise sobre o trabalho de Kompella, Pasquale e Polyzos (1993a) e uma comparação breve, do mesmo, aos trabalhos de Martins et al. (2000), Bastos e Ribeiro (2002) que aplicam o conceito de paralelismo.

Em seu trabalho, Kompella, Pasquale e Polyzos (1993a) apresentam o conceito de que a análise apenas do atraso ou apenas do custo da árvore de roteamento *multicast* é insuficiente. Kompella, Pasquale e Polyzos (1993a) apresentam dois algoritmos distribuídos que tratam, o roteamento *multicast*, sob esse ponto de vista.

Conforme definido neste capítulo, para uma árvore de Steiner restrita, considera-se um limitante superior Δ. No trabalho apresentado por Kompella, Pasquale e Polyzos (1993a), D representa a função do atraso da aresta e Δ restringe o atraso existente no percurso obtido entre os vértices v'_0, inicial, a um vértice v'_i, $0 < i \leq |V_s|$, pertencentes ao grupo *multicast* V_s.

Para cada percurso de v_0 a v_i, $P(v_0, v_i)$, pertencente a arvore, $T_{restrita}$, Kompella, Pasquale e Polyzos (1993a) consideram:

$$\sum_{e \in P(v_0, v_i)} D(e) < \Delta$$

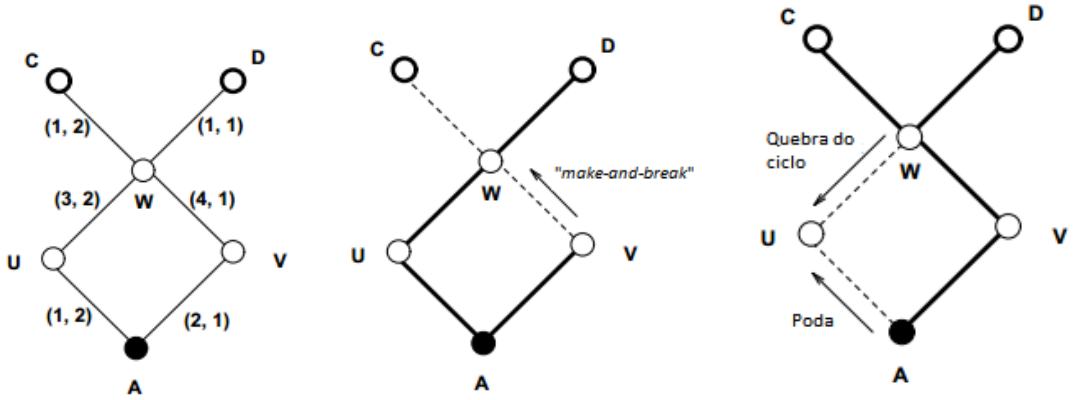
Os dois algoritmos são baseados no algoritmo de MST distribuído apresentado em (GALLAGER; HUMBLET; SPIRA, 1983). Entretanto, para o uso de algoritmos distribuídos, um desafio significante de acordo com Kompella, Pasquale e Polyzos (1993a), é verificar se a árvore está sendo construída corretamente.

O primeiro algoritmo em (KOMPELLA; PASQUALE; POLYZOS, 1993a), $DMCT_c$, utiliza uma função f_c para selecionar arestas. A função f_c consiste em uma estratégia gulosa que, por meio de uma sub-árvore existente, seleciona a aresta menos custosa.

O segundo algoritmo, $DMCT_{cd}$, utiliza uma função f_{cd} para selecionar arestas. Através da função f_{cd}, o $DMCT_{cd}$ verifica a diferença entre o atraso máximo permitido, Δ, e o atraso existente, $P(v)$, no percurso entre o vértice de partida a um vértice v na árvore. Caso $P(v)$ seja menor que Δ, então, o algoritmo reutiliza esse percurso para alcançar os vértices destino que restam.

Um problema que Kompella, Pasquale e Polyzos (1993a) destacam é o término falso da execução. O término falso ocorre quando, o algoritmo retorna que não existem soluções enquanto existem (KOMPELLA; PASQUALE; POLYZOS, 1993a). Com intuito de contornar o problema de término falso, Kompella, Pasquale e Polyzos (1993a) sugerem uma fase "*make-and-break*" de ciclos (Figura 5).

A fase de *make-and-break* consiste em, quando o algoritmo termina sua execução e não cobre todos vértices destino, criar um ciclo como na Figura 5(b) e selecionar o percurso menos custoso. Em consequência, as arestas que pertencem ao percurso com maior custo são podadas da árvore conforme apresentado na Figura 5(c).



(a) (*KOMPELLA; PASQUALE; POLYZOS, 1993a*) (b) (*KOMPELLA; PASQUALE; POLYZOS, 1993a*) (c) (*KOMPELLA; PASQUALE; POLYZOS, 1993a*)

Figura 5 – Execução do algoritmo $DMCT_c$ de Kompella, Pasquale e Polyzos (1993a) aplicando a fase de *make-and-break*.

Kompella, Pasquale e Polyzos (1993a) destacam, em seu trabalho, o uso de algoritmos distribuídos como uma estratégia com alta escalabilidade que permite a obtenção de soluções boas em grafos considerados grandes. Entretanto, o uso de algoritmos distribuídos está sujeito à necessidade de controlar cada uma de suas instâncias. A demanda por um controle de todas as instâncias acarreta um custo computacional elevado com trocas de mensagens entre cada instância, assim como um gasto considerável com a verificação das informações repassadas.

Nas propostas de Martins et al. (2000), Bastos e Ribeiro (2002), como apresentado na Seção 2.3, é utilizado o paralelismo. Como as tarefas são divididas entre diversos processos escravos, nos dois trabalhos, é necessário um controle sobre o envio de informações. Porém, o controle aplicado aos processos escravos é direcionado à comunicação entre eles. O controle não considera se as informações recebidas indicam se a árvore foi criada corretamente, isso ocorre devido a cada processo escravo trabalhar com a construção completa da árvore. Ao término de sua execução, o processo envia seu resultado ao processo mestre que irá selecionar a melhor solução dentre as enviadas a ele.

Ao comparar os trabalhos de Martins et al. (2000), Bastos e Ribeiro (2002) ao de Kompella, Pasquale e Polyzos (1993a) nota-se que, uma implementação distribuída, possui escalabilidade superior a uma implementação paralela quando submetidas a grafos considerados grandes. Entretanto, a implementação de um algoritmo distribuído requer um conjunto extenso de verificações enquanto que, para a implementação de algoritmos paralelos, o conjunto de verificações se restringe mais à comunicação. Por fim, ambas as estratégias são interessantes ao

ponto de vista computacional ressaltando que, a estratégia distribuída, requisita uma implementação mais complexa que a estratégia paralela.

3 Método de Solução

Como uma quantidade significativa das instâncias analisadas são esparsas e possuem elevada quantidade de elementos, a estrutura utilizada para tratá-las deve seguir uma estratégia para redução no consumo de memória. Devido a isso, determinar como representar as informações fornecidas pelas instâncias foi o primeiro problema a ser solucionado neste TCC.

Pela característica esparsa da maioria das instâncias, o uso de uma matriz de adjacências $M_{n \times n}$ para uma instância grande com n vértices iria requisitar uma reserva de memória além do que será utilizado. Então, para representar as informações a serem tratadas, a alternativa tomada como solução é a representada na Figura 6.

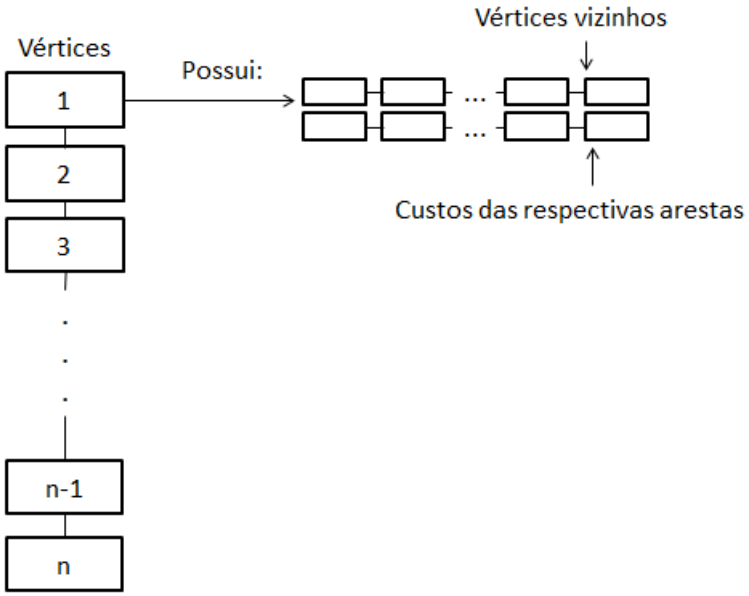


Figura 6 – Representação do grafo.

A estrutura representada na Figura 6 consiste em um vetor *N* no qual cada posição representa um vértice pertencente ao grafo estudado. Para os vértices que possuírem arestas conectando-os a outros, as arestas serão representadas em vetores acoplados às respectivas posições dos vértices em *N*. A representação das arestas será feita com dois vetores[1] relacionados, esses vetores devem indicar os vizinhos e o respectivo custo para alcançar cada um deles.

A vantagem da estrutura apresentada na Figura 6 está em não ter que reservar memória para arestas que não existem. Entretanto, uma desvantagem visível é a necessidade de alocar duas vezes a mesma aresta o que a torna inviável a grafos grandes e não esparsos. Em relação ao tempo de acesso a determinado vértice em *N*, a complexidade é $O(1)$ enquanto que para obter uma aresta a complexidade é $O(k)$ com k = tamanho da vizinhança do vértice analisado.

[1] Estes vértices terão tamanho igual ao grau do vértice em *N* no qual forem alocados.

Determinada a estrutura de representação do grafo, a segunda estrutura definida foi a necessária para representação da árvore criada a partir dele. Nessa estrutura, cada vértice possui uma referência ao seu predecessor, vértice pai, a outros vértices que tenham mesmo predecessor, vértices irmãos, e aos vértices posteriores a ele, vértices filhos, como representado na Figura 7.

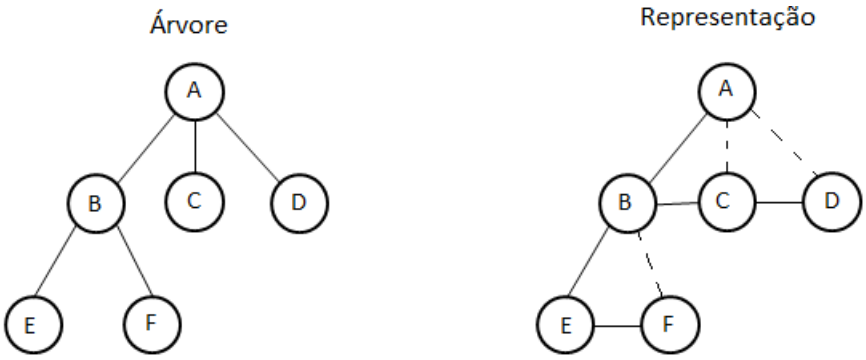


Figura 7 – Representação da árvore.

Dada a Figura 7, a representação de cada vértice da árvore é feita por meio de uma referência para o vértice pai, uma lista duplamente encadeada para os irmãos e uma referência a um de seus filhos. A vantagem dessa estrutura é uma representação mais intuitiva para trabalhar com as informações. Em contrapartida, o uso da mesma aumenta a complexidade das atualizações necessárias no decorrer da execução do algoritmo.

