UNIVERSIDADE FEDERAL DO PARANÁ

JAIME COHEN

ALGORITMOS PARALELOS PARA ÁRVORES DE CORTES E MEDIDAS DE CENTRALIDADE EM GRAFOS

CURITIBA

2013

JAIME COHEN

ALGORITMOS PARALELOS PARA ÁRVORES DE CORTES E MEDIDAS DE CENTRALIDADE EM GRAFOS

Tese de doutorado apresentada ao Programa de Pós-Graduação em Informática, Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Elias P. Duarte Jr.

CURITIBA

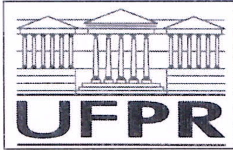
2013

Cohen, Jaime
Algoritmos paralelos para árvores de cortes e medidas de centralidade em grafos / Jaime Cohen. - Curitiba, 2013.
127 f. : il., graf., tab.

Tese (doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Elias Procópio Duarte Júnior

1. Teoria dos grafos. 2. Árvores (Teoria dos grafos). 3. Algoritmos paralelos. I. Duarte Júnior, Elias Procópio. II. Título.

CDD 005.275



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

PARECER

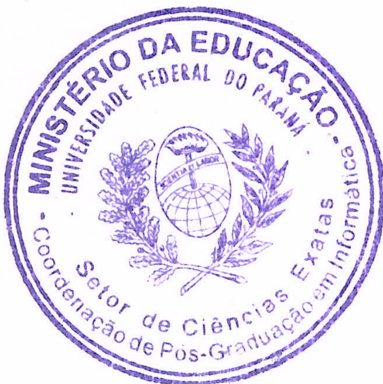
Nós, abaixo assinados, membros da Banca Examinadora da defesa do aluno de Doutorado em Ciência da Computação, Jaime Cohen, avaliamos a tese de doutorado intitulada *"Algoritmos Paralelos para Árvores de Cortes e Medidas de Centralidade em Grafos"*, cuja defesa pública foi realizada no dia 15 de março de 2013, às 13:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após avaliação, decidimos pela **aprovação** do candidato.

Curitiba, 15 de março de 2013.

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR – Orientador

Prof. Dr. Siang Wun Song
IME/USP – Membro Externo

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



Prof. Dr. Renato Carmo
DINF/UFPR – Membro Interno

Prof. Dr. André Luiz Pires Guedes
DINF/UFPR – Membro Interno

Esta tese é dedicada

à Sofia Pessoa Cohen, minha filha.

Agradecimentos

Primeiramente, agradeço ao Prof. Elias Procópio Duarte Júnior, meu orientador e amigo, pelo apoio incondicional que me deu desde muito antes de eu iniciar esta tese.

Eu sou imensamente grato aos colegas Jonatan Schroeder, Karine Pires e Luiz A. Rodrigues que colaboraram com a execução da pesquisa, ajudaram a melhorar a qualidade do trabalho e cujas contribuições foram importantes para que o trabalho fosse concluído dentro do prazo. Agradeço ao Guilherme Galante, colega do Larsis/UFPR, que se prontificou a apresentar em congresso um de nossos trabalhos quando não pudemos estar presentes para apresentá-lo.

Agradeço aos professores do Departamento de Informática da UFPR que direta ou indiretamente contribuíram para este trabalho.

Aos membros da banca examinadora, professores Siang W. Song, Murilo V. G. da Silva, Renato Carmo e André L. P. Guedes, os meus sinceros agradecimentos pela participação na banca e pelas sugestões feitas ao trabalho.

Agradeço ao Prof. Marcos Sunye, ex-coordenador do Programa de Pós-Graduação em Informática da UFPR, pelo esforço de obter a bolsa de estudos que me foi oferecida. Agradeço à Fundação Araucária pela concessão da bolsa.

Agradeço aos funcionários do Departamento de Informática da UFPR, em particular à Jucelia Miecznikowski e ao Rafael A. Pereira por serem sempre muito prestativos e por ajudarem a viabilizar as viagens aos congressos dos quais participei nos últimos anos.

Agradeço aos professores do Departamento de Informática da Universidade Estadual de Ponta Grossa pela aprovação da minha licença para cursar o doutorado.

Agradeço a minha família, aos meus pais, Saadia Cohen e Geni Cohen, a minha irmã Alice e ao meu tio José Schichman pelo apoio que sempre deram. Agradeço a minha esposa Ana Carolina que muitas alegrias tem me dado e que trouxe à vida a nossa amada filha Sofia, a quem dedico esta tese.

Sumário

Lista de Símbolos .... **ix**

Resumo .... **xi**

Abstract .... **xii**

1 Introdução **1**

1.1 Definições Preliminares: Grafos, Cortes, Fluxos e Conectividade 4

1.2 Organização deste Trabalho 8

2 Algoritmos para Árvores de Cortes **10**

2.1 O Algoritmo de Gusfield 10

2.2 Algoritmo de Gomory-Hu 13

2.3 Algoritmos Paralelos para Árvores de Cortes 16

2.4 Revisão da Literatura 19

3 Implementações Paralelas do Algoritmo de Gusfield para Árvores de Cortes **23**

3.1 Versões Paralelas do Algoritmo de Gusfield 23

3.1.1 Versão Paralela do Algoritmo de Gusfield: *MPI* 24

3.1.2 Versão Paralela do Algoritmo de Gusfield: *OpenMP* 26

3.2 Resultados Experimentais 27

4 Implementações Paralelas do Algoritmo de Gomory-Hu para Árvores de Cortes **43**

4.1 Uma Versão Paralela do Algoritmo de Gomory-Hu 43

4.2 Resultados Experimentais: Gomory-Hu Paralelo 50

4.2.1 Instâncias Variadas 51

4.2.2 Instâncias NOI 57

4.3 Algoritmo Híbrido para Construção de Árvores de Cortes 60

4.3.1 Resultados Experimentais: Algoritmo Híbrido 60

4.4 Heurísticas para o Algoritmo Paralelo de Gomory-Hu 65

4.4.1 Enumeração dos s-t-Cortes Mínimos 65

4.4.2 Escolha do Pivô e dos Vértices a Separar 68

4.4.3 Outras Heurísticas 69

4.5 Implementação 72

4.6 Conclusão 75

5 Medidas de Centralidade em Grafos 77

5.1 Medidas de Centralidade: Definições, Propriedades e Classificação 77

5.1.1 Medidas de Centralidade: Exemplos 78

5.1.2 Uma Classificação de Medidas de Centralidade 80

5.1.3 Medidas Locais e Globais 80

5.2 Medidas Baseadas em Conectividade 81

5.2.1 Medidas Baseadas em Cortes de Arestas 82

5.2.2 Medidas Baseadas em Cortes de Vértices 87

5.2.3 Grafos Orientados: Medidas Simétricas e Assimétricas 89

6 Algoritmos para Medidas de Conectividade 91

6.1 Algoritmos para Medidas de Conectividade de Arestas 91

6.1.1 Árvore de Componentes de Máxima Conectividade 94

6.2 Algoritmos para Medidas de Conectividade de Vértices 99

7 Avaliação Empírica das Medidas de Conectividade 101

7.1 Conectividade de Redes Complexas 101

7.2 Conectividade de Redes Sintéticas ... 103
7.3 Experimentos com Conectividade de Vértices ... 110
7.4 Comentários Finais sobre os Experimentos com as Medidas de Conectividade ... 113
8 Conclusão ... 117
Lista de Publicações ... 119
Referências Bibliográficas ... 120

Lista de Símbolos

Símbolo	Descrição
$\mathbb{Q}$	Conjunto dos números racionais.
$\mathbb{Q}_{\geq 0}$	Conjunto dos números racionais não negativos.
$\lvert A \rvert$	Cardinalidade do conjunto A.
$A \backslash B$	Diferença dos conjuntos A e B, isto é, $A \backslash B = \{x \mid x \in A \text{ e } x \notin B\}$.
$G = (V_G, E_G)$	Um multigrafo ou um grafo orientado.
$G = (V_G, E_G, c_G)$	Um multigrafo ou grafo orientado e capacitado.
$c_G(e)$	Capacidade da aresta e.
$\overline{X}$	Complemento do conjunto de vértices X em relação a V_G, isto é, $V_G \backslash X$.
$d_G(u,v)$	Comprimento de um caminho mínimo de u a v em um grafo G.
$E_G(X,Y)$	$\{\{u,v\} \in E(G) \mid u \in X \text{ e } v \in Y\}$, isto é, as arestas com um vértice em X e o outro em Y.
$E_G(X)$	$E_G(X,\overline{X})$.
$d(X,Y)$	$\sum_{e \in E_G(X,Y)} c(e)$.
$d(X)$	$d(X,\overline{X})$.
$d(v)$	Grau ponderado do vértice v, i.e., $d(v) = d(\{v\})$.
$V(G)$	Conjunto de vértices do grafo G.
$E(G)$	Conjunto de arestas do grafo G.
$G - X$	Grafo obtido pela remoção dos vértices em X e das arestas incidentes a eles.
G/X	Grafo obtido pela contração dos vértices em X no grafo G.
2^V	Conjunto de todos os subconjuntos de V.
$\lambda_G(s,t)$	Aresta-conectividade local entre s e t em G.
$\lambda(G)$	Aresta-conectividade de G.
$\kappa_G(s,t)$	Vértice-conectividade local entre s e t em G.
$\kappa(G)$	Vértice-conectividade de G.

$G/X_1,X_2,\ldots,X_k$	Grafo obtido pela contração de cada $X_1,X_2,\ldots,X_k$ em G.
$N(v,r)$	$\{u \in V \mid d(v,u) \leq r\}$.
$\lambda_G(X)$	Aresta-conectividade de $X \subseteq V$ em G.
$\lambda_i(v)$	i-aresta-conectividade do vértice v.
$MCC_i(v)$	Componente de máxima conectividade de índice i que contém v.
$LCC_k(v)$	Maior componente de conectividade k que contém v.
$\lambda_i(v,\ V')$	i-aresta-conectividade de v restrita a algum conjunto de vértices $V' \subseteq V$.
$\overline{\lambda}(v)$	Aresta-conectividade média de v.
$\kappa_G(X)$	Vértice-conectividade de $X \subseteq V$ em G.
$\kappa_i(v)$	i-vértice-conectividade do vértice v.
$\lambda_G^+(v,\ X)$	Aresta-conectividade de saída de v em $X \subseteq V$.
$\lambda_G^-(v,\ X)$	Aresta-conectividade de entrada de v em $X \subseteq V$.
$\lambda_G^+(v)$	i-aresta-conectividade de saída de v em G.
$\lambda_G^-(v)$	i-aresta-conectividade de entrada de v em G.
$\mathscr{T}_G$	Árvore de componentes de máxima conectividade de G.

Resumo

Uma árvore de cortes é uma representação compacta da aresta-conectividade de um grafo não orientado. As árvores de cortes resolvem de maneira eficiente o problema de calcular a aresta-conectividade entre todos os pares de vértices do grafo. As árvores de cortes têm muitas aplicações como, por exemplo, no projeto de redes confiáveis, na partição de grafos, no agrupamento em grafos, na análise de redes sociais, dentre outras. Dois algoritmos para a construção de árvores de cortes de grafos não orientados e capacitados são bem conhecidos: o algoritmo de Gomory-Hu e o algoritmo de Gusfield. Este trabalho apresenta propostas de implementações paralelas de três algoritmos para encontrar uma árvore de cortes. Versões paralelas para os algoritmos de Gusfield e de Gomory-Hu são descritas e avaliadas experimentalmente. Um algoritmo híbrido que combina esses dois algoritmos e que busca tirar proveito das vantagens de cada um deles também é apresentado. Resultados experimentais mostram que os três algoritmos apresentam boas acelerações nos tempos de execução. Os experimentos também mostram que o algoritmo híbrido é quase sempre mais rápido do que o algoritmo de Gomory-Hu e em certas instâncias ele é muito mais rápido do que o algoritmo de Gusfield. Heurísticas para a melhoria do algoritmo de Gomory-Hu e do algoritmo híbrido são propostas e analisadas.

Na segunda parte desta tese, são estudadas medidas de centralidade dos vértices de um grafo que são baseadas na conectividade – algumas delas podem ser calculadas a partir de árvores de cortes. As medidas de centralidade de vértices têm como objetivo quantificar a importância dos vértices de um grafo com base em diferentes critérios. Dentre as medidas de centralidade propostas, destaca-se a *i*-aresta-conectividade, que mede a aresta-conectividade dos vértices em relação ao grafo. Uma medida de conectividade baseada em cortes de vértices também é proposta. Um estudo experimental com as medidas de conectividade foi executado para avaliar a relação das medidas propostas com outras medidas de centralidade mais conhecidas. Esse estudo mostra empiricamente que vértices com alta conectividade tendem a ter baixa excentricidade. Além disso, experimentos mostram que as medidas de conectividade não são equivalentes ao grau como critério de ordenação dos vértices.

Palavras-chaves: conectividade em grafos, árvores de cortes, algoritmos paralelos.

Abstract

PARALLEL CUT TREE ALGORITHMS AND
CENTRALITY MEASURES IN GRAPHS

A cut tree is a combinatorial structure that represents the edge-connectivity between all pairs of nodes of an undirected graph. Cut trees solve the all pairs minimum s-t-cut problem efficiently. Cut trees have a large number of applications including the solution of important combinatorial problems in fields such as routing, graph partitioning, graph clustering and graph connectivity. Cut trees have also been applied in scheduling problems, social network analysis, biological data analysis, among others. Two algorithms to compute a cut tree of a capacitated undirected graph are well known: Gomory-Hu algorithm and Gusfield algorithm. A main contribution of this work is the proposal, implementation and evaluation of three parallel cut tree algorithms. Parallel versions of Gusfield and Gomory-Hu algorithms are presented as well as a hybrid algorithm that combines those two. Experimental results show that the three algorithms achieve significant speedups on real and synthetic graphs. The hybrid algorithm outperformed the Gomory-Hu algorithm on most instances and it was faster than Gusfield algorithm on a few graphs. Heuristics to improve the performance of both Gomory-Hu algorithm and the hybrid algorithm are described and evaluated.

The second part of this work is dedicated to centrality measures in graphs. Centrality measures quantify various intuitive notions of the importance of nodes to networks. Families of centrality measures based on connectivity concepts are presented – some of them can be computed using cut trees. The concepts of edge-cuts and vertex-cuts induce different families of connectivity measures. The proposed i-edge-connectivity index quantifies the edge-connectivity of a vertex with respect to the graph. The locality of the measure can be configured by changing the value of i: smaller values of i produce measures that are more local, while large values produce global measures. An experimental evaluation of the proposed centrality measures shows the relationship between the connectivity measures with other well known centrality measures. It is shown that vertices with high connectivity are likely to have low eccentricities. Furthermore the results imply that the connectivity measures are not equivalent to the degree as a vertex sorting criteria.

Keywords: graph connectivity, cut trees, parallel algorithms.

1 Introdução

Uma árvore de cortes é uma estrutura combinatória que representa, de maneira compacta, a aresta-conectividade de todos os pares de vértices de um grafo não orientado. As árvores de cortes são amplamente utilizadas na solução de importantes problemas computacionais nas áreas de roteamento [76], medidas de confiabilidade de redes [28, 74], partição de grafos [94, 38], agrupamento em grafos [40, 33, 47], conectividade [84, 70], escalonamento de tarefas [90], dentre outros. As árvores de cortes também têm sido aplicadas em problemas de diversas outras áreas, tais como na análise de redes sociais [6, 65] e militares [106] e na análise de dados biológicos [77, 101], dentre outras [20, 61, 92].

Uma árvore de cortes de um grafo capacitado e não orientado pode ser encontrada utilizando-se um dentre dois algoritmos bem conhecidos: o algoritmo de R. E. Gomory e T. C. Hu [46] e o algoritmo de D. Gusfield [50]. Ambos os algoritmos fazem $n-1$ chamadas a um procedimento que encontra um corte de arestas de capacidade mínima entre dois vértices. Esses algoritmos possuem a mesma complexidade de tempo de pior caso. Na prática, os tempos de execução de nenhum deles domina os tempos de execução do outro. Os algoritmos de Gomory-Hu e de Gusfield para árvores de cortes são similares, porém, enquanto o algoritmo de Gusfield calcula todos os cortes mínimos no grafo de entrada, o algoritmo de Gomory-Hu encontra os cortes mínimos em grafos formados pela contração de vértices do grafo de entrada.

Apesar de parecidos, os algoritmos de Gomory-Hu e de Gusfield têm comportamentos bastante distintos em diferentes instâncias do problema. Em particular, o algoritmo de Gomory-Hu pode ser muito mais rápido em grafos que possuem cortes mínimos balanceados, porque os tamanhos dos grafos intermediários podem ser reduzidos. Por outro lado, o algoritmo de Gusfield é o mais rápido em muitas instâncias, pois o algoritmo de Gomory-Hu gasta tempo na construção dos grafos contraídos que pode não ser compensado pelo ganho de tempo no procedimento que encontra cortes mínimos. Essa relação entre os algoritmos foi estudada previamente em [45].

A importância das árvores de cortes pode ser constatada pela extensa lista de aplica-

ções. Apesar disso, poucos trabalhos experimentais sobre o desempenho dos algoritmos existentes foram publicados. Em particular, foram pouco estudadas heurísticas que tornem esses algoritmos mais eficientes ou descritas implementações paralelas de algoritmos para construção de árvores de cortes. Com o intuito de preencher essa lacuna, esta tese propõe versões paralelas dos algoritmos de Gusfield e de Gomory-Hu. Além disso, também é proposta uma versão híbrida que combina esses dois algoritmos e que busca tirar proveito das vantagens de cada um. O objetivo desse algoritmo híbrido é alcançar um bom desempenho que não dependa tanto das características de cada instância. Resultados experimentais mostram que os três algoritmos paralelos produzem bons *speedups*[1] nos tempos de execução.

Uma análise superficial dos algoritmos de Gusfield e de Gomory-Hu pode levar à conclusão de que os seus laços principais não podem ser paralelizados com eficiência devido à dependência entre as iterações. Entretanto, os resultados apresentados adiante mostram que esses algoritmos com pequenas modificações podem ser paralelizados, de forma a produzir *speedups* significativos.

A estratégia usada para paralelizar os algoritmos de Gusfield e de Gomory-Hu foi a de executar de maneira otimista as $n-1$ iterações em paralelo, isto é, cortes mínimos são calculados assincronamente por diferentes processos. À medida que os cortes mínimos são encontrados, o algoritmo faz tentativas de modificar a árvore de cortes em construção. Essa tentativa será bem sucedida somente se os vértices separados pelo corte mínimo ainda não tiverem sido separados por outro corte previamente encontrado (os detalhes dessa condição são diferentes em cada um dos algoritmos). Caso outra execução paralela já tenha separado os dois vértices em questão, o corte mínimo encontrado é rejeitado e um novo corte mínimo deve ser calculado em seu lugar.

O desempenho das implementações paralelas dos algoritmos de Gusfield e de Gomory-Hu depende da estrutura do grafo. Por isso, experimentos foram executados em diversas classes de grafos sintéticos bem como algumas redes complexas reais com diferentes propriedades. Os experimentos mostram que os tempos de execução de nenhuma das versões paralelas desses algoritmos dominam os tempos das demais.

Tendo em vista a variabilidade do desempenho dos algoritmos de Gusfield e de Gomory-Hu nas diferentes instâncias do problema, uma versão híbrida desses algoritmos foi proposta. Esse novo algoritmo busca por cortes mínimos ora no grafo de entrada, ora em um grafo contraído, dependendo do tamanho desse último. A principal vantagem do algoritmo híbrido é evitar a construção do grafo contraído, uma operação computacionalmente custosa, quando isso

[1] O *speedup* é a razão entre o tempo de execução sequencial e o tempo de execução paralelo.

claramente não gera um ganho de tempo. Resultados experimentais com o algoritmo híbrido mostram que ele é quase sempre mais rápido do que o algoritmo de Gomory-Hu e em certas instâncias ele é muito mais rápido do que o algoritmo de Gusfield.

As implementações iniciais do algoritmo de Gomory-Hu e do algoritmo híbrido foram otimizadas com diversas heurísticas. A mais importante delas consiste na busca por cortes balanceados a partir da enumeração de todos os cortes mínimos que separam um par de vértices. Essa heurística foi implementada como um algoritmo recursivo de enumeração que possui uma complexidade de tempo linear por corte produzido. Uma heurística para a escolha dos pares de vértices a serem separados também é proposta. Outras heurísticas são descritas na tese.

Na segunda parte desta tese, são estudadas medidas de centralidade dos vértices de um grafo que são baseadas na conectividade – algumas delas podem ser calculadas a partir de árvores de cortes. Medidas de centralidade em grafos procuram mensurar a importância de seus vértices de acordo com diferentes critérios. Exemplos de medidas de centralidade em grafos previamente estudadas são o grau dos vértices, a excentricidade, o grau de proximidade (*closeness*) e o grau de intermediação (*betweenness*) [107, 34, 85, 16].

Na pesquisa em redes complexas[2], medidas de centralidade têm sido aplicadas no estudo de processos que ocorrem na rede, como, por exemplo, a propagação de epidemias e de suas formas de controle ou da vulnerabilidade das redes quando expostas a ataques [31, 82]. Na Ciência da Computação, conhecimento sobre medidas de centralidade podem contribuir para a melhoria da robustez de aplicações computacionais, a melhoria de protocolos de roteamento, a aceleração de mecanismos de buscas ou a melhoria de métodos de alocação de recursos [52, 102, 104, 4].

Dentre as medidas de centralidade propostas nesta tese, destaca-se a i-aresta-conectividade, que mede a aresta-conectividade dos vértices em relação ao grafo. Uma medida de conectividade baseada em cortes de vértices também é proposta. Medidas de centralidade podem ser mais locais ou mais globais. As medidas de centralidade propostas são parametrizadas de forma a produzir medidas com diferentes níveis de localidade. Algoritmos polinomiais para o cálculo das medidas de conectividade são apresentados.

Um estudo experimental com as medidas de conectividade foi executado para avaliar a relação das medidas propostas com outras medidas de centralidade mais conhecidas. Esse estudo mostra empiricamente que vértices com alta conectividade tendem a ter baixa excentrici-

[2] *Redes complexas* é o nome dado aos grafos cujo estudo é motivado por redes reais como, por exemplo, redes sociais, biológicas e tecnológicas. Resenhas sobre o assunto podem ser encontradas, por exemplo, em [81, 82, 35, 36].

dade. Além disso, experimentos mostram que as medidas de conectividade não são equivalentes ao grau como critério de ordenação dos vértices.

A próxima seção apresenta definições preliminares sobre grafos e conectividade que serão utilizadas ao longo do texto.

1.1 Definições Preliminares: Grafos, Cortes, Fluxos e Conectividade

Esta seção tem como objetivo apresentar definições importantes que serão utilizadas ao longo do trabalho. São definidos grafos, grafos orientados e diversos conceitos relacionados à conectividade de grafos. A notação utilizada é parcialmente baseada em [79].

Dado um conjunto C, o conjunto dos subconjuntos de C de tamanho 2 é denotado por $\binom{C}{2} = \{C' \subseteq C \mid |C'| = 2\}$. O símbolo $\mathbb{Q}$ denota o conjunto do números racionais. O símbolo $\mathbb{Q}_{\geq 0}$ denota o conjunto dos números racionais não negativos.

Grafos Um *multigrafo* ou *grafo* $G = (V_G, E_G)$ é um par em que V_G é um conjunto finito e E_G é um multiconjunto cujos elementos pertencem a $\binom{V_G}{2}$. Os elementos de V_G são chamados de *vértices*. Os elementos de E_G são chamados de *arestas*. Dada uma aresta $e = \{u, v\} \in E_G$, dizemos que e é *incidente* em u e v.

Um *grafo capacitado* $G = (V_G, E_G, c_G)$ é uma tripla tal que (V_G, E_G) é um grafo e $c_G : E_G \to \mathbb{Q}_{\geq 0}$ é uma função que define as *capacidades* das arestas em E_G.

Um *grafo orientado* $G = (V_G, E_G)$ é um par em que V_G é um conjunto finito e E_G é um multiconjunto cujos elementos pertencem a $V \times V$. Dada uma aresta $e = (u, v) \in E_G$ dizemos que u é a *origem* de e e v é o *destino* de e.

Seja um grafo capacitado $G = (V_G, E_G, c_G)$ e $X \subseteq V_G$. Denota-se por $G - X$ o grafo obtido de G após a remoção dos vértices de X e das arestas incidentes a algum vértice de X. O grafo G/X é o grafo obtido de G após a contração dos vértices de X em um único vértice, seguida da remoção dos laços formados e da união das arestas repetidas em uma única aresta que possui a soma das capacidades das arestas originais.

Uma sequência de vértices distintos e arestas $P = (v_1, e_1, v_2, e_2, \ldots e_{k-1}, v_k)$ é chamada de *caminho* entre v_1 e v_k (ou de v_1 para v_k em grafos orientados) se $v_1, v_2, \ldots v_k \in V_G$, $e_1, e_2, \ldots e_{k-1} \in E_G$ e $e_i = \{v_i, v_{i+1}\}$ (ou $e_i = (v_i, v_{i+1})$ se G é um grafo orientado), para $i = 1, 2, \ldots, k-1$. Um caminho entre u e v (ou de u para v em grafos orientados) é chamado de

um u-v-caminho. O *comprimento* de um caminho é o seu número de arestas. A *distância* entre dois vértices u e v, denotada por $d(u,v)$, é o comprimento de um u-v-caminho de comprimento mínimo.

Um grafo G é dito *conexo* se para todo $u,v \in V$ existe em G um u-v-caminho. Um conjunto de vértices $X \subseteq V_G$ é um *componente conexo* de G se para todo $u,v \in X$, existe um u-v-caminho em G e X é maximal [79].

Funções de Corte Seja $G = (V_G, E_G, c_G)$ um grafo capacitado. Sejam dois conjuntos $X,Y \subseteq V_G$. Definimos $E_G(X,Y) = \{\{u,v\} \in E_G \mid u \in X \text{ e } v \in Y\}$ e $d_G(X,Y) = \sum_{e \in E_G(X,Y)} c_G(e)$. Quando G é um grafo orientado, então $E_G(X,Y) = \{(u,v) \in E_G \mid u \in X \text{ e } v \in Y\}$. Quando $Y = V_G \backslash X$, então podemos denotar $E_G(X,Y)$ e $d_G(X,Y)$ por $E_G(X)$ e $d_G(X)$, respectivamente. O *grau ponderado* de um vértice é $d_G(v) = d_G(\{v\})$. Os grafos não capacitados serão tratados como grafos capacitados com $c_G(e) = 1$, para toda aresta $e \in E_G$. Dessa maneira, $d_G(v)$ corresponde ao número de arestas incidentes em v, que é o *grau* de v.

Cortes Um *corte* de um grafo $G = (V_G, E_G, c_G)$ é uma bipartição de V_G, ou seja, um par de conjuntos $\{X,Y\}$ tal que $X,Y \neq \emptyset$, $X \cap Y = \emptyset$ e $X \cup Y = V_G$. O corte *induzido* por um conjunto $X \subseteq V_G$ é a bipartição $\{X, \overline{X}\}$ de V_G induzida por X, onde $\overline{X} = V_G \backslash X$. Alternativamente, denotamos o corte $\{X, \overline{X}\}$ por X. O conjunto $E_G(X)$ contém as arestas que *cruzam* o corte $\{X, \overline{X}\}$. A *capacidade* de um corte X é $d_G(X)$.

Sejam $s,t \in V_G$. Um *s-t-corte* de G é um corte $\{X, \overline{X}\}$ tal que $s \in X$ e $t \in \overline{X}$. Um *s-t-corte mínimo* é um s-t-corte de capacidade mínima. A *aresta-conectividade local* entre s e t em G, denotada por $\lambda_G(s,t)$, é a capacidade de um s-t-corte mínimo de G.

Dado $G = (V_G, E_G, c_G)$, um conjunto $C \subseteq V_G$ é um *corte de vértices* de G se o grafo $G - C$ tiver mais componentes conexos do que G. Um corte de vértices C de um grafo conexo *separa* dois vértices s e t se s e t estiverem em componentes conexos distintos de $G - C$. Chamamos esse corte de um *s-t-corte de vértices* [79].

Dados dois vértices $s,t \in V_G$, um conjunto de s-t-caminhos é *internamente vértice-disjunto* se cada par de caminhos do conjunto tem em sua intersecção apenas os vértices s e t. A *vértice-conectividade local* entre s e t em G, denotada por $\kappa_G(s,t)$, é o número máximo de caminhos internamente vértice-disjuntos entre s e t. Se $\{s,t\} \notin E_G$, então $\kappa_G(s,t)$ é igual à mínima capacidade de um s-t-corte de vértices.

Em um grafo orientado $G = (V_G, E_G, c_G)$, um *corte* é um par $(X, \overline{X})$ onde $\{X, \overline{X}\}$ é

uma bipartição de V_G. O conjunto $E_G(X) = E_G(X,\overline{X}) = \{(u,v) \in E_G \mid u \in X \text{ e } v \in \overline{X}\}$ contém as arestas que *cruzam* o corte $(X,\overline{X})$. A *capacidade* do corte $(X,\overline{X})$ é $d_G(X) = \sum_{e \in E_G(X)} c_G(e)$. Um *corte orientado* de G é um corte $(X,\overline{X})$ tal que $E_G(\overline{X},X) = \emptyset$. Um corte $(X,\overline{X})$ é um s-t-corte de G se $s \in X$ e $t \in \overline{X}$.

O conceito de fluxo máximo entre dois vértices é definido abaixo e a sua relação com a conectividade local entre esses vértices é apresentada. Essa relação tem grande importância prática nesta tese, pois os algoritmos mais eficientes para determinar a conectividade local entre dois vértices são os algoritmos de fluxo máximo.

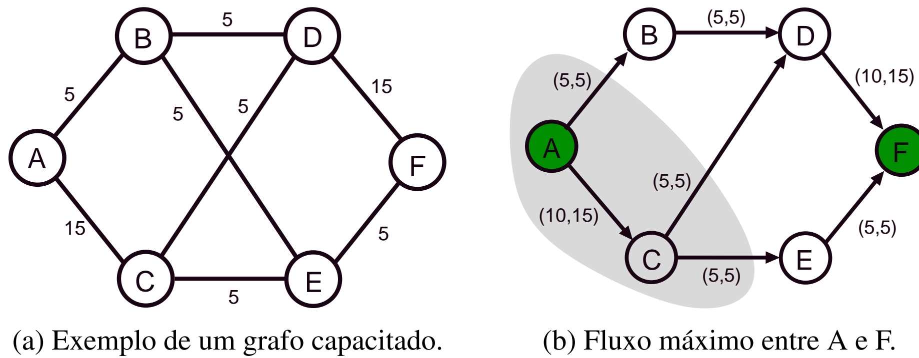
Fluxo em Redes Uma *rede de fluxos* é uma tripla $N = (G,s,t)$ onde $G = (V_G, E_G, c_G)$ é um grafo orientado e capacitado que contém dois vértices especiais s e t chamados de *origem* e *destino* da rede, respectivamente. A capacidade de uma aresta da rede de fluxos é interpretada informalmente como o máximo de fluxo que pode atravessar a aresta. Para facilitar a notação, consideramos que a capacidade da rede de fluxos satisfaz: $c_G(u,v) = 0$ sempre que $(u,v) \notin E_G$.

Dada uma rede de fluxos $N = (G,s,t)$, um *fluxo* em N é uma função $f : V_G \times V_G \to \mathbb{Q}$ que satisfaz:

1. $f(u,v) \leq c_G(u,v), \quad \forall u,v \in V_G$
2. $f(u,v) = -f(v,u), \quad \forall u,v \in V_G$
3. $\sum_{u \in V_G} f(u,v) = 0, \quad \forall v \in V_G \backslash \{s,t\}$.

O *valor do fluxo* f é igual a $\sum_{v \in V_G \backslash \{s\}} f(s,v)$. Um *fluxo máximo* em uma rede de fluxos é um fluxo de valor máximo. De acordo com o Teorema do Corte Mínimo-Fluxo Máximo [67, 30], o valor de um fluxo máximo entre dois vértices s e t de uma rede de fluxos é igual à capacidade de um s-t-corte mínimo. Portanto, a conectividade local entre dois vértices s e t pode ser encontrada utilizando-se qualquer algoritmo para o fluxo máximo entre dois vértices. Na verdade, não são conhecidos algoritmos mais eficientes para os cálculo da conectividade local entre dois vértices do que os algoritmos de fluxo máximo [79]. Esse resultado também se aplica a grafos não orientados, uma vez que a substituição de cada aresta $\{u,v\}$ de um grafo não orientado por duas arestas orientadas (u,v) e (v,u) cada uma com a mesma capacidade, o transforma em um grafo orientado sem alterar as capacidades dos cortes.

Considere o exemplo da Figura 1.1a em que as arestas são rotuladas com as suas capacidades. A Figura 1.1b mostra um fluxo máximo entre os vértices A e F. As arestas estão rotuladas por um par (f,c) onde f é o fluxo associado à aresta e c é a capacidade da aresta. O



(a) Exemplo de um grafo capacitado. (b) Fluxo máximo entre A e F.

Figura 1.1: Um grafo não direcionado e um fluxo máximo entre dois vértices desse grafo.

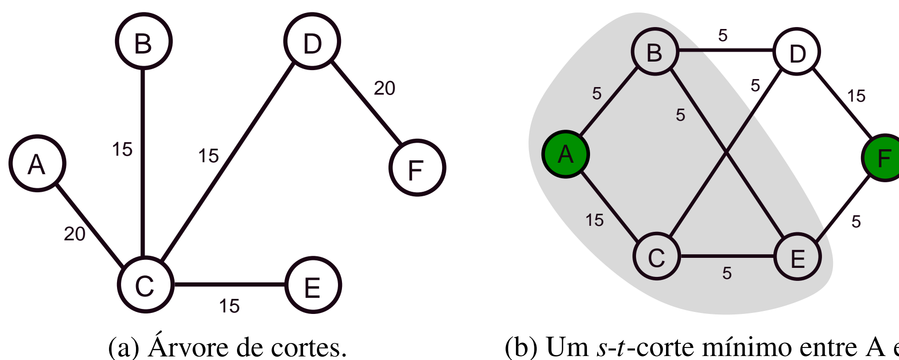
valor de um fluxo máximo entre A e F é 15. Arestas com fluxo igual a 0 foram omitidas. Um A-F-corte mínimo é mostrado na figura.

Os algoritmos para o cálculo do fluxo máximo entre dois vértices se dividem em duas categorias principais: os algoritmos baseados em *caminhos aumentantes* e aqueles baseados nas operações *push* e *relabel*. Exemplos de algoritmos do primeiro tipo são o algoritmo de L. R. Ford e D. R. Fulkerson [41], o algoritmo de J. Edmonds e R. Karp [37] e o algoritmo de Y. Dinitz [32]. Os algoritmos do segundo tipo são aqueles baseados no algoritmo *push-relabel* de A. V. Goldberg e R. E. Tarjan e variações [43, 24]. Nessa categoria de algoritmos, o algoritmo de A. V. Goldberg e R. E. Tarjan tem complexidade de $O(mn\log\frac{n^2}{m})$ e o algoritmo de V. King et al. [66] tem complexidade de $O(mn\log_{\frac{m}{n\log n}} n)$, onde $n = |V_G|$ e $m = |E_G|$. Os algoritmos da família *push-relabel* são mais rápidos assintoticamente no pior caso do que os algoritmos baseados em caminhos aumentantes e são rápidos na prática. Os algoritmos baseados em caminhos aumentantes são mais rápidos somente quando a conectividade local entre s e t é muito baixa.

Recentemente, D. Hochbaum [59] apresentou um algoritmo baseado no conceito de *pseudofluxos* que resolve o problema em $O(mn\ \log n)$. Experimentalmente, esse algoritmo mostrou ser competitivo e, muitas vezes, melhor do que o algoritmo *push-relabel* [21]. Mais recentemente, foi apresentado por J. B. Orlin um algoritmo de complexidade $O(mn)$ para o problema do fluxo máximo[3], porém nenhum resultado experimental foi apresentado.

Árvores de Cortes e Árvores de Fluxo Equivalente Considere o problema de calcular a aresta-conectividade local entre todos os pares de vértices de um grafo não orientado. A solução ingênua consiste em executar $\binom{|V_G|}{2}$ algoritmos de fluxo máximo, um para cada par de vértices. Em 1961, R. E. Gomory e T. C. Hu [46] mostraram que o cômputo de apenas $|V_G|-1$ fluxos máximos são suficientes. A solução descoberta consiste na construção de uma árvore capacitada

[3]Resultado ainda não aceito para publicação em março de 2013.



(a) Árvore de cortes.

(b) Um s-t-corte mínimo entre A e F.

Figura 1.2: Uma árvore de cortes e um s-t-corte mínimo do grafo da Figura 1.1.

sobre o conjunto de vértices do grafo que representa os valores das conectividades locais entre todos os pares de vértices. Essa árvore é chamada de *árvore de cortes* e é definida a seguir.

Uma *árvore de fluxo equivalente* de um grafo $G = (V_G, E_G)$ é uma árvore capacitada T sobre o conjunto de vértices V_G tal que para todos os pares de vértices $u, v \in V_G$, a menor capacidade de uma aresta no caminho entre u e v em T é igual à aresta-conectividade local entre u e v em G, isto é, $\lambda_G(u,v) = \lambda_T(u,v)$, para todos $u, v \in V_G$. A árvore T não precisa ser um subgrafo de G.

Uma *árvore de cortes* de um grafo G é uma árvore de fluxo equivalente T tal que, para todos $u, v \in V_G$, o corte induzido pela remoção da aresta de capacidade mínima do caminho entre u e v em T é um u-v-corte mínimo de G. As árvores de cortes também são conhecidas como *árvores de Gomory-Hu* e *árvores de cortes mínimos* [79].

A Figura 1.2a mostra uma árvore de cortes do grafo da Figura 1.1a. A Figura 1.2b mostra um corte mínimo entre os vértices A e F induzido pela remoção da aresta $\{C, D\}$ da árvore de cortes.

1.2 Organização deste Trabalho

O restante deste trabalho está organizado da seguinte maneira. O Capítulo 2 descreve os algoritmos sequenciais conhecidos para a construção de árvores de fluxo equivalente e de árvores de cortes. As Seções 2.1 e 2.2 apresentam descrições dos algoritmos sequenciais de Gusfield e de Gomory-Hu, respectivamente. A Seção 2.3 traz uma análise conceitual sobre a paralelização dos algoritmos para a construção de árvores de cortes.

O Capítulo 3 é dedicado à versão paralela do algoritmo de Gusfield e da avaliação experimental desse algoritmo. O Capítulo 4 apresenta a versão paralela do algoritmo de Gomory-Hu, o algoritmo híbrido, heurísticas para a melhoria do desempenho desses algoritmos e avaliações

experimentais.

O Capítulo 5 é dedicado à apresentação de medidas de centralidade de grafos baseadas em conectividade. Na Seção 5.2 são definidas as medidas de centralidade baseadas na aresta-conectividade e as medidas baseadas na vértice-conectividade. Também são apresentadas generalizações das medidas de centralidade para grafos orientados.

O Capítulo 6 apresenta algoritmos para o cálculo das medidas de centralidade baseadas em conectividade apresentadas no capítulo anterior. A Seção 6.1 apresenta algoritmos para o cálculo das medidas baseadas em cortes de arestas. A Seção 6.2 apresenta um algoritmo para o cálculo da medida baseada em cortes de vértices. O Capítulo 7 apresenta um estudo comparativo das medidas de conectividade em grafos reais e sintéticos. O Capítulo 8 traz as conclusões deste trabalho.

2 Algoritmos para Árvores de Cortes

Árvores de cortes são uma importante estrutura combinatória que representa a aresta-conectividade entre todos os pares de vértices de um multigrafo capacitado. As árvores de cortes têm inúmeras aplicações diretas (por exemplo [90, 20, 6, 61, 92, 77, 101]) e também são utilizadas na solução de inúmeros outros problemas combinatórios em áreas como particionamento de grafos, conectividade e roteamento (por exemplo [106, 28, 74, 94, 40, 76, 38, 84, 70, 47, 65]).

O conceito de árvores de cortes e um algoritmo para a sua construção foram descobertos por R. E. Gomory e T. C. Hu in 1961 [46]. Outro algoritmo para construção de árvores de cortes foi proposto por D. Gusfield em 1990 [50]. Ambos os algoritmos requerem o cômputo de $|V|-1$ s-t-cortes mínimos ou, equivalentemente, fluxos máximos. O algoritmo de Gomory-Hu é de divisão e conquista e faz contrações de vértices do grafo de entrada. O algoritmo de Gusfield calcula todos os cortes mínimos no grafo de entrada. Chamaremos esses algoritmos de GH e Gus, respectivamente.

Neste capítulo, a Seção 2.1 trata do algoritmo sequencial de Gusfield e a Seção 2.2 apresenta o algoritmo sequencial de Gomory e Hu. A Seção 2.3 apresenta uma discussão sobre a paralelização dos algoritmos para árvores de cortes, que serve de motivação para os Capítulos 3 e 4 que apresentam as principais contribuições desta tese. A Seção 2.4 apresenta uma revisão da literatura sobre árvores de cortes.

2.1 O Algoritmo de Gusfield

Dado um multigrafo não orientado e capacitado $G=(V,E_G,c_G)$, $c_G\colon E_G \to \mathbb{Q}_{\geq 0}$, uma *árvore de fluxo equivalente* de G é uma árvore $T=(V,E_T,c_T)$, $c_T\colon E_T \to \mathbb{Q}_{\geq 0}$, tal que para todos os pares de vértices $s,t \in V$, a aresta-conectividade entre s e t é a mesma em T e em G, isto é, $\lambda_G(s,t)=\lambda_T(s,t)$, onde $\lambda_G(s,t)$ e $\lambda_T(s,t)$ são as capacidades dos s-t-cortes mínimos em G e em T, respectivamente.

Uma *árvore de cortes* de G é uma árvore de fluxo equivalente de G, $T=(V,E_T,c_T)$,

tal que para todos os pares de vértices $s,t \in V$, o corte induzido pela remoção de uma aresta de capacidade mínima do caminho entre s e t em T é um s-t-corte mínimo em G.

O algoritmo de Gusfield para encontrar uma árvore de fluxo equivalente de um multigrafo capacitado consiste de $|V|-1$ iterações. Cada uma delas calcula um s-t-corte mínimo. O pseudocódigo pode ser visto no Algoritmo 2.1. O algoritmo recebe como entrada um grafo $G=(V,E_G,c_G)$ onde $c_G : E_G \to \mathbb{Q}_{\geq 0}$ é uma função que define as capacidades das arestas. A saída do algoritmo é uma árvore capacitada $T=(V,E_T,c_T)$ com capacidades definidas pela função $c_T : E_T \to \mathbb{Q}_{\geq 0}$. Os vértices de V são identificados pelos números $\{1,2,\ldots,|V|\}$. A árvore em construção é representada por um vetor $tree[.]$, tal que $tree[i]$, $2 \leq i \leq |V|$, é o nodo vizinho de i mais próximo da raiz. A raiz da árvore é sempre o nodo 1 e $tree[1]=1$.

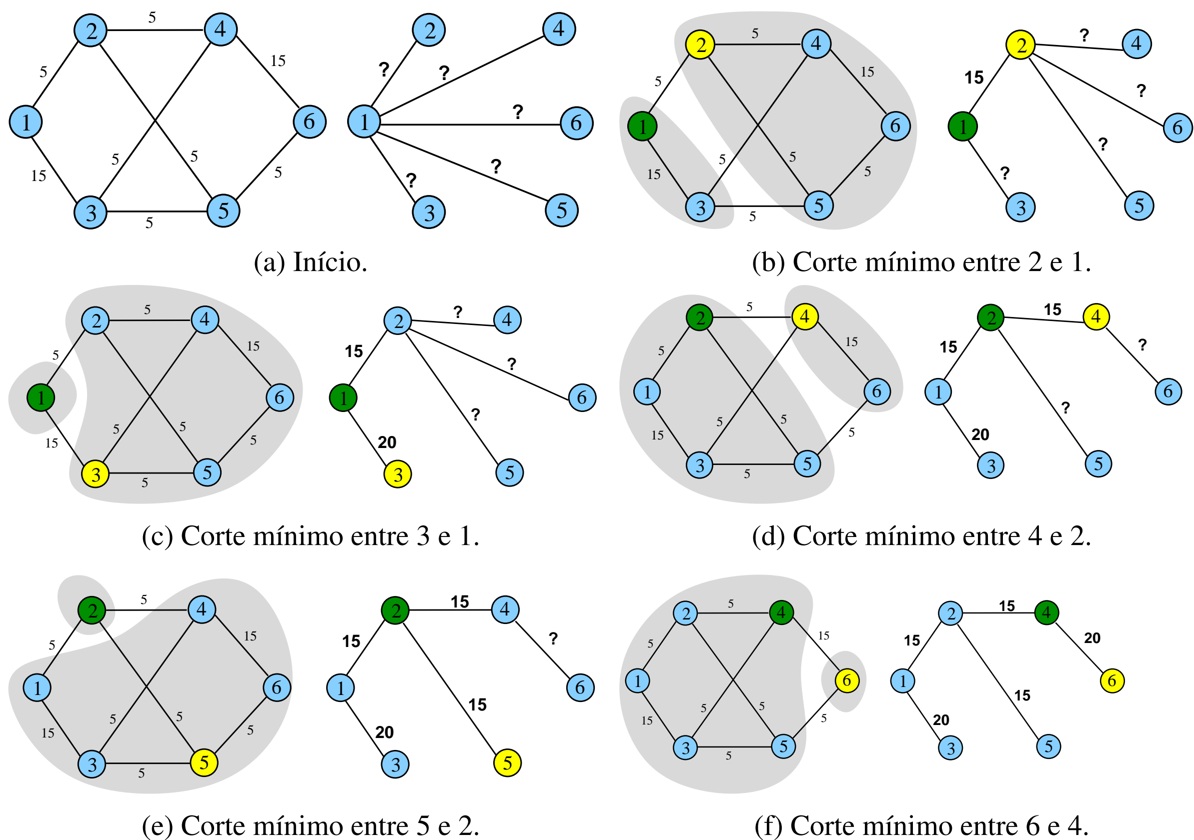
Algoritmo 2.1 Algoritmo de Gusfield para árvores de cortes.

