CHARLES MASKE

CONSTRUÇÃO PARALELA DE ÁRVORES DE CORTES UTILIZANDO CONTRAÇÕES DE GRAFO OTIMIZADAS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Elias P. Duarte Jr.
Co-Orientador: Prof. Dr. Jaime Cohen

CURITIBA

2015

M397c

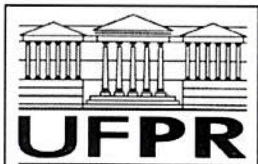
Maske, Charles
Construção paralela de árvores de cortes utilizando contrações de grafo otimizadas/ Charles Maske. – Curitiba, 2015.
48 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2015.

Orientador: Elias P. Duarte Jr. – Co-orientador: Jaime Cohen.
Bibliografia: p. 46-48.

1. Árvores (Teoria dos grafos). 2. Algoritmo - Otimização. 3. Redes de computadores. I. Universidade Federal do Paraná. II.Duarte Jr., Elias P.. III. Cohen, Jaime . IV. Título.

CDD: 511.52



Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Charles Maske, avaliamos o trabalho intitulado, "Construção Paralela de Árvores de Corte Utilizando Contrações de Grafos Otimizadas", cuja defesa foi realizada no dia 14 de dezembro de 2015, às 18:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

(X)**aprovação** do candidato. ( )**reprovação** do candidato.

Curitiba, 14 de dezembro de 2015.

Prof. Dr. Elias Procópio Duarte Jr.
PPGInf – Orientador

Prof. Dr. Jaime Cohen
Coorientador

Prof. Dr. Murilo Vicente Gonçalves da Silva
UTFPR – Membro Externo

Prof. Dr. Renato Carmo
PPGInf – Membro Interno

SUMÁRIO

RESUMO **ii**

ABSTRACT **iii**

1 INTRODUÇÃO **1**

2 ÁRVORES DE CORTE **4**

2.1 Definições Preliminares 4

2.2 Árvores de Cortes 7

2.3 O Algoritmo de Gomory-Hu 8

2.4 O Algoritmo de Gusfield 13

2.5 Conclusão 15

3 O ALGORITMO PARALELO DE GOMORY-HU **17**

3.1 Uma Versão Paralela do Algoritmo de Gomory-Hu Utilizando MPI 18

3.2 Uma Implementação do Algoritmo de Gomory-Hu Utilizando a Biblioteca Boost 25

3.3 Resultados Experimentais 26

4 CONSTRUÇÃO PARALELA DE ÁRVORES DE CORTES UTILIZANDO CONTRAÇÕES DE GRAFO OTIMIZADAS **33**

4.1 Especificação da Estratégia Proposta 33

4.2 Resultados Experimentais: Algoritmo Paralelo de Gomory-Hu 37

4.3 Algoritmo Híbrido Utilizando Contrações Otimizadas 40

4.4 Resultados Experimentais: Algoritmo Híbrido Utilizando Contrações Otimizadas 41

5 CONCLUSÃO **44**

REFERÊNCIAS BIBLIOGRÁFICAS **48**

RESUMO

As árvores de cortes representam, de forma compacta, a aresta conectividade entre todos os pares de vértices de um grafo com pesos nas arestas. Existem muitas aplicações de árvores de cortes como, por exemplo, em projetos de redes confiáveis, partição de grafos, análise de redes, segmentação de imagens entre outras. Dois algoritmos clássicos para construção de árvores de cortes são conhecidos: o algoritmo de Gomory-Hu e o algoritmo de Gusfield. Neste trabalho são apresentadas versões paralelas de um dos algoritmos clássicos para a construção de árvores de cortes, o algoritmo de Gomory-Hu. Este algoritmo faz múltiplas chamadas a um procedimento que encontra um corte de arestas de capacidade mínima entre dois vértices. Para encontrar os cortes mínimos, o algoritmo faz contrações de vértices do grafo de entrada. O procedimento para construção do grafo contraído é custoso e implementá-lo de forma eficiente não é trivial. Portanto é importante investigar pontos que podem ser otimizados nesse procedimento. A principal contribuição deste trabalho é a especificação de uma estratégia paralela baseada no modelo mestre-escravo que permite que processos aproveitem instâncias de grafos contraídos de passos anteriores. A otimização tem o objetivo de viabilizar uma forma eficiente de realizar as operações de contração nos processos escravos, já que estes sempre calculam o grafo contraído sobre o mesmo grafo. De forma geral, a implementação dessa otimização requer que o processo mestre tenha controle sobre as tarefas que foram enviadas para cada escravo, armazenando-as para que sejam utilizadas quando as respostas chegarem. Os processos escravos, por sua vez, precisam tomar decisões sobre quando aproveitar ou não, uma instância de grafo contraído. É também apresentada a aplicação da otimização proposta para um algoritmo híbrido que combina características do algoritmo de Gomory-Hu e do algoritmo de Gusfield, também baseado em contrações otimizadas. Para avaliar a eficiência desta versão foram realizados experimentos em um *cluster* de alto desempenho. Foram realizados testes de *speedup* e comparativos entre as versões paralelas existentes. Foram realizados também, experimentos com uma implementação do algoritmo paralelo de Gomory-Hu utilizando o conjunto de bibliotecas Boost para avaliar o desempenho de diferentes algoritmos de fluxo máximo (utilizados para calcular os s-t cortes mínimos).

ABSTRACT

Cut trees represent the edge-connectivity between all pairs of nodes of an undirected weighted graph. There are many cut tree applications, such as the design of reliable networks, graph partitioning, network analysis, image segmentation, among others. Two classical algorithms that solve the problem of finding a cut tree of an undirected weighted graph are known: the Gomory-Hu algorithm and the Gusfield algorithm. This work presents parallel versions of the Gomory-Hu algorithm. This algorithm makes multiple calls to a procedure which finds a cut with minimum capacity between a pair of vertices. At each step the algorithm apply contractions on the input graph. The contraction procedure is costly and its implementation is not trivial. Therefore it is important to investigate ways to optimize that procedure. Previous implementations of the algorithm build the contracted graph from the input graph. The main contribution of this work is the specification of a parallel strategy that allows processes to take advantage of instances of contracted graphs from previous steps. The optimization requires the master process to have control over the tasks that were sent to each slave, storing them so they are used when the answers arrive. The slave processes, in turn, must make decisions about when to use or not a contracted graph instance. It is also shown the application of the proposed optimization in a hybrid algorithm that combines features from Gomory-Hu and Gusfield algorithms. To evaluate the efficiency of the algorithms, experiments were performed on a high performance computer *cluster*. *Speedups* and comparison tests were performed against previous parallel implementations. Experiments were also carried out with an implementation of a parallel version of Gomory-Hu algorithm using Boost library to evaluate the performance of different maximum flow algorithms used to compute the minimum s-t cuts.

CAPÍTULO 1

INTRODUÇÃO

As árvores de cortes são estruturas combinatórias que representam, de forma compacta, a aresta conectividade entre todos os pares de vértices de um grafo [25]. A aresta conectividade entre um par de vértices s e t é a capacidade de um s-t corte mínimo. As árvores de cortes são utilizadas na solução de importantes problemas combinatórios incluindo roteamento [13], particionamento e agrupamento em grafos [15] além da análise de redes complexas, incluindo redes formadas por dados biológicos [28], redes sociais [22], entre outras.

Existem dois algoritmos sequenciais clássicos para encontrar a árvore de cortes de um grafo com capacidades nas arestas: o algoritmo de Gomory e Hu [19] e o algoritmo de Gusfiled [21]. Ambos os algoritmos são similares, já que fazem chamadas a um procedimento que calcula subproblemas de cortes mínimos entre pares de vértices $n - 1$ vezes para um grafo com n vértices. A principal diferença entre os dois algoritmos é o fato de que o algoritmo de Gusfield calcula os $n - 1$ cortes mínimos sobre o grafo de entrada e o algoritmo de Gomory-Hu realiza contrações e calcula os cortes mínimos sobre instâncias de grafos contraídos. O algoritmo de Gomory-Hu também exige a manipulação de estruturas de dados não triviais para controlar a construção da árvore de cortes e dos grafos contraídos.

Versões paralelas para os algoritmos de Gusfield e Gomory-Hu foram apresentadas, respectivamente, em [11] e [10]. Além disso, em [9] foi apresentado um algoritmo denominado híbrido que combina características de ambos os algoritmos de Gusfield e Gomory-Hu. De maneira geral, essas versões paralelas dos algoritmos para construção de árvores de cortes utilizam a arquitetura mestre-escravo para paralelizar os cálculos dos s-t cortes mínimos. O processo mestre distribui tarefas com pares de vértices s e t para os processos escravos. Estes, por sua vez, resolvem subproblemas de s-t cortes mínimos e enviam a resposta ao processo mestre. O processo mestre, ao receber as respostas dos processos escravos, atualiza a árvore e envia novas tarefas. O algoritmo termina com a árvore de cortes construída quando não existirem mais vértices a serem separados pelo procedimento de s-t

corte mínimo. Experimentos anteriores com versões paralelas do algoritmo de Gomory-Hu mostraram que as construções do grafos contraídos demandam muito tempo, chegando a consumir, em muitos casos, a metade do tempo de execução [9]. Implementar de maneira eficiente este procedimento no algoritmo paralelo de Gomory-Hu não é trivial, uma vez que os escravos não têm acesso a estruturas de dados compartilhadas.

Este trabalho propõe uma estratégia eficiente para realizar as contrações na versão paralela do algoritmo de Gomory-Hu bem como no algoritmo híbrido, uma vez que foi constatado, através de experimentos preliminares, que elas consomem quase metade do tempo de execução do algoritmo. A ideia principal é fazer com que os processos escravos aproveitem instâncias de grafos contraídos de passos anteriores sem a necessidade de refazer todas as contrações no grafo de entrada na execução de cada tarefa. De forma geral, a implementação dessa otimização requer que o processo mestre tenha controle sobre as tarefas que foram enviadas para cada escravo, armazenando-as para que sejam utilizadas quando as respostas chegarem. Os processos escravos, por sua vez, precisam tomar decisões sobre quando aproveitar, ou não, uma instância de grafo contraído.

Para avaliar a eficiência desta versão foram realizados experimentos em um *cluster* de alto desempenho. Foram realizados testes de *speedup* e comparativos entre as versões paralelas existentes. Os experimentos foram feitos em vários conjuntos de instâncias que incluíram grafos sintéticos e grafos reais. Os resultados demonstram *speedups* próximos de lineares para a maioria das instâncias. Nas comparações é possível comprovar os ganhos obtidos com a estratégia proposta, principalmente para os grafos onde o algoritmo consegue encontrar cortes mínimos balanceados que produzem grafos contraídos de tamanhos reduzidos. Outra contribuição deste trabalho é uma implementação do algoritmo paralelo de Gomory-Hu utilizando o conjunto de bibliotecas Boost[1]. Utilizando esta implementação foram realizados experimentos que possibilitaram comparar o desempenho do uso de diferentes algoritmos de fluxo máximo (utilizados para calcular os s-t cortes mínimos) dentro do algoritmo de Gomory-Hu.

O restante deste trabalho está organizado da seguinte forma. O capítulo 2 apresenta definições preliminares, inclusive das árvores de cortes, além do algoritmo de Gomory-Hu. No capítulo 3 é mostrada a versão paralela do algoritmo de Gomory-Hu apresentada no trabalho de Cohen et al. [10], além da implementação utilizando a biblioteca Boost. No

[1]Boost C++ Libraries, `www.boost.org`

capítulo 4 é apresentada a proposta de otimização para a versão paralela do algoritmo de Gomory-Hu. O capítulo 5 conclui o trabalho.

CAPÍTULO 2

ÁRVORES DE CORTE

Este capítulo apresenta definições preliminares incluindo as árvores de cortes, bem como os algoritmos clássicos para a sua construção. Na seção 2.1 são dadas definições básicas de grafos, seguidas da definição de árvore de cortes na seção 2.2. O algoritmo de Gomory-Hu é descrito na seção 2.3 e o algoritmo de Gusfield na seção 2.4. Na seção 2.5 são apresentadas conclusões a respeito dos dois algoritmos para calcular a árvore de cortes de um grafo.

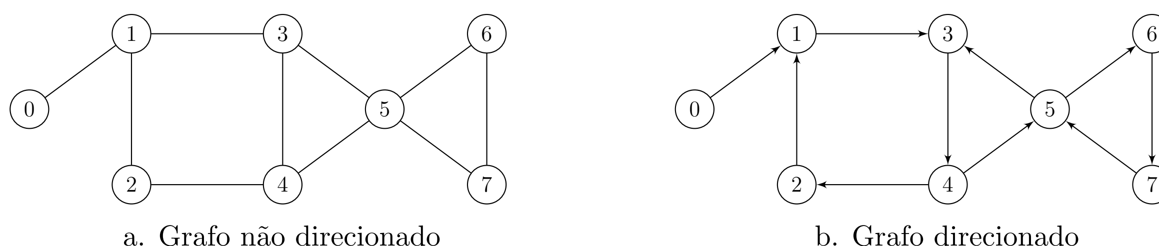
2.1 Definições Preliminares

Esta seção contém definições preliminares de teoria dos grafos que permitem a definição das árvores de cortes na próxima seção.

Definição 1 *Um grafo $G = (V, E)$ consiste de um conjunto de vértices V e um conjunto E de pares de vértices. Os elementos de E são chamados de arestas. Um grafo é dirigido ou direcionado se as arestas são pares ordenados de vértices.*

A figura 2.1a ilustra um exemplo de grafo. O conjunto de vértices é $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$ e o conjunto de arestas é $E = \{\{0, 1\}, \{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}, \{5, 6\}, \{5, 7\}, \{6, 7\}\}$. A figura 2.1b ilustra um exemplo de grafo direcionado. O conjunto de vértices é $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$ e o conjunto de arestas é $E = \{(0, 1), (1, 3), (2, 1), (3, 4), (4, 2), (4, 5), (5, 3), (5, 6), (6, 7), (7, 5)\}$.

Definição 2 *Uma aresta $e = \{u, v\}$ é* ***incidente*** *aos vértices u e v. Os vértices u e v por sua vez são* ***adjacentes*** *entre si e são chamados de extremos da aresta e.*



a. Grafo não direcionado

b. Grafo direcionado

Figura 2.1: Exemplos de grafo.

Como exemplo, conforme ilustra a figura 2.1a, a ligação entre os vértices 0 e 1 forma uma aresta portanto eles são adjacentes entre si.

Definição 3 *Uma sequência de vértices distintos* $(x_1, x_2, x_3, \ldots, x_{k+1})$ *tal que* $\{x_i, x_{i+1}\} \in E$ *é definida como um* ***caminho*** p *de tamanho* k *entre* x_1 *e* x_{k+1}.

Por exemplo, a sequência de vértices $(0, 1, 3, 4, 5)$ ou o conjunto de arestas $\{\{0, 1\}, \{1, 3\}, \{3, 4\}, \{4, 5\}\}$, ilustrados na figura 2.1a, formam um caminho de tamanho quatro entre os vértices 0 e 5.

Definição 4 *Um* ***ciclo*** *é um caminho de tamanho* $k \geq 3$ *tal que* $x_1 = x_{k+1}$.

Por exemplo, o caminho $\{1, 2, 4, 3, 1\}$, ilustrado na figura 2.1a, corresponde a um ciclo.

Definição 5 *Um grafo* G *é dito* ***conexo*** *se para todo par de vértices* $s, t \in G$*, existe um caminho entre* s *e* t.

Por exemplo, na figura 2.1a é possível encontrar caminhos entre todos os pares de vértices, portanto o grafo é conexo.

Definição 6 *Um grafo é uma* ***árvore*** *se for conexo e não possuir ciclos.*

A figura 2.2 ilustra uma árvore.

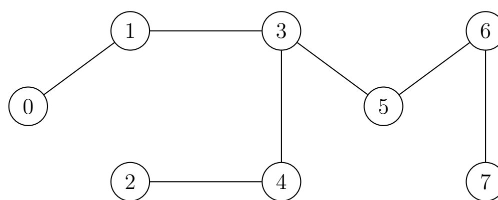


Figura 2.2: Grafo conexo sem ciclos ou árvore.

Definição 7 *Um grafo capacitado é um grafo* $G = (V, E)$ *associado a uma função* $c(u, v)$ *definida como um mapeamento de* $E \to \mathbb{Q}^+$ *chamado de* ***capacidade*** *da aresta* $\{u, v\}$.

A figura 2.3 ilustra um grafo capacitado. Note que o conjunto de vértices e arestas é o mesmo ilustrado na figura 2.1 porém para cada aresta $e \in E$ a função $c(u, v)$ define a capacidade entre os extremos u, v. Como exemplo $c(0, 1) = 2$.

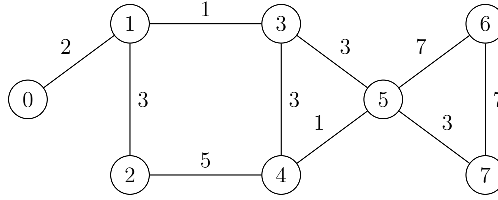


Figura 2.3: Exemplo de uma rede.

Definição 8 *Seja um grafo $G = (V, E)$ que contém dois vértices especiais: o vértice s que será chamado de origem e o vértice t que será chamado de destino. Considere dois subconjuntos complementares $S \subset V$ e $\overline{S} \subset V$ onde $S \cap \overline{S} = \emptyset$ e $S \cup \overline{S} = V$. O conjunto $(S, \overline{S})$ denota o conjunto das arestas com uma das extremidades em S e a outra extremidade em $\overline{S}$ e é chamado de* ***s-t corte*** *que separa os vértices s e t tal que $s \in S$ e $t \in \overline{S}$.*

Tome o grafo ilustrado na figura 2.3 e assuma o vértice $s = 1$ e o vértice $t = 7$. Podemos definir um *s-t corte* através dos conjuntos $S = \{0, 1, 2\}$ e $\overline{S} = \{3, 4, 5, 6, 7\}$. O conjunto de arestas que possuem extremidade em S e $\overline{S}$ é $(S, \overline{S}) = \{\{1, 3\}, \{2, 4\}\}$, logo a remoção destas arestas separa os vértices s, t, isto é, não haverá caminhos entre s e t.