No algoritmo desenvolvido, a estrutura apresentada na Figura 6 é denotada como *typeNode*, enquanto que a representada na Figura 7 é denotada por res. Vale ressaltar também a existência da estrutura *sol* que representa a solução e possui uma referência à raiz da árvore e armazena o custo total da mesma.

Com as estruturas definidas, o pseudo código no Algoritmo 5 demonstra a organização do GRASP implementado. Como entrada do algoritmo temos um grafo conexo com peso (G, W_g), um valor α para ser aplicado na seleção dos candidatos a entrar na árvore, os membros do grupo *multicast* $gM \in V(G)$ e o número máximo de iterações, mI, para o GRASP.

No Algoritmo 5, após receber as entradas, o primeiro passo é definir o número de *threads* que serão utilizadas. Para que o problema fique balanceado, o número de iterações imposto ao algoritmo corresponde a um múltiplo do número de *threads* utilizado. O número de iterações será, então, dividido pelo número de *threads* e cada *thread* executará $\frac{mI}{numero\ de\ threads}$. Por meio do comando *privado(Semente,* T*,* $V(T)$*)*, cada *thread* terá sua própria semente e solução.

Cada *thread* irá obter sua solução inicial com a fase de construção realizada no algoritmo *PRIM_SG* a ser explicado na Seção 3.1. Com a solução inicial obtida na linha 5, executa-se então o algoritmo *Busca_Local* descrito na Seção 3.2, que verificará a vizinhança da solução em busca de melhorias no custo da mesma.

O intervalo de linhas 6 a 8 representa uma seção crítica, nela apenas uma *thread* por

Algoritmo 5 GRASP_P

ENTRADA: (G, W_g), α, gM, mI;
SAÍDA: uma árvore ótima $T^* = (V^*, F^*)$ de G e sua função peso $W(F)$;
1 Define número de *threads*;
2 **Para** k = 1, ..., mI **faça** em paralelo privado(Semente, T), compartilhado(T^*) tal que T é a solução construída na iteração atual, Semente é um valor individual para cada iteração
3 Semente ← (time(NULL)*(thread_Atual + 1));
4 Solucao ← PRIM_SG(G, gM, Semente, α);
5 Solucao ← Busca_Local(G, gM, $V(T)$, T);
6 **Inicia seção crítica**;
7 Atualiza_Solucao(T, T^*);
8 **Acaba seção crítica**;
9 **fim**;
10 **retorna** Melhor_Solucao;
11 RETORNA T^*;
fim GRASP_P

vez poderá atualizar variável T^*. Esse trecho é um provável gargalo no algoritmo devido à necessidade de toda *thread* ter que passar por ele. Vale ressaltar que a atualização de T^* por T na linha 7 ocorrerá apenas se $custo(T) < custo(T^*)$ e que é necessária uma avaliação mais a fundo sobre o percentual de tempo de execução gasto devido ao uso da seção crítica. O algoritmo repetirá o processo até que mI seja atingido.

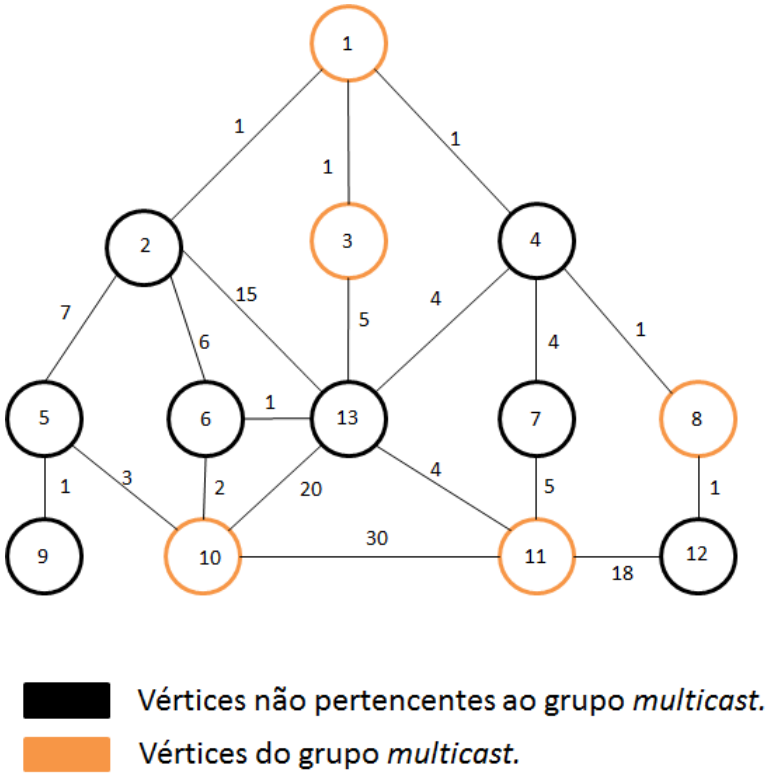


Figura 8 – Grafo conexo com peso (G, W).

Apresentamos na seção a seguir, utilizando o grafo na Figura 8 como base para os

exemplos, maiores detalhe sobre o algoritmo de Prim semi-guloso e na Seção 3.2, o algoritmo de busca local implementado para trabalhar com a solução obtida por meio do algoritmo de Prim adaptado para tratar o STPG.

3.1 Fase Construtiva

Inicialmente, a fase construtiva que pretendíamos utilizar era o algoritmo de Kruskal modificado para ser semi-guloso. Entretanto, como o Kruskal forma uma floresta durante o processo de construção da árvore geradora mínima, em consequência da estrutura de dados utilizada, o controle de cada árvore formada até se obter a solução final iria possuir um elevado custo computacional. Além disso, o algoritmo não teria características distribuídas.

Para a fase construtiva foi aplicado então o algoritmo de Prim adaptado à proposta semi-gulosa. A escolha do algoritmo de Prim deve-se à sua execução trabalhar com uma única árvore enraizada e por trocar os vértices da árvore caso encontre outros que os substituam e melhorem o custo da solução. O Algoritmo 6 apresenta a estratégia proposta, $PRIM\_SG$.

Algoritmo 6 PRIM_SG

ENTRADA: (G, W_g), α, gM, Semente
SAÍDA: uma árvore de Steiner $T = (V, F)$ de G e sua função peso $W(F)$

```
1    Para k = 1, ..., |V(G)|
2        custo_Nos[k] ← ∞;
3    fim;
4    copia_gM ← gM;
5    Enquanto copia_gM ≠ ∅ faça
6        a_e ← selecao(α, G, V(T), Semente, custo_Nos, bool_Mudanca) tal que a_e representa
a aresta escolhida;
7        Se bool_Mudanca faça
8            filho ← encontra_No(root, a_e.destino);
9            p_n ← encontra_No(root, a_e.fonte) tal que p_n é o novo pai;
10           custo_Nos[filho.valor] ← p_n.custo + W_g(a_e);
11           aplica_Mudanca(filho, NULL, p_n, V(T), gM, G, T, W_g(a_e), NULL);
12           mudar_Custos(T, filho, custo_Nos);
13       Senão
14           p ← encontra_No(root, a_e.fonte), tal que p ∈ V(T);
15           custo_Nos[ a_e.destino] ← pai.custo + W_g(a_e);
16           T ← add_No(T, a_e, V(T), copia_gM);
17       fim;
18   fim;
19   root ← poda_Arvore(T, gM, V(T));
20   RETORNA T;
fim PRIM_SG
```

O Algoritmo 6 utiliza outros 5 algoritmos. O *encontra_No* realiza uma busca em profundidade e os demais serão explicados mais adiante. Como entrada do algoritmo temos um

grafo conexo com peso (G, W), um valor α para ser aplicado na seleção dos candidatos a entrar na árvore, os membros do grupo *multicast* gM e a semente que servirá como uma das entradas para $selecao()$.

O primeiro passo realizado no Algoritmo 6 é inicializar o vetor $custo\_Nos$ e uma cópia de gM. Vale a pena lembrar que $n = |V(G)|$. O conjunto *copia_gM* inicialmente recebe o grupo *multicast*. O laço das linhas 5 a 18 é executado até que o conjunto $copia\_gM$ esteja vazio.

Dentro do laço, a seleção da aresta a ser adicionada na árvore é realizada de acordo com o pseudocódigo no Algoritmo 7. Na função $selecao$ são selecionadas, ordenadas de acordo com seus respectivos pesos, todas as arestas que tenham ao menos um de seus extremos na árvore. Escolhidas as arestas adjacentes à árvore, define-se o peso máximo , por meio do parâmetro α que dita o grau de aleatoriedade do algoritmo (se for 1, a lista toda é considerada, se próximo de 0, apenas as arestas de menor custo serão consideradas). Essa restrição implica em uma lista restrita de candidatos, a_p, que será submetida à seleção, na linha 8 do Algoritmo 7.

Algoritmo 7 selecao

ENTRADA: (G, W_g), α, $V(T)$, gM, Semente, custo_Nos
SAÍDA: a_e
1 $a_v \leftarrow$ pega_vizinhanca_ordenada($V(T)$), tal que a_v representa o conjunto de arestas selecionadas e que possuem incidência em T;
2 intervalo = menor($W_g(a_e)$) + α*($W_g(a_e)$) - menor($W_g(a_e)$));
3 $a_p \leftarrow$ toda $e \in a_v$ tal que $W_g(e) \leq$ intervalo;
4 a_e = a_p[random(Semente)%tamanho(a_p)];
5 RETORNA a_e
fim selecao

A variável *bool_Mudanca*, passada como parâmetro da função $selecao$ na linha 6, só recebe o valor *true* caso os dois extremos da aresta sejam vértices já pertencentes à árvore. Quando *bool_Mudanca* for *true*, serão executadas as instruções das linhas 8 a 12 que consistem em localizar os vértices destino e origem, atualizar o custo referente ao vértice destino, aplicar as mudanças necessárias e atualizar os custos do nó destino de seus filhos. Caso *bool_Mudanca* seja *false*, localizamos o vértice origem da aresta, que já pertence à árvore, atualizamos o custo referente ao vértice destino e adicionamo-o à árvore.

Acerca do trecho que envolve as linhas 8 a 13, a função para aplicar mudanças na árvore é abordada com maiores detalhes na Seção 3.1.1 e a função $mudar\_Custos$ segue o princípio da busca por profundidade para atualizar o custo de todos os vértices que descendam do vértice passado como parâmetro. Ao atingir a condição de parada do laço *while*, a árvore obtida pode possuir vértices que não pertencem ao grupo *multicast*, que estão como vértices folha e, consequentemente, estão adicionando custo à mesma sem que sejam usados para o roteamento. Como exemplo, na Figura 9 tem-se uma árvore de roteamento resultante do Algoritmo 9, até a linha 18, aplicado ao grafo apresentado na Figura 8.