Entrada: $G=(V,E_G,c_G)$ é um grafo não orientado capacitado
Saída: $T=(V,E_T,c_T)$ é uma árvore de fluxo equivalente de G
1: **para** $i=1$ to $|V|$ **faça**
2: $tree[i] \leftarrow 1$
// $|V|-1$ iterações que calculam s-t-cortes mínimos
3: **para** $s \leftarrow 2$ to $|V|$ **faça**
4: $t \leftarrow tree[s]$
5: $\lambda[s], X \leftarrow$ `corte_mínimo`(s,t,G)
6: **para** $u \in V,\ u > s$ **faça** // atualiza a árvore em construção
7: **se** $tree[u]=t$ e $u \in X$ **então**
8: $tree[u] \leftarrow s$
// Devolve uma árvore de fluxo equivalente
9: $E_T \leftarrow \emptyset$
10: **para** $s \leftarrow 2$ to $|V|$ **faça**
11: $E_T \leftarrow E_T \cup \{\{s,tree[s]\}\}$
12: $c_T(\{s,tree[s]\}) \leftarrow \lambda[s]$
13: **devolva** $T=(V,E_T,c_T)$

O algoritmo inicia com uma árvore com topologia de estrela em que o nodo 1 é o centro (linhas 1-2). A cada iteração (linhas 3-8), o algoritmo escolhe um nodo diferente, s, $s \geq 2$, como *origem*. Essa escolha determina o nodo de *destino*, t, como o vizinho atual de s na árvore, pela atribuição $t \leftarrow tree[s]$ da linha 4. Um s-t-corte mínimo é computado pela chamada do procedimento `corte_mínimo(s,t,G)` que devolve a capacidade de um s-t-corte mínimo e também um conjunto X tal que $\{X,\overline{X}\}$ é um s-t-corte mínimo (linha 5). A árvore em construção é modificada: cada nodo u vizinho de t com $u > s$ que esteja no lado de s do s-t-corte é desconectado de t e reconectado a s. O algoritmo termina quando todos os nodos de 2 até $|V|$ tiverem sido origem de um s-t-corte mínimo de alguma iteração. Uma prova da correção desse algoritmo pode ser encontrada no artigo original de D. Gusfield [50].

A implementação do algoritmo de Gusfield é simples, pois pode utilizar qualquer algoritmo para o problema do s-t-corte mínimo (ou do fluxo máximo). A versão descrita no Algoritmo 2.1 calcula uma árvore de fluxo equivalente e uma pequena modificação nesse algoritmo o faz encontrar uma árvore de cortes: na linha 6, permita que qualquer vizinho de t que esteja em X seja reconectado a s e não apenas os nodos maiores do que s.

Um exemplo de execução do algoritmo de Gusfield pode ser visto na Figura 2.1. Cada figura mostra, à esquerda, o grafo e um corte mínimo e, à direita, a árvore em construção. No primeiro passo, o algoritmo encontra um corte mínimo que separa os vértices $s = 2$ e $t = tree[2] = 1$. Os vértices $4, 5$ e 6 ficaram do lado do vértice 2 no corte mínimo e, por isso, os nodos $4, 5$ e 6 da árvore são reconectados ao nodo 2, como pode ser visto na Figura 2.1b. A aresta $\{1, 2\}$ da árvore recebe a capacidade 15 que é a mesma capacidade do s-t-corte utilizado. Na iteração seguinte, o algoritmo procura um corte mínimo que separa os vértices $s = 3$ e $t = tree[3] = 1$. A árvore não é modificada porque o nodo 1 não possui nenhum vizinho com um identificador maior do que 3. O algoritmo continua de maneira análoga até que a árvore de fluxo equivalente esteja completa.



(a) Início. (b) Corte mínimo entre 2 e 1.

(c) Corte mínimo entre 3 e 1. (d) Corte mínimo entre 4 e 2.

(e) Corte mínimo entre 5 e 2. (f) Corte mínimo entre 6 e 4.

Figura 2.1: Exemplo de execução do algoritmo de Gusfield para árvore de fluxo equivalente.

2.2 Algoritmo de Gomory-Hu

Assim como o algoritmo de Gusfield, o algoritmo de Gomory-Hu [46] encontra uma árvore de cortes de um multigrafo capacitado através de $|V|-1$ iterações; em cada uma delas um s-t-corte mínimo é calculado. Ao contrário do algoritmo de Gusfield, o algoritmo de Gomory-Hu utiliza uma estratégia de divisão e conquista que modifica o grafo de entrada.

Seja um grafo não orientado e capacitado $G=(V_G,E_G,c_G)$, onde $c_G\colon E_G \to \mathbb{Q}_{\geq 0}$ define as capacidades das arestas. Seja P uma partição[1] de V_G. Seja $T=(P,E_T,c_T)$ uma árvore ponderada com capacidades definidas pela função $c_T\colon E_T \to \mathbb{Q}_{\geq 0}$ e cujos vértices são partes da partição P de V_G. Dizemos que T é uma *árvore de cortes em relação à partição* P se para cada aresta $e=\{V_u,V_v\}\in E_T$, existem vértices $u\in V_u$ e $v\in V_v$ tais que $\lambda_G(u,v)=c_T(e)$ e a partição dos vértices obtida pela remoção de e de T induz um u-v-corte mínimo de G. O corte induzido em G pelos componentes de $T-e$ é denotado por $C_T(e)$.

Uma árvore de cortes em relação à partição de V_G em que cada parte possui exatamente um vértice é uma árvore de cortes de G, como demonstrado no lema abaixo, adaptado de [13]:

Lema 2.1. *Seja um grafo $G=(V_G,E_G,c_G)$ e $T=(P,E_T,c_T)$ uma árvore de cortes em relação à partição $P=\{\{v\}\mid v\in V_G\}$ de V_G. Então T é uma árvore de cortes de G.*

Demonstração. Sejam dois vértices $u,v\in V_G$. Seja $e=\{u',v'\}$ a aresta de capacidade mínima no caminho p entre u e v em T. Queremos provar que $\lambda_G(u,v)=c(e)$.

Por um lado, $\lambda_G(u,v)\geq c(e)$, porque qualquer u-v-corte separa algum par de vértices consecutivos do caminho p e, portanto, tem capacidade maior do que ou igual a de alguma aresta do caminho p. Como $c(e)$ é a capacidade mínima de uma aresta do caminho, então $\lambda_G(u,v)\geq c(e)$.

Por outro lado, $\lambda_G(u,v)\leq c(e)$, pois o corte $C_T(e)$, induzido em G pela remoção da aresta e de T, é um u-v-corte em G. □

O algoritmo de Gomory-Hu é baseado no seguinte lema [46]:

Lema 2.2. *Seja $\{X,\overline{X}\}$ um x-y-corte mínimo com $x\in X$. Considere dois vértices $s,t\in X$. Seja $\{S,\overline{S}\}$ um s-t-corte mínimo tal que $y\notin S$ (troque S por $\overline{S}$ e s por t, se necessário). Então, $\{X\cap S,\overline{X\cap S}\}$ é um s-t-corte mínimo.*

[1] Uma *partição* de um conjunto finito V é conjunto de subconjuntos não vazios de V tal que cada elemento de V está em exatamente um desses subconjuntos. Os subconjuntos da partição são chamados de *partes* da partição.

A Figura 2.2 ilustra o Lema 2.2. À esquerda está o caso em que $x \in S$ e à direita o caso em que $x \in \overline{S}$. O Lema 2.2 mostra que dados dois vértices s e t quaisquer em X, sempre existe um s-t-corte mínimo que *não cruza* o corte $\{X, \overline{X}\}$. A prova do lema consiste em um argumento de contagem das arestas que compõem os cortes, agrupados nas 4 intersecções possíveis dos conjuntos S, $\overline{S}$, X e $\overline{X}$.

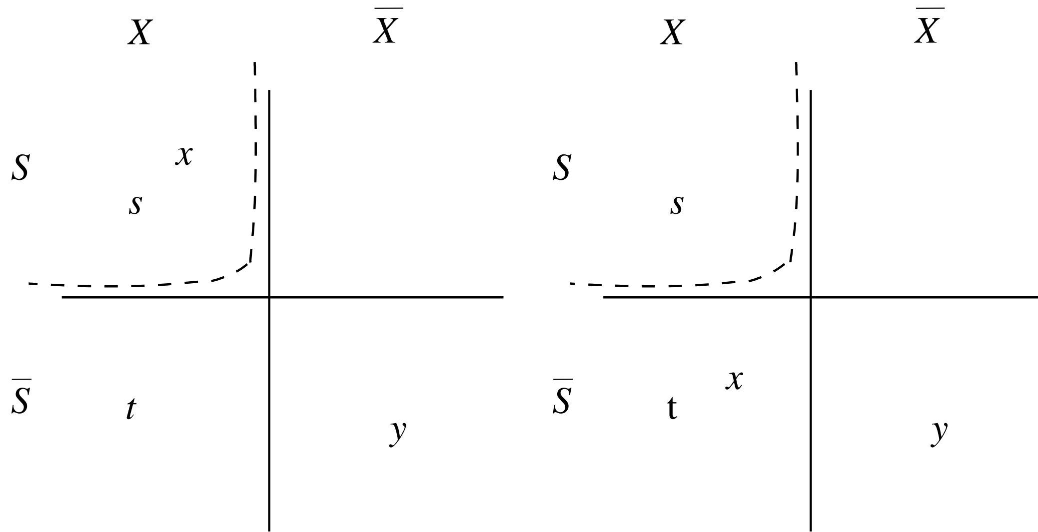


Figura 2.2: Ilustração do Lema 2.2. O corte $\{X, \overline{X}\}$ é um x-y-corte mínimo. Os vértices s e t pertencem a X e $\{S, \overline{S}\}$ é um s-t-corte mínimo. Se $y \in \overline{S}$, então $S \cap X$ define um s-t-corte mínimo que não cruza X.

Dados $X, Y \subseteq V$, vamos denotar por $d(X)$ a capacidade do corte $\{X, \overline{X}\}$ e por $d(X, Y)$ a soma das capacidades das arestas com uma ponta em X e a outra ponta em Y.

Demonstração do Lema 2.2. Temos que provar que $\{X \cap S, \overline{X \cap S}\}$ é um s-t-corte mínimo.

A propriedade de submodularidade dos cortes diz que $d(S \cap X) + d(\overline{S} \cap \overline{X}) \leq d(S) + d(X)$. Isso decorre do fato de que $d(S \cap X) + d(\overline{S} \cap \overline{X}) = d(S \cap X, S \cap \overline{X}) + d(S \cap X, \overline{S} \cap X) + d(S \cap X, \overline{S} \cap \overline{X}) + d(\overline{S} \cap \overline{X}, S \cap \overline{X}) + d(\overline{S} \cap \overline{X}, S \cap X) + d(\overline{S} \cap \overline{X}, \overline{S} \cap X) = d(S \cap X, S \cap \overline{X}) + d(S \cap X, \overline{S} \cap X) + d(\overline{S} \cap \overline{X}, S \cap \overline{X}) + d(\overline{S} \cap \overline{X}, \overline{S} \cap X) + 2 \cdot d(S \cap X, \overline{S} \cap \overline{X}) \leq d(S \cap X, S \cap \overline{X}) + d(S \cap X, \overline{S} \cap X) + d(\overline{S} \cap \overline{X}, S \cap \overline{X}) + d(\overline{S} \cap \overline{X}, \overline{S} \cap X) + 2 \cdot d(S \cap X, \overline{S} \cap \overline{X}) + 2 \cdot d(S \cap \overline{X}, \overline{S} \cap X) = d(S) + d(X)$.

Como $\{\overline{S} \cap \overline{X}, S \cup X\}$ é um x-y-corte, então $d(\overline{S} \cap \overline{X}) \geq d(X)$, pois $\{X, \overline{X}\}$ é um s-t-corte mínimo. As desigualdades $d(\overline{S} \cap \overline{X}) \geq d(X)$ e $d(S \cap X) + d(\overline{S} \cap \overline{X}) \leq d(S) + d(X)$ implicam que $d(S \cap X) \leq d(S)$. Logo, $\{S \cap X, \overline{S \cap X}\}$ deve ser também um s-t-corte mínimo. $\square$

Considere o grafo $G/\overline{X}$ obtido a partir de G através da contração dos vértices do conjunto $\overline{X}$. O Lema 2.2 mostra que $\lambda_G(s,t) = \lambda_{G/\overline{X}}(s,t)$, ou seja, para encontrar um s-t-corte mínimo de G, é possível considerar o grafo contraído $G/\overline{X}$.

Seja $X_1, X_2, \ldots, X_k$, subconjuntos de V_G, disjuntos dois a dois. O grafo obtido a partir de G, contraindo-se os conjuntos X_1 em um novo vértice x_1, o conjunto X_2 em um vértice x_2, e assim por diante, é denotado por $G/X_1, X_2, \ldots, X_k$. O grafo G após a remoção do vértice v e das arestas incidentes nele é denotado por $G - v$.

Para diferenciar os vértices do grafo original e os vértices da árvore, designaremos os últimos de *nodos*. O algoritmo de Gomory-Hu começa com a árvore de cortes trivial em relação à partição $P = \{V_G\}$ contendo apenas um nodo representado pelo conjunto V_G. Os nodos da árvore em construção são conjuntos de vértices do grafo e o conjunto de seus nodos é uma partição de V_G.

O Algoritmo 2.2 traz o pseudocódigo do algoritmo de GH. A cada iteração, o algoritmo GH escolhe um nodo X da árvore T que contenha mais de 1 vértice (linha 3). Os conjuntos de vértices representados pelos nodos de cada componente conexo de $T - X$ são contraídos em G (linhas 4-6). Dois vértices $s, t \in X$ são escolhidos e um s-t-corte mínimo no grafo contraído é computado (linhas 7-8). A árvore é então modificada: o vértice X é dividido em dois vértices S_1 e S_2 pela projeção do corte mínimo sobre X; as arestas da forma $\{A, X\}$ são reconectadas a S_1 ou a S_2 de acordo com o lado do corte em que encontra-se o vértice que resultou da contração da subárvore contendo A (linhas 11-19). O algoritmo termina quando cada nodo da árvore for formado por apenas 1 vértice do grafo.

Um exemplo de execução do algoritmo de Gomory-Hu em um grafo com 6 vértices pode ser visto na Figura 2.3. Cada figura mostra, à esquerda, o grafo e um corte mínimo e, à direita, a árvore em construção. Na primeira iteração, o algoritmo encontrou um corte mínimo que separa os vértices 1 e 2. O único nodo da árvore foi dividido nos nodos $\{1,3\}$ e $\{2,4,5,6\}$ de acordo com o corte indicado na Figura 2.3b. A nova aresta recebe a mesma capacidade do corte, 15. Na iteração seguinte, os vértices 1 e 3 foram escolhidos e um corte mínimo entre eles foi encontrado no grafo contraído da Figura 2.3c. Note que esse grafo contraído só contém 3 nodos. O conjunto de vértices passou a ser $V = \{\{1\},\{3\},\{2,4,5,6\}\}$. O algoritmo continua até que a árvore tenha 6 nodos, cada um representando um único vértice do grafo, que pode ser vista na Figura 2.3f.

Complexidade de tempo de pior caso dos algoritmos Gus e GH Os algoritmos determinísticos com o melhor tempo assintótico de pior caso para o problema do s-t-corte mínimo são os algoritmos de fluxo máximo citados na Seção 1.1 que têm complexidade de tempo de $O(mn)$ multiplicado por um fator logarítmico. Um fator extra de $O(n)$ produz a complexidade de tempo do melhor algoritmo determinístico conhecido para encontrar uma árvore de cortes de

Algoritmo 2.2 Algoritmo de Gomory-Hu

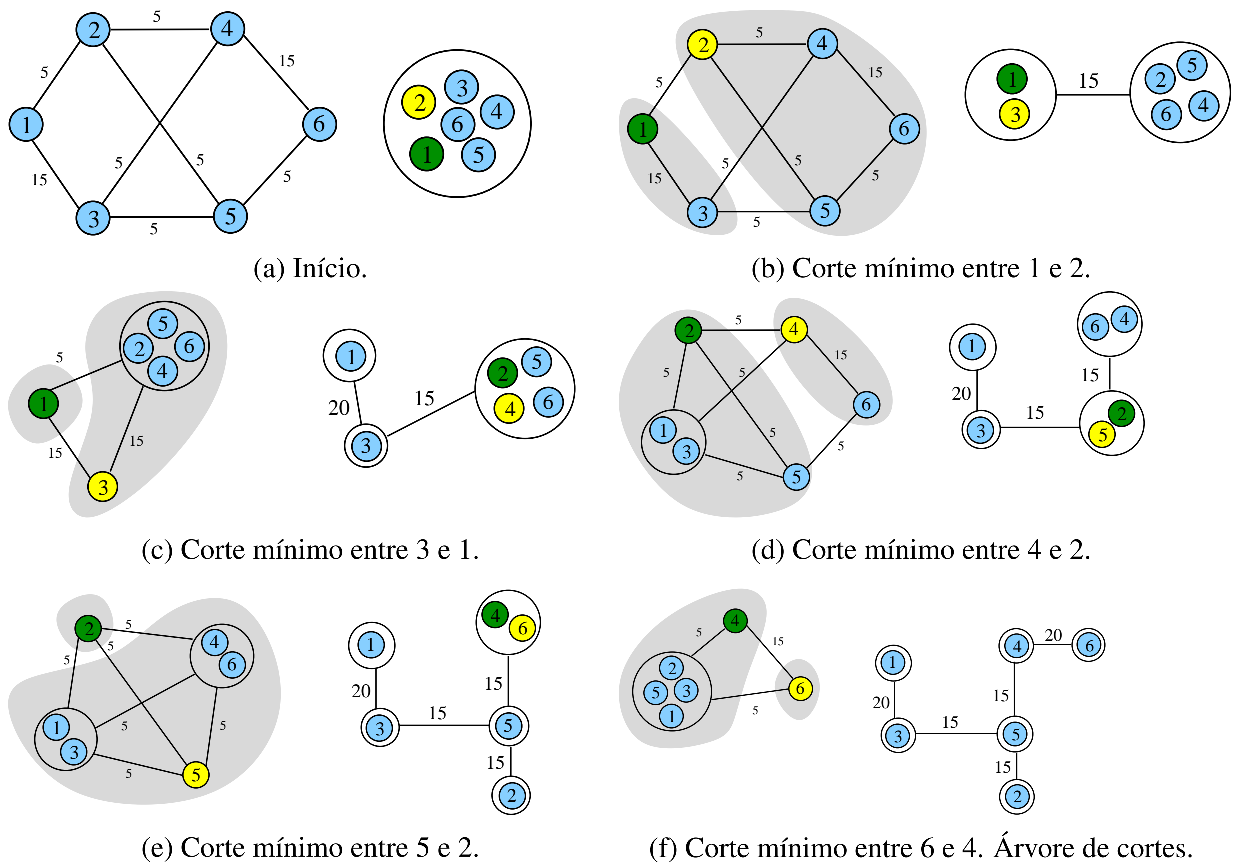
Entrada: $G = (V, E_G, c_G)$ é um grafo não orientado capacitado
Saída: $T = (V, E_T, c_T)$ é uma árvore de cortes de G
1: $T \leftarrow (V_T = \{V\},\ E_T = \emptyset)$ // Árvore em construção
2: **enquanto** $\exists X \in V_T$ tal que $|X| > 1$ **faça**
3: Seja $X \in V_T$ tal que $|X| > 1$
4: Sejam $V_1, V_2, \ldots, V_k$ os nodos dos componentes conexos de $T - X$
5: $X_i \leftarrow \bigcup_{V' \in V_i} V'$, para $1 \leq i \leq k$ // Todos os vértices representados em cada componente
6: $G' \leftarrow G/X_1, X_2, \ldots, X_k$ // Grafo contraído
7: Sejam $s, t \in X$
8: $Y \leftarrow$ `corte_mínimo`(s, t, G')
9: $S_1 \leftarrow Y \cap X$
10: $S_2 \leftarrow \overline{Y} \cap X$
// Atualiza a árvore em construção dividindo X em dois vértices S_1 e S_2
11: $e \leftarrow \{S_1, S_2\}$
12: $c_T[e] \leftarrow d(Y)$
13: **para toda aresta** $e' = \{A, X\} \in E_T$ incidente a X em T **faça**
14: **se** A está do lado Y do corte $\{Y, \overline{Y}\}$ **então**
15: $E_T \leftarrow E_T \cup \{\{A,\ S_1\}\} \backslash \{\{A, X\}\}$
16: **senão**
17: $E_T \leftarrow E_T \cup \{\{A,\ S_2\}\} \backslash \{\{A, X\}\}$
18: $V_T \leftarrow (V_T \backslash \{X\}) \cup \{S_1,\ S_2\}$ // Divide X em duas partes
19: $E_T \leftarrow E_T \cup \{e\}$
20: **devolva** T

um multigrafo capacitado.

Em grafos não capacitados, o algoritmo de Dinits para o problema do fluxo máximo tem complexidade de tempo de pior caso de $O(m^{\frac{3}{2}})$ em geral e $O(n\sqrt{n})$ em grafos esparsos [79]. Portanto, os algoritmos de Gusfield e de Gomory-Hu utilizando o algoritmo de Dinits têm complexidade de tempo de pior caso de $O(nm\sqrt{m})$ em grafos não capacitados e $O(n^2\sqrt{n})$ em grafos não capacitados e esparsos.

2.3 Algoritmos Paralelos para Árvores de Cortes

O processamento paralelo permite acelerar a execução de algoritmos que requerem muito processamento. A ampla disseminação de computadores paralelos e de *clusters* de computadores confere grande importância à disponibilidade de implementações paralelas de soluções para os mais variados problemas computacionais. Apesar da importância das árvores de cortes devido à sua extensa lista de aplicações, nenhuma implementação paralela de algoritmos para árvores de cortes estava publicamente disponível e não tínhamos conhecimento de que algum estudo experimental acerca do assunto tivesse sido publicado. A principal contribuição



(a) Início.

(b) Corte mínimo entre 1 e 2.

(c) Corte mínimo entre 3 e 1.

(d) Corte mínimo entre 4 e 2.

(e) Corte mínimo entre 5 e 2.

(f) Corte mínimo entre 6 e 4. Árvore de cortes.

Figura 2.3: Exemplo de execução do algoritmo de Gomory-Hu.

desta tese, descrita nos Capítulos 3 e 4, consiste na proposta e avaliação de algoritmos paralelos para a construção de árvores de cortes.

A paralelização de algoritmos para árvores de cortes envolve pelo menos duas escolhas. A primeira: determinar qual algoritmo paralelizar. A segunda: escolher o nível de paralelismo a ser implementado. Esse assunto é abordado a seguir.

Comecemos pela escolha do algoritmo. É fato que dentre os algoritmos sequenciais de Gus e GH, nenhum deles domina o outro nos tempos de execução [44]. Depois de executar extensos experimentos com implementações sequenciais desses algoritmos, A. V. Goldberg e K. Tsioutsiouliklis [44] concluem que o algoritmo GH é o mais “robusto”. O motivo é que há instâncias nas quais o desempenho do GH é muito superior, mas quando o desempenho do GH é inferior, a diferença nos tempos de execução é relativamente pequena. Apesar disso, o algoritmo de Gus é mais eficiente na maioria das instâncias, com exceção daquelas que possuem s-t-cortes mínimos balanceados.

Fica claro que o desempenho relativo entre os algoritmos GH e Gus depende da estrutura da instância. A existência de s-t-cortes mínimos balanceados favorece o algoritmo de

Gomory-Hu, porque o tamanho do grafo é reduzido a cada iteração que utilize um corte balanceado. Entretanto, a maioria dos grafos reais e grafos gerados por modelos aleatórios como o modelo de Erdös-Rényi (ER) e o modelo de Barabási-Albert (BA) raramente possuem s-t-cortes mínimos balanceados. Para ilustrar isso, executamos um algoritmo que calcula o s-t-corte mínimo mais balanceado para todos os pares de vértices do grafo. Ao executá-lo em 20 grafos ER com 300 vértices, o s-t-corte mínimo mais balanceado existente foi com 2 vértices de um lado do corte e o restante do outro. Um experimento análogo em grafos BA mostrou que todos os s-t-corte mínimos eram triviais (isto é, um vértice de um lado e os demais no outro). Outros resultados sobre o balanceamento dos cortes são apresentados na Seção 4.4.3. Além disso, nossos experimentos preliminares comparando as versões sequenciais desse dois algoritmos mostraram que o algoritmo de Gusfield é mais rápido do que o algoritmo de Gomory-Hu nas maiores instâncias reais de nosso conjunto de dados e em todos os grafos gerados pelo modelo ER e pelo modelo BA.

A inexistência de cortes balanceados implica que o algoritmo GH não consegue reduzir significativamente os tamanhos dos subproblemas ao longo de sua execução. Também, isso sugere que seja difícil garantir um balanceamento de carga entre os processos em implementações que explorem a sua característica de ser um algoritmo de divisão e conquista.

O algoritmo de Gusfield calcula cortes mínimos sempre no mesmo grafo de entrada. Assim, cada processo pode manter uma cópia do grafo de entrada de forma que não seja necessária a transmissão de grafos ao longo da execução paralela. A comunicação entre os processos consiste exclusivamente dos identificadores dos vértices de origem e destino, em um sentido, e dos cortes e de seus valores no outro. Entretanto, uma implementação paralela do algoritmo de Gomory-Hu necessita transferir mais dados entre os processos uma vez que os grafos ou os conjuntos de vértices a serem contraídos são necessários para cada execução do algoritmo de corte mínimo.

A implementação do algoritmo GH descrita em [44] é bastante otimizada. Em particular, os grafos contraídos não são construídos inteiramente a cada iteração, sendo obtidos a partir de operações sobre o grafo de entrada. Dessa maneira, a implementação evita alocar $\Theta(n^2)$ arestas ao longo das contrações do grafo. Para isso, a primeira recursão é feita no subproblema menor e a segunda recursão reutiliza arestas previamente utilizadas. Uma implementação paralela desse algoritmo de divisão e conquista não pode utilizar essa estratégia, já que necessita construir explicitamente cada grafo no qual um corte mínimo será computado em paralelo.

Os motivos apontados acima indicam que o algoritmo de Gusfield é o mais indicado a ser paralelizado. Entretanto, uma implementação paralela do algoritmo de Gomory-Hu pode

produzir bons resultados em grafos que contêm cortes balanceados. Assim, foram implementadas versões paralelas de ambos os algoritmos com o objetivo de avaliar a eficiência de cada uma delas e determinar se a implementação paralela do algoritmo de GH pode superar ou não a versão paralela do algoritmo de Gus em algumas instâncias. Por fim, uma implementação híbrida entre os dois algoritmos foi implementada na tentativa de obter um desempenho próximo do melhor entre os dois algoritmos independentemente da instância a ser resolvida.

Trataremos agora do grau de paralelismo a ser explorado e justificaremos a opção por paralelizar os laços principais dos algoritmos. A alternativa dessa escolha é utilizar um algoritmo paralelo para o problema do corte mínimo ou do fluxo máximo. Entretanto, os algoritmos para esses problemas são difíceis de serem paralelizados. Apesar de extensa pesquisa sobre fluxo em redes em paralelo, os trabalhos experimentais sobre o assunto reportam *speedups*[2] pequenos devido ao sincronismo necessário para a implementação dos algoritmos. Veja, por exemplo, [7, 60]. Por outro lado, implementações paralelas dos algoritmos Gus e GH podem resolver um corte mínimo por *linha de execução* ou processo, sem sincronização, e alcançar *speedups* elevados, como veremos nas Seções 3.2 e 4.2.

2.4 Revisão da Literatura

Nesta seção apresentamos outros resultados conhecidos sobre as árvores de cortes em casos mais específicos, em casos mais gerais e também sobre as suas aplicações.

Os algoritmos de Gomory-Hu e de Gusfield para a construção de árvores de cortes funcionam em multigrafos não orientados e capacitados. Para casos mais específicos, foram descobertos algoritmos mais eficientes. Dentre eles, destaca-se o algoritmo aleatorizado proposto por A. Bhalgat et al. em [55], que encontra uma árvore de cortes de um grafo *não* capacitado em tempo esperado de $O(mn)$. Entretanto, trata-se de um algoritmo complexo do qual não temos conhecimento de nenhuma implementação. No caso de grafos planares capacitados, um algoritmo de tempo $O(n.\log^4 n)$ foi apresentado por G. Borradaile et al. [15].

O único estudo experimental sobre árvores de cortes foi publicado com o título *"Cut Tree Algorithms"* por A. V. Goldberg and K. Tsioutsiouliklis [45]. O trabalho tem como objetivo comparar os algoritmos GH e Gus. Os autores concluem que uma versão otimizada do algoritmo de GH é mais robusta do que o algoritmo de Gus com base no fato de que o algoritmo de Gus teve um desempenho bastante pior do que o algoritmo de GH em algumas instâncias. No entanto, vale observar que as instâncias onde isso ocorreu são, em sua maioria, grafos sintéticos

[2] O *speedup* é a razão entre o tempo de execução sequencial e o tempo de execução paralelo.

nos quais cortes balanceados existem por construção. Por outro lado, a implementação do algoritmo de Gus foi a mais rápida em muitas das classes de grafos utilizadas nos experimentos.

O problema de encontrar uma árvore de cortes parcial para um grafo não ponderado que represente todos os cortes de aresta de cardinalidade no máximo k foi estudado por Hariharan et al. [54] que apresentam um algoritmo aleatorizado de complexidade esperada de $O(m+nk^3)$ para o problema.

Arikati et al. [5] apresentam algoritmos eficientes para o problema do corte mínimo para todos os pares de vértices em grafos com largura arbórea (*treewidth*) limitada, grafos planares e grafos esparsos.

O conceito de árvores de cortes não tem analogia direta para a aresta conectividade de grafos orientados, como mostra Benczúr em [10], onde apresenta um grafo orientado que possui um número quadrático de s-t-cortes com capacidades distintas. Esse resultado revelou um erro no trabalho de C. P. Schnorr [97] que apresentou uma prova (incorreta) da existência das árvores de cortes de grafos orientados. O mesmo se deu com o artigo de D. Gusfield e D. Naor [49] que com base no artigo de C. P. Schnorr apresentava um algoritmo (incorreto) para a construção de tais árvores.

Os cortes mínimos de *vértices* também não podem ser representados através de uma árvore de cortes [10]. No caso mais geral de grafos com custos associados aos vértices, R. Hassin e A. Levin [58] mostraram que não existe qualquer representação compacta dos cortes de vértices com custo mínimo, pois podem existir $\Theta(n^2)$ valores possíveis para os cortes mínimos.

Uma variação do conceito de corte de vértices, denominado de *separador*, permite a construção de uma árvore de fluxo equivalente para grafos não orientados com uma função de custo arbitrária sobre o conjunto de vértices [58]. Em grafos com capacidades nos vértices e arestas, definimos um *separador* de dois vértices s e t como sendo um conjunto de vértices e/ou arestas que se removidos separam s e t e que pode conter um dos vértices s ou t. Grafos planares com capacidades nos vértices e arestas possuem árvore de fluxo equivalente para separadores de vértices e arestas com custo mínimo [108].

O conceito de árvores de cortes foi generalizado para matróides por D. Hartvigsen em [57] que mostrou que a existência de árvore de cortes de grafos não orientados é um caso particular do caso mais geral das matróides. M. Conforti et al. [29] generalizam as árvores de fluxo equivalente para permitir a recuperação eficiente de $\lambda(s,t)$ caminhos aresta-disjuntos entre qualquer par de vértices de um grafo. S. N. Kabadi et al. [64] estudam a existência de árvores de cortes para o problema do fluxo máximo entre um par de vértices sobre múltiplas

rotas (*multiroute flows*).

O problema de encontrar cortes mínimos entre todos os pares de vértices de um grafo também foi estudado no caso paramétrico, em que os pesos de algumas arestas podem variar. P. Berthomé et al. [11] mostram que 2^k árvores de cortes permitem determinar eficientemente o valor do fluxo máximo entre qualquer par de vértices em um grafo com até k arestas com capacidades variáveis. Os mesmos autores mostram em [8] como determinar cortes mínimos entre todos os pares de vértices sob as mesmas condições. Em [3], outra versão paramétrica do problema é estudada.

As árvores de cortes e as árvores de fluxo equivalente foram amplamente aplicadas na solução de problemas computacionais. Algumas dessas aplicações são apresentadas a seguir.

Dado um grafo $G = (V,E)$ e um conjunto $T \subseteq V$, $|T|$ par, um T-corte em G é um corte $\{X,\overline{X}\}$ tal que $|T \cap X|$ é ímpar. Algoritmos para o problema do T-corte mínimo têm diversas aplicações. Eles são usados, por exemplo, como parte de heurísticas para acelerar soluções exatas para o problema do Caixeiro Viajante [70]. Um T-corte mínimo em um grafo pode ser encontrado através de um algoritmo que encontra primeiro uma árvore de cortes do grafo [84].

Dado um grafo com custos não negativos nas arestas, o problema do k-corte de custo mínimo consiste em encontrar um conjunto de arestas de custo total mínimo que quando removido deixa o grafo com k componentes conexos. Um algoritmo de aproximação com fator 2 para o problema do k-corte de custo mínimo foi proposto por H. Saran e V. V. Vazirani [95] e consiste em remover as arestas dos cortes com menores capacidades, representados na árvore de cortes, até que o grafo tenha k componentes conexos. Generalizações do problema do k-corte cuja solução também envolve árvores de cortes foram estudadas por R. Engelberg et al. [38].

O agrupamento de dados (*clustering*) tem aplicações diversas, como, por exemplo, a localização de grupos em redes sociais e de suas hierarquias [73], na análise da expressão genética [62], na classificação de páginas da Web [40], dentre outras. Métodos de agrupamento em grafos têm como objetivo particionar o grafo em partes com alta conectividade interna e baixa conectividade entre as partes, de forma que conceitos de cortes mínimos são naturalmente úteis. E. Hartuv e R. Shamir [56] apresentam um método de agrupamento baseado em cortes mínimos e o comparam com outros métodos de agrupamento em dados biológicos. G. W. Flake et al. [40] apresentam um método de agrupamento em grafos que acrescenta um vértice sorvedouro artificial ao grafo e particiona o grafo com base em uma árvore de cortes. O artigo mostra que o agrupamento encontrado satisfaz certos critérios de qualidade. O algoritmo proposto é aplicado ao agrupamento de artigos científicos e à categorização de páginas da Web. Mais recentemente, B. Saha e P. Mitra [93] apresentam uma variação do algoritmo de agrupamento de G. W. Flake

et al. para o caso dinâmico em que arestas podem ser inseridas e removidas do grafo. Esse trabalho foi corrigido e estendido por R. Görke et al. [48, 47]. Outro exemplo de aplicação prática de agrupamento baseado em árvores de cortes é a modularização de software [61].

3 Implementações Paralelas do Algoritmo de Gusfield para Árvores de Cortes

Este capítulo e o próximo apresentam descrições de implementações paralelas de algoritmos para a construção de árvores de cortes. São descritos resultados experimentais obtidos com essas implementações que mostram uma boa aceleração no tempo de execução em um conjunto variado de instâncias do problema. Foram implementadas versões paralelas do algoritmo de Gusfield e do algoritmo de Gomory-Hu e também uma versão híbrida que combina os dois algoritmos em uma implementação que busca tirar proveito das vantagens de ambos os algoritmos.

As implementações têm como alvo arquiteturas MIMD (*Multiple Instruction stream, Multiple Data stream*). Do algoritmo de Gusfield, apresentamos uma implementação destinada à execução em arquiteturas de multiprocessamento simétrico ou *SMP* (*Symmetric Multiprocessing*) que utiliza a biblioteca *OpenMP* [22, 89]. As outras implementações podem ser utilizadas tanto em *clusters* computacionais como em arquiteturas *SMP* e utilizam a biblioteca *MPI* (*Message Passing Interface*) [83, 89].

Este capítulo está dividido da seguinte maneira: a Seção 3.1 apresenta as descrições das implementações paralelas do algoritmo de Gusfield para árvores de cortes utilizando *MPI* e *OpenMP* e a Seção 3.2 apresenta os resultados experimentais dessas implementações.

3.1 Versões Paralelas do Algoritmo de Gusfield

Uma análise superficial do algoritmo de Gusfield pode levar à conclusão de que o seu laço principal não pode ser paralelizado com eficiência, pois existe dependência entre as iterações, uma vez que a árvore em construção é atualizada a cada corte mínimo encontrado. Entretanto, os resultados apresentados adiante mostram que o algoritmo de Gusfield pode ser paralelizado com relativa facilidade e de forma a produzir *speedups* significativos.

Como visto no capítulo anterior, o algoritmo sequencial de Gusfield executa $|V|-1$

chamadas a um algoritmo para o problema do s-t-corte mínimo. Cada execução encontra um corte mínimo no grafo de entrada que separa um vértice, chamado de *origem*, de seu vizinho na árvore em construção. A estratégia usada para paralelizar o algoritmo foi a de executar essas $|V|-1$ iterações em paralelo de forma otimista. À medida que as execuções da rotina de corte mínimo terminam, o algoritmo faz tentativas de modificar a árvore de cortes em construção. Essa tentativa será bem sucedida somente se os vértices separados pelo corte mínimo encontrado ainda forem vizinhos na árvore. Caso outra execução paralela tenha modificado a árvore e os dois vértices sendo separados não sejam mais vizinhos na árvore, o corte mínimo encontrado é rejeitado e um novo corte mínimo separando a origem de seu novo vizinho na árvore deve ser calculado.

A estratégia de paralelização descrita acima pode ser facilmente implementada tanto com *OpenMP* quanto com *MPI*. Com *OpenMP*, o laço principal do algoritmo Gus é executado em paralelo e cada iteração é repetida até que o algoritmo seja bem sucedido em modificar a árvore em construção. As operações sobre a árvore em construção são executadas em exclusão mútua para garantir que sejam efetuadas corretamente. O escalonamento das tarefas nos núcleos dos processadores é feito implicitamente pela *OpenMP*. A implementação em *MPI* utiliza o modelo mestre/escravo no qual o processo mestre faz as atualizações da árvore e os escravos computam os cortes mínimos.

O desempenho das implementações paralelas do algoritmo de Gusfield depende da estrutura do grafo. Por isso, executamos um conjunto de experimentos que cobre diversas classes de grafos sintéticos bem como algumas redes complexas reais. A seguir são apresentadas as descrições das versões em *MPI* e *OpenMP* do algoritmo de Gusfield.

3.1.1 Versão Paralela do Algoritmo de Gusfield: *MPI*

Message Passing Interface (*MPI*) é uma biblioteca de rotinas para o gerenciamento de processos e para a troca de mensagens em arquiteturas paralelas com memória distribuída [22]. Uma das aplicações da *MPI* é a programação paralela em *clusters* que são formados por um conjunto de computadores independentes conectados por uma rede. A principal vantagem da *MPI* é a escalabilidade, já que, em princípio, as aplicações podem executar em um número ilimitado de computadores sem que concorram pelo acesso a uma memória centralizada. A eficiência de algoritmos paralelos implementados com *MPI* depende da quantidade de sincronização entre os processos e da frequência e do volume da comunicação entre eles.

A implementação do algoritmo de Gusfield com *MPI* utiliza o modelo mestre-escravo. O processo mestre é responsável por enviar os subproblemas aos escravos e também por ma-

nipular a árvore de cortes em construção. O pseudocódigo da implementação com *MPI* do algoritmo de Gusfield pode ser visto no Algoritmo 3.1.

Algoritmo 3.1 Algoritmo de Gusfield - Versão Paralela com *MPI*

Entrada: $G = (V, E_G, c_G)$, $proc_j$ processadores $(0 \leq j < P)$
Saída: $T = (V, E_T, c_T)$ uma árvore de fluxo equivalente G

```
 1: se proc_j = 0 então // processo mestre
 2:    para i ← 1 até |V_T| faça
 3:       tree[i] ← 1
 4:    para s ← 1 até P − 1 faça
 5:       envie tarefa (s, tree[s]) para proc_s
 6:    enquanto s < |V| faça
 7:       receba resultado (s′, t′, X) de proc_j
 8:       se tree[s′] = t′ então // atualiza a árvore
 9:          λ[s′] ← d(X)
10:          para u ∈ V, u > s′ faça
11:             se tree[u] = t′ e u ∈ X então
12:                tree[u] ← s′
13:          s ← s + 1
14:          envie tarefa (s, tree[s]) para proc_j
15:       senão // a tarefa falhou; gere nova tarefa para s
16:          envie tarefa (s′, tree[s′]) para proc_j
17: senão // processo escravo
18:    enquanto existe uma tarefa faça
19:       receba tarefa (s, t)
20:       X ← corte_mínimo(s, t, G)
21:       envie (s, t, X) para proc_0
    // Construa a árvore T
22: E_T ← ∅
23: para s ← 2 to |V| faça
24:    E_T ← E_T ∪ {{s, tree[s]}}
25:    c_T({s, tree[s]}) ← λ[s]
26: devolva T = (V, E_T, c_T)
```

Os processos são numerados de 0 até $p-1$. O processo mestre é o $proc_0$ e os escravos são os processos $proc_1, .., proc_{p-1}$. Cada processo mantém uma cópia do grafo de entrada. O processo mestre cria tarefas e as envia para os processos escravos (veja a linha 5 do pseudocódigo). Uma tarefa é definida por um par de nodos (s,t) que define a origem e o destino de uma instância do problema do s-t-corte mínimo. Como na versão sequencial, a árvore é representada por um vetor $tree[.]$ tal que $tree[i]$ é o nodo vizinho de i mais próximo da raiz. A raiz é o vértice 1. A capacidade da aresta $\{i, tree[i]\}$ é armazenada em $\lambda[i]$.

Quando um processo escravo conclui a execução do algoritmo de corte mínimo, ele envia ao processo mestre a capacidade do corte mínimo encontrado e uma lista dos nodos que

definem o corte. Ao receber a resposta, o processo mestre pode atualizar a árvore de cortes em construção desde que o nodo s ainda seja vizinho do nodo t (veja a linha 8 do pseudocódigo). A atualização da árvore de cortes é semelhante à que é feita no algoritmo sequencial. Se s e t não forem mais vizinhos no momento desse processamento, então dizemos que a tarefa associada ***falhou*** e outra tarefa cuja origem é s e o destino é o novo vizinho de s na árvore é produzida (linha 16).

A estrutura do grafo influencia no número de tarefas falhas. Quando o conjunto X que define o s-t-corte $\{X,\overline{X}\}$ é pequeno, a árvore sofre poucas alterações e, portanto, menos tarefas devem falhar. O *speedup* da execução paralela depende do número de tarefas falhas ao longo de sua execução.

3.1.2 Versão Paralela do Algoritmo de Gusfield: *OpenMP*

OpenMP (*Open Multi-Processing*) é uma interface de programação de aplicação (*Application Programming Interface* - API) projetada para facilitar a programação paralela em arquiteturas de memória compartilhada SMP (*Symmetric Multiprocessing*). Essa API fornece diretivas que estendem as linguagens Fortran, C e C++, que definem como o processamento é compartilhado por *threads* a serem executadas em diferentes processadores ou núcleos e também como os dados da memória compartilhada são acessados pelas *threads* [22].

A implementação do algoritmo de Gusfield em *OpenMP* foi feita pela paralelização do laço principal que faz as $|V|-1$ chamadas da rotina que calcula fluxo máximo. Veja o Algoritmo 3.2. A árvore é representada pelor vetor $tree[.]$ e a capacidade da aresta $\{i, tree[i]\}$ é armazenada em $\lambda[i]$, como nas versões do algoritmo de Gusfield apresentadas anteriormente.

Seja k o número máximo pré-definido de *threads*. O algoritmo utiliza uma estratégia otimista e encontra k s-t-cortes mínimos em paralelo. Cada um desses cortes é usado em uma tentativa de modificar a árvore de cortes em construção. Cada *thread*, após encontrar um s-t-corte, verifica se o vértice de destino, t, ainda é o vizinho de s na árvore. Isso não ocorre quando um corte encontrado por outra *thread* foi bem sucedido em modificar a árvore e separou s de t. Nesse caso, dizemos que a *thread* ***falhou*** e um outro s-t-corte é calculado para separar s do seu novo vizinho na árvore. Caso a *thread* seja bem sucedida em separar s de t, ela atualiza a árvore em construção. Os testes e as modificações da árvore são feitos em exclusão mútua, dentro de uma região crítica, para garantir a correção do algoritmo.

Algoritmo 3.2 Algoritmo de Gusfield - Versão Paralela com *OpenMP*

Entrada: $G = (V, E_G, c_G)$
Saída: $T = (V, E_T, c_T)$ é uma árvore de fluxo equivalente de G

```
    // Inicialização
 1: para i ← 1 até |V| faça
 2:     tree[i] ← 1
    // |V|−1 iterações
 3: para s ← 2 até |V| em paralelo faça
 4:     sucesso ← falso
 5:     repita
 6:         t ← tree[s]
 7:         X ← corte_mínimo(s, t, G)
 8:         Início da Região Crítica
 9:         se tree[s] = t então // Atualiza a árvore
10:             λ[s] ← d(X)
11:             para u ∈ V, u > s faça
12:                 se tree[u] = t′ e u ∈ X então
13:                     tree[u] ← s
14:             sucesso ← verdadeiro
15:         Fim da Região Crítica
16:     until sucesso
    // Construa a árvore T
17: E_T ← ∅
18: para s ← 2 to |V| faça
19:     E_T ← E_T ∪ {{s, tree[s]}}
20:     c_T({s, tree[s]}) ← λ[s]
21: devolva T = (V, E_T, c_T)
```

3.2 Resultados Experimentais

A seguir apresentamos a descrição dos experimentos e os resultados obtidos com as versões paralelas do algoritmo de Gusfield.

Ambiente Computacional Os experimentos com *MPI* foram rodados em um *cluster* composto de 14 computadores com processadores Intel Core 2 Quad 2.4 GHz, com 2 GB de memória e 4MB de memória *cache* interconectados por uma rede Ethernet de 1Gbps[1]. Os experimentos para determinar o desempenho da implementação paralela do algoritmo de Gusfield com *OpenMP* foram executados em um computador com processador Quad-Core AMD Opteron com 2.8 GHz de *clock* de 8 núcleos, 8 GB de memória e 512 KB de memória *cache*.

As implementações foram escritas em linguagem C e compiladas com o *gcc* com nível

[1] O *cluster* utilizado nos experimentos é o do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é financiado pela FINEP através dos projetos CT-INFRA/UFPR.

de otimização O3.