Definição 9 *A* ***capacidade do corte*** *$(S, \overline{S})$ é definida como $c(S, \overline{S}) = \sum_{\{s,t\} \in (S, \overline{S})} c(s, t)$.*

Por exemplo, na Figura 2.3, a capacidade do corte $(S, \overline{S}) = \{\{1, 3\}, \{2, 4\}\}$ é dada como a soma das capacidades das arestas $e \in (S, \overline{S})$, portanto $c(S, \overline{S}) = 6$.

Definição 10 *Dentre todos os cortes $(S, \overline{S})$ possíveis que separam s e t, aqueles de capacidade mínima são chamados de* ***s-t cortes mínimos****.*

É possível enumerar mais de um corte que separa os vértices 1 e 7 do grafo na Figura 2.3. Um corte de capacidade mínima (que não é único) é o corte $(\{1, 3\}, \{1, 2\})$ cuja capacidade é 4. Ainda podemos notar na figura 2.3 que há pelo menos mais um corte que separa os vértices 1 e 7 e que possui capacidade 4: $(\{3, 5\}, \{4, 5\})$.

O problema de encontrar um *s-t* corte mínimo é resolvido utilizando algoritmos de fluxo máximo entre dois vértices. Por isso, definimos abaixo o problema do fluxo máximo.

Definição 11 *Considere um grafo orientado $G = (V, E)$ no qual cada aresta $\{u, v\} \in E$ tem uma capacidade não negativa $c(u, v) \geq 0$. A capacidade representa a quantidade*

máxima de ***fluxo*** *que pode passar por uma aresta. Seja* $s \in V$ *a origem e* $t \in V$ *o destino do fluxo. Um* ***fluxo*** *em* G *é uma função* $f : V \times V \to \mathbb{R}$ *que satisfaz as seguintes propriedades [12]:*

- ***Restrição de capacidade:*** $\forall (u, v) \in E$ *é necessário que* $0 \leq f(u, v) \leq c(u, v)$
- ***Conservação de fluxo:*** $\forall u \in V \setminus \{s, t\}$ *é necessário que* $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$. *Quando* $\{u, v\} \notin E$ *não haverá fluxo de* u *para* v *e* $f(u, v) = 0$

O valor $|f|$ de um fluxo f é definido como: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$. Note que se as arestas de um s-t corte forem removidas então $|f| = 0$.

Definição 12 *Seja* $G = (V, E)$ *um grafo capacitado. Uma* ***contração*** *de vértices* $v_1, v_2, \ldots, v_k$ *de* G *em único vértice* $\overline{v}$*, chamado super nodo, consiste em:*

1. *substituir em* G *os vértices* $v_1, v_2, \ldots, v_k$ *pelo super nodo* $\overline{v}$
2. *substituir, para todo vértice* v_i*,* $i = \{1, \ldots, k\}$*, as arestas* (v_i, u) *onde* $u \in V$ *é um vértice não contraído, pela aresta* $(\overline{v}, u)$ *cuja capacidade* $c(\overline{v}, u) = \sum c(v_i, u)$

Definição 13 *Um grafo capacitado que possui um super nodo é chamada de* ***grafo contraído****.*

2.2 Árvores de Cortes

Considere o problema de descobrir um *s-t corte mínimo* entre cada par de vértices de um grafo. A solução trivial é calcular os *s-t cortes mínimos* entre todas as $\binom{|V|}{2}$ combinações de vértices dois a dois. Gomory e Hu [19] mostram que existem exatamente $|V| - 1$ cortes que incluem ao menos um corte mínimo entre cada par de vértices do grafo. Esses $|V| - 1$ cortes são representados sucintamente através de uma árvore de cortes.

Definição 14 *Uma árvore de cortes de um grafo capacitado* $G = (V, E, c)$ *é uma árvore capacitada* $T = (V, E_T, c)$ *tal que para todo par de vértices* s *e* t *distintos:*

- *as capacidade dos s-t cortes mínimos de* T *e* G *são as mesmas, e*
- *os s-t cortes mínimos de* T *induzem s-t cortes mínimos em* G

Uma *árvore de cortes* é uma estrutura combinatória que representa a aresta conectividade entre todos os pares de vértices em um grafo com peso nas arestas. São dois os algoritmos clássicos para construção de árvores de cortes, o algoritmo de Gomory-Hu que é apresentado na seção 2.3 e o algoritmo de Gusfield que é apresentado na seção 2.4.

2.3 O Algoritmo de Gomory-Hu

Gomory e Hu [19] mostram que um grafo capacitado G com n vértices possui exatamente $n-1$ cortes que incluem ao menos um corte mínimo entre cada par de vértices do grafo. Esses cortes são representados por uma árvore de cortes.

O Algoritmo 1 traz o pseudocódigo do algoritmo de Gomory-Hu. Para auxiliar a descrição do algoritmo, o termo *nodo* refere-se à representação de vértices da árvore em construção.

Algoritmo 1: Algoritmo de Gomory-Hu

```
   Entrada: G = (V, E_G, c_G) um grafo não dirigido e com peso nas arestas
   Saída: T = (V, E_T, c_T) uma árvore de cortes de G
 1 T ← (V_T = {V}, E_T = ∅);
 2 enquanto ∃X ∈ V_T tal que |X| > 1 faça
 3     Sejam V_1, V_2, ··· , V_k vértices das componentes conexas de T \ X;
 4     X_i ← ⋃_{V'∈V_i} V', para 1 ≤ i ≤ k;
 5     G' ← G \ X_1, X_2, ··· , X_k;
 6     Seja s, t ∈ X;
 7     {S, S̄} ← s-t-corte mínimo de G';
 8     X_s ← X ∩ S;
 9     X_t ← X ∩ S̄;
10     e ← {X_s, X_t};
11     c(e) ← c(S, S̄);
12     para cada aresta e' = {X, Y} ∈ E_T incidente em X na árvore T faça
13         se Y ⊆ S então
14             E_T ← E_T ∪ {{X_s, Y}} \ {{X, Y}};
15         senão
16             E_T ← E_T ∪ {{X_t, Y}} \ {{X, Y}};
17     V_T ← (V_T \ {X}) ∪ {X_s, X_t};
18     E_T ← E_T ∪ {e};
19 retorna T
```

Seja $G = (V_G, E_G, c_G)$ o grafo de entrada com peso nas arestas. O algoritmo inicia a árvore $T = (V_T, E_T, c_T)$ com um nodo representando todos os vértices do grafo de entrada tal que $V_T = V_G$ e $E_T = \emptyset$. A cada passo do algoritmo, escolhe-se um nodo $X \in V_T$, designado como *pivô*, tal que $|X| \geq 2$. Os vértices de G associados aos nodos de cada

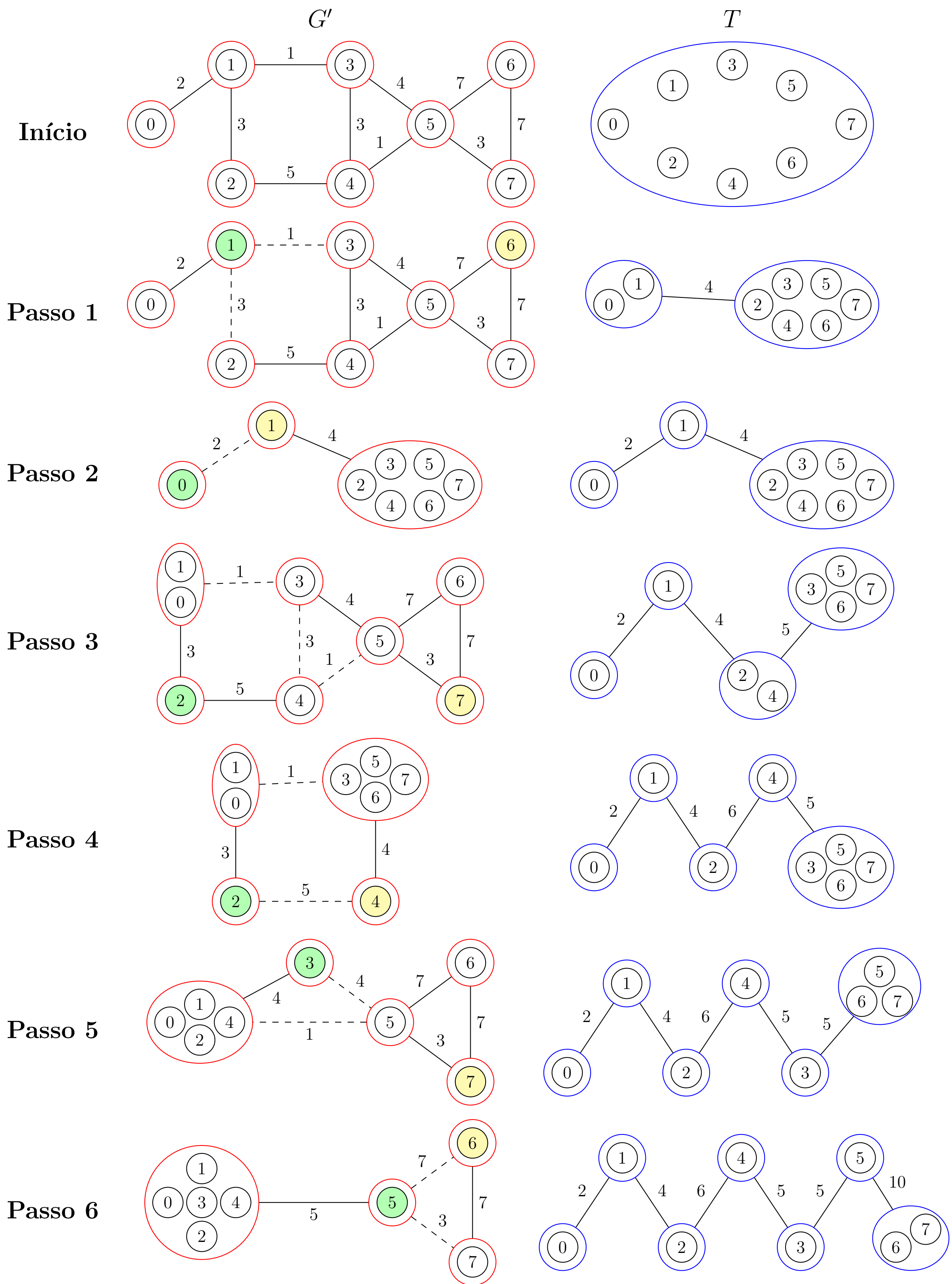
componente conexa de $T \backslash X$ são contraídos em G. O grafo obtido dessas contrações é denotado por $G' = G \setminus X_1, X_2, \cdots, X_k$, onde $X_1, X_2, \cdots, X_k$ são os conjuntos de vértices contraídos.

A construção do grafo contraído, na linha 5, consiste em percorrer a lista de arestas do grafo de entrada e determinar, para cada uma delas, se existe uma aresta correspondente no grafo contraído G'.

Em seguida, o algoritmo escolhe dois vértices $s, t \in X$ e calcula o corte mínimo entre s, t sobre o grafo contraído G' obtendo a partição $\{S, \overline{S}\}$ e o valor do corte $c(\{S, \overline{S}\})$. São criados dois novos nodos X_s, X_t na árvore T tal que $X_s = \{X \cap S\}$ e $X_t = \{X \cap \overline{S}\}$ e uma nova aresta $e = \{X_s, X_t\}$ com $c(e) = c(\{S, \overline{S}\})$. Para cada aresta $e' = \{X, Y\} \in E_T$ incidente em X na árvore T, verificamos em qual lado do *s-t-corte mínimo* $\{S, \overline{S}\}$ os vértices pertencentes ao nodo Y estão. Caso estejam no lado S então liga-se o nodo X_s a Y formando a aresta $\{X_s, Y\}$ caso contrário liga-se o nodo X_t a Y formando a aresta $\{X_t, Y\}$. O algoritmo atualiza a árvore de tal forma que o conjunto de vértices seja $V_T = (V_T \setminus \{X\}) \cup \{X_s, X_t\}$ e o conjunto de arestas seja $E_T = E_T \cup \{e\}$. Em seguida o algoritmo escolhe um novo nodo *pivô* $X \in V_T$ tal que $|X| > 1$. Um novo par de vértices $s, t \in X$ é aleatoriamente escolhido e é computado o *s-t corte mínimo* em G'. Ao obter a nova partição $\{S, \overline{S}\}$ dois novos nodos X_s e X_t são criados na árvore T analogamente ao passo anterior.

O algoritmo iterativamente computa $|V| - 1$ cortes para $|V| - 1$ pares de vértices. Estes pares são escolhidos aleatoriamente de um conjunto de nodos na árvore T que contêm vértices ainda não separados pelos cortes até então calculados. Os cortes são computados sobre o grafo contraído que após a primeira iteração é construído a partir da estrutura parcial da árvore e do grafo de entrada.

No Algoritmo 1 é descrito o pseudocódigo para construção da árvores de cortes $T = (V_T, E_T, c_T)$ de um grafo $G = (V_G, E_G, c_G)$.


G'
T
Início
Passo 1
Passo 2
Passo 3
Passo 4
Passo 5
Passo 6

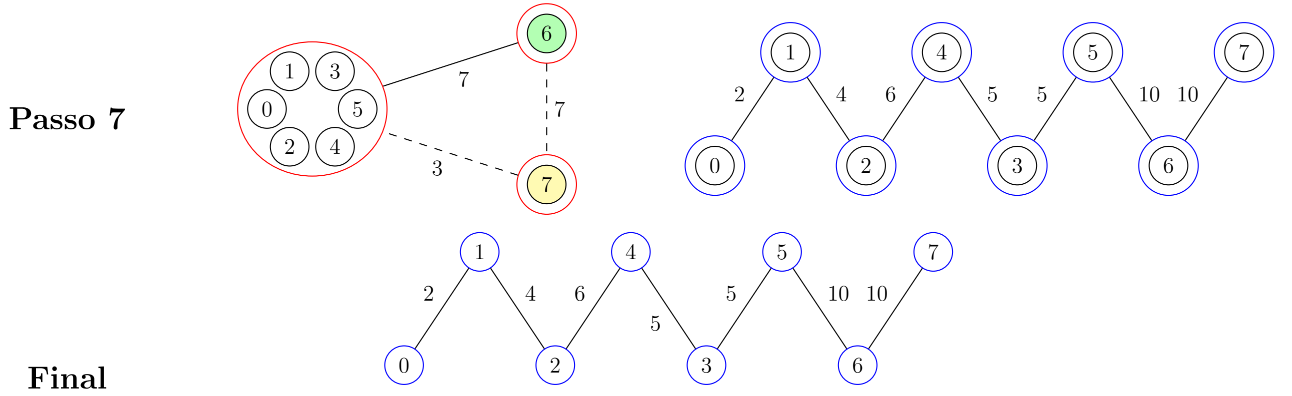


Figura 2.4: Sequência de iterações do algoritmo 1.

A figura 2.4 ilustra um exemplo de execução do Algoritmo 1 para o grafo G da figura 2.3. No passo inicial o Algoritmo 1 constrói a árvore T com apenas um nodo X contendo todos os vértices. Nesse passo, como $T \setminus X = \emptyset$, o grafo contraído G' é uma representação do próprio grafo de entrada. No passo 1, são escolhidos os vértices *1* e *6* pertencentes ao nodo X, calcula-se o *st-corte mínimo* obtendo as partições $S = \{0, 1\}$ e $\overline{S} = \{2, 3, 4, 5, 6, 7\}$ e o valor do corte mínimo $c(S, \overline{S}) = 4$. Em seguida é criado um novo nodo no conjunto de vértices V_T da árvore T tal que $V_T = (V_T \setminus X) \cup \{X \cap S, X \cap \overline{S}\}$. A aresta $e = \{X \cap S, X \cap \overline{S}\}$ é adicionada ao conjunto de arestas $E_T = E_T \cup \{e\}$ com custo $c(e) = 4$.

No passo 2 escolhe-se o nodo $X \in T$ contendo os vértices *0* e *1*. O conjunto com as componentes conexas de $T \setminus X$ é $S_C = \{\{2, 3, 4, 5, 6, 7\}\}$. Em seguida para cada componente em S_C contrai-se G para formar o grafo G' com $V_{G'} = \{\{0\}, \{1\}, \{2, 3, 4, 5, 6, 7\}\}$ e $E_{G'} = \{\{\{0\}, \{1\}\}, \{\{1\}, \{2, 3, 4, 5, 6, 7\}\}\}$. Calcula-se o *st-corte mínimo* em G' entre os vértices *0* e *1* obtendo as partições $S = \{0\}$ e $\overline{S} = \{1, 2, 3, 4, 5, 6, 7\}$ com $c(S, \overline{S}) = 2$. Analogamente ao passo anterior o conjunto de vértices da árvore será $V_T = (V_T \setminus X) \cup \{X \cap S, X \cap \overline{S}\}$ e o conjunto de arestas $E_T = E_T \cup \{e\}$ tal que $e = \{X \cap S, X \cap \overline{S}\}$.

No passo 3, o nodo $X = \{2, 3, 4, 5, 6, 7\}$ é escolhido. O conjunto com as componentes conexas de $T \setminus X$ é $S_C = \{\{0, 1\}\}$. Em seguida, para cada componente em S_C contrai-se G para formar G' conforme ilustra figura 2.4 no passo 3. Os vértices *2* e *7* pertencentes ao nodo X são escolhidos para o cálculo do *st-corte mínimo*, obtendo as partições $S = \{0, 1, 2, 4\}$ e $\overline{S} = \{3, 5, 6, 7\}$ e valor do corte mínimo $c(S, \overline{S}) = 5$. Então o nodo X é dividido separando os vértices *2* e *7* em T.

No passo 4, o nodo $X = \{2, 4\}$ é escolhido. O conjunto com as componentes conexas

de $T \setminus X$ é $S_C = \{\{0,1\},\{3,5,6,7\}\}$. Para cada componente conexa em S_C contrai-se G formando G' conforme ilustra a figura 2.4 no passo 4. Em seguida, calcula-se o *st-corte mínimo* entre os vértices *2* e *4* obtendo as partições $S = \{0,1,2\}$ e $\overline{S} = \{3,4,5,6,7\}$ com valor do corte mínimo $c(S,\overline{S}) = 6$. O nodo X é, então, dividido para separar os vértices *2* e *4* em T.