Na literatura, o processo de poda, por vezes, é utilizado como um pré-processamento

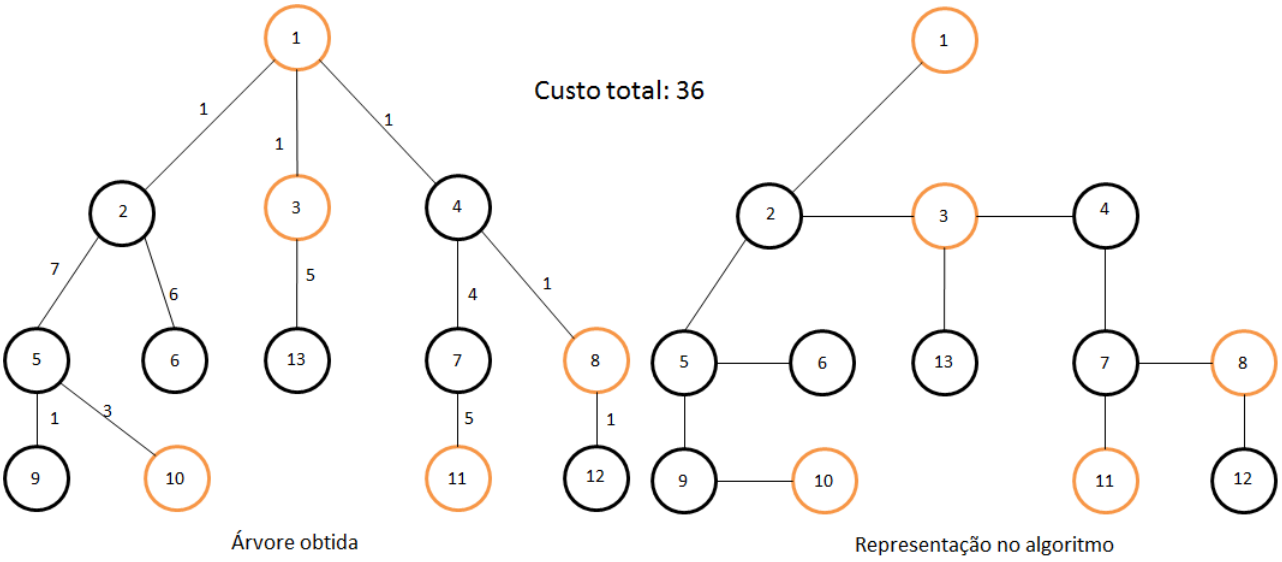


Figura 9 – Árvore resultante das operações dentro do laço *while*.

do grafo. O trabalho apresentado por Skorin-Kapov e Kos (2006) é um exemplo. O processo de poda aqui apresentado possui a finalidade de retirar da árvore os vértices não utilizados e deixar apenas os membros do grupo *multicast* como folhas na árvore.

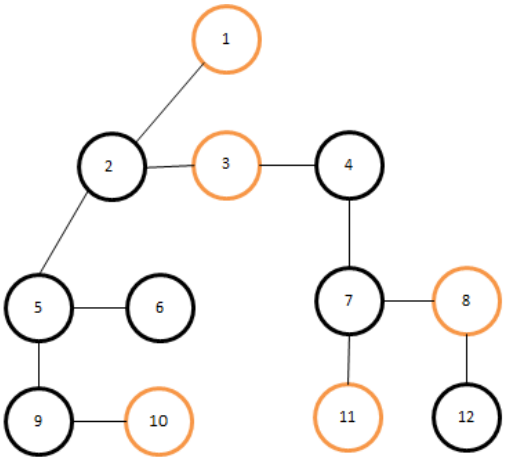
A operação de poda é representada nas Figuras 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21 e apresentada pelo pseudocódigo no Algoritmo 8. Para explicação do Algoritmo 8, considere a Figura 10(a) como a entrada para o algoritmo de poda, que o vértice raiz da árvore recebida pelo algoritmo de poda sempre pertencerá ao grupo *multicast*, os vértices a serem analisados serão descendentes deste vértice raiz e o critério de parada será alcançar o vértice raiz.

Algoritmo 8 poda_Arvore

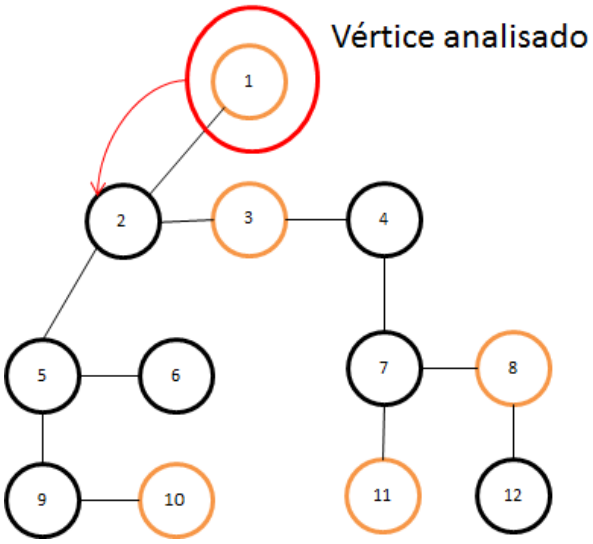
ENTRADA: (G, W_g), $T\ V(T)$, gM
SAÍDA: uma árvore de Steiner $T = (V, F)$ de G e sua função peso $W(F)$

1 **Enquanto** $v_{atual} \neq root$ **faça** tal que v_{aux} é o vértice atual que é inicializado com um filho de $root$;
2 check = verifica_Se_Está_No_Grupo_Multicast($v_{atual}.valor$);
3 **Se** $\exists\ v_{atual}.filho$ **faça**
4 $v_{atual} \leftarrow v_{atual}.filho$;
5 **Senão Se** $\exists\ v_{atual}.proxIrmao$ **faça**
6 **Se** check **faça**
7 $v_{atual} \leftarrow v_{atual}.proxIrmao$;
8 **Senão**
9 **Se** $\exists\ v_{atual}.prevIrmao$ **faça**
10 $v_{atual}.prevIrmao.proxIrmao \leftarrow v_{atual}.proxIrmao$;
11 $auxiliar \leftarrow v_{atual}$;
12 $v_{atual} \leftarrow v_{atual}.proxIrmao$;
13 **Senão**
14 $v_{atual}.pai.filho \leftarrow v_{atual}.proxIrmao$;
15 $v_{atual}.proxIrmao.prevIrmao \leftarrow$ NULL;
16 $auxiliar \leftarrow v_{atual}$;
17 $v_{atual} \leftarrow v_{atual}.proxIrmao$;
18 **fim**
19 remove($auxiliar.valor$, $V(T)$);
20 **fim**
21 **Senão Se** !check **faça**
22 **Se** $\nexists\ v_{atual}.prevIrmaos$ **faça**
23 $v_{atual}.pai.filho \leftarrow$ NULL;
24 $auxiliar \leftarrow v_{atual}$
25 $v_{atual} \leftarrow v_{atual}.pai$;
26 **Senão**
27 $v_{atual}.prevIrmaos.proxIrmao \leftarrow$ NULL;
28 $auxiliar \leftarrow v_{atual}$;
29 $executar \leftarrow true$;
30 **Enquanto** $executar$ **faça**
31 **Se** $v_{atual}.pai \neq root$ && $\exists\ v_{atual}.pai.proxIrmao$ **faça**
32 $v_{atual} \leftarrow v_{atual}.pai.proxIrmao$;
33 $executar \leftarrow false$;
34 **Senão Se** $v_{atual}.pai$ == $root$ **faça**
35 $v_{atual} \leftarrow v_{atual}.pai$;
36 $executar \leftarrow false$;
37 **Senão**
38 $v_{atual} \leftarrow v_{atual}.pai$;
39 **fim**
40 **fim**
41 **fim**
42 remove($auxiliar.valor$, $V(T)$);
43 **Senão**
44 Executa laço semelhante ao localizado na linha 34.
45 **fim**
46 **fim**

fim poda_Arvore



(a) *Árvore submetida à função de poda com os valores* $\{1, 3, 8, 10, 11\} \in grupo\_Multicast$



(b) *Execução, 1ª iteração.*

Figura 10 – Execução do algoritmo $poda\_Arvore$.

A primeira iteração, apresentada Figura 10(b), representa a operação na linha 2 do Algoritmo 8. Nela o vértice 2 é atribuído como o próximo vértice a ser analisado. As iterações posteriores são as seguintes:

1. O vértice 2 possui filhos e o próximo vértice para ser analisado é o de número 5 (Figura 11). Execução do intervalo de linhas [3,4] do Algoritmo 8;

2. O vértice 5 também possui filhos e o próximo vértice para ser analisado é o de número 9 (Figura 12). Executado no mesmo intervalo de linhas que o vértice 2;

3. O vértice 9 não possui filhos, mas possui um irmão. Devido a isso, o vértice 9 ativa a condicional da linha 5;

    - Como o vértice $9 \notin gM$ ativa a condicional da linha 8;

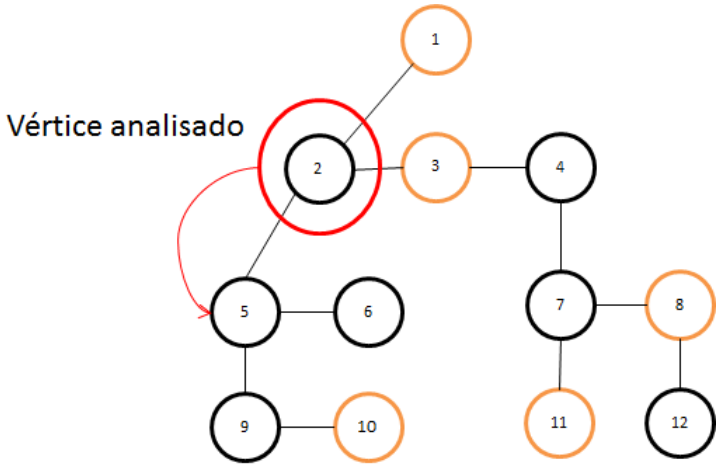


Figura 11 – Execução do algoritmo $poda\_Arvore$, 2ª iteração.

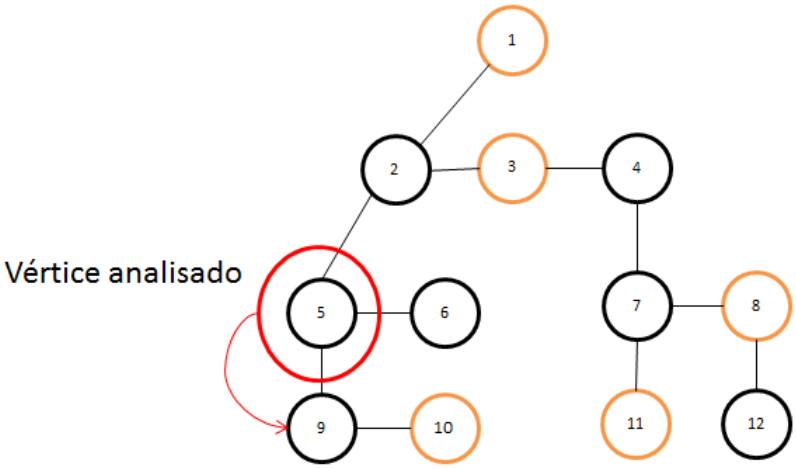


Figura 12 – Execução do algoritmo $poda\_Arvore$, 3ª iteração.

- Na condicional da linha 8, ativa-se a condicional da linha 13 dado que o vértice 9 não possui um irmão anterior a ele;
- Por fim, o vértice 10 é atribuído como primeiro da lista de filhos do vértice 5 e atribuído como próximo vértice a ser visitado. O vértice 9 é podado (Figura 13).

4. O vértice 10 não possui filhos, não possui um próximo irmão, mas $10 \in mG$ e, consequentemente, aciona a condicional da linha 45.

- Na linha 44, é executado um laço igual ao presente no intervalo de linhas [34,43] no qual: Verifica se o pai, vértice 5, é o raiz e se ele possui algum irmão;
- Como o vértice 5 não é raiz e possui um irmão, o próximo vértice a ser avaliado é o vértice 6 (Figura 14).