Descrição do Conjunto de Instâncias O conjunto de dados de testes é composto por 10 grafos representando diferentes classes de grafos reais e sintéticos como mostrado na Tabela 3.1. Os 4 primeiros grafos foram obtidos a partir de redes reais: 2 redes de colaboradores em trabalhos científicos [63, 9], uma rede de transmissão elétrica [105] e uma rede *peer-to-peer* [63]. Duas redes foram geradas por modelos aleatórios: o modelo binomial de Erdös-Rényi (ER) [12] e o modelo de conexões preferenciais de Barabási-Albert (BA) [2]. As outras 4 instâncias são grafos sintéticos de diferentes tipos utilizados em outros trabalhos experimentais sobre cortes mínimos e árvores de cortes [23, 44]. Apresentamos a seguir uma descrição sucinta desses grafos sintéticos. Para maiores detalhes, consulte [23].

Tabela 3.1: Lista de grafos utilizados nos experimentos.

Grafo	$\|V\|$	$\|E\|$
CA-HepPh	11204	235238
GeoComp	3621	9461
PowerGrid	4941	6594
P2P-Gnutella	10876	39994
BA	10000	49995
ER	10000	49841
DBLCYC	1024	2048
NOI	1500	562125
PATH	2000	21990
TREE	1500	563625

Grafos ER Um grafo do modelo Erdös-Rényi ou $G_{n,p}$ é produzido tomando-se n vértices e conectando cada par de vértices com probabilidade p.

Grafos BA O modelo de Barabási-Albert produz grafos através de um processo iterativo que adiciona um vértice por rodada. Um parâmetro m diz quantas arestas são adicionadas em cada rodada. Novas arestas ligam o novo vértice a um vértice previamente inserido cuja escolha é feita com uma probabilidade proporcional ao grau de cada vértice.

NOI - Grafos com Agrupamentos São grafos aleatórios com o conjunto de vértices dividido em grupos de forma que as arestas internas aos grupos tendem a ter capacidades maiores do que as arestas entre grupos. A ideia é forçar a existência de cortes pequenos que separam conjuntos grandes de vértices. Essa classe de grafos foi proposta por H. Nagamochi, T. Ono e T. Ibaraki em um trabalho experimental sobre cortes mínimos [80], por isso o acrônimo NOI.

Os grafos NOI são gerados com os seguintes parâmetros:

n número de vértices

d densidade de arestas em porcentagem: $m = \frac{d}{100}\frac{n(n-1)}{2}$.

k número de componentes densos.

P a capacidade das arestas internas aos componentes é um valor aleatório no intervalo $[1, 100 \cdot P]$ e das arestas entre componentes é um valor aleatório no intervalo $[1, 100]$.

O grafo é construído assim: no início, os n vértices são rotulados com valores de $\{1, 2, \ldots, k\}$ alternadamente. Um ciclo aleatório é introduzido para garantir que o grafo será conexo. Depois, $m - n$ arestas são adicionadas aleatoriamente. As capacidades são aleatórias, respeitando: se a aresta é formada por vértices de rótulos diferentes, então a capacidade é um número no intervalo $[1, 100]$; senão, a capacidade estará no intervalo $[1, 100 \cdot P]$.

A instância NOI utilizada nos experimentos tem 1500 vértices divididos em 6 grupos e com densidade de arestas de 50%.

PATH Os grafos PATH são grafos aleatórios que possuem um caminho de tamanho k com capacidades altas. Cada vértice fora desse caminho é conectado aleatoriamente ao caminho por uma aresta também com capacidade alta. As demais arestas são aleatórias e têm capacidades baixas. Os cortes mínimos tendem a utilizar somente uma aresta com capacidade alta, o que aumenta as chances de existirem cortes mínimos não triviais. Observe que se $k = n$ então o grafo possui um caminho de capacidades altas contendo todos os vértices. Se $k = 1$ então o subgrafo com capacidades altas é uma estrela e os s-t-cortes mínimos tendem a ser triviais. Com efeito, o comprimento desse caminho controla a existência de cortes mais balanceados.

Os parâmetros são:

n número de vértices

d densidade do grafo em porcentagem

k comprimento do caminho inicial

P as arestas do caminho têm capacidades aleatórias no intervalo $[1, 100 \cdot P]$ e as demais em $[1, 100]$.

O grafo PATH de nosso conjunto de instâncias tem 2000 vértices, $k = 5$ e densidade de 50%.

TREE Essa classe é semelhante à classe PATH, descrita acima, porém constrói uma árvore aleatória com capacidades altas conectando os primeiros k vértices ao invés de construir um caminho. Os vértices de 2 até k são conectados aleatoriamente a um dos vértices previamente inseridos. Arestas de capacidade baixa são inseridas aleatoriamente até que o grafo atinja a densidade desejada. Se $k = n$ então o grafo contém uma árvore espalhada aleatória cujas arestas têm capacidades altas. Os parâmetros são análogos aos da classe PATH.

DBLCYC 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 [71].

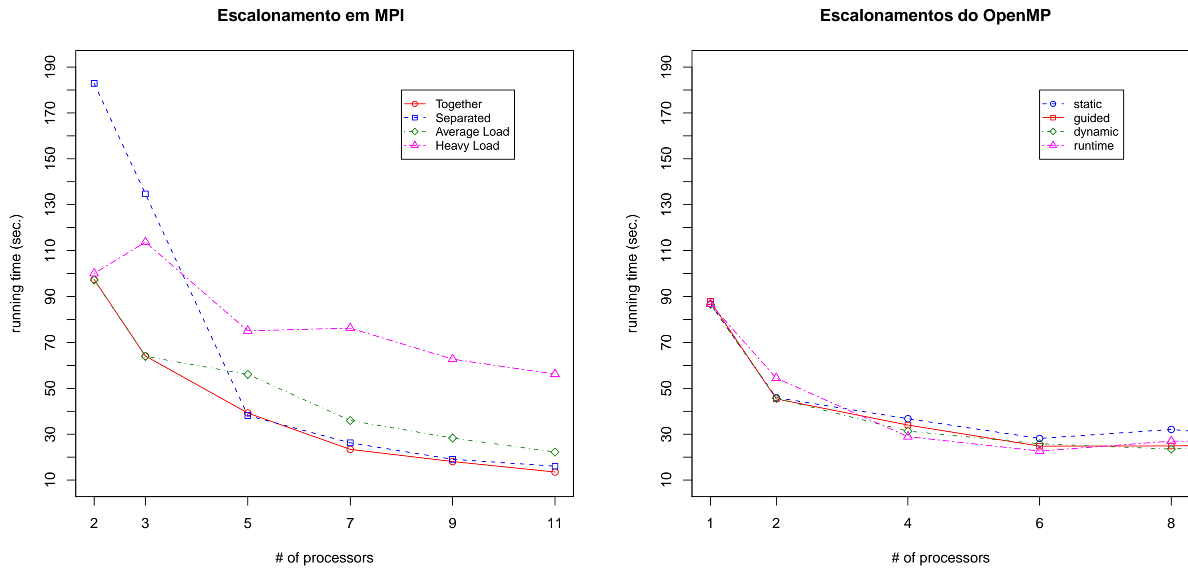
Medidas de Desempenho Paralelo O *speedup*, denotado por S, é uma medida frequentemente utilizada para quantificar a melhora do tempo de execução paralelo em relação ao tempo de execução sequencial e é definido como $S = T_S/T_P$, onde T_S é o tempo da execução sequencial e T_P é o tempo da execução paralelo utilizando P processadores. A *eficiência*, E, é o *speedup* normalizado e é calculada como $E = S/P$.

Vamos deduzir limitantes superiores para o *speedup* e para a eficiência das execuções da implementação do algoritmo de Gusfield com MPI. Seja P o número de processos. Para $P = 2$, o algoritmo executa o processo mestre e 1 processo escravo. Enquanto o mestre processa a árvore em construção, o processo escravo aguarda por uma tarefa e, portanto, a execução é sequencial. Assim, o *speedup* $S_2 \leq 1$ e a eficiência $E_2 \leq 0.50$.

Para $P > 2$, pelo menos um dentre os $P-1$ processos escravos aguarda enquanto o mestre atualiza a árvore em construção. Uma aproximação do tempo sequencial é dada por $T_s = T_F + T_T$, onde T_F é o tempo de computação de $|V|-1$ execuções do algoritmo de fluxo máximo e T_T é o tempo gasto nas atualizações da árvore em construção. Uma cota inferior para o tempo paralelo pode ser obtida tomando-se o caso em que as tarefas de fluxo máximo são divididas de forma balanceada entre os $P-1$ processos escravos. Cada escravo leva $\frac{T_F}{P-1}$ para computar fluxos máximos e espera parado por novas tarefas o tempo de $\frac{T_T}{P-1}$. Portanto, o tempo paralelo com P processadores, T_P, é maior do que ou igual a $\frac{T_F+T_T}{P-1}$. O *speedup* tem limite superior de $P-1$, porque

$$S_P = \frac{T_s}{T_P} \leq \frac{T_F + T_T}{\frac{T_F+T_T}{P-1}} = P-1.$$

Uma cota superior para a eficiência é $E_P = \frac{S_P}{P} \leq \frac{P-1}{P}$.



(a) Comparação de estratégias de escalonamento de processos em *MPI*.

(b) Comparação de estratégias de escalonamento de *threads* com *OpenMP*.

Figura 3.1: Testes preliminares para comparar estratégias de escalonamento.

Resultados Experimentais O primeiro experimento com *MPI* teve como objetivo identificar a melhor forma de distribuir os processos mestre e escravos entre os computadores. Foram testadas as seguintes combinações:

Juntos (*Together*) O processo mestre e um processo escravo são executados em um mesmo computador e os demais processos escravos são executados sozinhos nos outros computadores.

Separados (*Separated*) O processo mestre e cada processo escravo rodam em computadores diferentes.

Carga média (*Medium Load*) Cada computador recebe 2 processos.

Carga Alta (*Heavy Load*) Cada computador recebe 4 processos.

A Figura 3.1a mostra os resultados da comparação desses estratégias. Os resultados foram melhores para a estratégia que executa um processo escravo no mesmo computador do processo mestre e os demais processos em computadores separados. Esse resultado era esperado, pois o processo mestre executa menos processamento que os escravos já que o cálculo dos cortes mínimos consome mais tempo computacional do que o tempo de modificação da árvore em construção. Dessa maneira, a alocação de um computador exclusivo para o processo mestre faz com que o computador fique ocioso enquanto aguarda a resposta dos processos escravos. Por outro lado, alocar mais de um processo escravo por computador reduz a eficiência devido à contenção pela memória.

Testes inicias com *OpenMP* foram realizados para determinar a melhor estratégia de escalonamento das *threads*. O *OpenMP* provê as seguintes estratégias de escalonamento para os laços:

***static*:** A carga é dividida de forma estática em tempo de compilação. Não há balanceamento de carga em tempo de execução. O código produzido para essa estratégia é mais eficiente, porém processadores podem ficar ociosos caso as *threads* não sejam muito parecidas.

***dynamic*:** As *threads* recebem blocos de tarefas de tamanho fixo à medida que ficam ociosas.

***guided*:** As *threads* recebem blocos de tarefas à medida que ficam ociosas. O tamanho dos blocos diminui ao longo da execução com o objetivo de melhorar o balanceamento da carga.

***runtime*:** Permite a escolha, em tempo de execução, de uma das 3 estratégias listadas acima.

Os resultados comparando as diferentes estratégias de escalonamento são mostrados na Figura 3.1b. Os resultados foram muito parecidos, com pequena vantagem para a estratégia dinâmica. Todos os demais experimentos aqui reportados utilizam a estratégia dinâmica de escalonamento.

Resultados com *MPI* Os resultados dos experimentos com a implementação em *MPI* são mostrados na Tabela 3.2 e nas Figuras 3.2, 3.3 e 3.4. Os dados apresentados correspondem a médias de 10 execuções para cada situação. A tabela apresenta as médias dos tempos de execução em segundos, dos *speedups* e das eficiências para cada instância e para cada número de processos de 2 até 15. A Figura 3.2 apresenta um gráfico com os *speedups* médios de todas as execuções.

As Figuras 3.3 e 3.4 apresentam os tempos de execução e o número de laços falhos em um gráfico separado para cada instância. Os tempos de execução com o número de processos variando de 2 até 15 são plotados como círculos conectados por linhas. Os números de laços falhos são plotados como triângulos.

A eficiência das execuções nas instâncias TREE e ER alcança o limite superior de $\frac{P-1}{P}$ para P entre 2 e 5 e não estão longe do máximo para valores maiores de P. Para o número máximo de processos, $P = 15$, nessas instâncias, a eficiência foi de 89% e 86%, respectivamente (o valor máximo é igual a $14/15 \sim 0,93$). Veja a Tabela 3.2.

A eficiência das execuções paralelas foi alta, em particular para as instâncias CA-HepPH, P2P-Gnutella, BA, ER, NOI e TREE. Nas instâncias DBLCY e PATH a eficiência foi

em torno de $0,60$. Somente nas instâncias Geocomp e Powergrid a eficiência foi inferior a $0,50$. A eficiência diminui a medida que o número de laços falhos aumenta como pode ser visto, por exemplo, na Figura 3.3a, onde os triângulos representam a média do número de laços falhos (como definidos na Seção 3.1.2).

A implementação do algoritmo de Gusfield com *MPI* produziu bons *speedups* sem a necessidade de heurísticas para evitar laços falhos. Entretanto, nos casos de instâncias que produzem um maior número de falhas, algumas heurísticas podem ser consideradas. Uma delas consiste em escolher o s-t-corte mínimo $\{X, \overline{X}\}$, que minimize $|X|$, na tentativa de reduzir o número de alterações da árvore em construção. Outra heurística de consequência menos direta consiste em escolher os vértices de origem em uma ordem particular, por exemplo, em ordem decrescente dos graus dos vértices, com o objetivo de maximizar a chance do corte escolhido ser trivial em relação à origem. Essas heurísticas foram implementadas e os resultados são apresentados adiante.

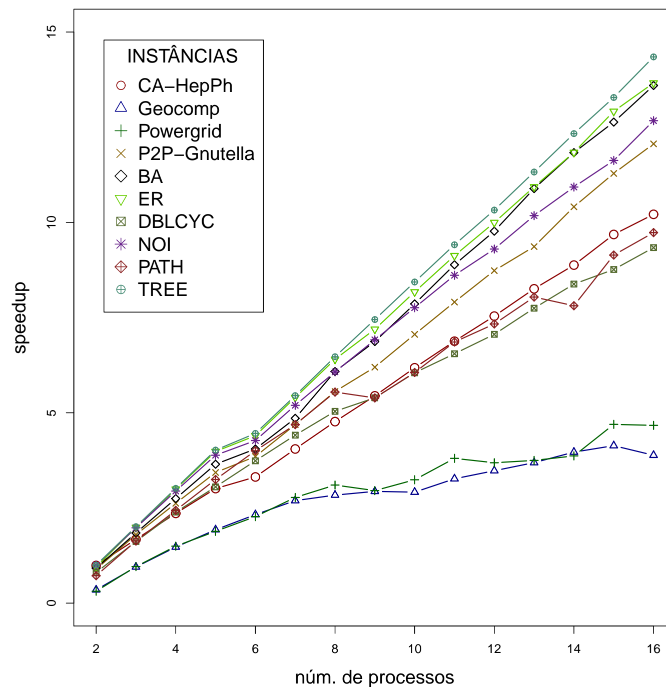
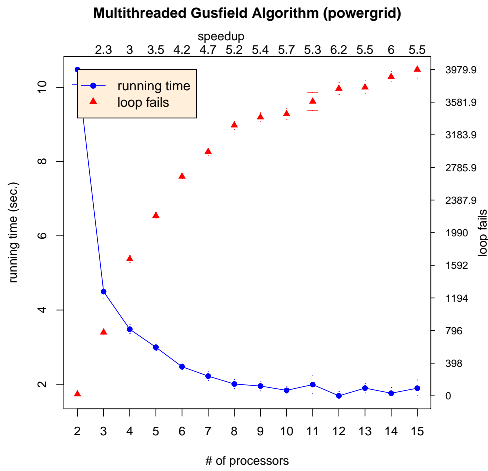


Figura 3.2: *Speedups* das execuções com *MPI*.

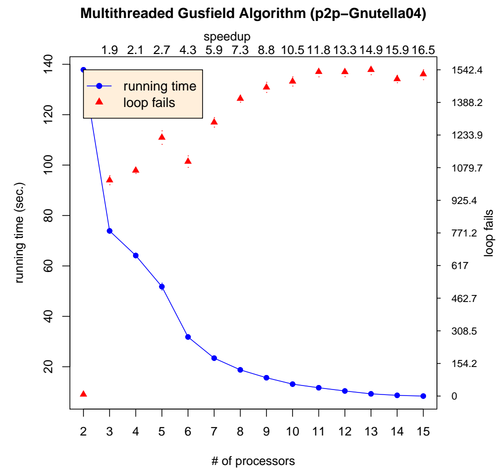
Tabela 3.2: Resultados da implementação do algoritmo de Gusfield com *MPI*. A tabela mostra as médias dos tempos de execução em segundos, dos *speedups* (S) e da eficiência (E) para cada instância.

Número de	**CA-HepPh**			**Geocomp**			**Powergrid**			**P2P-Gnutella**			**BA**		
procs.	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E
sequencial	475.01	-	-	2.25	-	-	3.85	-	-	65.23	-	-	87.71	-	-
2	479.93	0.99	0.49	6.45	0.35	0.17	12.31	0.31	0.16	73.24	0.89	0.45	95.49	0.92	0.46
3	283.42	1.68	0.56	2.37	0.95	0.32	4.01	0.96	0.32	36.25	1.80	0.60	47.58	1.84	0.61
4	201.56	2.36	0.59	1.53	1.47	0.37	2.57	1.50	0.37	24.88	2.62	0.66	31.94	2.75	0.69
5	157.89	3.01	0.60	1.17	1.93	0.39	2.04	1.89	0.38	19.01	3.43	0.69	24.04	3.65	0.73
6	143.35	3.31	0.55	0.97	2.32	0.39	1.69	2.27	0.38	16.94	3.85	0.64	21.66	4.05	0.67
7	117.33	4.05	0.58	0.84	2.69	0.38	1.39	2.77	0.40	13.92	4.69	0.67	18.07	4.85	0.69
8	99.64	4.77	0.60	0.79	2.84	0.35	1.24	3.10	0.39	11.73	5.56	0.69	14.42	6.08	0.76
9	87.25	5.44	0.60	0.77	2.94	0.33	1.30	2.96	0.33	10.52	6.20	0.69	12.77	6.87	0.76
10	76.84	6.18	0.62	0.77	2.91	0.29	1.19	3.24	0.32	9.24	7.06	0.71	11.16	7.86	0.79
11	69.05	6.88	0.63	0.69	3.27	0.30	1.01	3.80	0.35	8.25	7.91	0.72	9.87	8.89	0.81
12	63.01	7.54	0.63	0.65	3.48	0.29	1.04	3.69	0.31	7.47	8.73	0.73	8.98	9.77	0.81
13	57.54	8.26	0.64	0.61	3.69	0.28	1.03	3.75	0.29	6.97	9.36	0.72	8.06	10.89	0.84
14	53.50	8.88	0.63	0.57	3.97	0.28	0.99	3.87	0.28	6.27	10.41	0.74	7.41	11.83	0.85
15	49.06	9.68	0.65	0.54	4.14	0.28	0.82	4.70	0.31	5.78	11.29	0.75	6.94	12.63	0.84

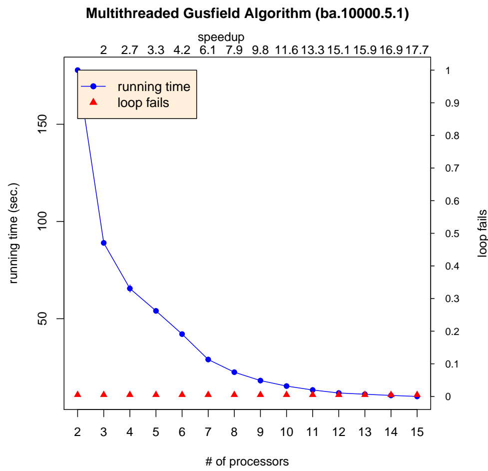
Número de	**DBLCYC**			**ER**			**NOI**			**PATH**			**TREE**		
procs.	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E
sequencial	11.13	-	-	104.23	-	-	384.84	-	-	5.42	-	-	236.78	-	-
2	13.83	0.81	0.40	109.90	0.95	0.47	385.61	1.00	0.50	7.52	0.72	0.36	237.10	1.00	0.50
3	6.88	1.62	0.54	52.40	1.99	0.66	194.59	1.98	0.66	3.32	1.63	0.54	117.89	2.01	0.67
4	4.66	2.39	0.60	34.82	2.99	0.75	130.75	2.94	0.74	2.22	2.44	0.61	78.53	3.02	0.75
5	3.65	3.05	0.61	26.17	3.98	0.80	99.07	3.88	0.78	1.67	3.25	0.65	58.90	4.02	0.80
6	2.98	3.74	0.62	23.70	4.40	0.73	90.11	4.27	0.71	1.35	4.00	0.67	53.11	4.46	0.74
7	2.52	4.41	0.63	19.30	5.40	0.77	74.08	5.20	0.74	1.16	4.69	0.67	43.45	5.45	0.78
8	2.21	5.04	0.63	16.27	6.40	0.80	63.36	6.07	0.76	0.98	5.54	0.69	36.59	6.47	0.81
9	2.06	5.39	0.60	14.49	7.19	0.80	55.65	6.92	0.77	1.01	5.39	0.60	31.79	7.45	0.83
10	1.84	6.05	0.60	12.75	8.17	0.82	49.59	7.76	0.78	0.89	6.06	0.61	28.07	8.43	0.84
11	1.70	6.55	0.60	11.42	9.13	0.83	44.70	8.61	0.78	0.79	6.86	0.62	25.16	9.41	0.86
12	1.58	7.06	0.59	10.43	10.00	0.83	41.38	9.30	0.77	0.74	7.33	0.61	22.94	10.32	0.86
13	1.44	7.75	0.60	9.53	10.93	0.84	37.81	10.18	0.78	0.67	8.04	0.62	20.91	11.32	0.87
14	1.33	8.38	0.60	8.80	11.84	0.85	35.21	10.93	0.78	0.69	7.81	0.56	19.20	12.33	0.88
15	1.27	8.76	0.58	8.07	12.92	0.86	33.10	11.63	0.78	0.59	9.14	0.61	17.83	13.28	0.89



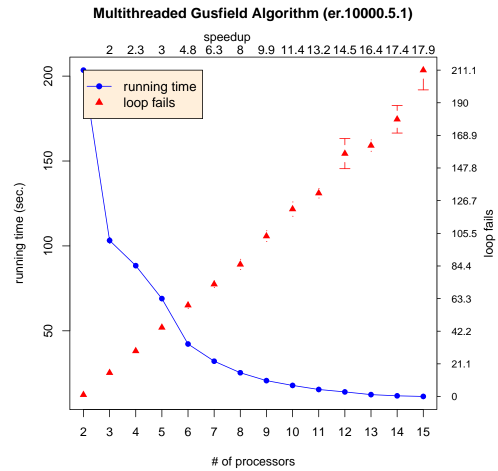
(a) Powergrid



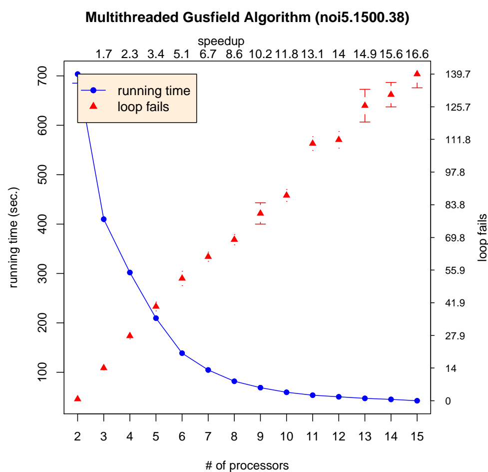
(b) P2P-Gnutella



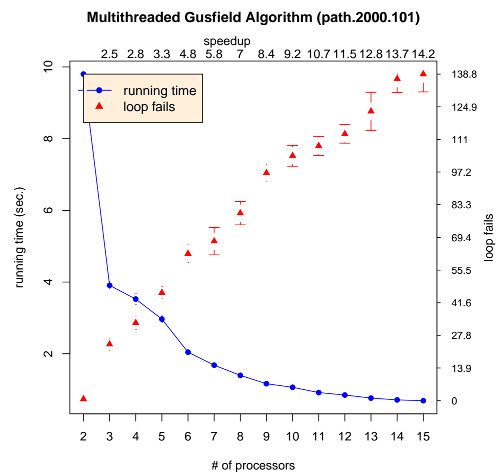
(c) BA



(d) ER

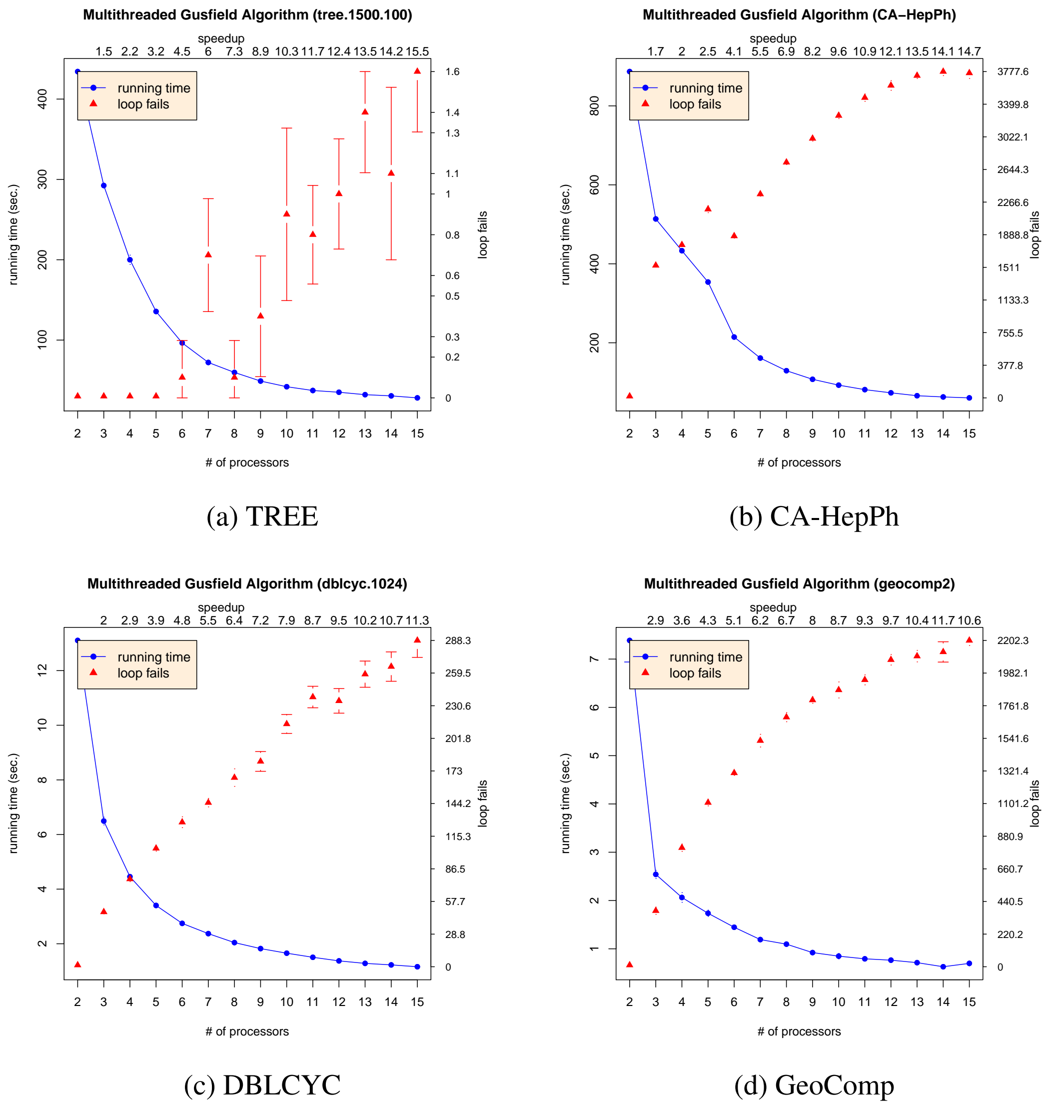


(e) NOI



(f) PATH

Figura 3.3: Tempos de execução e número de laços falhos em execuções do algoritmo de Gusfield com *MPI*. Os círculos representam as médias dos tempos de execução em segundos e os triângulos representam o número de laços falhos. Os *speedups* médios são mostrados no topo de gráfico. Continua na Figura 3.4.



(a) TREE

(b) CA-HepPh

(c) DBLCYC

(d) GeoComp

Figura 3.4: Continuação da Figura 3.3. Tempos de execução e número de laços falhos em execuções do algoritmo de Gusfield com *MPI*. Os pontos representam as médias dos tempos de execução em segundos e os triângulos representam o número de laços falhos. Os *speedups* médios são mostrados no topo de gráfico.

Resultados com *OpenMP* Os resultados obtidos com a implementação baseada em *OpenMP* do algoritmo de Gusfield executados em uma máquina com 8 núcleos são mostrados na Tabela 3.3 e nas Figuras 3.5, 3.6 e 3.7, análogas às apresentadas na seção sobre *MPI*.

A média das eficiências foi acima de $0,50$ na maioria das instâncias. O melhor *speedup* obtido com *OpenMP* em grafos reais foi de 4,47 e em grafos sintéticos foi de 6,02. Os piores *speedups* em redes reais e em redes sintéticas foram de 3,18 e 3,77, respectivamente. A Figura 3.5 apresenta os *speedups* médios de todas as execuções.

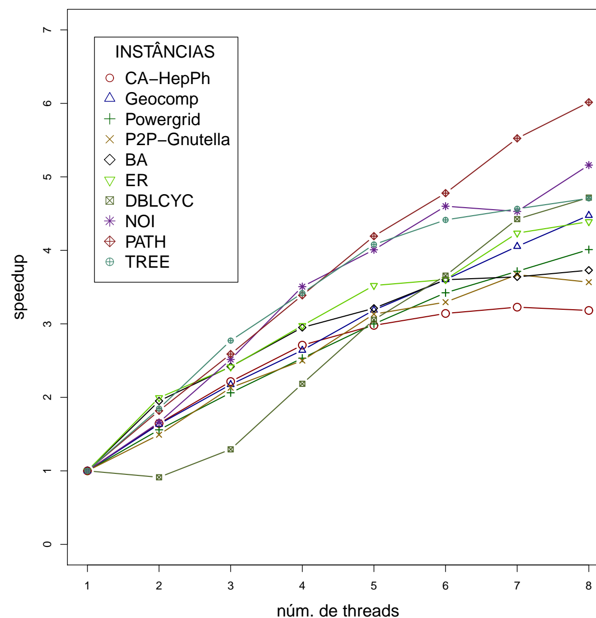


Figura 3.5: *Speedups* das execuções com *OpenMP*.

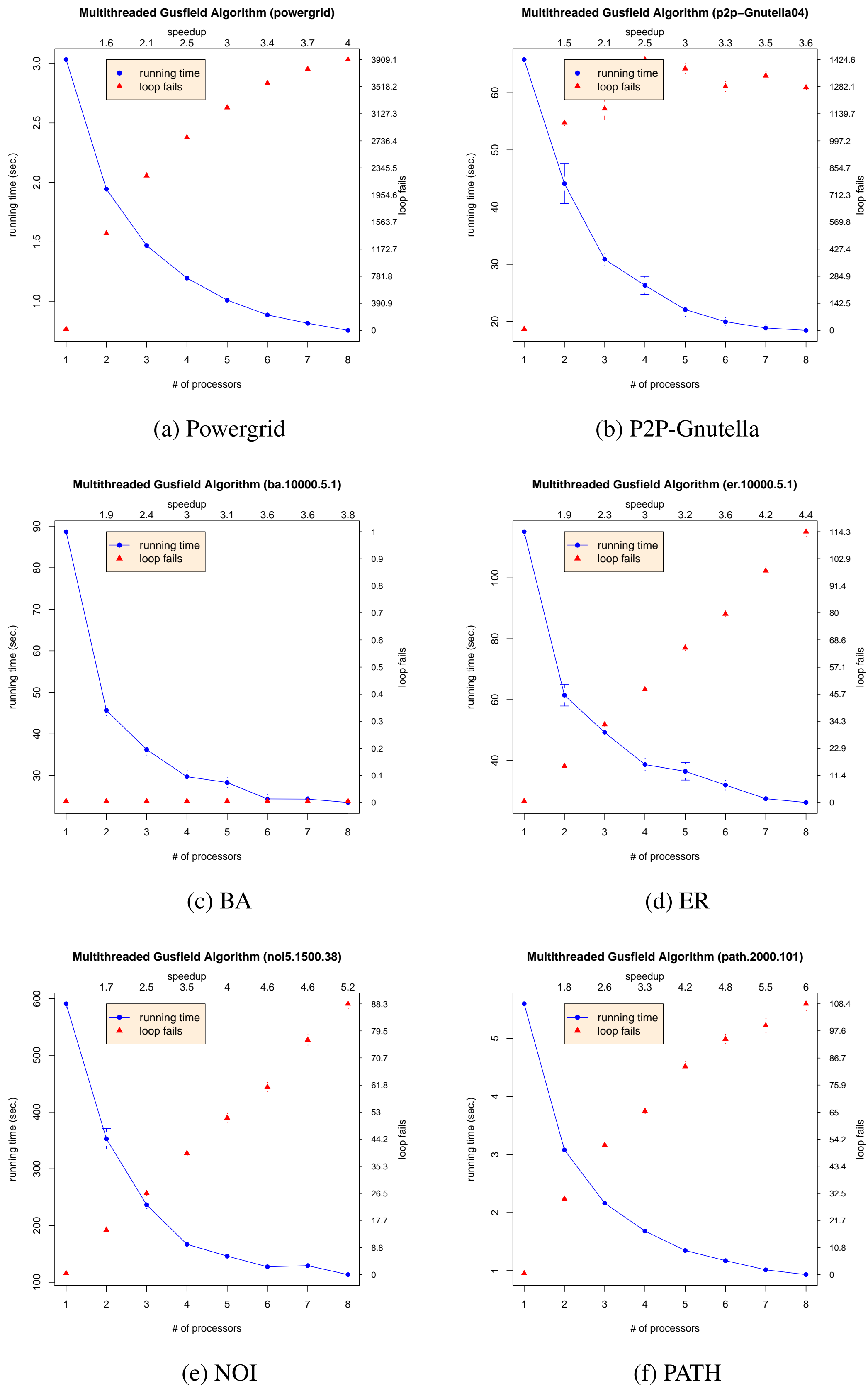
Outros bons resultados foram obtidos em experimentos em um computador com 16 núcleos. O melhor *speedup* foi de 9,4 utilizando 16 *threads* no grafo NOI5. Um experimento com o grafo ER com 3000 vértices e grau médio 5 alcançou um *speedup* de 9,2 utilizando 10 *threads*.

A implementação com *OpenMP* em arquitetura com memória compartilhada não é tão escalável quanto a implementação com *MPI* em *clusters*. A memória compartilhada é um gargalo na execução com *OpenMP*. Além disso, como cada *thread* necessita uma cópia do grafo, a utilização de memória RAM se torna excessiva nos grafos maiores com um número elevado de *threads*.

Assim como ocorre na implementação com *MPI*, heurísticas podem reduzir o número de laços falhos e melhorar o desempenho do algoritmo em alguns casos. A seguir, os resultados obtidos com a utilização de duas heurísticas são apresentados.

Tabela 3.3: Resultados da implementação do algoritmo de Gusfield com *OpenMP*. A tabela mostra as médias dos tempos de execução em segundos, dos *speedups* (S) e da eficiência (E) para cada instância.

Number of threads	**CA-HepPh**			**Geocomp**			**Powergrid**			**P2P-Gnutella**			**BA**		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	517.07	-	-	2.10	-	-	3.03	-	-	65.79	-	-	87.66	-	-
2	314.32	1.65	0.82	1.29	1.63	0.82	1.94	1.56	0.78	44.10	1.49	0.75	44.94	1.95	0.98
3	233.42	2.22	0.74	0.97	2.17	0.72	1.47	2.06	0.69	30.87	2.13	0.71	36.24	2.42	0.81
4	190.77	2.71	0.68	0.80	2.64	0.66	1.20	2.53	0.63	26.31	2.50	0.63	29.71	2.95	0.74
5	173.56	2.98	0.60	0.66	3.19	0.64	1.01	3.00	0.60	21.00	3.13	0.63	27.29	3.21	0.64
6	164.59	3.14	0.52	0.58	3.60	0.60	0.88	3.42	0.57	19.96	3.30	0.55	24.35	3.60	0.60
7	160.22	3.23	0.46	0.52	4.05	0.58	0.82	3.72	0.53	17.93	3.67	0.52	24.08	3.64	0.52
8	162.48	3.18	0.40	0.47	4.48	0.56	0.76	4.01	0.50	18.44	3.57	0.45	23.50	3.73	0.47
Number of threads	**DBLCYC**			**ER**			**NOI**			**PATH**			**TREE**		
	time	S	E	time	S	E	time	S	E	time	S	E	time	S	E
1	10.12	-	-	115.16	-	-	585.37	-	-	5.60	-	-	315.09	-	-
2	11.09	0.91	0.46	57.71	2.00	1.00	352.84	1.66	0.83	3.08	1.82	0.91	170.51	1.85	0.92
3	7.83	1.29	0.43	47.71	2.41	0.80	232.72	2.52	0.84	2.16	2.59	0.86	113.66	2.77	0.92
4	4.63	2.18	0.55	38.69	2.98	0.74	166.95	3.51	0.88	1.65	3.39	0.85	92.18	3.42	0.85
5	3.31	3.06	0.61	32.70	3.52	0.70	146.06	4.01	0.80	1.33	4.19	0.84	77.23	4.08	0.82
6	2.77	3.66	0.61	31.96	3.60	0.60	127.23	4.60	0.77	1.17	4.78	0.80	71.38	4.41	0.74
7	2.29	4.43	0.63	27.19	4.23	0.60	129.23	4.53	0.65	1.01	5.52	0.79	69.02	4.57	0.65
8	2.14	4.72	0.59	26.23	4.39	0.55	113.44	5.16	0.65	0.93	6.01	0.75	66.93	4.71	0.59

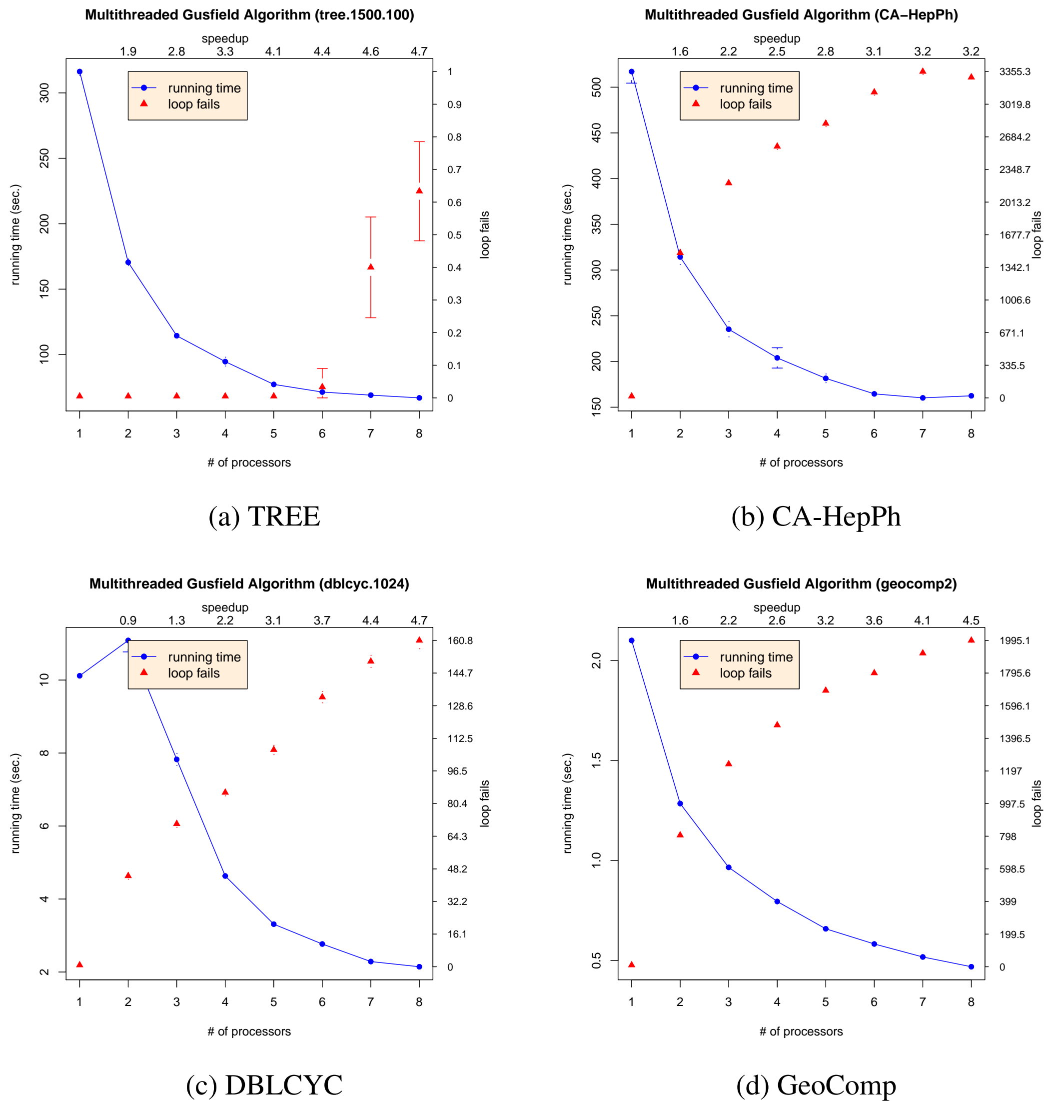


(a) Powergrid (b) P2P-Gnutella

(c) BA (d) ER

(e) NOI (f) PATH

Figura 3.6: Tempos de execução e número de laços falhos em execuções do algoritmo de Gusfield com *OpenMP*. Os pontos representam as médias dos tempos de execução e os triângulos representam o número de laços falhos. Os *speedups* médios são mostrados no topo de gráfico. Continua na Figura 3.7.



(a) TREE

(b) CA-HepPh

(c) DBLCYC

(d) GeoComp

Figura 3.7: Continuação da Figura 3.6. Tempos de execução e número de laços falhos em execuções do algoritmo de Gusfield com *OpenMP*. Os pontos representam as médias dos tempos de execução e os triângulos representam o número de laços falhos. Os *speedups* médios são mostrados no topo de gráfico.

Tabela 3.4: Número de laços falhos com e sem as heurísticas da implementação em *MPI* executada em 15 máquinas e da implementação com *OpenMP* executada com 8 *threads*.

	MPI		OpenMP	
Instância	sem heurística	com heurísticas	sem heurística	com heurísticas
powergrid	3887.0	306.4	3296.3	3.3
CA-HepPh	2907.2	29.0	3909.1	32.7
geocomp2	1997.6	44.8	1995.0	14.9
p2p-Gnutella04	1279.2	0.0	1277.8	3.6
dblcyc.1024	317.4	14.0	160.8	15.0
er.10000.5.1	182.4	0.0	114.3	0.0
noi5.1500.38	170.2	38.8	88.3	12.6
path.2000.101	112.2	42.4	108.4	26.2
tree.1500.100	1.2	0.6	0.6	2.3
ba.10000.5.1	0.0	0.0	0.0	0.0

Heurísticas para Redução do Número de Laços Falhos Duas heurísticas foram implementadas na tentativa de reduzir o número de laços falhos. A primeira consiste em verificar se o corte trivial formado pelo vértice de origem é um corte mínimo. A escolha desse corte implica que a separação do vértice e de seu vizinho não produz modificações na árvore em construção e, portanto, não interfere com o trabalho paralelo dos outros processos.

A segunda heurística consiste em ordenar os vértices de forma a maximizar as chances dos cortes encontrados terem margens pequenas do lado da origem. A ordenação escolhida foi a decrescente de graus dos vértices, pois ela implica que se o s-t-corte mínimo é trivial, então $\{\{s\}, V - \{s\}\}$ é um s-t-corte mínimo cuja utilização não modifica a árvore em construção.

A Tabela 3.4 mostra a redução do número de falhas nas execuções da implementação usando *OpenMP* com 8 *threads* e da implementação usando *MPI* em 16 processos em todas as 10 instâncias do conjunto de testes. Pode-se verificar que as duas heurísticas apresentadas foram suficientes para reduzir o número de laços falhos a valores muito baixos. A Figura 3.8 mostra os *speedups* das execuções com e sem as heurísticas da implementação em *MPI* com 16 processos.

Speedups com e sem as heurísticas.

Instância	sem heurística	heurística 1	heurísticas 1 & 2
powergrid	2.1	4.0	6.1
CA-HepPh	9.9	10.5	12.1
geocomp2	3.2	4.3	5.6
p2p-Gnutella04	9.3	9.7	11.2
dblcyc.1024	9.9	12.0	12.8
er.10000.5.1	9.6	9.8	9.8
noi5.1500.38	5.8	5.9	8.5
path.2000.101	6.8	7.1	7.9
tree.1500.100	7.0	7.1	7.1
ba.10000.5.1	15.2	15.5	15.3

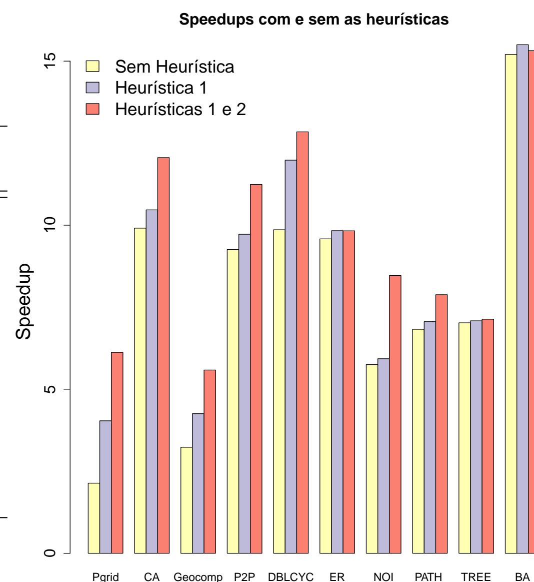


Figura 3.8: *Speedups* das heurísticas da implementação em *MPI* executada em 15 máquinas (16 processos). A heurística 1 é a detecção de cortes triviais formados pelos vértices de origem e a heurística 2 é a ordenação dos vértices em ordem não crescente de grau.