No passo 5, o nodo $X = \{3,5,6,7\}$ é escolhido. O conjunto com as componentes conexas de $T \setminus X$ é $S_C = \{\{0,1,2,4\}\}$. Para cada componente conexa em S_C contrai-se o grafo G formando G' conforme ilustra figura 2.4 no passo 5. Em seguida, os vértices *3* e *7* são escolhidos. Calcula-se o *st-corte mínimo* entre esses vértices obtendo as partições $S = \{0,1,2,3,4\}$ e $\overline{S} = \{5,6,7\}$ com valor do corte $c(S,\overline{S}) = 5$. O nodo X é então dividido, separando os vértices *3* e *7* em T.

No passo 6, o nodo $X = \{5,6,7\}$ é escolhido. O conjunto com as componentes conexas de $T \setminus X$ é $S_C = \{\{0,1,2,3,4\}\}$. Para cada componente conexa em S_C contrai-se o grafo G formando G' conforme ilustra figura 2.4 no passo 6. Em seguida, os vértices *5* e *6* são escolhidos. Calcula-se o *st-corte mínimo* entre esses vértices obtendo as partições $S = \{0,1,2,3,4,5\}$ e $\overline{S} = \{6,7\}$ com valor do corte $c(S,\overline{S}) = 10$. O nodo X é então dividido, separando os vértices *5* e *6* em T.

No passo 7, o nodo $X = \{6,7\}$ é escolhido. O conjunto com as componentes conexas de $T \setminus X$ é $S_C = \{\{0,1,2,3,4,5\}\}$. Para cada componente conexa em S_C contrai-se o grafo G formando G' conforme ilustra figura 2.4 no passo 7. Em seguida, os vértices *6* e *7* são escolhidos. Calcula-se o *st-corte mínimo* entre esses obtendo as partições $S = \{0,1,2,3,4,5,6\}$ e $\overline{S} = \{7\}$ com valor do corte $c(S,\overline{S}) = 10$. O nodo X é então dividido, separando os vértices *6* e *7* em T. Ao final desse passo a árvore T não possui mais nenhum nodo X tal que $|X| > 1$, logo o algoritmo 1 devolve T conforme ilustra figura 2.4 no passo final.

Quanto ao desempenho, análises experimentais da execução do algoritmo de Gomory-Hu sobre conjuntos de instâncias diversas mostram que os pontos que mais consomem o tempo de execução incluem o cálculo dos *s-t* cortes mínimos e as contrações dos grafos. O que torna o algoritmo eficiente é a descoberta de cortes balanceados, que fazem as operações de contração e cálculo dos *s-t cortes mínimos* serem realizadas sobre grafos de tamanho menor a cada iteração.

2.4 O Algoritmo de Gusfield

O algoritmo de Gusfield, assim como algoritmo de Gomory-Hu, também encontra uma árvore de cortes de um grafo capacitado. A principal diferença é que o algoritmo de Gusfield não realiza contrações e, com isso, calcula os cortes mínimos diretamente sobre o grafo de entrada. O Algoritmo 2 traz o pseudocódigo do algoritmo de Gusfield.

Algoritmo 2: Algoritmo de Gusfield

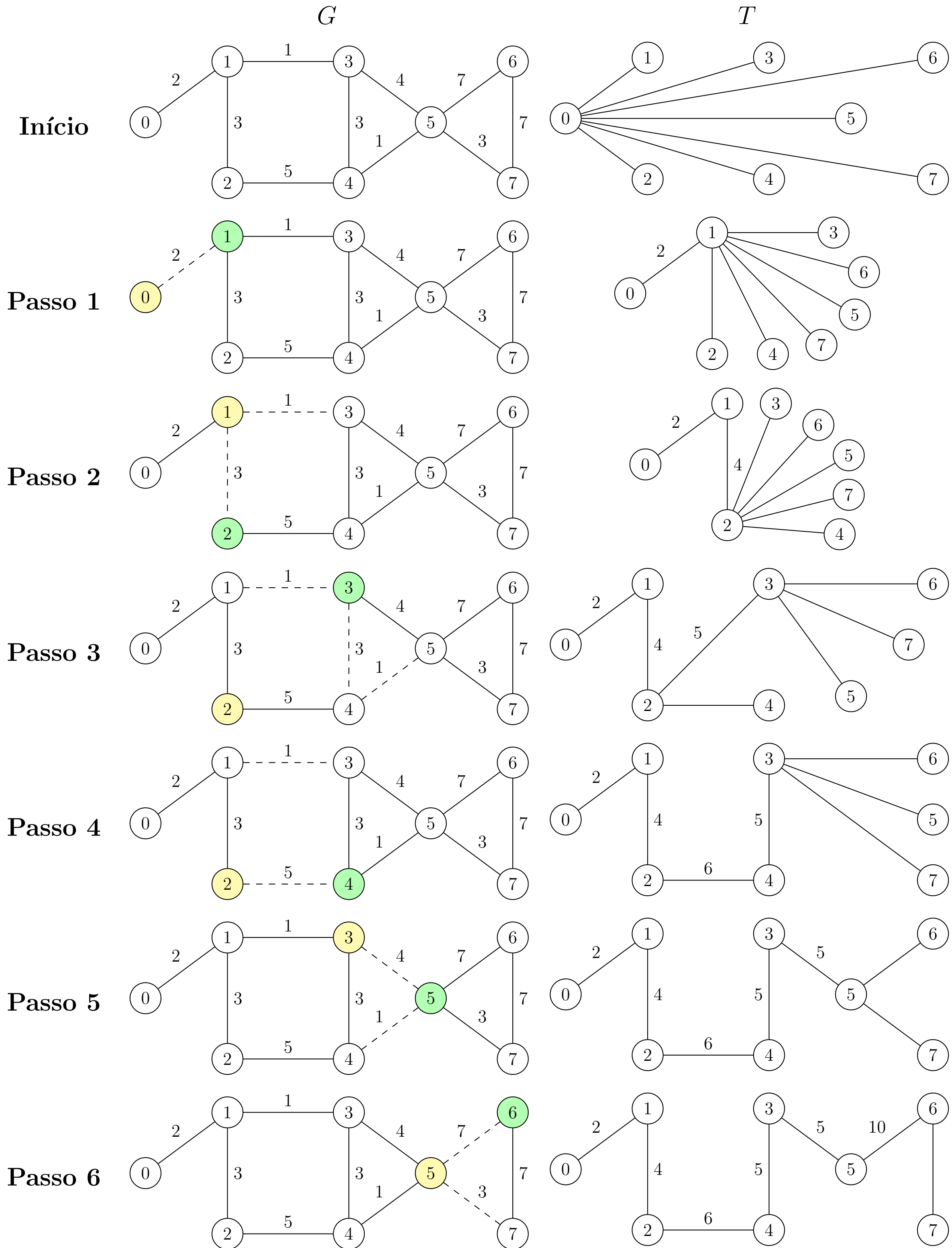
Entrada: $G = (V, E_G, c_G)$ um grafo não dirigido e com peso nas arestas
Saída: $T = (V, E_T, c_T)$ uma árvore de cortes de G

1 Seja *tree* um vetor de tamanho $|V|$ inicializado com 0;
2 Seja λ um vetor de tamanho $|V|$ inicializado com 0;
3 **para** $s = 1$ *até* $|V| - 1$ **faça**
4 $\quad t \leftarrow tree[s]$;
5 $\quad \{X, \overline{X}\} \leftarrow$ s-t-corte mínimo de G;
6 $\quad \lambda[s] \leftarrow c(X, \overline{X})$;
7 $\quad$ **para** $v = 0$ *até* $|V| - 1$ **faça**
8 $\quad\quad$ **se** $v \neq s$ *e* $v \in X$ *e* $tree[v] = t$ **então**
9 $\quad\quad\quad tree[v] \leftarrow s$;
10 $\quad$ **se** $tree[t] \in X$ **então**
11 $\quad\quad tree[s] \leftarrow tree[t]$;
12 $\quad\quad tree[t] \leftarrow s$;
13 $\quad\quad \lambda[s] \leftarrow \lambda[t]$;
14 $\quad\quad \lambda[t] \leftarrow c(X, \overline{X})$;
15 $T \leftarrow (V, E_T = \emptyset, c_T)$;
16 **para** $i = 1$ *até* $|V| - 1$ **faça**
17 $\quad e_T \leftarrow \{i, tree[i]\}$;
18 $\quad c(e_T) \leftarrow \lambda[i]$;
19 $\quad E_T = E_T \cup e_T$;
20 **retorna** T

O algoritmo pode ser implementado de maneira simples e sem estruturas de dados complexas. O vetor *tree* define um vizinho para cada vértice na árvore. No vetor λ os valores $\lambda[i]$ correspondem ao custo dos s-t cortes mínimos para as arestas $\{i, tree[i]\}$ para todos os vértices $i \in V$.

O algoritmo inicia a árvore com uma topologia estrela em que o nodo 0 é o centro (linha 1). A cada iteração (linhas 3-3), o algoritmo escolhe um vértice s, $s \geq 1$, diferente como *origem*. Essa escolha determina o vértice de *destino* t como o vizinho atual de s na árvore, pela atribuição $t \leftarrow tree[s]$ na linha 4. Na linha 5 o algoritmo calcula um s-t corte mínimo e atualiza a árvore da seguinte forma: para cada vértice v vizinho de t que esteja do lado de s do s-t corte mínimo, conjunto X, é desconectado de t e reconectado

a s (linhas 7-7). O algoritmo finaliza quando todos os vértices de 1 até $|V|$ tiverem sido origem de s-t corte mínimo de alguma iteração.



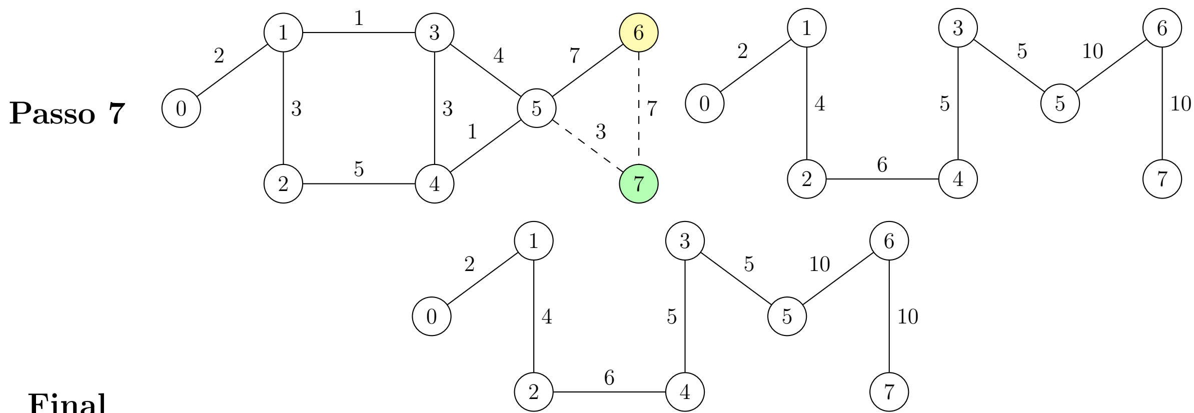


Figura 2.5: Sequência de iterações do Algoritmo 2.

A Figura 2.4 apresenta um exemplo de execução do algoritmo de Gusfield. Na coluna da esquerda é representado o grafo de entrada com o vértice de origem s destacado em verde e o vértice de destino t destacado em amarelo. As arestas tracejadas representam o s-t corte mínimo. Na coluna da direita são representadas as atualizações na árvore. No passo final é representada a árvore que será retornada pelo algoritmo.

No passo 1 são escolhidos os vértices origem $s = 1$ e destino $t = 0$ e o algoritmo encontra um s-t corte mínimo de custo 2 que os separa. Como os vértices $2, 3, 4, 5, 6, 7$ estão do lado do vértice s no s-t corte mínimo então a árvore é atualizada de forma a conectar todos eles ao vértice 1. A aresta que liga os vértices s, t na árvore recebe a capacidade 2, que foi encontrada no cálculo do s-t corte mínimo. No passo 2 o algoritmo busca um s-t corte mínimo que separa os vértices $s = 2$ e $t = 1$. O custo do s-t corte mínimo encontrado é 4 e a árvore é atualizada ligando os vértices $3, 4, 5, 6, 7$ ao vértice 2. O algoritmo continua de maneira análoga até que a árvore esteja completa.

2.5 Conclusão

Uma árvore de cortes representa de forma sucinta a aresta conectividade entre todos os pares de vértices de um grafo. Neste capítulo foram apresentados os algoritmos clássicos para a construção de árvores de cortes: o algoritmo de Gomory-Hu e o algoritmo de Gusfield. Ambos os algoritmos são similares já que fazem chamadas a um procedimento que calcula subproblemas de cortes mínimos entre pares de vértices $n - 1$ vezes para um grafo com n vértices. A principal diferença entre os dois é o fato de que o algoritmo de

Gusfield calcula os $n-1$ cortes mínimos sobre o grafo de entrada e o algoritmo de Gomory-Hu realiza contrações e calcula os cortes mínimos sobre instâncias de grafos contraídos. O algoritmo de Gomory-Hu também exige a manipulação de estruturas de dados não triviais para controlar a construção da árvore de cortes e dos grafos contraídos.

No próximo capítulo será apresentada uma versão paralela do algoritmo de Gomory-Hu. No capítulo 4 é apresentada a principal contribuição do trabalho: uma estratégia para otimização da contração de grafos na construção paralela de árvores de cortes.

CAPÍTULO 3

O ALGORITMO PARALELO DE GOMORY-HU

Na computação paralela, são utilizados diversos processos em conjunto para resolver um problema, dividindo o esforço de uma determinada tarefa. A chave da computação paralela é a existência de concorrência explorável [24], de forma que porções do problema possam executar consistentemente ao mesmo tempo. Um programa paralelo quando executado em um ambiente com múltiplas unidades de processamento tem, potencialmente, seu tempo de resposta reduzido. Se dispusermos de vários processadores podemos particionar as tarefas de forma a serem executadas simultaneamente em diferentes unidades. Os resultados parciais podem ser combinados chegando ao resultado final. Porém, há casos em que algumas tarefas utilizam dados produzidos por outras tarefas e com isso necessitam aguardar até o término dessas [23].

Os ambientes para computação paralela disponibilizam ferramentas e recursos básicos necessários para construir programas paralelos [24]. Estes recursos incluem uma interface de programação (API - *Application Programming Interface*). Dentre as principais API's para desenvolvimento de programas de computação paralela destacam-se a MPI[1] e a OpenMP[2], descritas a seguir.

A MPI [20] é uma especificação de API que permite criar programas paralelos no paradigma de troca de mensagens. A MPI tornou-se um padrão acadêmico e industrial para o desenvolvimento de aplicações paralelas e distribuídas.

A OpenMP [7] é uma especificação de API que permite criar programas paralelos no paradigma de memória compartilhada. Através de diretivas que estendem várias linguagens de programação como Fortran, C e C++, pode-se definir como o processamento será compartilhado pelas *threads* (linhas de execução) quanto ao acesso à memória e execução.

Este capítulo apresenta, na seção 3.1, a versão paralela do algoritmo de Gomory-Hu. Na seção 3.2 são mostrados detalhes da implementação utilizando a biblioteca Boost e na seção 3.3 resultados experimentais sobre um conjunto de instâncias diversas.

[1]The Message Passing Interface (MPI) standard, `http://www.mcs.anl.gov/research/projects/mpi`
[2]OpenMP® API specification for parallel programming, `http://openmp.org`

3.1 Uma Versão Paralela do Algoritmo de Gomory-Hu Utilizando MPI

Em [10] é apresentada uma versão paralela do algoritmo de Gomory e Hu. É descrita uma implementação MPI seguindo a arquitetura mestre-escravo – modelo de comunicação em que um dispositivo ou processo, chamado de mestre, tem controle sobre os demais processos, chamados de escravos.

O processo mestre é responsável por realizar as atualizações na árvore em construção e gerar instâncias do problema do *s-t-corte mínimo* para os processos escravos. Estes constroem instâncias de grafo contraído e calculam o *s-t-corte mínimo* retornando ao mestre o resultado. A estratégia, segundo os autores, é a do melhor esforço, ou seja, os processos escravos calculam os *s-t-cortes mínimos* independentemente.

O Algoritmo 3 mostra a versão paralela do algoritmo de Gomory-Hu, em que as operações *envie* e *receba* são chamadas para funções de troca de mensagens da biblioteca MPI.

Seja p o número total de processos e $proc_0$, $proc_1$, ... , $proc_{p-1}$ os processos que executam o algoritmo paralelo de Gomory-Hu. Cada processo, ao iniciar, mantém uma cópia do grafo de entrada $G = (V_G, E_G, c_G)$. O processo mestre, identificado como $proc_0$, inicializa a árvore $T = (V_T, E_T, c_T)$, escolhe um nodo pivô $X \in V_T$ através do procedimento *escolhe_pivô(T)*, tal que $|X| \geq 2$. Em seguida é criada uma *tarefa*, composta por um par de vértices $s, t \in X$ e uma *partição* dos vértices de G que os associa aos nodos do grafo contraído que será construído pelo processo escravo. Os vértices s, t são escolhidos através de um procedimento denominado *escolhe_par_st(X)* e a *partição* é criada através do procedimento *constrói_partição(X, T)*. Só então o processo mestre envia para cada processo escravo $proc_s$, $s > 0$, uma *tarefa* e fica aguardando a resposta.