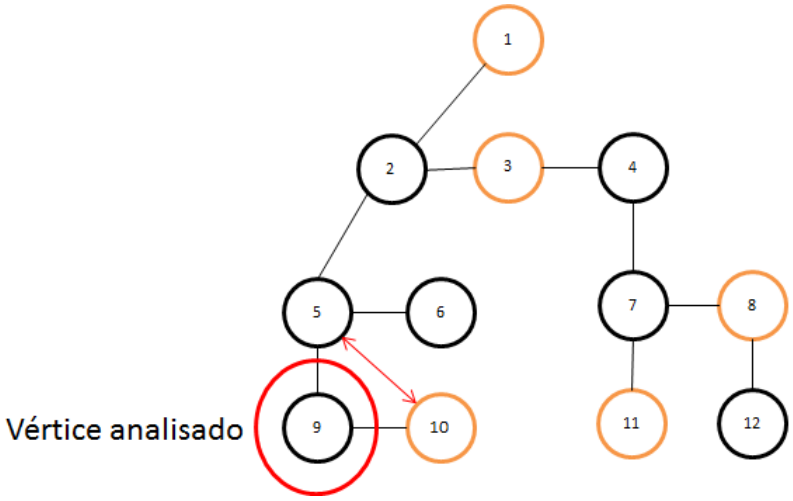


Figura 13 – Execução do algoritmo $poda\_Arvore$, 4ª iteração.

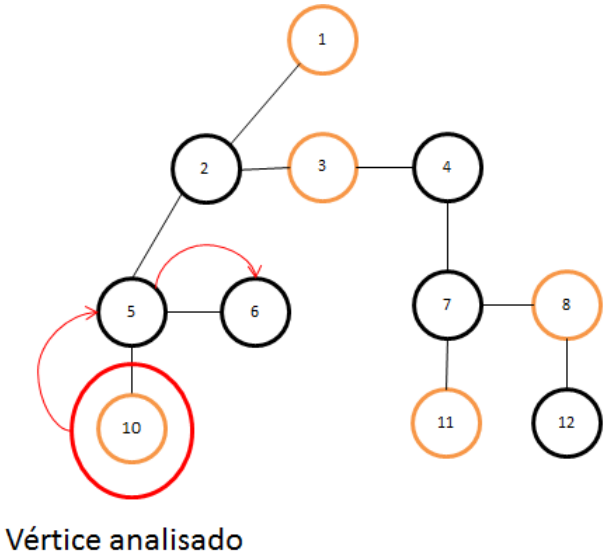


Figura 14 – Execução do algoritmo $poda\_Arvore$, 5ª iteração.

5. O vértice 6 não possui filhos, não possui um próximo irmão e $6 \notin grupo\_Multicast$ ativando a condicional na linha 21;

   - 6 possui o vértice 5 como seu irmão predecessor na lista encadeada e cai na condicional da linha 26;
   - No intervalo de linhas [27,28], são alteradas as referências de 5;
   - O laço no intervalo de linhas [30,40] encontra o vértice 3 como irmão de 2 e o atribui como próximo vértice a ser analisado;
   - O algoritmo poda o vértice 6 na linha 42 (Figura 15).

6. O vértice 3 não possui filhos, possui um próximo irmão e $3 \in gM$. Sob esta configuração são ativadas as condicionais nas linhas 5 e 6 e o vértice 4 passa a ser o próximo vértice a

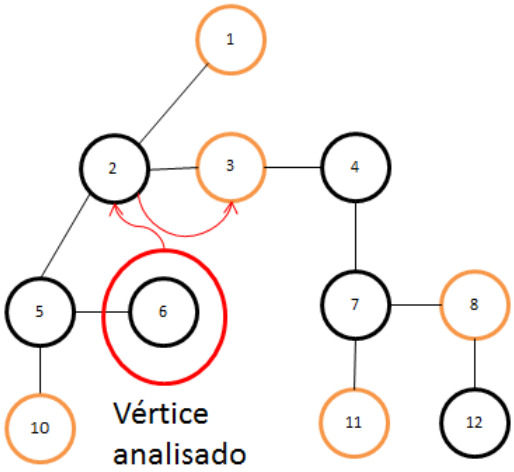


Figura 15 – Execução do algoritmo *poda_Arvore*, 6ª iteração.

ser analisado (Figura 16);

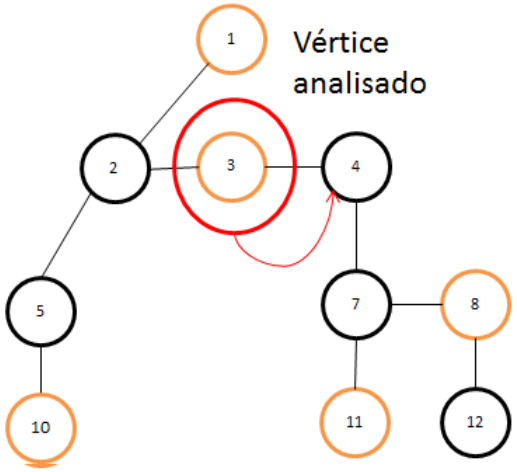


Figura 16 – Execução do algoritmo *poda_Arvore*, 7ª iteração.

7. O vértice 4 possui filhos e o próximo vértice a ser analisado é o vértice 7 (Figura 17);

8. O vértice 7 possui filhos e o próximo vértice a ser analisado é o vértice 11 (Figura 18);

9. O vértice 11 ativa a mesma operação feita com o vértice 10. O próximo vértice a ser analisado passa a ser o de número 8 (Figura 19);

10. O vértice 8 possui um filho e ativa a condicional da linha 7. O próximo vértice a ser analisado passa a ser o de número 12 (Figura 20);

11. O vértice 12 não possui filhos, não possui um próximo irmão e $12 \notin gM$;

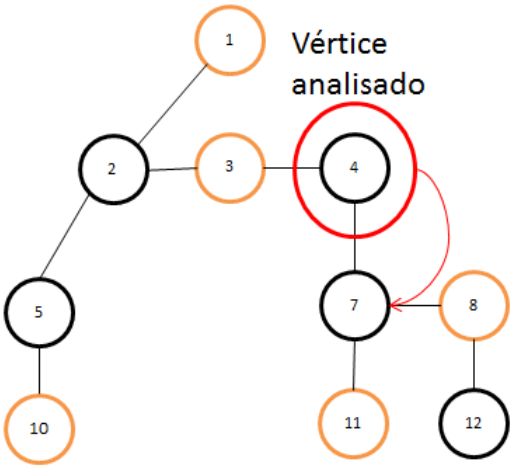


Figura 17 – Execução do algoritmo *poda_Arvore*, 8ª iteração.

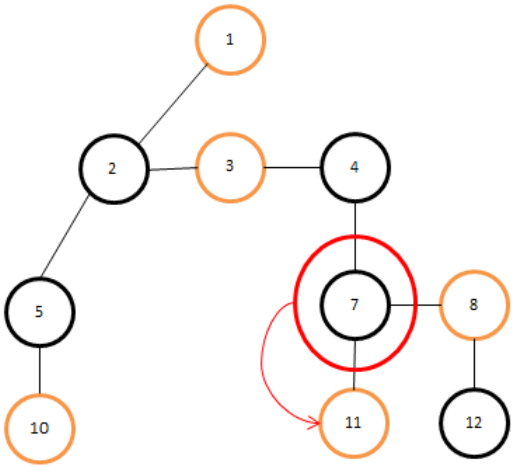


Figura 18 – Execução do algoritmo *poda_Arvore*, 9ª iteração.

- 12 não possui um vértice como irmão predecessor e, portanto, cai nas condicionais das linhas 26 e 27 e é podado;
- O vértice 8 retorna a ser analisado.

12. Devido à nova configuração do vértice 8, ativa-se a condicional na linha 43;

- Durante a execução do laço, os vértices 8 e 4 caem numa condicional igual à da linha 37;
- Por fim, é alcançado o vértice 1 e se encerra a execução (Figura 21).

Como resultado dos passos executados, se obtém a árvore representada na Figura 22.

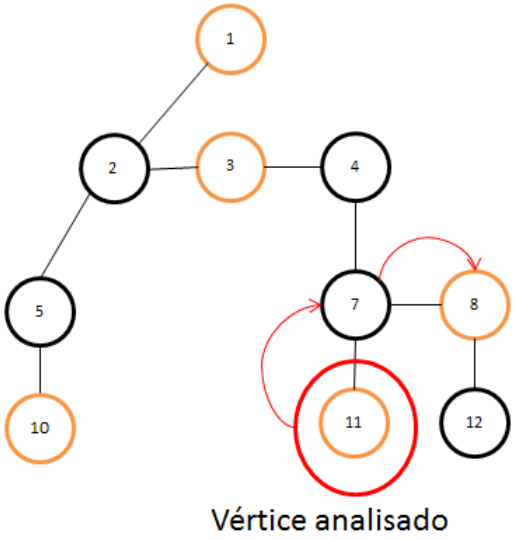


Figura 19 – Execução do algoritmo *poda_Arvore*, 10ª iteração.

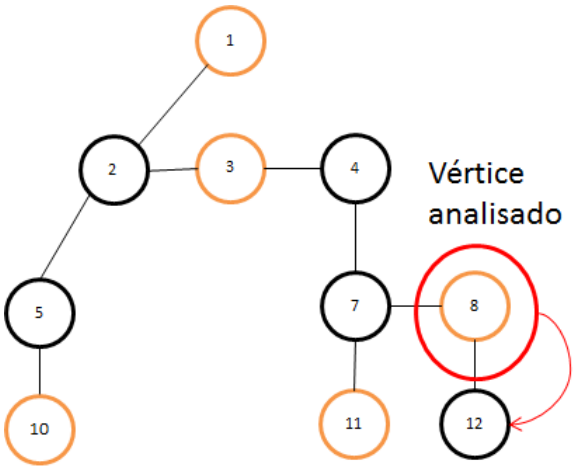


Figura 20 – Execução do algoritmo *poda_Arvore*, 11ª iteração.

Assim como a função para atualizar custos, a função *poda_Arvore* segue o conceito de busca por profundidade de maneira iterativa. Uma explicação sucinta do algoritmo pode ser tomada pelas seguintes assertivas:

- A partir do nó raiz da árvore, escolha um de seus filhos para ser v_{atual}
- Se v_{atual} possuir ao menos um filho e $\exists$ filho que não foi visitado, vá para o $v_{atual} \leftarrow v_{atual}.filho$;
- Se v_{atual} não possuir filho, mas possuir irmão não visitado e está no grupo *multicast*, $v_{atual} \leftarrow v_{atual}.irmão$;
- Se v_{atual} não possuir filho mas possuir irmão não visitado e não pertencer ao grupo *multicast*:

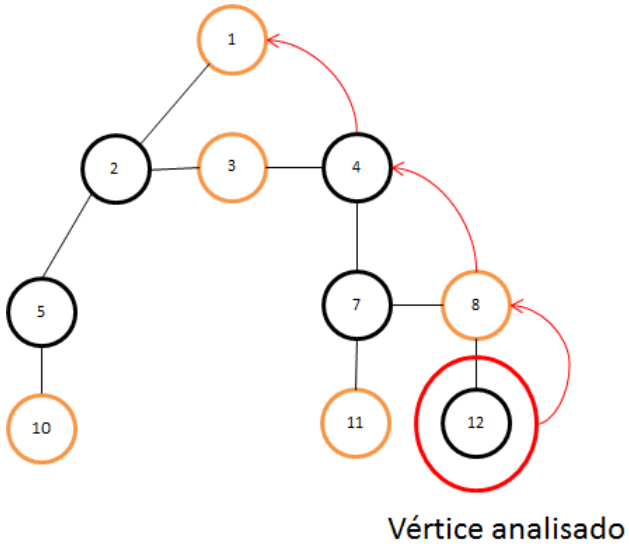


Figura 21 – Execução do algoritmo *poda_Arvore*, 12ª e última iteração.