4 Implementações Paralelas do Algoritmo de Gomory-Hu para Árvores de Cortes

A implementação paralela do algoritmo de Gusfield utilizando *MPI* descrita no Capítulo 3 é baseadas no modelo mestre/escravo. O processo mestre realiza as atualizações da árvore em construção e gera instâncias do problema do s-t-corte mínimo. Os processos escravos recebem instâncias do problema do s-t-corte mínimo, calculam uma resposta e enviam-na de volta ao processo mestre. A estratégia é a de melhor esforço, uma vez que processos escravos calculam os cortes mínimos sem sincronismo, apesar da possibilidade de alguns cortes poderem invalidar outros. Com base nos resultados experimentais, foi possível concluir que apesar de haver cortes descartados, os *speedups* alcançados pela solução são bons.

Neste capítulo, é descrita uma implementação paralela do algoritmo de Gomory-Hu para árvores de cortes também baseada no modelo mestre/escravo, na qual os processos escravos resolvem problemas de corte mínimo de maneira assíncrona. A Seção 4.1 descreve essa solução paralela e a Seção 4.2 apresenta os resultados experimentais da solução.

Na Seção 4.3 é proposta uma solução híbrida entre os algoritmos Gus e GH e são apresentados os resultados experimentais dessa solução. O objetivo dessa implementação é ter um desempenho mais robusto, que não dependa tanto das características de cada instância.

A Seção 4.4 descreve em detalhes heurísticas implementadas para a melhoria do desempenho do algoritmo GH. A Seção 4.4.1 descreve um algoritmo de enumeração dos s-t-cortes mínimos que permite a busca de cortes mais balanceados. Os algoritmos utilizados para a escolha do nodo pivô e do par de vértices s e t são descritos na Seção 4.4.2. Outras heurísticas são descritas na Seção 4.4.3. A conclusão do capítulo está na Seção 4.6.

4.1 Uma Versão Paralela do Algoritmo de Gomory-Hu

A implementação do algoritmo GH é bem mais complexa do que a do algoritmo de Gusfield. Em primeiro lugar, a árvore em construção deve representar uma partição dos vértices

do grafo de entrada através da associação de um conjunto de vértices para cada nodo da árvore. Uma estratégia deve ser definida para a escolha do nodo de onde sairá o próximo par de vértices a ser separado. Chamamos esse nodo de *pivô*. A escolha do par de vértices s e t também afeta o desempenho do algoritmo e uma heurística para essa escolha deve ser implementada. Cada subproblema define uma partição dos vértices que determina como o grafo deve ser contraído. O processo escravo recebe uma instância do problema do corte mínimo e a partição dos vértices do grafo, bem como a lista dos vértices que não devem ser contraídos. Então, o grafo contraído é construído e o algoritmo do corte mínimo é executado.

A seguir o algoritmo é descrito em detalhes e na Seção 4.5 é descrita a implementação. Lembre-se que o termo *nodo* sempre se refere à árvore de cortes em construção e o termo *vértice* se refere ao grafo de entrada ou ao grafo contraído, dependendo do contexto.

Na implementação paralela do algoritmo GH, cada processo mantém uma cópia do grafo de entrada. O processo mestre cria tarefas e envia aos processos escravos. Cada tarefa descreve uma partição de V_G e os dois vértices s e t a serem separados por um corte mínimo. Essa informação é utilizada pelo processo escravo na construção do grafo contraído e no cálculo de um corte mínimo que separa s e t.

Quando o processo escravo termina a execução do algoritmo de corte mínimo, ele envia a resposta ao processo mestre, contendo o corte mínimo, o valor do corte e outras informações estatísticas como os tempos de execução, o tamanho do grafo contraído, um fator de balanceamento do corte e o número de cortes enumerados pela heurística que procura por cortes balanceados.

Ao receber as informações sobre um corte mínimo, o processo mestre decide se pode utilizá-lo para uma atualização da árvore em construção. Isso se dá quando o par de vértices que foi separado ainda está contido no mesmo nodo da árvore. Quando o par de vértices já tiver sido separado, então o corte deve ser descartado. O algoritmo termina quando cada nodo da árvore contém um único vértice.

O Algoritmo 4.1 apresenta o pseudocódigo da versão paralela do algoritmo de Gomory-Hu. O algoritmo recebe um grafo capacitado $G = (V_G, E_G, c_G)$ e devolve uma árvore capacitada $T = (V_T, E_T, c_T)$. Dada uma aresta $e = \{u, v\} \in E_T$, denota-se por $e.v$ o vértice de V_G que foi origem, s, ou destino, t, do corte que originou a aresta e e que ficou do mesmo lado desse corte que os vértices em v. Exemplo: a Figura 4.1 apresenta o primeiro passo do algoritmo GH. Os vértices 1 e 2 foram separados e a árvore passou a ter 2 nodos que *são* os conjuntos de vértices $u = \{1, 3\}$ e $v = \{2, 4, 5, 6\}$. A aresta $e = \{u, v\}$ foi criada com peso 15. Nesse caso, $e.u = 1$ e $e.v = 2$, que são os vértices utilizados nas definições de s e de t da instância do s-t-corte mínimo

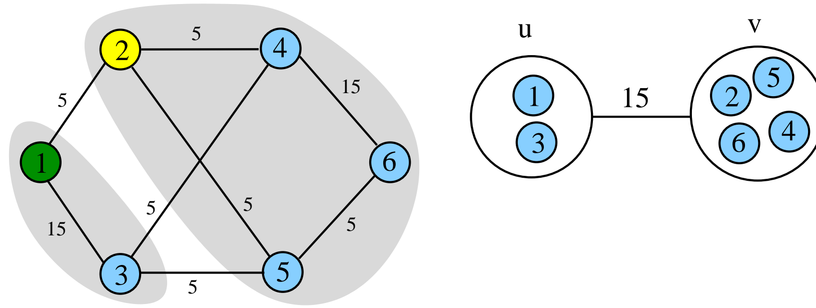


Figura 4.1: Primeiro passo do algoritmo GH. A árvore contém os 2 nodos $u = \{1,3\}$ e $v = \{2,4,5,6\}$. Os vértices associados à aresta $e = \{u,v\}$ são $e.u = 1$ e $e.v = 2$ que foram utilizados no cômputo do corte que originou a aresta e.

que originou a aresta e. Ao longo da execução do algoritmo, os valores de $e.u$ e $e.v$ permanecem os mesmos que foram atribuídos no momento da criação da aresta.

Os processos são identificados por $proc_0, proc_1, .., proc_{p-1}$. O processo mestre é o $proc_0$ e os demais são os processos escravos. Cada processo mantém uma cópia estática do grafo de entrada $G = (V_G, E_G, c_G)$. Inicialmente, o processo mestre cria uma tarefa para cada processo escravo e as envia sequencialmente (linha 6). No laço principal, o processo mestre aguarda uma resposta de um processo escravo contendo um par s, t e um s-t-corte mínimo $\{X, \overline{X}\}$. Se os vértices s e t ainda estiverem no mesmo nodo da árvore, então a árvore é atualizada (linha 9). O nodo contendo s e t é denotado no pseudocódigo por *pivô*. A atualização da árvore em construção consiste em:

1. criar um nodo *novo* contendo os vértices do conjunto $X \cap pivô$;

2. remover de *pivô* os vértices em $X \cap pivô$;

3. retirar *pivô* da lista de candidatos a pivô, caso $|pivô| = 1$;

4. inserir o nodo *novo* na lista de candidatos a pivô, caso $|novo| > 1$;

5. para cada aresta incidente ao *pivô*, $e = \{v, pivô\}$, remover e de E_T e inserir $\{v, nodo\}$ em E_T, se $e.v \in X$; ,

6. adicionar a aresta $\{pivô, novo\}$ em E_T, com capacidade igual à capacidade do corte $\{X, \overline{X}\}$.

A operação de número 5 define como as arestas da árvore em construção são atualizadas. Essa atualização difere da maneira como o algoritmo sequencial a faz. Uma mudança é necessária porque nas execuções paralelas pode ocorrer que um nodo v incidente a *pivô* fez parte do nodo pivô quando a tarefa foi criada. Dessa forma, existe a possibilidade do s-t-corte

cruzar o nodo v ou outros nodos de sua subárvore. Veja a Figura 4.2. As linhas cheias mostram a árvore atual e a elipse pontilhada representa o nodo pivô no momento em que os vértices s e t foram *escolhidos* para serem separados. O corte $\{X_e, \overline{X_e}\}$ separou os vértices x e y, dividindo o nodo pivô em dois com a criação do vértice v e da aresta $\{piv\hat{o}, v\}$. O s-t-corte $\{X, \overline{X}\}$ cruza o nodo v. O algoritmo GH paralelo conecta a aresta e de acordo com o lado do corte em que $y = e.v$ se encontra: se $y \in X$ então a aresta e é conectada ao nodo $novo = piv\hat{o} \cap X$ que contém s. Do contrário, a aresta permanece conectada ao nodo *pivô* que contém t. A validade dessa estratégia de atualização da árvore será abordada logo após a conclusão da descrição do pseudocódigo.

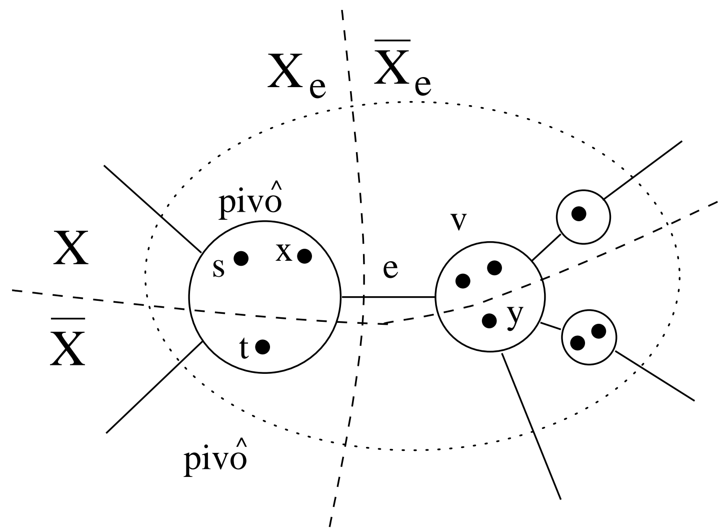
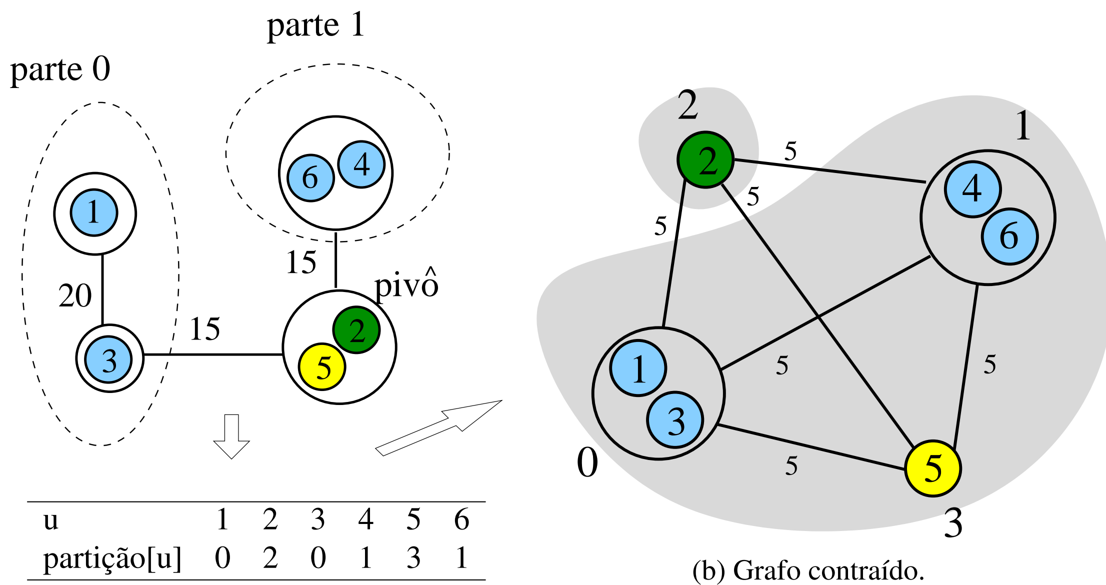


Figura 4.2: Exemplo de cruzamento entre o s-t-corte mínimo e um nodo vizinho do pivô.

Para criar uma tarefa, o processo mestre chama o procedimento `escolhe_pivô` para obter o próximo pivô de onde o par s e t deve ser escolhido. Depois, é feita a chamada ao procedimento `constrói_partição` que devolve um vetor que: (i) associa a cada vértice de $V_G \backslash piv\hat{o}$ o identificador de uma das partes de uma partição de $V_G \backslash piv\hat{o}$. Essa partição é formada pelos subconjuntos de vértices contidos nos nodos de cada árvore de $T - piv\hat{o}$. (ii) Cada vértice de *pivô* recebe um número sequencial único; esses são os vértices que não sofrerão contração. Esse vetor e o grafo de entrada contêm informações suficientes para a construção do grafo contraído, cujos identificadores dos vértices obedecem a essa numeração. Esse vetor é enviado a um processo escravo. Veja um exemplo na Figura 4.3a. Os vértices 1 e 3 são mapeados em 0, os vértices 4 e 6 são mapeados em 1. Os vértices 2 e 5, contidos no nodo pivô, recebem números sequenciais únicos maiores do que 1, ou seja, 2 e 3, respectivamente. O grafo contraído construído com base nessa numeração aparece na Figura 4.3b (o grafo de entrada que originou esse grafo é o mesmo da Figura 2.3).

O procedimento `escolhe_par_st` é chamado para fazer a escolha do par de vértices



u	1	2	3	4	5	6
partição[u]	0	2	0	1	3	1

(b) Grafo contraído.

(a) Exemplo de partição e de numeração dos vértices.

Figura 4.3: Resultado do procedimento `constrói_partição` que numera os vértices de acordo com a partição induzida pelas árvores de $T - piv\hat{o}$. Os vértices em *pivô* são numerados sequencialmente. À direita está o grafo contraído construído com base na partição apresentada.

de *pivô* que deve ser separado por um corte mínimo. Finalmente, a nova tarefa é enviada para o processo escravo que enviou a resposta sendo processada e que, a essa altura, está ocioso. Quando a árvore contiver $|V_G|$ nodos, o processo mestre envia mensagens de finalização para todos os processos escravos (linha 22).

Ao receber uma tarefa, o processo escravo constrói o grafo contraído, calcula um s-t-corte mínimo nesse grafo e envia o resultado ao processo mestre (linha 36). A construção do grafo contraído pelo processo escravo consiste em percorrer a lista de arestas do grafo de entrada e, para cada uma delas, determinar se existe uma aresta correspondente no grafo contraído, como mostra o Algoritmo 4.2. Se a aresta conecta vértices de uma mesma árvore de $T - piv\hat{o}$, então nenhuma aresta é adiciona ao grafo contraído. Por outro lado, arestas entre vértices de componentes diferentes de $T - piv\hat{o}$ ou arestas que contenham pelo menos um vértice contido no pivô produzem uma aresta nova no grafo contraído ou o aumento da capacidade de uma aresta já existente. Como o processo escravo não tem acesso à árvore T, o vetor *partição* é utilizado para determinar em qual subárvore está o vértice ou se ele está em *pivô*.

Como discutido anteriormente, a versão paralela do algoritmo GH difere da versão sequencial na forma de determinar como cada nodo vizinho do pivô é reconectado durante a atualização da árvore. Na versão sequencial, os vértices contidos em cada subárvore de $T - piv\hat{o}$ ficam necessariamente em um dos lados do s-t-corte mínimo. Porém, na versão paralela, o nodo

Algoritmo 4.1 Versão paralela do Algoritmo de Gomory-Hu para Árvore de Cortes.

Entrada: $G = (V_G, E_G, c_G)$, $proc_j$, $0 \leq j < P$, processos
Saída: $T = (V_T, E_T, c_T)$ uma árvore de cortes de G

1: **se** $proc_j = 0$ **então** // processo mestre
2: **para** $s \leftarrow 1$ **to** $P - 1$ **faça**
3: *pivô* $\leftarrow$ `escolhe_pivô`(T)
4: *partição* $\leftarrow$ `constrói_partição`(*pivô*, T)
5: $(s,t) \leftarrow$ `escolhe_par_st`(*pivô*)
6: **envie** tarefa $(s, t,$ *partição*$)$ para $proc_s$
7: **laço**
8: **receba** de $proc_j$ resposta (s, t, X), onde $\{X, \overline{X}\}$ é um s-t-corte mínimo de G
9: **se** s e t pertencem ao mesmo nodo *pivô* de V_T **então** // atualiza árvore
10: *novo* $\leftarrow$ *pivô* $\cap X$
11: *pivô* $\leftarrow$ *pivô* $\backslash X$
12: $V_T \leftarrow V_T \cup \{novo\}$
13: **se** $|pivô| = 1$ **então**
14: retire *pivô* da lista de candidatos a pivô
15: **se** $|novo| > 1$ **então**
16: insira *novo* na lista de candidatos a pivô
17: **para toda aresta** $e = \{pivô, v\} \in E_T$ **faça**
18: **se** $e.v \in X$ **então** // reconecta a aresta em *novo*
19: $E_T \leftarrow (E_T \backslash \{e\}) \cup \{\{novo, v\}\}$
20: adicione a aresta $\{pivô, novo\}$ em E_T com a capacidade de $\{X, \overline{X}\}$
21: **se** $|V_T| = |V_G|$ **então**
22: **envie** mensagem de finalização para todos os processos escravos
23: **imprima** a árvore T
24: **termine**
25: *pivô* $\leftarrow$ `escolhe_pivô`(T)
26: *partição* $\leftarrow$ `constrói_partição`(*pivô*, T)
27: $(s,t) \leftarrow$ `escolhe_par_st`(*pivô*)
28: **envie** tarefa $(s, t,$ *partição*$)$ para $proc_j$
29: **senão** // processo escravo
30: **laço**
31: **receba** tarefa $(s, t,$ *partição*$)$
32: **se** `Tarefa` = fim **então**
33: **termine**
34: $G_c \leftarrow$ `constrói_grafo_contraído`(G, *partição*)
35: $X \leftarrow$ `corte_mínimo`(G_c, s, t)
36: **envie** resposta (s, t, X) para $proc_0$

Algoritmo 4.2 `constrói_grafo_contraído`(G, *partição*)

Entrada: $G = (V_G, E_G, c_G)$, grafo de entrada e *partição*
Saída: $G_c = (V_c, E_c, c_c)$, grafo contraído

1: **para cada** $\{u, v\} \in E_G$ **faça**
2: **se** *partição*$[u] \neq$ *partição*$[v]$ **então**
3: `adiciona_aresta` (*partição*$[u]$, *partição*$[v]$, $c_G[\{u, v\}]$, G_c)

pivô pode ter sido dividido uma ou mais vezes no momento em que um corte que separa vértices desse nodo é recebido pelo processo mestre. A Figura 4.2 descrita anteriormente traz uma representação dessa situação. O vértice $e.v$ é o vértice y, que pode estar em v ou em outro nodo da mesma subárvore de $T - piv\hat{o}$. Como explicado anteriormente, o algoritmo GH paralelo conecta a aresta e de acordo com o lado do corte em que $y = e.v$ se encontra. O seguinte lema mostra que o Algoritmo 4.1 atualiza a árvore de cortes corretamente.

Lema 4.1. *O Algoritmo 4.1 atualiza a árvore de cortes corretamente.*

Demonstração. Seja $G = (V_G, E_G, c_G)$ o grafo capacitado de entrada. Seja $T = (V_T, E_T, c_T)$ a árvore em construção no momento em que um s-t-corte é recebido pelo processo mestre. Mostraremos que a atualização da árvore feita pelo Algoritmo 4.1 é equivalente a uma atualização feita pelo algoritmo sequencial na mesma situação. Note que cada corte do grafo contraído corresponde um corte do grafo de entrada com a mesma capacidade. Podemos, então, definir todos os cortes como conjuntos de vértices do grafo de entrada para facilitar a notação.

Como na Figura 4.2, seja $piv\hat{o} \in V_T$ tal que $s, t \in piv\hat{o}$. Seja $e = \{piv\hat{o}, v\} \in E_T$ uma aresta incidente a $piv\hat{o}$. Seja $X \subseteq V_G$, $s \in X$, tal que $\{X, \overline{X}\}$ é o s-t-corte mínimo recebido pelo processo mestre. Seja $X_e \subseteq V_G$, $s, t, x \in X_e$, o x-y-corte mínimo associado à aresta e. Suponha que os cortes $\{X, \overline{X}\}$ e $\{X_e, \overline{X_e}\}$ se cruzam, isto é, $X \cap \overline{X_e} \neq \emptyset$ e $\overline{X} \cap \overline{X_e} \neq \emptyset$ (note que, por construção, $s \in X \cap X_e$ e $t \in \overline{X} \cap X_e$).

Pelo Lema 2.2 apresentado na Seção 2.2, quando $y \in \overline{X}$, o conjunto $S = X \cap X_e$ define um s-t-corte mínimo que não cruza o corte X_e, independentemente de x pertencer a X ou a $\overline{X}$. O corte $\{S, \overline{S}\}$ é consistente com o algoritmo GH sequencial (no que diz respeito à aresta e) e é tal que a subárvore de $T - piv\hat{o}$ que contém v está contida em $\overline{S}$. Portanto, o algoritmo sequencial atualizaria a árvore conectando o nodo v ao nodo formado por $piv\hat{o} \cap \overline{S}$. Como $e.v = y \in \overline{X}$, o algoritmo GH paralelo não executa a linha 19, mantendo v conectado ao pivô que se tornou o conjunto $piv\hat{o} \cap \overline{X}$ que é o mesmo que $piv\hat{o} \cap \overline{S}$.

Por outro lado, se $y \in X$, então o Lema 2.2 implica que $T = \overline{X} \cap X_e$ é um s-t-corte mínimo. Utilizando esse corte, o algoritmo sequencial ligaria v ao nodo $piv\hat{o} \cap \overline{T}$. Por sua vez, o algoritmo paralelo, na linha 19, conecta v a $piv\hat{o} \cap X$ que é o mesmo que $piv\hat{o} \cap \overline{T}$.

No caso do corte $\{X, \overline{X}\}$ cruzar várias subárvores de $T - piv\hat{o}$, o mesmo procedimento de descruzamento dos cortes utilizado acima deve ser sucessivamente aplicado a cada subárvore. O corte resultante não irá cruzar nenhuma subárvore, pois um descruzamento de conjuntos não desfaz os descruzamentos anteriores por tratar-se de operações de intersecção de conjuntos. $\square$

A próxima seção apresenta os resultados experimentais com o algoritmo GH paralelo.

4.2 Resultados Experimentais: Gomory-Hu Paralelo

A versão paralela do algoritmo GH foi implementada com o objetivo de verificar se ela pode alcançar bons *speedups* e determinar se ela pode superar a implementação paralela do algoritmo Gus em algumas instâncias, como é o caso da versão sequencial.

Foram executados experimentos em um conjunto de 10 instâncias variadas e também em uma família de grafos (NOI) que contém um número parametrizável de *clusters*. Essa variação no número de *clusters* permite a observação da variação de desempenho do algoritmo GH à medida que mais cortes balanceados são encontrados pelo algoritmo.

Informalmente, um corte $\{X, \overline{X}\}$ é desbalanceado se X ou $\overline{X}$ é pequeno. Essa noção tem um papel importante no desempenho do algoritmo paralelo de Gusfield: quanto mais desbalanceado for um corte utilizado ao longo do algoritmo, menos provável é que ele interfira com outros cortes sendo computados concorrentemente. No caso do algoritmo GH, o balanceamento dos cortes afeta o desempenho do algoritmo de duas maneiras opostas. Por um lado, os cortes balanceados permitem que o tamanho do grafo seja reduzido, que é uma condição necessária para que o algoritmo GH supere o algoritmo de Gus. Por outro lado, cortes balanceados podem invalidar outros cortes que estejam sendo computados de forma concorrente com maior probabilidade. Certamente, a redução do tamanho do grafo tem um potencial muito maior de melhorar o desempenho do algoritmo do que o descarte de cortes tem de piorar. Assim, as implementações do algoritmo GH devem buscar encontrar cortes balanceados quando possível.

Além do algoritmo descrito acima, diversas heurísticas foram implementadas para melhorar o desempenho do código. As mais importantes dentre elas são a enumeração dos s-t-cortes mínimos para a busca de um corte mais balanceado e a enumeração de pares de vértices disjuntos. As heurísticas são descritas em detalhes na Seção 4.5.

Ambiente Computacional Os experimentos foram executados em um cluster com 12 computadores com processadores Intel Core 2 Quad 2.4 GHz, 2 GB de memória e 4096 KB de cache, interconectados por uma rede Ethernet de 1Gbps[1]. O código escrito em C++ foi compilado com o *gcc* com nível de otimização `O3`.

[1] O *cluster* utilizado nos experimentos é o do Laboratório Central de Processamento de Alto Desempenho (LCPAD) da UFPR que é financiado pela FINEP através dos projetos CT-INFRA/UFPR.

Tabela 4.1: Primeiro conjunto de instâncias e seus tamanhos.

Grafo	$\|V\|$	$\|E\|$
ROME99	3.353	8.879
GEOCOMP	3.621	9.461
POWERGRID	4.941	6.594
POLBLOGS	1.222	16.714
BA	10.000	49.995
ER	10.000	9.995
DCYC	1.024	2.048
NOI	1.000	99.900
PATH	2.000	21.990
TREE	2.000	21.990

4.2.1 Instâncias Variadas

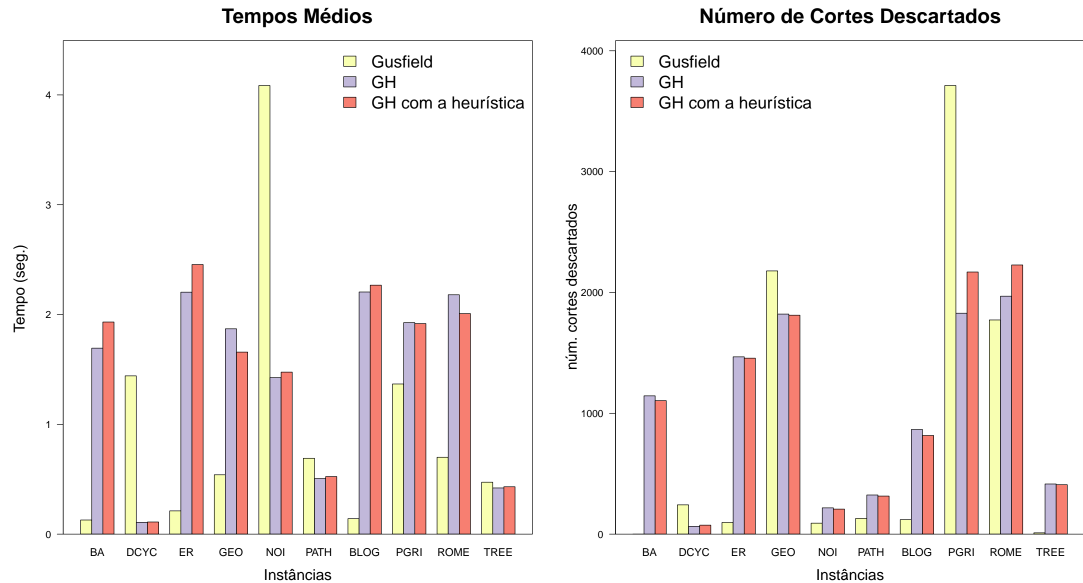
O primeiro conjunto de instâncias é formado por 10 grafos cujos tamanhos são mostrados na Tabela 4.1. Os primeiros 4 grafos foram obtidos de dados reais: uma rede de ruas da cidade de Roma (1999) [99], uma rede de colaboração científica [9], uma rede de transmissão de eletricidade [105] e uma rede de *blogs* [1]. Duas redes, ER e BA, foram geradas por modelos aleatórios: o modelo Erdös-Rényi [12] e o modelo de ligações preferenciais [2], respectivamente. Os outros 4 grafos são sintéticos de diferentes tipos gerados por algoritmos que já foram usados em trabalhos experimentais de algoritmos de cortes mínimos e árvores de cortes [23, 44] que foram descritos no capítulo anterior.

Tempos de Execução A Figura 4.4a mostra os tempos de execução dos algoritmos Gus e GH com e sem a heurística dos cortes balanceados no primeiro conjunto de instâncias executados em 12 máquinas com 1 processo por máquina.

O experimento mostra que nenhum dos dois algoritmos domina o outro. Nos grafos DCYC e NOI a implementação do algoritmo GH foi mais rápida do que a implementação do algoritmo Gus. Essa relação mostra que a versão paralela do algoritmo GH preserva a vantagem em relação ao algoritmo de Gus em algumas instâncias, fato observado nas implementações sequenciais desses algoritmos. O algoritmo GH rodou 13 vezes mais rápido do que o algoritmo Gus na instância DCYC e 2,9 vezes mais rápido na instância NOI.

A heurística dos cortes balanceados melhorou marginalmente o desempenho nas instâncias PGRI, GEO e ROME. Essa heurística não melhora o desempenho na maioria das instâncias testadas porque ou elas não possuem cortes balanceados, como as instâncias ER e BA, ou os cortes balanceados existem mas tendem a ser únicos, como nas instâncias NOI, PATH e

TREE, pois as capacidades das arestas são geradas aleatoriamente em intervalos grandes. Assim, a heurística gasta tempo construindo uma estrutura de dados auxiliar que não melhora o desempenho do algoritmo.



(a) Tempo médio de execução com 12 processadores.

(b) Número médio de cortes descartados com 12 processadores.

Figura 4.4: Resultados do GH paralelo nas instâncias variadas.

A escolha entre os dois algoritmos pode ser feita tendo-se algum conhecimento prévio da estrutura da instância. Da mesma maneira, pode-se determinar se é ou não interessante o uso da heurística dos cortes balanceados. De qualquer maneira, podemos concluir que a implementação paralela do algoritmo GH pode superar aquela do algoritmo Gus em algumas instâncias quando cortes balanceados puderem ser encontrados.

Número de Cortes Descartados O número de cortes descartados mostra a eficácia da estratégia otimista na qual os processos escravos computam cortes mínimos de maneira assíncrona. O gráfico da Figura 4.4b mostra a média de cortes descartados pelos algoritmos em cada uma das instâncias.

A estrutura da instância, a estratégia de escolha dos vértices a serem separados e os cortes escolhidos afetam o número de cortes descartados. A implementação do algoritmo de GH procura minimizar esse número de algumas maneiras: (*i*) alterna o nodo pivô entre os nodos contendo mais de um vértice e (*ii*) em cada nodo pivô, escolhe pares disjuntos de vértices para serem separados, evitando repetição de vértices quando possível. A Seção 4.4.2 descreve o algoritmo de enumeração dos pares.

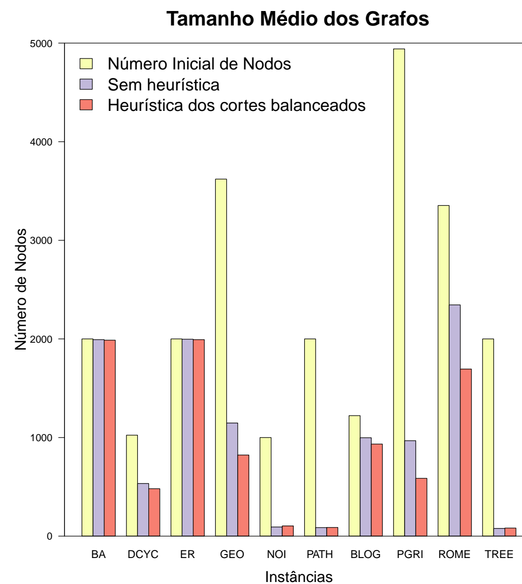


Figura 4.5: Tamanho médio dos grafos nas execuções do algoritmo GH com e sem a heurística dos cortes balanceados.

Em síntese, os resultados experimentais obtidos para o número de cortes descartados mostram que a implementação ainda pode ser otimizada nesse aspecto, pois para algumas instâncias esse número pode ser considerado alto.

Tamanho dos Grafos Contraídos O algoritmo GH reduz o tamanho do grafo ao longo de sua execução. A Figura 4.5 mostra o tamanho médio dos grafos durante a execução do algoritmo em cada instância. Resultados são apresentados para o algoritmo GH com e sem a heurística de enumeração de cortes. Nem sempre a redução do tamanho do grafo implica em tempos de execução menores porque a instância do problema do corte mínimo pode ser computacionalmente mais difícil apesar de menor. Além disso, existe um gasto computacional para a enumeração dos cortes e a construção do grafo contraído. Por isso, o fato do algoritmo GH conseguir reduzir significativamente o tamanho de alguns grafos, sem melhorar o tempo de execução, indica que outras otimizações podem melhorar ainda mais o desempenho do algoritmo.

Speedups Os *speedups* do algoritmo GH aparecem nos gráficos da Figura 4.7. Para comparação, os *speedups* do algoritmo de Gus são mostrados na Figura 4.6. As Tabelas 4.2 e 4.3 mostram os tempos de execução, os *speedups* e as eficiências das execuções dos algoritmos Gus e GH. Os melhores *speedups* alcançados pelo algoritmo Gus rodando em 12 máquinas foi de 9,42 e 8,85 nas instâncias NOI e TREE, respectivamente. O melhor *speedup* alcançado pelo algoritmo GH foi 8,08 na instância BA. Em geral, os *speedups* do algoritmo Gus são melhores

pois a sua versão paralela não tem grandes modificações em relação a sua versão sequencial. Por outro lado, algumas otimizações possíveis na implementação sequencial do algoritmo GH não foram reproduzidas na versão paralela.

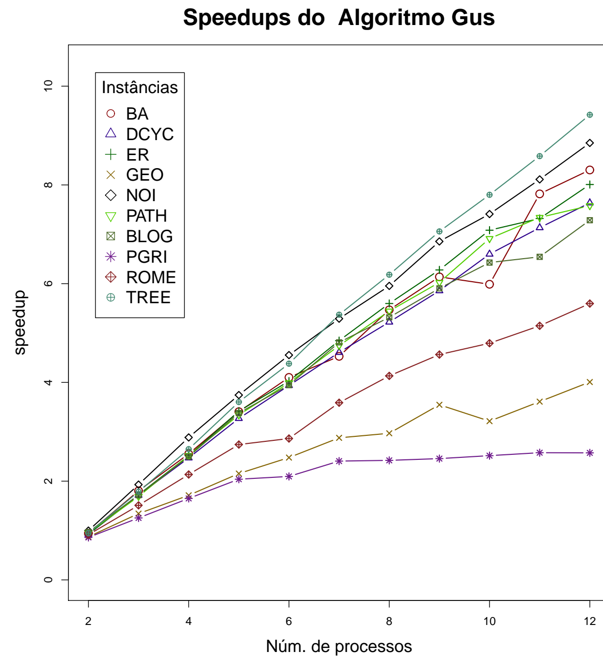
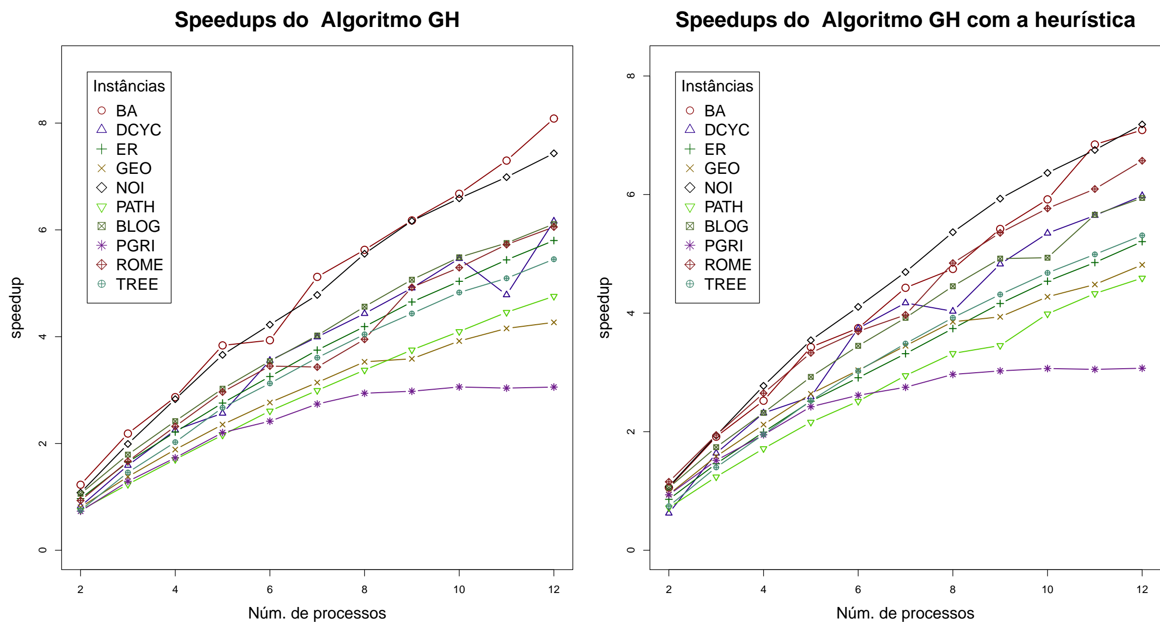


Figura 4.6: *Speedups* da implementação paralela do algoritmo Gus nas instâncias variadas.



(a) Sem a heurística dos cortes balanceados.

(b) Com a heurística dos cortes balanceados.

Figura 4.7: *Speedups* da implementação paralela do algoritmo GH sem e com a heurística dos cortes balanceados nas instâncias variadas.

Tabela 4.2: Comparação das implementações paralelas dos algoritmos Gus e GH nas instâncias variadas. A tabela mostra as médias dos tempos de execução em segundos, dos *speedups* (S) e da eficiência (E) para cada instância. Continua na próxima tabela.

Núm. de		**BA**			**DCYC**			**ER**			**GEO**			**NOI**		
procs.	**Alg.**	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E
sequencial	**ParGus**	1.07	-	-	11.01	-	-	1.70	-	-	2.17	-	-	36.16	-	-
	ParaGH	13.69	-	-	0.66	-	-	12.78	-	-	7.98	-	-	10.60	-	-
2	**ParGus**	1.13	0.95	0.47	11.91	0.92	0.46	1.83	0.93	0.46	2.48	0.87	0.44	36.22	1.00	0.50
	ParGH	11.16	1.23	0.61	0.80	0.82	0.41	13.21	0.97	0.48	9.92	0.80	0.40	9.83	1.08	0.54
3	**ParGus**	0.59	1.80	0.60	6.42	1.72	0.57	0.98	1.72	0.57	1.61	1.34	0.45	18.73	1.93	0.64
	ParGH	6.26	2.19	0.73	0.42	1.59	0.53	7.74	1.65	0.55	5.79	1.38	0.46	5.32	1.99	0.66
4	**ParGus**	0.42	2.55	0.64	4.46	2.47	0.62	0.67	2.52	0.63	1.26	1.71	0.43	12.53	2.89	0.72
	ParGH	4.77	2.87	0.72	0.29	2.25	0.56	5.77	2.21	0.55	4.23	1.89	0.47	3.75	2.83	0.71
5	**ParGus**	0.31	3.41	0.68	3.37	3.27	0.65	0.50	3.42	0.68	1.01	2.15	0.43	9.66	3.74	0.75
	ParGH	3.57	3.84	0.77	0.26	2.57	0.51	4.64	2.76	0.55	3.39	2.35	0.47	2.90	3.66	0.73
6	**ParGus**	0.26	4.10	0.68	2.80	3.94	0.66	0.42	4.03	0.67	0.87	2.48	0.41	7.94	4.55	0.76
	ParGH	3.48	3.93	0.66	0.19	3.55	0.59	3.93	3.25	0.54	2.88	2.77	0.46	2.51	4.22	0.70
7	**ParGus**	0.24	4.53	0.65	2.39	4.61	0.66	0.35	4.85	0.69	0.75	2.88	0.41	6.83	5.29	0.76
	ParGH	2.67	5.12	0.73	0.17	3.99	0.57	3.41	3.75	0.54	2.54	3.14	0.45	2.22	4.78	0.68
8	**ParGus**	0.20	5.47	0.68	2.11	5.22	0.65	0.30	5.60	0.70	0.73	2.97	0.37	6.07	5.95	0.74
	ParGH	2.43	5.63	0.70	0.15	4.43	0.55	3.05	4.19	0.52	2.26	3.53	0.44	1.91	5.55	0.69
9	**ParGus**	0.17	6.14	0.68	1.88	5.86	0.65	0.27	6.28	0.70	0.61	3.55	0.39	5.27	6.86	0.76
	ParGH	2.22	6.18	0.69	0.13	4.91	0.55	2.75	4.65	0.52	2.23	3.59	0.40	1.72	6.17	0.69
10	**ParGus**	0.18	5.99	0.60	1.67	6.60	0.66	0.24	7.08	0.71	0.67	3.22	0.32	4.88	7.41	0.74
	ParGH	2.05	6.68	0.67	0.12	5.46	0.55	2.54	5.04	0.50	2.04	3.92	0.39	1.61	6.59	0.66
11	**ParGus**	0.14	7.82	0.71	1.54	7.13	0.65	0.23	7.32	0.67	0.60	3.61	0.33	4.46	8.11	0.74
	ParGH	1.88	7.30	0.66	0.14	4.78	0.43	2.35	5.44	0.49	1.92	4.15	0.38	1.52	6.99	0.64
12	**ParGus**	0.13	8.30	0.69	1.44	7.64	0.64	0.21	8.01	0.67	0.54	4.01	0.33	4.09	8.85	0.74
	ParGH	1.69	8.09	0.67	0.11	6.16	0.51	2.20	5.80	0.48	1.87	4.27	0.36	1.43	7.43	0.62

Tabela 4.3: Continuação da tabela anterior. Comparação das implementações paralelas dos algoritmos Gus e GH nas instâncias variadas. A tabela mostra as médias dos tempos de execução em segundos, dos *speedups* (S) e da eficiência (E) para cada instância.

Núm. de procs.	**Alg.**	**PATH**			**BLOG**			**PGRI**			**ROME**			**TREE**		
		tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E	tempo	S	E
sequencial	**ParGus**	5.24	-	-	1.03	-	-	3.52	-	-	3.92	-	-	4.45	-	-
	ParaGH	2.41	-	-	13.47	-	-	5.89	-	-	13.20	-	-	2.29	-	-
2	**ParGus**	5.50	0.95	0.48	1.06	0.97	0.48	4.09	0.86	0.43	4.28	0.91	0.46	4.71	0.95	0.47
	ParGH	3.22	0.75	0.37	12.74	1.06	0.53	8.05	0.73	0.37	14.17	0.93	0.47	3.05	0.75	0.38
3	**ParGus**	3.10	1.69	0.56	0.60	1.72	0.57	2.80	1.25	0.42	2.60	1.51	0.50	2.49	1.79	0.60
	ParGH	1.95	1.23	0.41	7.52	1.79	0.60	4.56	1.29	0.43	7.92	1.67	0.56	1.57	1.46	0.49
4	**ParGus**	2.08	2.52	0.63	0.41	2.50	0.62	2.13	1.65	0.41	1.84	2.13	0.53	1.68	2.65	0.66
	ParGH	1.42	1.70	0.43	5.57	2.42	0.60	3.40	1.73	0.43	5.69	2.32	0.58	1.13	2.02	0.51
5	**ParGus**	1.57	3.34	0.67	0.30	3.38	0.68	1.73	2.04	0.41	1.43	2.74	0.55	1.24	3.60	0.72
	ParGH	1.12	2.16	0.43	4.45	3.02	0.60	2.67	2.20	0.44	4.45	2.96	0.59	0.86	2.67	0.53
6	**ParGus**	1.31	3.99	0.67	0.26	3.95	0.66	1.68	2.10	0.35	1.37	2.86	0.48	1.02	4.38	0.73
	ParGH	0.92	2.61	0.43	3.81	3.54	0.59	2.44	2.42	0.40	3.83	3.45	0.57	0.73	3.13	0.52
7	**ParGus**	1.10	4.75	0.68	0.21	4.81	0.69	1.46	2.41	0.34	1.09	3.59	0.51	0.83	5.37	0.77
	ParGH	0.80	2.99	0.43	3.35	4.02	0.57	2.15	2.74	0.39	3.85	3.43	0.49	0.64	3.61	0.52
8	**ParGus**	0.96	5.44	0.68	0.19	5.32	0.66	1.45	2.42	0.30	0.95	4.13	0.52	0.72	6.18	0.77
	ParGH	0.71	3.37	0.42	2.95	4.56	0.57	2.00	2.94	0.37	3.34	3.95	0.49	0.57	4.04	0.51
9	**ParGus**	0.87	6.03	0.67	0.17	5.91	0.66	1.43	2.46	0.27	0.86	4.56	0.51	0.63	7.06	0.78
	ParGH	0.64	3.75	0.42	2.66	5.07	0.56	1.98	2.98	0.33	2.68	4.92	0.55	0.52	4.43	0.49
10	**ParGus**	0.76	6.91	0.69	0.16	6.43	0.64	1.40	2.52	0.25	0.82	4.79	0.48	0.57	7.80	0.78
	ParGH	0.59	4.10	0.41	2.46	5.49	0.55	1.93	3.06	0.31	2.49	5.29	0.53	0.47	4.83	0.48
11	**ParGus**	0.71	7.34	0.67	0.16	6.54	0.59	1.37	2.58	0.23	0.76	5.15	0.47	0.52	8.58	0.78
	ParGH	0.54	4.46	0.41	2.34	5.75	0.52	1.94	3.04	0.28	2.31	5.72	0.52	0.45	5.09	0.46
12	**ParGus**	0.69	7.58	0.63	0.14	7.29	0.61	1.37	2.57	0.21	0.70	5.60	0.47	0.47	9.42	0.79
	ParGH	0.51	4.76	0.40	2.21	6.11	0.51	1.93	3.06	0.25	2.18	6.05	0.50	0.42	5.45	0.45

4.2.2 Instâncias NOI

O segundo conjunto de instâncias é formado por uma família de grafos da classe NOI descritos anteriormente. Esses são grafos sintéticos gerados por um algoritmo cujo parâmetro k determina o número de *clusters* do grafo. A variação desse parâmetro permite a obtenção de grafos com cortes mínimos mais ou menos balanceados: valores intermediários de k produzem cortes mais balanceados. Com $k = 1$, os grafos gerados não possuem cortes balanceados. Geramos grafos com 1000 vértices e k igual a 1, 2, 3, 5, 10, 15, 20, 30, 40, 50, 100, 200, 300, 400 e 500. Os grafos têm densidade de arestas igual a 0,2. Essa família de grafos nos permitiu observar a variação de desempenho do algoritmo de GH à medida que ele encontra cortes balanceados que permitem a redução do tamanho do grafo.

