Problemas Combinatórios de Congestionamento

Arlindo Flávio da Conceição

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Problemas Combinatórios de Congestionamento

Arlindo Flávio da Conceição

Julho de 2000

Banca Examinadora:

- Prof. Dr. João Carlos Setubal (Orientador)
- Prof. Dr. Jorge Stolfi
- Profa. Dra. Yoshiko Wakabayashi
- Prof. Dr. Cid Carvalho de Souza (suplente)

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA CENTRAL DA UNICAMP

C744p

Conceição, Arlindo Flávio da
Problemas combinatórios de congestionamento / Arlindo Flávio da Conceição. -- Campinas, SP : [s.n.], 2000.

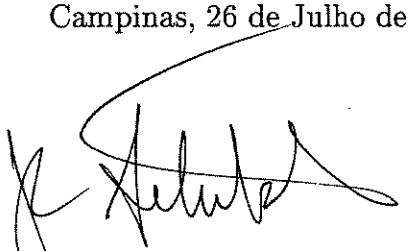
Orientador: João Carlos Setubal.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Árvores (Teoria dos grafos). 2. Otimização combinatória. 3. Algoritmos. I. Setubal, João Carlos. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Problemas Combinatórios de Congestionamento

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Arlindo Flávio da Conceição e aprovada pela Banca Examinadora.

Campinas, 26 de Julho de 2000.



Prof. Dr. João Carlos Setubal (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 09 de junho de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Profa Dra. Yoshiko Wakabayashi
IME - USP

Prof. Dr. Jorge Stolfi
IC-UNICAMP

Prof. Dr. Cid Carvalho de Souza
IC-UNICAMP

Prof. Dr. João Carlos Setubal
IC-UNICAMP

Resumo

Nesta dissertação estudamos o aspecto combinatório dos problemas de congestionamento. O principal problema estudado consiste em, dado um grafo G e um inteiro positivo k, encontrar k árvores espalhadas de G, não necessariamente disjuntas, de peso total mínimo. Neste problema o congestionamento é caracterizado por funções que penalizam o peso de uma aresta se ela é usada por mais de uma árvore. *Roskind* e *Tarjan* apresentaram um algoritmo para a versão deste problema onde as árvores devem ser disjuntas nas arestas. Nós apresentamos uma descrição detalhada do algoritmo de *Roskind* e *Tarjan* e então mostramos um algoritmo polinomial para o problema de congestionamento, o algoritmo consiste em uma redução ao problema disjunto. O nosso algoritmo é quadrático em k. Apresentamos ainda duas heurísticas de complexidade linear em k. Baseados na mesma técnica, desenvolvemos algoritmos polinomiais para problemas combinatórios de congestionamento relacionados aos problemas de encontrar um caminho mínimo e de encontrar um emparelhamento de custo mínimo.

Abstract

This work studies the combinatorial aspects of congestion problems. The main problem studied is the following: Given a graph G and a positive integer k, we want to find k spanning trees on G, not necessarily disjoint, of minimum total weight, such that the weight of each edge is subject to a penalty function if it belongs to more than one tree. This penalty function models congestion situations. Roskind and Tarjan have developed an algorithm for the disjoint version of this problem. We present a detailed description of their algorithm and then show that a polynomial algorithm for the congestion problem can be obtained by reducing it to the disjoint problem. The complexity of our algorithm is quadratic in k. We also present two heuristics with complexity linear in k. Based on the same idea, we then present polynomial algorithms for combinatorial congestion problems related to shortest paths and minimum weight matchings.

Agradecimentos

Gostaria de agradecer a todos aqueles que contribuíram na realização deste trabalho. Devo ressaltar que esta tese jamais teria sido concluída não fosse o apoio de algumas pessoas e entidades.

Agradeço ao Prof. Dr. Setubal pela dedicação e paciência com que conduziu este trabalho. Agradeço especialmente pelas horas que ele não tinha e que gastou revisando este material.

Agradeço também ao Prof. Dr. Stolfi pelas valorosas dicas e discussões a respeito do nosso tema de pesquisa. Em uma destas discussões, foi do Prof. Stolfi a idéia que serviu de alicerce para os algoritmos mostrados neste trabalho.

Não poderia deixar de agradecer ao corpo docente do Instituto de Computação pela valiosa contribuição à minha formação. Especificamente agradeço aos professores Lucchesi, Setubal, Buzato, Cid e Meidanis. Novamente agradeço ao professor Cid pela confiança e apoio dados na ocasião da seleção para o programa de doutorado.

Este trabalho também não poderia ser realizado sem o apoio financeiro do CNPq.

Dedicatória

Quanto vale uma tese?

Como tudo na vida, vale quanto se paga. E nós não pagamos pouco. Pagamos com esforço, com ausência e com saudades. Pagamos com a distância dos pais, que não vimos envelhecer, com a surpresa ao rever os sobrinhos, que não vimos crescer, pagamos com o tempo, que não vimos passar. Enfim, pagamos com o que não tem preço, por isso, para nós, uma tese não tem preço.

Mas não estou reclamando. Se me afastei daqueles que amo, hoje amo mais uma pessoa. Se perdi bons momentos entre amigos, agora tenho o dobro de amigos. Se tive momentos de cansaço, foi porque tive prazer no que fiz.

Gostaria de dedicar esta tese a todas as pessoas que contribuíram para fazer da minha jornada em Campinas uma das melhores fases da minha vida. Fiz então uma lista onde tentei não esquecer de ninguém. E para relembrar cada momento desta fase acrescentei lugares e codinomes, tal que cada palavra da lista representa uma pessoa querida ou um lugar de onde guardo boas lembranças.

Clayton, Leandro, Lauro, Silvia, Ticum, Susan, Bicho, Márcia, Simone, Denimar, Gabriela, Kimi e Daniel, Aquino, Eduardo Voigt, Jairo Panetta, César, Franklin, Pimentel, Erlon, Salinha, Vizinha, Alternativo, Hélio, Dário, Delano, Flávia, Batatinha, Moradia, P8, P10, F6-A, Marcinha, Lourdinha, Giovana, Cacilda, Cintia, Roberta, Academia, Marcelo, Escola de Computação no Rio, SBC-UFMG, Lídio, Eduardo Rodrigues, Bandeijão, Martas, Hebe, Maras, Cláudio, Silvio, Aiuruoca, Cintra, Garden, Ubatuba, Neri, Célia, Rodrigo, Cooperativa, Quermesses, Caipora, Bar do Marcão, Paróquia de Santana, Festas, Shows, Piscina, Balde, Grito, Marlene, Eva, Mestre, Caubi, Vilmo, Soninha, Teatro de Arena, IFCH, Pedagogia, Porta, Flávio, Felipe, Rafaela, Mariana, Fernada, Bianca, Ângela, Rosângela, Fátima, Miguel, José, Silvia, Bô, Maurício, Valdeci, Pipoca, Catraca, Wagner, Congregação e outros membros da Liga.

Gostaria ainda de dedicar esta tese especialmente aos meus pais e à Daniela.

Termino respondendo a questão inicial: quanto vale uma tese?

Vale a pena experimentar!

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A palavra congestionamento invariavelmente nos faz lembrar do trânsito, isto é natural pois o problema do trânsito está muito presente no nosso cotidiano. Além disso, o problema do trânsito é o exemplo clássico de um problema de congestionamento. Mas o termo congestionamento não se restringe ao trânsito, outros problemas também se enquadram nesta classe de problemas.

Os problemas de congestionamento têm como característica comum a competição por um determinado recurso. Tal competição pode ser gerada pela escassez de recursos ou pelo excesso de competidores e geralmente resulta em lentidão (ou alguma outra conseqüência indesejável). Por exemplo, no trânsito um congestionamento pode ocorrer porque as vias de tráfego são insuficientes para o volume de carros, ou porque o número de veículos é superior ao espaço nas ruas e avenidas.

A mesma competição que forja o elo de semelhança entre os problemas de congestionamento é também responsável por duas características importantes e que devem ser observadas quando queremos resolver tais problemas: a forte interação entre os competidores e a freqüente não-linearidade do problema. Estas duas características são importantes porque podem inviabilizar a busca por uma solução exata para tais problemas.

Imagine que precisamos modelar e resolver de forma exata um problema de congestionamento. Para isto, devemos modelar o comportamento de cada competidor e a interação entre eles. Mas isto é inviável para instâncias reais da maioria dos problemas de congestionamento. Inviável não só pelo tamanho das instâncias, mas principalmente pelo caráter por vezes imprevisível de tal comportamento. O trânsito é um exemplo prático de tal imprevisibilidade.

Além do tamanho e da complexidade da modelagem exata de um problema de congestionamento, estes problemas são em geral não lineares. Por exemplo, se um carro gasta x para pagar pedágio, dois carros certamente gastarão mais do que $2x$ para continuarem a viagem. Assim, se queremos resolver estes problemas de forma exata, então precisaremos

trabalhar com problemas não lineares e estes problemas são geralmente mais difíceis de serem resolvidos do que os problemas lineares.

Dada a dificuldade de se encontrar uma solução exata para a maioria dos problemas de congestionamento, uma alternativa é modelar um congestionamento usando funções que aproximam o comportamento do sistema e então resolver o modelo aproximado. Estas funções de aproximação são chamadas funções de penalização.

Nesta dissertação serão mostrados algoritmos e heurísticas para resolver problemas combinatórios de congestionamento modelados a partir de funções de penalização.

1.1 Funções de penalização e congestionamentos

Neste trabalho, as funções de penalização são usadas para penalizar as escolhas feitas por um algoritmo. O exemplo a seguir ilustra como a utilização das funções de penalização pode simular o comportamento de um problema de congestionamento.

Dado um conjunto S, de n elementos ponderados, e um inteiro positivo k, queremos encontrar k subconjuntos $S_1, S_2, \ldots, S_k$ de cardinalidade dois, tal que o peso total *penalizado* destes conjuntos seja mínimo. A fim de caracterizar o congestionamento, vamos definir que quando um elemento do conjunto S, digamos s, é usado mais de uma vez nos k subconjuntos, então ele deve ser penalizado. Denotamos por $w_p(s, i_s)$ o peso penalizado de s, onde i_s é número de vezes que o elemento s foi utilizado.

Uma instância para o problema seria: $S = \{a, b, c, d, e, f\}$ (onde os pesos dos elementos de S são $w(a) = 3$, $w(b) = 4$, $w(c) = 7$, $w(d) = 11$, $w(e) = 13$ e $w(f) = 18$), $k = 3$ e a função de penalização escolhida poderia ser $w_p(e, i) = iw(e)$. Adotando esta função de penalização temos que se $w(e) = 3$, então $w_p(e, 0) = 0, w_p(e, 1) = 3, w_p(e, 2) = 6, \ldots, w_p(e, n) = nw(e)$. Esta função de penalização é bastante simples e devido à sua simplicidade será exaustivamente usada no decorrer do texto.

Dada a instância acima, podemos ter diversas soluções viáveis para o problema, mas nem todas elas serão uma solução ótima. Por exemplo, a solução $\{a, b\}$, $\{c, d\}$ e $\{e, f\}$ tem peso total penalizado igual a 56. O peso total penalizado para esta solução é igual ao peso total não penalizado, pois esta é uma solução onde os subconjuntos são completamente disjuntos, ou seja, não há elementos repetidos (apenas lembrando, $w_p(e, 1) = w(e)$). Outra solução viável, cujos subconjuntos não são totalmente disjuntos, seria $\{a, b\}$, $\{b, c\}$ e $\{d, e\}$. E esta solução é melhor do que a primeira, pois tem peso total 50. Outras soluções menos disjuntas podem ser ainda melhores, como a solução ótima $\{a, b\}$, $\{a, b\}$ e $\{c, d\}$, cujo peso total penalizado é 46. Mas, uma solução menos disjunta do que a anterior, como $\{a, b\}$, $\{a, b\}$ e $\{a, d\}$, não é tão boa, pois tem peso total 54.

No exemplo acima é possível notar que uma solução ótima para o problema combinatório de congestionamento que mencionamos é um compromisso entre o peso e a

disjunção da solução. Este compromisso pode ser ajustado de acordo com a penalização utilizada. Ou seja, para encontrarmos soluções mais disjuntas podemos escolher outra função de penalização cujo crescimento seja maior do que o da função $w_p(e,i) = iw(e)$. Ou o contrário, se soluções menos disjuntas forem satisfatórias.

1.2 Árvores mínimas

Um exemplo interessante de problema combinatório de congestionamento é o problema de encontrar k árvores espalhadas, tal que o custo total penalizado seja mínimo. Para a simplicidade do texto chamaremos este problema de kMSTc (*k Minimum Congestion Spanning Trees*). Nesta dissertação o problema kMSTc será estudado em detalhes, tanto em seus aspectos teóricos, quanto experimentais.

Na abordagem teórica, mostraremos em detalhes um algoritmo polinomial para kMSTc. O algoritmo é baseado em uma redução de kMSTc ao problema de encontrar k árvores espalhadas disjuntas nas arestas e de peso total mínimo (*k Minimum Disjoint Spanning Trees*, ou simplesmente kMSTd).

Na abordagem experimental, mostraremos algumas heurísticas para resolver kMSTc que apresentaram soluções de qualidade e de baixo custo computacional. Cabe dizer que as abordagens experimentais foram motivadas pelo fato de inicialmente não sabermos a complexidade de kMSTc. Cogitávamos a possibilidade de kMSTc ser um problema NP-Difícil.

1.3 Definições e notações

Antes de prosseguir, vamos fazer um apanhado das definições e das notações usadas nos próximos capítulos; porém, estão aqui inclusas apenas as notações e nomenclaturas que consideramos não usuais, ou que merecem ser relembradas.

Dado um grafo $G = (V, E)$ não orientado, onde $|V| = n$ e $|E| = m$, uma *árvore espalhada* de G é um subgrafo conexo e acíclico de G com n vértices e $n-1$ arestas. Duas arestas são ditas paralelas se possuem seus dois extremos em comum, como as mostradas na figura 1.1. Dado um multigrafo ponderado G e um conjunto de arestas pertencentes a G e paralelas entre si, dizemos que estas arestas são *contíguas* se e somente se não existe outra aresta que seja paralela às arestas desse conjunto e que tenha peso menor do que o peso da aresta mais pesada do conjunto. Ou seja, são um bloco de arestas vizinhas em uma ordenação por pesos e as arestas de menor peso sempre fazem parte do conjunto. Na figura 1.1, as arestas a e b e as arestas a, b e c formam conjuntos de arestas contíguas entre si. Mas não formam um conjunto de arestas contíguas as arestas a e c, e nem as

arestas b e c.

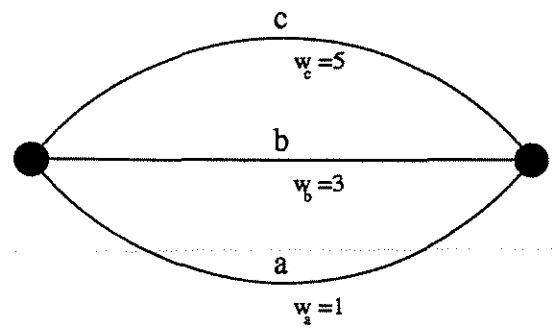


Figura 1.1: Arestas paralelas.

Representamos uma aresta pelo par de vértices que a definem. Ou seja, se os vértices x e y são os extremos de uma aresta e, então $e = (x, y)$. Dado um grafo $G = (V, E)$, usamos $|G|$ para representar o número de vértices $|V|$ e $||G||$ para representar o número de arestas $|E|$. Dado $G = (V, E)$ e um conjunto S, tal que $S \subseteq V$, usamos $G[S]$ para denotar o subgrafo de G induzido pelos vértices de S. Para grafos ponderados usaremos $w(e)$ para denotar o peso de uma aresta e e $w_p(e, i_e)$ para denotar o peso penalizado de e, onde i_e é o número de vezes que um elemento e foi utilizado.

1.4 Estrutura do texto

Como mencionamos, o objetivo principal desta dissertação foi estudar kMSTc. Para isso, fizemos abordagens teóricas e experimentais que mostraremos em detalhes no decorrer deste trabalho. Além disso, aplicamos a estratégia de solução encontrada para kMSTc para outros problemas combinatórios de congestionamento.

No próximo capítulo serão tratados os aspectos teóricos do problema kMSTc. Serão apresentados uma redução polinomial de kMSTc para kMSTd, um algoritmo polinomial para kMSTd, um algoritmo polinomial para kMSTc baseado no algoritmo para kMSTd e as respectivas análises de complexidade.