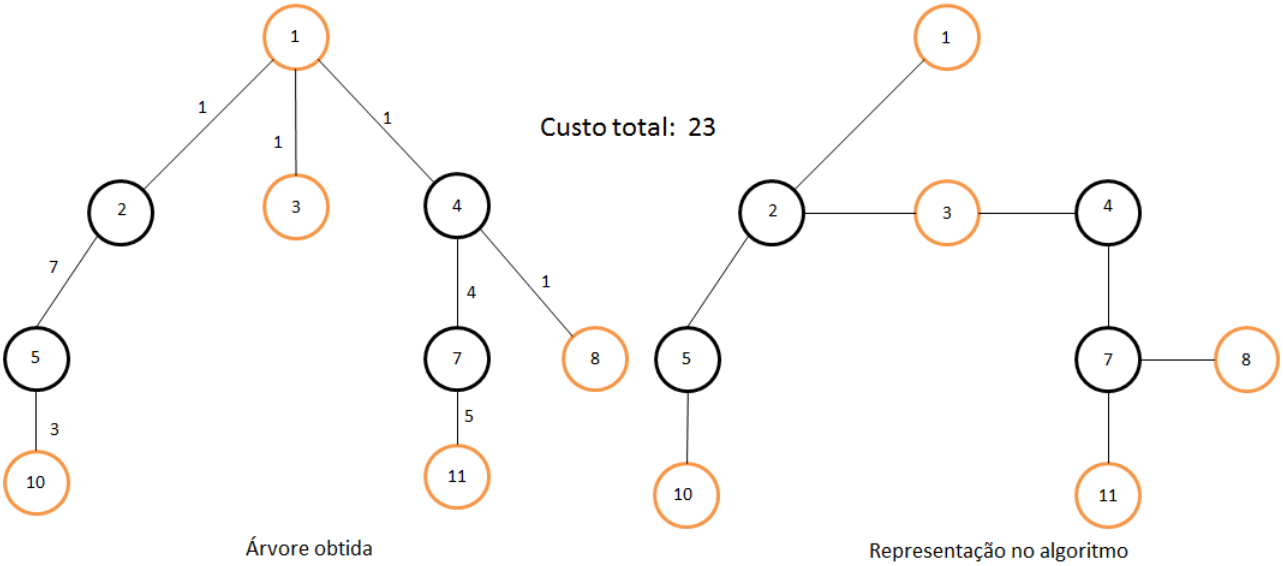


Figura 22 – Árvore resultante da operação de poda.

  - faça as atualizações necessárias referentes ao vértice v_{atual};
  - realize a operação $auxiliar \leftarrow v_{atual}$, $v_{atual} \leftarrow v_{atual}.$*irmão*;
  - delete *auxiliar*;

- Se v_{atual} não possuir nem filho nem irmão e está no grupo *multicast*, $v_{atual} \leftarrow v_{atual}.pai$;
- Se v_{atual} não possuir nem filho nem irmão e não está no grupo *multicast*:
  - faça as atualizações necessárias referentes ao vértice v_{atual};
  - realize a operação $auxiliar \leftarrow v_{atual}$, $v_{atual} \leftarrow v_{atual}.pai$;

– delete *auxiliar*;

- O processo executa o percurso em pós-ordem e repetirá até que alcance ao vértice raiz.

Retomando sobre a função $aplica\_Mudanca$, ela será explicada detalhadamente a seguir na Seção 3.1.1.

3.1.1 Função $aplica\_Mudanca$

A função $aplica\_Mudanca$ possui grande importância na execução deste trabalho. Ela é utilizada tanto durante a fase de construção quanto na busca local e sua organização é representada pelo pseudocódigo no Algoritmo 9. Como entrada do algoritmo, temos um grafo conexo com peso (G, W), os membros do grupo *multicast* gM, os nós que se encontram na árvore e um caminho C aqui representado como um vetor s-dimensional, com $s \geq 2$, de vértices consecutivos segundo o caminho, sendo o primeiro e último vértices, aqueles de grau 1.

Algoritmo 9 aplica_Mudanca

ENTRADA: (G, W), T, $V(T)$, gM, C
SAÍDA: uma árvore de Steiner $T = (V, F)$ de G atualizada e sua função peso $W(F)$

1 $v_p \leftarrow$ preparar_Para_Mudanca(v_f), tal que v_p representa o pai atual do vértice v_f;
2 **Se** $tamanho(C)$ == 2 **faça**
3 $v_f.pai \leftarrow p_n$;
4 $v_f.pesoAresta \leftarrow W_g(e)$, tal que $e \in E(G)$ e e possui representação em C;
5 $v_f.custo \leftarrow p_n.custo$ + $W_g(e)$;
6 Atribuir v_f como um dos filhos de p_n;
7 **Senão**
8 Insere os vértices do caminho C na árvore;
9 Ao adicionar o penúltimo vértice, u, do caminho C fazer:
10 $u.filho \leftarrow v_f$;
11 $v_f.pai \leftarrow u$;
12 $v_f.pesoAresta \leftarrow W_g(e)$;
13 $v_f.custo \leftarrow u.custo$ + $W_g(e)$;
14 **fim**
15 Caso necessário, realiza uma poda simples desalocando os antigos predecessores de no_Filho;

fim aplica_Mudanca

1. A função *preparar_Para_Mudanca* no Algoritmo 9 consiste em realizar atualizações necessárias em no_Filho e armazenar seu pai atual (Linha 1). As atualizações podem ser:

   - Se for filho único, atualizar $v_p.filho$ e $v_f.pai$ para NULL;
   - Se for o último filho na lista, $v_f.prevIrmao.proxIrmao$ e $v_f.prevIrmao$ para NULL;

- Se for o primeiro filho na lista, $v_p.filho \leftarrow v_f.proxIrmao$, $v_f.proxIrmao.prevIrmao$ e $v_f.proxIrmao$ para NULL;
- Se for o filho intermediário, $v_f.proxIrmao.prevIrmao \leftarrow v_f.prevIrmao$ e $v_f.prevIrmao.proxIrmao \leftarrow v_f.proxIrmao$.

2. Se a mudança for apenas uma mudança interna de arestas na árvore (Linha 2):

   - A aresta entre v_f e v_p já foi deletada na linha 1;
   - Atualizar custo de T considerando a mudança feita;
   - Adicionar a aresta entre v_f e o p_n e atualizar as informações necessárias de v_f (Linhas 3 a 6).

3. Se a mudança envolve a adição de novos vértices, $C > 2$, à árvore (Linha 7):

   - A aresta entre v_f e v_p já foi deletada na linha 1;
   - Adicionar os vértices de $C[2]$ a $C[s-1]$ na árvore (Linha 9). Para $C[1]$, $C[s]$ e $C[s-1]$ representam respectivamente p_n, v_f e u;
   - Adicionar a aresta entre v_f e o u e atualizar custo de T considerando a mudança feita (Linhas 9 a 14).

4. Por fim, realizar uma poda simples que consiste em: eliminar os vértices que não pertencem ao gM e que se tornaram folha devido à realocação de v_f.

A importância atribuída a esta função é consequência de sua aplicação nos Algoritmos 6 e 10. Seu uso nesses dois algoritmos implica que um erro nela pode acarretar na falha de todo o projeto. Para analisar a corretude do algoritmo, verificamos aqui se esta estratégia:

1. Não produzirá um ciclo na árvore;
2. Não irá tornar a árvore desconexa.

Para, isso iremos provar dois resultados, os Corolários 1, 2, tendo em vista o Teorema 1.

Teorema 1. Uma árvore com n vértices terá obrigatoriamente $n-1$ arestas.

A demonstração desse teorema não irá ser apresentada neste documento por ela estar presente na referência (BONDY; MURTY, 2008). Esse estudo foi realizado na disciplina de Teoria dos Grafos. Sua demonstração é por meio de indução fraca sobre o número de vértices do grafo.

A primeira demonstração do Corolário 1 e a terceira demonstração do Corolário 2 referem-se ao caso no qual $tamanho(C) = 2$ enquanto a segunda demonstração do Corolário 1

e a quarta demonstração do Corolário 2 referem-se à $tamanho(C) > 2$. Em ambas, consideramos que a as operações de retirada e de inserção de vértice à árvore implicam na retirada e inserção de uma sub-árvore com d vértices, com $d \geq 1$.

Proposição 1. Para o processo de remoção da aresta que liga o v_f ao v_p e a adição de nova aresta que o conectará ao p_n à árvore (linhas 2 a 7) não gera ciclos.

Demonstração. Suponhamos por absurdo que, para uma árvore T com n vértices, realocar um vértice por meio da estratégia do Algoritmo 9 irá inserir um ciclo em T.

Dado que T é uma árvore, ela é acíclica e conexa. Pelo Teorema 1, T possui, antes da realocação, $n-1$ arestas ($|V_T| = n, |E_T| = n-1$).

No primeiro passo retiramos a aresta que liga v_f com o vértice v_p e o próprio v_f de T. Dessa remoção, $|V_T| = n - d$ e $|E_T| = n - d - 1$.

O passo final é adicionar a aresta que ligue v_f ao p_n e repor v_f na árvore. Essa operação adiciona um vértice e uma aresta à árvore e nos fornece $|V_T| = n$ e $|E_T| = n-1$.

A suposição é de que a realocação de v_f acarretaria em um ciclo na árvore T. Para isso, precisaríamos que $|V_T| - |E_T| > 1$. Entretanto, temos que $|V_T| - |E_T| = n - (n-1) = 1$. Logo, chegamos a um absurdo. $\Longrightarrow\Longleftarrow$

□

Proposição 2. Para o processo de adição de novos vértices à árvore (linhas 8 a 15) a fim de realocar um vértice da árvore (v_f) devido à adição de novos vértices, o algoritmo não gera ciclos.

Demonstração. Sabemos que $C[1] \in A$ (por construção). Suponhamos por absurdo que, para uma árvore T com n vértices, realocar um vértice com adição de novos vértices por meio da estratégia do Algoritmo 9 irá inserir um ciclo em T.

Dado que T é uma árvore, ela é acíclica e conexa. Pelo Teorema 1, T possui, antes da realocação, $n-1$ arestas ($|V_T| = n, |E_T| = n-1$).

No primeiro passo retiramos a aresta que liga v_f com o vértice v_p e o próprio v_f de T. Dessa remoção, $|V_T| = n - d$ e $|E_T| = n - d - 1$.

Para o segundo passo, adicionamos os vértices de $C[2]$ a $C[s-1]$ na árvore. Para que sejam conexos à árvore, adicionamos a aresta que liga $C[2]$ a $C[1]$.

Visto que C é um caminho de s vértices, C possui $s-1$ arestas. Como são adicionados $s-2$ vértices mais suas $s-3$ arestas, temos que $|V_T| = n+s-d-2$ e $|E_T| = n+s-d-4$. Vale ressaltar a adição da aresta entre $C[2]$ a $C[1]$ e, portanto, $|V_T| = n+s-d-2$ e $|E_T| = n+s-d-3$.

O passo final é adicionar a aresta que ligue v_f a $C[s-1]$ e repor v_f na árvore. Essa operação adiciona um vértice e uma aresta à árvore e nos fornece $|V_T| = n+s-2$ *e* $|E_T| = n+s-3$.

A suposição é de que a realocação de no_Filho, com adição de novos vértices, acarretaria em um ciclo na árvore T. Para isso, precisamos que $|V_T| - |E_T| > 1$. Entretanto, temos que $|V_T| - |E_T| = (n+s-2) - (n+s-3) = 1$. Logo, chegamos a um absurdo. $\Longrightarrow\Longleftarrow$ □

Corolário 1. Quando um vértice é realocado para um novo pai, pelo Algoritmo 9, não há a geração de ciclos.

Para a demonstração do Corolário 1, basta considerar as Proposições 1 e 2.