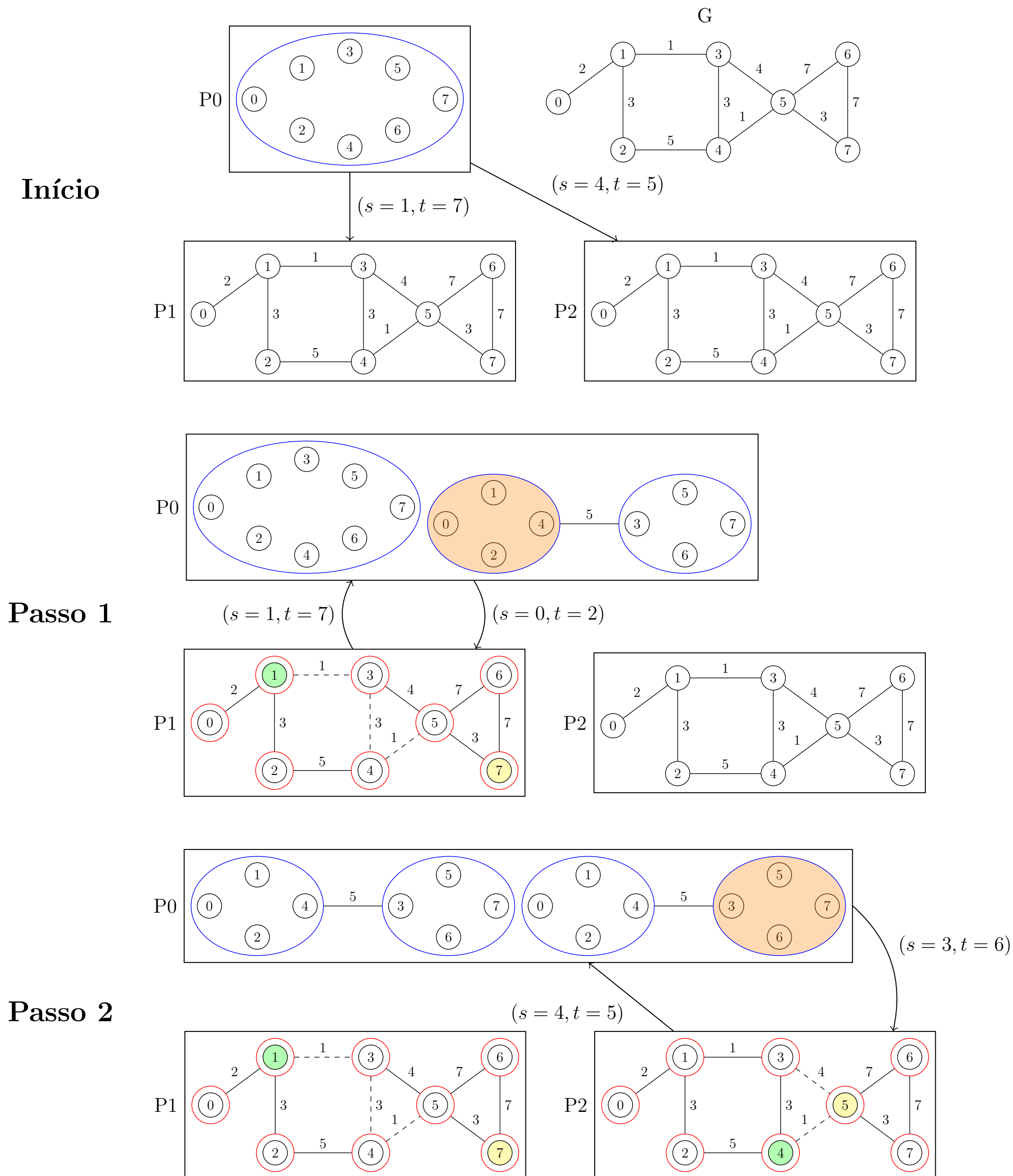
Um processo escravo $proc_s$, por sua vez, recebe a tarefa do processo mestre e constrói um grafo contraído G'. Em seguida, com o par de vértices s e t, o processo $proc_s$ resolve um subproblema de *s-t corte mínimo* e retorna para o processo mestre uma *resposta* composta pelos vértices s e t recebidos na tarefa e por uma partição de vértices S resultantes do *s-t corte mínimo*. O processo mestre recebe de $proc_s$ a *resposta* e verifica se o par de vértices s e t está contido no mesmo nodo da árvore. Caso esteja, a árvore T será atualizada, caso contrário nada é feito e neste caso o trabalho realizado pelo escravo é perdido. No

Algoritmo 3: GomoryHu-MPI

```
    Entrada: G = (V, E_G, c_G) , proc_j, 0 ≤ j < p processos
    Saída: T = (V, E_T, c_T) uma árvore de cortes de G
 1  se proc_j = 0 // processo mestre
 2  então
 3  |  para s ← 1 até p − 1 faça
 4  |  |  X ← escolhe_pivô(T);
 5  |  |  partição ← constroi_partição(pivô, T) ;
 6  |  |  (s, t) ← escolhe_par_st(pivô);
 7  |  |_ envie tarefa (s,t,partição) para processo proc_s;
 8  |  enquanto |V_T| < |V_G| faça
 9  |  |  receba de proc_j resposta (s,t,S), onde {S, S̄} é um s-t-corte mínimo de G ;
10  |  |  se s e t pertencem ao mesmo nodo pivô de V_T então
11  |  |  |  X_s ← X ∩ S;
12  |  |  |  X_t ← X ∩ S̄;
13  |  |  |  V_T ← V_T ∪ {novo} ;
14  |  |  |  se |X_s| = 1 então
15  |  |  |  |  retire X_s da lista de candidatos a pivô;
16  |  |  |  senão
17  |  |  |  |_ insira X_s na lista de candidatos a pivô ;
18  |  |  |  se |X_t| = 1 então
19  |  |  |  |  retire X_t da lista de candidatos a pivô;
20  |  |  |  senão
21  |  |  |  |_ insira X_t na lista de candidatos a pivô ;
22  |  |  |  e ← {X_s, X_t};
23  |  |  |  c(e) ← c(S, S̄);
24  |  |  |  para cada aresta e' = {X, Y} ∈ E_T incidente em X na árvore T faça
25  |  |  |  |  se Y ⊆ S então
26  |  |  |  |  |  E_T ← E_T ∪ {{X_s, Y}} \ {{X, Y}};
27  |  |  |  |  senão
28  |  |  |  |_ |_ E_T ← E_T ∪ {{X_t, Y}} \ {{X, Y}};
29  |  |  |  V_T ← (V_T \ {X}) ∪ {X_s, X_t};
30  |  |  |  E_T ← E_T ∪ {e};
31  |  |  se |V_T| = |V_G| então
32  |  |  |  envie mensagem de finalização ao processo proc_j
33  |  |  senão
34  |  |  |  X ← escolhe_pivô(T);
35  |  |  |  partição ← constroi_partição(pivô, T) ;
36  |  |  |  (s, t) ← escolhe_par_st(pivô);
37  |  |_ |_ envie tarefa (s,t,partição) para processo proc_s;
38  |  retorna T ;
39  senão
    |  // processo escravo
40  |  enquanto receber tarefas faça
41  |  |  receba tarefa (s,t,partição);
42  |  |  se tarefa = fim então
43  |  |  |  termine
44  |  |  G' ← constrói_grafo_contraído(G, partição) ;
45  |  |  S ← corte_mínimo(G', s, t) ;
46  |_ |_ envie reposta (s,t,S) para proc_0
```

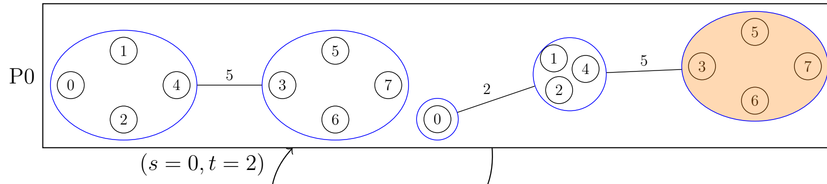
passo seguinte, o processo mestre escolhe um novo nodo *pivô* através do procedimento *escolhe_pivô(T)* e envia uma nova *tarefa* para $proc_s$ e processa outras *respostas* ou fica aguardando outro processo escravo se comunicar.

A atualização da árvore deve ser feita de forma serial pois não são permitidas atualizações concorrentes. Esta característica faz com que o mestre processe uma resposta de cada vez. Algumas respostas poderão ser descartadas, já que os vértices s e t podem já ter sido separados por outra atualização de T resultante de outro *s-t corte mínimo* encontrado por outro processo.



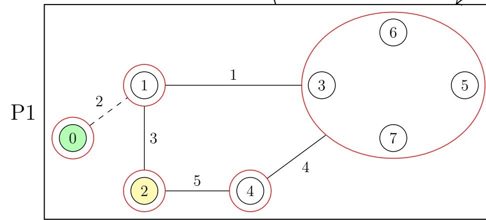
Passo 3

P0

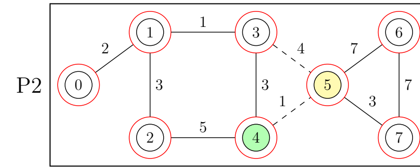


$(s = 0, t = 2)$ $(s = 6, t = 7)$

P1

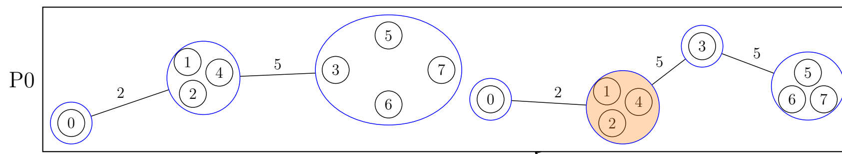


P2



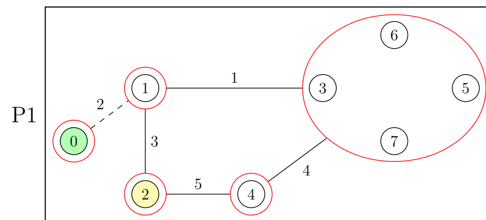
Passo 4

P0

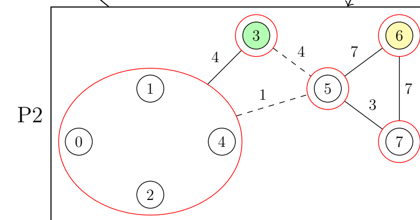


$(s = 4, t = 2)$ $(s = 3, t = 6)$

P1

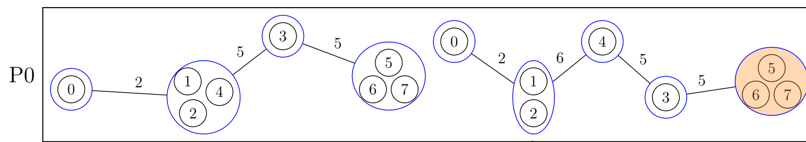


P2



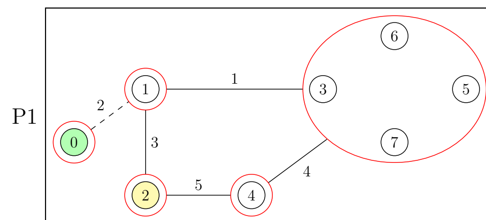
Passo 5

P0

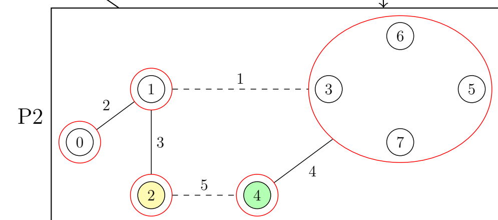


$(s = 5, t = 6)$ $(s = 4, t = 2)$

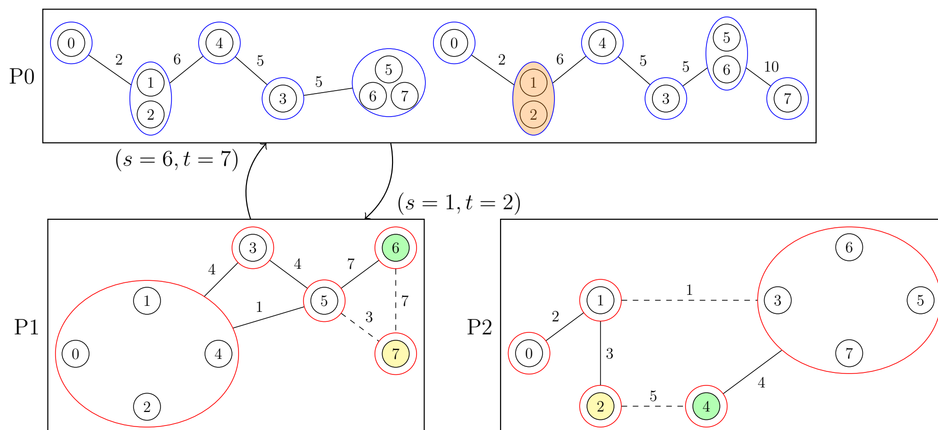
P1



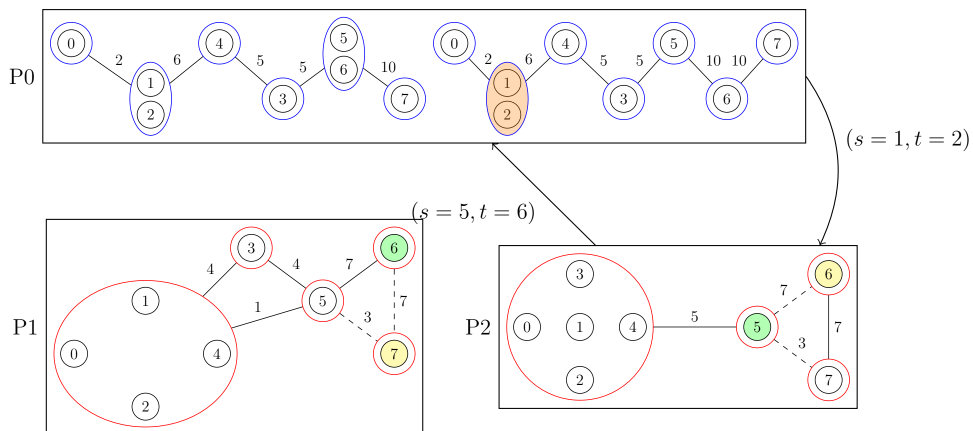
P2



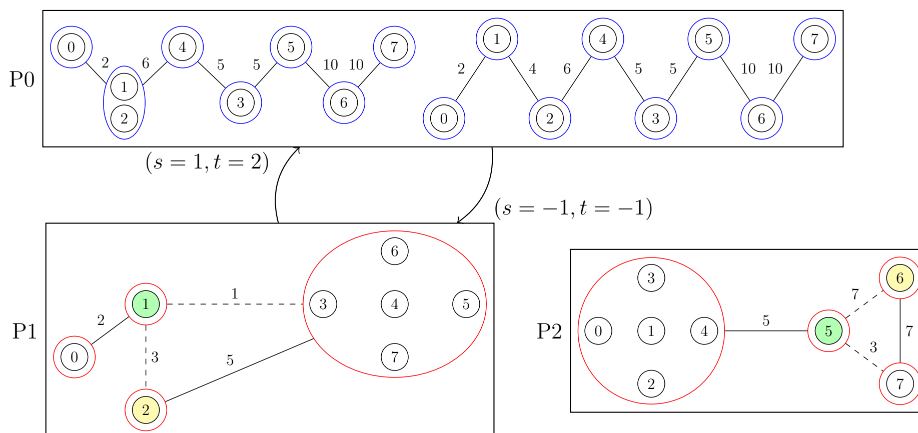
Passo 6



Passo 7



Passo 8



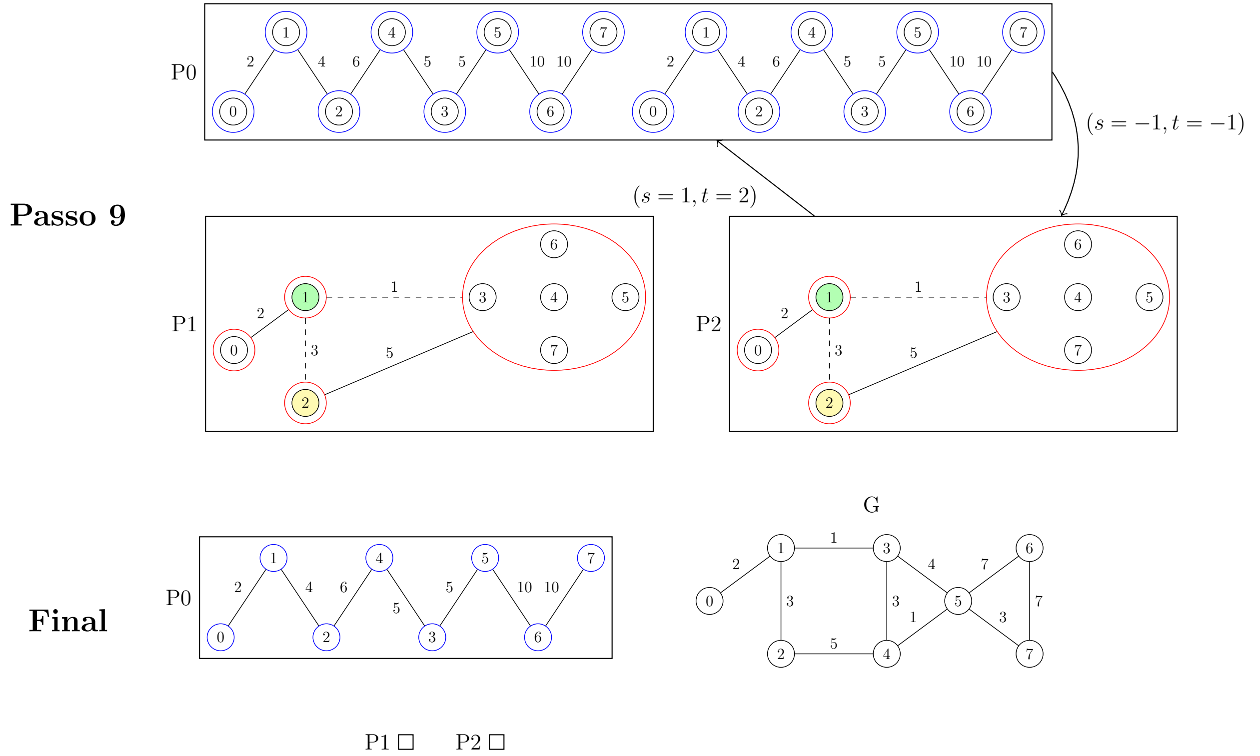


Figura 3.1: Exemplo de execução do algoritmo paralelo de Gomory-Hu.

A Figura 3.1 mostra um exemplo de execução do algoritmo paralelo de Gomory-Hu. Neste exemplo o processo $P0$ inicia a árvore T com todos os vértices do grafo G representados por um único nodo. Em seguida, o processo $P0$ escolhe pares de vértices s e t e envia para os processos escravos $P1$ e $P2$. Os processos escravos por sua vez, contraem o grafo de entrada G, calculam o corte mínimo entre os vértices s e t e retornam ao processo $P0$ a resposta.

O passo 1 mostra a comunicação do processo $P1$ retornando a resposta do corte mínimo entre $s = 1$ e $t = 7$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 0$ e $t = 2$ para o cálculo do corte mínimo e é enviado ao processo $P1$.