Os aspectos experimentais de kMSTc serão abordados no capítulo 3. Neste capítulo, veremos a descrição de algumas heurísticas e as comparações (de desempenho e de qualidade) entre as heurísticas propostas e o algoritmo exato.

Após explorarmos kMSTc, teórica e experimentalmente, no capítulo 4 exploraremos outros problemas combinatórios de congestionamento, como o problema dos caminhos mínimos, o problema dos emparelhamentos e o problema dos roteamentos.

Por fim, no capítulo de conclusões resumimos as contribuições deste trabalho e o que enxergamos no horizonte dos trabalhos futuros.

Capítulo 2

Árvores mínimas: algoritmos exatos

Um problema combinatório que envolve congestionamentos é: dado um grafo ponderado $G = (V, E)$ e um inteiro k, encontrar k árvores espalhadas $T_1, T_2, \ldots, T_k$ de G, tal que o peso total penalizado seja mínimo. Este problema, que chamamos de kMSTc, será abordado e resolvido neste capítulo.

2.1 Abordagens iniciais

Para encontrarmos uma solução para kMSTc, que foi o primeiro problema combinatório de congestionamento em que trabalhamos, tentamos vários caminhos. O primeiro deles foi modelar kMSTc usando fluxo em redes. Porém, nesta abordagem não conseguimos garantir que as arestas retornadas como resposta pudessem ser particionadas em k árvores espalhadas. Outra tentativa foi usar programação inteira para modelar kMSTc.

2.1.1 Abordagem combinatória

Resolver kMSTc implica minimizar a expressão

$$\Gamma = \sum_{j=1}^{k} \sum_{e \in T_j} w_p(e, i_e) \tag{2.1}$$

Observe que se uma aresta e pertence a i_e árvores, ela contribuirá i_e vezes com $w_p(e, i_e)$ para o somatório da equação 2.1. Portanto a função a ser minimizada pode ser reescrita como

$$\Gamma = \sum_{e \in E} i_e w_p(e, i_e) \tag{2.2}$$

Dado que $w_p(e, i_e)$ deve ser crescente em função de i_e para caractecterizar um congestionamento, então kMSTc pode ser reduzido a um problema de programação não-linear inteira, mas estes problemas são em geral NP-difíceis [16].

Apesar de nossas abordagens iniciais *sugerirem* a não existência de um algoritmo eficiente para o problema, encontramos um algoritmo polinomial para kMSTc.

2.2 Uma solução para kMSTc

Uma das estratégias para resolver um problema de solução desconhecida é reduzi-lo a um problema conhecido. A solução encontrada para o kMSTc consiste em reduzi-lo ao problema kMSTd. O kMSTd consiste em: dado um multigrafo ponderado $G' = (V, E')$ e um inteiro k, encontrar k árvores espalhadas de G', tal que estas árvores sejam disjuntas nas arestas e tal que o peso da solução seja mínimo. Ou seja, queremos k árvores disjuntas $T_1, T_2, \ldots, T_k$, tal que $\sum_{j=1}^{k} \sum_{e \in T_j} w(e)$ seja mínimo.

A redução consiste em, dadas as entradas para kMSTc, um grafo ponderado $G = (V, E)$ e um inteiro k, construir o multigrafo $G' = (V, E')$, tal que para cada aresta $e = (x, y) \in E$ existem k arestas, $e_1 = (x, y), e_2 = (x, y), \ldots, e_k = (x, y)$, em E', onde o peso da aresta e_j é dado por

$$w(e_j) = j w_p(e, j) - (j-1) w_p(e, j-1) \qquad para\ 1 \leq j \leq k$$

Onde $w_p(e, j) = 0$. A figura 2.1 exemplifica a construção do grafo G'. A figura 2.1.a é o grafo ponderado original G e a figura 2.1.b é o grafo G' para $k = 3$. No exemplo, se $w(e) = 3$ e $w_p(e, j) = jw(e)$, então os pesos das arestas e_1, e_2 e e_3 serão, respectivamente, $w(e_1) = 3$, $w(e_2) = 9$ e $w(e_3) = 15$. Construir o grafo G' custa $O(n + km) = O(km)$.

Figura 2.1: Construção de G' para $k = 3$.

Os problemas kMSTc e kMSTd estão diretamente relacionados, pois dada uma solução X para kMSTc, podemos facilmente construir uma solução Y para kMSTd (e vice-versa),

tal que X e Y tenham o mesmo peso. Para isto basta fazer cada árvore da resposta X corresponder a uma árvore disjunta em Y, desde que esta correspondência observe a ordem de peso das arestas (isto é, devem ser usadas apenas arestas contíguas). Por exemplo, sendo uma aresta e usada j vezes em X de kMSTc, na solução Y para kMSTd devemos usar as arestas contíguas $e_1, e_2, \ldots, e_j$. Assim, a contribuição de e para o peso total de X e a contribuição de e para o peso total de Y serão de mesmo valor. Isto é fácil de ser notado quando o valor destas contribuições é expandido. A contribuição de e para X é $jw_p(e,j)$, e para Y é $\sum_{j=1}^{k} jw_p(e,j) - (j-1)w_p(e,j-1)$, para $1 \leq j \leq k$. Se abrirmos o somatório veremos que $\sum_{j=1}^{k} jw_p(e,j) - (j-1)w_p(e,j-1) = jw_p(e,j)$.

É importante lembrar que toda solução mínima para kMSTd deve conter apenas arestas contíguas, caso contrário facilmente construímos outra solução de menor peso. Podemos executar uma construção inversa à do parágrafo anterior para construir uma resposta para kMSTc a partir de uma resposta para kMSTd, ambas com o mesmo peso.

Teorema 2.1 *Os problemas* k*MSTc e* k*MSTd são equivalentes.*

Prova. Como sabemos construir uma resposta para kMSTc a partir de uma resposta para kMSTd e vice-versa, provar que kMSTc e kMSTd são problemas equivalentes implica apenas mostrar que para toda solução mínima X de kMSTc, a solução Y correspondente é mínima para kMSTd (e vice-versa).

Se X é uma solução ótima para kMSTc e Y não é ótima para kMSTd, então existe uma solução Y', tal que $Y' < Y$. Se existe Y', podemos construir uma solução com o mesmo peso que Y' para kMSTc, digamos X', tal que $X' < X$, contradizendo a hipótese de otimalidade de X. O mesmo raciocínio pode ser usado para mostrar que a solução correspondente a uma solução ótima para kMSTd é uma solução ótima para kMSTc. □

Então, se tivermos um algoritmo polinomial para kMSTd, teremos um algoritmo polinomial para kMSTc. Nas próximas duas seções, veremos um algoritmo $O(m \log m + k^2 n^2)$ para kMSTd. Primeiro veremos um algoritmo para grafos (seção 4.3) e a seguir veremos a extensão deste algoritmo para multigrafos (seção 4.4).

2.3 Um algoritmo para kMSTd em grafos

Nesta seção vamos mostrar um algoritmo proposto por *Roskind* e *Tarjan* para resolver kMSTd em grafos. Para simplicidade do texto, nesta seção vamos usar kMSTd para nos referir especificamente ao problema em grafos. O problema em multigrafos será abordado na próxima seção.

O problema kMSTd foi inicialmente estudado por Tutte [23] e por Nash-Williams [20], que encontraram independentemente uma condição necessária e suficiente para a existência de k árvores espalhadas de G que fossem disjuntas nas arestas.

Teorema 2.2 (Tutte e Nash-Williams) *Um grafo $G = (V, E)$ possui k árvores espalhadas disjuntas nas arestas se e somente se para toda partição P de V*

$$|E_G\langle P\rangle| \geq k(|P| - 1)$$

onde $E_G\langle P\rangle$ é o conjunto das arestas de G que possuem extremos em componentes distintas de P e $|P|$ é o número de componentes de P. □

Uma prova para o teorema 2.2, mais simples do que as originais, pode ser encontrada em [8]. Outra prova, esta algorítmica, para o teorema foi dada por Feofiloff e Lucchesi [10]. O teorema serviu ainda como base para diversos outros trabalhos relacionados, como por exemplo o trabalho de Francisco Barahona [26].

Edmonds [9] e Clausen e Hansen [6] mostraram que kMSTd pode ser representado por um matróide $M = (E, I)$ (veja o apêndice A para a definição de matróides), onde E é o conjunto de arestas de um grafo ponderado $G = (V, E)$ e I é o conjunto de todos os subconjuntos $A \subseteq E$, tal que A pode ser particionado em k florestas. O conjunto I é chamado conjunto dos conjuntos independentes e o teste que define se A pertence a I é chamado teste de independência. Dado que kMSTd é um matróide, então existe um algoritmo guloso para encontrar um conjunto A maximal e de peso mínimo (esta é uma propriedade dos matróides). Repare que tal conjunto A é uma solução ótima para kMSTd: pois é maximal (a adição de mais uma aresta ao conjunto implicará criação de ciclos em pelo menos uma das k florestas) e tem peso mínimo.

Roskind e Tarjan [22] resolveram kMSTd apresentando uma implementação eficiente de um algoritmo guloso para o matróide $M = (E, I)$. Em poucas palavras, o algoritmo guloso consiste em construir uma ordenação das arestas de E, e segundo esta ordenação adicionar arestas, digamos e, a um conjunto, digamos Γ, enquanto $\Gamma \cup \{e\}$ puder ser particionado em k florestas.

Algoritmo guloso
$\Gamma \leftarrow \emptyset$
Ordenar as arestas de G em ordem não decrescente de pesos
Para cada aresta e tomada segundo a ordenação
  Se $\Gamma \cup \{e\} \in I$
    $\Gamma \leftarrow \Gamma \cup \{e\}$
  Fim Se

Fim Para
Retornar Γ
Fim

Como o algoritmo proposto por Roskind e Tarjan para kMSTd é a base para a solução de kMSTc, nas próximas seções vamos descrevê-lo detalhadamente. Para isto, veremos como é feito o teste de indepêndencia que determina se um conjnto de arestas pode ser particionado em k florestas, veremos as estruturas de dados necessárias para construir estas partições e, por fim, veremos a implementação completa e a respectiva análise de complexidade.

As subseções desta seção estão fortemente baseadas no artigo de Roskind e Tarjan, onde fortemente baseadas significa que utilizamos a mesma divisão de lemas e teoremas e que alguns dos lemas e respectivas demonstrações apresentadas são simples traduções do texto que consta em [22].

2.3.1 Teste de independência

Por teste de independência entende-se, dado um conjunto independente $\Gamma \in I$ e um elemento e, verificar se $\Gamma \cup \{e\}$ também pertence a I. Para o problema kMSTd, Γ é um conjunto de arestas que podem ser particionadas em k florestas e mostrar que $\Gamma \cup \{e\}$ é independente significa encontrar uma partição de $\Gamma \cup \{e\}$ em k florestas. Para isto, podemos aproveitar o fato de conhecermos uma partição de Γ em k florestas e apenas atualizar esta partição, de modo que a aresta e possa ser adicionada à essa nova partição sem criar ciclos. Para mostrar como encontrar tal atualização precisamos definir o que é uma *seqüência aumentante*, entre outras definições.

Se F é uma floresta e $e = (v, w)$ é uma aresta tal que v e w estão numa mesma árvore de F, definimos $F(e)$ como o caminho em F que une v a w. Note que como F é uma floresta este caminho é único. Definimos $i+$ como $(i \ mod \ k) + 1$, onde i é um inteiro qualquer. Chamamos de *seqüência de troca* uma seqüência de arestas $e_0, e_1, \ldots, e_l$, tal que $e_{j+1} \in F_{j+}(e_j)$, para todo $j \in \{0, 1, \ldots, l-1\}$. Dizemos que uma seqüência de troca é *aumentante* se e_0 não está em nenhuma das florestas $F_1, F_2, \ldots, F_k$ e se os extremos de e_l estão em árvores diferentes de F_{l+}.

Dada uma seqüência aumentante $e_0, e_1, \ldots, e_l$ e k florestas $F_1, F_2, \ldots, F_k$, atualizamos as florestas e aumentamos o tamanho de $\Gamma = \cup_{i=1}^{k} F_i$ fazendo $F_{j+} \leftarrow F_{j+} \cup \{e_j\} - \{e_{j+1}\}$, para todo $j \in \{0, 1, \ldots, l-1\}$, e adicionando a aresta e_l à floresta F_{l+}. Chamamos este processo de *atualização*.

Para entender as definições acima, vejamos um exemplo. Queremos incluir a aresta (a, b) na partição de arestas mostrada na figura 2.2, mas não podemos fazê-lo diretamente

pois ela cria um ciclo em todas as florestas. Porém, podemos fazê-lo indiretamente através da seqüência aumentante $(a, b), (a, c), (c, d)$.

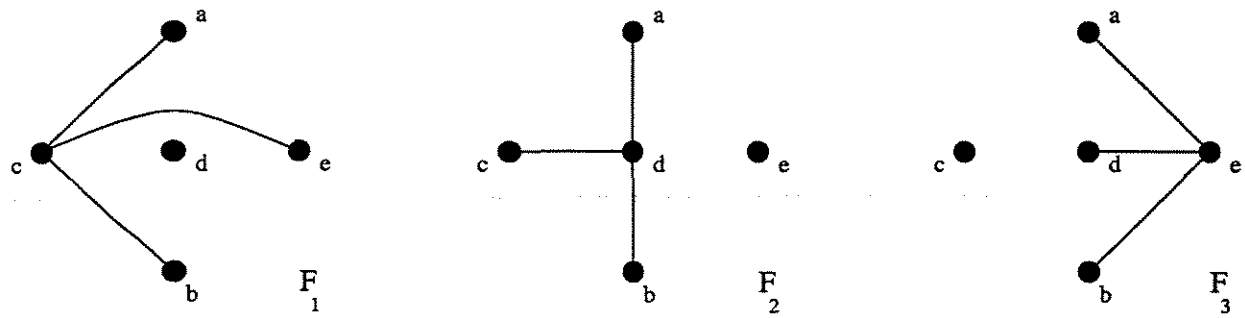


Figura 2.2: Partição em três florestas.

A seqüência de arestas $(a, b), (a, c), (c, d)$ é uma seqüência aumentante para a partição mostrada na figura 2.2, pois a aresta (a, b) não está em nenhuma das florestas, $(a, c) \in F_1(a, b)$, $(c, d) \in F_2(a, c)$ e os extremos de (c, d) estão em árvores diferentes de F_3, como exigem as condições para que uma seqüência seja aumentante. A figura 2.3 mostra as florestas F_1, F_2 e F_3 após a sua atualização. No exemplo fizemos $F_1 \leftarrow F_1 \cup \{e_0\} - \{e_1\}$, $F_2 \leftarrow F_2 \cup \{e_1\} - \{e_2\}$ e $F_3 \leftarrow F_3 \cup \{e_2\}$.

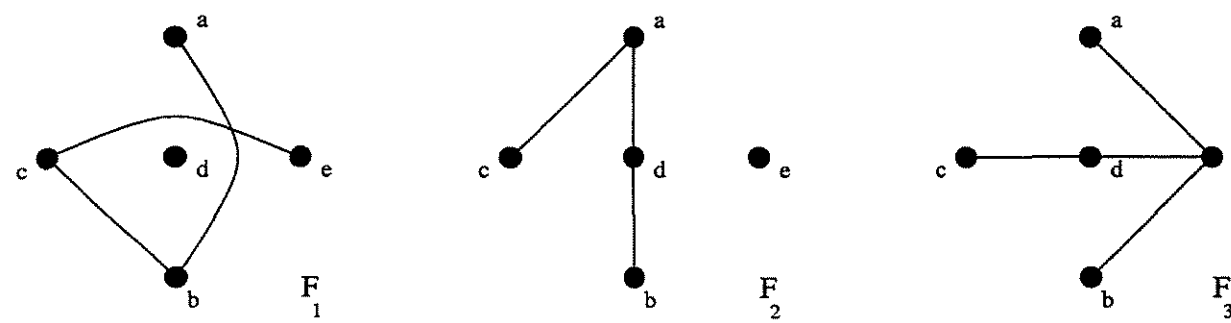


Figura 2.3: Partição após a atualização.

O teste de independência de $\Gamma \cup \{e_0\}$ consiste em procurar uma seqüência aumentante. Se encontrarmos, fazemos a respectiva atualização e continuamos com a execução do algoritmo. Mas, se não encontrarmos uma seqüência aumentante, então provamos que $\Gamma \cup \{e_0\}$ não é independente (isto será provado no lema 2.3).