A Figura 4.8 apresenta os tempos de execução dos algoritmos GH e Gus nas instâncias NOI. O algoritmo Gus alcançou tempos de execução mais baixos nas instâncias geradas com k igual a 1, 200, 300, 400 e 500. Nas demais 10 instâncias, com valores de k entre 2 e 100, o algoritmo GH foi mais rápido. O tempo total de execução do algoritmo GH chegou a ser 10,9 vezes menor do que o tempo do algoritmo Gus (instância com $k = 10$). Essas instâncias com valores de k intermediários possuem cortes mínimos balanceados que separam *clusters*. Esse desempenho mostra que a implementação do algoritmo GH foi capaz de encontrar os cortes balanceados e reduzir o tamanho do grafo eficazmente.

A Tabela 4.4 mostra estatísticas detalhadas das execuções. As colunas da tabela apresentam o número de *clusters* de cada instância, os acrônimos dos algoritmos, os tempos de inicialização, os tempos para atualização da árvore, os tempos para as contrações, os tempos para cálculo dos cortes mínimos nos grafos contraídos, os tempos para cálculos dos cortes mínimos no grafo de entrada, o tempo total (real), o número de cortes descartados e o número médio de vértices do grafo.

Podemos notar que o número médio de vértices do grafo contraído chegou a ser reduzido em 90% (para $k = 30$) pelo algoritmo GH. São apresentados resultados do experimento feito sem a heurística dos cortes balanceados, porque as instâncias NOI são ponderadas e os cortes tendem a ser únicos. Os tempos da versão do algoritmo GH com essa heurística são ligeiramente piores e essa diferença não altera a interpretação dos resultados.

O tempo de construção do grafo contraído é considerável e é um fator bastante importante no desempenho relativo dos dois algoritmos. Nota-se que nas instâncias com cortes balanceados, esses tempos foram bastante inferiores do que nas demais instâncias.

O número de cortes descartados foi pequeno em todas as execuções, o que facilitou

a comparação entre os dois algoritmos. Os tempos de inicialização são muito pequenos. Os tempos de atualização da árvore também são pequenos, o que mostra que o processo mestre pode suportar um número bem mais elevado de escravos antes de se tornar um gargalo[2].

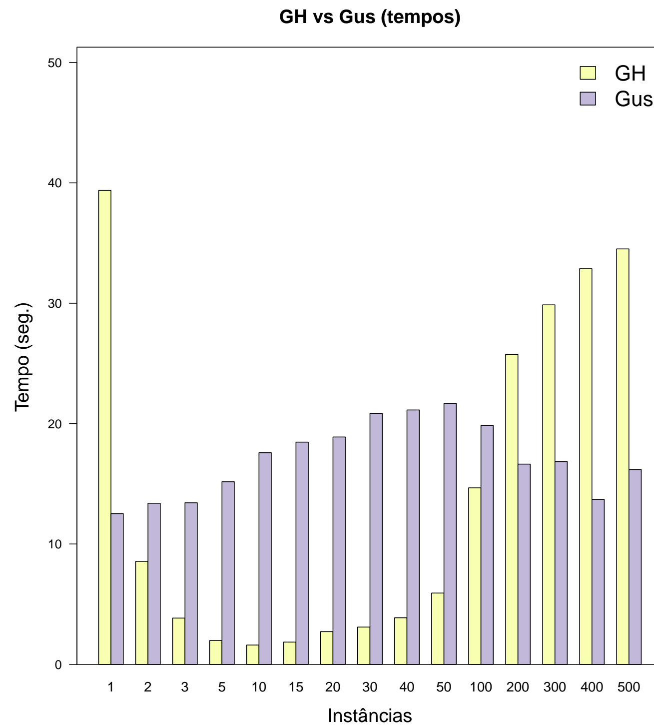


Figura 4.8: Tempos de execução dos algoritmos GH e Gus nas instâncias da classe NOI com número de *clusters* variável.

[2]O número máximo de máquinas semelhantes as quais tivemos acesso no *cluster* do LCPAD/UFPR foi 16. Durante a execução desse experimento, apenas 7 dessas máquinas estavam disponíveis. Por isso, a escalabilidade das soluções não foi testada para um número maior de máquinas.

Tabela 4.4: Resultados da implementação paralela do algoritmo de GH nas instâncias NOI. As colunas apresentam, respectivamente: número de *clusters* da instância, acrônimos dos algoritmos, tempos de inicialização, tempos para atualização da árvore, tempos para as contrações, tempos para cálculo dos cortes mínimos nos grafos contraídos, tempos para cálculos dos cortes mínimos no grafo de entrada, tempo total (real), número de cortes não descartados e o tamanho médio do grafo usado.

Núm. de Clusters	**Alg.**	**Inic.**	**Atual. da Árvore**	**Contrações**	**Fluxo Grafo Contraído**	**Fluxo Grafo de Entrada**	**Tempo Real**	**Cortes não usados**	**Tam. Médio Grafo**
1	**ParGH**	0.064	0.111	161.965	71.935		39.37	89	1000
	ParGus	0.107	0.013			73.330	12.52	44	1000
2	**ParGH**	0.059	0.054	40.877	9.323		8.56	66	505
	ParGus	0.109	0.007			78.665	13.39	55	1000
3	**ParGH**	0.056	0.033	19.658	2.492		3.85	57	340
	ParGus	0.103	0.014			78.753	13.42	40	1000
5	**ParGH**	0.059	0.014	9.262	1.893		1.98	38	212
	ParGus	0.099	0.012			89.271	15.17	40	1000
10	**ParGH**	0.055	0.006	6.421	2.542		1.61	31	122
	ParGus	0.104	0.013			103.605	17.58	33	1000
15	**ParGH**	0.061	0.006	6.952	3.502		1.85	26	101
	ParGus	0.098	0.011			108.922	18.46	27	1000
20	**ParGH**	0.062	0.007	9.443	6.204		2.72	26	105
	ParGus	0.102	0.016			111.553	18.89	24	1000
30	**ParGH**	0.057	0.011	11.014	6.878		3.10	37	100
	ParGus	0.093	0.011			123.215	20.85	43	1000
40	**ParGH**	0.066	0.018	13.090	9.266		3.87	42	102
	ParGus	0.101	0.017			124.884	21.13	34	1000
50	**ParGH**	0.060	0.016	20.539	14.136		5.92	51	147
	ParGus	0.112	0.018			128.161	21.68	34	1000
100	**ParGH**	0.062	0.046	51.986	34.752		14.66	54	309
	ParGus	0.106	0.011			117.176	19.85	32	1000
200	**ParGH**	0.057	0.091	98.560	54.121		25.75	46	578
	ParGus	0.096	0.013			98.081	16.63	31	1000
300	**ParGH**	0.060	0.094	114.583	62.817		29.87	57	685
	ParGus	0.103	0.011			99.158	16.85	48	1000
400	**ParGH**	0.059	0.105	127.759	67.463		32.87	61	779
	ParGus	0.099	0.011			80.406	13.70	20	1000
500	**ParGH**	0.063	0.101	135.102	69.880		34.51	65	830
	ParGus	0.113	0.009			95.299	16.18	39	1000

4.3 Algoritmo Híbrido para Construção de Árvores de Cortes

Os algoritmos GH e Gus têm desempenhos bastantes distintos em diferentes instâncias. O algoritmo Gus é melhor em muitas delas, porém existem instâncias em que o algoritmo GH é muito melhor. O principal motivo disso é o fato do algoritmo GH gastar tempo para construir os grafos contraídos, uma operação computacionalmente custosa, que pode ou não ser compensada pelo ganho de tempo no cálculo dos cortes mínimos. Isso sugere o desenvolvimento de uma versão híbrida desses algoritmos, que pode computar os cortes mínimos tanto no grafo de entrada quanto no grafo contraído. O objetivo desse algoritmo híbrido é alcançar um desempenho mais robusto, que não dependa tanto das características de cada instância. Abaixo a versão híbrida dos algoritmos GH e Gus é descrita e os resultados experimentais da avaliação de desempenho do algoritmo híbrido são apresentados na seção seguinte.

A versão híbrida dos algoritmos de GH e Gus mantém a árvore em construção como no algoritmo GH. O grafo contraído é construído de forma condicional, dependendo do número de vértices do grafo contraído. Estabelecemos um limiar, T, $0 \leq T \leq 1$, que é utilizado da seguinte maneira: na execução do algoritmo de GH, depois da escolha do nodo pivô e do par de vértices a ser separado, é verificado se o grafo contraído terá menos vértices do que T vezes o número de vértices do grafo de entrada. Nesse caso, o grafo contraído é construído e, em caso contrário, o algoritmo utiliza o grafo de entrada para o cálculo do corte mínimo.

Como mostrado na Seção 4.1, é possível atualizar a árvore em construção do algoritmo GH mesmo quando o corte encontrado cruza os cortes utilizados anteriormente. A atualização da árvore é feita como no algoritmo GH paralelo: (i) o nodo pivô é particionado de acordo com o corte. (ii) cada aresta $e = \{piv\hat{o}, v\}$ incidente ao nodo pivô é reconectada ao nodo que contém s ou ao nodo que contém t de acordo com o lado do corte que se encontra o vértice $e.v$.

A seguir apresentamos os resultados experimentais do algoritmo híbrido.

4.3.1 Resultados Experimentais: Algoritmo Híbrido

Os experimentos com o algoritmo híbrido foram feitos no mesmo ambiente computacional descrito na Seção 4.2. Foram comparados os tempos de execução dos algoritmo Gus, GH e híbrido nas instâncias NOI com 15 diferentes números de *clusters* e em 10 instâncias variadas. São mostrados gráficos com os tempos de execução e também tabelas com estatísticas detalhadas das execuções que includem o tempo gasto nas principais operações do algoritmo.

Nos experimentos executados, a versão híbrida teve um bom desempenho e foi melhor

do que o algoritmo GH em quase todas as instâncias. Nas instâncias em que o algoritmo Gus é mais rápido do que o algoritmo GH, o desempenho da versão híbrida foi mais próximo do desempenho do algoritmo Gus na maioria das instâncias.

O gráfico da Figura 4.9 mostra o desempenho dos 3 algoritmos nas instâncias NOI com diferentes números de *clusters*. O algoritmo Gus é mais rápido do que o algoritmo GH nas instâncias com $k = 1, 200, 300, 400, 500$. Por outro lado, o algoritmo GH é mais rápido do que o algoritmo Gus nas outras 10 instâncias geradas com valores intermediários de k. O desempenho da versão híbrida foi próximo do melhor dentre os algoritmos Gus e GH. Em várias instâncias, o algoritmo híbrido foi o mais rápido dentre todos. Além disso, o algoritmo híbrido foi melhor do que o algoritmo GH em *todas* as instâncias.

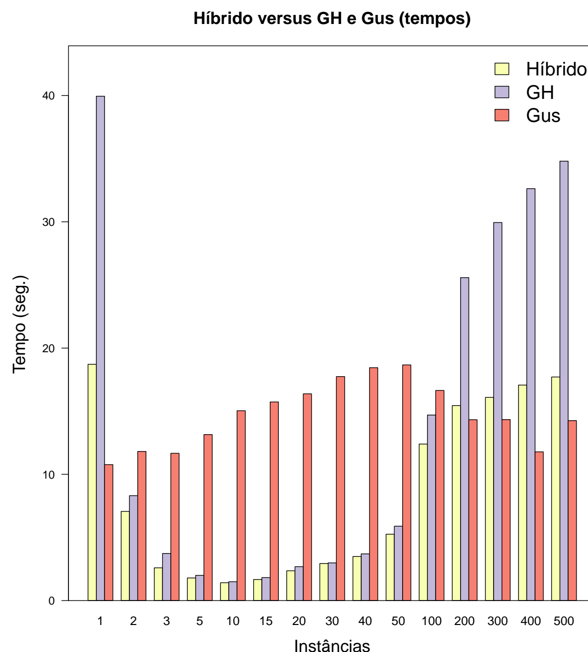


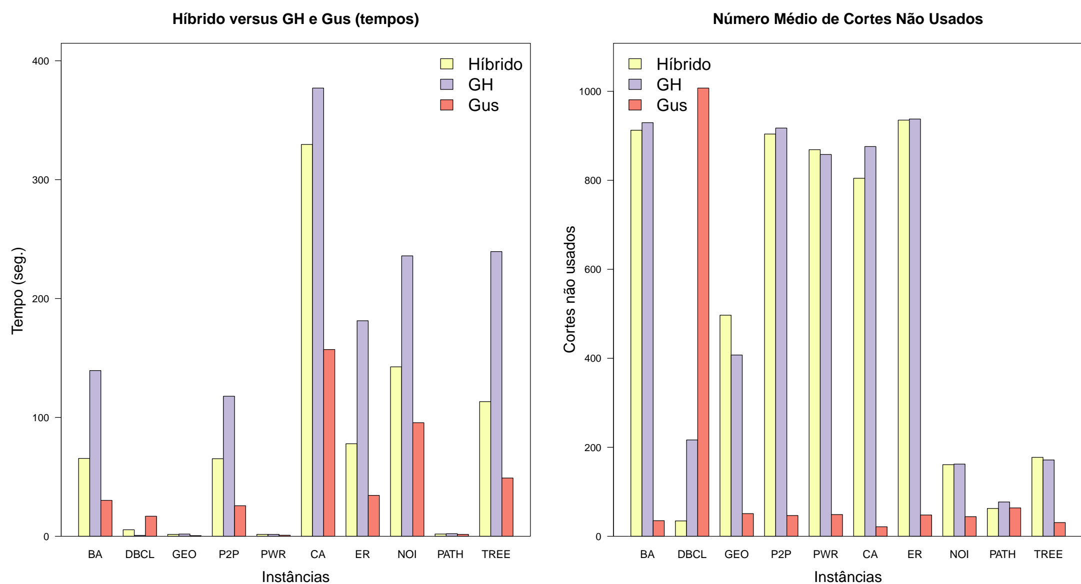
Figura 4.9: Resultados do algoritmo híbrido (executado em 7 máquinas; tempos em segundos).

A Tabela 4.5 mostra as estatísticas detalhadas das execuções dos 3 algoritmos. Note que o tempo gasto nas operações de criar o grafo contraído é menor no algoritmo híbrido do que no algoritmo GH.

Nas instâncias em que o algoritmo Gus é melhor do que o algoritmo GH, o algoritmo híbrido foi um pouco pior do que o algoritmo de Gus, mesmo tendo executado praticamente todos os fluxos no grafo de entrada e não gastando tempo com as operações de contração. Na Tabela 4.5 vemos que o tempo de cálculo dos cortes nessas instâncias foi maior no algoritmo híbrido do que no algoritmo Gus. Podemos enumerar alguns fatores que podem explicar essa diferença. (i) Os algoritmos fazem diferentes escolhas de pares de vértices a serem separados

e, portanto, as instâncias dos problemas de cortes mínimos não são idênticas e isso tem influência nos tempos de execução. (*ii*) O algoritmo híbrido trabalha com estruturas de dados mais complexas e que ocupam mais memória, o que provoca uma maior contenção pela memória principal e um menor aproveitamento da memória *cache*. (*iii*) Na versão paralela do algoritmo Gus, o processo mestre define uma instância do corte mínimo enviando ao processo escravo apenas os pares de vértices s e t. Por outro lado, no algoritmo híbrido, a instância é definida por um vetor de tamanho $|V|$. Essa diferença afeta o tempo de comunicação entre os processos.

O algoritmo híbrido também foi executado no conjunto de 10 instâncias variadas. As instâncias são as mesmas descritas na Tabela 4.1 do capítulo anterior, exceto a instância DBCL que passou a ter 2048 vértices e a instância TREE que passou a ter 2000 vértices. Os tempos de execução e o número de cortes descartados pelos algoritmos aparecem na Figura 4.10. Estatísticas detalhadas aparecem na Tabela 4.6. O algoritmo híbrido foi melhor do que o algoritmo GH na maioria das instâncias, mas não foi tão rápido quanto o algoritmo Gus em várias instâncias. O número de cortes descartados pelos algoritmos híbrido e GH foi bem maior do que pelo algoritmo Gus. Isso explica a diferença nos tempos de execução e mostra que outras otimizações na estratégia de escolha dos pares de vértices s e t podem ainda reduzir os tempos de execução do algoritmos GH e do algoritmo híbrido.



(a) Tempos de execução em segundos.

(b) Número de cortes descartados.

Figura 4.10: Resultados do algoritmo híbrido (executado em 7 máquinas).

Tabela 4.5: Resultados da implementação paralela do algoritmo híbrido nas instâncias NOI. O algoritmo híbrido foi executado com $T = 0.7$ em 7 máquinas. As colunas apresentam, respectivamente: número de *clusters* da instância, acrônimos dos algoritmos, tempos de inicialização, tempos para atualização da árvore, tempos para as contrações, tempos para cálculo dos cortes mínimos nos grafos contraídos, tempos para cálculos dos cortes mínimos no grafo de entrada, tempo total (real), número de cortes não descartados e o tamanho médio do grafo usado. As linhas para k igual a 50, 100 e 400 foram excluídas por falta de espaço.

Núm. de Clusters	Alg.	Inic.	Atual. da Árvore	Contrações	Fluxo Grafo Contraído	Fluxo Grafo de Entrada	Tempo Real	Cortes não usados	Tam. Médio Grafo
1	**Híbrido**	0.062	0.102	0.000	0.000	128.789	18.71	94	1000
	ParGH	0.065	0.112	162.876	113.663		39.95	93	1000
	ParGus	0.102	0.010			73.245	10.77	57	1000
2	**Híbrido**	0.061	0.053	31.982	13.622	1.871	7.06	72	506
	ParGH	0.060	0.056	41.688	15.304		8.31	68	506
	ParGus	0.119	0.012			80.530	11.81	68	1000
3	**Híbrido**	0.068	0.042	12.189	2.423	1.709	2.59	67	341
	ParGH	0.066	0.043	20.980	4.057		3.73	64	341
	ParGus	0.104	0.015			79.509	11.66	43	1000
5	**Híbrido**	0.054	0.014	7.528	1.041	2.381	1.79	44	213
	ParGH	0.061	0.011	10.009	3.167		2.00	50	214
	ParGus	0.099	0.008			90.063	13.15	48	1000
10	**Híbrido**	0.059	0.007	3.732	0.503	4.165	1.41	36	124
	ParGH	0.067	0.007	6.472	3.260		1.49	38	122
	ParGus	0.105	0.014			103.098	15.04	36	1000
15	**Híbrido**	0.058	0.009	3.964	1.040	5.211	1.67	29	105
	ParGH	0.060	0.008	7.438	4.646		1.82	32	102
	ParGus	0.098	0.012			108.009	15.73	34	1000
20	**Híbrido**	0.057	0.007	4.418	1.636	8.936	2.36	32	108
	ParGH	0.061	0.006	9.949	8.116		2.68	35	106
	ParGus	0.105	0.009			112.500	16.37	28	1000
30	**Híbrido**	0.063	0.018	6.256	2.833	9.764	2.94	41	108
	ParGH	0.064	0.012	11.272	8.712		2.99	44	101
	ParGus	0.107	0.016			121.939	17.74	42	1000
40	**Híbrido**	0.059	0.011	7.030	3.557	12.282	3.50	48	112
	ParGH	0.061	0.016	13.246	11.880		3.70	47	102
	ParGus	0.101	0.013			126.857	18.45	42	1000
200	**Híbrido**	0.060	0.080	9.519	6.395	90.455	15.45	43	660
	ParGH	0.060	0.085	98.664	78.512		25.58	57	576
	ParGus	0.105	0.015			98.169	14.32	35	1000
300	**Híbrido**	0.061	0.081	1.428	0.004	109.974	16.09	46	767
	ParGH	0.061	0.104	116.335	91.273		29.94	54	685
	ParGus	0.108	0.016			98.048	14.33	44	1000
500	**Híbrido**	0.058	0.093	1.285	0.003	121.397	17.71	65	873
	ParGH	0.061	0.110	136.530	104.572		34.80	68	831
	ParGus	0.097	0.014			97.669	14.25	46	1000

Tabela 4.6: Resultados da implementação paralela do algoritmo híbrido nas instâncias variadas. O algoritmo híbrido foi executado com $T = 0.7$ em 7 máquinas. As colunas apresentam, respectivamente: número de *clusters* da instância, acrônimos dos algoritmos, tempos de inicialização, tempos para atualização da árvore, tempos para as contrações, tempos para cálculo dos cortes mínimos nos grafos contraídos, tempos para cálculos dos cortes mínimos no grafo de entrada, tempo total (real), número de cortes não descartados e o tamanho médio do grafo usado.

Núm. de Clusters	Alg.	Inic.	Atual. da Árvore	Contrações	Fluxo Grafo Contraído	Fluxo Grafo de Entrada	Tempo Real	Cortes não usados	Tam. Médio Grafo
BA	**Híbrido**	0.031	2.058	0.000	0.000	376.821	65.53	913	10000
	ParGH	0.034	2.360	461.896	355.848		139.43	930	9999
	ParGus	0.054	0.006			179.463	30.24	35	10000
DBCL	**Híbrido**	0.012	0.044	1.226	1.134	30.003	5.50	35	1543
	ParGH	0.014	0.044	1.304	2.358		0.79	217	1025
	ParGus	0.010	0.010			100.013	16.88	1007	2048
GEO	**Híbrido**	0.015	0.138	3.040	1.949	1.618	1.61	497	1485
	ParGH	0.016	0.138	5.091	3.388		1.90	407	1385
	ParGus	0.011	0.006			2.482	0.56	51	3621
P2P	**Híbrido**	0.027	2.266	72.839	42.635	257.349	65.25	904	9348
	ParGH	0.027	2.960	396.369	282.044		117.82	917	8316
	ParGus	0.049	0.007			152.048	25.70	47	10876
PWR	**Híbrido**	0.009	0.100	1.306	1.209	0.294	1.63	869	587
	ParGH	0.014	0.101	1.533	1.459		1.62	858	572
	ParGus	0.011	0.010			4.198	0.90	49	4941
CA	**Híbrido**	0.117	2.090	613.483	295.804	1054.932	329.58	805	7939
	ParGH	0.118	2.188	1441.114	796.372		377.03	876	6946
	ParGus	0.223	0.022			938.367	157.11	21	11204
ER	**Híbrido**	0.034	4.720	4.873	0.000	447.657	77.86	935	9985
	ParGH	0.035	5.158	597.285	457.558		181.32	938	9963
	ParGus	0.073	0.007			204.251	34.42	48	10000
NOI	**Híbrido**	0.158	0.070	4.956	0.007	849.409	142.52	161	1901
	ParGH	0.162	0.070	723.810	688.068		235.90	162	1763
	ParGus	0.268	0.002			570.090	95.56	44	2000
PATH	**Híbrido**	0.021	0.033	0.329	0.017	10.121	1.90	63	1018
	ParGH	0.016	0.030	6.549	5.488		2.18	77	569
	ParGus	0.020	0.001			9.155	1.64	64	2000
TREE	**Híbrido**	0.155	0.079	1.330	0.001	674.054	113.30	177	1978
	ParGH	0.161	0.074	821.729	612.342		239.50	172	1947
	ParGus	0.265	0.002			291.087	49.02	31	2000

4.4 Heurísticas para o Algoritmo Paralelo de Gomory-Hu

Nesta seção são descritas as heurísticas que objetivam a melhoria do desempenho dos algoritmos paralelos de Gomory-Hu e híbrido para árvores de cortes. As heurísticas mais importantes são: a enumeração dos s-t-cortes mínimos para a busca de cortes mais balanceados, descrita na Seção 4.4.1 e os algoritmos para a escolha do nodo pivô e do par de vértices s e t descritos na Seção 4.4.2. Outras heurísticas são descritas na Seção 4.4.3.

4.4.1 Enumeração dos s-t-Cortes Mínimos

Dado um grafo G e dois vértices s e t, pode existir mais de um s-t-corte mínimo. Por exemplo, o grafo da Figura 4.11 possui um número exponencial de s-t-cortes mínimos: o grafo tem $n+2$ vértices e todos os 2^n s-t-cortes são mínimos. O algoritmo de fluxo máximo utilizado para encontrar os s-t-cortes mínimos produz ao seu final uma rede residual contendo as arestas não saturadas do grafo original e uma aresta (v,u) para cada aresta (u,v) associada a um fluxo positivo. Um s-t-corte $(S,\overline{S})$, $s \in S$, $t \in \overline{S}$, é um s-t-corte mínimo do grafo original se, e somente se, $(\overline{S},S)$ é um corte orientado da rede residual [30], isto é, $E_G(\overline{S},S) = \emptyset$, onde $E_G(\overline{S},S)$ é o conjunto das arestas com o vértice de origem em $\overline{S}$ e o vértice de destino em S.

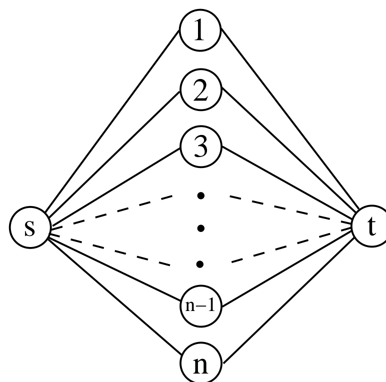


Figura 4.11: Grafo com um número exponencial de s-t-cortes mínimos.

Para encontrar um único s-t-corte mínimo, pode-se fazer uma busca a partir do vértice de origem na rede residual: os vértices alcançados pela busca definem um s-t-corte mínimo. Esse método tem a característica de retornar o corte mínimo com o menor conjunto S possível. Se desejamos que o algoritmo localize cortes mais balanceados, um outro procedimento deve ser utilizado.

A enumeração de cortes é um problema que já foi estudado anteriormente. Em particular, J. S. Provan e D. R. Shier em [88] mostram um esquema para enumerar s-t-cortes que pode ser aplicado a diversos problemas relacionados, em particular, o caso de interesse aqui,

que é enumerar s-t-cortes mínimos em grafos ponderados. Com base nesse algoritmo, foi implementada uma rotina de enumeração de cortes. Essa rotina é integrada ao código HIPR que computa o corte mínimo.

O primeiro passo do algoritmo é transformar a rede residual em um grafo orientado e acíclico através da contração dos componentes fortemente conexos. Utilizamos o algoritmo para enumerar os componentes fortemente conexos descrito em [30], que resolve o problema em tempo linear fazendo duas buscas em profundidade. Vamos denotar esse grafo acíclico por $R_G = (V_R, E_R)$.

O próximo passo para enumerar os s-t-cortes mínimos consiste em enumerar os cortes orientados no grafo acíclico R_G. O algoritmo de enumeração dos cortes orientados é recursivo. Ele mantém três conjuntos de vértices S, T e C. O conjunto S contém os vértices que ficarão do lado de s do próximo s-t-corte a ser enumerado. O conjunto T contém os vértices que ficarão do lado de t. O conjunto C contém vértices candidatos a entrar em S, que são os vértices com grau de saída igual a 0 em relação ao conjunto $\overline{S}$. Esse valor é denotado no pseudocódigo por $grau^{+}_{\overline{S}}(v)$. Um vértice v entra em C quando $grau^{+}_{\overline{S}}(v) = 0$. Quando o conjunto C estiver vazio, a partição $(S, \overline{S})$ é um corte a ser enumerado. O algoritmo mantém a propriedade de que $\{S, \overline{S}\}$ é um corte orientado sem arestas saindo de S como invariante. Um vértice v somente entra em S se $E(\{v\}, \overline{S}) = \emptyset$.

O Algoritmo 4.3 mostra o pseudocódigo do algoritmo de enumeração. O algoritmo deve ser iniciado com uma chamada a `Enumera_Cortes` (S, T, S_{out}) com o conjunto S contendo o vértice s e todos os vértices que podem ser alcançados a partir de s. O conjunto T deve conter o vértice t. O melhor corte encontrado ao longo da enumeração é devolvido em S_{out}.

No procedimento `Enumera_Cortes`, se existe pelo menos um vértice v tal que $grau^{+}_{\overline{S}}(v)$ é igual a 0, então duas recursões são feitas: uma com v em S e outra com v em T. Caso não haja nenhum vértice com $grau^{+}_{\overline{S}}(v) = 0$, então o corte $(S, \overline{S})$ é processado. Como o objetivo do procedimento é encontrar o corte mais balanceado, um índice de balanceamento do corte definido como $\mathtt{B}(S) = ||S| - \frac{n}{2}|$ é calculado e o corte com o menor índice é mantido.

A árvore de recursão de uma execução do algoritmo de enumeração é definida da seguinte maneira. Os nodos internos da árvore de recursão correspondem à escolha de um pivô do conjunto C definido na linha 1 do pseudocódigo. As arestas da árvore correspondem às chamadas recursivas das linhas 5 e 8: à esquerda com *pivô* em S e à direita com o *pivo* em T. As folhas da árvore de recursão correspondem ao processamento do corte $(S, \overline{S})$. A Figura 4.12 mostra um grafo orientado acíclico e a árvore de recursão do algoritmo de enumeração dos cortes orientados. A raiz da árvore corresponde à escolha do vértice A como pivô. O outro nodo

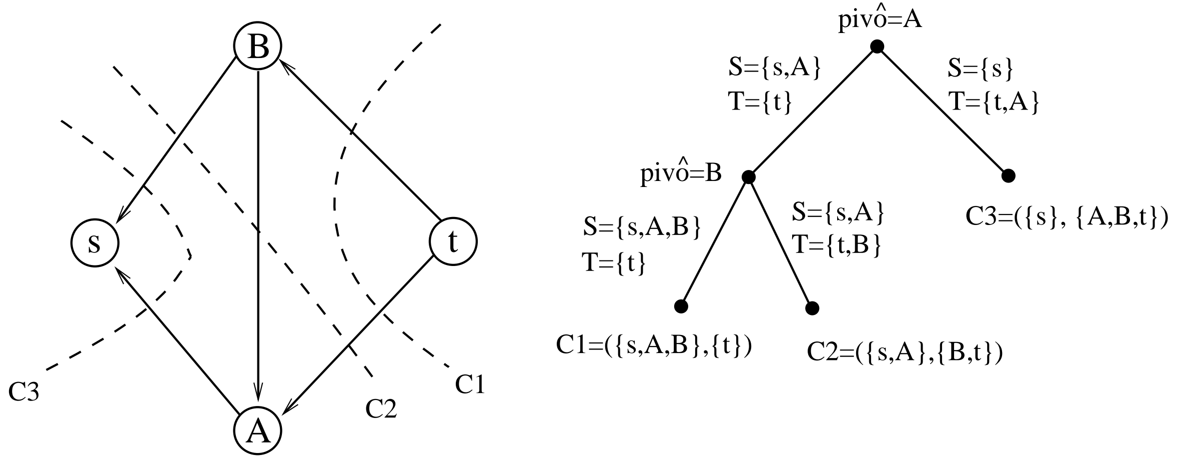


Figura 4.12: Grafo orientado acíclico e a árvore de recursão do algoritmo de enumeração de cortes.

interno corresponde à escolha do vértice B como pivô. As três folhas da árvore correspondem aos cortes C_1, C_2 e C_3 indicados no grafo da Figura 4.12.

O algoritmo de enumeração tem complexidade de tempo linear por corte produzido, como demonstrado no Lema 4.2 abaixo. Como o número de cortes pode ser grande (até mesmo exponencial), a busca foi limitada a 1000 cortes na implementação.

Lema 4.2. *O algoritmo de enumeração de cortes tem complexidade de tempo de $O(|V|+|E|)$ por corte processado.*

Demonstração. A altura máxima da árvore de recursão é $|V|$, pois cada chamada recursiva corresponde a inserir um vértice no conjunto S ou no conjunto T e esse vértice não pode ser novamente um pivô em chamadas recursivas subsequentes, pois não pode entrar novamente no conjunto C definido na linha 1. Os graus de saída dos vértices são atualizados cada vez que um vértice pivô entra ou sai do conjunto S. Isso é feito percorrendo as arestas que chegam em pivô. Logo, ao longo de um caminho da raiz até uma folha da árvore de recursão, no máximo $|E|$ dessas atualizações de grau podem ser efetuadas. □

Em grafos cujos s-t-cortes mínimos tendem a ser únicos, a heurística do corte balanceado irá aumentar o tempo de execução sem proporcionar ganhos. Entretanto, se os s-t-cortes mínimos não são únicos, então essa heurística é importante para que o algoritmo GH utilize cortes mais balanceados.

Algoritmo 4.3 Algoritmo de enumeração dos *s*-*t*-cortes mínimos.

Procedimento `Enumera_Cortes` (S, T, S_{out})

Entrada: Grafo acíclico $R = (V_R, E_R)$ obtido da rede residual e os vértices s e t. Conjuntos S e T.

Saída: S_{out} tal que $(S_{out}, \overline{S}_{out})$ é um corte de G

1: $C = \{v \in V_R \backslash (S \cup T) \mid grau^{+}_{\overline{S}}(v) = 0\}$ // grau de saída igual a 0
2: **se** $C \neq \emptyset$ **então**
3: $\quad$ $piv\hat{o} \leftarrow$ retira um vértice de C
4: $\quad$ $S \leftarrow S \cup \{piv\hat{o}\}$
5: $\quad$ `Enumera_Cortes` (S, T, S_{out})
6: $\quad$ $S \leftarrow S \backslash \{piv\hat{o}\}$
7: $\quad$ $T \leftarrow T \cup \{piv\hat{o}\}$
8: $\quad$ `Enumera_Cortes` (S, T, S_{out})
9: **senão** // processa o corte $(S, \overline{S})$
10: $\quad$ **se** $||S| - \frac{n}{2}| < ||S_{out}| - \frac{n}{2}|$ **então**
11: $\quad\quad$ $S_{out} \leftarrow S$ $\quad$ // S é mais balanceado
12: **Fim do Procedimento**
// Chamada inicial do procedimento `Enumera_Cortes`
13: $S \leftarrow \{s\} \cup \{v \in V_R \mid$ existe caminho orientado de s a $v\}$
14: $T \leftarrow \{t\}$
15: $S_{out} \leftarrow S$
16: `Enumera_Cortes` (S, T, S_{out})

4.4.2 Escolha do Pivô e dos Vértices a Separar

A condição para que o processo mestre aproveite um corte mínimo calculado por um processo escravo é que o par de vértices separado pelo corte esteja contido em um mesmo nodo da árvore no momento de sua atualização. A primeira medida para evitar que cortes calculados em paralelo provoquem o descarte de outros cortes é alternar a escolha do nodo pivô de onde sairá o par a ser separado. Para isso, os nodos da árvore que contém mais de um vértice são mantidos em uma lista. Cada par de vértices escolhido está contido no nodo posterior ao nodo de onde o par da tarefa anterior foi escolhido. Quando todos os possíveis pares de vértices de um nodo tiverem sido enumerados, o nodo é marcado para ser excluído de futuras escolhas. Caso todos os nodos estejam marcados, então todos os nodos são desmarcados para que o algoritmo possa prosseguir com uma nova escolha de nodo.

A escolha do par de vértices a ser separado é feita por um algoritmo de enumeração de pares. A forma mais simples de enumerar pares é aquela que segue a ordem $(1,2),(1,3),(1,4)$, $\ldots,(2,3),(2,4)$, e assim por diante. Dado que cortes triviais são bastante frequentes, essa ordem pode não ser interessante: suponha que os k pares de vértices $(1,2),(1,3),(1,4),\ldots,(1,k)$ estejam sendo separados concorrentemente. Caso o primeiro corte encontrado seja um corte trivial da forma $(\{1\}, V \backslash \{1\})$, então todos os outros $k-1$ cortes serão desperdiçados.

Uma ordenação dos pares que pode ser mais efetiva é aquela que enumera pares disjuntos tanto quanto possível antes de repetir qualquer vértice. Por exemplo, uma ordenação dos pares de um conjunto com 4 vértices com essa propriedade é a seguinte:

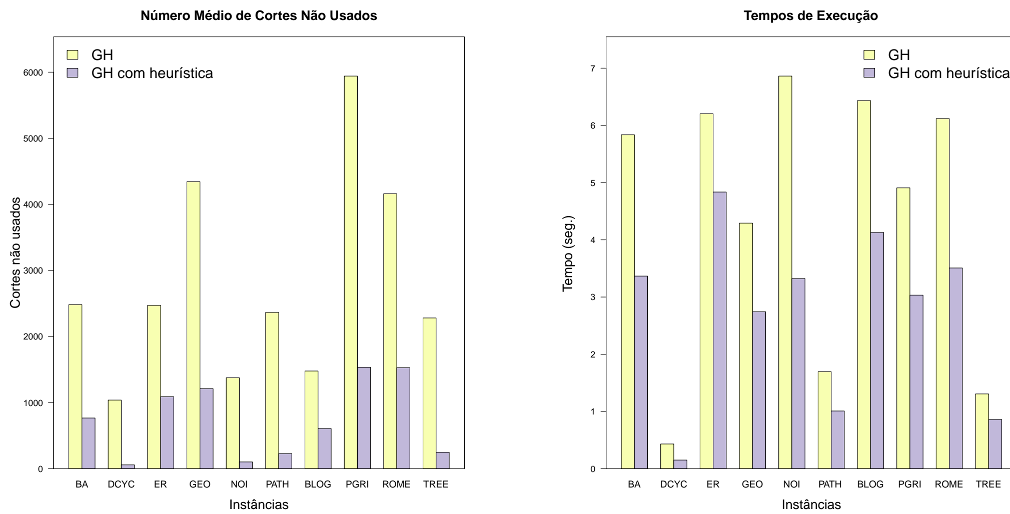
$$\overbrace{(1,2),(3,4)},\overbrace{(2,3),(4,1)},\overbrace{(1,3),(2,4)}$$

As chaves indicam os grupos de pares que são disjuntos. Essa lógica de enumeração de pares é a mesma dos emparceiramentos em torneios *round-robin* com n jogadores. Cada jogador deve jogar contra todos os outros e cada rodada deve ser formada por $\lfloor n/2 \rfloor$ partidas. Esse problema tem solução conhecida [27] que produz um emparceiramento com $n-1$ rodadas se n é par e n rodadas se n é ímpar. Essa solução resolve o problema estático, quando o conjunto de elementos é fixo. Porém, a escolha de pares de vértices no algoritmo GH é feita em um conjunto do qual elementos são removidos. O algoritmo de enumeração de pares implementado gera primeiro todos os pares cujos valores diferem de 1, seguidos de todos os pares cujos valores diferem de 2, e assim por diante. Essa enumeração não produz um emparelhamento *round-robin* para todos os valores de n, porém gera sequências longas de pares disjuntos. Essa solução tem a vantagem de poder ser implementada com dois iteradores e mais algumas variáveis de controle que podem ser mantidas consistentes quando elementos são removidos do conjunto.

Experimento com o algoritmo de enumeração de pares O desempenho da heurística de escolha dos pares de vértices descrita acima foi avaliado no conjunto de instâncias variadas. A comparação foi feita com a escolha dos vértices seguindo a ordem $(1,2),(1,3),(1,4),\ldots$ e sem alternar o nodo pivô. O número médio de cortes descartados em cada instância pode ser visto na Figura 4.13a. Os tempos médios das execuções são apresentados na Figura 4.13b. O número de cortes descartados foi bastante reduzido em todas as instâncias, gerando um impacto significativo nos tempos de execução mostrados na Figura 4.13b. A maior redução no tempo de execução foi na instância NOI na qual a heurística produziu uma redução de tempo de quase 50%.

4.4.3 Outras Heurísticas

Além das heurísticas já apresentadas, outras heurísticas ou otimizações foram acrescentadas ao código da implementação paralela do algoritmo GH, descritas nesta seção.



(a) Número de cortes descartados.

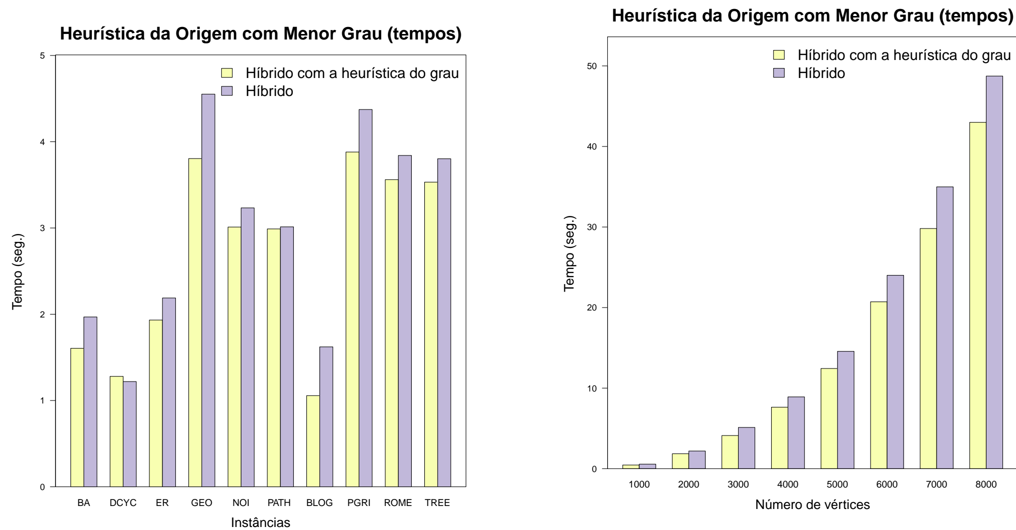
(b) Tempos de execução.

Figura 4.13: Comparação do desempenho do algoritmo GH com e sem a heurística de enumeração de pares de vértices disjuntos.

Origem como vértice de menor grau O algoritmo *push-relabel* [43] para o cálculo do fluxo máximo inicia a sua execução enviando o máximo possível de fluxo a partir do vértice de origem. Antes do final de sua execução, o excesso de fluxo necessita retornar ao vértice de origem. A escolha do vértice de menor grau para a origem tem como consequência uma redução no fluxo que é processado ao longo da execução do algoritmo, o que produz tempos de execução um pouco menores.

A Figura 4.14a mostra a diferença nos tempos de execução da versão híbrida do algoritmo GH com e sem a heurística em um conjunto de grafos diversos. Esses dois experimentos foram executados com dois processos (um mestre e um escravo) rodando em uma máquina com processador Intel Core 2 Duo E7400 de 2.80GHz. A versão com a heurística teve um melhor desempenho em 9 das 10 instâncias. Nos grafos da classe BA, a diferença entre o grau máximo de um vértice e o grau mínimo é grande. Por isso, executamos o mesmo experimento em grafos da classe BA, com $|V| = 1000, 2000, 3000, \ldots, 8000$. Os tempos de execução são mostrados na Figura 4.14b. A versão com a heurística foi a mais eficiente em todas as instâncias da classe BA.

Mover o menor número de vértices para o novo nodo da árvore A cada atualização da árvore em construção, um nodo deve ser dividido em dois. Na implementação, essa operação consiste na criação de um novo nodo e na transferência de uma parte dos vértices do nodo a ser dividido para o novo nodo. Mover sempre o menor número de vértices tem duas vantagens. A primeira: o algoritmo faz um número menor de operações no conjunto de vértices, o que torna



(a) Instâncias variadas.

(b) Instâncias da classe BA com $|V| = 1000, 2000, \ldots, 8000$.

Figura 4.14: Comparação do desempenho do algoritmo híbrido com e sem a heurística da origem como vértice de menor grau.

a atualização da árvore mais rápida. A segunda: os iteradores usados para enumerar os pares de vértices, como descrito na Seção 4.4.2, sofrem menos alterações e, com isso, a enumeração dos pares é mais consistente com a ordenação desejada.

Eliminação ou não de arestas múltiplas À medida que vértices do grafo são contraídos, arestas múltiplas são criadas. Uma opção para lidar com essas arestas consiste em identificá-las como uma única aresta cuja capacidade é a soma das capacidades das arestas que a originou.

O grafo contraído é construído por uma rotina que percorre as arestas do grafo de entrada e que para cada aresta verifica se existe ou não uma aresta associada no grafo contraído. O procedimento que adiciona a aresta no grafo contraído determina se o grafo admite arestas múltiplas ou não. Duas versões do procedimento que não admitem arestas múltiplas foram implementadas: uma utilizando o *container set* da *Standard Template Library* (STL) [78] que é implementado como uma árvore balanceada de busca e outra utilizando o *container unordered_set* da *Boost Library* [96] que é implementado como uma tabela de espalhamento. A versão que utiliza o tipo *unordered_set* se mostrou significativamente mais rápida.

Uma vez que o algoritmo GH gasta um tempo considerável na rotina que cria o grafo contraído e que a operação mais demorada dessa rotina é a inserção de arestas no grafo contraído, foi feita uma tentativa de implementar essa rotina sem a eliminação das arestas múltiplas. Sem dúvida, essa mudança tornou o tempo de construção do grafo contraído menor. Porém, o tempo para o cálculo dos cortes aumentou, já que ele é dependente do número de arestas.

Principalmente nas instâncias nas quais o tamanho do grafo contraído pode ser reduzido, o desempenho final do algoritmo de GH piorou.

Concluímos que a eliminação das arestas múltiplas é a melhor estratégia, principalmente quando o grafo contraído se torna relativamente pequeno. Apesar disso, o tempo de construção do grafo contraído é elevado e essa é uma parte da implementação onde o ganho de tempo com otimizações pode render bons resultados. Por exemplo, trabalhos futuros podem se concentrar em evitar que o processo escravo tenha que sempre percorrer todas as arestas do grafo de entrada para construir os grafos contraídos. Isso é possível já que alguns grafos contraídos são, em certo sentido, refinamentos de grafos previamente construídos.

Sobre a existência de cortes balanceados A existência de cortes pelo menos moderadamente balanceados é a principal característica dos grafos nos quais o algoritmo GH supera o algoritmo Gus. Utilizando o algoritmo de enumeração de cortes descrito na Seção 4.4.1 para todos os pares de vértices do grafo, podemos determinar qual é o corte mais balanceado do grafo e estabelecer um limite na diminuição do tamanho do grafo contraído.