O passo 2 mostra a comunicação do processo $P2$ retornando a resposta do corte mínimo entre $s = 1$ e $t = 5$ ao mestre. O processo mestre identifica que os vértices já estão separados na árvore e não faz nenhuma atualização. Em seguida, escolhe um outro nodo da árvore e o par de vértices $s = 3$ e $t = 6$ para o cálculo do corte mínimo enviando ao processo $P1$ essa tarefa.

O passo 3 mostra a comunicação do processo $P1$ retornando a resposta do corte mínimo

entre $s = 0$ e $t = 2$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 6$ e $t = 7$ para o cálculo do corte mínimo e é enviado ao processo $P1$.

O passo 4 mostra a comunicação do processo $P2$ retornando a resposta do corte mínimo entre $s = 3$ e $t = 6$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 4$ e $t = 2$ para o cálculo do corte mínimo e é enviado ao processo $P2$.

O passo 5 mostra a comunicação do processo $P2$ retornando a resposta do corte mínimo entre $s = 4$ e $t = 2$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 5$ e $t = 6$ para o cálculo do corte mínimo e é enviado ao processo $P2$.

O passo 6 mostra a comunicação do processo $P1$ retornando a resposta do corte mínimo entre $s = 6$ e $t = 7$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 1$ e $t = 2$ para o cálculo do corte mínimo e é enviado ao processo $P1$.

O passo 7 mostra a comunicação do processo $P2$ retornando a resposta do corte mínimo entre $s = 5$ e $t = 7$ ao mestre. O processo mestre atualiza a árvore e escolhe um outro nodo da árvore. O nodo destacado é escolhido e, após, é escolhido o par de vértices $s = 1$ e $t = 2$ para o cálculo do corte mínimo e é enviado ao processo $P2$.

O passo 8 mostra a comunicação do processo $P1$ retornando a resposta do corte mínimo entre $s = 1$ e $t = 2$ ao mestre. O processo mestre atualiza a árvore e desta vez não há nenhum nodo que contenha mais de dois vértices na árvore, portanto o processo $P0$ envia a tarefa $s = -1$ e $t = -1$ indicando fim de processamento. O processo $P1$ ao receber esta mensagem irá finalizar.

O passo 9 mostra a comunicação do processo $P2$ retornando a resposta do corte mínimo entre $s = 1$ e $t = 2$ ao mestre. O processo mestre não atualiza a árvore pois os vértices 1 e 2 já estão separados, e como não há nenhum nodo que contenha mais de dois vértices na árvore o processo $P0$ envia a tarefa $s = -1$ e $t = -1$ indicando fim de processamento. O processo $P1$ ao receber esta mensagem irá finalizar.

No passo final o algoritmo retorna a árvore construída e finaliza.

O processo $P0$ aproveita os cortes mínimos que possibilitam a atualização da árvore, separando os vértices s e t, e descarta aqueles que não possibilitam a atualização da

árvore, toda vez que os vértices s e t já tenham sido separados por outro corte mínimo encontrado por outro processo. Quando não existirem nodos na árvore que contenham mais de 2 vértices, o processo $P0$ envia uma tarefa com valores $s = -1$ e $t = -1$ sinalizando fim do algoritmo. A árvore construída é então retornada e o algoritmo termina.

3.2 Uma Implementação do Algoritmo de Gomory-Hu Utilizando a Biblioteca Boost

Para uma das implementações práticas dos algoritmos descritos neste trabalho foi utilizada a linguagem de programação C++ juntamente com o conjunto de bibliotecas Boost[3]. A Boost estende funcionalidades da linguagem padrão C++, com um grande número de bibliotecas que implementam estruturas de dados, *containers*, iteradores e algoritmos. Para garantir eficiência e flexibilidade, a Boost faz o uso intensivo de *templates* fazendo com que seja uma forte referência para os conceitos de programação genérica e metaprogramação em C++ [1].

Como existem várias implementações bem testadas, estáveis e eficientes de MPI, e o objetivo deste trabalho não é comparar estas implementações, foi escolhida a biblioteca OpenMPI[4] com a utilização da *API Boost.MPI* para implementação do algoritmo paralelo de Gomory-Hu.

Foram utilizadas as estruturas de dados e algoritmos da *BGL* (*Boost Graph Library*) a qual possui diversos recursos e algoritmos para tratar grafos. Na implementação realizada, foram avaliados diversos algoritmos de *fluxo máximo* fornecidos pela biblioteca *BGL* na construção das árvores de cortes. Foram utilizados os algoritmos de Edmonds e Karp [14], Goldberg e Tarjan [17] e o algoritmo de Boykov e Kolmogorov [6]. Para auxiliar na descrição dos experimentos na seção seguinte, esses algoritmos são referenciados como *EK*, *PR* e *BK* respectivamente.

O algoritmo *EK* é uma implementação do algoritmo de Ford e Fulkerson [16] e possui complexidade $O(VE^2)$. O algoritmo *BK* é também baseado em caminhos aumentantes e foi elaborado para ser eficiente em problemas de visão computacional. Seja $|C|$ a capacidade de um corte mínimo, então a complexidade do algoritmo de fluxo *BK* é, teoricamente, $O(VE^2|C|)$, porém na prática seu desempenho é melhor do que seus antecessores base-

[3]Boost C++ Libraries, `www.boost.org`
[4]Open Source High Performance Computing, `www.open-mpi.org`

ados em caminhos aumentantes. O algoritmo *PR* possui complexidade $O(V^2E)$ em sua versão genérica. Este algoritmo difere dos outros dois uma vez que não utiliza caminhos aumentantes.

3.3 Resultados Experimentais

Para a execução dos experimentos foram utilizadas as instâncias apresentadas na Tabela 3.2. As características de cada instância também são descritas nessa tabela. As instâncias BLOG, PGRI, ROME e GEO são instâncias de grafos com dados reais: uma rede de blogs [2], uma rede de distribuição de energia [29], uma rede representando as ruas de Roma [27] e uma rede colaborativa científica [4]. Duas instâncias foram geradas utilizando os modelos Erdős–Rényi [5] e Barabási–Albert [3]. As demais instâncias são grafos sintéticos que foram utilizadas em *benchmarks* para algoritmos de corte mínimo e árvores de cortes [26, 8, 18]. As instâncias do tipo *NOI* são grafos aleatórios com conjuntos de vértices divididos em grupos de forma que as arestas internas ao grupo tendem a ter capacidades maiores do que as arestas entre grupos. As instâncias do tipo *PATH* são grafos aleatórios que possuem um caminho de tamanho k com arestas de capacidades altas. Cada vértice fora desse caminho é conectado aleatoriamente ao caminho por uma aresta também com capacidade alta e as demais arestas são aleatórias e têm capacidades baixas. As instâncias do tipo *TREE* são semelhantes às instâncias do tipo *PATH*, porém são grafos aleatórios baseados em uma árvore aleatória com arestas de capacidades altas conectando os primeiros k vértices.

O ambiente de execução foi um *cluster* homogêneo com 18 servidores biprocessados interligados por uma rede Ethernet de 1Gbps[5]. Os processadores são do tipo Intel® Xeon® Processor E5-2670[6] com *clock* de 2.60GHz com 8 núcleos e 16 *threads* cada. Cada servidor possui 128GB de memória RAM e 20MB de memória *cache*. Em todos os experimentos foram utilizados um processo por servidor.

A implementação foi compilada com *g++* utilizando o nível de otimização O3.

Os seguintes experimentos foram realizados:

- Testes comparativos dos algoritmos de fluxo sobre todo o conjunto de instâncias

[5] *Cluster* do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é financiado pela FINEP através dos projetos CT-INFRA/UFPR

[6] Intel® Xeon® Processor E5-2670 `http://ark.intel.com/pt-br/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI`

Tabela 3.2: Características das instâncias utilizadas nos experimentos.

Instância	$\|V\|$	$\|E\|$	Descrição
BA	2000	9995	Grafos do modelo Barabási-Albert [3].
DBCYC	1024	2048	É um grafo formado por dois ciclos intercalados com arestas cujas capacidades produzem um corte mínimo perfeitamente balanceado e muitos cortes com capacidades próximas da capacidade do corte mínimo [8].
ER	2000	10079	Grafos do modelo Erdös-Rényi ou $G_{n,p}$ [5].
GEO	3621	9461	Uma rede de colaboração científica [4].
NOI	1000	99900	Grafos aleatórios com conjunto de vértices divididos em grupos de forma que as arestas internas aos grupos tendem a ter capacidades maiores do que as arestas entre grupos [26].
PATH	2000	21990	Grafos aleatórios que possuem um caminho de tamanho k com capacidades altas [8, 18].
BLOG	1222	16714	Uma rede de **blogs** [2].
PGRI	4941	6594	Uma de rede de transmissão de eletricidade [29].
ROME	3353	8870	Uma rede de ruas da cidade de Roma [27].
TREE	2000	21990	Classe de grafos semelhante à classe **PATH**, porém constrói uma árvore aleatória com capacidades altas conectando os primeiros k vértices ao invés de construir um caminho [8, 18].

utilizando a capacidade máxima do cluster;

- Testes comparativos dos algoritmos de fluxo sobre as instâncias *ER* e *BA* variando o grau médio dos grafos gerados aleatoriamente.
- Testes de escalabilidade da implementação sobre todo o conjunto de instâncias da Tabela 1 iniciando com dois processos e incrementando o número de processos até o máximo do cluster, que são 18 servidores;

O gráfico ilustrado na Figura 3.2 representa os comparativos dos algoritmos de fluxo máximo sobre todo o conjunto de instâncias da tabela 3.2. Como o algoritmo *EK* [14] gastou um tempo muito maior do que os outros em algoritmos nas instâncias *NOI*, *TREE* e *PATH*, foi necessário utilizar escala logarítmica para o tempo. Nesta figura é possível verificar que a implementação do algoritmo de fluxo máximo *BK*, da biblioteca Boost, faz com que a implementação do algoritmo de Gomory-Hu, também utilizando a biblioteca Boost, tenha melhor desempenho em todas as instâncias, sendo que nas instâncias *ROME*, *BLOG*, *GEO*, *ER* e *BA* a vantagem no tempo total de execução foi expressiva. O algoritmo de Gomory-Hu obteve melhores tempos de execução nas instâncias *ROME*, *PGRI*, *GEO* e *DBCYC* utilizando a implementação do algoritmo de fluxo máximo *EK* comparado ao algoritmo *PR*. De forma geral, para instâncias de grafos esparsos, com poucas arestas, o algoritmo de fluxo máximo *EK* mostra-se ser eficiente para resolução do problema,

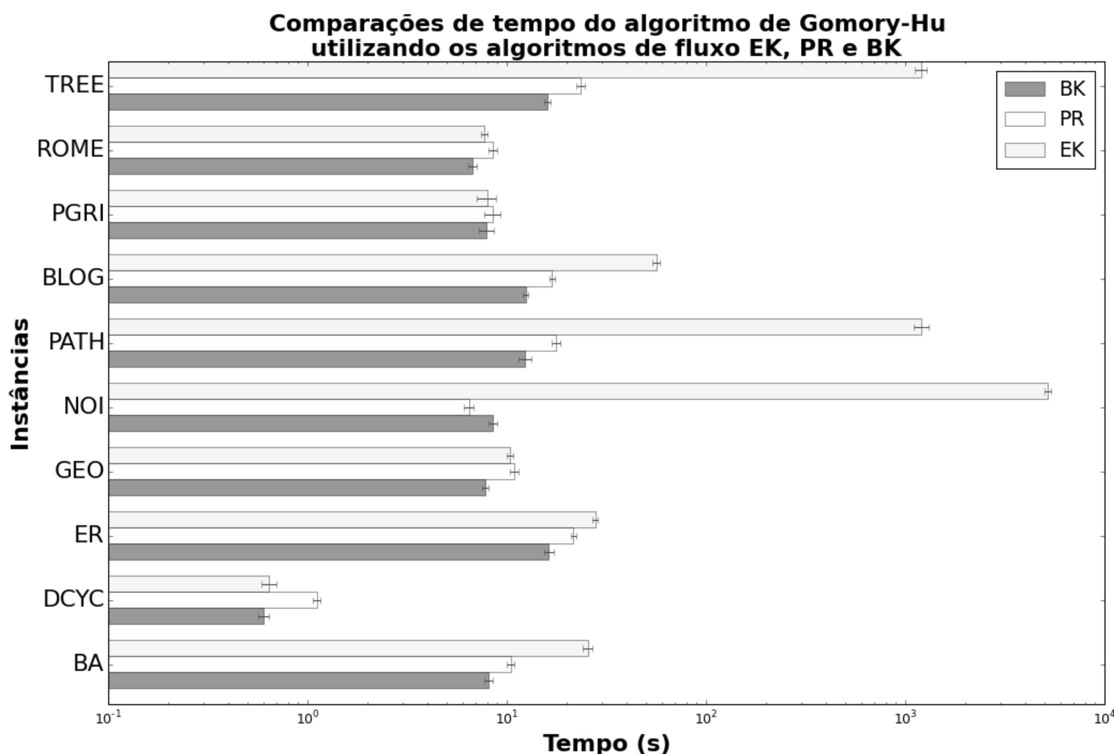


Figura 3.2: Comparativos dos algoritmos de fluxo máximo.

porém, na medida em o que o número de arestas no grafo aumenta seu desempenho cai drasticamente como pode ser visto nos gráficos descritos a seguir.

Os gráficos ilustrados na figura 3.3 e 3.4 representam os experimentos comparando os algoritmos de fluxo máximo sobre as instâncias *ER* (*Erdös-Rényi*). Nesses experimentos foram variados os graus médios de cada instância *ER* entre 5 e 205 com 1000 vértices. O gráfico da figura 3.3 representa os experimentos realizados em instâncias de *ER* com pesos unitários nas arestas, ao passo que o gráfico da figura 3.4 foram realizados experimentos com variação dos pesos nas arestas. Em ambas as situações o algoritmo de fluxo máximo *EK* apresentou desempenho significativamente pior do que os outros dois algoritmos, na medida em que a densidade das instâncias aumenta. Os algoritmos *BK* e *PR* apresentaram desempenhos próximos de lineares com uma leve vantagem para o algoritmo *BK*. Utilizar pesos aleatórios, com distribuição uniforme, nas arestas provocou um impacto negativo apenas no algoritmo *EK*, não apresentando alterações significativas de desempenho para os os outros dois algoritmos de fluxo.

Analogamente aos experimentos com as instâncias *ER*, foram realizados experimentos com as instâncias *BA* (Barabási-Albert) ilustrados nos gráficos das figuras 3.5 e 3.6.

Nesses experimentos foram utilizadas as mesmas parametrizações das instâncias *ER*, as quais definem os graus médios de cada instância *BA* entre 5 e 205 com 1000 vértices. O gráfico da figura 3.5 representa os experimentos realizados em instâncias de *BA* com pesos unitários nas arestas, ao passo que o gráfico da figura 3.6 foram realizados experimentos com variação dos pesos nas arestas. Analogamente aos experimentos com as instâncias do tipo *ER*, em ambas as situações o algoritmo de fluxo máximo *EK* apresentou desempenho significativamente pior do que os outros dois algoritmos, à medida que o número de arestas das instâncias aumentou. Os algoritmos *BK* e *PR* apresentaram desempenhos próximos com uma leve vantagem para o algoritmo *BK*. Utilizar pesos aleatórios nas arestas provocou um impacto negativo apenas no algoritmo *EK*, não apresentando alterações de desempenho para os os outros dois algoritmos de fluxo.

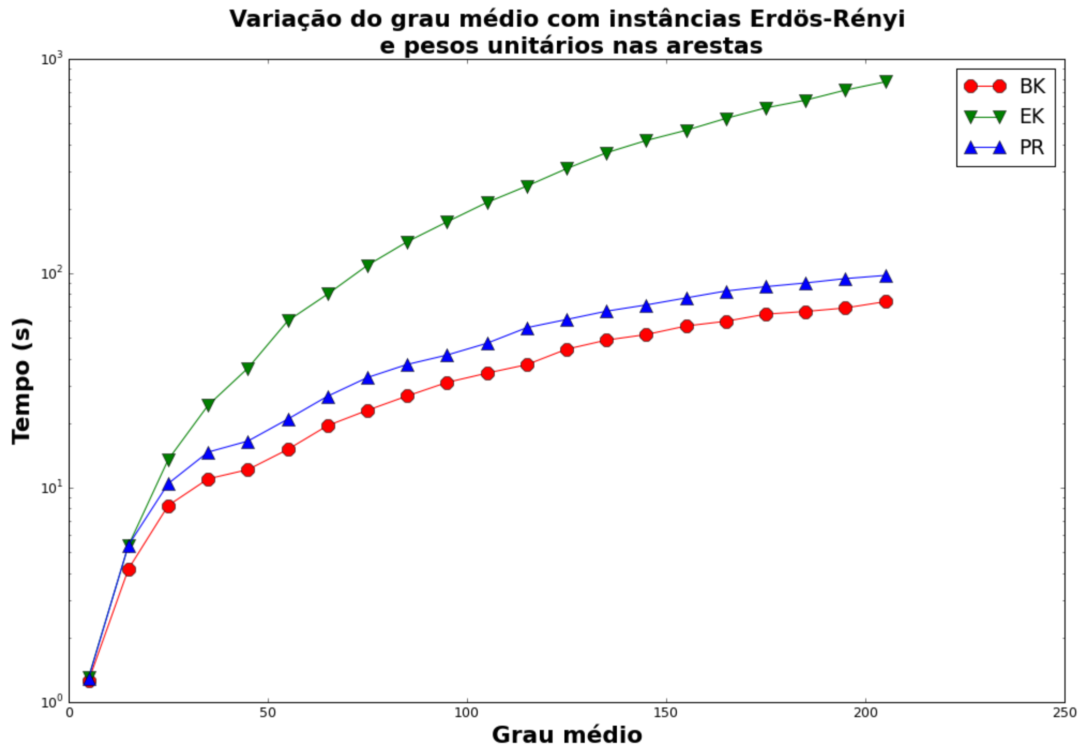


Figura 3.3: Comparativos dos algoritmos de fluxo máximo utilizando as instâncias Erdös-Rényi com variação do grau médio e arestas com pesos unitários.