Para procurar uma seqüência aumentante, usamos um algoritmo para rotulação de arestas. O algoritmo consiste basicamente em uma busca em largura nas arestas "alcançáveis" por uma seqüência de troca que começa com a aresta e_0. Para marcarmos se uma aresta já foi alcançada, atribuímos à aresta um *rótulo*, que é o "nome"da aresta que a alcançou pela primeira vez. Uma vez atribuído um rótulo para uma aresta, o valor

deste não muda até o fim do processo de rotulação. No início da fase de rotulação todas as arestas te êm seu rótulo igual a *null*.

Para manipular os rótulos durante o algoritmo usamos a função $label(e)$. Esta função retorna o rótulo da aresta e, se a aresta e ainda não tem um rótulo, então $label(e) = null$. E ainda, $label(e) \leftarrow e'$ denota a atribuição do rótulo e' à aresta e. No algoritmo também usamos a função $index(e)$ para retornar o índice i de uma floresta F_i, tal que a aresta $e \in F_i$. Se não existir tal floresta (ocorre se $e = e_0$), então definimos que $index(e) = 0$. Usamos ainda uma fila para implementar a busca em largura.

Vejamos como é o algoritmo de rotulação que testa a independência de $\Gamma \cup \{e_0\}$, onde Γ é a união das florestas $F_1, F_2, \ldots, F_k$.

Algoritmo de rotulação
Crie uma fila e a inicialize com e_0
Enquanto a fila não estiver vazia
Tire a primeira aresta, digamos $e = (v, w)$, da fila
$i = (index(e) \ mod \ k) + 1$
Se v e w estão em árvores diferentes de F_i
Pare, pois há uma seqüência aumentante.
Senão
Para toda aresta e' em $F_i(e)$ que ainda não foi rotulada **faça**
$label(e') \leftarrow e$
Fim Para
Adicione estas arestas à fila na ordem que aparecem em $F_i(e)$
Fim Se
Fim Enquanto
Fim

A figura 2.4 mostra como ficaram os rótulos das florestas da figura 2.2 após a execução do algoritmo de rotulação com relação à aresta $e_0 = (a, b)$. Na figura 2.4, os índices das arestas representam a ordem em que as arestas foram rotuladas e, conseqüentemente, processadas. Neste exemplo encontramos uma seqüência aumentante após processar cinco arestas ((a, b), (a, c), (b, c), (a, d) e (c, d)) e rotular sete arestas ((a, c), (b, c), (a, d) , (c, d), (b, d), (a, e) e (d, e)). O algoritmo parou quando processou a aresta (c, d), que pertence a F_2, e verificou que os vértices c e d pertenciam a florestas diferentes de F_3.

Se o algoritmo de rotulação parar por ter encontrado uma aresta e, tal que os extremos de e estão em árvores diferentes de uma floresta F, podemos construir facilmente a respectiva seqüência aumentante executando o algoritmo a seguir.

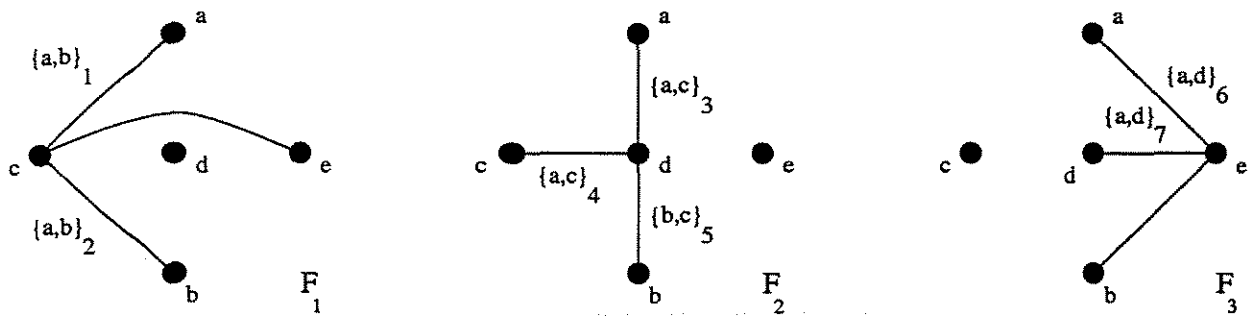


Figura 2.4: Florestas rotuladas.

Algoritmo de recuperação

Enquanto *label(e)*$\neq e_0$

Coloque e no começo da lista.

Faça $e \leftarrow$*label(e)*

Fim Enquanto

Coloque e_0 no começo da lista.

Retorne a lista.

Fim

A execução do algoritmo de recuperação nas florestas rotuladas da figura 2.4 retornaria a seqüência aumentante $(a,b),(a,c),(c,d)$. Esta seqüência é a mesma que usamos para adicionar a aresta (a,b) às florestas da figura 2.2, resultando nas florestas mostradas na figura 2.3.

O algoritmo de rotulação pode parar por dois motivos. Primeiro, se encontrou uma seqüência aumentante (como no exemplo acima). Neste caso podemos fazer a respectiva atualização e continuar com a execução do algoritmo guloso testando a independência da aresta seguinte a e_0 na ordenação por pesos.

O segundo motivo pelo qual o algoritmo de rotulação pode parar é se a fila ficar vazia. Neste caso o lema 2.3 mostra que $\Gamma \cup \{e_0\}$ não é independente. Ou seja, $\Gamma \cup \{e_0\}$ não pode ser particionado em k florestas.

Lema 2.3 *Se o algoritmo pára devido a fila estar vazia, então* $\Gamma \cup \{e_0\}$ *é dependente.*

Prova. Queremos mostrar que se o algoritmo parou porque a pilha ficou vazia, três propriedades são verdadeiras, e sendo verdadeiras implicam a dependência de $\Gamma \cup \{e_0\}$. A primeira propriedade é que $S_1 = S_2 = \ldots = S_k$, onde S_i é o conjunto dos vértices tocados pelas arestas rotuladas de F_i. A segunda é que os extremos de e_0 estão contidos em S,

onde S é o valor comum de $S_1, S_2, \ldots, S_k$. E a terceira é que as arestas rotuladas de F_i formam uma árvore espalhada de S, para todo $i \in \{1, 2, \ldots, k\}$.

Para verificar as duas primeiras condições, chamamos de e_i uma aresta rotulada qualquer de F_i. Dado que o passo de rotulação foi aplicado a e_i sem encontrar uma seqüência aumentante, os extremos de e_i estão na mesma árvore de F_{i+} e toda aresta que pertence ao caminho $F_{i+}(e_i)$ é rotulada. Isto implica que $S_i \subseteq S_{i+}$ e, por indução, que $S_1 = S_2 = \ldots = S_k$. O mesmo argumento mostra que os extremos de e_0 estão em $S_1 = S$.

Para verificar a terceira e última condição devemos mostrar que existe um caminho de x até y em F_i composto apenas por arestas rotuladas para todo $i \in \{1, 2, \ldots, k\}$, onde x é um dos extremos de e_0 e y é um vértice pertencente a S. Vamos supor que isto não seja verdade, ou seja, existe $e \in F_{i+}$ tal que para algum extremo x de e_0 e para algum extremo y de e (e é uma aresta rotulada), não existe um caminho unindo x a y em F_{i+} que seja composto apenas por arestas rotuladas. Seja e' o rotulo da aresta e (ou seja, $e \in F_{i+}(e')$ e e não havia sido rotulada até a aplicação da rotulação usando e'). Por construção existe um caminho de arestas rotuladas em F_i ligando x e um extremo de e', digamos z. Cada aresta deste caminho define um caminho de arestas rotuladas em F_{i+}, então, x e z são ligados por caminhos de arestas rotuladas em F_{i+}. Uma vez que tanto y (porque $e \in F_{i+}(e')$ e y é um dos extremos de e) quanto z (porque z é um dos extremos de e') pertencem a $F_{i+}(e')$, onde todas as arestas são rotuladas, também existe um caminho de arestas rotuladas entre y e z em F_{i+}. Então x e y são ligados por um caminho de arestas rotuladas em F_{i+}, o que contradiz a hipótese. A contradição gerada prova a validade da terceira condição. As condições implicam que $\Gamma \cup \{e_0\}$ contém $k(|S| - 1) + 1$ arestas com ambos os extremos em S. Portanto $\Gamma \cup \{e_0\}$ é dependente, dado que qualquer conjunto para ser independente deve ter no máximo $k(|S| - 1)$ arestas com ambos os extremos em S. □

Já mostramos que o algoritmo de rotulação encontra uma seqüência aumentante ou mostra que $\Gamma \cup \{e_0\}$ não pode ser particionado em k florestas. Deste modo, resta apenas mostrar que, dada uma seqüência aumentante, o processo de atualização das florestas resultará em k florestas.

Dada uma uma seqüência aumentante $e_0, e_1, \ldots, e_l$ e k florestas $F_1, F_2, \ldots, F_k$ é intuitivo que a atualização resultará em um novo conjunto independente quando $l \leq k$. Porém isto nem sempre é verdade. Para simplificar uma prova genérica precisamos definir o que é uma *seqüência aumentante minimal*.

Uma seqüência aumentante $e_0, e_1, \ldots, e_l$ é minimal se não existem i e j tais que $j > i + 1$ e $e_j \in F_{i+}(e_i)$. O lema 2.4 mostra que toda seqüência aumentante encontrada pelo algoritmo de rotulação é minimal.

Lema 2.4 *Se o algoritmo de rotulação encontrou uma seqüência aumentante, esta é minimal.*

Prova. Para provarmos que uma seqüência aumentante retornada pelo algoritmo de teste de independência é sempre minimal, vamos supor que o algoritmo retorne uma seqüência não minimal e então mostrar a contradição gerada.

Dada uma seqüência de arestas $e_0, e_1, e_{1+1}, \ldots, e_l$ não minimal, existem i e j tais que $j > i+1$ e que $e_j \in F_{i+}(e_i)$. Se $e_j \in F_{i+}(e_i)$ e $j > i$, então $label(e_j) = e_i$. Existe ainda uma aresta e_{i+1} tal que $label(e_{i+1}) = e_i$. Como $label(e_j) = e_i$ e $label(e_{i+1}) = e_i$, a seqüência aumentante em questão deveria ser $e_i, e_{i+1}, \ldots, e_i, e_j$, o que é uma contradição, pois uma aresta e_i não pode aparecer mais do que uma vez em uma seqüência. □

O lema 2.5 mostra que uma atualização é válida se a respectiva seqüência é aumentante e minimal.

Lema 2.5 *Após uma atualização por uma seqüência aumentante e minimal, toda floresta F_i continua sendo uma floresta, para $i \in \{1, 2, \ldots, k\}$.*

Prova. Seja $e_0, e_1, \ldots, e_l$ uma seqüência aumentante minimal, seja $i \in \{1, 2, \ldots, k\}$, seja Δ o maior número inteiro tal que $i + \Delta k \leq l$ e sejam F_i e F_i', respectivamente, F_i antes e depois de uma atualização. Considere o algoritmo mostrado na figura 2.5 como sendo o algoritmo que manipula uma floresta F, inicialmente igual a F_i.

Algoritmo
  Para $j = i, i+k, i+2k, \ldots, i+\Delta k$
    Adicione e_{j-1} a F.
    Contraia o ciclo gerado de forma que da ...
      ... contração resulte apenas um vértice.
  Fim Para
  Para $j = i + \Delta k, i + (\Delta - 1)k, \ldots, i$
    Expanda o ciclo formado em F pela adição de e_{j-1}.
    Retire e_j de F.
  Fim Para
Fim

Figura 2.5: Algoritmo para manipulação de uma floresta F.

Como a seqüência aumentante é minimal (conforme lema 2.4), cada passo de contração combina dois ou mais vértices de F. (Isto é, quando e_{j-1} é adicionado a F, seus extremos ainda não foram contraídos juntos.) Isto implica que depois de cada contração, ou expansão, F é uma floresta cujas árvores *tocam* o mesmo conjunto de vértices que as

árvores de F_i. O valor final de F é F_i' se $i \neq l+$ ou $F_i' - e_l$ se $i = l+$. Ambos os casos implicam que F_i' é uma floresta. □

Na próxima seção vamos apresentar uma característica que implica na não necessidade de executarmos o teste de independência (algoritmo de rotulação) para todas as arestas, permitindo assim melhorarmos o algoritmo apresentado.

2.3.2 Moitas

Dizemos que um conjunto de vértices C forma uma *moita*[1], se para toda floresta F_i, onde $i \in \{1, 2, \ldots, k\}$, o grafo induzido por C em F_i é uma árvore. Ou seja, todo F_i possui um subgrafo T_i que é uma árvore espalhada com relação aos vértices do conjunto C.

Se os extremos de uma aresta e pertencem a uma moita, podemos afirmar que $\Gamma \cup \{e\}$ não é independente. É facil verificar esta afirmação, pois para $\Gamma \cup \{e\}$ ser independente precisaríamos particionar $k(|C| - 1) + 1$ arestas em k florestas (onde $|C|$ é o número de vértices de cada floresta), o que sabemos ser impossível, dado que o número máximo de arestas de uma união de k florestas é $k(|C| - 1)$.

Lema 2.6 *Se durante o processo de busca por uma seqüência aumentante um conjunto de vértices C forma uma moita com relação a Γ e Γ é atualizado usando uma seqüência aumentante, então C é uma moita para o conjunto aumentado Γ'.*

Prova. Nenhuma das arestas que ligam os vértices de uma moita C e que pertencem a uma floresta F_i de Γ pode pertencer a uma seqüência aumentante. Uma vez que estas arestas permanecem intactas em todas as florestas, depois de uma atualização C continua sendo uma moita. □

O lema 2.6 é direto, "uma vez moita, sempre moita". O lema a seguir mostra que moitas maximais são disjuntas com relação aos vértices.

Lema 2.7 *Se dois conjuntos de vértices, A e B, são moitas e contêm um vértice x em comum, então $A \cup B$ é uma moita.*

Prova. Para quaisquer vértices $y \in A$ e $z \in B$ e qualquer $i \in [1 \ldots k]$, existe uma caminho em F_i de y até x contendo apenas vértices de A e outro caminho de x até z contendo apenas vértices de B, resultando que um caminho de y para z contém apenas vértices de $A \cup B$. Portanto $A \cup B$ é uma moita. □

[1] *Roskind* e *Tarjan* usam o termo *clump*. Dentre as diversas possíveis traduções mantivemos *moita*, devido às florestas, árvores, ...

Como qualquer aresta que possua extremos na mesma moita não pode estar em uma seqüência aumentante, podemos modificar o algoritmo para não executar o algoritmo de rotulação para as arestas com tal característica, pois tais buscas por uma seqüência aumentante são inúteis. E se executamos o algoritmo de rotulação para uma aresta $e_0 = (x, y)$ e não encontramos uma seqüência aumentante, podemos afirmar baseados na prova do lema 2.3 que x e y pertencem à uma mesma moita. Neste caso o lema 2.7 garante que podemos substituir as moitas que contêm x e y pela sua união. Quando iniciamos o algoritmo temos n moitas, onde cada moita corresponde a um, e apenas um, vértice do grafo. Usando esta estratégia, cada iteração do algoritmo de rotulação encontra uma seqüência aumentante (aumentando Γ) ou une duas moitas (diminuindo o número de moitas). Como o número de seqüências aumentantes necessárias para encontrar k árvores espalhadas disjuntas nas arestas é $k(n-1)$ e como a união de moitas pode ocorrer no máximo $n-1$ vezes, o número de iterações do algoritmo de rotulação será no máximo $k(n-1)+n-1$, implicando ordem $O(kn)$.

2.3.3 Estruturas de dados

O algoritmo deve armazenar as k florestas da solução corrente. Representamos uma floresta usando uma partição dos vértices. Nesta representação, se dois vértices pertencem à uma mesma árvore de uma floresta, então estes vértices pertencem à uma mesma componente da partição. Antes de executarmos o algoritmo, inicializamos k partições de vértices, onde cada partição corresponde a uma floresta e possui n componentes, onde um vértice diferente do grafo é atribuido a cada componente. Em suma, ao iniciarmos o algoritmo de *Roskind* e *Tarjan* temos k partições de vértices, onde cada partição tem n componentes e cada componente representa um vértice do grafo.