Os resultados para algumas das instâncias utilizadas nos experimentos são apresentados na Tabela 4.7. Note que os grafos BA e ER não possuem cortes balanceados. Os grafos TREE, GEO, NOI, PATH, ROME e BLOG possuem cortes um pouco balanceados, com 233, 80, 83, 93, 107 e 40 vértices, respectivamente. O grafo POWER, obtido de uma rede de transmissão de eletricidade, possui pelo menos um corte bastante balanceado com 1226 vértices no lado menor. Note, entretanto, que execuções do algoritmo de GH nesse grafo podem não encontrar esse corte balanceado ou utilizá-lo tarde demais para conseguir uma diminuição no tempo de processamento. Esses resultados mostram que heurísticas para a escolha do par de vértices que favoreçam a busca de cortes balanceados podem ser exploradas para melhorar ainda mais o desempenho do algoritmo GH e da sua versão híbrida.

4.5 Implementação

O código da implementação paralela do algoritmo GH foi implementado em linguagem C e C++ e compilado com o g++ com nível de otimização `O3`. Foram utilizadas algumas rotinas da STL (*Standard Template Library*) para representação de vetores e conjuntos. Também utilizamos tabelas de espalhamento implementadas na *Boost Library*. Utilizou-se a implementação

Grafo	N	$\min\{\lvert S\rvert, \lvert\overline{S}\rvert\}$
BA	300	1
ER	300	2
GEO	3621	80
POWER	4941	1226
NOI (n=1000,k=15)	1000	83
PATH	2000	93
TREE	2000	233
ROME	3353	107
BLOG	1222	40
DBLC	1024	512

Tabela 4.7: Cortes mais balanceados em algumas instâncias utilizadas nos experimentos. Considerando um s-t-corte $\{S, \overline{S}\}$ mais balanceado, a tabela acima mostra o valor mínimo entre $|S|$ e $|\overline{S}|$.

do algoritmo de fluxo máximo *push-relabel* [43] chamada HIPR versão 3.7[3], desenvolvida por B.V. Cherkassky e A.V. Goldberg [24]. O HIPR está implementado em linguagem C e teve mínimas alterações para permitir a sua compilação em C++.

Estrutura do código O código está estruturado nos seguinte arquivos:

`ghsmpi.cc` Contém a função principal com o teste condicional para determinar se o processo é o mestre ou o escravo. E executa as seguintes operações (não necessariamente implementadas ali):

- chama as funções de inicialização e término do *MPI*;
- verifica os argumentos da linha de comando;
- faz a leitura do grafo de entrada;

Como processo mestre:

- mantém a árvore em construção;
- escolhe os nodos pivôs com os pares de vértices a serem separados;
- escolhe os pares de vértices a serem separados;
- gera tarefas para os processos escravos que consistem de uma partição dos vértices (que depende do pivô escolhido) e o par de vértices a ser separado;
- envia as tarefas para processos escravos;

[3]De direitos reservados por IG Systems, Inc. Copyright 1995-2004. O código fonte está disponível para uso não-comercial.

- envia mensagens de término aos processos escravos;
- recebe as respostas dos processos escravos e a partir delas atualiza a árvore em construção;
- grava a árvore de saída com dados estatísticos;
- calcula medidas estatísticas: tempo total (real) da execução, número de cortes não usados, tamanho médio dos grafos contraídos, tempo de inicialização, tempo para atualizar a árvore;
- grava um relatório com informações sobre cada corte (opcional);

Como processo escravo:

- Chama a função de inicialização das estruturas de dados para o cálculo dos cortes;
- recebe uma tarefa;
- constrói o grafo contraído;
- chama a função que calcula o corte mínimo no grafo contraído;
- envia o resultado ao processo mestre;
- calcula o tempo gasto na construção do grafo contraído e no cálculo do corte mínimo. Envia esses tempos ao processo mestre.

`edgelist.{hpp|cpp}` Contém as classes Edgelist e Edgelist_vector com a estrutura de dados para representar um grafo como uma lista de arestas. Possui uma versão que elimina arestas múltiplas (somando as capacidades) e outra que não. Veja a discussão na seção 4.4.3. Inclui uma função para leitura de arquivos com grafos em formato DIMACS[4].

`hi_pr.{hpp|cpp}` Contém o código para o cálculo de cortes mínimos. Implementa o algoritmo *push-relabel* para fluxos máximos. O código do fluxo máximo foi implementado por B.V. Cherkassky e A.V. Goldberg [24]. O arquivo `hi_pr.cpp` também contém as rotinas para enumeração dos cortes mínimos que permite a busca por um corte mais balanceado.

`tree.{hpp|cpp}` Esse arquivo contém a classe `GHTree` que representa uma árvore de cortes parcial. Cada nodo da árvore contém um conjunto de inteiros que representa vértices do grafo de entrada, implementado como um `STL/set`. O nodo contém outras informações de controle para representar a lista encadeada de pivôs e para auxiliar na escolha dos pares de vértices a serem separados. As arestas formam uma lista encadeada e contêm, além

[4] O formato DIMACS para representação de grafos consiste em um cabeçalho que define o número de vértices e o número de arestas do grafo e a lista de arestas em um arquivo em formato ASCII.

dos nodos que são extremos da aresta, o índice do par de vértices que produziu o corte mínimo que originou a aresta. Cada aresta tem um ponteiro para a sua aresta reversa. Métodos dessa classe implementam operações para atualizar a árvore, para escolher um pivô e para construir uma representação de uma partição dos vértices em relação ao pivô.

`task.{hpp|cpp}` Contém as classes `Task` e `Response` usadas na comunicação entre o processo mestre e os processos escravos.

4.6 Conclusão

Esse capítulo apresentou uma versão paralela do algoritmo de Gomory-Hu para a construção de árvores de cortes e resultados de diversos experimentos cujo objetivo foi a avaliação do desempenho do algoritmo proposto. Uma versão híbrida e original dos algoritmos Gus e GH foi proposta e resultados experimentais foram descritos. Finalmente, algumas heurísticas e otimizações para os algoritmos GH e híbrido foram propostas e avaliadas. A versão paralela do algoritmo GH apresentou bons *speedups* e superou o desempenho da versão paralela do algoritmo de Gusfield em algumas instâncias. Por sua vez, o algoritmo híbrido foi melhor do que o algoritmo GH em quase todas as instâncias e teve um bom desempenho também em relação ao algoritmo Gus.

Os experimentos utilizaram conjuntos de 10 grafos variados e uma família de grafos sintéticos (NOI) na qual a existência de cortes balanceados pode ser controlada por um parâmetro. As instâncias da família NOI permitiram a avaliação do algoritmo GH tanto em instâncias que favorecem esse algoritmo, quanto em instâncias mais desfavoráveis para ele. Esses experimentos mostraram precisamente como a variação do parâmetro k (o número de *clusters*) interfere no desempenho relativo dos algoritmo GH e Gus. Para valores intermediários de k, o algoritmo GH tem um desempenho muito superior ao algoritmo Gus, pois consegue reduzir o tamanho do grafo à medida que encontra cortes balanceados. Para valores extremos de k, que geram grafos com cortes menos balanceados, o algoritmo de Gus é o mais rápido.

A versão híbrida dos algoritmos GH e Gus tem por objetivo prover uma implementação mais robusta, que tenha um desempenho quase tão bom quanto o melhor dos dois algoritmos em cada instância. Na classe de instâncias NOI, esse resultado foi alcançado, como mostra o gráfico da Figura 4.9. Os resultados no conjunto de instâncias variadas mostra que a versão híbrida ainda precisa de otimizações para executar tão rápido quando o algoritmo Gus em todas as instâncias na qual este algoritmo é superior.

As implementações paralelas propostas são baseadas no modelo mestre/escravo. Es-

tatísticas detalhadas mostram que esse modelo é adequado e indicam que a escalabilidade da solução é boa, pois o tempo gasto pelo processo mestre em atualizações da árvore em construção é relativamente pequeno (veja, por exemplo, a Tabela 4.4).

Se por um lado os resultados foram satisfatórios, por outro ainda podemos sugerir várias possíveis otimizações: (i) heurísticas para a escolha dos pares de vértices que favoreçam a busca por cortes balanceados podem ser investigadas. (ii) Notamos que o que impede o algoritmo GH de superar o algoritmo Gus em algumas instâncias é a grande diferença no número de cortes descartados pelos algoritmos. Assim, heurísticas para reduzir o número de cortes descartados pelo algoritmo GH podem ser mais estudadas. (iii) Finalmente, a construção do grafo contraído leva um tempo de computação considerável. Otimizar esse procedimento pode levar os algoritmos GH e híbrido a um ganho expressivo de tempo. Como mencionado na Seção 4.4.3, a construção do grafo contraído talvez possa ser otimizada para que não seja necessário percorrer o grafo de entrada inteiro durante a construção de cada grafo contraído. Isso pode ser feito porque alguns grafos contraídos são refinamentos de outros desses grafos.

Os próximos capítulos tratam de uma das muitas aplicações das árvores de cortes, a saber, o cálculo de medidas de centralidade para os vértices de grafos.

5 Medidas de Centralidade em Grafos

Este capítulo é dedicado à apresentação de medidas de centralidade em redes. Apresentamos algumas medidas de centralidade previamente estudadas e amplamente utilizadas. Na sequência do capítulo, definimos as diversas medidas de centralidade baseadas em conectividade que são propostas originais deste trabalho. Na Seção 5.2.1 são definidas medidas de centralidade baseadas em arestas e na Seção 5.2.2 são definidas as medidas baseadas em cortes de vértices. A Seção 5.2.3 traz uma discussão sobre a generalização das medidas previamente apresentadas para grafos orientados.

5.1 Medidas de Centralidade: Definições, Propriedades e Classificação

Medidas de centralidade em grafos têm como objetivo quantificar diferentes noções intuitivas da importância dos vértices ou das arestas dos grafos. Diferentes critérios são utilizados: conectividade (e.g. grau), distâncias (e.g. grau de proximidade) ou a participação do vértice ou da aresta em caminhos relevantes (e.g. grau de intermediação). Na área de redes complexas, várias dessas medidas foram bem estudadas. Alguns exemplos são: grau, excentricidade, coeficiente de agrupamento, grau de proximidade, grau de intermediação e vulnerabilidade [107, 34, 85, 68, 82]. As medidas de centralidade estudadas neste trabalho são medidas de centralidade de vértices.

Não existe uma medida de centralidade adequada a todas as aplicações e, por isso, um grande número de diferentes medidas foram propostas [68]. Em [14], mais de 40 medidas de centralidade são citadas e depois classificadas de acordo com os critérios descritos na próxima seção.

Aplicações de medidas de centralidade incluem a detecção de nodos com maior influência, prestígio e controle sobre a rede (e.g. redes sociais) e a quantificação da importância de nodos para o fluxo de informações ou para a comunicação entre os nodos [68]. No estudo de

processos dinâmicos sobre redes, como, por exemplo, a disseminação de epidemias, medidas de centralidade permitem determinar um conjunto de nodos nos quais uma intervenção pode alterar o processo de maneira favorável [19]. Outra aplicação de medidas de centralidade consiste na alocação de recursos em uma rede. Diferentes critérios são otimizados por diferentes medidas de centralidade. Por exemplo, minimizar o tempo de resposta máximo corresponde à escolha de um vértice com excentricidade mínima [51].

Diversos autores afirmam que a pesquisa sobre medidas de centralidade em grafos teve o seu início no final da década de 1940 e início da década de 1950 com os trabalhos dos cientistas sociais A. Bevelas e H. J. Levitt, que estudaram o impacto da topologia de redes de pessoas na solução de problemas de lógica [103, 91]. Previamente, a ideia de centro de um grafo foi proposta pelo matemático C. Jordan no século XIX, que definiu o centro de uma árvore como o conjunto de vértices com excentricidade mínima [42]. Mais recentemente, na década de 2000, com o crescimento da pesquisa em redes complexas, iniciada predominantemente por físicos, biólogos e cientistas da computação, o número de artigos que tratam de medidas de centralidade e de suas aplicações cresceu bastante. L. C. Freeman, em [42], trata do histórico de transição de ideias sobre centralidade da área de Ciências Sociais para a Física e a Biologia.

5.1.1 Medidas de Centralidade: Exemplos

Com o intuito de ilustrar o conceito de centralidade, são apresentadas nesta seção algumas medidas de centralidade em grafos que são amplamente utilizadas.

Grau O *grau* de um vértice é a medida de centralidade mais simples. Em grafos não orientados, o grau é definido como o conjunto de arestas incidentes ao vértice. Denotamos o grau do vértice v por $d(v)$. Se A é a matriz de adjacência do grafo e a_{ij} o elemento da linha i e coluna j de A, então o grau do vértice i é definido por $d(i) = \sum_{1 \le j \le |V|} a_{ij}$.

Em um grafo orientado $G = (V, E)$, o *grau de entrada* do vértice v, denotado por $d^-(v)$, é definido por $d^-(v) = |\{(u,v) \in E\}|$ e o *grau de saída*, denotado por $d^+(v)$, é definido por $d^+(v) = |\{(v,u) \in E\}|$.

Excentricidade A *excentricidade* de um vértice v, denotada por $\mathrm{ecc}(v)$, é a distância máxima de v a um outro vértice do grafo, isto é,

$$\mathrm{ecc}(v) = \max_{u \in V} d(v,u).$$

Excentricidade Efetiva Seja $N(v,r)$ o conjunto de vértices alcançáveis a partir de v por caminhos de comprimentos menores do que ou iguais a r, isto é,

$$N(v,r) = \{u \in V \mid d(v,u) \leq r\}.$$

A *excentricidade efetiva*, denotada por ecc_eff(v,p), onde p é uma constante, é definida por

$$\text{ecc_eff}(v,p) = \min\{r \in \mathbb{N} \mid \ |N(v,r)| >= p|V|\}.$$

Ou seja, a excentricidade efetiva determina a distância mínima a ser percorrida a partir de v para que uma busca alcance uma fração p dos vértices do grafo. A excentricidade efetiva, definida por C. R. Palmer et al. [85], é uma versão mais robusta da excentricidade e foi utilizada no estudo da topologia da Internet em nível de roteadores. Os autores utilizaram o parâmetro $p = 0.90$.

Status O *status* é a soma das distâncias de um vértice a todos os demais vértices do grafo:

$$status(v) = \sum_{u \in V} d(v,u).$$

Grau de Proximidade (*closeness*) O *grau de proximidade* (*closeness*) é o inverso da soma das distâncias de um vértice a todos os vértices do grafo, isto é,

$$\text{closeness}(v) = \frac{1}{\sum_{u \in V} d(v,u)}.$$

Grau de Intermediação (*betweenness*) O grau de intermediação de um vértice v está relacionado com a quantidade de caminhos mínimos entre outros pares de vértices que contém v. Seja σ_{st} o número de caminhos mínimos entre os vértices s e t. Seja $\sigma_{st}(v)$ o número de caminhos mínimos entre s e t que contém v. Definimos o grau de intermediação de v por:

$$\text{betweenness}(v) = \sum_{\substack{s \in V \\ s \neq v}} \sum_{\substack{t \in V \\ t \neq v}} \frac{\sigma_{st}(v)}{\sigma_{st}}.$$

O estudo de medidas em redes complexas é extenso e tem contribuições de pesquisadores de várias áreas como Sociologia, Ciência da Computação, Matemática, Física, Estatística e Biologia ao longo de várias décadas. Muitos outros conceitos de centralidade em redes foram propostos com base em diferentes ideias incluindo distâncias, agrupamentos e ciclos, métodos espectrais, hierarquia, dentre outros. As seguintes resenhas [14, 68, 69, 31] contém levantamen-

tos bibliográficos extensos sobre o assunto.

5.1.2 Uma Classificação de Medidas de Centralidade

A grande diversidade de medidas de centralidade em redes torna conveniente organizá-las de maneira sistemática. Uma classificação de medidas de centralidade em grafos foi proposta por S. P. Borgatti e M. G. Everett [14]. Eles afirmam que as mais diversas medidas de centralidade dizem respeito à estrutura de caminhos do grafo. Além disso, as medidas de centralidade avaliam a quantidade ou o comprimento de caminhos que se originam ou terminam no vértice ou passam por ele. Por fim, as medidas de centralidade são calculadas utilizando alguma função como média, soma, máximo, entre outras.

A classificação proposta em [14] baseia-se, portanto, em 4 dimensões, resumidas abaixo:

Tipo dos Caminhos: caminhos mínimos ou caminhos de comprimento restrito, por exemplo.

Propriedade dos Caminhos: volume (número de caminhos) ou comprimento dos caminhos.

Posição nos Caminhos: radial ou medial.

Tipo de Sumarização: soma, média, máximo, entre outros.

A Tabela 5.1 abaixo traz exemplos de medidas de centralidade com as suas respectivas classificações.

Tabela 5.1: Classificação de medidas de centralidade.

Medida	Tipo dos Caminhos	Propriedade dos Caminhos	Posição nos Caminhos	Tipo de Sumarização
grau	comprimento um	volume	radial	soma
excentricidade	caminhos mínimos	comprimento	radial	máximo
grau de proximidade	caminhos mínimos	comprimento	radial	soma
grau de intermediação	caminhos mínimos	volume	medial	soma

5.1.3 Medidas Locais e Globais

Medidas de centralidade podem ser locais ou globais. Uma medida de centralidade de nodos é local se o seu valor permanece constante quando a rede sofre alterações que não envolvam as suas arestas incidentes ou os seus nodos vizinhos. Medidas globais são aquelas

nas quais mudanças distantes do nodo podem alterar o valor da medida desse nodo. O grau dos nodos é o arquétipo de medida local. A classificação entre medidas locais e globais é sujeita a gradações e a comparação entre algumas medidas só pode ser feita de forma relativa.

H. Tangmunarunkit et al. [100] observam que propriedades locais e globais são independentes, pois grafos com propriedades locais semelhantes podem apresentar grandes diferenças globais. Por exemplo, árvores com nodos internos de grau 4, reticulados bidimensionais cujos nodos têm grau 4 e grafos 4-regulares aleatórios têm distribuições de graus semelhantes, porém estruturas globais diferentes. Por outro lado, redes com distribuições dos graus bastante diferentes, podem compartilhar uma mesma estrutura global. Por exemplo, grafos aleatórios da classe G_{np} e redes livres de escala possuem distribuições de graus totalmente diferentes, porém, grafos gerados por ambos os modelos têm diâmetro pequeno com alta probabilidade [81].

Propriedades locais de uma rede, como o grau, nem sempre descrevem as propriedades relevantes para uma dada aplicação. Y. Shavitt e Y. Singer [98] estudaram mecanismos de roteamento tolerante a falhas na Internet e mostram que o grau dos nodos não é uma boa medida para classificar a importância dos nodos utilizados no roteamento. Por isso, os autores definiram outras medidas para melhor caracterizar os nodos em termos de importância para o roteamento, que levam em conta a existência de caminhos alternativos entre dois nodos.

S. Wuchty e P. F. Stadler [107] estudaram propriedades de centralidade de grafos no contexto de redes complexas biológicas. Os autores observam que o grau dos nodos é uma medida local que somente provê informação sobre os nodos quando o grafo foi gerado por um modelo cuja distribuição dos graus é conhecida. Esse trabalho compara as medidas excentricidade e *status* com o grau. Mostra-se que essas outras medidas frequentemente correlacionam com o grau, mas também que as diferenças indicam propriedades estruturais da rede que não são capturadas pelo grau.

As medidas de centralidade propostas na próxima seção baseiam-se nos conceitos de cortes em grafos e são parametrizadas de forma a produzir medidas que podem ser mais locais ou mais globais.

5.2 Medidas Baseadas em Conectividade

O objetivo das medidas de centralidade baseadas em conectividade é quantificar a conectividade do nodo em relação à rede. A medida de conectividade mais imediata é o grau do vértice. Entretanto, existem aspectos da conectividade do nodo que não são medidos pelo grau. Por exemplo, um nodo pode ter um grau elevado e a remoção de uma única aresta pode

desconectá-lo de qualquer outro nodo pré-definido; esse é o caso quando todas as arestas incidentes ao vértice são arestas de corte do grafo. Além disso, um vértice de grau alto pode encontrar-se na "periferia do grafo" por ter a excentricidade alta. Intuitivamente, percebemos que a limitação do grau como medida de conectividade está no fato de considerar apenas a vizinhança imediata do nodo, isto é, em ser uma medida local.

Com a ideia de contornar as limitações do grau, definimos duas classes de medidas de conectividade chamadas de *i-aresta-conectividade* e de *i-vértice-conectividade*, dependendo do tipo de cortes utilizados.

5.2.1 Medidas Baseadas em Cortes de Arestas

Nesta seção descrevemos as medidas de centralidade de nodos baseadas em aresta-conectividade. Como vimos na seção 1.1, a conectividade local entre dois vértices s e t é a capacidade de um s-t-corte mínimo e é denotada por $\lambda(u,v)$. Essa definição pode ser estendida para conjuntos com mais de dois vértices:

Definição 5.1. Seja $G=(V,E)$ um grafo ponderado (ou um grafo orientado ponderado). Considere um conjunto de vértices $X \subseteq V$, $|X| \geq 2$. A *aresta-conectividade* de X em relação a G, denotada por $\lambda_G(X)$, é a capacidade de um corte mínimo que separa quaisquer dois vértices de X, isto é, $\lambda_G(X) = \min\{\lambda_G(u,v) \mid u,v \in X\}$.

A aresta-conectividade de um grafo (grafo orientado) G, denotada por $\lambda(G)$, é o valor de $\lambda_G(V)$.

Ressaltamos que a conectividade de um conjunto X é diferente da conectividade do subgrafo induzido por X, pois a conectividade é relativa ao grafo e independe da existência de arestas entre os vértices de X. A Figura 5.1 ilustra melhor o conceito. O conjunto $X=\{a,b,c,d\}$ tem aresta-conectividade 3 embora o grafo induzido por X não contenha nenhuma aresta.

A seguir, definimos as medidas de aresta-conectividade para os vértices do grafo:

Definição 5.2 (***i*-aresta-conectividade**)**.** Seja $G=(V,E,c)$ um grafo (ou grafo orientado) capacitado e $i \in \mathbb{N}$, $2 \leq i \leq |V|$. A i-aresta-conetividade de um vértice v, denotada por $\lambda_i(v)$, é a máxima aresta-conectividade de um conjunto $X \subseteq V$ que satisfaz:

i. $v \in X$, e

ii. $|X| \geq i$.

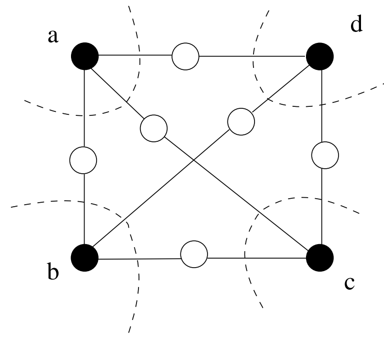


Figura 5.1: O conjunto $X = \{a,b,c,d\}$ tem aresta-conectividade 3, isto é, $\lambda(X) = 3$. As linhas tracejadas representam cortes mínimos que separam vértices de X.

Estendemos a definição acima para incluir os índices $i = 1$ e $i > |V|$. Temos:

Definição 5.3 (*i*-**aresta-conectividade estendida).** Seja $G = (V,E)$ um grafo (ou grafo orientado) e $i \in \mathbb{N}$, $i \geq 1$

$$\lambda_i(v) = \begin{cases} grau(v), & \text{se } i = 1, \\ \max_{X \subseteq V} \lambda_G(X) \mid v \in X \text{ e } |X| \geq i, & \text{se } 2 \leq i \leq |V|, \\ 0, & \text{caso contrário.} \end{cases}$$

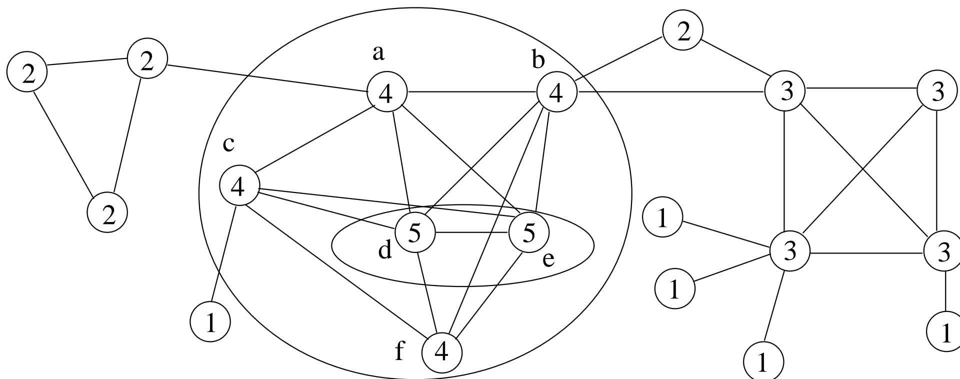


Figura 5.2: Exemplo do conceito de 2-aresta-conectividade, $\lambda_2(v)$. Alguns componentes de máxima conectividade são destacados.

A Figura 5.2 mostra um exemplo de grafo com a 2-aresta-conectividade indicada dentro dos vértices. A 2-aresta-conectividade é definida como a máxima aresta-conectividade entre pares de nodos. Por exemplo, $\lambda_2(a) = 4$ porque $\lambda_G(a,b) = 4$ e essa é a maior aresta-conectividade entre a e qualquer nodo. A 3-aresta-conectividade é definida em função de conjuntos de 3 vértices. Por exemplo, $\lambda_3(d) = 4$ porque a máxima aresta-conectividade de conjuntos com no mínimo 3 vértices contendo d é no máximo 4. Observe, porém, que $\lambda_2(d) = 5$.

Note que decorre da definição da *i*-aresta-conectividade que $\lambda_{|V|}(v) = \lambda(G)$, para todo $v \in V$. Para valores pequenos de i, a *i*-aresta-conectividade é uma medida mais local do grafo

e se torna global na medida que i se aproxima de $|V|$. A classificação da i-aresta-conectividade de acordo com os conceitos apresentados na Seção 5.1.2 é: (i) os caminhos são disjuntos, (ii) a propriedade de interesse é o volume, (iii) a posição do vértice nos caminhos é radial e (iv) a função de sumarização é o máximo.

Uma possível interpretação da medida apresentada acima é a seguinte. Considere uma rede de computadores ponto-a-ponto onde um conjunto de i servidores devem ser executados em computadores distintos. Caso um dos servidores seja posicionado no nodo v e os demais sejam posicionados de forma a maximizar o número enlaces falhos necessários para separar algum par de servidores do conjunto, então a i-aresta-conectividade corresponde a esse número de enlaces.

A escolha do grau do vértice como valor para $\lambda_1(v)$ é motivada pelas desigualdades apresentadas no lema abaixo:

Lema 5.4. *Seja $G = (V, E, c)$ um grafo (ou grafo orientado) capacitado e $v \in V$. Então,*

$$d(v) = \lambda_1(v) \geq \lambda_2(v) \geq \lambda_3(v) \geq \cdots \geq \lambda_n(v) = \lambda(G).$$

Demonstração. Seja $G = (V, E, c)$ um grafo (ou grafo orientado) capacitado e $v \in V$. Um corte mínimo que separa v de qualquer outro vértice tem capacidade de pelo menos $d(v)$, já que o corte $\{v, V - \{v\}\}$ separa v de qualquer outro vértice. Portanto, $d(v) \geq \lambda_2(v)$.

Considere um índice $k \geq 3$ e um conjunto S tal que $v \in S$, $|S| \geq 3$ e tal que qualquer corte que separa vértices de S tenha capacidade de pelo menos $\lambda_k(v)$. Seja $S' \subset S$ tal que $|S'| = k - 1$ e $v \in S'$. Assim, $\lambda(S') \geq \lambda(S)$, pois cortes que separam vértices de S' também separam vértices de S. Como $\lambda_{k-1}(v) \geq \lambda(S')$ temos que $\lambda_{k-1}(v) \geq \lambda(S) = \lambda_k(v)$. □

O próximo lema mostra como a 2-aresta-conectividade dos vértices pode ser calculada.

Lema 5.5. *Dado um grafo capacitado $G = (V, E, c)$ e um vértice $v \in V$, o valor de $\lambda_2(v)$ é igual à máxima aresta-conectividade local entre v e qualquer outro vértice de V.*

Demonstração. O resultado decorre da definição de 2-aresta-conectividade. Seja $u \in V$, $u \neq v$, tal que $\lambda(v, u)$ é máximo. O conjunto $X = \{v, u\}$ é um conjunto de máxima aresta-conectividade que contém v e cuja cardinalidade é maior do que ou igual a dois. Pela definição de 2-aresta-conectividade, $\lambda_2(v) = \lambda(X) = \lambda(v, u)$. □

Definição 5.6. O *componente de máxima conectividade de índice i*, denotado por $MCC_i(v)$, é o maior conjunto de vértices V' que contém v tal que $\lambda(V') \geq \lambda_i(v)$.

O lema abaixo mostra que o componente de máxima conectividade está bem definido, isto é, o conjunto definido no Lema 5.5 é único.

Lema 5.7. *Seja $G = (V,E,c)$ um grafo capacitado. Se $V' \subseteq V$ é maximal tal que $v \in V'$ e $\lambda(V') \geq \lambda_i(v)$, então V' é único.*

Demonstração. Sejam $V', V'' \subseteq V$, $v \in V' \cap V''$, dois conjuntos maximais com $\lambda(V') \geq \lambda_i(v)$ e $\lambda(V'') \geq \lambda_i(v)$. Suponha que $V' \neq V''$ e, sem perda de generalidade, vamos assumir que $\exists u \in V' \backslash V''$.

Seja $x \in V''$ e considere um x-u-corte mínimo $C = \{X, \overline{X}\}$, com $x \in X$. Se $v \in X$, então C é um u-v-corte e, portanto, $d(X) \geq \lambda_i(v)$. Por outro lado, se $v \notin X$, então C é um x-v-corte e $d(X) \geq \lambda_i(v)$. Podemos concluir que para todo $x \in V''$, qualquer u-x-corte tem capacidade de pelo menos $\lambda_i(v)$. Assim, $\lambda(V'' \cup \{u\}) \geq \lambda_i(v)$ e, portanto, V'' não é maximal, o que é uma contradição. Essa contradição mostra que $V' = V''$. □

Conceitos equivalentes a $\lambda_2(v)$ e $MCC_2(v)$ e algoritmos para computá-los apareceram originalmente em nossos trabalhos [25, 75]. Em [79], um conceito análogo ao do $MCC_2(v)$ é descrito com o nome de componente maximal l-aresta-conectado.

O $MMC_i(v)$ é o conjunto maximal de maior aresta-conectividade que contém v e que possui pelo menos i vértices. Também podemos definir o maior conjunto que contém v com uma dada conectividade k:

Definição 5.8. Seja $G = (V,E,c)$ um grafo (orientado) capacitado, $v \in V$ e $k \in \mathbb{N}$. O *componente máximo de conectividade k*, denotado por $LCC_k(v)$, é o maior conjunto $V' \subseteq V$ tal que $v \in V'$ e $\lambda(V') \geq k$. Se $k > \lambda_2(v)$ então $LCC_k(v) = \{v\}$.

A relação entre $MCC_i(v)$ e $LCC_k(v)$ é dada por:

$$MCC_i(v) = LCC_{\lambda_i(v)}(v).$$

Aresta-conectividade restrita As medidas de conectividade apresentadas até aqui consideram o grafo na sua totalidade. Iremos adicionar à medida um argumento extra que consiste de um subconjunto de vértices. A ideia é restringir a medida a um subconjunto de vértices de interesse. A i-aresta-conectividade restrita é assim definida:

Definição 5.9 (i-aresta-conectividade restrita). Seja $G = (V,E,c)$ um grafo capacitado e $i \in \mathbb{N}$, $i \geq 1$. Seja $V' \subseteq V$ e $v \in V'$.

A i-aresta-conectividade de v restrita a V', denotada por $\lambda_i(v,\ V')$, é definida por:

$$\lambda_i(v,\ V') = \begin{cases} d(v), & \text{se } i = 1, \\ \max\limits_{X \subseteq V'} \lambda_G(X) \mid v \in X, \ \text{ e } |X| \geq i, & \text{se } 2 \leq i \leq |V'|, \\ 0, & \text{caso contrário.} \end{cases}$$

A definição original é equivalente a $\lambda_i(v,\ V)$.

O equivalente ao componente de máxima conectividade é definido assim:

Definição 5.10. O *componente de máxima conectividade restrito a V' de índice i*, denotado por $MCC_i(v,\ V')$, é o maior conjunto de vértices $X \subseteq V'$ que contém v tal que cada par de vértices de X não pode ser separado por um corte com capacidade menor do que $\lambda_i(v,\ V')$.

Uma aplicação das definições acima é a seguinte. Considere uma rede de computadores em que alguns computadores são servidores de processamento (que correspondem ao conjunto V'). Esses servidores querem replicar um processo (ou dados) de forma que as réplicas precisem se comunicar com alta confiabilidade em relação a falhas de enlaces. Se um servidor s precisa fazer i réplicas de um processo, então $\lambda_{i+1}(s,\ V')$ é a melhor confiabilidade que ele pode conseguir. Essa confiabilidade é alcançada desde que o servidor escolha i outros servidores que estejam em $MCC_{i+1}(s, V')$.

A mesma intuição aplicada a cortes de vértices leva à definição de $\kappa_i(v,\ V')$ que deve ser aplicada quando falhas de nodos são mais relevantes do que falhas de enlaces. Ver a Seção 5.2.2 adiante. o

Aresta-Conectividade Média A medida de aresta-conectividade dos vértices apresentada anteriormente é baseada na máxima conectividade de conjuntos que contém o vértice, sendo esse o único critério mensurado. Em particular, o tamanho do conjunto que define a conectividade do vértice não afeta o valor da medida. Apresentamos abaixo uma outra medida, que quantifica a conectividade entre um vértice e todos os demais e utiliza a média das conectividades ao invés da conectividade máxima. Essa medida pode diferenciar vértices que possuem a mesma conectividade máxima.

Definição 5.11 (Aresta-conectividade média). Seja $G = (V, E, c)$ um grafo (ou grafo orientado) capacitado. A aresta-conectividade média de um nodo v é definida por:

$$\overline{\lambda}(v) = \frac{\sum\limits_{u \in V, u \neq v} \lambda(v, u)}{|V|}.$$

Mostramos no próximo capítulo que a aresta-conectividade média dos nodos pode ser computada com a mesma eficiência com que calculamos as outras medidas de aresta-conectividade. A implementação utiliza uma estrutura de dados que chamamos de *árvore de componentes de máxima conectividade* descrita na seção 6.1.1.

5.2.2 Medidas Baseadas em Cortes de Vértices

Nesta seção apresentamos medidas de centralidade de nodos baseadas em vértice-conectividade análogas às medidas baseadas em aresta-conectividade apresentadas na seção 5.2.1[1].

As medidas apresentadas nesta seção seguem a mesma intuição das medidas baseadas em aresta-conectividade, porém se baseiam em cortes de vértices. Elas têm por objetivo atribuir aos nodos valores que determinem a dificuldade de separá-los de outros nodos da rede através da remoção de nodos.

Definimos a seguir a *vértice-conectividade* de um conjunto de vértices. Lembre-se que a vértice-conectividade local entre dois vértices s e t é igual à máxima cardinalidade de um conjunto de s-t-caminhos internamente vértice-disjuntos (cada aresta $\{s,t\}$ conta como um caminho).

Definição 5.12. A vértice-conectividade de um conjunto $X \subseteq V$ é a menor vértice-conectividade local entre qualquer par de vértices em X e é denotada por $\kappa(X)$.

A medida de conectividade de nodos baseada em vértice-conectividade é definida a seguir:

Definição 5.13. Seja G um multigrafo capacitado e $i \in \mathbb{N}$, $2 \leq i \leq |V|$. A *i-vértice-conectividade* de um vértice $v \in V$, denotada por $\kappa_i(v)$, é a maior vértice-conectividade de um conjunto $X \subseteq V$ que satisfaz:

i. $v \in X$, e

ii. $|X| \geq i$

A seguinte desigualdade relaciona a i-aresta-conectividade e a i-vértice-conectividade de um nodo.

Lema 5.14. *Seja um grafo $G = (V,E)$ e $v \in V$, então $\kappa_i(v) \leq \lambda_i(v)$.*

[1] Esta seção é baseada no artigo *"Medidas de conectividade baseadas em cortes de vértices para redes complexas"* de autoria de Karine Pires, Jaime Cohen e Elias P. Duarte Jr. [87]. As medidas aqui apresentadas foram o tema da dissertação de mestrado da Karine Pires [86].

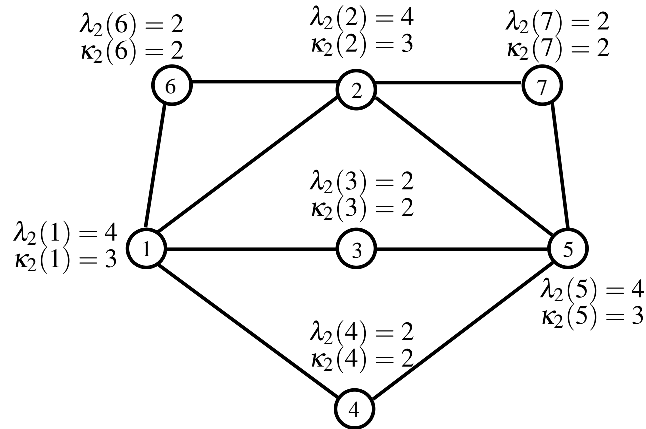


Figura 5.3: Grafo com discrepância entre os valores da aresta-conectividade e da vértice-conectividade.

Algoritmo 5.1 Gera grafo com discrepância entre aresta-conectividade e vértice-conectividade

---

Entrada: $k \in \mathbb{N}$, $k \geq 1$, discrepância desejada
Saída: grafo com discrepância k entre a aresta-conectividade e a vértice-conectividade de pelo menos dois vértices
// *Expandir* a aresta $\{u, v\}$ consiste em acrescentar um vértice k e as arestas $\{u, k\}$ e $\{k, v\}$.
Inicie com o grafo $(\{1,2,3,4,5\}, \{\{1,2\}, \{2,5\}, \{1,3\}, \{3,5\}, \{1,4\}, \{4,5\}\})$
Faça a expansão das arestas do caminho $(1,2,5)$
para i de 2 até k **faça**
    Faça a expansão das arestas do caminho de 1 a 5 criadas no passo anterior

---

Demonstração. A aresta-conectividade é um limitante superior para a vértice-conectividade porque todo caminho vértice-disjunto também é um caminho aresta-disjunto. □

A Figura 5.3 mostra um exemplo de grafo no qual há discrepância entre os valores da vértice-conectividade e da aresta-conectividade. Os vértices 1, 2 e 5 possuem valores com discrepância. Temos que $\lambda_2(1) = 4$ pois a máxima aresta-conectividade dos conjuntos da forma $X = \{1, u\}$, $u \in V - \{1\}$, é 4. Por outro lado, $\kappa_2(1) = 3$ pois a máxima vértice-conectividade dos conjuntos da forma $X = \{1, u\}$, $u \in V - \{1\}$, é 3. Note que o conjunto de vértices $\{2,3,4\}$ separa os vértices 1 e 5 e que o número máximo de caminhos vértice-disjuntos entre 1 e 2 também é 3.

O Algoritmo 5.1 produz grafos com uma discrepância arbitrária entre a vértice--conectividade e a aresta-conectividade. Os grafos gerados são grafos série-paralelos em que o número de caminhos aresta-disjuntos entre os vértices 1 e 5 aumenta enquanto o conjunto $\{2,3,4\} \subseteq V$ permanece sendo um corte de vértices entre 1 e 5. Para discrepância igual a 1, o Algoritmo 5.1 produz o grafo da Figura 5.3 e para discrepância igual a 2, o grafo produzido é o da Figura 5.4.

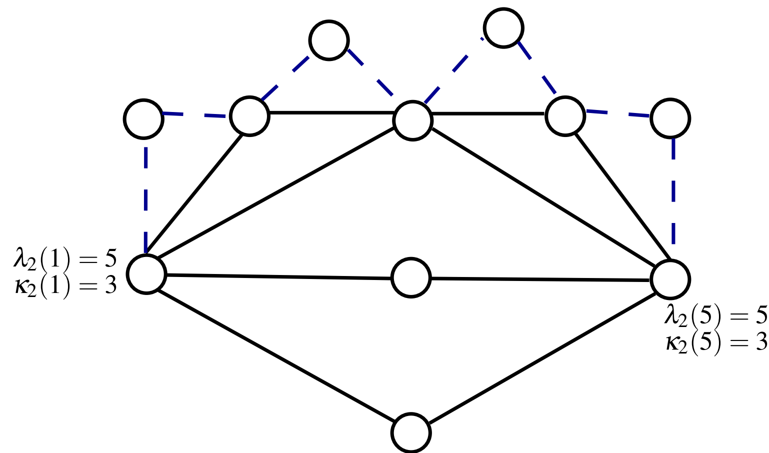


Figura 5.4: Grafo com discrepância entre os valores da aresta-conectividade e da vértice-conectividade produzido pelo Algoritmo 5.1. O caminho tracejado foi produzido na última iteração do algoritmo.

Vértice-Conectividade Baseada na Média Assim como no caso da aresta-conectividade, podemos definir a vértice-conectividade com base na média das conectividades entre o vértice e os demais. Enquanto que a medida baseada na máxima conectividade quantifica a dificuldade de separar o vértice de algum outro vértice, a utilização da média quantifica a conectividade entre o vértice e todos os demais.

Definição 5.15 (Vértice-conectividade média de um nodo). Seja $G = (V, E, c)$ um grafo capacitado. A aresta-conetividade-média de um nodo v é definida por:

$$\overline{\kappa}(v) = \frac{\sum_{u \in V, u \neq v} \kappa(v,u)}{|V| - 1}.$$

5.2.3 Grafos Orientados: Medidas Simétricas e Assimétricas

Em grafos orientados, as definições de $\lambda_i(v)$ e $\kappa_i(v)$ apresentadas anteriormente utilizam uma noção de conectividade que é simétrica. Para $X = \{u, v\}$, as definições 5.1 e 5.12 implicam que $\lambda(X) = \min\{\lambda(u,v), \lambda(v,u)\}$ e $\kappa(X) = \min\{\kappa(u,v), \kappa(v,u)\}$. Portanto, as medidas de conectividade dos vértices quantificam a conectividade do vértice ao seu par mais conexo e vice-versa.

Alternativamente, podemos medir apenas o número de caminhos de um vértice v aos demais vértices do conjunto ou apenas o número de caminhos dos demais vértices até o vértice v:

Definição 5.16. Seja $G = (V, E)$ um grafo orientado ponderado e $v \in V$. Considere um conjunto de vértices $X \subseteq V$, $|X| \geq 2$. A *aresta-conectividade de saída* de v em X, denotada por $\lambda_G^+(v, X)$,

é igual ao mínimo $\lambda(v,u), \forall u \in X - \{v\}$. A *aresta-conectividade de entrada* de v em X, denotada por $\lambda_G^-(v,\ X)$, é igual ao mínimo $\lambda(u,v), \forall u \in X - \{v\}$.

A medida $\lambda_G^+(v,\ X)$ mede o número de caminhos orientados aresta-disjuntos de v a outro vértice $u \in X$, enquanto que $\lambda_G^-(v,\ X)$ mede o número de caminhos orientados aresta-disjuntos de outro vértice de X até v.

A definição acima induz a definição das medidas de vértices $\lambda_G^+(v)$ e $\lambda_G^-(v)$:

Definição 5.17 (**i-aresta-conectividade de saída e de entrada de** v **em** G). Seja $G = (V,E)$ um grafo orientado e $i \in \mathbb{N}$, $i \geq 1$

$$\lambda_i^+(v) = \begin{cases} d^+(v), & \text{se } i = 1, \\ \max\left\{\lambda_G^+(X) \mid v \in X \text{ e } |X| \geq i\right\}, & \text{se } 2 \leq i \leq |V|, \\ 0, & \text{caso contrário.} \end{cases}$$

$$\lambda_i^-(v) = \begin{cases} d^-(v), & \text{se } i = 1, \\ \max\left\{\lambda_G^-(X) \mid v \in X \text{ e } |X| \geq i\right\}, & \text{se } 2 \leq i \leq |V|, \\ 0, & \text{caso contrário.} \end{cases}$$

Definições semelhantes para cortes de vértices, $\kappa_i^+(v)$ e $\kappa_i^-(v)$, podem ser definidas de forma análoga.

No próximo capítulo são apresentados algoritmos para o cálculo das principais medidas de conectividade definidas acima. No Capítulo 7 são apresentados resultados experimentais das medidas de conectividade.

6 Algoritmos para Medidas de Conectividade

Este capítulo trata de algoritmos para calcular as diversas medidas de conectividade definidas no capítulo anterior. A Seção 6.1 trata de algoritmos para o cálculo das medidas baseadas em aresta-conectividade. Todas elas podem ser calculadas a partir de uma árvore de fluxo equivalente ou uma árvore de cortes. O Algoritmo 6.1 mostra como calcular o valor de $\lambda_i(v)$ para um vértice em particular. Na Seção 6.1.1 apresentamos a *árvore de componentes* que é uma estrutura de dados que contém as informações de uma árvore de cortes do grafo e permite o cômputo eficiente de várias medidas de conectividade para todos os vértices do grafo com uma única varredura da estrutura em tempo linear. A Seção 6.2 trata dos algoritmos para as medidas de conectividade baseadas em cortes de vértices.

6.1 Algoritmos para Medidas de Conectividade de Arestas

Esta seção descreve um algoritmo para o cálculo da *i*-aresta-conectividade. Inicialmente, é apresentado um algoritmo para a 2-aresta-conectividade, que é generalizado adiante. De acordo com o Lema 5.5, a 2-aresta-conectividade de um vértice v é igual à maior aresta-conectividade local entre v e algum outro vértice. O algoritmo utiliza uma árvore de fluxo equivalente do grafo, definida na seção 1.1, página 4.

A árvore de fluxo equivalente é uma representação compacta das capacidades dos cortes mínimos que separam todos os pares de vértices do grafo. Neste capítulo mostramos que as medidas de aresta-conectividade definidas no capítulo anterior podem ser computadas eficientemente dada uma árvore de fluxo equivalente do grafo.

Lembre-se que uma árvore de fluxo equivalente $T = (V, E_T, c_T)$ de um grafo $G = (V, E_G, c_G)$ é uma árvore capacitada tal que $\lambda_T(u,v) = \lambda_G(u,v), \forall u, v \in V$. A Figura 6.1 mostra um exemplo de uma árvore de fluxo equivalente de um grafo. Para determinar a capacidade de um corte mínimo entre dois vértices, procuramos pela aresta de menor capacidade no único caminho na árvore que tem esses vértices como extremos.