Os dados coletados mostraram que os algoritmos de Boykov e Kolmogorov [6] e Goldberg e Tarjan [17] apresentaram resultados superiores ao algoritmo de Edmonds e Karp [14]. O algoritmo de Boykov e Kolmogorov teve uma pequena vantagem no tempo de execução sobre a implementação do algoritmo de Goldberg e Tarjan. Os desvios nas cur-

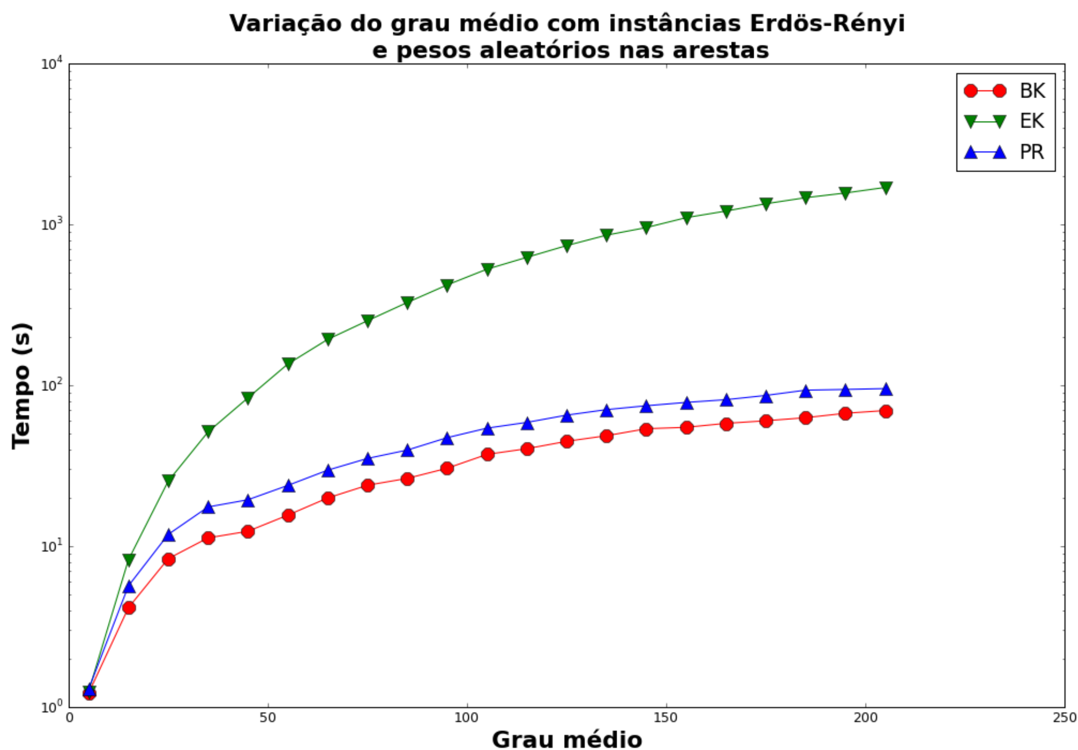


Figura 3.4: Comparativos dos algoritmos de fluxo máximo utilizando as instâncias Erdös-Rényi com variação do grau médio e arestas com pesos aleatórios.

vas demonstradas nas figuras 3.5 e 3.6 representam oscilações na carga de processamento dos servidores do cluster utilizado, uma vez que os experimentos foram realizados com prioridade baixa.

O gráfico da Figura 3.7 representa os resultados dos testes de escalabilidade. Os *speedups* foram calculados como $S = T_S/T_P$, tal que T_S corresponde ao tempo da implementação do algoritmo sequencial e T_P corresponde ao tempo de execução da implementação do algoritmo paralelo com P processos. Todos os experimentos foram executados 30 vezes em cada uma das instâncias da Tabela 3.2, variando entre 2 e 18 o número de processos. O algoritmo de fluxo máximo utilizado foi o *BK*, já que esse apresentou melhor tempo de execução nos testes anteriores. É possível identificar que as instâncias *PGRI*, *ROME*, *GEO*, *NOI* e *DCYC* ou pararam ou obtiveram poucos *speedups* positivos a partir de aproximadamente 8 processos. As demais instâncias obtiveram *speedups* próximos de lineares.

Os experimentos demonstram que o desempenho do algoritmo de Gomory-Hu é dependente tanto da instância na qual é aplicado, bem como do algoritmo de fluxo máximo

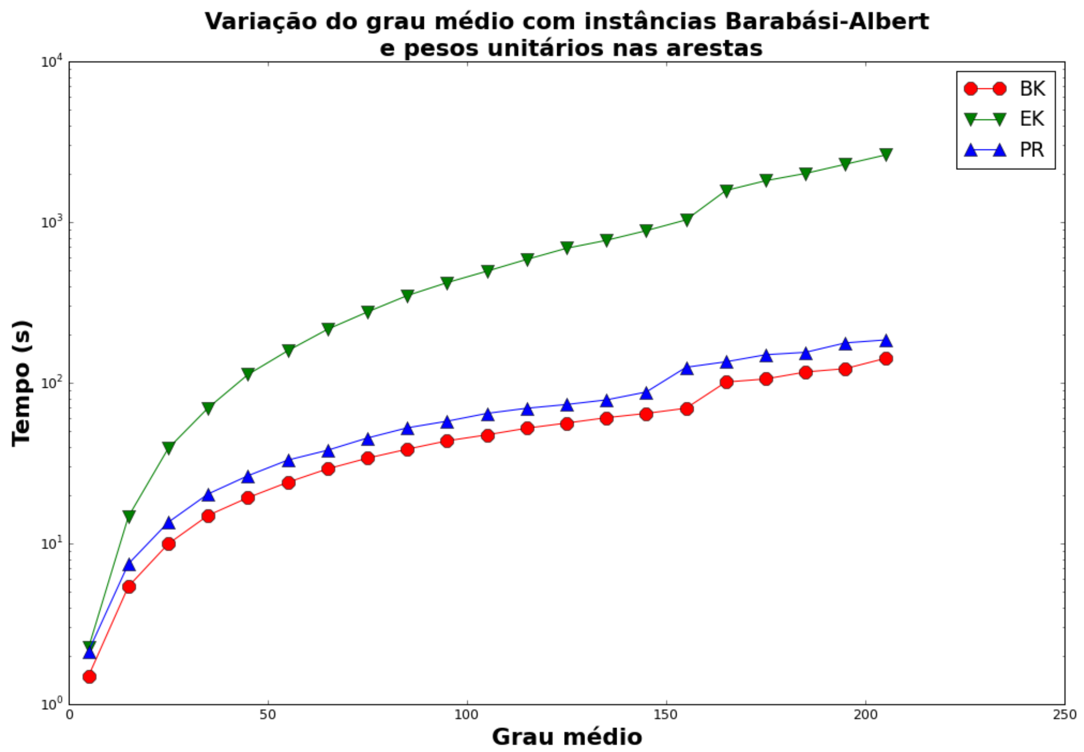


Figura 3.5: Comparativos dos algoritmos de fluxo máximo utilizando as instâncias Barabási-Albert com variação do grau médio e arestas com pesos unitários.

utilizado para o cálculo dos cortes mínimos.

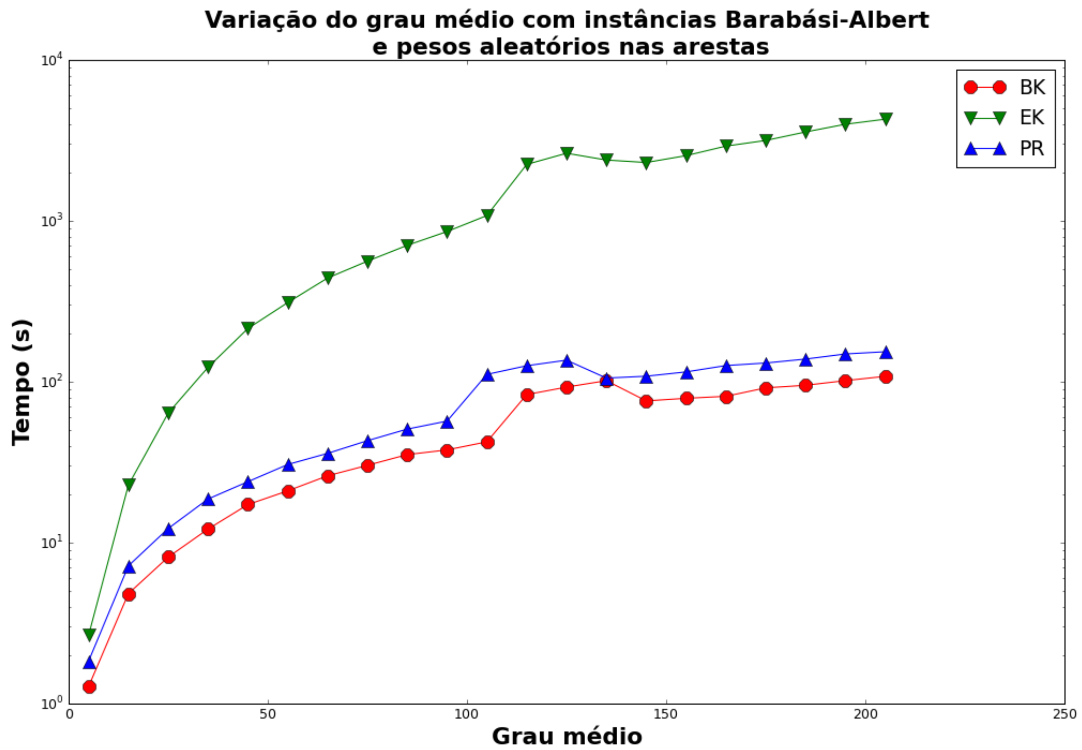


Figura 3.6: Comparativos dos algoritmos de fluxo máximo utilizando as instâncias Barabási-Albert com variação do grau médio e arestas com pesos aleatórios.

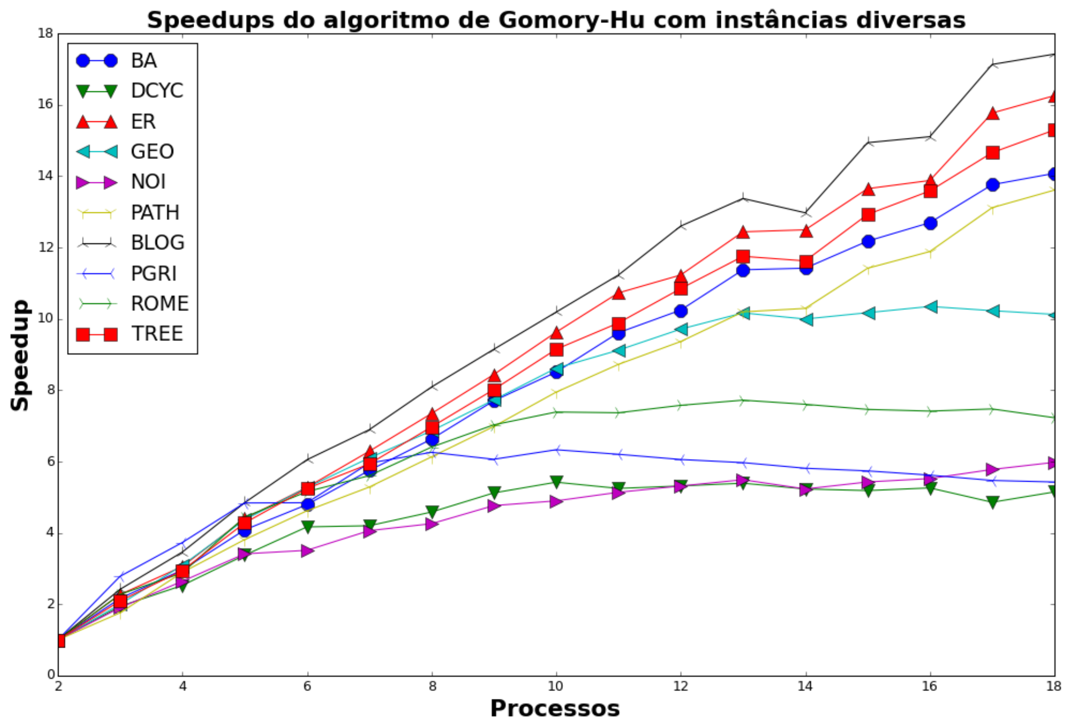


Figura 3.7: *Speedup* do algoritmo paralelo de Gomory-Hu.

CAPÍTULO 4

CONSTRUÇÃO PARALELA DE ÁRVORES DE CORTES UTILIZANDO CONTRAÇÕES DE GRAFO OTIMIZADAS

Resultados da literatura [9] mostram que a construção do grafo contraído é um dos pontos que consomem uma grande quantidade de tempo na execução da versão paralela do algoritmo de Gomory-Hu. Portanto, melhorar a eficiência deste procedimento é um caminho para trazer ganhos no tempo de execução. Implementar de forma eficiente as operações de contração no algoritmo de Gomory-Hu não é trivial. Neste capítulo é apresentada uma estratégia para a otimização da construção dos grafos contraídos na versão paralela do algoritmo. O capítulo está organizado da seguinte forma. Na seção 4.1 é descrita a especificação da estratégia proposta. Na seção 4.2 são apresentados os resultados experimentais com as versões do algoritmo paralelo de Gomory-Hu com e sem contrações otimizadas. Na seção 4.3 é descrito o algoritmo híbrido e na seção 4.4 os resultados experimentais da avaliação de contrações otimizadas no algoritmo híbrido.

4.1 Especificação da Estratégia Proposta

Na versão paralela do algoritmo de Gomory-Hu de [10], as contrações são realizadas sobre o grafo de entrada em todas tarefas executadas pelos processos escravos. O objetivo principal deste trabalho é especificar uma forma de realizar as contrações de forma eficiente, possibilitando aos processos escravos aproveitar instâncias de grafos contraídos para que novas contrações sejam realizadas sobre estes, e não sobre o grafo de entrada, sempre que possível. Como já vimos anteriormente na Seção 2.3, a construção do grafo contraído depende da partição de vértices construída a partir do pivô escolhido. Para melhor compreensão da otimização considere as seguintes definições:

Definição 1 *Dado um grafo $G = (V, E)$, dizemos que $G' = (V', E')$ é um grafo contraído a partir de G se:*

- *O conjunto de vértices V' de G' é uma partição de V*

- *Para cada* $\{u_1, u_2\} \in E$ *existe uma aresta* $\{U_1, U_2\} \in E'$ *se* $u_1 \in U_1$ *e* $u_2 \in U_2$

Definição 2 *Dadas duas partições* P *e* Q *de um conjunto* X*, dizemos que* P *é um refinamento de* Q *se todo elemento de* P *é subconjunto de algum elemento de* Q*.*

Por exemplo, a partição $\{\{1, 3\}, \{2, 4, 7\}, \{5\}, \{6, 8\}\}$ de $\{1, 2, 3, 4, 5, 6, 7, 8\}$ é um refinamento da partição $\{\{1, 3, 2, 4, 7\}, \{5, 6, 8\}\}$.

Definição 3 *Dizemos que o grafo contraído* $G'' = (V'', E'')$ *é um refinamento de* $G' = (V', E')$ *se* V'' *é um refinamento de* V'*.*

No contexto do algoritmo de Gomory-Hu paralelo, as operações de contração de grafos podem ser otimizadas, desde que o grafo contraído a ser construído seja um refinamento do grafo contraído utilizado no passo anterior. Além disso, é necessário um teste eficiente para determinar quais tarefas definem grafos contraídos que são refinamentos de outros. Seja $G'(X, T, G)$ o grafo contraído induzido pela escolha do pivô X na árvore T, uma árvore de cortes parcial de G. Então, o grafo contraído $G_1 = G'(X, T_1, G)$ é um refinamento do grafo contraído $G_2 = G'(Y, T_2, G)$, se, e somente se, o nodo pivô X está contido em Y. Esse fato indica que o mestre deve escolher um pivô cujos vértices devem estar contidos no pivô utilizado na tarefa anterior. Essa estratégia é suficiente para garantir que o novo grafo contraído seja um refinamento daquele utilizado na iteração anterior pelo mesmo processo. Assim, o grafo contraído é construído a partir do grafo da iteração anterior ao invés de ser construído a partir do grafo de entrada.

Para que o processo mestre possa fazer a escolha do pivô, ele necessita armazenar os últimos pivôs escolhidos nas tarefas enviadas aos processos escravos. O nodo escolhido para ser o novo pivô será aquele com o maior número de vértices dentre aqueles contidos no pivô da iteração anterior. Esta escolha aumenta a chance de futuros refinamentos serem encontrados.

Outra estratégia de otimização implementada consiste em não executar contrações que não produzam um grafo muito menor. Dessa forma, se a operação de contração não reduzir o número de vértices de uma constante k, a contração não é efetuada e o grafo da iteração anterior é utilizado no cômputo do corte mínimo. Nos experimentos relatados adiante, usa-se uma constante igual a 10.

O Algoritmo 4 corresponde à especificação do algoritmo paralelo proposto.