Proposição 3. Para o processo de remoção da aresta que liga o v_f ao v_p e a adição de nova aresta que o conectará ao p_n à árvore (linhas 3 a 7) não deixa a árvore desconexa.

Demonstração. Suponhamos por absurdo que T' fica desconexa de T devido à realocação do seu vértice raiz de um pai nativo da árvore para outro também nativo dela.

No primeiro passo retiramos v_f, raiz de T', e a aresta que o ligava com v_p da árvore T.

O último passo consiste em adicionar a aresta ligando v_f ao vértice p_n e, em consequência, T' à árvore T. Sabemos que $p_n \in A$ e, portanto, $\exists$ um passeio que ligue p_n $\forall$ vértices em T. Devido a isso, chegamos que $\exists$ um passeio que ligue v_f a qualquer vértice em T e $\exists$ um passeio tal que para todo vértice em T', podemos alcançar qualquer vértice em T.

A suposição era de que T' ficaria desconexo de T devido à realocação do seu vértice raiz de um pai nativo da árvore para outro também nativo dela. Para isso, deveríamos chegar que não existe passeio que permitisse qualquer um vértice em T' alcançar outro em T. Como encontramos que existe um passeio tal que para qualquer vértice em T', podemos alcançar qualquer vértice em T chegamos a um absurdo. $\Longrightarrow\Longleftarrow$ □

Proposição 4. Para o processo de adição de novos vértices à árvore (linhas 8 a 15) a fim de realocar um vértice da árvore (v_f) devido à adição de novos vértices, o algoritmo não deixa a árvore desconexa.

Demonstração. Suponhamos por absurdo que T' fique desconexa de T devido à realocação do seu vértice raiz de um pai nativo da árvore para um recém-adicionado na mesma.

No primeiro passo retiramos v_f, raiz de T', e a aresta que o ligava com v_p da árvore T.

Para o segundo passo, conectamos o trecho de vértices de $C[2]$ a $C[s-1]$ do caminho C a T, por meio da adição da aresta entre $C[2]$ e $C[1]$ com $C[1] \in A$.

Como além de acíclico C é conexo, sabemos que $\exists$ um passeio P ligando $C[s-1]$ a $C[2]$. Sabemos também que $\exists$ um passeio P' ligando $C[2]$ a $C[1]$ e que $C[1] \in A$. Então temos

que P pode ser estendido para ligar $C[s-1]$ a $C[1]$ e $\exists$ um passeio P^* de $C[s-1]$ passando por todos os vértices de T.

O passo final consiste em adicionar a aresta ligando v_f ao vértice $C[s-1]$ e, em consequência, T' à árvore T. Como T' é uma árvore, obtemos que $\exists$ um passeio tal que $\forall$ vértices em T' é possível alcançar $C[s-1]$. Portanto, por transitividade, $\exists$ um passeio tal que $\forall$ vértices em T' podemos alcançar qualquer vértice em T.

A suposição era de que T' ficaria desconexo de T devido à realocação do seu vértice raiz de um pai nativo de T para um recém-adicionado na mesma. Para isso, deveríamos chegar que $\not\exists$ passeio algum que permitisse qualquer um vértice em T' alcançar outro em T. Como encontramos que $\exists$ um passeio tal que $\forall$ vértices em T' podemos alcançar qualquer vértice em T chegamos a um absurdo. $\Longrightarrow\Longleftarrow$ □

Corolário 2. Quando um vértice é realocado para um novo pai, pelo Algoritmo 9, a árvore não se torna desconexa.

Para a demonstração do Corolário 2, basta considerar as Proposições 3 e 4.

3.2 Busca Local

Para a busca local, no começo do projeto propomos o uso de uma Busca Tabu (GLOVER, 1989; GLOVER, 1990) com a qual tínhamos a intenção de obter de resultados próximos ao ótimo e em tempo computacional aceitável em relação às aplicações já existentes. Entretanto, com a definição das estruturas deste TCC e considerando o tamanho das instâncias alvo, o uso da Busca Tabu tornou-se inviável, pois é computacionalmente cara.

O sistema para armazenamento de movimentos tabu requisitaria uma grande parcela de memória do computador devido ao tamanho das instâncias. Em relação à estrutura utilizada, a Busca Tabu iria falhar por sua necessidade de algumas vezes aceitar pioras na solução para escapar de ótimos locais. A falha ocorre pelo fato de que, para evitar ciclos, foi determinado ao algoritmo que um vértice não poderia ser submetido a movimentos que causem o aumento do seu custo.

Tendo em vista as colocações sobre a Busca Tabu, o pseudocódigo da Busca Local proposta é apresentado no Algoritmo 10. Como entrada do algoritmo temos, respectivamente, um grafo conexo com peso (G, W_g), a solução T obtida na fase de construção, o grupo de nós que já se encontram na árvore $V(T)$ e os membros do grupo *multicast* mG.

O Algoritmo 10 possui a desvantagem de não conseguir escapar de um ótimo local caso encontre um, entretanto sua vantagem é a compatibilidade com a estrutura usada e a redução do consumo de memória se comparado à Busca Tabu. A Busca Local que pretendíamos implementar (foi implementada, porém, não conseguimos validar sua implementação) visita cada vértice

Algoritmo 10 busca_local

ENTRADA: (G, W_g), T, $V(T)$, gM
SAÍDA: uma árvore de Steiner $T = (V, F)$ de G ótima e sua função peso $W(F)$

```
1   faça
2       melhora ← false;
3       copia ← V(T);
4       Enquanto copia ≠ ∅ faça
5           v_atual ← pop(copia);
6           Se v_atual ≠ root.valor faça
7               Verifica vizinhança de v_atual;
8               Se ∃ um caminho C a partir de v_atual tal que reduza o custo de T faça
9                   aplica_Mudanca(v_f, C, p_n, V(T), gM, G, T, W_g(a_e), copia);
10              fim
11          fim
12          Se houve melhora: melhora ← true;
13      fim
14  Enquanto melhora = true
fim busca_local
```

pertencente à árvore (Linha 5) por meio da função *pop*[2] e, com estratégia gulosa, verifica se há alguma aresta adjacente a ele com peso menor que a entre ele e seu pai.

Caso exista uma aresta com peso inferior dentre o peso do vértice e de seu atual pai, são possíveis duas ocorrências: esta aresta ligar diretamente a um vértice já adicionado à árvore ou ligar a um vértice não incluso na mesma. Ambas as ocorrências são submetidas à $aplica\_Mudanca$ (Seção 3.1.1).

Para qualquer um dos dois casos, as informações só serão passados à $aplica\_Mudanca$ se $p_n.custo + W_g(a_e) < v_{atual}.custo$. Para o segundo caso, primeiro será feita uma busca por um caminho C tal que tenha pontas em v_{atual} e em outro vértice da árvore, não possua ciclos para então ser submetido à avaliação.

Por fim, o critério de parada para o algoritmo é a ocorrência de uma iteração sem melhora. Seja considerado o grafo presente na Figura 8, dada a árvore obtida a partir dele na fase de construção, Figura 22, segue o exemplo da execução do Algoritmo 10:

Considere o caso no qual falta ao algoritmo apenas verificar, respectivamente, os vértices 10 e 11. Para o vértice 10, o algoritmo ao verificar a vizinhança obtém que a aresta com menor custo é a que liga o vértice 10 ao vértice 6. Entretanto, o vértice 6 não se encontra na árvore e, em consequência, o algoritmo tenta conseguir um caminho que possua o vértice 6, tenha uma extremidade em 10 e outra em algum vértice pertencente à árvore. Para este caso, o caminho obtido é representado na Figura 23.

Com o uso do caminho apresentado na Figura 23 o custo do vértice 10 reduz de 11 para 8 e, portanto, esta alteração é submetida ao Algoritmo 9. Na Figura 24, a aresta em vermelho será

[2] A função *pop* realiza a remoção do elemento de uma pilha e o retorna como resposta.

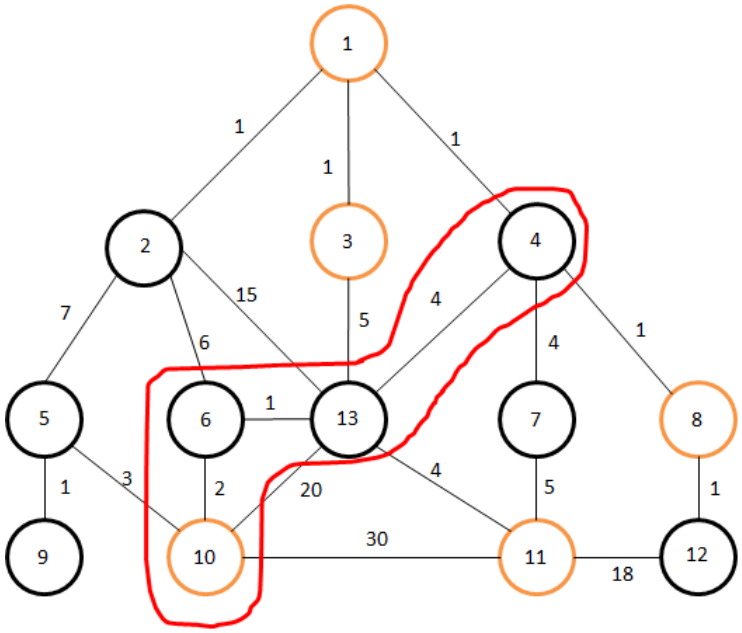


Figura 23 – Caminho encontrado.

retirada da árvore enquanto que em verde se encontra os vértices e arestas que serão adicionados à árvore proporcionando o resultado apresentado na Figura 25.

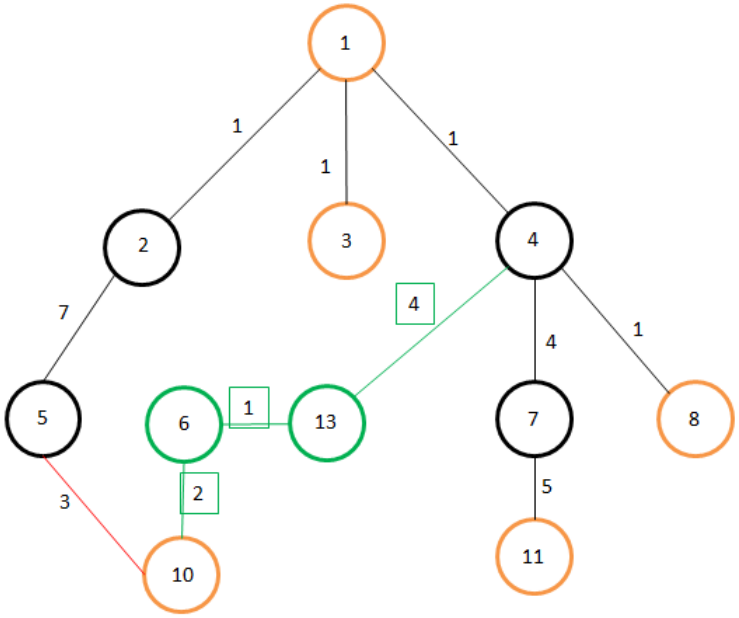


Figura 24 – Execução do Algoritmo 9.

Com a árvore representada na Figura 25, ao ser realizada a busca local sobre o vértice 11 será encontrada a aresta entre os vértices 11 e 13, Figura 26, como a com menor peso. O vértice 13 já se encontra na árvore, a alteração que transforma o vértice 11 em filho de 13 faz que o custo de 11 reduza de 10 para 9 e, portanto, esta alteração é submetida ao Algoritmo 9.