Devemos também manter as moitas conhecidas. Para representar as moitas também usamos uma partição dos vértices, mas neste caso fazemos com que cada componente da partição represente uma moita. Ou seja, se dois vértices pertencem a mesma moita, então eles pertencem à mesma componente da partição.

Precisamos então de $k+1$ partições, k para representar as florestas e uma para armazenar as moitas conhecidas. São duas as operações necessárias para manipular estas estruturas.

find(v): Retorna o índice da componente que contém o vértice v.

union(v,w): Une as componentes que contêm v e w, e atribui um índice para a nova componente. A operação destrói as componentes antigas.

Vamos usar $find_0$ e $union_0$ para denotar as operações relativas às moitas e $find_i$ e $union_i$ para as operações que manipulam a floresta F_i.

Usando estas estruturas, para sabermos se uma aresta $e = (x, y)$ tem seus extremos na

mesma árvore de uma floresta F_i, basta verificarmos se $find_i(x) = find_i(y)$. Da mesma forma, para sabermos se a aresta e pertence a uma moita, verificamos se $find_0(x) = find_0(y)$. Para implementar tais estruturas podemos usar um vetor com n posições, onde armazenamos o índice da componente de cada um dos n elementos. A inicialização do vetor gasta $\Theta(n)$ e consiste em atribuir i para cada posição i do vetor. A operação *find(v)* gasta $\Theta(1)$, pois apenas lê o valor da posição v do vetor. A operação *union(v,w)*, por sua vez, gasta $O(n)$, pois consiste em fazer $i \leftarrow find(v)$, $j \leftarrow find(w)$ e percorrer o vetor linearmente atribuindo i às posições do vetor cujo valor é j. Para o nosso caso basta uma implementação tão simples quanto esta, pois apesar de existirem implementações mais sofisticadas, a utilização destas não implica melhora no tempo assintótico de execução do algoritmo. Mas nos testes práticos que fizemos usamos implementações mais complexas.

Precisamos armazenar para todas as arestas e de G as informações $label(e)$ e $index(e)$. Vamos considerar que tanto a leitura quanto a escrita destas informações gaste tempo $\Theta(1)$.

Na próxima seção vamos descrever o algoritmo de *Roskind* e *Tarjan* e sua implementação.

2.3.4 O algoritmo para kMSTd em grafos e sua complexidade

Uma característica importante do algoritmo de *Roskind* e *Tarjan* é usar uma propriedade das arestas rotuladas para ser mais eficiente. Esta propriedade é aproveitada no trecho do algoritmo que rotula as arestas. Sendo $e_0 = (x, y)$, chamamos de *raiz* um vértice arbitrariamente escolhido entre x e y, digamos x, e denotamos por T_i a árvore de F_i que contém a raiz x.

Lema 2.8 *As arestas rotuladas em F_i formam uma árvore que toca o vértice x e consiste em um subgrafo de T_i. Esta árvore pode inclusive ser igual a T_i. E quando uma aresta $e = (v, w)$ é processada no passo de rotulação, pelo menos um dos vértices v e w é tocado por uma aresta rotulada de T_i, onde $i = (index(e) \bmod k) + 1$.*

Prova. Se rotularmos as arestas numa ordem tal que a primeira aresta a ser rotulada é sempre a que compartilha um vértice com outra aresta anteriormente rotulada, a primeira parte do lema é imediata, visto que quando é rotulada qualquer aresta de F_i tem x como extremo ou compartilha um extremo com uma aresta já rotulada em F_i (no algoritmo esta propriedade será implementada por um mecanismo de pilha). Prova-se a segunda parte por indução no número de arestas processadas. Seja $e = (v, w)$ uma aresta processada, $x \in \{v, w\}$ ou existe pelo menos uma aresta anteriormente processada em F_i com um extremo em comum com e, digamos v. Então v está em T_i pela hipótese de indução. □

A contribuição do lema 2.8 para a eficiência é que para rotular as arestas em $F_i(e)$, onde $e = (v, w)$, podemos simplesmente verificar qual dos vértices de e não é extremo de uma aresta anteriormente rotulada, digamos v. Assim, resta traçar um caminho de v até x (vértice raiz) e rotular as arestas que pertencem a este caminho e que ainda não foram rotuladas. (Se tanto v quanto w forem extremos de uma aresta já rotulada então não existem arestas para serem rotuladas.) Mas é importante que estas arestas sejam adicionadas à fila de forma que sempre compartilhem um extremo com uma aresta rotulada, ou seja, no sentido que aparecem no caminho de x para v.

Para facilitar o trabalho de traçar um caminho de v até x, vamos fazer uma pré-computação antes de executarmos a rotulação. Neste processamento armazenamos o valor da função *"pai"* $p_i(v)$ para todo vértice v em T_i, onde $p_i(v)$ é o vizinho de v no caminho de v até x em T_i. Por exemplo, se existe um caminho em T_i, tal que para ir do vértice b para o vértice raiz x usamos a aresta (a, b), então $p_i(b) = a$. Definimos ainda que $p_i(x) = x$ e $p_i(v) = null$ se $v \notin T_i$.

A figura 2.6 mostra o algoritmo completo de *Roskind* e *Tarjan*. As entradas do algoritmo são um grafo $G = (V, E)$ e um inteiro k, e o retorno é o conjunto de arestas (independente, maximal e de peso mínimo) Γ. Se conhecemos tal conjunto Γ de arestas, podemos facilmente construir as k florestas $F_1, F_2, \ldots, F_k$, onde $\Gamma = \cup_{i=1}^{k} F_i$, chamando $index(e)$ para cada aresta $e \in \Gamma$.

Como sabemos que ordenar as arestas de E custa $O(m \log m)$ e vimos na seção 2.3.2 que o número de execuções da rotulação é limitado por $O(kn)$, para facilitar a análise de complexidade podemos dividir o laço **Para**, que possui m iterações, em dois blocos. Um bloco que não executa a rotulação ($O(m - kn)$ iterações) e outro que executa a rotulação ($O(kn)$ iterações). Como as operações *find* usam tempo constante, o tempo de execução do primeiro bloco é $O(m)$. Assim o tempo total de execução do algoritmo é $O(m \log m + m + kn * f(k, n, m)) = O(m \log m + kn * f(k, n, m))$, onde $f(k, n, m)$ é o tempo de execução do algoritmo de rotulação.

O algoritmo requer duas inicializações, mas o tempo de execução destas é dominado pelo tempo de ordenação das arestas e pelo número de execuções do algoritmo de rotulação. As inicializações são: criar as $k + 1$ partições com n componentes (gasta tempo $O(kn)$) e fazer $index(e) \leftarrow 0$ para toda aresta e ($O(m)$).

O trecho do algoritmo que faz a rotulação (interno ao condicional Se $find_0(x) \neq find_0(y)$) requer também três inicializações. Primeiro, inicializar a fila com e_0, que custa tempo constante. Segundo, fazer $label(e) \leftarrow null$ para toda $e \in \Gamma$, que gasta $O(kn)$, dado que $|\Gamma| \leq k(n-1)$. E por fim, computar a função *pai*, que consiste em fazer uma busca em profundidade em cada uma das k florestas, sabemos que fazer uma busca em profundidade em um grafo qualquer é linear no tamanho do grafo ($O(n+m)$) [7]. Como os grafos em questão são florestas ($m < n$), cada busca custa $O(n)$. Sendo k o número de

Algoritmo de Roskind e Tarjan

Ordenar as arestas de E em ordem não decrescente de pesos
$\Gamma \leftarrow \emptyset$
Para cada $e_0 = (x, y) \in E$, tomada segundo a ordenação **faça**
Se $find_0(x) \neq find_0(y)$
//Inicializações para rotulação
Inicialize a fila com e_0
Inicialize com *null* todas as arestas que pertencem a Γ
Compute a função *pai* usando x como raiz
//Processo de rotulação
Enquanto a fila não estiver vazia
Retire a aresta $e = (v, w)$ da fila
$i = (index(e) \ mod \ k) + 1$
Se $find_i(v) \neq find_i(w)$
Saia do laço
Senão
Se $label(v, p_i(v)) = null$
$u \leftarrow v$
Senão
$u \leftarrow w$
Fim Se
Repita até $u = x$ ou $label(u, p_i(u)) \neq null$
Coloque $(u, p_i(u))$ na pilha
$u \leftarrow p_i(u)$
Fim Repita
Repita até esvaziar a pilha
Retire e' da pilha
$label(e') = e$
Coloque e' na fila
Fim Repita
Fim Se
Fim Enquanto
Se $find_i(v) \neq find_i(w)$
Construa uma seqüência aumentante
Faça a respectiva atualização
Senão
$union_0(x, y)$
Fim Se
Fim Se
Fim Para
Retornar Γ
Fim

Figura 2.6: Algoritmo de *Roskind* e *Tarjan*.

buscas, uma para cada árvore, gastamos $O(kn)$ para computar a função *pai*. Portanto o tempo gasto com as inicializações do algoritmo de rotulaçao é $O(1 + kn + kn) = O(kn)$.

Feitas as inicializações para a rotulação, devemos observar que o número de iterações do laço **Enquanto** é de $O(|\Gamma|) = O(kn)$, dado que uma aresta é adicionada à fila apenas uma vez por execução do algoritmo de rotulação. Repare ainda que uma iteração do laço **Enquanto** que não rotula nenhuma aresta gasta $\Theta(1)$ e que uma iteração que rotula x arestas gasta $O(x)$ e implica mais x iterações do laço **Enquanto**. Sabemos que uma execução da rotulação não pode ter mais do que $k(n-1)$ iterações que não rotulam arestas, pois cada uma destas iterações consome uma aresta da fila e o número máximo de arestas que pode ser adicionado à fila é $k(n-1)$, dado que existem no máximo $k(n-1)$ arestas em Γ e uma aresta de Γ só é adicionada uma vez à fila. Como cada uma destas iterações gasta $\Theta(1)$, o tempo total gasto com elas é $O(k(n-1)) = O(kn)$. Com relação às iterações que rotulam, podemos afirmar que elas não rotulam mais do que $k(n-1)$ arestas e portanto não gastam mais do que $O(k(n-1)) = O(kn)$ de tempo total. Como só ocorrem estes dois tipos de iteração, as que rotulam e as que não rotulam, podemos afirmar que o tempo gasto no laço **Enquanto** é $O(kn + kn) = O(kn)$.

Uma vez que a rotulação foi executada, ou encontramos uma seqüência aumentante, ou fazemos uma união de moitas. No primeiro caso devemos construir a seqüência aumentante encontrada (para isto podemos usar o algoritmo dado na página 12). Como o tempo de execução do algoritmo é proporcional ao tamanho da seqüência, que é limitado pelo tamanho de Γ, então podemos concluir que construir uma seqüência aumentante gasta $O(kn)$. Após encontrarmos uma seqüência devemos fazer a respectiva atualização. Dada a lista de arestas, que representa a seqüência aumentante, atualizamos as florestas executando o algoritmo mostrado na figura 2.7.

Algoritmo para atualização
    $i \leftarrow 0$
    Repita até a lista ficar vazia
        remova a aresta $e = (v, w)$ do começo da lista
        $index(e) \leftarrow i+$
        $i \leftarrow i + 1$
    Fim Repita
    $union_{i \bmod k}(v, w)$
Fim

Figura 2.7: Algoritmo para atualização.

Na atualização fazemos $O(kn)$ iterações do laço **Repita** (onde cada iteração gasta tempo constante) e uma chamada a *union* (que gasta $O(n)$). Assim a atualização é feita em $O(kn + n) = O(kn)$.

Para o caso de não termos encontrado uma seqüência aumentante, precisamos apenas gastar $O(n)$ fazendo uma chamada a $union_0$.

Resumindo, a fase de rotulação do algoritmo gasta $O(kn)$ com inicializações, gasta também $O(kn)$ no laço **Repita** e gasta $O(kn)$ para atualizar as florestas ou a moita, então o tempo total gasto com a execução do algoritmo de rotulação é $O(kn)$. Lembrando que o algoritmo de *Roskind e Tarjan* gasta $O(m \log m + kn * f(k,n,m))$, onde $f(k,n,m)$ é o tempo de execução do algoritmo de rotulação, fechamos a análise de complexidade do algoritmo concluindo que o tempo total de execução do algoritmo é $O(m \log m + k^2n^2)$. Isto completa a prova do teorema a seguir.

Teorema 2.9 (Roskind e Tarjan) *O problema kMSTd pode ser resolvido em* $O(m \log m + k^2n^2)$. □

2.4 Adaptação do algoritmo de Roskind e Tarjan para multigrafos

O algoritmo de *Roskind* e *Tarjan*, visto na seção anterior, foi inicialmente proposto para grafos. Nesta seção vamos mostrar que o algoritmo pode ser estendido para multigrafos.

Roskind e *Tarjan* definiram que para uma seqüência de arestas $e_0, e_1, \ldots, e_l$ ser aumentante, e_0 não deveria estar em nenhuma das florestas $F_1, F_2, \ldots, F_k$. Na verdade, este era apenas o modo mais simples de dizer, para grafos, que adicionando e_0 à Γ estaríamos aumentando o tamanho do conjunto Γ. Para multigrafos, deve-se esclarecer que as arestas $e_0 = (u,v)$ e $e_i = (u,v)$ são arestas diferentes e que e_0 e e_i podem estar em um conjunto Γ de arestas, tal que Γ pode ser particionado em k florestas disjuntas nas arestas.

O fato de os extremos de e_0 serem ligados por uma aresta e em uma floresta F de Γ não significa que Γ não possa ser aumentado com a adição da aresta e_0. (Obviamente e e e_0 jamais poderão estar na mesma floresta, pois formariam um ciclo.) Observando que e e e_0 são arestas diferentes, a definição de seqüência aumentante para grafos pode também ser usada para multigrafos.

A confusão gerada se resume à ocorrência de arestas e_{i-1}, e_i e e_j em uma seqüência aumentante, tal que e_i tem os mesmos extremos de e_j (por isso ditas *paralelas*), $j > i+1$ e $e_i \in F_{i \bmod k}(e_{i-1})$. Como e_i pertence ao caminho $F_{(i \bmod k)}(e_{i-1})$ e e_j é "igual" a e_i, caracterizamos uma seqüência não minimal que é perfeitamente passível de ser encontrada. Por exemplo, a figura 2.8 mostra um conjunto de florestas onde a seqüência de arestas $\{(a,d),(a,b),(a,c),(a,b),(a,c)\}$ é uma seqüência aumentante e gera as florestas da figura 2.9. Note que segundo a definição original (página 9) a seqüência não é minimal

e mesmo assim gerou um conjunto de florestas. No exemplo, as arestas e_{i-1}, e_i e e_j são, respectivamente, as arestas $e_1 = (a, b)$, $e_2 = (a, c)$ e $e_4 = (a, c)$.

Mas a seqüência é minimal porque e_j não pertence ao caminho $F_{(i \bmod k)}(e_{i-1})$. Para pertencer ao caminho, e_j precisaria pertencer a floresta $F_{(i \bmod k)}$, mas isto é impossível, pois a aresta e_i já pertence a esta floresta e e_i forma um ciclo com a aresta e_j, portanto a seqüência é minimal.

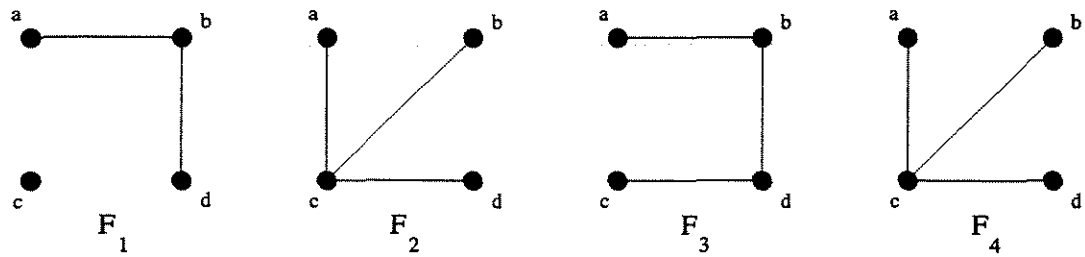


Figura 2.8: Árvores de um multigrafo.