Algoritmo 4: Algoritmo Paralelo De Gomory-Hu Otimizado

```
   Entrada: G = (V, E_G, c_G) , proc_j, 0 ≤ j < p processos
   Saída: T = (V, E_T, c_T) uma árvore de cortes de G
 1 se proc_j = 0 // processo mestre
 2 então
 3 |  processos_estados[j] ← V para todo j, 1 ≤ j < p ;
 4 |  T ← {{V_G}, ∅} ;
 5 |  distribua tarefas para todos os processos;
 6 |  enquanto |V_T| < |V_G| faça
 7 |  |  receba de proc_j resposta (s,t,S), onde {S, S̄} é um s-t-corte mínimo de G ;
 8 |  |  se s e t pertencem ao mesmo nodo X de V_T // atualização da árvore
 9 |  |  então
10 |  |  |  X_s ← X ∩ S;
11 |  |  |  X_t ← X ∩ S̄;
12 |  |  |  e ← {X_s, X_t};
13 |  |  |  c(e) ← c(S, S̄);
14 |  |  |  para cada aresta e' = {X, Y} ∈ E_T incidente em X na árvore T faça
15 |  |  |  |  se Y ⊆ S então
16 |  |  |  |  |  E_T ← E_T ∪ {{X_s, Y}} \ {{X, Y}};
17 |  |  |  |  senão
18 |  |  |  |  |  E_T ← E_T ∪ {{X_t, Y}} \ {{X, Y}};
19 |  |  |  V_T ← (V_T \ {X}) ∪ {X_s, X_t};
20 |  |  |  E_T ← E_T ∪ {e};
21 |  |  se |V_T| = |V_G| então
22 |  |  |  envie mensagem de finalização ao processo proc_j;
23 |  |  senão
24 |  |  |  (X, refinar) ← escolhe_pivô(T, processos_estados[proc_s]);
25 |  |  |  partição ← constroi_partição(X, T) ;
26 |  |  |  (s, t) ← escolhe_par_st(X);
27 |  |  |  processos_estados[proc_j] ← X;
28 |  |  |  envie tarefa (s,t,partição,refinar) para processo proc_s;
29 |  retorne T ;
30 senão
   |  // processo escravo
31 |  enquanto receber tarefas faça
32 |  |  receba tarefa (s,t,partição,refinar);
33 |  |  se tarefa = fim então
34 |  |  |  termine;
35 |  |  se refinar então
36 |  |  |  G' ← refinar_grafo_contraído(G', partição) ;
37 |  |  senão
38 |  |  |  G' ← constrói_grafo_contraído(G, partição) ;
39 |  |  S ← corte_mínimo(G', s, t) ;
40 |  |  envie reposta (s,t,S) para proc_0;
```

O Algoritmo 4 recebe como entrada um grafo capacitado $G = (V, E_G, c_G)$ e devolve uma árvore de cortes $T = (V, E_T, c_T)$. Seja p o número total de processos, o processo $proc_0$

corresponde ao processo mestre e os processos $proc_j$, $1 \leq j \leq p$, os processos escravos. Na linha 3, o algoritmo inicializa o vetor *processos_estados* com o conjunto de vértices do grafo de entrada. Em seguida, o processo mestre envia *tarefas* contendo pares (s, t) de vértices para todos os processos escravos. Na linha 6, o processo mestre aguarda as respostas dos escravos. Ao receber uma resposta contendo um par (s, t) e um s-t corte mínimo $\{S, \overline{S}\}$, o mestre verifica se os vértices s e t encontram-se no mesmo nodo da árvore e, em caso afirmativo, a árvore é atualizada. A atualização é realizada da seguinte forma: toma-se o nodo $X \in T$ tal que $s, t \in X$. São criados os nodos X_s e X_t na árvore tal que X_s receberá os vértices de $X \cap S$ e X_t receberá os vértices de $X \cap \overline{S}$. Uma nova aresta $e = \{X_s, X_t\}$ em E_T com capacidade $c(e) = c(S, \overline{S})$ é adicionada à árvore T. Em seguida, entre as linhas 14 e 17, itera-se sobre todos as arestas $e' = \{X, Y\}$ atualizando-as conforme o lado do corte $\{S, \overline{S}\}$ em que os vértices de Y se encontram. Se após a atualização a condição $V_T = V_G$ for satisfeita, a árvore de cortes foi construída e o algoritmo aguarda as respostas restantes para enviar mensagem de finalização para todos os processos. Caso contrário, o processo mestre escolhe um novo nodo pivô X com procedimento *escolhe_pivô* para o processo $proc_j$, verificando o vértice do nodo armazenado em *processos_estados*$[j]$ que está contido em um nodo de maior cardinalidade. Se for possível encontrar um nodo pivô desta forma, o processo mestre enviará comando ao escravo para que este *refine* o grafo contraído, caso contrário, o processo escravo fará as contrações a partir do grafo de entrada. Após escolhido o novo nodo pivô, o vetor *processos_estados*$[j]$ é atualizado e uma nova tarefa é enviada ao processo $proc_j$.

Na linha 32 um processo escravo $proc_j$ recebe uma tarefa do processo mestre. Caso a tarefa recebida contenha uma informação indicando *fim* então o processo escravo finaliza. Em seguida, o processo escravo faz a construção do grafo contraído G' a partir do grafo contraído anterior através do procedimento *refinar_grafo_contraído*, se foi enviado na tarefa o comando para $refinar$. Caso contrário, o processo de contração será realizado sobre o grafo de entrada através do procedimento *constrói_grafo_contraído*. Após o processo de contração, o processo escravo $proc_j$ calcula o s-t corte mínimo $\{S, \overline{S}\}$ sobre o grafo G' e envia uma resposta (s, t, S) ao mestre.

4.2 Resultados Experimentais: Algoritmo Paralelo de Gomory-Hu

Nesta seção são reportados resultados obtidos a partir da execução do algoritmo em instâncias variadas, estão descritas na Tabela 3.2. O ambiente de execução foi um *cluster* homogêneo com 18 servidores biprocessados interligados por uma rede Ethernet de 1Gbps[1]. Os processadores são do tipo Intel® Xeon® Processor E5-2670[2] com *clock* de 2.60GHz com 8 núcleos e 16 *threads* cada. Cada servidor possui 128GB de memória RAM e 20MB de memória *cache*. Em todos os experimentos foram utilizados um processo por servidor.

O código foi escrito em C/C++ e compilado com `gcc` utilizando nível máximo de otimização do compilador (`-O3`). Os *speedups* foram calculados como $S = T_S/T_P$, tal que T_S corresponde ao tempo da implementação do algoritmo sequencial e T_P corresponde ao tempo de execução da implementação do algoritmo paralelo com P processos. Todos os experimentos foram executados 10 vezes cada. A versão *GH-Opt* é a implementação do algoritmo proposto neste trabalho e o algoritmo *GH* é a implementação sem otimização.

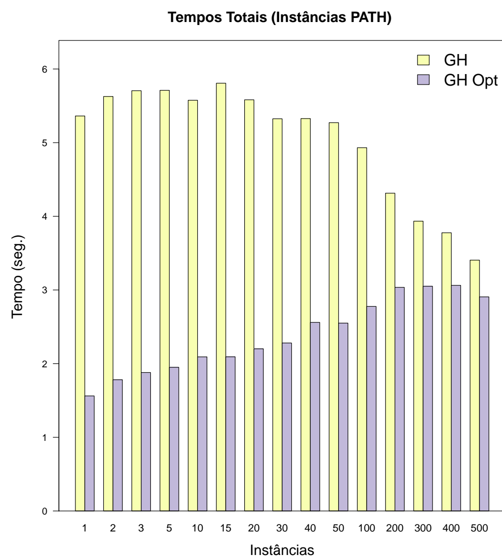


Figura 4.1: Tempos totais de execução para os algoritmos *GH* e *GH-Opt* utilizando instâncias do tipo *PATH* variadas.

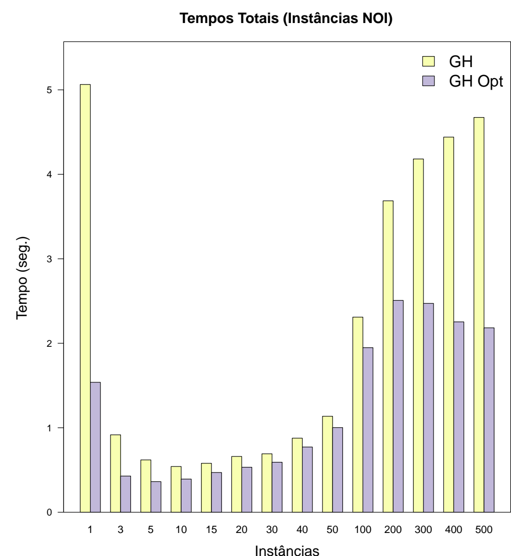


Figura 4.2: Tempos totais de execução para os algoritmos *GH* e *GH-Opt* utilizando instâncias do tipo *NOI* variadas.

[1] *Cluster* do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é financiado pela FINEP através dos projetos CT-INFRA/UFPR

[2] Intel® Xeon® Processor E5-2670 `http://ark.intel.com/pt-br/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI`

As figuras 4.1, 4.2 e 4.3 mostram os tempos totais de execução para as duas versões do algoritmo, com e sem as contrações de vértices otimizadas, utilizando instâncias do tipo *PATH*, *NOI* e *TREE*, respectivamente, com 1000 vértices e densidade de 20% variando o parâmetro k.

No gráfico da Figura 4.1 é possível verificar que o desempenho das duas versões do algoritmo, com e sem contrações otimizadas, tende a convergir, uma vez que, dadas as características da instância *PATH*, à medida em que o parâmetro k, neste caso referente ao tamanho do caminho, aumenta, a quantidade de s-t cortes mínimos que separam poucos vértices também aumenta, degradando a eficiência da otimização que não consegue reaproveitar as instâncias de grafos contraídos.

No gráfico da Figura 4.2 constata-se o ganho da otimização em todas as instâncias geradas. Dadas as características das instâncias *NOI* utilizadas para esse experimento, o desempenho das duas versões do algoritmo é muito bom para parâmetro k, que neste caso refere-se ao tamanho dos agrupamentos de vértices, entre $1 < k \leq 50$. Particularmente para $k = 1$, os grafos são aleatórios e sem muitos cortes mínimos que possibilitam separar muitos vértices, ainda assim, a diferença de desempenho da versão otimizada foi bem expressiva frente a versão não otimizada. Esta diferença se dá pelo fato da heurística de escolha de pares de vértices da versão otimizada, que proporcionou bom reaproveitamento de instâncias de grafos contraídos mesmo com cortes mínimos que separam poucos vértices. Para o valor de $k = 500$ o grafo terá dois agrupamentos de vértices, uma vez que número de vértices é 1000, e, portanto, terá apenas um corte mínimo balanceado. De maneira geral o algoritmo proposto neste trabalho, descrito como *GH-Opt*, obteve os melhores resultados em todas as instâncias nesses experimentos.

O gráfico da Figura 4.3 apresenta os tempos do algoritmo paralelo de Gomory-Hu com e sem contrações otimizadas para as instâncias do tipo *TREE*. Para todos os valores de k utilizados na geração das instâncias, o desempenho da versão otimizada foi significativamente melhor do que a versão não otimizada.

A Figura 4.4 mostra os tempos totais de execução para as duas versões do algoritmo utilizando as instâncias da Tabela 3.2. Para a maioria das instâncias, o algoritmo *GH-Opt* obteve melhores resultados. Apenas nas instâncias *PGRI* e *ROME* o algoritmo *GH-Opt* finalizou com tempos de execução maiores do que a versão *GH* e nas instâncias *GEO* e *DCYC* cujo os tempos foram iguais para ambas as implementações. As instâncias

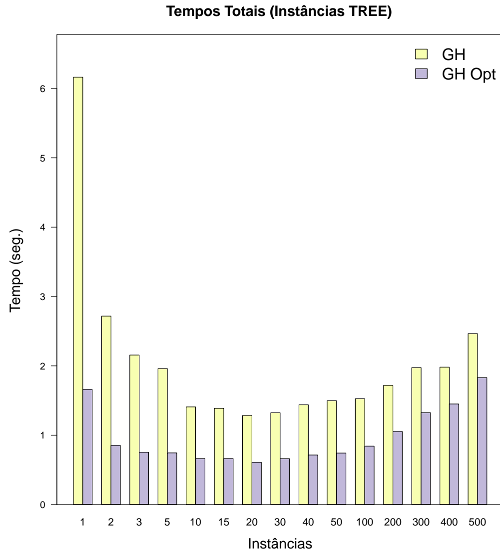


Figura 4.3: Tempos totais de execução para os algoritmos *GH* e *GH-Opt* utilizando instâncias do tipo *TREE* variadas.

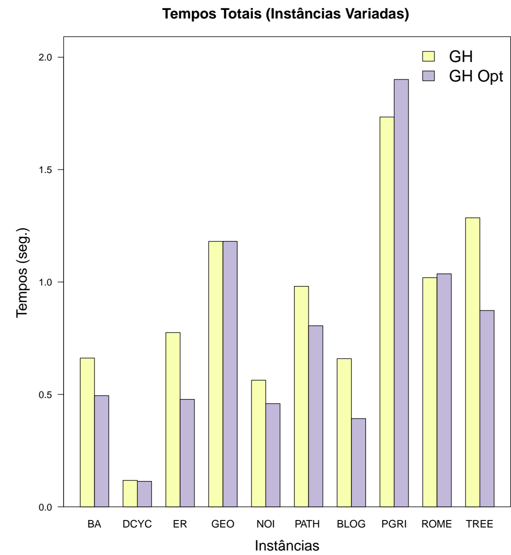


Figura 4.4: Tempos totais de execução para para os algoritmos *GH* e *GH-Opt* com as instâncias da Tabela 3.2.

ROME e *PGRI* correspondem a modelos de grafos reais, como esses tipos de grafos têm a característica de possuir a maioria dos cortes mínimos pouco balanceados, ou seja, que separam poucos vértices, a otimização não consegue um bom reaproveitamento dos grafos contraídos e consequentemente acaba não apresentando bom desempenho.

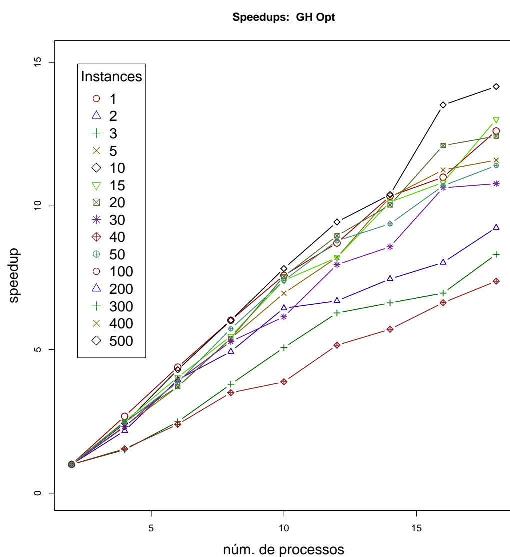


Figura 4.5: *Speedup* do algoritmo *GH-Opt* utilizando instâncias do tipo *NOI* com 2000 vértices.

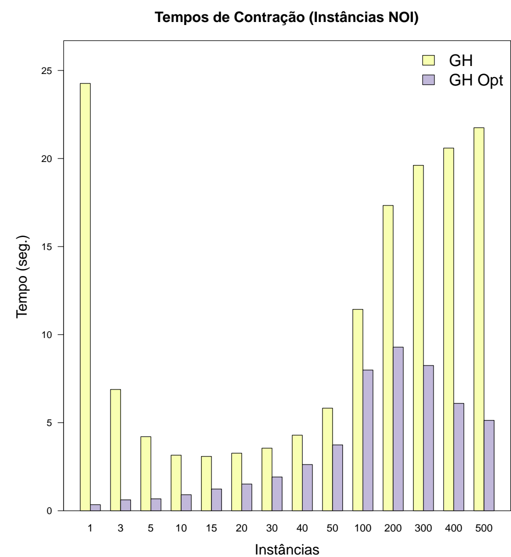


Figura 4.6: Tempos médios de contração para os algoritmos *GH* e *GH-Opt* utilizando instâncias do tipo *NOI*.

Na Figura 4.5 é representado o *speedup* do algoritmo Gomory-Hu com contrações oti-

mizadas utilizando instâncias do tipo *NOI* com 2000 vértices e densidade de 20% variando o parâmetro k. Para o limite de 18 máquinas utilizadas na execução dos experimentos os *speedups* foram próximos de lineares.

Na Figura 4.6 são representados os tempos médios de contração nos experimentos com as instâncias do tipo *NOI*. É possível constatar os ganhos que a estratégia proposta concede à implementação refletindo nos tempos totais do algoritmo. O baixo tempo médio de contração é dado pela implementação da estratégia em somente realizar as contrações quando for identificado que serão contraídos mais do que 10 vértices, caso contrário o grafo contraído do passo anterior será reaproveitado, economizando o tempo de contração.

4.3 Algoritmo Híbrido Utilizando Contrações Otimizadas

O algoritmo de Gusfield não gasta tempo fazendo contrações e calcula os cortes mínimos diretamente sobre o grafo de entrada. O algoritmo de Gomory-Hu contrai o grafo de entrada e calcula os cortes mínimos sobre um grafo que, possivelmente, será menor do que o grafo de entrada. Nenhum dos dois algoritmos domina o tempo sobre o outro, afinal o tempo economizado com as contrações no algoritmo de Gusfield é compensado pelos cálculos dos cortes mínimos sobre uma instância de grafo com mais vértices e arestas. Por outro lado, o algoritmo de Gomory-Hu pode gastar muito tempo contraindo e não reduzir significativamente o tamanho do grafo. A diferença de desempenho dos dois algoritmos depende muito das características das instâncias, assim, nas instâncias em que for possível reduzir significativamente o grafo através das contrações o algoritmo de Gomory-Hu terá melhor desempenho e nas instâncias em que não for possível, o algoritmo de Gusfield será melhor.

Cohen [9] apresenta um algoritmo híbrido que combina as técnicas utilizadas nos algoritmos de Gusfield e Gomory-Hu. Este algoritmo faz com que as contrações fiquem condicionadas ao número de vértices do grafo contraído, de forma que se forem inferiores a um limiar T, $0 \leq T \leq 1$, o algoritmo não faz a contração e realiza o cálculo do corte mínimo sobre o grafo de entrada diretamente. O algoritmo híbrido paraleliza os cálculos dos s-t cortes mínimos da mesma forma que a versão paralela do algoritmo de Gomory-Hu, porém no processo mestre é decidido se os processos escravos deverão utilizar o grafo de entrada ou contraí-lo. As contrações, quando necessárias, são feitas sempre sobre o

grafo de entrada. Calcular os *s*-*t* cortes mínimos diretamente sobre o grafo de entrada é uma característica do algoritmo de Gusfield, assim nessas situações o algoritmo não gasta tempo contraindo quando detecta que não obterá vantagem com as contrações. Com esta abordagem, quando algoritmo híbrido não for o melhor, através de análises experimentais apresentadas em [9], ele sempre terá tempo de execução próximo do algoritmo que obtiver o melhor tempo, uma vez que dependendo das características das instâncias ou o algoritmo de Gomory-Hu será melhor, ou o algoritmo de Gusfield será melhor.

Neste trabalho foi implementada a estratégia proposta no algoritmo híbrido apresentado em [9]. Na próxima seção são apresentados os resultados experimentais comparando os tempos do algoritmo híbrido com e sem contrações otimizadas.