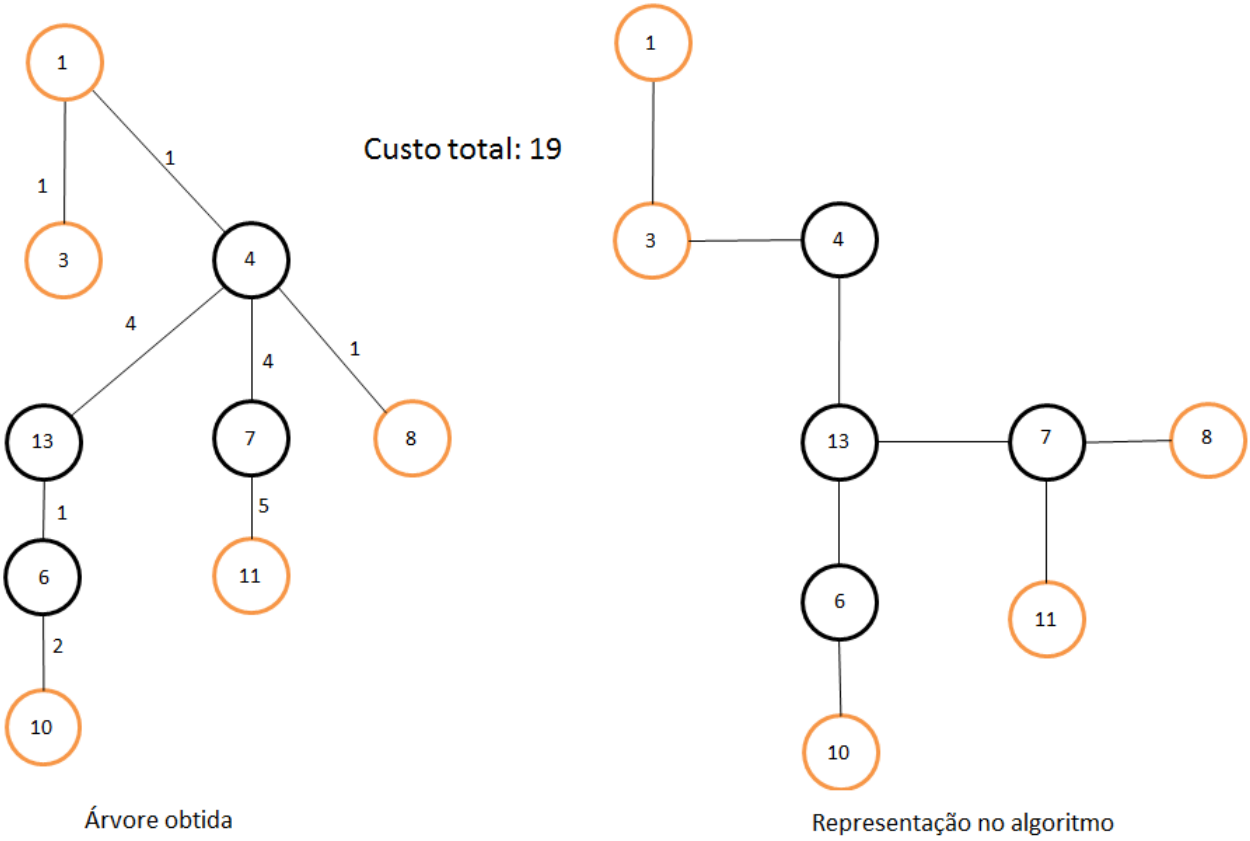


Figura 25 – Resultado da busca local sobre o vértice 10.

Na Figura 27 a aresta em vermelho será a eliminada enquanto que a em verde será adicionada à árvore. Vale ressaltar que, a toda operação de troca efetuada, o pais antigo do nó avaliado é submetido a um processo de poda.

Como o vértice 11 é o último vértice a ser analisado, o laço da linha 4 no Algoritmo 10 acaba. Devido à ocorrência de melhoras na solução, o algoritmo executa mais uma vez o laço da linha 1, conclui que não há mais melhorias possíveis e para de executar. Então, o retorno do Algoritmo 10 é a solução presente na Figura 28 que possui um custo inferior ao da solução obtida na fase de construção (Figura 22).

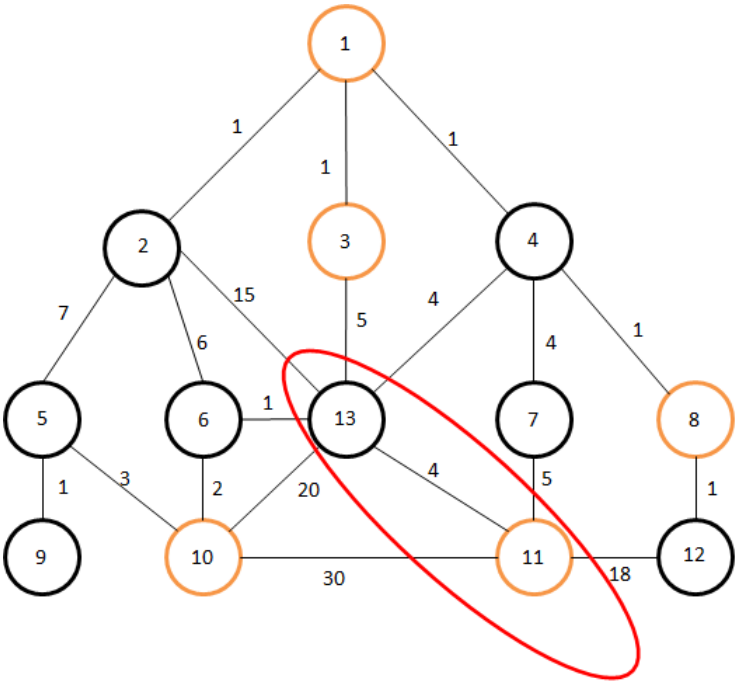


Figura 26 – Aresta selecionada.

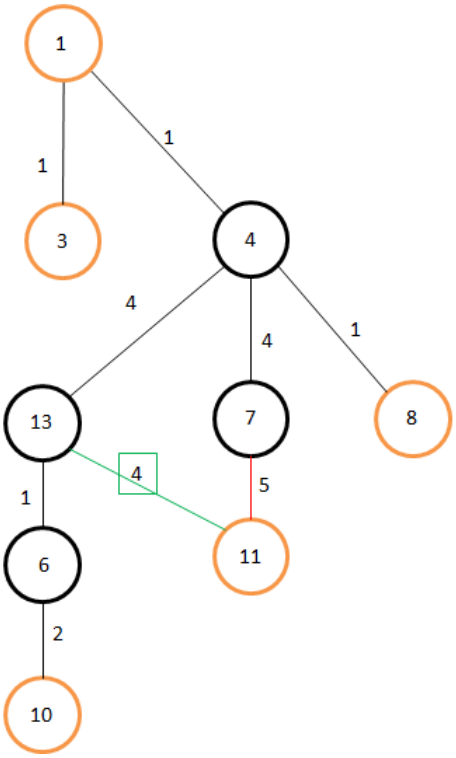


Figura 27 – Execução do Algoritmo 9.

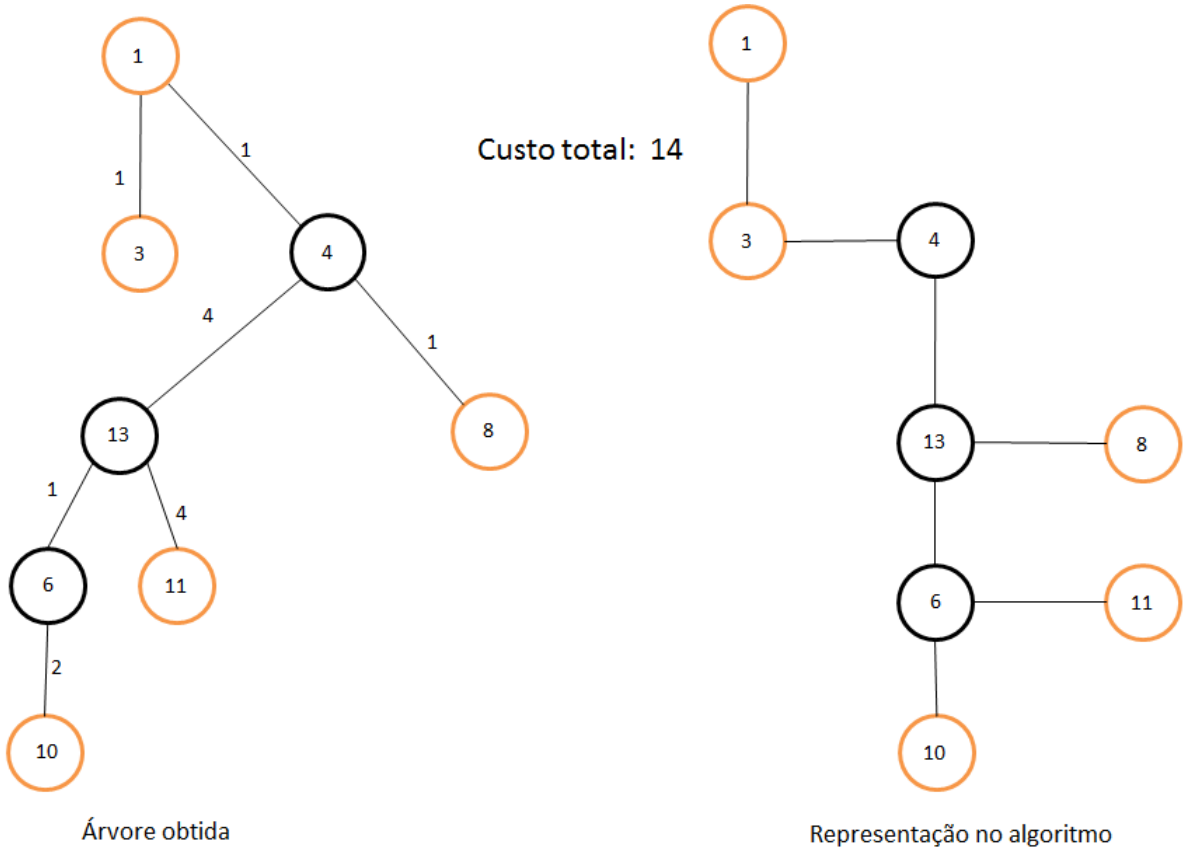


Figura 28 – Resultado da busca local sobre o vértice 11.

4 Conclusão

Este TCC trata do problema de roteamento *multicast* por meio da construção da árvore de Steiner. Conforme relatado no texto, esse problema pode ser abordado de diversas formas devido a algumas limitações relacionadas à tarefa de roteamento. Nesse sentido, para atenuar os problemas associados ao uso da rede, são necessários métodos que proporcionem um funcionamento eficiente. A eficiência dos métodos buscados pode ser interpretada por meio de conceitos fornecidos em trabalhos como o de Oliveira e Pardalos (2005).

Um limitante à obtenção de uma árvore com custo mínimo é a existência de muitos dispositivos conectados à rede de computadores e a constante mudança da mesma. Em consequência, métodos eficientes devem possuir baixa complexidade para que, para obter uma árvore de roteamento, não sejam necessários uma taxa de processamento ou tempo de execução muito elevados.

Neste trabalho, propomos a utilização do conceito de STPG pela meta-heurística GRASP. Consideramos uma adaptação do algoritmo que produz uma árvore geradora mínima, o algoritm de Jarnik-Prim a fim de compor a fase construtiva da heurística proposta. A estratégia de busca local consiste de movimentações do ramo da árvore caso essas movimentações produzam uma árvore de custo melhor do que a antes de ser atualizada. Esperávamos que com o algoritmo desenvolvido tivéssemos como resultado esperado a obtenção de uma árvore de Steiner com baixo custo, de boa qualidade para o roteamento *multicast*. A corretude do algoritmo proposto foi apresentada na Seção 3.1.1, apesar de experimentos computacionais não terem sido possíveis até o corrente momento.