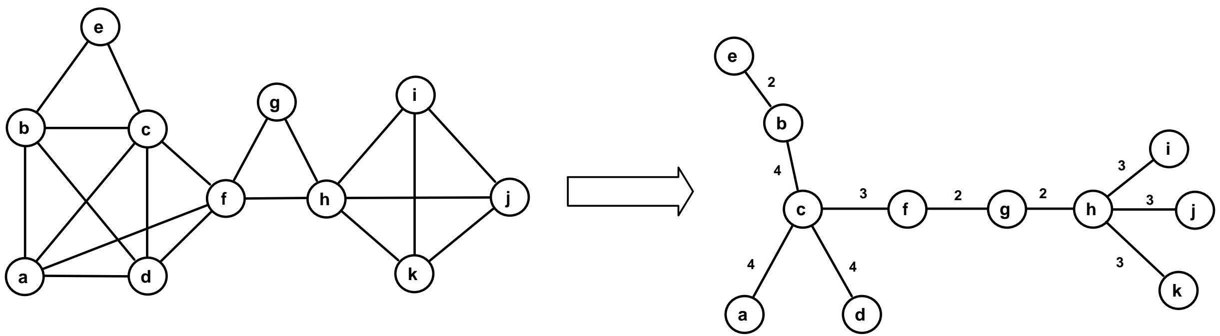


Figura 6.1: Exemplo de árvore de fluxo equivalente de um grafo.

Algoritmos para árvores de fluxo equivalente são discutidos no Capítulo 2. Assumimos aqui que uma árvore de fluxo equivalente foi computada e utilizamos essa árvore para calcular as medidas de 2-aresta-conectividade de acordo com o lema abaixo:

Lema 6.1. *Seja um grafo $G = (V, E)$, uma árvore de fluxo equivalente T de G e $v \in V$. Então, $\lambda_2(v)$ é igual à máxima capacidade de uma aresta incidente a v em T.*

Demonstração. De acordo com o Lema 5.5, $\lambda_2(v)$ é a máxima conectividade local entre v e outro vértice. Pela definição da árvore de fluxo equivalente, a capacidade de um corte mínimo entre v e um vértice u é a capacidade mínima de uma aresta do caminho entre v e u em T. Além disso, esse valor é menor do que ou igual à capacidade da aresta desse caminho incidente a v em T. Logo, a maior capacidade de um corte mínimo que separa v de outro vértice é igual à capacidade de alguma aresta incidente a v em T, em particular, a aresta incidente a v com maior capacidade. □

O Lema 6.1 mostra que $\lambda_2(v)$, $\forall v \in V$, pode ser calculado em tempo $O(|V|)$ a partir da árvore de fluxo equivalente. O algoritmo consiste em inicializar a conectividade de cada vértice, $lambda[v]$, com o valor 0 e percorrer as arestas da árvore de fluxo equivalente, atribuindo a cada $lambda[v]$ a máxima capacidade de uma aresta incidente a v.

Os valores de $\lambda_i(v)$, para algum i, $1 \leq i \leq n$, são calculados pelo algoritmo abaixo:

Algoritmo 6.1 Algoritmo para $\lambda_i(v)$ e $MCC_i(v)$

Entrada: Um grafo $G = (V, E_G, c_G)$, $v \in V$ e um inteiro i, $1 \leq i \leq |V|$

Saída: $\lambda_i(v)$ e $MCC_i(v)$

1: Seja $T = (V, E_T, c_T)$ uma árvore de fluxo equivalente de G;
2: $C \leftarrow \{v\}$
3: $\lambda \leftarrow d(v)$
4: **repita**
5: Seja $e = (x, y) \in E_T$ uma aresta com $x \in C$ e $y \in V \backslash C$ com capacidade máxima;
6: **se** $|C| < i$ **então**
7: $C \leftarrow C \cup \{y\}$
8: **se** $c_T(e) < \lambda$ **então**
9: $\lambda \leftarrow c_T(e)$
10: **senão**
11: **se** $c_T(e) \geq \lambda$ **então**
12: $C \leftarrow C \cup \{y\}$
13: **until** $(|C| \geq i$ e $c_T(e) < \lambda)$ ou $|C| = |V|$
14: **devolva** λ e C

Teorema 6.2. *O Algoritmo 6.1 devolve o valor $\lambda_i(v)$ e o conjunto $MCC_i(v)$.*

Demonstração. Vamos mostrar que, ao término do algoritmo, o conjunto C é tal que $\lambda(C) = \max_{C' \subseteq V}\{\lambda(C') \mid v \in C' \text{ e } |C'| \geq i\}$ e que o valor devolvido pelo algoritmo em λ é $\lambda(C)$.

O Algoritmo 6.1 constrói um conjunto C de vértices e a linha 2 implica que $v \in C$. A condição de parada do laço na linha 13 garante que $|C| \geq i$ ao término do algoritmo.

Os vértices de C pertencem a uma subárvore da árvore de fluxo equivalente T, pois o conjunto C cresce a partir de arestas de T escolhidas na linha 5. Assim, a aresta-conectividade local entre qualquer par de vértices de C é maior do que ou igual ao valor da variável λ que contém a mínima capacidade de uma aresta da subárvore visitada pelo algoritmo.

Seja C_f o conjunto C ao final do algoritmo. Suponha, por contradição, que existe um conjunto $C' \subseteq V$, $|C'| \geq i$, $v \in C'$, tal que $\lambda(C') > \lambda(C_f)$. Seja $u \in C' \backslash C_f$. Considere o caminho P_{uv} de u até v em T. Como $\lambda(u, v) \geq \lambda(C') > \lambda(C_f)$, todas as arestas de P_{uv} têm capacidades maiores do que $\lambda(C_f)$. Considere a aresta e_{min} tal que $c(e_{min}) = \lambda(C_f)$ que foi escolhida pelo algoritmo na linha 5 e o conjunto C imediatamente anterior a essa escolha. Como $u \notin C$, existe uma aresta em P_{uv} com um vértice em C e outro vértice fora de C. A existência dessa aresta contradiz a escolha de e_{min}. Assim, $\lambda = \lambda(C_f) = \max_{C \subseteq V}\{\lambda(C) \mid v \in C \text{ e } |C| \geq i\}$.

O algoritmo devolve $MCC_i(v)$ porque, ao final da execução, $\lambda_G(v,u) < \lambda$ para todo $u \in V \setminus C$ já que pela condição de término do laço, todas as arestas que saem de C têm capacidade menor do que λ. $\square$

Seja $n = |V|$ e $m = |E_G|$. A construção da árvore de fluxo equivalente domina a complexidade de tempo do algoritmo. Dada a árvore, a execução do Algoritmo 6.1 tem tempo de execução $O(n.log\ n)$ desde que a implementação utilize um *heap* como fila de prioridades. Na próxima seção mostramos como calcular de forma eficiente os valores da aresta-conectividade de todos os vértices do grafo.

6.1.1 Árvore de Componentes de Máxima Conectividade

Esta seção descreve uma estrutura de dados associada à árvore de cortes que permite o cálculo eficiente de diversas medidas relacionadas à aresta-conectividade, incluindo os componentes de máxima conectividade e a conectividade média dos vértices.

Seja $G = (V_G, E_G, c_G)$ um grafo capacitado. Considere o conjunto dos componentes de máxima conectividade $MCC_i(v)$ para todo $v \in V_G$. Esse conjunto possui uma estrutura, descrita abaixo, que facilita a sua representação.

Definição 6.3. Uma coleção de conjuntos $\mathscr{L}$ é *laminar* se para qualquer $A, B \in \mathscr{L}$, uma das seguintes três afirmações é verdadeira: $A \cap B = \emptyset$, $A \subseteq B$ ou $B \subseteq A$.

A Figura 6.2 mostra uma representação esquemática de uma coleção laminar de conjuntos.

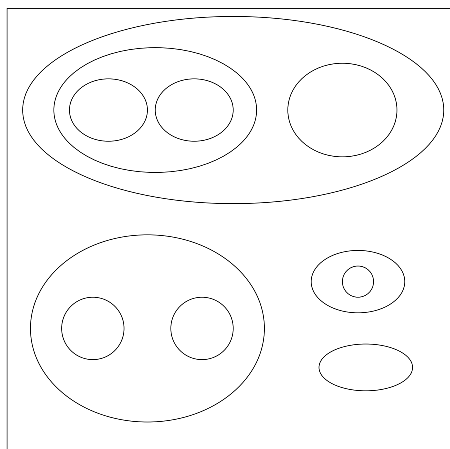


Figura 6.2: Representação de uma coleção laminar de conjuntos.

Para provar que os conjuntos de máxima conectividade formam uma coleção laminar, precisaremos do seguinte lema.

Lema 6.4. *[46] Seja $G = (V,E)$ um grafo capacitado. Então, $\lambda(x,y) \geq \min\{\lambda(x,a),\lambda(a,y)\}$ para quaisquer $x,y,a \in V$, distintos.*

Demonstração. Sejam x,y,a elementos distintos de V. Seja $\{X,\overline{X}\}$ um x-y-corte mínimo. Se $a \in X$, então $\lambda(x,y) = d(X) \geq \lambda(a,y)$. Por outro lado, se $a \in V \backslash X$, então $\lambda(x,y) = d(X) \geq \lambda(x,a)$. Logo, $\lambda(x,y) \geq \min\{\lambda(x,a),\lambda(a,y)\}$. □

Abaixo está o lema sobre a coleção dos conjuntos de máxima conectividade e a sua demonstração:

Lema 6.5. *O conjunto $\{MCC_i(v) : v \in V,\ i \in \mathbb{N},\ 2 \leq i \leq |V|\}$ é laminar.*

Demonstração. Sejam $u,v \in V$. Devemos provar que se $MCC_i(u) \cap MCC_j(v) \neq \emptyset$ então ou $MCC_i(u) \subseteq MCC_j(v)$ ou $MCC_i(u) \supseteq MCC_j(v)$. Vamos supor que $MCC_i(u) \cap MCC_j(v) \neq \emptyset$. Sem perda de generalidade, assumimos que $\lambda_i(u) \geq \lambda_j(v)$. Seja $a \in MCC_i(u) \cap MCC_j(v)$. Vamos mostrar que $MCC_i(u) \subseteq MCC_j(v)$. Seja $y \in MCC_i(u) \backslash \{a\}$. Vamos mostrar que $y \in MCC_j(v)$, ou seja, que $\lambda(x,y) \geq \lambda_j(v), \forall x \in MCC_j(v)$. Seja $x \in MCC_j(v)$. Se $x = a$, então $\lambda(x,y) \geq \lambda_i(u) \geq \lambda_j(v)$. Se $x \neq a$, então, pelo lema 6.4, $\lambda(x,y) \geq \min\{\lambda(x,a),\lambda(a,y\} \geq \min\{\lambda_j(v),\lambda_i(u)\} = \lambda_j(v)$. Logo, $\lambda(x,y) \geq \lambda_j(v)$. Portanto, $y \in MCC_j(v)$. □

As famílias laminares possuem uma estrutura hierárquica e podem ser representadas por árvores. No texto a seguir, o termo *vértice* sempre se refere a elementos do grafo de entrada e o termo *nodo* se refere a elementos da árvore associada.

Definição 6.6 (Árvore de Componentes de Máxima Conectividade). Seja $G = (V,E)$ um grafo capacitado. Uma *árvore de componentes de máxima conectividade* de G, denotada por $\mathscr{T}_G$, é construída a partir de uma árvore de cortes de G. A árvore $\mathscr{T}_G$ contém $2|V|-1$ nodos. Cada folha de $\mathscr{T}_G$ *representa* um vértice de V. Cada nodo interno de $\mathscr{T}_G$ *representa* o conjunto de vértices associados às folhas da subárvore que o tem como raiz. Os vértices de V representados por um nodo u de $\mathscr{T}_G$ são denotados por $\mathscr{T}_G(u)$. A conectividade de um nodo interno u de $\mathscr{T}_G$, denotada por $\lambda_{\mathscr{T}_G}(u)$, é igual a $\lambda_G(\mathscr{T}_G(u))$. Além disso, para todo $v \in V$ e para todo i, $2 \leq i \leq |V|$, existe um nodo u de $\mathscr{T}_G$, tal que $\mathscr{T}_G(u) = MCC_i(v)$.

Um exemplo de árvore de componentes de máxima conectividade pode ser visto na Figura 6.3. Os nodos escuros dessa árvore correspondem a nodos u tal que $\mathscr{T}_G(u)$ é um componente de máxima conectividade.

A árvore $\mathscr{T}_G$ permite determinar de forma eficiente todos os $MCC_i(v)$, para quaisquer valores de i e v. Também, a partir de $\mathscr{T}_G$, podemos calcular todos os $\lambda_i(v)$ com uma única

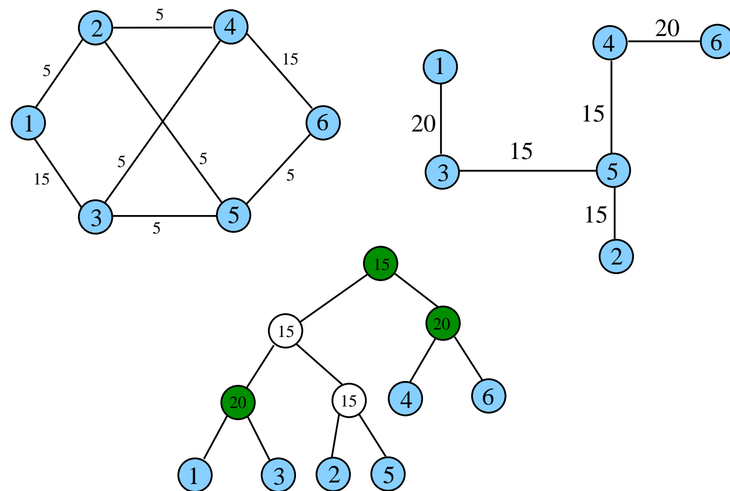


Figura 6.3: Exemplo de grafo, uma árvore de cortes e uma árvore de componentes de máxima conectividade desse grafo. Os nodos escuros correspondem a algum componente de máxima conectividade.

varredura de $\mathscr{T}_G$. Os valores da aresta-conectividade média de todos os vértices também podem ser calculados em tempo linear com uma única varredura de $\mathscr{T}_G$.

Algoritmo para Construção de $\mathscr{T}_G$

A estrutura de dados utilizada para representar a árvore $\mathscr{T}_G$ é uma árvore binária em que cada nodo u de $\mathscr{T}_G$ possui os seguintes atributos:

vérticeID Se u é uma folha de $\mathscr{T}_G$ então vérticeID é o índice do vértice de G representado por u. Caso contrário, vértice ID é igual a -1.

lambda é a aresta-conectividade de $\mathscr{T}_G(u)$ em G, isto é, $\lambda_G(\mathscr{T}_G(u))$. As folhas têm *lambda* igual a -1.

esq é o filho esquerdo do nodo u.

dir é o filho direito do nodo u.

numVértices é o número de folhas abaixo do nodo u.

Sejam $l_1, l_2, \ldots, l_{n-1}$ as capacidades das arestas de uma árvore de cortes de G tal que $l_1 \geq l_2 \geq \cdots \geq l_{n-1}$.

O Algoritmo 6.2 constrói a árvore de componentes de máxima conectividade. A árvore $\mathscr{T}_G$ é construída das folhas em direção à raiz. As linhas 1-5 criam uma folha da árvore para cada vértice do grafo e inicializam os campos *vérticeID* com os índices de cada vértice. O laço que se inicia na linha 6 percorre as arestas da árvore de cortes em ordem não-crescente de

capacidades. A próxima aresta é denotada por $e = \{v_1, v_2\}$ e tem capacidade l_i. Na linha 10 um novo nodo interno, *raiz*, é criado, que terá como filhos as duas subárvores que contém os nodos que representam os vértices v_1 e v_2. O campo *raiz.lambda* recebe o valor l_i que corresponde a aresta-conectividade no componente $\mathscr{T}_G(raiz)$. Assim, os componentes de maior conectividade são representados por subárvores de $\mathscr{T}_G$ e a aresta-conectividade do grafo é representada na raiz da árvore $\mathscr{T}_G$. Veja o exemplo da Figura 6.3.

Algoritmo 6.2 Constrói $\mathscr{T}_G$.

Entrada: Grafo $G = (V_G, E_G, c_G)$ e uma árvore de cortes $T = (V, E_T, c_T)$ de G.
Saída: Árvore de componentes de máxima conectividade $\mathscr{T}_G$

```
    // cada vértice do grafo corresponde a uma folha da árvore 𝒯_G
 1: para i ← 1 até |V| faça
 2:     u ← novo_nodo()
 3:     u.verticeID ← i
 4:     u.numVertices ← 1
 5:     u.lambda = −1
 6: para i ← 1 até |V| − 1 faça
 7:     e ← {v1, v2} tal que c(e) = l_i, onde e é a i-ésima aresta de maior capacidade em T
 8:     raiz1 ← raiz da subárvore que contém v1
 9:     raiz2 ← raiz da subárvore que contém v2
        // as subárvores contendo v1 e v2 são unidas por uma nova raiz
10:     raiz = novo_nodo()
11:     raiz.lambda = l_i
12:     raiz.verticeID = −1
13:     raiz.esq = raiz1;   raiz.dir = raiz2
14:     raiz.numVertices = raiz1.numVertices + raiz2.numVertices
15: devolva raiz      // a raiz de 𝒯_G
```

A árvore de componentes $\mathscr{T}_G$ representa a estrutura de aresta-conectividade do grafo e permite o cálculo de medidas de aresta-conectividade de todos os vértices com uma única varredura de $\mathscr{T}_G$ como mostrado a seguir.

Aresta-Conectividade de Todos os Vértices O Algoritmo 6.3 calcula a aresta-conectividade de índice i, $\lambda_i(v)$, de todos os vértices do grafo. Seja o *nível* de um nodo igual à distância da raiz até ele. Então, dada uma folha v de $\mathscr{T}_G$, $\lambda_i(v.v\acute{e}rticeID)$ é a conectividade $\lambda_{\mathscr{T}_G}(u)$ do nodo u ancestral de v com maior nível tal que $u.numV\acute{e}rtices \geq i$.

O Algoritmo 6.3 percorre recursivamente $\mathscr{T}_G$ a partir da raiz e mantém o valor da maior conectividade de um nodo cuja subárvore contém pelo menos i folhas. Quando uma folha é alcançada, esse é o valor da aresta-conectividade de índice i do vértice representado pela folha. A chamada inicial do algoritmo é LAMBDA$(\mathscr{T}_G.raiz, 0, i)$.

Algoritmo 6.3 Aresta-Conectividade, $\lambda_i(v), \forall v \in V$

Procedimento `Lambda`(*nodo*, *lambda*, *i*)
Entrada: Um grafo $G = (V, E, c_G)$, uma árvore de componentes $\mathscr{T}_G$ e um índice i
Saída: Vetor $\lambda_i : V \to \mathbb{R}$ com a aresta-conectividade $\lambda_i(v)$ dos vértices

se $nodo.verticeID \neq -1$ **então** // *nodo* é uma folha
  $\lambda_i[nodo.verticeID] \leftarrow lambda$
senão
  se $nodo.numVértices \geq i$ **então**
    $lambda \leftarrow nodo.lambda$
  `Lambda`(*nodo.esq*, *lambda*, *i*)
  `Lambda`(*nodo.dir*, *lambda*, *i*)

Aresta-Conectividade Média O Algoritmo 6.4 calcula a aresta-conectividade média, $\overline{\lambda}(v)$, de todos os nodos do grafo. Seja v uma folha de $\mathscr{T}_G$. Sejam $n_1, n_2, \ldots n_k$ os nodos internos no caminho da raiz até v e tais que n_i é o pai de n_{i+1}, para $1 \leq i \leq k-1$ e n_k é o pai de v. Seja s_i, $1 \leq i \leq k$, o número de folhas na subárvore ligada a n_i que não contém n_{i+1}. Note que $\lambda_{\mathscr{T}_G}(n_i)$ é a aresta-conectividade local entre v e cada vértice representado na subárvore de n_i que não contém n_{i+1}. Portanto, a aresta-conectividade média de v é dada por

$$\overline{\lambda}(v) = \frac{\sum_{i=1}^{k} \lambda_{\mathscr{T}_G}(n_i) \times s_i}{|V| - 1}.$$

O Algoritmo 6.4 percorre a árvore $\mathscr{T}_G$ recursivamente e mantém uma soma ponderada das conectividades. Ao alcançar uma folha, o valor da média do vértice correspondente é calculado. A chamada inicial do procedimento recursivo deve ser `CM`$(\mathscr{T}_G.raiz, 0)$.

Algoritmo 6.4 Conectividade Média, $\overline{\lambda}()$ de todos os nodos

---

Procedimento `CM`$(nodo, soma_ponderada)$
Entrada: Um grafo $G = (V, E, c_G)$ e uma árvore de componentes $\mathscr{T}_G$
Saída: Vetor $\overline{\lambda}$ com a aresta-conectividade média dos vértices

se $nodo.verticeID \neq -1$ **então** // *nodo* é uma folha
  $\overline{\lambda}[nodo.verticeID] \leftarrow soma_ponderada/(|V| - 1)$
senão
  `CM`$(nodo.esq, soma_ponderada + nodo.lambda * nodo.dir.numVertices)$
  `CM`$(nodo.dir, soma_ponderada + nodo.lambda * nodo.esq.numVertices)$

Consultas em Tempo Constante A árvore de componentes de máxima conectividade também permite consultas sobre a aresta-conectividade local entre dois vértices em tempo $O(1)$ sem a necessidade de armazenar $O(|V|^2)$ valores que uma matriz com todas as conectividades ocuparia. Dadas duas folhas u e v de $\mathcal{T}_G$, a conectividade local entre os vértices *u.numVértice* e *v.numVértice* de G é igual a conectividade do nodo que é o ancestral comum mais baixo de u e v em $\mathcal{T}_G$. O problema do ancestral comum mais baixo, conhecido por LCA (do inglês *least common ancestor*), é bem conhecido e pode ser resolvido com tempo de pré-processamento $O(n)$ e tempo de consulta constante [53, 39].

6.2 Algoritmos para Medidas de Conectividade de Vértices

Esta seção apresenta um algoritmo para calcular as medidas de conectividade baseadas em cortes de vértices, $\kappa_i(v)$. Ao contrário do que ocorre com a aresta-conectividade, as vértice-conectividades locais entre todos os pares de vértices de um grafo não podem ser representadas por uma árvore, já que o número de valores distintos de vértice-conectividades locais pode ser $O(n^2)$ [10]. Assim, por exemplo, o cálculo de $\kappa_2(v)$ requer o cômputo das vértice-conectividades locais entre v e todos os demais vértices do grafo. Como $\kappa_2(v) \leq \lambda_2(v)$, para todo $v \in V$, podemos interromper o cômputo caso um vértice u com $\kappa_2(v,u) = \lambda_2(v)$ seja encontrado.

Ao invés de resolver diretamente o problema dos cortes de vértices, utilizamos uma estratégia descrita em [79] que consiste em reduzir o problema dos cortes de vértices ao problema dos cortes de arestas. Essa redução se aplica a grafos orientados, portanto os grafos não-orientados são, primeiramente, transformados em grafos orientados. Um grafo não orientado pode ser transformado em um grafo orientado através da duplicação das arestas do grafo original, fazendo com que uma aresta $\{a,b\}$ corresponda a duas arestas direcionadas (a,b) e (b,a) no novo grafo.

Para encontrar os cortes de vértices do grafo orientado, criamos um novo grafo, de forma que os cortes de arestas mínimos desse grafo correspondem aos cortes de vértices mínimos do grafo original. Veja a Figura 6.4. Para cada vértice v do grafo original, o grafo transformado contém dois vértices, v' e v''. As arestas com origem no vértice v do grafo original têm como origem no novo grafo o vértice v''. As arestas com destino no vértice v no grafo original têm como destino v' no grafo transformado. Por fim, adicionamos arestas direcionadas (v',v'') entre todos os pares de vértices associados. As arestas do tipo (v',v'') recebem capacidades de valor 1 e as demais arestas recebem capacidades $|V|-1$. Essas capacidades garantem

que os cortes de arestas de capacidade mínima utilizem somente as arestas que correspondem aos vértices do grafo original, o que justifica a redução entre os problemas.

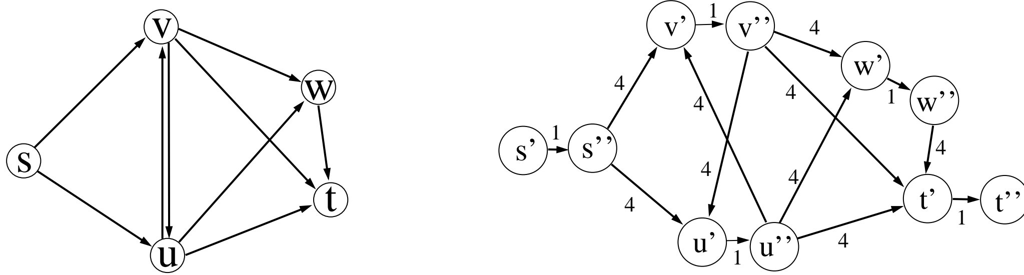


Figura 6.4: Exemplo de transformação de grafo orientado para resolver o problema do s-t-corte de vértices mínimo [79].

No grafo transformado, um corte de arestas mínimo entre os vértices s'' e t' corresponde a um s-t-corte de vértices mínimo do grafo original. Para que a transformação corretamente encontre a vértice-conectividade entre s e t, que foi definida em termos de caminhos vértice-disjuntos entre s e t, ainda é necessário tratar o caso em que a aresta $(s,t) \in E_G$. Nesse caso, a aresta (s'',t') recebe capacidade 1 para que seja incluída no corte mínimo. Assim, o valor do fluxo máximo entre s'' e t' no grafo transformado corresponde à vértice-conectividade entre s e t no grafo original.

Para calcular $\kappa_2(v)$ utilizamos a transformação descrita acima entre v e todos os demais vértices do grafo. A máxima vértice-conectividade encontrada é o valor $\kappa_2(v)$. Seja $n = |V_G|$ e $m = |E_G|$. A complexidade de tempo do algoritmo para o cálculo da 2-vértice-conectividade requer o cálculo de $n-1$ s-t-cortes mínimo (ou fluxos máximos) para cada vértice. O cálculo do fluxo máximo é facilitado pela estrutura do grafo transformado que possui vértices que são ou o destino ou a origem de apenas uma aresta [79]. Nesse caso, o algoritmo de Y. Dinitz [32] tem complexidade de tempo de $O(m\sqrt{n})$. Logo, a complexidade de tempo para o cálculo de $\kappa_2(v)$ é $O(nm\sqrt{n})$.

Os valores de $\kappa_i(v)$, para $i > 2$, podem ser calculados pela enumeração de conjuntos de vértices de cardinalidade i, porém tal abordagem não é eficiente, pois tem complexidade exponencial em função de i. Nós não conhecemos algoritmos eficientes para calcular $\kappa_i(v)$ em geral, ao contrário do que ocorre com $\lambda_i(v)$ que pode ser calculado em tempo polinomial independentemente do valor de i.

7 Avaliação Empírica das Medidas de Conectividade

Neste capítulo apresentamos resultados experimentais utilizando as medidas de conectividade em redes complexas. O objetivo é comparar as medidas de conectividade com outras medidas de centralidade em redes complexas reais e sintéticos. As comparações entre as medidas servem para detectar correlações entre elas e determinar as suas diferenças e semelhanças.

Mostramos, por exemplo, que apesar da forte correlação nas ordenações de todos os nodos de redes complexas utilizando as medidas de conectividade e o grau, existem casos em que os nodos mais bem conectados da rede não são necessariamente os nodos de maior grau. Também mostramos que os nodos com maior conectividade tendem a ter baixa excentricidade e que a recíproca não se verifica.

O capítulo está organizado da seguinte maneira. A Seção 7.1 apresenta estatísticas sobre a conectividade de 10 redes complexas. A Seção 7.2 traz comparações entre medidas de centralidade de redes sintéticas do modelo GLP (*Generalized Linear Preference*). A Seção 7.3 apresenta o resultado de experimentos com as medidas de conectividade baseadas em cortes de vértices. A Seção 7.4 conclui o capítulo.

7.1 Conectividade de Redes Complexas

As 10 redes complexas utilizadas nos experimentos pertencem a diferentes domínios e foram obtidas de diferentes fontes: *CA-AstroPh, CA-CondMat, CA-HepPh, CA-GrQc* [63] and *geocomp* [9] são redes de colaboração científica, *powergrid* [105] é uma rede de distribuição de energia elétrica, p2p-Gnutella04 e p2p-Gnutella25 [63] são redes P2P (*peer-to-peer*), yeast [17] é uma rede de interações de proteínas e *USAir97* representa os aeroportos dos EUA. A Tabela 7.1 mostra os tamanhos das redes. Apenas os maiores componentes conexos das redes originais foram utilizados em nossas análises.

Tabela 7.1: Tamanhos das redes complexas utilizadas nos experimentos.

Rede	$\|V\|$	$\|E\|$
CA-AstroPh	17903	393944
CA-CondMat	21363	182572
CA-GrQc	4158	26844
CA-HepPh	11204	235238
geocomp	3621	9461
p2p-Gnut04	10876	39994
p2p-Gnut25	22663	54693
yeast	2221	6602
powergrid	4941	6594
USAir97	332	2126

Tabela 7.2: Comparação dos graus com a 2-aresta-conectividade e a 3-aresta-conectividade de 10 redes complexas. A tabela apresenta os valores mínimos, máximos e médios.

	Mínimo			Máximo			Média		
Rede	Grau	$\lambda_2(v)$	$\lambda_3(v)$	Grau	$\lambda_2(v)$	$\lambda_3(v)$	Grau	$\lambda_2(v)$	$\lambda_3(v)$
CA-AstroPh	2	2	2	1008	854	838	44	43,8	43,7
CA-CondMat	2	2	2	558	504	394	17,1	16,6	16,6
CA-GrQc	2	2	2	162	154	150	12,9	12,1	12,1
CA-HepPh	2	2	2	982	972	964	42	41,6	41,5
geocomp2	1	1	1	102	94	86	5,2	4,7	4,7
p2p-Gnutella25	1	1	1	66	57	45	4,8	4,4	4,4
p2p-Gnutella04	1	1	1	103	81	65	7,4	7,1	7,1
yeast	1	1	1	64	59	55	5,9	5,6	5,6
powergrid	1	1	1	19	12	11	2,7	2,3	2,3
USAir97	1	1	1	139	104	99	12,8	12,4	12,4

A Tabela 7.2 mostra algumas estatísticas das redes complexas usadas nos experimentos. São apresentados os mínimos, os máximos e as médias das medidas grau, 2-aresta-conectividade e 3-aresta-conectividade. Algumas considerações podem ser enumeradas acerca desses dados:

1. Existe uma lacuna entre os valores do grau máximo e das medidas de conectividade máximas. Em média, a diferença entre o grau máximo e a 2-aresta-conectividade máxima é de 13% do valor do maior grau. Na rede powergrid essa diferença é de 37%.

2. Em algumas redes, a diferença entre a 2-aresta-conectividade máxima e a 3-aresta-conectividade máxima é significativa, passando de 20% da 2-aresta-conectividade máxima nas redes CA-CondMat, p2p-Gnutella25 e p2p-Gnutella04. Isso mostra que, nessas redes, o par de nodos mais bem conectado tem a conectividade significativamente maior do que a tripla de nodos mais bem conectada.

Classificação de Nodos de Redes Complexas

Medidas de nodos são frequentemente utilizadas para classificar os nodos das redes. No caso da conectividade o objetivo é determinar quais são os nodos mais bem conectados da rede. O próximo experimento teve como objetivo quantificar as diferenças entre as ordenações dos nodos por ordem de grau e por ordem da i-aresta-conectividade. Uma correlação de ordem dessas ordenações para os primeiros 10 e primeiros 100 vértices da ordenação por grau é apresentada. Utilizamos a correlação de ordem de *Kendall* que produz o valor +1 para sequências iguais e o valor -1 para sequências em ordem inversa.

Os resultados são mostrados na Tabela 7.3. As redes com a maior correlação entre grau e 2-aresta-conectividade são as 4 primeiras redes de colaboração. As redes que apresentam os menores valores de correlação são *powergrid* e *p2p-gnuttella25* com correlações mínimas de 0.2 e 0.54, respectivamente. Gráficos de dispersão comparando o grau e a 2-aresta-conectividade das redes *powergrid* e *p2p-Gnutella25* aparecem nas Figuras 7.1 e 7.2, respectivamente.

7.2 Conectividade de Redes Sintéticas

Os próximos experimentos foram feitos em uma rede sintética conhecida por GLP (*Generalized Linear Preference*) que é uma generalização do modelo de Barabási-Albert proposta por R. Bu e D. Towsley [18]. Nesse modelo, da mesma maneira que no modelo BA, o grafo é

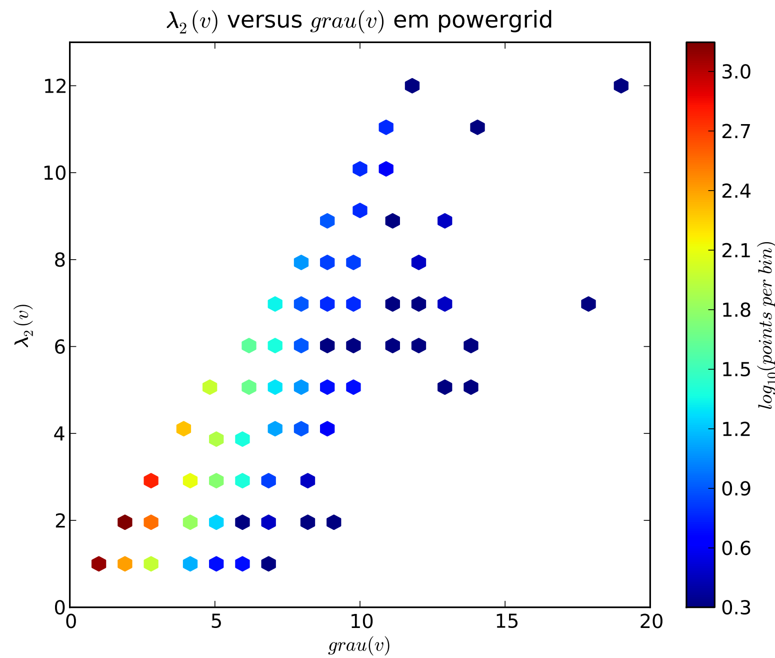


Figura 7.1: Grau versus 2-aresta-conectividade da rede powergrid.

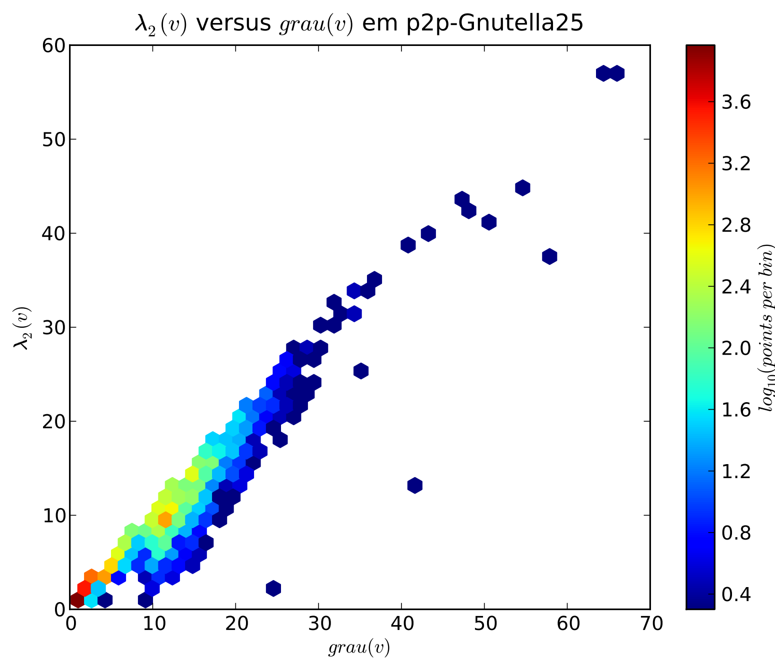


Figura 7.2: Grau versus 2-aresta-conectividade da rede p2p-Gnutella25.

Tabela 7.3: Correlação de ordem entre as ordenações pelo grau e pela 2-conectividade dos 10 e 100 nodos de maior grau.

	Correlação de Ordem			
	Maiores 10		Maiores 100	
Rede	$\lambda_2(v)$	$\lambda_3(v)$	$\lambda_2(v)$	$\lambda_3(v)$
CA-AstroPh	0,99	0,97	1[a]	1[a]
CA-CondMat	0,98	0,95	0,94	0,94
CA-GrQc	0,81	0,83	0,94	0,93
CA-HepPh	0,91	0,88	0,99	0,99
geocomp2	0,75	0,72	0,82	0,82
p2p-Gnutella25	0,54	0,51	0,64	0,64
p2p-Gnutella04	0,78	0,72	0,92	0,92
yeast	0,8	0,77	0,85	0,85
powergrid	0,2	0,14	0,35	0,33
USAir97	0,98	0,95	0,97	0,97

[a]Valores arredondados para 1.

produzido por um processo iterativo. A diferença é que nele as rodadas podem adicionar novas arestas entre vértices existentes. Especificamente, o grafo inicia com m_0 vértices conectados por $m_0 - 1$ arestas. A cada rodada e com probabilidade $\rho \in [0,1]$, são adicionadas $m < m_0$ arestas entre vértices existentes, ou, com probabilidade $1-\rho$, um novo vértice é adicionado e conectado a m vértices existentes. As pontas das arestas são sempre escolhidas usando o modelo de preferência linear generalizada dado por

$$\Pi(v) = \frac{d(v) - \beta}{\sum_j (d(j) - \beta)}$$

onde β é uma constante no intervalo $(-\infty, 1)$. Os parâmetros utilizados foram os propostos em [18] para modelar a Internet em nível de sistemas autônomos que são $\rho = 0.4695$ e $\beta = 0.6447$. O modelo GLP produz grafos cujas distribuições dos graus respeitam uma lei de potência.

Experimentos foram feitos com o objetivo de comparar as medidas de conectividade com outras medidas de centralidade. Foram comparados o grau dos vértices, a 2-aresta-conectividade, a 3-aresta-conectividade, a excentricidade e a excentricidade efetiva. Os resultados são médias obtidas de 100 instâncias de grafos com 1000 vértices.

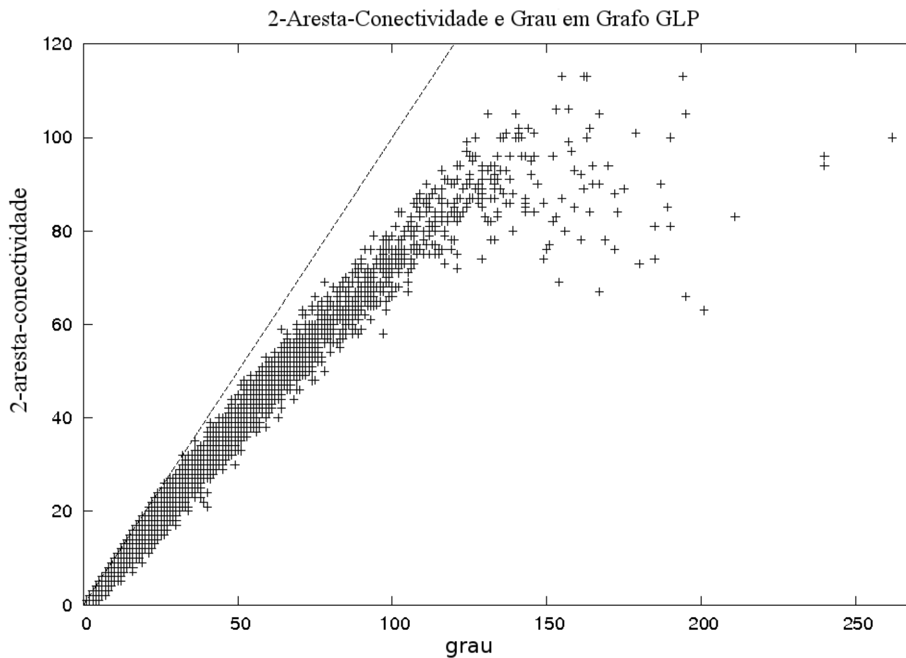


Figura 7.3: Comparação entre grau e 2-aresta-conectividade.

Grau e 2-Aresta-Conectividade O gráfico da comparação do grau dos vértices e da 2-aresta-conectividade pode ser visto na Figura 7.3[1]. A linha diagonal corresponde à função identidade e ajuda a perceber a correlação entre as duas medidas. Pontos no gráfico de dispersão podem corresponder a várias ocorrências.

Para vértices de grau mais alto, a diferença entre o grau e a 2-aresta-conectividade é maior e a correlação entre as duas medidas diminui. Além disso, os vértices com maior grau não são necessariamente aqueles com a maior 2-aresta-conectividade. Por exemplo, um vértice de grau 262 tem 2-aresta-conectividade igual a 100 enquanto que um vértice do mesmo grafo com grau 140 tem 2-aresta-conectividade maior do que 100.

Excentricidade e 2-Aresta-Conectividade A excentricidade de um vértice v é a maior distância entre v e qualquer outro vértice do grafo. Vértices com baixa excentricidade são mais centrais e vértices com alta excentricidade são periféricos. A excentricidade efetiva de um vértice v é uma variação mais robusta da excentricidade e é definida como a mínima distância h tal que pelos menos 90% dos vértices estão a uma distância menor do que ou igual a h.

As Figuras 7.4 e 7.5 mostram os gráficos da comparação da 2-aresta-conectividade com a excentricidade e a excentricidade efetiva dos vértices de um grafo GLP com 1000 vértices. Os resultados mostram que vértices com alta conectividade são centrais, isto é, têm baixa

[1] A notação $\#C_2(v)$ que aparece nos gráficos é equivalente a $\lambda_2(v)$ e foi utilizada em [26] onde essas figuras foram publicadas originalmente.

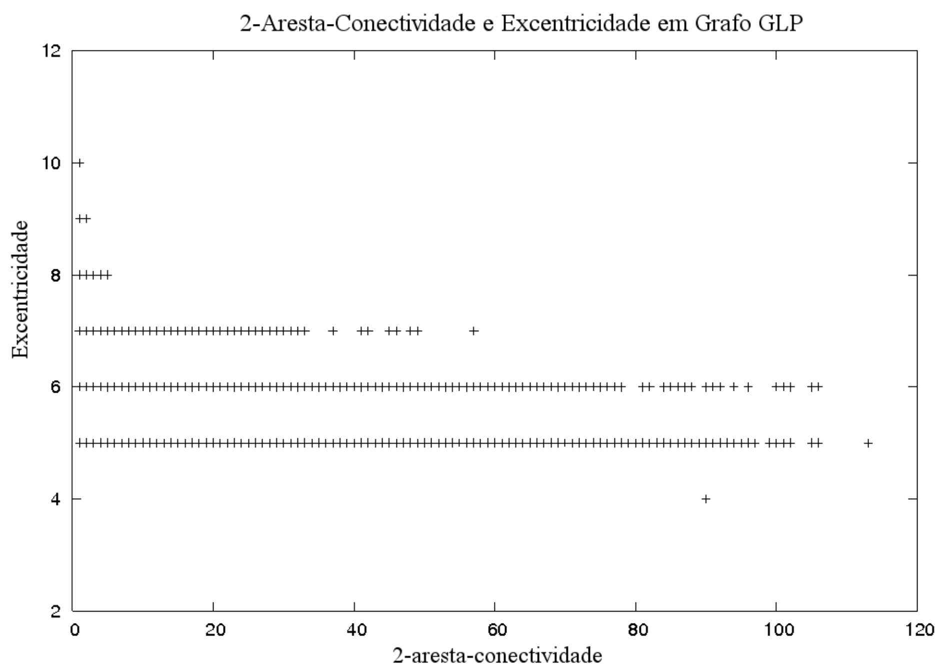


Figura 7.4: Comparação entre excentricidade e 2-aresta-conectividade.

excentricidade e vértices com alta excentricidade tendem a ter baixa conectividade. Além disso, a existência de muitos nodos com baixa conectividade e baixa excentricidade mostra que, em grafos GLP, nodos centrais pelo critério de distância podem não ser importantes quando a conectividade é utilizada como critério de ordenação. Os resultados para a excentricidade efetiva são similares.

Tamanhos dos componentes de máxima conectividade $MCC_2(v)$ À medida que a conectividade aumenta, os tamanhos dos componentes de máxima conectividade tendem a diminuir. A Figura 7.6 mostra a distribuição dos tamanhos dos componentes de máxima conectividade dos vértices ordenados pela 2-aresta-conectividade de um grafo GLP.

2-Aresta-Conectividade e 3-Aresta-Conectividade Uma comparação entre a 2-aresta-conectividade e a 3-aresta-conectividade mostra a redução de conectividade dos conjuntos de maior conectividade que contêm 2 e 3 vértices.

A Figura 7.7 mostra a relação entre a 2-aresta-conectividade e a 3-aresta-conectividade dos vértices de grafos GLP com 500 e 1000 vértices, respectivamente. No gráfico, pontos sobre a diagonal correspondem a vértices cuja medida de conectividade não é alterada quando o índice vai de 2 para 3.

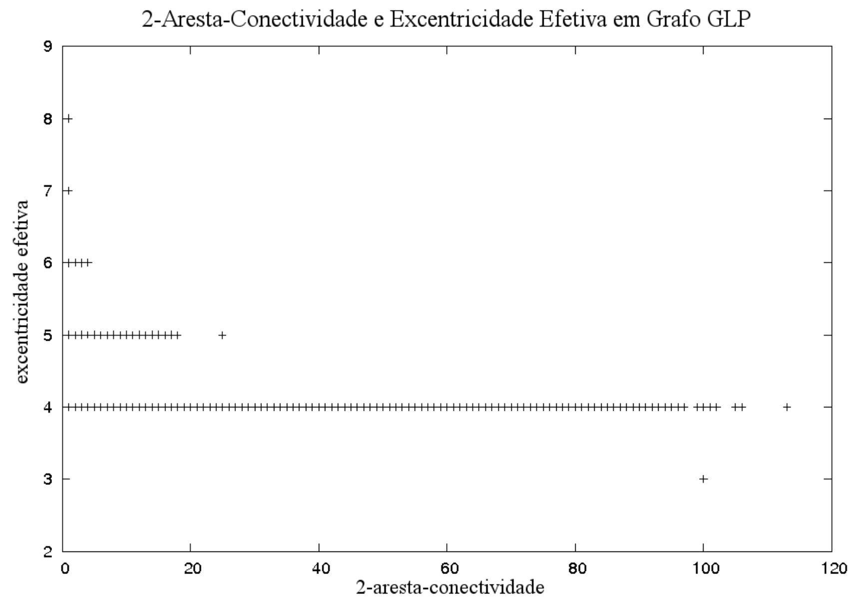


Figura 7.5: Comparação entre a excentricidade efetiva e a 2-aresta-conectividade.