4.4 Resultados Experimentais: Algoritmo Híbrido Utilizando Contrações Otimizadas

Para execução dos experimentos foram utilizadas instâncias variadas que estão descritas na Tabela 3.2. O ambiente de execução foi um *cluster* homogêneo com 18 servidores biprocessados interligados por uma rede Ethernet de 1Gbps[3]. Os processadores são do tipo Intel® Xeon® Processor E5-2670[4] com *clock* de 2.60GHz com 8 núcleos e 16 *threads* cada. Cada servidor possui 128GB de memória RAM e 20MB de memória *cache*. Em todos os experimentos foram utilizados um processo por servidor.

O código foi escrito em C/C++ e compilado com `gcc` utilizando nível máximo de otimização do compilador (`-O3`). Os *speedups* foram calculados como $S = T_S/T_P$, tal que T_S corresponde ao tempo da implementação do algoritmo sequencial e T_P corresponde ao tempo de execução da implementação do algoritmo paralelo com P processos. Todos os experimentos foram executados 10 vezes cada. A versão *Hybrid-Opt* é a implementação do algoritmo híbrido utilizando a estratégia proposta neste trabalho e o algoritmo *Hybrid* é a implementação sem otimização.

A Figura 4.7 apresenta os gráficos comparativos dos tempos totais dos algoritmos *Hybrid* e *Hybrid-Opt* utilizando instâncias do tipo *PATH*. Neste experimento a estratégia proposta proporcionou uma pequena melhora nos resultados para a maioria das instâncias

[3] *Cluster* do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é financiado pela FINEP através dos projetos CT-INFRA/UFPR

[4] Intel® Xeon® Processor E5-2670 `http://ark.intel.com/pt-br/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI`

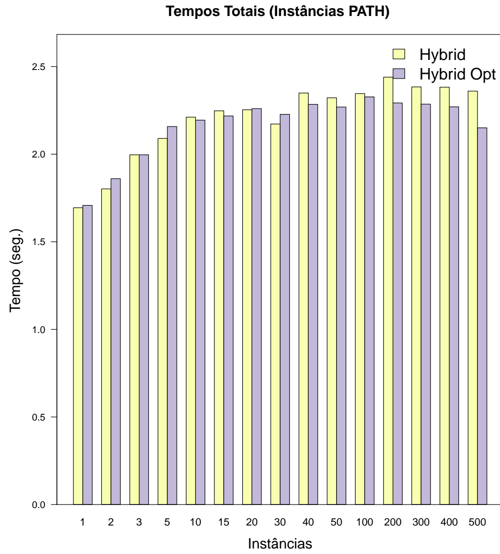


Figura 4.7: Tempos totais de execução para os algoritmos *Hybrid* e *Hybrid-Opt* utilizando instâncias do tipo *PATH* variadas.

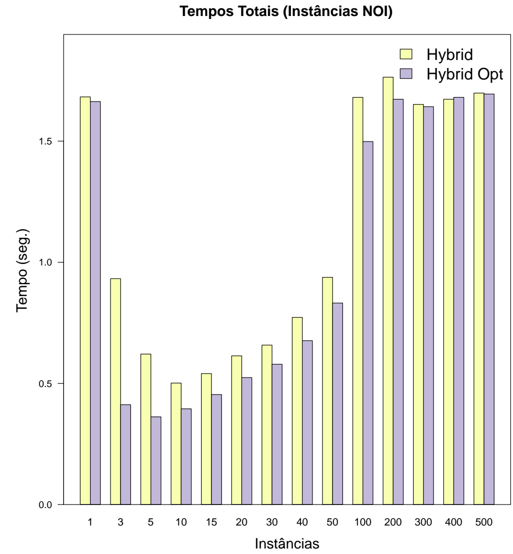


Figura 4.8: Tempos totais de execução para os algoritmos *Hybrid* e *Hybrid-Opt* utilizando instâncias do tipo *NOI* variadas.

geradas.

A Figura 4.8 apresenta os gráficos comparativos dos tempos totais dos algoritmos híbrido e híbrido otimizado utilizando instâncias do tipo *NOI*. Neste experimento a estratégia proposta proporcionou melhores resultados para a maioria das instâncias geradas.

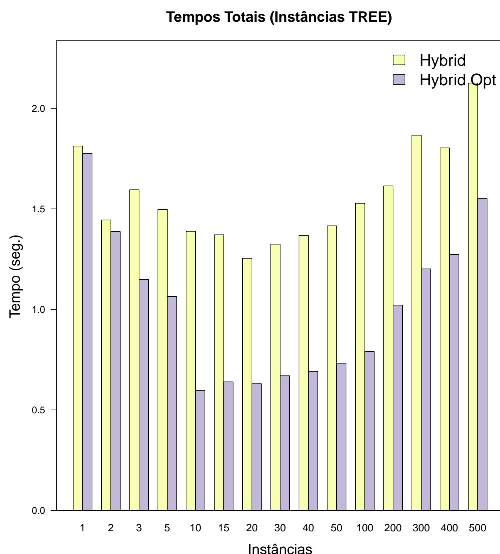


Figura 4.9: Tempos totais de execução para os algoritmos *Hybrid* e *Hybrid-Opt* utilizando instâncias do tipo *TREE* variadas.

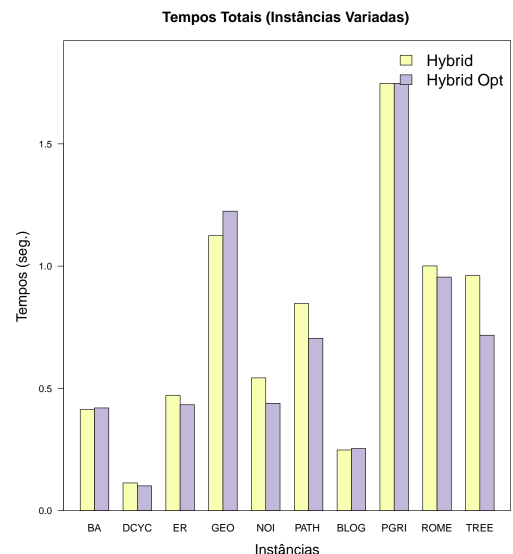


Figura 4.10: Tempos totais de execução para para os algoritmos *Hybrid* e *Hybrid-Opt* com as instâncias da Tabela 3.2.

A Figura 4.9 apresenta os gráficos comparativos dos tempos totais dos algoritmos híbrido e híbrido otimizado utilizando instâncias do tipo *TREE*. Neste experimento a estratégia proposta proporcionou melhores resultados para todas instâncias geradas. Para essas instâncias a versão otimizada apresentou diferenças bem significativas na maioria dos casos.

A Figura 4.10 mostra os tempos totais de execução para as duas versões do algoritmo híbrido utilizando as instâncias da Tabela 3.2. Nas as instâncias *DCYC*, *ER*, *NOI*, *PATH*, *ROME* e *TREE* a versão otimizada do algoritmo obteve melhores resultados.

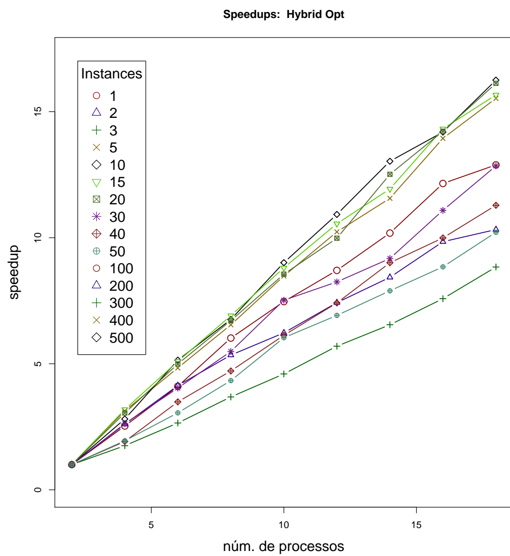


Figura 4.11: *Speedup* do algoritmo *Hybrid-Opt* utilizando instâncias do tipo *NOI* com 2000 vértices.

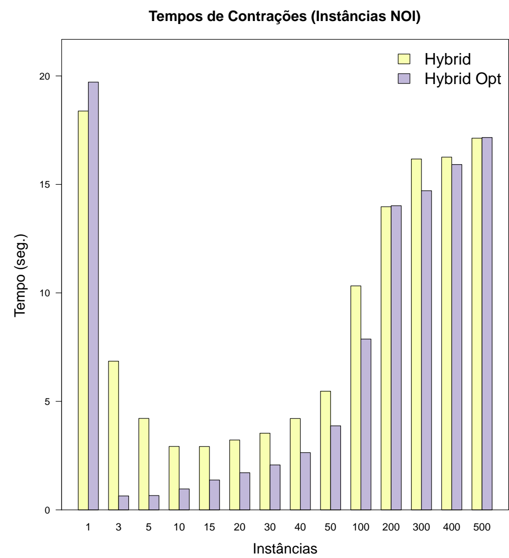


Figura 4.12: Tempos médios de contração para os algoritmos *Hybrid* e *Hybrid-Opt* utilizando instâncias do tipo *NOI*.

A Figura 4.12 apresenta os comparativos dos tempos médios de contração para as duas versões do algoritmo híbrido utilizando as instâncias da Tabela 3.2. É possível verificar que, apesar da versão otimizada do algoritmo apresentar tempos totais muito próximos da versão não otimizada, a maioria dos testes apresentou ganhos positivos no tempo médio de contração.

CAPÍTULO 5

CONCLUSÃO

Neste trabalho foi proposta uma estratégia eficiente para realizar as contrações de grafos na construção paralela de árvores de corte. A ideia principal é fazer com que os processos escravos aproveitem instâncias de grafos contraídos de passos anteriores sem a necessidade de refazer todas as contrações no grafo de entrada durante a execução de cada tarefa. De forma geral, a implementação desta otimização requer que o processo mestre tenha controle sobre as tarefas que foram enviadas para cada escravo, armazenando-as para que sejam utilizadas quando as respostas chegarem. Os processos escravos, por sua vez, precisam tomar decisões sobre quando aproveitar, ou não, uma instância de grafo contraído.

Para avaliar a eficiência desta versão foram realizados experimentos em um *cluster* de alto desempenho. Foram realizados testes de *speedup* e comparativos entre as versões paralelas. Os experimentos foram feitos em vários conjuntos de instâncias que incluíram grafos sintéticos e grafos reais. Foram escolhidas três tipos de instâncias de grafos sintéticos para testar as versões paralelas do algoritmo utilizando variações dessas instâncias. Nos experimentos realizados com as instâncias *NOI*, *PATH* e *TREE* as versões do algoritmo com a otimização proposta apresentaram melhores resultados no tempo de execução para a maioria das instâncias. Para as instâncias *GEO*, *BLOG*, *PGRI* e *ROME*, que representam grafos reais, as versões otimizadas não desempenharam resultados melhores do que as versões não otimizadas do algoritmo. Nas instâncias de grafos reais, o algoritmo de Gomory-Hu encontra muitos cortes que separam poucos vértices, dificultando o aproveitamento dos grafos contraídos e, na maioria dos casos, realizando as contrações diretamente sobre o grafo de entrada. Ainda assim, nos comparativos dos tempos médios de contração foi possível constatar que a otimização proporciona ganhos nos tempos de contração, uma vez que o algoritmo realiza as contrações sobre grafos de tamanho menor sempre que possível. Nos experimentos de escalabilidade, variando o número de processos, os resultados demonstraram *speedups* próximos de lineares para a maioria das instâncias.

De forma geral, as comparações permitem comprovar os ganhos obtidos com a es-

tratégia proposta, principalmente para os grafos onde o algoritmo consegue encontrar cortes mínimos balanceados que produzem grafos contraídos de tamanhos reduzidos.

Uma outra contribuição deste trabalho é uma implementação do algoritmo de Gomory-Hu paralelo utilizando a biblioteca Boost. Com essa implementação foram comparadas as implementações dos algoritmos de *fluxo máximo* existentes nessa biblioteca. Os resultados experimentais permitiram avaliar o desempenho do algoritmo paralelo de Gomory-Hu utilizando diferentes algoritmos de *fluxo máximo*. Através dos comparativos realizados, foi possível constatar que a implementação do algoritmo de *fluxo máximo* de Boykov e Kolmogorov [6] obteve os melhores resultados em todos os testes.

Para trabalhos futuros ainda são possíveis outras otimizações para a versão paralela do algoritmo de Gomory-Hu. Particularmente relevante é a paralelização do processo mestre, já que esse pode tornar-se um gargalo a partir de um determinado número de processos escravos. Neste sentido seria necessário, através de análises experimentais, conhecer o limite máximo de processos escravos que o mestre suporta, para então através de novos testes avaliar a viabilidade da paralelização do mesmo. É possível, também, ser considerada como alternativa para paralelizar a construção do grafo contraído a utilização da biblioteca OpenMP. Esta biblioteca é uma especificação de API que permite criar programas paralelos no paradigma de memória compartilhada. Combinar OpenMP e MPI traz um alto grau de paralelismo para a implementação. Apesar da alta dependência de dados durante o processo de contração, acreditamos que é possível, através da enumeração do conjunto de arestas E_G dividir o esforço em várias *threads*. Para alcançar esse objetivo, uma *thread* não pode manipular a lista de adjacências de um mesmo vértice de forma concorrente. Este requisito exige que as listas de adjacências sejam *thread-safe*, ou seja, deve haver um controle de acesso na leitura e escrita concorrentes das estruturas de dados, fazendo com que sejam necessários pontos de exclusão mútua no código.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] David Abrahams e Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] Lada A. Adamic e Natalie Glance. "The Political Blogosphere and the 2004 U.S. Election: Divided They Blog". Em: *Proceedings of the 3rd International Workshop on Link Discovery*. ACM, 2005, pp. 36–43.

[3] Réka Albert e Albert-lászló Barabási. "Statistical mechanics of complex networks". Em: *Reviews of Modern Physics* (2001), p. 54.

[4] Vladimir Batagelj e Andrej Mrvar. *Pajek datasets*. URL: `http://vlado.fmf.uni-lj.si/pub/networks/data/` (acesso em 16/11/2015).

[5] Béla Bollobás. *Random Graphs*. Second. Cambridge Books Online. Cambridge University Press, 2001.

[6] Yuri Boykov e Vladimir Kolmogorov. "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision". Em: *IEEE Trans. Pattern Anal. Mach. Intell.* 26.9 (2004), pp. 1124–1137.

[7] B. Chapman, G. Jost e R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific Computation Series v. 10. MIT Press, 2008.

[8] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine e Cliff Stein. "Experimental Study of Minimum Cut Algorithms". Em: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '97. New Orleans, Louisiana, USA: Society for Industrial e Applied Mathematics, 1997, pp. 324–333.

[9] Jaime Cohen. "Algoritmos Paralelos Para Árvores De Cortes E Medidas De Centralidade Em Grafos". Tese de doutorado. Universidade Federal do Paraná - UFPR, mar. de 2013.

[10] Jaime Cohen, Luiz A. Rodrigues e Elias P. Duarte Jr. “A Parallel Implementation of Gomory-Hu’s Cut Tree Algorithm”. Em: *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing.* SBAC-PAD ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 124–131.

[11] Jaime Cohen, Luiz A. Rodrigues, Fabiano Silva, Renato Carmo, André L. P. Guedes e Elias P. Duarte. “Parallel implementations of gusfield’s cut tree algorithm”. Em: *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I.* ICA3PP’11. Melbourne, Australia: Springer-Verlag, 2011, pp. 258–269.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, 2009.

[13] E.P. Duarte Jr, R. Santini e J. Cohen. “Delivering packets during the routing convergence latency interval through highly connected detours”. Em: *Dependable Systems and Networks, 2004 International Conference on.* 2004, pp. 495–504.

[14] Jack Edmonds e Richard M. Karp. “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. Em: *J. ACM* 19.2 (1972), pp. 248–264.

[15] Roee Engelberg, Jochen Könemann, Stefano Leonardi e Joseph(Seffi) Naor. “Cut Problems in Graphs with a Budget Constraint”. Em: *LATIN 2006: Theoretical Informatics.* Vol. 3887. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 435–446.

[16] L. R. Ford e D. R. Fulkerson. “Maximal flow through a network”. Em: *Journal canadien de mathématiques* 8 (1956), pp. 399–404.

[17] Andrew V. Goldberg e Robert E. Tarjan. “A New Approach to the Maximum-flow Problem”. Em: *J. ACM* 35.4 (1988), pp. 921–940.

[18] Andrew V. Goldberg e Kostas Tsioutsiouliklis. “Cut Tree Algorithms”. Em: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA ’99. Baltimore, Maryland, USA: Society for Industrial e Applied Mathematics, 1999, pp. 376–385.

[19] R. E. Gomory e T. C. Hu. “Multi-Terminal Network Flows”. Em: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 551–570.

[20] William Gropp, Ewing Lusk e Anthony Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. Cambridge, MA, USA: MIT Press, 1999.

[21] D. Gusfield. “Very simple methods for all pairs network flow analysis”. Em: *SIAM Journal on Computing* 19.1 (1990), pp. 143–155.

[22] Krishna Yeshwanth Kamath e James Caverlee. “Transient Crowd Discovery on the Real-time Social Web”. Em: *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*. WSDM ’11. ACM, 2011, pp. 585–594.

[23] Vipin Kumar. *Introduction to Parallel Computing*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[24] Timothy Mattson, Beverly Sanders e Berna Massingill. *Patterns for Parallel Programming*. First. Addison-Wesley Professional, 2004.

[25] H. Nagamochi e T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Algorithmic Aspects of Graph Connectivity. Cambridge University Press, 2008.

[26] Hiroshi Nagamochi, Tadashi Ono e Toshihide Ibaraki. “Implementing an efficient minimum capacity cut algorithm”. Em: *Mathematical Programming* 67.1-3 (1994), pp. 325–341. ISSN: 0025-5610.

[27] Gianni Storchi, Paolo Dell’Olmo e Monica Gentili. *Road network of the city of rome*. 1999. URL: `http://www.dis.uniroma1.it/challenge9/data/rome/rome99.gr` (acesso em 16/11/2015).

[28] Nurcan Tuncbag, F. Sibel Salman, Ozlem Keskin e Attila Gursoy. “Analysis and network representation of hotspots in protein interfaces using minimum cut trees”. Em: *Proteins: Structure, Function, and Bioinformatics* 78.10 (2010), pp. 2283–2294.

[29] Duncan J Watts e Steven H Strogatz. “Collective dynamics of ‘small-world’ networks”. Em: 393.June (1998), pp. 440–442.