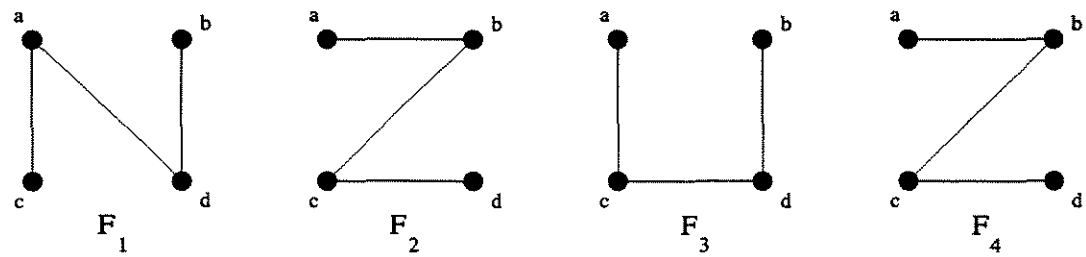


Figura 2.9: Árvores de um multigrafo após uma atualização.

Apesar da necessidade desta discussão para deixarmos claro que as seqüência encontradas para multigrafos também são minimais, a discussão é inócua por assumir a existência de arestas "iguais" em um conjunto de florestas **disjuntas** nas arestas.

Mas ainda precisamos prover o algoritmo com um mecanismo capaz de diferenciar arestas "iguais" em G'. Para isto, basta usarmos três parâmetros na identificação de uma aresta, os seus dois extremos e um índice, que inclusive pode ser usado para calcular o peso penalizado de uma aresta. Por exemplo, a aresta (x, y, l) de G' une os vértices x e y e seu peso penalizado é $lw_p((x, y), l) - (l-1)w_p((x, y), l-1)$.

Teorema 2.10 *O algoritmo de* Roskind e Tarjan *para kMSTd, em multigrafos, continua válido e tem tempo de execução* $O(m \log m + k^2 n^2)$.

Prova. Para mostrar que o algoritmo continua correto quando executado sobre multigrafos precisamos mostrar que os lemas, sobre os quais o algoritmo se baseia, continuam verdadeiros quando nos referimos a multigrafos. Uma vez que duas arestas "iguais" jamais

estarão numa mesma floresta, as expansões e contrações utilizadas no lema 2.5 continuam combinando dois ou mais vértices. Assim, o lema 2.5 continua válido, ou seja, dada uma seqüência aumentante minimal, a atualização é válida também para os multigrafos. Na prova do lema 2.4, usamos o fato de que uma aresta não aparece duas vezes em uma seqüência aumentante. Isto também é verdadeiro para multigrafos, pois são arestas diferentes apesar de terem os mesmos extremos.

Os lemas restantes (lemas 2.3, 2.6, 2.7 e 2.8) também continuam verdadeiros para os multigrafos. Dado que todos os lemas continuam válidos, então o algoritmo de *Roskind e Tarjan* também é correto para multigrafos.

Com relação à análise da complexidade do algoritmo para multigrafos, o limite assintótico continua sendo $O(m \log m + k^2 n^2)$, dado que as operações de rotulação, entre outras, continuam sendo executadas sobre o mesmo número de arestas ($O(kn)$) e as operações básicas, como *find* e *union*, continuam com os mesmos tempos de execução. □

Corolário 2.11 *O problema kMSTc pode ser resolvido em* $O(km \log km + k^2 n^2)$.

Prova. Segundo o teorema 2.1, resolver kMSTc se reduz a construir $G' = (V, E')$ e resolver kMSTd sobre G'. Construir G' pode ser feito em $O(km)$ e resolver kMSTd em G' custa $O(km \log km + k^2 n^2)$, dado que $|E'| = km$. Portanto, resolver kMSTc custa $O(km) + O(km \log km + k^2 n^2) = O(km \log km + k^2 n^2)$. □

Assim, aplicando de forma direta o algoritmo para kMSTd, temos um algoritmo $O(km \log km + k^2 n^2)$ para kMSTc. Mas, podemos aproveitar o fato de existir uma pré-ordenação nas arestas de G' para melhorar este limite.

2.5 Um algoritmo $O(m \log m + k^2 n^2)$ para kMSTc

O algoritmo apresentado na seção anterior gastaria $O(km \log km)$ para ordenar as km arestas de G', mas podemos aproveitar o fato de existir uma pré-ordenação das arestas de G' para melhorar o tempo gasto com esta ordenação. O algoritmo pode ser alterado, tal que o tempo gasto com a ordenação das arestas de G', $O(km \log km)$, pode ser reduzido a $O(m \log m)$. Para isto, precisamos implementar uma ordenação que aproveite o fato de que $w_p(e,1) < w_p(e,2) < \ldots < w_p(e,k)$. Esta implementação pode ser feita usando heaps binários.

Para manipular uma estrutura de heap precisamos de algumas operações básicas, como inserção e busca de elementos. Particularmente para os heaps binários, criar um heap custa $\Theta(1)$, encontrar o elemento de peso mínimo custa $\Theta(1)$, extrair um elemento do heap custa $O(\log n)$ e, também, custa $O(\log n)$ inserir um elemento novo no heap.

Podemos implementar a seguinte modificação no algoritmo de *Roskind* e *Tarjan*. Em vez de ordenar km arestas, como seria a aplicação direta do algoritmo à kMSTc, construímos um heap de tamanho m com as arestas do grafo original, o que custa $O(m \log m)$. Encontrar a próxima aresta de menor menor peso, com esta modificação, implica obter o menor elemento do heap, o que custa $\Theta(1)$. Digamos que este menor elemento seja a aresta $e = (x, y, i)$. Ao processar e podem ocorrer duas situações: e pertence a uma moita ou e não pertence a uma moita.

No primeiro caso, precisamos apenas extrair e do heap, o que custa $O(\log m)$. Como esta operação tem m como limite superior, pois existem m arestas, o custo total desta operação é $O(m \log m)$.

No segundo caso, executamos o teste de independência (algoritmo de rotulação) e atualizamos as árvores (se $\Gamma \cup \{e\}$ é um conjunto independente) ou atualizamos as moitas (se $\Gamma \cup \{e\}$ não é um conjunto independente). No caso em que $\Gamma \cup \{e\}$ é independente, após a respectiva atualização de Γ, precisamos atualizar a ordenação extraindo $e = (x, y, i)$ do heap e inserindo $e = (x, y, i + 1)$, o que custa $O(\log m)$. Assim, após cada execução do algoritmo de rotulação, que custa $O(kn)$, gastamos $O(\log m)$ para atualizar o heap. Deste modo, com a modificação do algoritmo de *Roskind* e *Tarjan*, gastamos $O(\log m + kn)$ em cada iteração do algoritmo de rotulação que resulta em uma seqüência aumentante. Porém, como $m = O(n^2)$, então $O(\log m + kn) = O(\log n^2 + kn) = O(2 \log n + kn) = O(kn)$. Ou seja, o tempo gasto com o processo de rotulação pelo algoritmo modificado é o mesmo gasto pelo algoritmo original, $O(kn)$.

A adaptação do algoritmo de *Roskind* e *Tarjan* para kMSTc ficaria como mostrado na figura 2.10.

Teorema 2.12 *O problema kMSTc pode ser resolvido em* $O(m \log m + k^2 n^2)$.

Prova. A inclusão do mecanismo de heap no algoritmo de *Rokind* e *Tarjan* não invalida a análise de correção do algoritmo para multigrafos, dada pelo lema 2.10, pois apenas a forma como as arestas são ordenadas mudou.

A análise de complexidade pode ser dividida em duas partes. A primeira parte consiste na construção de um heap binário com m elementos, $O(m \log m)$.

A segunda parte consiste na análise do laço **Enquanto.** As iterações deste laço também podem ser divididas em dois grupos: as iterações onde os extremos de e pertencem à uma mesma moita ($O(\log m)$ para retirar e do heap) e as iterações onde os extremos de e pertencem a moitas diferentes. O número de iterações do primeiro grupo é $O(m)$, então gastamos $O(m \log m)$ com o primeiro grupo. O número de iterações do segundo grupo é $O(kn)$, então gastamos $O(k^2 n^2)$ com o segundo grupo, pois cada iteração custa $O(kn)$.

Assim, o custo total do algoritmo é $O(m \log m + k^2 n^2 + m \log m) = O(m \log m + k^2 n^2)$.
□

Algoritmo para kMSTc
Construir um heap binário com as m arestas de G
Enquanto existirem arestas no heap
Encontrar a menor aresta do heap, digamos $e = (x, y, i)$
Se $find_o(x) \neq find_o(y)$
Se $\Gamma \cup \{e\}$ é um conjunto independente
Atualizar Γ
Atualizar o heap (retirar (x, y, i) e inserir $(x, y, i + 1)$)
Senão
Atualizar as moitas fazendo $union_0(x, y)$
Fim Se
Senão
Retirar e do heap
Fim Se
Fim Enquanto
Fim

Figura 2.10: Algoritmo para kMSTc.

Cabe ainda a observação de que este capítulo mostra um algoritmo polinomial para kMSTc supondo que k seja polinomial no tamanho do grafo de entrada.

Capítulo 3

Árvores mínimas: heurísticas

No capítulo anterior vimos um algoritmo $O(m \log m + k^2 n^2)$ para resolver kMSTc. Neste capítulo veremos duas heurísticas para kMSTc, as heurísticas são de simples implementação (não usam a complicada rotina de rotulação) e têm tempo de execução inferior ao do algoritmo exato (são lineares em k). Além disso, as soluções encontradas pelas heurísticas mostraram-se bastante próximas das soluções exatas.

3.1 Heurística A

A heurística A é intuitiva, de simples implementação e consiste em encontrar uma árvore espalhada T_i de G (para $1 \leq i \leq k$), onde a cada árvore T_i encontrada atualizamos convenientemente em G o peso das arestas usadas em T_i.

Heurística A
Para $i = 1$ até k
Encontre uma árvore espalhada mínima T_i de G.
Atualize em G o peso de toda aresta e, tal que $e \in T_i$.
Fim Para
Fim

A atualização dos pesos das arestas de G é feita com o objetivo de aproximar o comportamento da penalização usada. Por exemplo, se usarmos a função de penalização $w_p(e, j_e) = j_e w(e)$, então o peso de uma aresta e de G na i-ésima iteração deve ser $j_e w_p(e, j_e) - (j_e - 1) w_p(e, j_e - 1)$, onde j_e é o número de vezes que a aresta e foi usada nas árvores $T_1, T_2, \ldots, T_{i-1}$. A figura 3.1 mostra um grafo G e a respectiva árvore T_1 encontrada na primeira iteração da heurística.

Na figura 3.2 mostra o grafo G, atualizado segundo a penalização que descrevemos, e

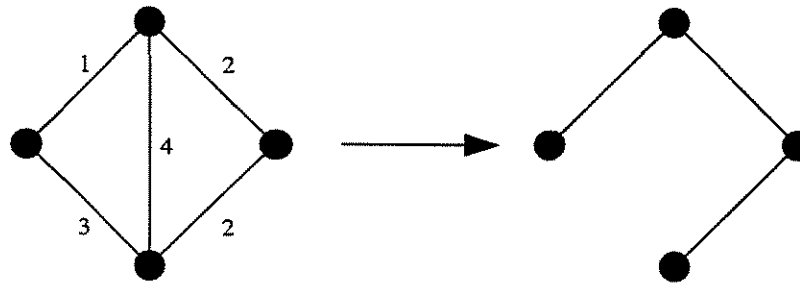


Figura 3.1: G na primeira iteração e a respectiva árvore.

a árvore T_2 encontrada na segunda iteração.

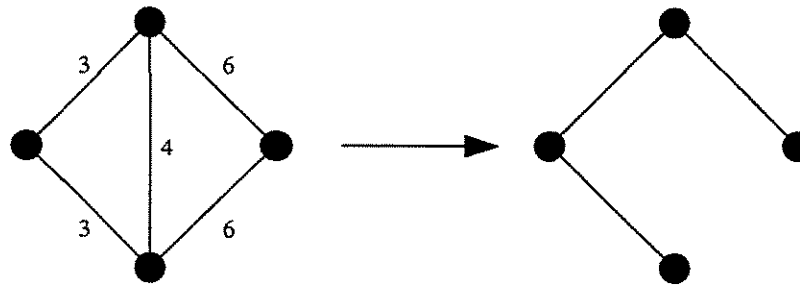


Figura 3.2: G na segunda iteração e a respectiva árvore.

Vale notar que as soluções encontradas pela heurística A e pelo algoritmo exato são diferentes, as figuras 3.3 e 3.4 mostram, respectivamente, as árvores que cada um dos métodos retornaria para $k = 2$. A solução retornada pela heurística tem custo total penalizado 17, enquanto a solução ótima retornada pelo algoritmo tem custo total 15.

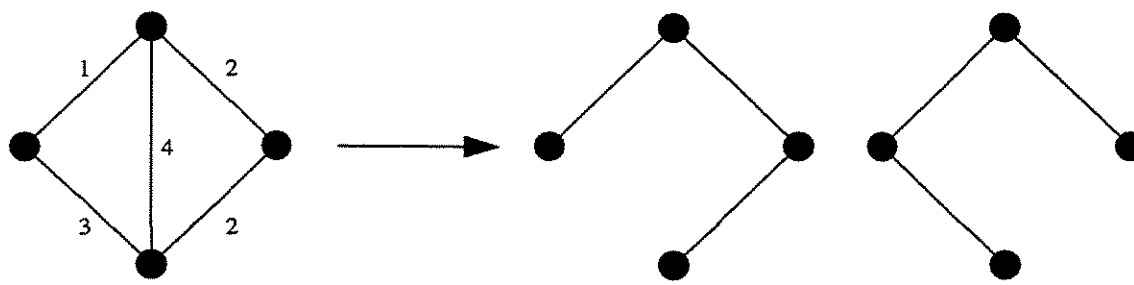


Figura 3.3: G e as árvores encontradas pela heurística A para $k = 2$.

Complexidade

A heurística A consiste em k iterações do laço **Para**. Por sua vez, cada iteração consiste em encontrar uma árvore espalhada mínima T_i e atualizar em G os pesos das $n - 1$

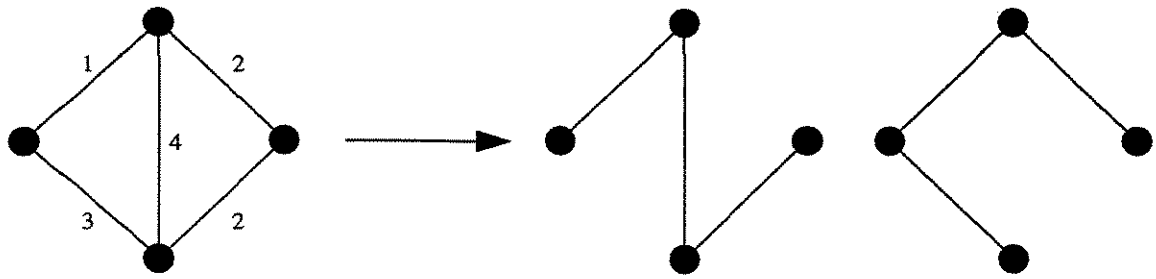


Figura 3.4: G e as árvores ótimas para $k = 2$.

arestas usadas em T_i. Para encontrar uma árvore espalhada existem vários algoritmos (veja [17], [21], [11] e [5]). Dentre eles podemos escolher o algoritmo de *Kruskal*, que custa $O(m \log m)$. Para atualizar G, atualizamos $n-1$ arestas e cada atualização pode ser feita em tempo constante, então gastamos $O(n-1) = O(n)$ com a atualização de G.

Portanto, para executarmos a heurística A fazemos k iterações, onde cada iteração gasta $O(m \log m + n) = O(m \log m)$, ou seja, uma implementação simples da heurística gasta $O(km \log m)$. Mas, assim como fizemos no capítulo anterior, podemos aproveitar a existência de uma pré-ordenação das arestas para melhorar o limite assintótico da heurística.

A primeira execução do algoritmo de *Kruskal* requer a ordenação das m arestas de G; porém, para as $k-1$ execuções seguintes basta ordenarmos as $n-1$ arestas que foram atualizadas e juntá-las convenientemente (operação de *merge*) com as outras arestas. Após a i-ésima execução do algoritmo de *Kruskal*, as arestas podem ser particionadas em dois conjuntos, o conjunto das $n-1$ arestas que pertencem à árvore encontrada e o conjunto das arestas restantes. Então ordenamos o primeiro conjunto e executamos uma operação de *merge* do primeiro conjunto (agora ordenado) com o segundo conjunto (que já estava ordenado), isto custa $O(n \log n + m\alpha(2n, n))$. Assim, o custo total da heurística A é $O(m \log m + k(n \log n + m\alpha(2n, n)))$.