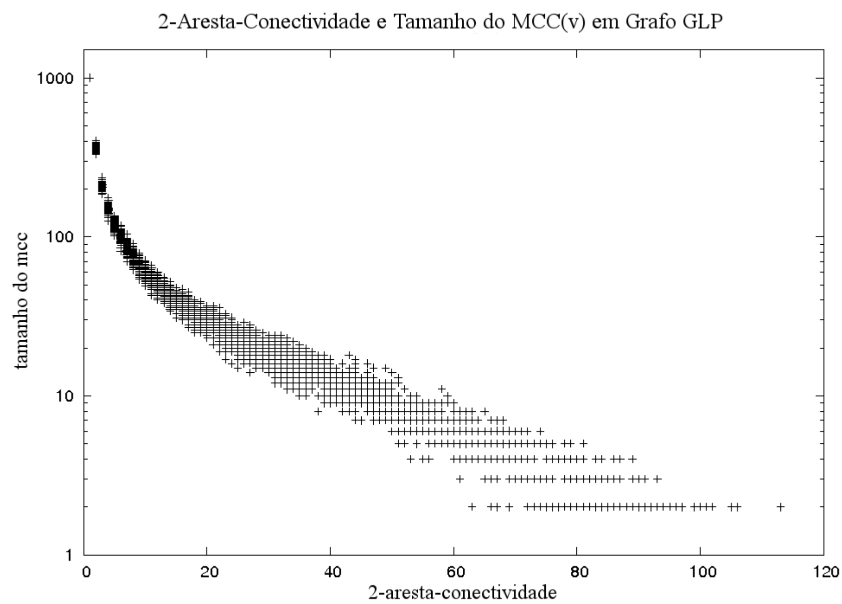
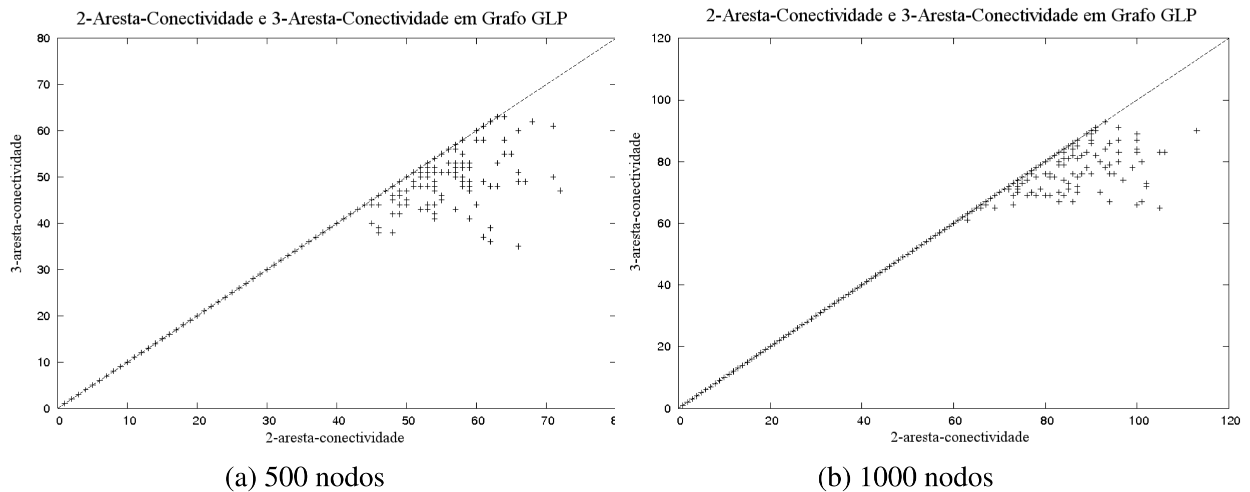


Figura 7.6: Comparação entre a 2-aresta-conectividade e o tamanho dos componentes de máxima conectividade de grafos GLP com 1000 nodos.



(a) 500 nodos (b) 1000 nodos

Figura 7.7: Comparação entre a 2-aresta-conectividade e a 3-aresta-conectividade em grafos GLP com 500 e 1000 nodos.

Remoção sequencial de vértices A robustez de uma rede pode ser medida através de métricas de conectividade. Por exemplo, pode-se medir a conectividade da rede à medida que nodos são removidos. Exemplos de métricas de conectividade utilizadas para isso são: (*i*) o número de pares de vértices que estão conectados por algum caminho, (*ii*) o tamanho do maior componente conexo e (*iii*) o inverso do número de componentes conexos.

A remoção de nodos de uma rede pode ser feita de forma aleatória ou seguir uma estratégia. Por exemplo, falhas de roteadores de uma rede de computadores podem ocorrer de forma aleatória, porém ataques contra a infraestrutura podem se dirigir aos nodos cuja remoção provoque o maior dano. Nodos cuja remoção provocam uma maior redução da conectividade da rede podem ser considerados mais centrais ou mais importantes. As redes cujos graus dos vértices seguem uma distribuição de lei de potência são vulneráveis à remoção dos vértices de maior grau [72].

Um experimento para comparar as medidas de centralidade aplicadas na escolha dos nodos removidos da rede foi feito. A conectividade das redes foi medida pelo número de pares de vértices conectados por algum caminho. Nesse experimento, os nodos foram removidos de acordo com as medidas de centralidade calculadas no grafo inicial. A escolha dos nodos foi feita por ordem aleatória, por ordem decrescente em relação à 2-aresta-conectividade e ao grau e por ordem crescente em relação à excentricidade e à excentricidade efetiva.

A Figura 7.8 mostra os resultados do experimento. O gráfico mostra a relação entre o número de pares de vértices conectados e o número de vértices removidos pelos vários critérios. Ao final, 20% dos vértices do grafo foram removidos.

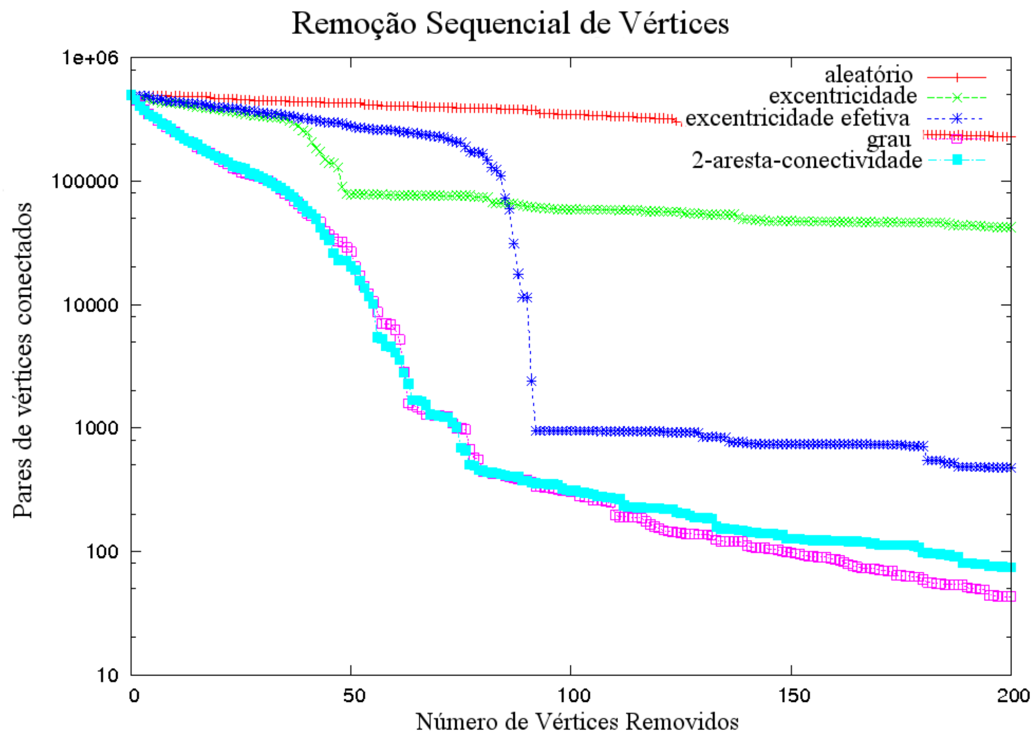


Figura 7.8: Número de pares de nodos conectados após a remoção sequencial de nodos.

O experimento mostrou que as remoções de vértices pela 2-aresta-conectividade e pelo grau desconectam o grafo mais rapidamente do que as demais medidas.

7.3 Experimentos com Conectividade de Vértices

Nesta seção apresentamos experimentos com a medida de conectividade baseada em cortes de vértices. Os resultados desta seção foram previamente publicados em [87] e [86].

Os experimentos foram realizados em uma rede real de aeroportos dos Estados Unidos (USAir97) [9] com 332 nodos e em uma rede real de distribuição de energia elétrica (*powergrid*) [105] com 4941 nodos. Foram analisados os 300 nodos de maior grau.

Rede de Aeroportos Americanos A rede de aeroportos dos Estados Unidos [9] contém 332 nodos que representam aeroportos e 2126 arestas que representam voos entre os aeroportos.

A Figura 7.9 apresenta um gráfico com a relação entre a 2-aresta-conectividade e a 2-vértice-conectividade dos 300 vértices de maior grau do grafo dos aeroportos norte americanos. Apesar da alta correlação entre as medidas, podemos ver que há inversões quando os vértices são ordenados com base em cada medida. As inversões são vértices u, v tais que $\kappa_2(u) > \kappa_2(v)$ e $\lambda_2(u) < \lambda_2(v)$.

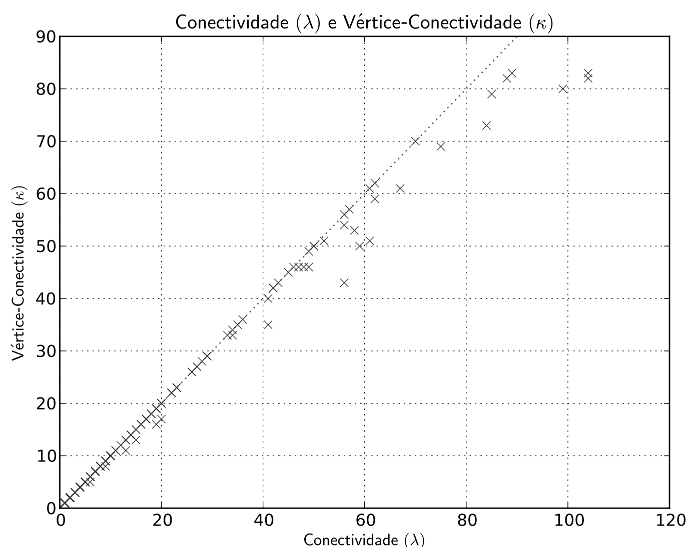


Figura 7.9: Relação entre a 2-aresta-conectividade e a 2-vértice-conectividade do grafo dos aeroportos [86, 87].

A Figura 7.10 mostra um gráfico comparando a 2-vértice-conectividade com o grau dos nodos da rede dos aeroportos. Novamente verificamos algumas inversões nas ordenações.

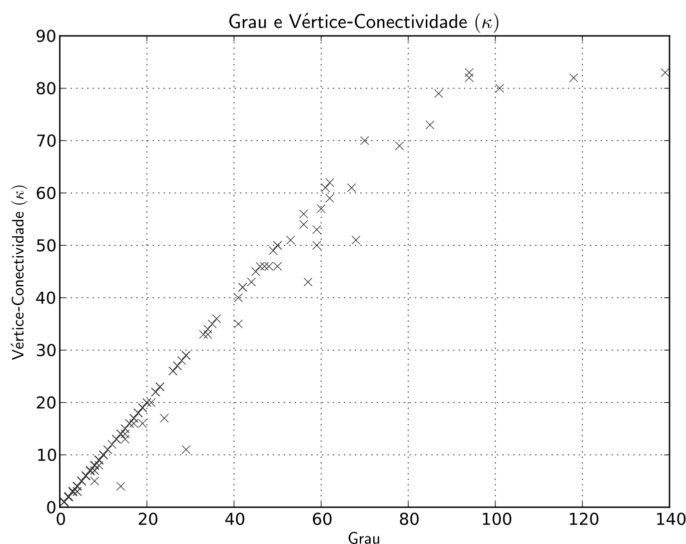


Figura 7.10: Grau e 2-vértice-conectividade na rede de aeroportos [86, 87].

A Figura 7.11 mostra a relação entre a 2-vértice-conectividade e o grau de intermediação (*betweenness*). Os vértices com as maiores 2-vértice-conectividade têm grau de intermediação alto. Além disso, o gráfico mostra que existem aeroportos pouco conexos com alto grau de intermediação. Isso pode ocorrer quando um aeroporto pequeno está ligado a dois aeroportos grandes.

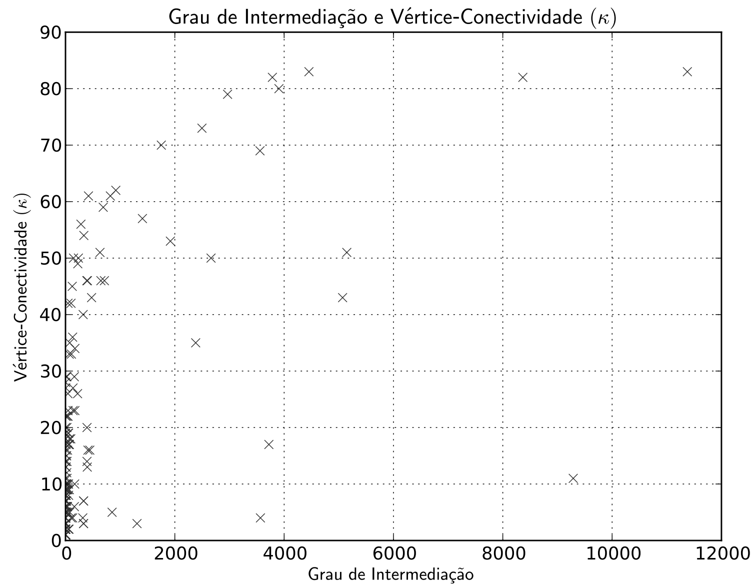


Figura 7.11: Grau de intermediação e 2-vértice-conectividade do grafo dos aeroportos [86, 87].

O gráfico da Figura 7.12 mostra a relação entre a 2-vértice-conectividade e o grau de proximidade (*closeness*) da redes de aeroportos. Podemos observar uma correlação entre as medidas de forma que os vértices de maior 2-vértice-conectividade são mais centrais também em relação ao grau de proximidade.

Rede de Distribuição de Energia Nesta seção descrevemos os experimentos com a vértice-conectividade no grafo *powergrid* [105] que corresponde a uma rede de distribuição de eletricidade com 4941 nodos e 6594 arestas. Os 300 vértices de maior grau foram analisados. O gráfico da Figura 7.13 mostra a relação entre a 2-aresta-conectividade e as 2-vértice-conectividade do grafo *powergrid*. Vemos no gráfico que as medidas diferem para diversos vértices e que existem múltiplas inversões na ordenação dos vértices pelas duas medidas.

A Figura 7.14 mostra a relação entre a 2-vértice-conectividade e o grau no grafo *powergrid*. As duas medidas são bastante distintas nesse grafo, apresentando pouca correlação, provavelmente devido à presença de vértices periféricos com grau elevado. No grafo *powergrid* os vértices com maior grau de intermediação, com maior grau de proximidade e com menor excentricidade *não* correspondem aos vértices com maior vértice-conectividade, como pode ser visto nos gráficos das Figuras 7.15, 7.16 e 7.17. Nota-se que estas três medidas apresentam pouca correlação com a vértice-conectividade.

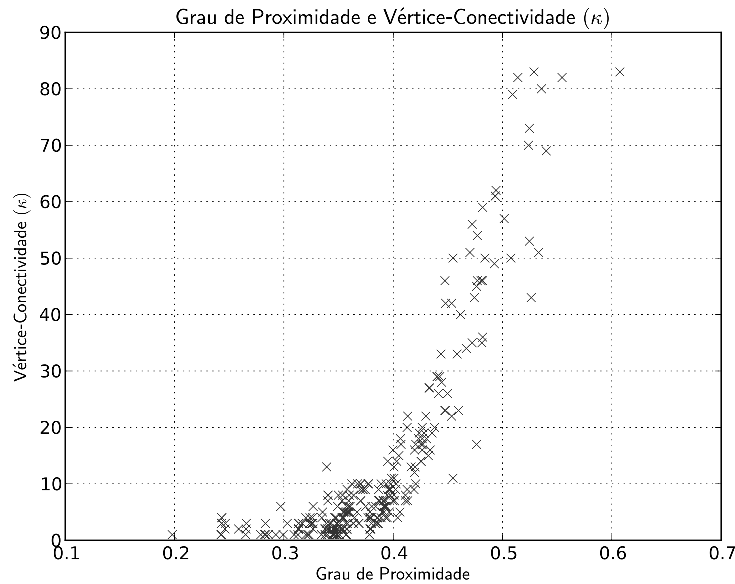


Figura 7.12: Grau de proximidade e 2-vértice-conectividade do grafo dos aeroportos [86, 87].

7.4 Comentários Finais sobre os Experimentos com as Medidas de Conectividade

Esse capítulo apresentou resultados experimentais envolvendo algumas das medidas de conectividade propostas na Seção 5.2.1. Mostramos que embora exista uma alta correlação entre o grau dos vértices, a 2-aresta-conectividade e a 2-vértice-conectividade em algumas redes, pode-se verificar múltiplas inversões na ordenação dos vértices utilizando cada medida.

Experimentos em redes complexas reais e sintéticas mostraram a existência de vértices com grau alto, mas aresta-conectividade relativamente baixa. Mostramos que nodos de alta conectividade têm baixa excentricidade em redes do modelo GLP, o que justifica a utilização das medidas de conectividade como uma medida de centralidade em redes.

Essa seção também apresentou resultados experimentais sobre vértice-conectividade de duas redes complexas e mostrou que a correlação entre a vértice-conectividade e a aresta-conectividade é alta, porém inversões nas ordenações dos vértices utilizando cada medida podem ser encontradas.

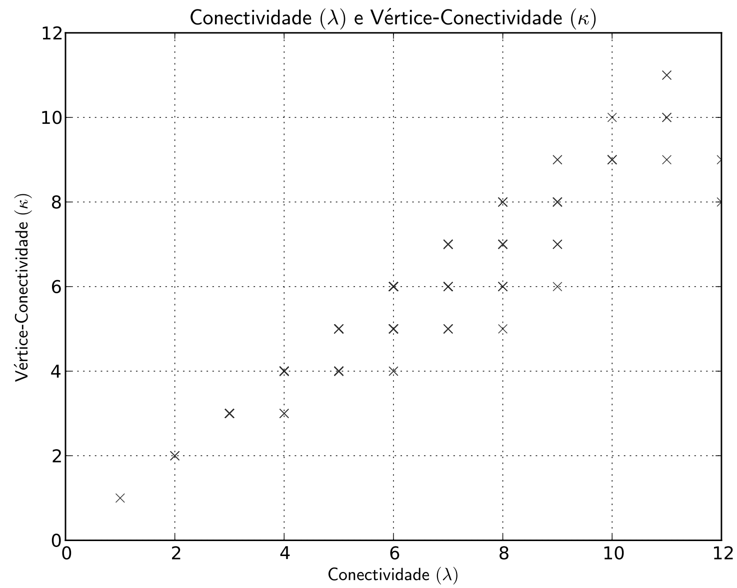


Figura 7.13: Relação entre a 2-aresta-conectividade e a 2-vértice-conectividade no grafo *power-grid* [86, 87].

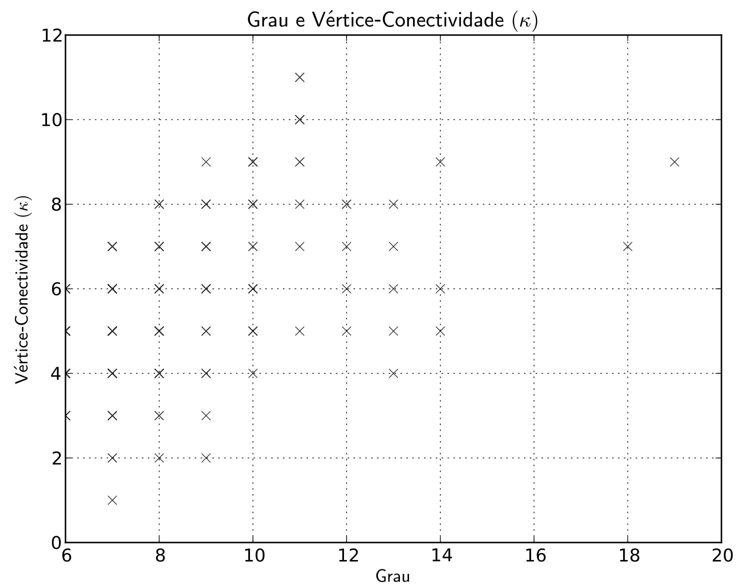


Figura 7.14: Relação entre a 2-vértice-conectividade e o grau no grafo *powergrid* [86, 87].

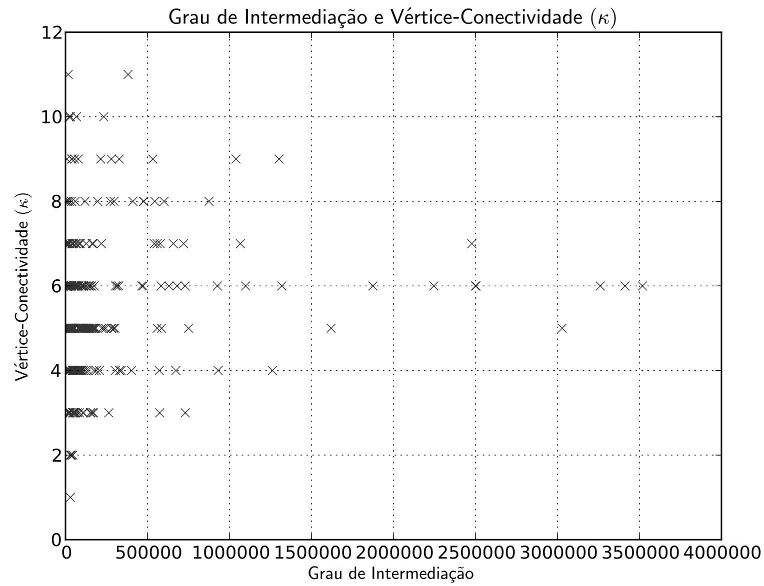


Figura 7.15: Relação entre a 2-vértice-conectividade e o grau de intermediação no grafo *power-grid* [86, 87].

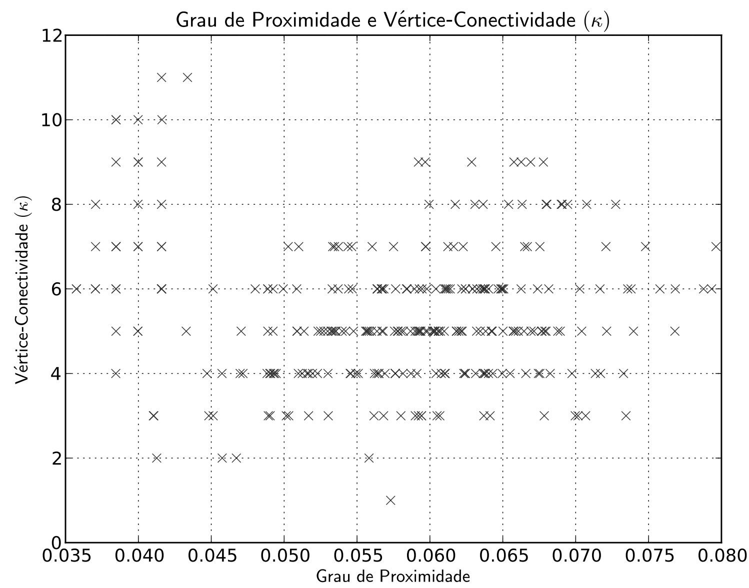


Figura 7.16: Relação entre a 2-vértice-conectividade e o grau de proximidade no grafo *power-grid* [86, 87].

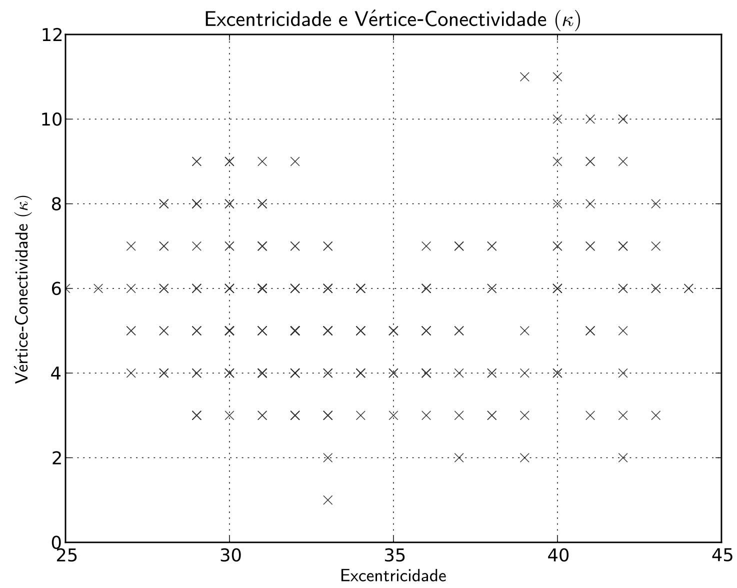


Figura 7.17: Relação entre a 2-vértice-conectividade e a excentricidade no grafo *powergrid* [86, 87].

8 Conclusão

As árvores de cortes são uma conhecida estrutura combinatória que representa os cortes de arestas de multigrafos não orientados e possuem diversas aplicações. Foram propostas e implementadas versões paralelas do algoritmo de Gusfield e do algoritmo de Gomory-Hu para a construção de árvores de cortes. Um novo algoritmo híbrido que combina esses dois algoritmos foi proposto e implementado. Os algoritmos foram avaliados experimentalmente e apresentaram boas acelerações nos tempos de execução na maioria das instâncias testadas. Os experimentos mostraram que o algoritmo híbrido é mais rápido do que o algoritmo de Gomory-Hu em quase todas as instâncias. Além disso, o algoritmo híbrido foi muito mais rápido do que o algoritmo de Gusfield em algumas instâncias. Várias heurísticas para o algoritmo de Gomory-Hu e para o algoritmo híbrido foram propostas, implementadas e avaliadas.

A presente tese traz definições originais de diversas medidas de centralidade de nodos de grafos baseadas em conceitos de conectividade. Algoritmos polinomiais para o cálculo das medidas propostas foram apresentados. A avaliação empírica das medidas mostrou que embora exista correlação entre o grau dos vértices, a 2-aresta-conectividade e a 2-vértice-conectividade em algumas redes, pode-se verificar múltiplas inversões na ordenação dos vértices utilizando cada medida. Experimentos em redes complexas reais e sintéticas mostraram a existência de vértices com grau alto, mas aresta-conectividade relativamente baixa. Mostramos que nodos de alta conectividade têm baixa excentricidade em redes do modelo GLP.

Podemos listar alguns trabalhos futuros que decorrem da presente tese. Em relação aos algoritmos paralelos para árvores de cortes, heurísticas que reduzam o número de cortes descartados nos algoritmos de Gomory-Hu e híbrido podem ser estudadas. Também podem ser estudadas heurísticas que determinem uma ordenação dos pares de vértices a serem separados que favoreça a busca por cortes mais balanceados. O procedimento de construção do grafo contraído é computacionalmente custoso e pode ser acelerado evitando-se que todas as arestas do grafo de entrada sejam percorridas cada vez que um grafo contraído é construído. Análises formais do desempenho dos algoritmos paralelos podem ser investigadas.

Em relação às medidas de centralidade, aplicações das medidas propostas na tese podem ser procuradas. Por exemplo, as medidas de centralidade podem ser usadas em métricas para determinar a conectividade de grafos, o que pode ser utilizado na classificação dos grafos quanto à sua conectividade. A possibilidade de aplicar as medidas de centralidade aqui propostas em outras áreas do conhecimento onde as redes complexas têm sido estudadas deve ser investigada.

Lista de Publicações

Os seguintes artigos foram publicados ao longo do desenvolvimento da tese:

- Jaime Cohen, Luiz A. Rodrigues, Elias P. Duarte Jr., **"A Parallel Implementation of Gomory-Hu's Cut Tree Algorithm,"** *The 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012),* New York, U.S.A., 2012.
- Jaime Cohen, Luiz A. Rodrigues, Fabiano Silva, Renato Carmo, André Guedes, Elias P. Duarte Jr., **"Parallel Implementations of Gusfield's Cut Tree Algorithm,"** *11th International Conference Algorithms and Architectures for Parallel Processing (ICA3PP), pp. 258-269, Lecture Notes in Computer Science (LNCS) 7016, ISSN 0302-9743,* Melbourne, Australia, 2011.
- Jaime Cohen, Elias P. Duarte Jr., Jonatan Schroeder, **"Connectivity Criteria for Ranking Network Nodes,"** *Communications in Computer and Information Science (CCIS), ISSN 1865-0929, Vol. 116. pp. 35-45, 2011.*, Rio de Janeiro, 2010.
- Jaime Cohen, Luiz A. Rodrigues, Elias P. Duarte Jr., **"Improved Parallel Implementations of Gusfield's Cut Tree Algorithm",** *XII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-SSC'2011),* Vitória, ES, 2011.
- Karine Pires, Jaime Cohen, Elias P. Duarte Jr., **"Medidas de Conectividade Baseadas em Cortes de Vértices para Redes Complexas,"** *12o Workshop de Testes e Tolerância a Falhas (WTF), 29o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), pp. 77-89,* Campo Grande, MS, 2011.

Referências Bibliográficas

[1] Lada A. Adamic e Natalie Glance. The political blogosphere and the 2004 U.S. election: divided they blog. *Proceedings of the 3rd international workshop on Link discovery*, LinkKDD '05, páginas 36–43, New York, NY, USA, 2005. ACM.

[2] Réka Albert e Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, 2002.

[3] Yash P. Aneja, Ramaswamy Chandrasekaran, e Kunhiraman P. K. Nair. Parametric min-cuts analysis in a network. *Discrete Appl. Math.*, 127(3):679–689, 2003.

[4] John G. Apostolopoulos e Mitchell D. Trott. Path diversity for enhanced media streaming. *IEEE Communications Magazine*, 42(8), 2004.

[5] Srinivasa R. Arikati, Shiva Chaudhuri, e Christos D. Zaroliagis. All-pairs min-cut in sparse networks. *J. Algorithms*, 29(1):82–110, 1998.

[6] Lars Backstrom, Cynthia Dwork, e Jon Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. *Proceedings of the 16th international conference on World Wide Web*, WWW '07, páginas 181–190, New York, NY, USA, 2007. ACM.

[7] David A. Bader e Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. Michael J. Oudshoorn e Sanguthevar Rajasekaran, editors, *ISCA PDCS*, páginas 41–48. ISCA, 2005.

[8] Dominique Barth, Pascal Berthomé, e Madiagne Diallo. Effects of capacities variations on maximum flows, minimum cuts and edge saturation. Relatório técnico, LRI-1395, 2004.

[9] Vladimir Batagelj e Andrej Mrvar. Pajek datasets. *Disponível em http://vlado.fmf.uni-lj.si/pub/networks/data/*, Acessado em abril de 2011.

[10] András A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. Comput.*, 24(3):505–510, 1995.

[11] Pascal Berthomé, Madiagne Diallo, e Afonso Ferreira. Generalized parametric multi-terminal flows problem. Hans L. Bodlaender, editor, *WG*, volume 2880 of *Lecture Notes in Computer Science*, páginas 71–80. Springer, 2003.

[12] Béla Bollobás. *Random Graphs*. Cambridge University Press, 2 edition, 2001.

[13] John A. Bondy e Uppaluri S.R. Murty. *Graph theory*. Graduate texts in mathematics. Springer, 2008.

[14] Steve Borgatti e Martin G. Everett. A graph-theoretic perspective on centrality. *Social Networks*, 28(4):466–484, 2006.

[15] Glencora Borradaile, Piotr Sankowski, e Christian Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:601–610, 2010.

[16] Ulrik Brandes e Thomas Erlebach. *Network Analysis: Methodological Foundations*. Springer, 2005.

[17] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, e Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.

[18] Tian Bu e Donald F. Towsley. On distinguishing between internet power law topology generators. *Proc. IEEE INFOCOM*, 2002.

[19] Geoffrey S. Canright e Kenth Engø-Monsen. Spreading on networks: A topographic view. *Complexus*, 2006.

[20] Kim Chae-Bogk, Foote Bobbie L., e Pulat P. Simin. Cut-tree construction for facility layout. *Computers and Industrial Engineering*, 28(4):721–730, 1995.

[21] Bala G. Chandran e Dorit S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Oper. Res.*, 57(2):358–376, março de 2009.

[22] Barbara Chapman, Gabriele Jost, e Ruud Van der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT Press, 2008.

[23] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, e Cliff Stein. Experimental study of minimum cut algorithms. *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, páginas 324–333, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[24] Boris V. Cherkassky e Andrew V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997.

[25] Jaime Cohen. Caminhos virtuais: Aplicações dependentes da internet tolerantes a falhas nas rotas de rede. *Congresso Brasileiro de Computação*, 2001.

[26] Jaime Cohen, Jonatan Schroeder, e Elias P. Duarte Jr. Connectivity criteria for ranking network nodes. *Proceedings of the 2nd Workshop on Complex Networks*, 2010.

[27] Charles J. Colbourn e Jeffrey H. Dinitz. *Handbook of Combinatorial Designs*. Number v. 10 in Discrete Mathematics And Its Applications. Chapman & Hall/Taylor & Francis, 2007.

[28] Charles J. Colbourn e Eugene I. Litvak. Bounding network parameters by approximating graphs. *Reliability of Computer and Communications Networks*, 1991.

[29] Michele Conforti, Rafael Hassin, e R. Ravi. Reconstructing edge-disjoint paths. *Operations Research Letters*, 31(4):273 – 276, 2003.

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

[31] Luciano da F. Costa, Francisco A. Rodrigues, Gonzalo Travieso, e P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in Physics*, 56(1):167–242, 2007.

[32] Yefim Dinitz. Dinitz' Algorithm: The Original Version and Even's Version. Oded Goldreich, Arnold Rosenberg, e Alan Selman, editors, *Theoretical Computer Science*, volume 3895 of *Lecture Notes in Computer Science*, capítulo 10, páginas 218–240. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[33] Christof Doll, Tanja Hartmann, e Dorothea Wagner. Fully-dynamic hierarchical graph clustering using cut trees. Frank Dehne, John Iacono, e Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, páginas 338–349. Springer Berlin Heidelberg, 2011.

[34] Sergey. N. Dorogovtsev e José F. Mendes. Evolution of networks. *Advances in Physics*, 51:1079, 2002.

[35] Sergey N. Dorogovtsev e José F. Mendes. The shortest path to complex networks, 2004.

[36] David. Easley e Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Networks, Crowds, and Markets: Reasoning about a Highly Connected World. Cambridge University Press, 2010.

[37] Jack Edmonds e Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, 1972.

[38] Roee Engelberg, Jochen Könemann, Stefano Leonardi, e Joseph S. Naor. Cut problems in graphs with a budget constraint. *J. of Discrete Algorithms*, 5:262–279, 2007.

[39] Johannes Fischer e Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. Moshe Lewenstein e Gabriel Valiente, editors, *Combinatorial Pattern Matching*, volume 4009 of *Lecture Notes in Computer Science*, páginas 36–48. Springer Berlin / Heidelberg, 2006.

[40] Gary William Flake, Robert Endre Tarjan, e Kostas Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4), 2003.

[41] Lester R. Ford e Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1955.

[42] Linton C. Freeman. Going the wrong way on a one-way street: Centrality in physics and biology. *Journal of Social Structure*, 2008.

[43] Andrew V. Goldberg e Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940, 1988.

[44] Andrew V. Goldberg e Kostas Tsioutsiouliklis. Cut tree algorithms. *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, páginas 376–385, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[45] Andrew V Goldberg e Kostas Tsioutsiouliklis. Cut tree algorithms: An experimental study. *Journal of Algorithms*, 38(1):51 – 83, 2001.

[46] Ralph E. Gomory e Te C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[47] Robert Görke, Tanja Hartmann, e Dorothea Wagner. Dynamic graph clustering using Minimum-Cut trees. Frank Dehne, Marina Gavrilova, Jörg-Rüdiger Sack, e Csaba Tóth, editors, *Algorithms and Data Structures*, volume 5664 of *Lecture Notes in Computer Science*, capítulo 30, páginas 339–350. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[48] Robert Görke, Tanja Hartmann, e Dorothea Wagner. Dynamic graph clustering using minimum-cut trees. *J. Graph Algorithms Appl.*, 16(2):411–446, 2012.

[49] Dan Gusfield. A little knowledge goes a long way: Faster detection of compromised data in 2-d tables. *IEEE Symposium on Security and Privacy*, páginas 86–94, 1990.

[50] Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.

[51] S. Louis Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *OPERATIONS RESEARCH*, 12(3):450–459, 1964.

[52] Junghee Han, David Watson, e Farnam Jahanian. An experimental study of internet path diversity. *IEEE Transactions on Dependable and Secure Computing*, 3(4), 2006.

[53] Dov Harel e Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[54] Ramesh Hariharan, Telikepalli Kavitha, e Debmalya Panigrahi. Efficient algorithms for computing all low s-t edge connectivities and related problems. *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, páginas 127–136, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[55] Ramesh Hariharan, Telikepalli Kavitha, Debmalya Panigrahi, e Anand Bhalgat. An O(mn) Gomory-Hu tree construction algorithm for unweighted graphs. *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, páginas 605–614, New York, NY, USA, 2007. ACM.

[56] Erez Hartuv e Ron Shamir. A clustering algorithm based on graph connectivity. *Inf. Process. Lett.*, 76(4-6):175–181, 2000.

[57] David Hartvigsen. Generalizing the all-pairs min cut problem. *Discrete Mathematics*, 147(1-3):151–169, 1995.

[58] Refael Hassin e Asaf Levin. Flow trees for vertex-capacitated networks. *Discrete Applied Mathematics*, 155(4):572–578, 2007.

[59] Dorit S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.

[60] Bo Hong e Zhengyu He. An asynchronous multi-threaded algorithm for the maximum network flow problem with non-blocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[61] Chris Jermaine. Computing program modularizations using the k-cut method. *WCRE*, páginas 224–234, 1999.

[62] Daxin Jiang, Chun Tang, e Aidong Zhang. Cluster analysis for gene expression data: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 16(11):1370–1386, 2004.

[63] Jon M. Kleinberg Jure Leskovec e Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*, 2007.

[64] Santosh N. Kabadi, Ramaswamy Chandrasekaran, e Kunhiraman P.K. Nair. Multiroute flows: Cut-trees and realizability. *Discrete Optimization*, 2(3), 2005.

[65] Krishna Y. Kamath e James Caverlee. Transient crowd discovery on the real-time social web. *Proceedings of the 4th ACM Web Search and Data Mining Conference (WSDM)*, 2011.

[66] Valerie King, Satish Rao, e Rorbert Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 17(3):447–474, novembro de 1994.

[67] Bernhard Korte e Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, Germany, 2nd edition, 2002.

[68] Dirk Koschützki, Katharina Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, e Oliver Zlotowski. Centrality indices. Ulrik Brandes e Thomas Erlebach, editors, *Network Analysis*, volume 3418 of *Lecture Notes in Computer Science*, páginas 16–61. Springer Berlin / Heidelberg, 2005.

[69] Dirk Koschützki, Katharina Lehmann, Dagmar Tenfelde-Podehl, e Oliver Zlotowski. Advanced centrality concepts. Ulrik Brandes e Thomas Erlebach, editors, *Network Analysis*, volume 3418 of *Lecture Notes in Computer Science*, páginas 83–111. Springer Berlin / Heidelberg, 2005.

[70] Adam N. Letchford, Gerhard Reinelt, e Dirk O. Theis. Odd minimum cut-sets and b-matchings revisited. *SIAM Journal on Discrete Mathematics*, 22(4), 2008.

[71] Matthew S. Levine. Experimental study of minimum cut algorithms. Relatório técnico, Massachusetts Institute of Technology, Cambridge, MA, USA, 1997.

[72] Clémence Magnien, Matthieu Latapy, e Guillau Jean-Loup. Impact of random failures and attacks on poisson and power-law random networks. *ACM Comput. Surv.*, 43:13:1–13:31, April de 2011.

[73] Charles F. Mann, David W. Matula, e Eli V. Olinick. The use of sparsest cuts to reveal the hierarchical community structure of social networks. *Social Networks*, 30(3):223 – 234, 2008.

[74] Clyde L. Monma Martin Grötschel e Mechthild Stoer. Polyhedral and computational investigations for designing communication networks with high survivability requirements. *Operations Research*, 1995.

[75] Elias P. Duarte Jr. e Jaime Cohen. Fault-tolerant routing of tcp/ip pdu's in general topology backbones. *The 3rd IEEE International Workshop on the Design of Reliable Communication Networks (DRCN'2001)*, 2001.

[76] Elias Procópio Duarte Jr., Rogério Santini, e Jaime Cohen. Delivering packets during the routing convergence latency interval through highly connected detours. *DSN*, páginas 495–. IEEE Computer Society, 2004.

[77] Antonina Mitrofanova, Martin Farach-Colton, e Bud Mishra. Efficient and robust prediction algorithms for protein complexes using gomory-hu trees. Russ B. Altman, A. Keith Dunker, Lawrence Hunter, Tiffany Murray, e Teri E. Klein, editors, *Pacific Symposium on Biocomputing*, páginas 215–226, 2009.

[78] David R. Musser, Gilmer J. Derge, e Atul Saini. *STL tutorial and reference guide, second edition: C++ programming with the standard template library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[79] Hiroshi Nagamochi e Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, New York, NY, USA, 2008.

[80] Hiroshi Nagamochi, Tadashi Ono, e Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67:325–341, 1994.

[81] Mark Newman. *Networks: An Introduction*. OUP Oxford, 2010.

[82] Mark E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[83] Peter S. Pacheco. *A User's Guide to MPI*. University of San Francisco, 1998.

[84] Manfred W. Padberg e M. R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7, 1982.

[85] Christopher R. Palmer, Georgos Siganos, Michalis Faloutsos, Christos Faloutsos, e Phillip B. Gibbons. The connectivity and fault-tolerance of the internet topology. *In Workshop on Network-Related Data Management (NRDM-2001*, 2001.

[86] Karine Pires. Medidas de conectividade baseadas em cortes de vértices para redes complexas. Dissertação de Mestrado, Departamento de Informática - Universidade Federal do Paraná, 2011.

[87] Karine Pires, Jaime Cohen, e Elias P. Duarte Jr. Medidas de conectividade baseadas em cortes de vértices para redes complexas. *12º Workshop de Testes e Tolerância a Falhas (WTF'2011), Anais do SBRC'2011*, 2011.

[88] John S. Provan e Douglas R. Shier. A paradigm for listing (s, t)-cuts in graphs. *Algorithmica*, 15(4):351–372, 1996.

[89] Michael J. Quinn. *Parallel Programming in C With MPI and OpenMP*. Mcgraw Hill Higher Education, 2003.

[90] Gururaj S. Rao, Harold S. Stone, e Te C. Hu. Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, 28:291–299, 1979.

[91] Britta Ruhnau. Eigenvector-centrality – a node-centrality? *Social Networks*, 22(4):357–365, outubro de 2000.

[92] Barna Saha e Pabitra Mitra. Dynamic algorithm for graph clustering using minimum cut tree. *Proceedings of the Sixth IEEE International Conference on Data Mining - Workshops*, páginas 667–671, Washington, DC, USA, 2006. IEEE Computer Society.

[93] Barna Saha e Pabitra Mitra. Dynamic algorithm for graph clustering using minimum cut tree. *SDM*. SIAM, 2007.

[94] Huzur Saran e Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108, 1995.

[95] Huzur Saran e Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108, 1995.

[96] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[97] Claus-Peter Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM J. Comput.*, 8(2):265–274, 1979.

[98] Yuval Shavitt e Yaron Singer. Beyond centrality: classifying topological significance using backup efficiency and alternative paths. *Proceedings of the 6th international IFIP-TC6 conference on Ad Hoc and sensor networks, wireless networks, next generation internet*, NETWORKING'07, páginas 774–785, Berlin, Heidelberg, 2007. Springer-Verlag.

[99] Gianni Storchi, Paolo Dell'Olmo, e Monica Gentili. Road network of the city of rome, 1999.

[100] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, e Walter Willinger. Network topology generators: degree-based vs. structural. *SIGCOMM Comput. Commun. Rev.*, 32:147–159, 2002.

[101] Nurcan Tuncbag, F. Sibel Salman, Ozlem Keskin, e Attila Gursoy. Analysis and network representation of hotspots in protein interfaces using minimum cut trees. *Proteins: Structure, Function, and Bioinformatics*, 78(10):2283–2294, 2010.

[102] Matthias Wählisch. Modeling the Network Topology. Klaus Wehrle, Mesut Günes, e James Gross, editors, *Modeling and Tools for Network Simulation*, páginas 471–486. Springer, Heidelberg, 2010.

[103] Stanley Wasserman e Katherine Faust. *Social network analysis: methods and applications*. Structural analysis in the social sciences. Cambridge University Press, 1994.

[104] Duncan J. Watts, Peter S. Dodds, e Mark E. J. Newman. Identity and search in social networks. *Science*, 296:1302–1305, 2002.

[105] Duncan J. Watts e Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[106] Lin Wu e Pramod K. Varshney. On survivability measures for military networks. *Military Communications Conference*, páginas 1120–1124, 1990.

[107] Stefan Wuchty e Peter F. Stadler. Centers of complex networks. *J Theor Biol.*, 223(1), 2003.

[108] Xianchao Zhang, Weifa Liang, e He Jiang. Flow equivalent trees in undirected node-edge-capacitated planar graphs. *Inf. Process. Lett.*, 100(3):110–115, 2006.