O desenvolvimento apresentado na Seção 3 apresenta pontos que devem ser melhorados. Dentre esses pontos há um destaque especial à estrutura utilizada para representação das informações passadas ao algoritmo. Para desenvolvimentos futuros é necessária uma maneira mais flexível para representar as informações tratadas. Também sugerimos o desenvolvimento de uma estratégia que permita um uso menor da capacidade de memória da máquina utilizada e que, com um tempo computacional aceitável, obtenha resultados competitivos ao do estado da arte. Por fim, arrumar os erros presentes no algoritmo desenvolvido e deixar sua versão funcional disponível em (COSTA; NASCIMENTO, 2014).

4.1 Aprendizado do Aluno

Para este trabalho foi realizada a revisão bibliográfica, apresentada na Seção 2, sobre o STPG e sobre trabalhos que utilizaram heurísticas e meta-heurísticas para o tratamento do STPG. As heurísticas vistas nesses trabalhos possuem foco na contrução de uma MST e, as

meta-heurísticas vistas aplicam algumas das heurísticas aprendidas e em alguns casos buscam, através de outras heurísticas, uma melhora nas soluções obtidas.

Das heurísticas vistas, foram revisadas as de Prim (1957), Kruskal (1956) e Dijkstra (1959) já aprendidas antes nas disciplinas de Teoria dos Grafos e Algoritmo e de Estrutura de Dados. Foram aprendidas as heurísticas de Feng e Yum (1999) com uso para grafos direcionados e a de Kou, Markowsky e Berman (1981) que possui bons resultados para redes reais segundo o trabalho de Doar e Leslie (1993).

Sobre as meta-heurísticas, pelo trabalho de Esbensen (1995) foi revisto o tópico de Algoritmo Genético aprendido na disciplina de Inteligência Artificial enquanto que, foram aprendidas as meta-heurísticas GRASP (FEO; RESENDE, 1995) a Busca Tabu (GLOVER, 1989; GLOVER, 1990) e ambas foram estudadas com implementação serial e com implementação paralela.

De acordo com o obtido durante a revisão bibliográfica, na fase de desenvolvimento, foram aplicados os conhecimentos adquiridos sobre o GRASP e Busca Tabu assim como sobre Prim e Kruskal. Durante o desenvolvimento, foi observada a influência da estrutura escolhida, para representar as informações a serem trabalhadas, sobre a organização do algoritmo como ressaltado na Seção 4.

Com a estrutura utilizada neste trabalho, a representação do grafo recebido e do resultado processado ficou mais intuitiva. Entretanto, a mesma restringiu a manipulação das informações implicando, em parte, na mudança da escolha das estratégias tomadas para tratar o STPG. Em consequência disso, deve-se destacar como aprendizado a necessidade do planejamento considerando não apenas a interpretação mas também a aplicação direta no projeto.

Em suma, foram vistas heurísticas e meta-heurísticas aplicadas ao problema da Árvore de Steiner. Da fase de desenvolvimento, foi revisto o conhecimento da necessidade de um bom planejamento com foco na melhor organização do algoritmo a fim de evitar o aumento da complexidade na codificação do projeto.

Referências

AHARONI, E.; COHEN, R. Restricted dynamic steiner trees for scalable multicast in datagram networks. *IEEE/ACM Transactions on Networking (TON)*, IEEE Press, v. 6, n. 3, p. 286–297, 1998. Citado 2 vezes nas páginas 2 e 7.

BALDI, M.; OFEK, Y.; YENER, B. Adaptive real-time group multicast. In: IEEE. *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*. [S.l.], 1997. v. 2, p. 683–691. Citado na página 2.

BALLARDIE, T.; FRANCIS, P.; CROWCROFT, J. Core based trees (cbt). *ACM SIGCOMM Computer Communication Review*, ACM, v. 23, n. 4, p. 85–95, 1993. Citado na página 3.

BASTOS, M. P.; RIBEIRO, C. C. Reactive tabu search with path-relinking for the steiner problem in graphs. In: *Essays and surveys in metaheuristics*. [S.l.]: Springer, 2002. p. 39–58. Citado 5 vezes nas páginas 7, 15, 16, 17 e 18.

BEASLEY, J. E. *OR-Library*. 2014. [Accessed 2014-06-13]. Disponível em: <https://files-.nyu.edu/jeb21/public/jeb/info.html>. Citado 2 vezes nas páginas 13 e 16.

BEZENŠEK, M.; ROBIČ, B. A survey of parallel and distributed algorithms for the steiner tree problem. *International Journal of Parallel Programming*, Springer US, v. 42, p. 287–319, 2014. Citado 4 vezes nas páginas 2, 4, 7 e 13.

BIENSTOCK, D. et al. A note on the prize collecting traveling salesman problem. *Mathematical programming*, Springer, v. 59, n. 1-3, p. 413–420, 1993. Citado na página 2.

BONDY, J. A.; MURTY, U. *Graph theory, volume 244 of Graduate Texts in Mathematics*. [S.l.]: Springer, New York, 2008. Citado na página 36.

COSTA, C. R.; NASCIMENTO, M. C. V. *Problema da árvore de Steiner em Grafos para melhor tratar o roteamento multicast - Algoritmo*. 2014. [Accessed 2014-12-13]. Disponível em: <https://github.com/NivlacSeugirdor/SteinerTreeToMulticast.git>. Citado na página 45.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, Springer, v. 1, n. 1, p. 269–271, 1959. Citado 3 vezes nas páginas 9, 10 e 46.

DOAR, M.; LESLIE, I. How bad is naive multicast routing? In: IEEE. *INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*. [S.l.], 1993. p. 82–89. Citado 2 vezes nas páginas 11 e 46.

ESBENSEN, H. Computing near-optimal solutions to the steiner problem in a graph using a genetic algorithm. *Networks*, Wiley Online Library, v. 26, n. 4, p. 173–185, 1995. Citado 2 vezes nas páginas 13 e 46.

FENG, G.; YUM, T.-S. P. *Efficient multicast routing with delay constraints*. 1999. Citado 5 vezes nas páginas 11, 7, 9, 10 e 46.

FEO, T. A.; RESENDE, M. G. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, Elsevier, v. 8, n. 2, p. 67–71, 1989. Citado na página 2.

FEO, T. A.; RESENDE, M. G. Greedy randomized adaptive search procedures. *Journal of global optimization*, Springer, v. 6, n. 2, p. 109–133, 1995. Citado 2 vezes nas páginas 13 e 46.

GALLAGER, R. G.; HUMBLET, P. A.; SPIRA, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, ACM, v. 5, n. 1, p. 66–77, 1983. Citado na página 17.

GLOVER, F. Tabu search-part i. *ORSA Journal on computing*, INFORMS, v. 1, n. 3, p. 190–206, 1989. Citado 3 vezes nas páginas 15, 39 e 46.

GLOVER, F. Tabu search—part ii. *ORSA Journal on computing*, INFORMS, v. 2, n. 1, p. 4–32, 1990. Citado 3 vezes nas páginas 15, 39 e 46.

GLOVER, F.; KOCHENBERGER, G. A. *Handbook of metaheuristics*. [S.l.]: Springer, 2003. Citado na página 13.

HWANG, F. K.; RICHARDS, D. S.; WINTER, P. *The Steiner tree problem*. [S.l.]: Elsevier, 1992. Citado na página 2.

IMASE, M.; MAXMAN, B. M. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, SIAM, v. 4, n. 3, p. 369–384, 1991. Citado na página 2.

KARP, R. M. *Reducibility among combinatorial problems*. [S.l.]: Springer, 1972. Citado na página 4.

KOCH, T.; MARTIN, A.; Voß, S. *SteinLib: An Updated Library on Steiner Tree Problems in Graphs*. Takustr. 7, Berlin, 2000. Disponível em: <http://elib.zib.de/steinlib>. Citado 2 vezes nas páginas 5 e 15.

KOMPELLA, V.; PASQUALE, J.; POLYZOS, G. Two distributed algorithms for the constrained steiner tree problem. *Proc. Comput. Commun. and Netw., San Diego, CA*, 1993. Citado 5 vezes nas páginas 11, 2, 7, 17 e 18.

KOMPELLA, V. P.; PASQUALE, J. C.; POLYZOS, G. C. Multicast routing for multimedia communication. *IEEE/ACM Transactions on Networking (TON)*, IEEE Press, v. 1, n. 3, p. 286–292, 1993. Citado 2 vezes nas páginas 2 e 15.

KOU, L.; MARKOWSKY, G.; BERMAN, L. A fast algorithm for steiner trees. *Acta informatica*, Springer, v. 15, n. 2, p. 141–145, 1981. Citado 4 vezes nas páginas 11, 10, 12 e 46.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, JSTOR, v. 7, n. 1, p. 48–50, 1956. Citado 3 vezes nas páginas 11, 12 e 46.

MACULAN, N. The steiner problem in graphs. *Annals of Discrete Mathematics*, v. 31, p. 185–212, 1987. Citado 2 vezes nas páginas 2 e 7.

MARTINS, S. Greedy randomized adaptive search procedures for the steiner problem in graphs sl martins, pm pardalos, mgc resende. and cc riheiro. In: AMERICAN MATHEMATICAL SOC. *Randomization Methods in Algorithm Design: DIMACS Workshop, December 12-14, 1997*. [S.l.], 1999. v. 43, p. 133. Citado na página 16.

MARTINS, S. et al. A parallel grasp for the steiner tree problem in graphs using a hybrid local search strategy. *Journal of Global Optimization*, Springer, v. 17, n. 1-4, p. 267–283, 2000. Citado 7 vezes nas páginas 11, 12, 14, 15, 16, 17 e 18.

OLIVEIRA, C. A.; PARDALOS, P. M. A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research*, Elsevier, v. 32, n. 8, p. 1953–1981, 2005. Citado 8 vezes nas páginas 1, 2, 3, 4, 7, 9, 11 e 45.

PRIM, R. C. Shortest connection networks and some generalizations. *Bell system technical journal*, Wiley Online Library, v. 36, n. 6, p. 1389–1401, 1957. Citado 2 vezes nas páginas 2 e 46.

RESENDE, M.; RIBEIRO, C. Greedy randomized adaptive search procedures. In: GLOVER, F.; KOCHENBERGER, G. (Ed.). *Handbook of Metaheuristics*. [S.l.]: Kluwer Academic Publishers, 2002. p. 219–249. Citado na página 14.

SEDGEWICK, R. *Algorithms in C (part 5: Graph Algorithms)*. 3 ed.. ed. [S.l.]: Longman, 2002. Citado na página 8.

SKORIN-KAPOV, N.; KOS, M. The application of steiner trees to delay constrained multicast routing: a tabu search approach. In: IEEE. *Telecommunications, 2003. ConTEL 2003. Proceedings of the 7th International Conference on*. [S.l.], 2003. v. 2, p. 443–448. Citado 2 vezes nas páginas 2 e 15.

SKORIN-KAPOV, N.; KOS, M. A grasp heuristic for the delay-constrained multicast routing problem. *Telecommunication Systems*, Springer, v. 32, n. 1, p. 55–69, 2006. Citado 3 vezes nas páginas 2, 15 e 26.

TANENBAUM, A. S. *Redes de Computadores*. trad. 5 ed. São Paulo: Pearson, 2011. Citado na página 1.

WALL, D. W. Mechanisms for broadcast and selective broadcast. Stanford University, 1980. Citado na página 2.