3.2 Heurística B

A heurística B, cuja idéia foi inspirada no algoritmo exato, mantém uma ordenação das arestas e k partições de vértices para representar as florestas e consiste em encontrar, para cada aresta e tomada segundo a ordenação, uma árvore T_i (dentre $T_1, T_2, \ldots, T_k$), tal que $T_i \cup \{e\}$ continue sendo uma árvore.

A principal diferença da heurística para o algoritmo ótimo é que, em vez de executarmos o algoritmo de rotulação para cada aresta e que forme um ciclo com T_1, simplesmente verificamos se $T_i \cup \{e\}$ (para $1 \leq i \leq k$ e sempre nesta ordem) continua sendo uma árvore.

Se encontramos T_i, tal que $T_i \cup \{e\}$ não forma um ciclo, então acrescentamos e à T_i e atualizamos o valor de e na fila. Mas, se e forma um ciclo com todas as k florestas, simplesmente retiramos a aresta e da fila.

Heurística B

Ordene as m arestas de G
Enquanto não forem encontradas $k(n-1)$ arestas
Encontre e, tal que e seja a próxima aresta na ordenação
Se existe T_i, tal que $T_i \cup \{e\}$ seja uma floresta
$T_i \leftarrow T_i \cup \{e\}$
Atualize e na ordenaçao
Senão
Retire e da ordenaçao
Fim Se
Fim Enquanto
Fim

É importante notar que, sendo G conexo, a heurística sempre retornará k árvores espalhadas de G. Pois, para a heurística retornar um conjunto de arestas T_i que não é uma árvore espalhada de G (ou seja, T_i é uma floresta desconexa) seria necessário que uma aresta e, tal que $T_i \cup \{e\}$ é uma floresta, fosse descartada, o que é impossível por construção.

Complexidade

Uma implementação simples da heurística gastaria $O(m \log m)$ para ordenar as arestas, $O(\log m)$ para manipular tais arestas (encontrar e, atualizar o peso de e e retirar e da ordenação usando um heap binário), $O(k)$ para encontrar T_i e $O(n)$ para fazer a atualização de T_i. Observe que o processamento de uma aresta pode resultar em apenas duas atividades: na atualização de T_i ou na exclusão de e da ordenação. A primeira (atualização) custa $O(\log m)$ para encontrar e, $O(k)$ para encontrar T_i, $O(n)$ para atualizar T_i e $O(\log m)$ para atualizar o peso de e na ordenação, totalizando $O(\log m + k + n)$. A segunda (exclusão) custa $O(\log m)$ para encontrar e, $O(k)$ para verificar que não exite T_i e $O(\log m)$ para retirar e da ordenação, totalizando $O(\log m + k)$.

Observe também que $O(kn)$ processamentos de arestas podem resultar em atualizações de T_i e que $O(m)$ processamentos de arestas podem resultar em extração de e. Assim, gastamos $O(kn(\log m + k + n) + m(\log m + k))$ com o processamento das arestas, além de $O(m \log m)$ gastos para construir a ordenação. Ou seja, uma implementação simples da heurística B custa $O(m \log m + kn(\log m + k + n))$.

Em [25] mostramos uma implementação $O(m \log m + kn(\log m + \alpha(2n, n) \log n))$ para a heurística B. Nesta implementação, as principais melhorias são o tratamento de moitas (como no algoritmo exato) e a otimização da busca por T_i. A idéia e a análise desta heurística são de R. Werneck.

3.3 Contra-exemplo

Apesar de sempre fornecerem valores muito próximos dos valores ótimos, como veremos adiante, as heurísticas não oferecem garantia de otimalidade nem mesmo para grafos com topologias simples como a do ciclo mostrado na figura 3.5. Neste exemplo, para $k = 4$, as heurísticas propostas encontram uma solução de valor 57 ($x_a = 3$, $x_b = 3$ e $x_c = 2$), enquanto o algoritmo exato encontra uma solução com valor total 56 ($x_a = 4$, $x_b = 2$ e $x_c = 2$).

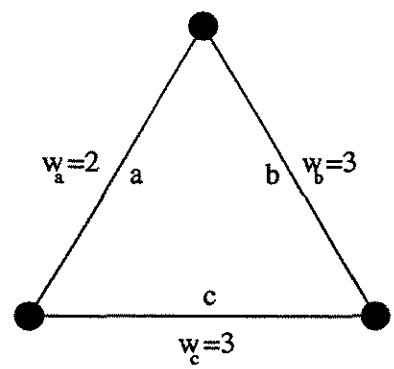


Figura 3.5: Contra-exemplo.

3.4 Solução exata × heurísticas

Nesta seção, vamos mostrar os resultados obtidos nos testes executados com o algoritmo exato, com a heurística A (usando *Kruskal*) e com a heurística B. O comportamento teórico das heurísticas e do algoritmo estão resumidos na tabela 3.1.

Para avaliar o comportamento prático vamos comparar os tempos de execução e a qualidade das soluções geradas, onde a qualidade de uma solução é razão entre o seu peso e o peso da solução ótima. Os resultados aqui apresentados foram publicados na referência [25], sendo que todo o trabalho de implementação foi realizado por R. Werneck.

Todos os tempos de execução que mostraremos são a média de pelo menos três execuções (caso médio) e os valores de qualidade apresentados são os maiores encontrados (pior caso entre as três execuções). As instâncias sobre as quais foram executados os

Algoritmo	Tempo de execução
exato	$O(m \log m + k^2 n^2)$
A-Kruskal	$O(m \log m + k(m\alpha(2n, n) + n \log n))$
B	$O(m \log m + kn(\log m + \alpha(2n, n) \log n))$

Tabela 3.1: Limites teóricos

testes podem ser divididas em três classes: grafos completos, grafos hipercúbicos esparsos ($m = 4n$) e grafos com topologia aleatória.

A função de penalização utilizada foi $w_p(e, i) = iw(e)$. Todos os programas foram implementados em C++ e compilados com o GNU C, usando a diretiva de otimização -03, e executados em uma máquina DEC Alpha 600au com 1GB de RAM.

Na implementação do algoritmo exato, ao invés de executar o algoritmo de rotulação quando não era possível inserir e_0 em F_1, tentamos inserir e_0 em F_i, (para $1 \leq i \leq k$), antes de executarmos o algoritmo de rotulação. Esta alteração não implica alteração no limite assintótico do algoritmo, mas melhora o seu desempenho prático. Em testes preliminares, a implementação da heurística A usando *Prim* (A-*Prim*) mostrou-se mais lenta do que A-*Kruskal*, por isso apresentamos apenas os resultados de A-*Kruskal*. Também foi implementado um algoritmo (chamaremos de aleatório) que simplesmente encontra k árvores espalhadas de forma aleatória. Apresentaremos também os resultados deste algoritmo para mostrar o quanto uma solução qualquer fica longe da ótima.

Grafos completos

A tabela 3.2 mostra o tempo médio de execução (em segundos) do algoritmo exato e das duas heurísticas em um grafo completo, com $n = 100$, para diversos valores de k. Neste grafo completo e ponderado, os pesos das arestas são distintos e uniformemente distribuídos.

k	Alg.	Heu. A	Heu. B
100	2,32	0,08	0,04
200	6,97	0,13	0,08
300	12,37	0,20	0,12
400	17,67	0,28	0,16
500	21,99	0,36	0,20

Tabela 3.2: Tempos de execução para um grafo completo com $n = 100$

A tabela 3.3 mostra a qualidade das soluções geradas pelas heurísticas e por um algoritmo que retorna uma solução aleatória. A qualidade mostrada para as heurísticas é a pior (maior valor) retornada para cada conjunto de três instâncias, já a qualidade mostrada para a solução aleatória é a melhor (menor valor) encontrada nas três instâncias.

k	**Heu. A**	**Heu. B**	**Aleatório**
100	1,000492	1,000496	4,92
200	1,000150	1,000149	4,51
300	1,000297	1,000302	4,29
400	1,000198	1,000197	4,13
500	1,000130	1,000130	4,06

Tabela 3.3: Qualidade das heurísticas para um grafo completo com $n = 100$

Ainda na classe dos grafos completos, testamos o comportamento das heurísticas A e B, com relação a k. Para isto usamos um grafo completo da TSPLIB, o grafo tem $n = 58$ e é chamado **brazil58**[1]. A figura 3.6 mostra tal comportamento apenas da heurística A, pois as heurísticas A e B apresentaram comportamentos semelhantes neste teste.

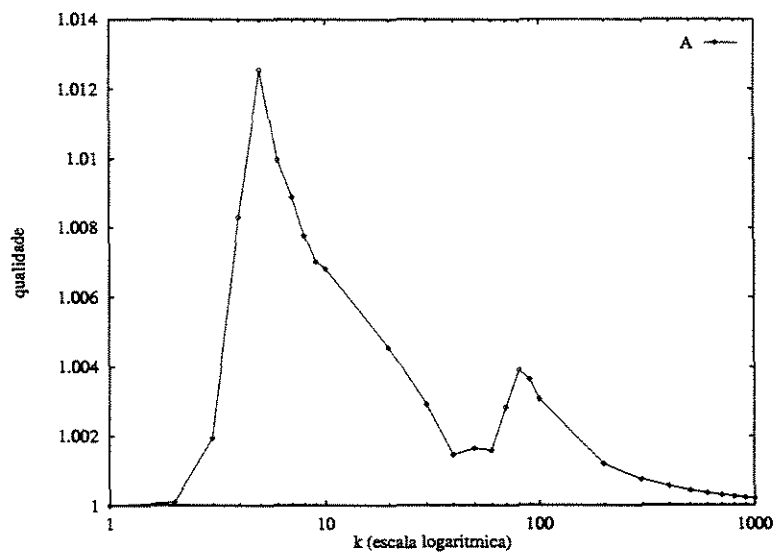


Figura 3.6: Qualidade da heurística A com relação a k.

[1]O grafo **brazil58** e os outros grafos da TSPLIB podem ser encontradas em *http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html*

Grafos hipercúbicos

A classe de grafos tratada nesta seção é a dos grafos hipercúbicos, estes grafos consistem em *grids* circulares de ordem l e dimensão d. Nestes *grids* temos l^d vértices e dl^d arestas. Para $d = 2$ e $l = 5$, o resultado é um grid circular bi-dimensional de 5×5. Nos nossos testes usamos $d = 4$ e diversos valores de l, resultando em grafos esparsos onde $m = 4n$. Os pesos das arestas são distintos e distribuídos dando preferência a valores pequenos.

As tabelas 3.4 e 3.5 mostram, respectivamente, os tempos de execução (em segundos) e a qualidade das soluções geradas para os grafos hipercúbicos.

n	Alg.	Heu. A	Heu. B
81	0,81	0,01	0,02
256	10,75	0,05	0,10
625	76,93	0,17	0,32
1296	364,21	0,41	0,85

Tabela 3.4: Tempos de execução para um grafo hipercúbico com $k = 100$

n	Heu. A	Heu. B	Aleatório
81	1,000461	1,000403	52,29
256	1,000254	1,000258	54,48
625	1,000310	1,000310	29,14
1296	1,000325	1,000325	15,69

Tabela 3.5: Qualidade das heurísticas para um grafo hipercúbico com $k = 100$

Grafos aleatórios

Na família dos grafos aleatórios, fizemos um último experimento que compara o comportamento das heurísticas A e B com relação ao número de arestas. A figura 3.7 mostra tal comportamento.

Neste teste, usamos $n = 500$, $k = 1000$ e variamos m de 1200 (esparso) até 124750 (completo).

3.4.1 Análise dos resultados

Com relação ao desempenho, como já podíamos esperar, as heurísticas são muito mais rápidas do que o algoritmo exato, especialmente em grafos esparsos. Entre as heurísticas,

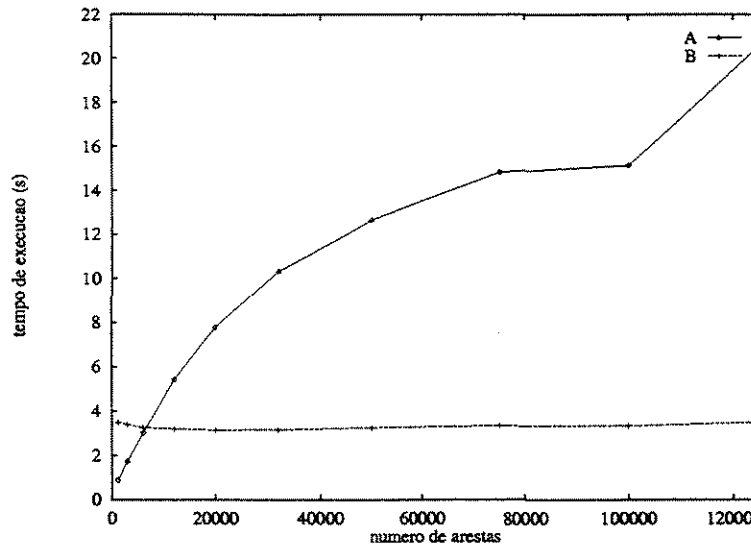


Figura 3.7: Comparação entre as heurísticas A e B com relação a m.

a heurística A tem uma ligeira vantagem sobre a heurística B em grafos esparsos. Porém, esta vantagem se inverte conforme os grafos se tornam mais densos, como se percebe claramente na figura 3.7.

Com relação à qualidade, as heurísticas alcançaram resultados muito próximos do ótimo, cerca de $0,05\%$ no pior caso. Também podemos notar que as soluções aleatórias são muito piores do que as retornadas pelas heurísticas, o que mostra que as heurísticas, apesar de simples, obtêm soluções que não seriam facilmente encontradas de forma aleatória. Assim, resumimos nossas conclusões sobre os testes afirmando que as heurísticas apresentadas são uma ótima opção quando a solução exata não é estritamente necessária.

Capítulo 4

Outros problemas combinatórios

Neste capítulo apresentaremos a aplicação da técnica utilizada para resolver o problema das árvores mínimas em outros três conhecidos problemas combinatórios: o problema dos caminhos mínimos, dos emparelhamentos e dos roteamentos.

As soluções propostas para estes problemas consistem em reduções ao problema de encontrar um fluxo de custo mínimo, com pequenas variações para cada caso. Para simplicidade do texto, vamos nos referir ao problema de fluxo de custo mínimo simplesmente por NF.

4.1 Caminhos mínimos

Nesta seção vamos apresentar e resolver o problema de encontrar os k caminhos mais curtos sujeitos a congestionamento. A solução consiste em uma redução polinomial aos problemas de fluxos em redes. No decorrer do texto, vamos nos referir ao problema simplesmente como kSPc (*k minimum congestion shortest paths*).

4.1.1 Formulação do problema kSPc

Resolver o problema kSPc consiste em, dado um grafo ponderado $G = (V, E)$, encontrar k caminhos não necessariamente disjuntos $C_1, C_2, \ldots, C_k$ que começam em um vértice s e terminam em um vértice t e que minimizam o custo total da solução. Para caracterizar o congestionamento usamos a função de penalização $w_p(e, i_e) = i_e w(e)$. Assim, o custo total da solução é dado por $\sum_1^k \sum_{e \in C_i} i_e w(e) = \sum_{e \in E} i_e^2 w(e)$.

Para formular o problema kSPc é necessário levar em conta a orientação das arestas, assim como em alguns problemas de fluxo em redes (vide [1]). Para isto, definiremos que para uma aresta $e = (i, j)$, $i_e = x_e = x_{ij} + x_{ji}$, onde x_{ij} é o fluxo enviado do vértice i

para o vértice j (e x_{ji} é o fluxo de j para i). Usando esta notação, o problema pode ser formulado como a seguir

Minimizar

$$\sum_{e \in E} x_e^2 w_e \tag{4.1}$$

Sujeito à

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = 0 \qquad \textit{para todo } i \in \{V - s - t\} \tag{4.2}$$

$$\sum_{i:(s,i)\in E} x_{si} = k \tag{4.3}$$

$$\sum_{i:(i,t)\in E} x_{it} = k \tag{4.4}$$

$$x_{it} \geq k \tag{4.5}$$

As expressões 4.3 e 4.4 garantem, respectivamente, que saem k caminhos do vértice s e que chegam k caminhos ao vértice t e a restrição 4.2 garante que todo caminho que chega a um vértice também sai, exceto para os vértices s e t.

4.1.2 Redução de kSPc ao problema de fluxo de custo mínimo

O problema combinatório, quadrático e inteiro, formulado na seção anterior, é muito parecido com o bem estudado problema NF [1], onde dado um grafo ponderado, um vértice fonte s e um vértice sorvedouro (ou ralo) t, queremos enviar k unidades de fluxo de s até t gastando o mínimo possível. Mas devido ao comportamento quadrático da função 4.1, não podemos usar diretamente o problema de fluxo de custo mínimo para resolver kSPc. Porém, usando a mesma estratégia utilizada no capítulo 2, podemos reduzir o problema kSPc a um problema de fluxo em redes onde não aparecem os fatores quadráticos.

Transformando kSPc em um problema de fluxo em redes

Sendo o grafo ponderado $G = (V, E)$ uma das entradas para o problema kSPc, construímos um multigrafo G', assim como fizemos para o problema kMSTc, fazendo cada aresta e de G corresponder a k arestas paralelas $e_1, e_2 \ldots, e_k$ em G'. Se usarmos a função de penalização $w_p(e, i_e) = i_e w(e)$, então o peso $w(e_j)$ de uma aresta e_j deve ser $jw_p(e, j) -$

$(j-1)w_p(e,j-1)$, para $1 \leq j \leq k$. Além disso, no multigrafo G', que no restante desta seção chamaremos de rede G', fazemos com que cada aresta tenha capacidade unitária de transporte de fluxo, ou seja, apenas uma unidade de fluxo poderá ser enviada por cada aresta. Na figura 4.1 podemos ver um grafo G e a respectiva rede (multigrafo) G' para $k = 3$.

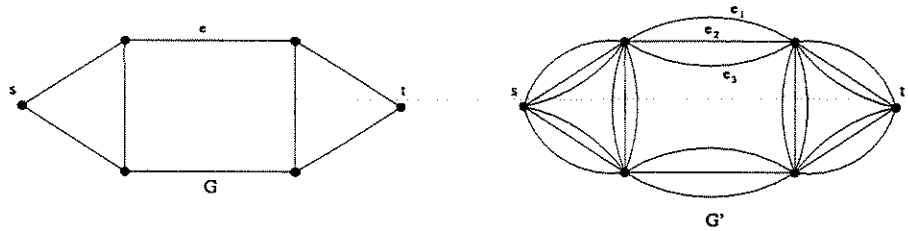


Figura 4.1: Construção da rede G' para $k = 3$.

Teorema 4.1 *O problema kSPc é eqivalente ao problema* NF.

Prova. A partir de uma solução para NF podemos facilmente construir uma solução para kSPc. Para isto, basta fazermos cada fluxo unitário ser um caminho entre s e t. Além disso, baseados no fato de que toda resposta ótima para NF é composta apenas por arestas contíguas (caso contrário, poderíamos facilmente mostrar uma solução melhor do que a ótima), podemos afirmar que uma solução para NF tem o mesmo peso da respectiva solução para kSPc. É também verdade que de uma solução para kSPc podemos construir uma solução para NF de mesmo peso. Então, se temos uma solução ótima A^* para NF, temos também uma solução ótima B^* para kSPc, pois se B^* não é ótima existe outra solução B' para NF, tal que o peso de B' é menor do que o peso de B^*. Conseqüentemente existe uma solução A', construída a partir de B', tal que o peso de A' é menor do que o peso de A^*, o que contradiz a otimalidade de A^*. □

4.1.3 Complexidade

Para obtermos uma solução para kSPc são necessárias três fases. Primeiro, precisamos construir a rede G'. Isto pode ser feito em $O(km+n)$. Segundo, precisamos executar algum método capaz de resolver NF em G' e obter uma resposta. Existem diversos algoritmos para resolver NF. A referência [1] aponta o método *Enhanced capacity scaling algorithm* como sendo o mais eficiente dos algoritmos polinomiais conhecidos até então. Usando este método NF pode ser resolvido em $O((m\log n)(m+n\log n))$. Como a rede G' possui km arestas, gastaremos $O((km\log n)(km+n\log n))$ para resolver NF em G'.

Na terceira e última fase, precisamos converter a resposta para NF em uma resposta para kSPc. Esta fase certamente é limitada superiomente pelo número de arestas da rede, que é $O(km)$.

Somando-se as contribuições das três fases, temos $O(km+n)+O((km\log n)(km+n\log n))+O(km)$. Ou seja, resolver kSPc custa $O((km\log n)(km+n\log n))$. Vale notar que tempo gasto com a resolução de NF dominou a expressão.

4.2 Emparelhamentos perfeitos

Nesta seção vamos resolver o problema de encontrar k emparelhamentos perfeitos e não necessariamente disjuntos de um grafo ponderado e bipartido $G=(A\cup B,E)$, onde $|A|=|B|$, tal que o custo total penalizado da solução seja mínimo. Usaremos o mesmo tipo de penalização dos problemas anteriores e chamaremos este problema de problema kBMc (*k minimum congestion bipartite matching*).

4.2.1 Algoritmo

Dizemos que um grafo bipartido $G=(A\cup B,E)$ possui um *emparelhamento perfeito* se existe um subgrafo espalhado de G, onde todo vértice de A é adjacente a um único vértice de B e vice-versa. Podemos então dizer que um emparelhamento perfeito de G é um subgrafo espalhado de G tal que todos os seus vértices possuem grau 1. Este subgrafo é também chamado de *1-fator*. Dado um grafo G qualquer, um *j-fator* de G é um subgrafo espalhado de G, onde todo vértice do subgrafo possui grau j, ou seja, um *j-fator* de G é um subgrafo espalhado *j-regular* de G.

Uma solução intuitiva para kBMc consiste em encontrar um emparelhamento e atualizar G até encontrar k emparelhamentos, mas este algoritmo não garante a otimalidade da solução (como as heurísticas para kMSTc). Uma abordagem que resulta em uma solução ótima é reduzir o problema a um problema de fluxo em redes.

A redução consiste em costruir uma rede de fluxo F e encontrar um conjunto M de arestas (onde $|M|=k|A|$ e o peso de M é mínimo) e particionar M em k emparelhamentos perfeitos de G. Seja um multigrafo bipartido e ponderado G', construído a partir de G como fizemos nos capítulos anteriores (veja a figura 4.2).

Dado o multigrafo bipartido e ponderado G', podemos construir uma rede de fluxo F orientando todas as arestas de G' de A para B e adicionando mais dois vértices, os vértices s e t, tal que existe uma aresta $\{s,a\}$, para todo vértice $a\in A$, e tal que existe uma aresta $\{b,t\}$, para todo vértice $b\in B$. Na rede F, mantemos o custo das arestas que pertenciam a G' e atribuímos custo zero às novas arestas. Com relação às capacidades

de cada aresta, atribuímos capacidade unitária a todas as arestas de $F \cap G'$ e atribuímos capacidade k a todas as arestas de $F \backslash G'$. A figura 4.3 mostra o grafo G' e a rede F.

Se o grafo bipartido G possui um emparelhamento perfeito, então é verdade que existe uma solução para kBMc, mas é claro que se G não possui um emparelhamento perfeito, então não podemos encontrar uma solução para kBMc, ou seja, não podemos encontrar k emparelhamentos perfeitos. Assim, para simplificar, sempre que nos referirmos a um grafo bipartido G, consideramos que G possui um emparelhamento perfeito. Esta hipótese não implica qualquer alteração no tempo de execução assintótico do algoritmo a ser apresentado, pois o custo assintótico de verificar se um grafo bipartido G possui um *emparelhamento perfeito* é dominado pelo resto do algoritmo. Esta verificação é simples e consiste em encontrar um emparelhamento de cardinalidade máxima em G ($O(m\sqrt{n})$ [1]) e testar se a cardinalidade do emparelhamento encontrado é igual a $|A|$ ($O(1)$).

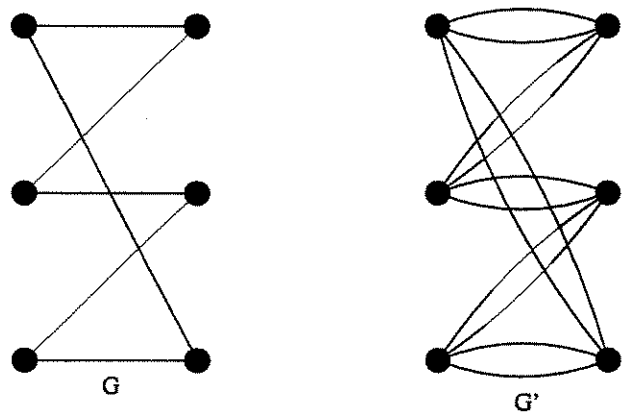


Figura 4.2: Construção da rede G' a partir de G, para $k = 2$.

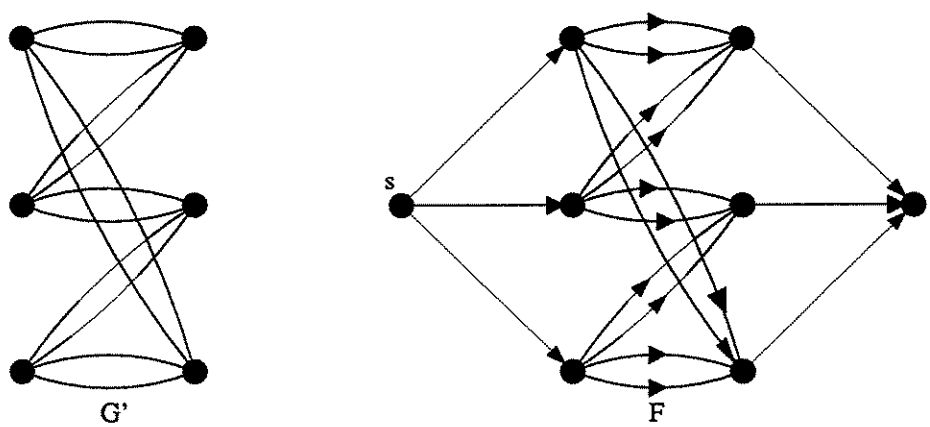


Figura 4.3: Construção da rede F a partir de G'.

Dado que G possui um *1-fator*, então é trivial que G' possui um *k-fator* como subgrafo.

Também é simples notar que se G' possui um *k-fator*, então a rede F tem um $k|A|$-*fluxo*, ou seja, é possível enviar $k|A|$ unidades de fluxo de s para t. Chamemos de F^* uma solução para o problema de encontrar um $k|A|$-*fluxo* de custo mínimo em F.

Usando apenas as arestas utilizadas em F^* (lembrando que as soluções retornadas para o problema de fluxo de custo mínimo são sempre inteiras [1]), podemos construir um *k-fator* de G', digamos G'', como ilustra a figura 4.4.

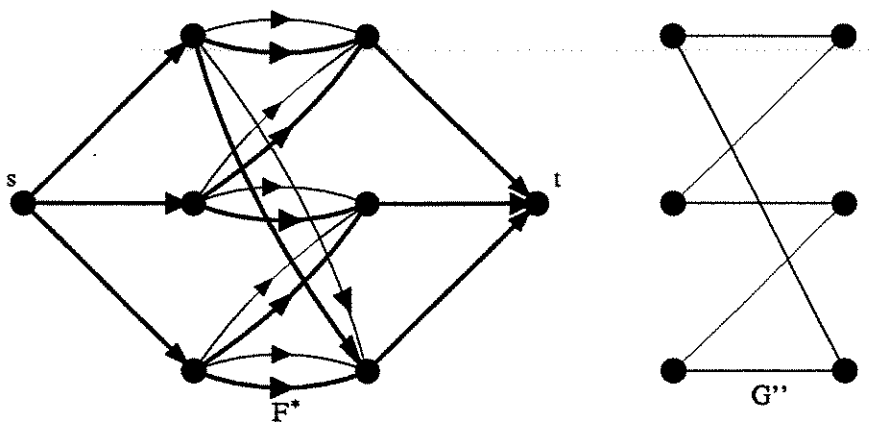


Figura 4.4: Construção do grafo *k-regular* G'' a partir de um fluxo ótimo F^*.

É importante notar que o custo da solução ótima para kBMc é maior ou igual ao custo de F^*, pois se existissem k emparelhamentos disjuntos em G' que custassem menos do que F^* poderíamos facilmente construir um $k|A|$-*fluxo* com custo menor do que o custo de F^*, o que é uma contradição. Em outras palavras, um $k|A|$-*fluxo* de custo mínimo na rede F é um limite inferior para o problema de encontrar k emparelhamentos perfeitos disjuntos nas arestas em G'. Como conseqüência direta disto, temos que toda solução M para kBMc que use as mesmas arestas que G'' (que são as mesmas arestas usadas em F^*) é uma solução ótima.

Um algoritmo para resolver kBMc seria

Algoritmo(G,k)

Se G possui um emparelhamento perfeito
Construa a rede F.
Encontre um fluxo de custo mínimo F^* em F.
Construa G'' a partir de F^*.
Para $i = 1$ até k
Encontre um emparelhamento de G'', digamos M_i.
$G'' \leftarrow G'' \setminus M_i$
Fim Para

Fim Se

Fim

Verificar se um grafo G possui um emparelhamento perfeito pode ser feito em tempo polinomial e se reduz à um problema do fluxo máximo (emparelhamento bipartido não ponderado), onde todas as capacidades são unitárias ($O(m\sqrt{n})$). Após verificarmos que G possui um emparelhamento perfeito, precisamos construir F e encontrar um $k|A|$-fluxo de custo mínimo em F, o que custa $O((km\log n)(km + n\log n))$ se usarmos o método *Enhanced capacity scaling algorithm*. Encontrado um $k|A|$-fluxo de custo mínimo, construimos G'' ($O(kn)$) e fazemos k iterações, onde cada iteração encontra um emparelhamento de G'' ($O(kn\sqrt{n})$) e atualiza G'' ($O(n)$).

Resumindo, resolver kBMc custa $O(m\sqrt{n})$ para verificar se o grafo G possui um emparelhamento perfeito, $O((km\log n)(km + n\log n))$ para encontrar um subgrafo *k-regular* G'' e $O(k^2n\sqrt{n})$ para particionar G'' em k emparelhamentos perfeitos de G. Então, resolver kBMc custa $O(m\sqrt{n}) + O((km\log n)(km + n\log n)) + O(k^2n\sqrt{n})$. Como $O((km\log n)(km + n\log n))$ domina a expressão, então podemos afirmar que kBMc é $O((km\log n)(km + n\log n))$.

4.3 Roteamentos

Existem diversos problemas reais que consistem em problemas de roteamento. Especialmente em computação existem vários como, por exemplo, o problema de roteamento de mensagens. Um problema de roteamento, a bem grosso modo, consiste em um conjunto de problemas de caminho mínimo relacionados entre si. Podemos inclusive dizer que encontrar um caminho mínimo em um grafo significa encontrar um roteamento ótimo onde temos apenas uma origem e um destino.

Encontrar um roteamento consiste em, dado um grafo ponderado $G = (V, E)$ e um conjunto U de pares de vértices, encontrar um caminho em G que une cada par de vértices $u \in U$. Encontrar um roteamento ótimo consiste em encontrar um roteamento que, além de unir todos os pares de vértices presentes em U, minimize o custo total da solução (dado pelo somátorio dos pesos das arestas utilizadas).

Um roteamento seria um conjunto de problemas de caminho mínimo não fosse o comportamento não linear do peso das arestas em função da sua utilização. Esta é uma característica dos problemas práticos de roteamento e é neste ponto que encaixaremos as técnicas até aqui aplicadas para os problemas de congestionamento.

Nesta seção mostraremos como são formulados e resolvidos os problemas de roteamento e vamos propor uma nova estratégia de resolução destes problemas.

4.3.1 Formulação do problema de roteamento de mensagens

Bertsekas e *Gallager* [4] apresentaram um estudo sobre um problema prático de roteamento, o problema de roteamento de mensagens. Neste estudo eles apresentam uma formulação para o problema e uma estratégia de resolução. Para manter a uniformidade das notações, adaptamos a nomeclatura original para a notação usual em teoria da computação com o cuidado de não distorcer o conteúdo essencial do trabalho. Vale apenas destacar que, de forma simplista, chamamos de $w(e)$ o peso de uma aresta e, quando esta grandeza representa na verdade o tempo médio que uma mensagem leva para ser transmitida por um canal de comunicação e.

Em [4], *Bertsekas* e *Gallager* definem que, dado um grafo $G = (V, E)$, um roteamento ótimo é aquele que minimiza a expressão

$$\sum_{e \in E} w_p(e) \tag{4.6}$$

Onde $w_p(e)$ é uma função monotonicamente crescente que representa o peso penalizado de uma aresta e. Uma fórmula freqüentemente usada para $w_p(e)$ é

$$w_p(e) = \frac{i_e}{C_e - i_e} + i_e w(e) \tag{4.7}$$

Onde C_e é a capacidade de transmissão da aresta e, i_e é o número de mensagens transmitidas por e e $w(e)$ é o peso de e.

Para cada par de vértices $u = (i, j)$ existe uma demanda de comunicação, digamos r_u. Ou seja, se $r_u = 3$, precisamos enviar 3 mensagens do vértice i para o vértice j, não necessariamente pelo mesmo caminho. Nosso objetivo é dividir cada r_u entre os diversos caminhos que saem da origem i e chegam ao destino j tal que a expressão 4.6 seja minimizada. Se denotarmos U como sendo o conjunto de todos os pares de vértices que possuem uma demanda de comunicação, P_u como o conjunto de todos os caminhos possíveis que unem os vértices de um par $u = (i, j)$ e x_p como sendo o número de mensagens enviadas por um caminho p, então podemos formular um problema de roteamento como

Minimizar

$$\sum_{e \in E} w_p(e) = \sum_{e \in E} \left(\frac{i_e}{C_e - i_e} + i_e w(e) \right) \tag{4.8}$$

Sujeito à

$$\sum_{p \in P_u} x_p = r_u \qquad \textit{para todo } u \in U \tag{4.9}$$

$$x_p \geq 0 \qquad \textit{para todo } p \in P_u \textit{ e para todo } u \in U \tag{4.10}$$

O comportamento da função objetivo é não linear quando a utilização de uma aresta tende a sua capacidade máxima, deste modo, temos um problema de otimização não linear inteira.

4.3.2 Métodos de resolução

Como vimos, encontrar um roteamento ótimo de mensagens implica resolver um problema de otimização não linear inteira. Os métodos usados para encontrar tal roteamento ótimo são na verdade métodos aproximativos (veja [13], [3], [12] e [2], entre outros), pois relaxam as condições de integralidade do problema para resolvê-los usando métodos de otimização não linear [24].

Relaxando as condições de integralidade do problema de otimização não linear é possível obter uma condição necessária para a otimalidade de um roteamento. Para isto, precisamos definir um vetor X onde cada posição x_p representa o número de vezes que um caminho p é utilizado. Ou seja, $x_p = 0$ se o caminho p não é utilizado, $x_p = 1$ se o caminho p é utilizado uma vez, e assim por diante.

Definido o vetor X, podemos reescrever $w_p(e)$, tal que $w_p(e)$ seja definida em função de X $(w_p(X))$. Isto é possível dado que $i_e = \sum x_p$, onde p é todo caminho que contém a aresta e.

Podemos então relaxar a integralidade das variáveis x_p, fazendo isto podemos diferenciar parcialmente $w_p(X)$ com relação ao fluxo x_p enviado por um caminho p. Assim, $\frac{\partial w_p(X)}{\partial x_p}$ denota a taxa de variação da função $w_p(X)$ em função de x_p. De posse deste artifício, podemos afirmar que para um roteamento X^* ser ótimo a seguinte propriedade precisa ser verdadeira

$$\frac{\partial w_p(X^*)}{\partial x_{p'}} \geq \frac{\partial w_p(X^*)}{\partial x_p} \qquad \textit{para todo } p' \in P_u \tag{4.11}$$

Onde p é todo caminho $p \in P_u$ tal que $x_p > 0$. A condição deve ser verdadeira para uma solução ótima X^*, senão posso escolher convenientemente um número δ, tal que $x_{p'} \leftarrow x_{p'} + \delta$ e $x_p \leftarrow x_p - \delta$ resulte em uma solução melhor do que X^*. A condição pode ainda, além de ser necessária, ser suficiente se a função w_p é convexa [16].

Em linhas gerais, os métodos usados para encontrar um roteamento ótimo são algoritmos iterativos que partem de uma solução X_i para uma solução X_{i+1} encontrando caminhos p e p', tal que a condição 4.11 não seja verdadeira, então atualizam X_{i+1} fazendo $x_{p'} \leftarrow x_{p'} + \delta$ e $x_p \leftarrow x_p - \delta$, para um valor δ convenientemente escolhido. A execução

dos algoritmos continua até que as novas soluções encontradas não melhorem a solução atual o suficiente para justificar novas buscas.

As desvantagens destes métodos de otimização não linear são basicamente três. Primeiro, não são uma solução exata, pois são inseridos erros ao se relaxar as condições de integralidade. Segundo, mesmo não sendo uma solução exata, não possuem um limite superior polinomial (os problemas de otimização não linear em geral são *NP-difíceis* [16]) para o tempo de parada do algoritmo e, terceiro, o tempo de execução dos algoritmos pode variar segundo fatores pouco mensuráveis, como a topologia da instância a ser resolvida. Além destes, ainda existem os problemas intrínsecos aos métodos tradicionais de otimização não linear, como a existência de mínimos locais.

4.3.3 Formulação alternativa

Uma vez que todas as variáveis da formulação original do problema de roteamento de mensagens são inteiras, podemos aplicar a mesma estratégia que utilizamos com os problemas de congestionamento para transformar o problema de otimização não linear inteira em um problema de otimização linear inteira.

Lema 4.2 *As expressões* $(\frac{i_e}{C_e - i_e} + i_e w(e))$ *e* $\sum_{j=0}^{i_e-1}(\frac{C_e}{C_e^2+j^2-2C_ej-C_e+j} + w(e))$ *são equivalentes.*

Prova. Vamos chamar de $F(i_e)$ a função dada por $\frac{i_e}{C_e-i_e}+i_e w(e)$ e vamos definir a variação Δ de F em função de i_e, tal que $\Delta(i_e) = F(i_e) - F(i_e - 1)$. Deste modo

$$\Delta(i_e) = \frac{i_e}{C_e - i_e} + i_e w(e) - \frac{(i_e - 1)}{C_e - (i_e - 1)} - (i_e - 1)w(e)$$

$$\Delta(i_e) = \frac{i_e}{C_e - i_e} - \frac{i_e - 1}{C_e - i_e + 1} + i_e w(e) - i_e w(e) + w(e)$$

$$\Delta(i_e) = \frac{C_e i_e - i_e^2 + i_e - C_e i_e + C_e + i_e^2 - i_e}{(C_e - i_e)(C_e - i_e + 1)} + w(e)$$

$$\Delta(i_e) = \frac{C_e}{C_e^2 + i_e^2 - 2C_e i_e + C_e - i_e} + w(e)$$

Como $F(0) = 0$ e $F(i_e) = F(i_e - 1) + \Delta(i_e)$ podemos dizer que $F(i_e) = \Delta(1) + \Delta(2) + \ldots + \Delta(i_e)$. Então

$$F(i_e) = \sum_{j=1}^{i_e} \left(\frac{C_e}{C_e^2 + j^2 - 2C_e j + C_e - j} + w(e) \right)$$

□

Para todo grafo ponderado G dado como entrada para o problema de otimização não linear, podemos criar um multigrafo ponderado $G' = (V, E')$ (do mesmo modo que fizemos para as árvores), onde G é topologicamente idêntico a G', exceto pelo fato de que para cada aresta e de G existem k arestas $e_1, e_2, \ldots, e_k$ em G', como podemos ver na figura 2.1 da página 6. O peso destas arestas é dado por $\frac{C_e}{C_e^2+j^2-2C_ej+C_e-j} + w(e)$, para j de 1 até k. Ou seja, $w_p(e_1) = \frac{1}{C_e-1} + w(e), w_p(e_2) = \frac{C_e}{C_e^2-3C_e+2} + w(e), \ldots, w_p(e_k) = \frac{C_e}{C_e^2+k^2-2kC_e+C_e-k} + w(e)$.

Se usarmos as variáveis x_{e_i} para representar a utilização de uma arestas e_i do grafo G', então podemos formular um roteamento como

Minimizar

$$\sum_{e \in E} w_p(e) = \sum_{e_i \in E'} \left(\frac{C_e}{C_e^2 + i^2 - 2C_e i + C_e - i} + w(e) \right) x_{e_i} \tag{4.12}$$

Sujeito à

$$\sum_{p \in Pu} x_p = r_u \qquad \textit{para todo } u \in U \tag{4.13}$$

$$x_p \geq 0 \qquad \textit{para todo } p \in P_u \ \textit{ e para todo } u \in U \tag{4.14}$$

$$0 \leq x_{e_i} \leq 1 \qquad \textit{para todo } e_i \in E' \tag{4.15}$$

O comportamento da função objetivo desta formulação é linear, diferentemente da formulação original que apresentava comportamento não linear (seção 4.3.1).

É fácil mostrar que toda solução ótima para a formulação acima corresponde a uma solução ótima para a formulação original. Ou seja, as formulações são equivalentes. A demonstração de equivalência entre as modelagens é idêntica as demonstrações de equivalência entre problemas que fizemos anteriormente. Chamando a formulação original de formulação A e a formulação acima proposta de formulação B, podemos enunciar que

Teorema 4.3 *As formulações* A *e* B *são equivalentes.* □

O problema de otimização linear inteira em que resulta a formulação B é conhecido na literatura como MULTICOMMODITY FLOW (MF) e é um problema *NP-Difícil* até mesmo quando queremos encontrar apenas dois caminhos [14].

4.3.4 Qual é a melhor abordagem?

Temos então duas abordagens possíveis para resolver o mesmo problema. Podemos resolver o problema de otimização não linear inteira (formulação A), ou podemos resolver o problema de otimização linear inteira (formulação B). Porém, ambos são problemas *NP-difíceis*.

O problema de encontrar um roteamento ótimo de mensagens tem sido resolvido relaxando-se as condições de integralidade da formulação A. Mas será esta a melhor forma de resolver o problema? Comunicações mais recentes como [18] e [15] apresentaram avanços nas técnicas para se encontrar uma solução ótima para MF. Avanços estes que sugerem novos estudos para podermos afirmar, dados os últimos avanços, qual abordagem seria melhor.

Capítulo 5

Conclusões

Dentre as contribuições deste trabalho gostaria de destacar:

i) A proposição de uma nova classe de problemas, os problemas combinatórios de congestionamento;

ii) Algoritmos exatos para alguns problemas desta classe (kMSTc, kSPc e kBMc);

iii) A descrição detalhada de um algoritmo eficiente para kMSTd;

iv) A proposição e análise de comportamento de heurísticas eficientes para kMSTc.

Mas, talvez a maior contribuição deste trabalho seja a abertura de uma vasta gama de possibilidades para trabalhos futuros. Existirá um algoritmo para kMSTc melhor do que o apresentado no capítulo 2? Seriam algoritmos aproximativos as heurísticas apresentadas no capítulo 3? Como seria o comportamento das heurísticas para outras funções de penalização? Em que outros problemas combinatórios podemos aplicar as técnicas apresentadas?

Enfim, a nossa principal contribuição foram os problemas que ainda não resolvemos.

Referências Bibliográficas

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows.* Prentice Hall, 1993.

[2] D. P. Bertekas, E. M. Gafni, and R. G. Gallager. Second derivate algorithms for minimum delay routing in networks. *IEEE Trans. Comm.*, COM-32:911–919, 1984.

[3] D. P. Bertsekas. Algorithms for nonlinear multicommodity network flow problems. International Symposium in systems optimization and analysis, pages 210–224, New York, 1979. Springer-Verlag.

[4] Dimitri Bertsekas and Robert Gallager. *Data networks.* Prentice-Hall, second edition, 1992.

[5] Andrew Chi Chih Yao. An $O(|e| \log\log |v|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4(1):21–23, September 1975.

[6] Jens Clausen and Lone A. Hansen. Finding k edge-disjoint spanning trees of minimum total weight in a network: an application of matroid theory. *Mathematical Programming Study*, 13:88–101, 1980.

[7] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[8] Reinhard Diestel. *Graph Theory*, chapter 3, pages 57–59. Springer-Verlag, 1997.

[9] Jack Edmonds. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards-B*, 69B:67–72, 1965.

[10] Paulo Feofiloff and Cláudio Leonardo Lucchesi. *Algoritmos para Igualdades Minimax em Grafos*, chapter 8, pages 69–77. 1988. Livro texto da VI Escola de Computação.

[11] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[12] E. M. Gafni. Convergence of a routing algorithm. Master's thesis, University of Illinois, Dept. Electrical Engineering, Urbana, IL, 1979.

[13] R. G. Gallager. Conflict resolution in random access broadcast networks. *Proc. AFOSR Workshop Comm, Theory Appl.*, pages 74–76, 1978.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[15] Kaj Holmberg and Johan Hellstrand. Solving the uncapacitated network design problem by a lagrangean heuristic and branch-and-bound. *Operations research*, 46(2):247–259, March 1998.

[16] R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*, volume 3. Kluwer Academic Publishers, 1995.

[17] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[18] R. D. McBride. Advances in solving the multicommodity flow problem. *INTERFACES*, 28(2):32–41, March 1998.

[19] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. *Proc. 2nd ann. workshop on data structs. and algs. WADS-91*, 1991. Lecture Notes in Computer Science 519, 400-411.

[20] C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal London Math. Soc.*, 36:445–450, 1961.

[21] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Tech. J.*, 36:1389–1401, 1957.

[22] James Roskind and Robert E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708, November 1985.

[23] W. T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal London Math. Soc.*, 36:221–230, 1961.

[24] Garret N. Vanderplaats. *Numerical optimization techniques for engineering design.* McGraw-Hill, 1984.

[25] Renato Fonseca F. Werneck, João Carlos Setubal, and Arlindo Flávio da Conceição. Finding minimum congestion spanning trees. *3rd Workshop on Algorithm Engineering, London, 1999.*

[26] Francisco Barahona. Packing spanning trees. *Mathematics of Operations Research*, 20(1):104–115, February 1995.

Apêndice A

Matróides

Um matróide é um par ordenado $M = (S, I)$ que satisfaz às seguintes propriedades:

i) S é um conjunto finito e não vazio;

ii) I é um conjunto não vazio de subconjuntos de S, chamados *conjuntos independentes*, tal que se $B \in I$ e $A \subseteq B$, então $A \in I$;

iii) Se $A \in I$, $B \in I$ e $|A| < |B|$, então existe um elemento $x \in \{B - A\}$ tal que $\{A \cup x\} \in I$.

Se um problema pode ser representado por um matróide, então existe um algoritmo guloso conhecido e correto para o problema de encontrar um conjunto independente de peso máximo. Como consequência direta da propriedade **iii**, todo conjunto independente de peso máximo é também maximal no número de elementos se todos os pesos envolvidos são positivos.

Se em uma instância M do problema nem todos os pesos são positivos, podemos adicionar a cada um dos elementos de S uma constante positiva, convenientemente escolhida, construindo uma nova instância M' tal que todos os pesos fiquem positivos. É fácil notar que uma solução X para M' também é solução para a instância M (que possuia elementos de peso negativo), pois se X não é solução para M existe uma solução Y para M, tal que $Y > X$. E deste modo existe uma solução Y para M', tal que $Y > X$. O que é uma contradição dado que X é por hipótese máximo com relação a M'. Então podemos estender a aplicação dos matróides para encontrar um conjunto independente maximal de peso mínimo, dado que encontrar um conjunto de peso mínimo, onde todos os pesos são positivos, é o mesmo que encontrar um conjunto de peso máximo onde todos os pesos são negativos.

Algoritmo guloso para matróides

Nesta seção veremos o algoritmo guloso usado para encontrar uma solução para um matróide. A prova de que o algoritmo guloso sempre retorna um conjunto independente maximal e ótimo (de peso total mínimo) pode ser encontrada em [7]. Na mesma referência podem ser encontrados alguns exemplos de problemas que formam um matróide.

Algoritmo guloso(S,I)
    $\Gamma \leftarrow \emptyset$
    Ordenar os elementos de S em ordem não decrescente de pesos
    Para cada $e \in S$, tomados segundo a ordenação
        Se $\Gamma \cup \{e\} \in I$
            $\Gamma \leftarrow \Gamma \cup \{e\}$
        Fim Se
    Fim Para
    Retornar Γ
Fim

Diversos problemas que formam matróides podem ser resolvidos pelo mesmo algoritmo e a única diferença entre as resoluções é o teste de independência. Neste teste verificamos se $\Gamma \cup \{e\} \in I$, ou seja, verificamos se $\Gamma \cup \{e\}$ é um conjunto independente.