Algoritmos de aproximação para o projeto de redes seguras

Antonio João Ferreira Francisco

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO DE MESTRE
EM
CIÊNCIAS

Área de Concentração: Ciência da Computação
Orientador: *Prof. Dr. Carlos Eduardo Ferreira*

São Paulo, fevereiro de 2006

Resumo

O objetivo no projeto de redes seguras é garantir que, mesmo na ocorrência de falhas em elementos da rede, a mesma continue operando. Além disso, é importante que o custo para garantir tal conectividade seja o menor possível. Neste trabalho tratamos de alguns problemas de otimização combinatória que podem ser usados para modelar o projeto de redes seguras.

Os problemas tratados são computacionalmente difíceis e concentramos nossa atenção no estudo e implementação de algoritmos de aproximação para resolvê-los.

Abstract

The goal in the design of survivable networks is to guarantee that, even if some network elements fail, the network itself keeps operational. Moreover, it is important that the cost to guarantee such connectivity should be as low as possible. In this work we study some combinatorial optimization problems that can be used to the design of survivable networks.

The problems in this work are computationally hard and we focus our attention in studying and implementing approximation algorithms to solve them.

Sumário

Introdução	1
1 Definição do problema e notação	4
1.1 Programação linear e programação inteira	4
1.2 Funções próprias	6
1.3 Funções não cruzáveis	6
1.4 Alguma notações	6
2 Algoritmos de aproximação	7
2.1 Técnicas de aproximação	7
2.1.1 Arredondamento	8
2.1.2 O método dual	9
2.2 O método primal-dual	9
2.2.1 Primal-dual clássico	10
2.2.2 Aproximação primal-dual	11
3 Uma $2k$-aproximação para o SNDP com custo de tempo $O(n^4k^3)$	13
3.1 Algoritmo de aproximação para funções não cruzáveis	15
3.2 Conjuntos ativos	17
3.3 Implementação	18
3.3.1 Estruturas de dados	19
3.3.2 Função principal	19
3.3.3 Função <i>uncrossable</i>	19
3.3.4 Testes	29
3.3.5 Prova da garantia de $2k$ -aproximação e do lema 3.0.2	30
4 Uma $2k$-aproximação para o SNDP com custo de tempo $O(k^2n^2 + kn^2\sqrt{\log \log n})$	33
4.1 Conjuntos ativos	35
4.2 Seleção de arestas	36
4.2.1 Estrutura de dados para a seleção de arestas	38
4.2.2 Funcionamento do algoritmo	38
4.2.3 Custo de tempo do algoritmo	39
4.3 Processo de limpeza	39
4.3.1 Implementação do processo de remoção	40
4.4 Custo total do algoritmo	42

4.5	Exemplo de funcionamento do algoritmo	42
5	Uma variante do problema	46
5.1	Descrição do algoritmo	47
5.1.1	O algoritmo para uma fase	48
5.2	Prova do lema 5.1.1	52
5.3	Exemplo de funcionamento do algoritmo	52
6	Conclusão	56
A	Código da implementação	58
A.1	Função principal	58
A.2	Função uncrossable	59
A.3	Função minimal_mincut	61
A.4	Função minimal_sets	61
A.5	Função mark_active	62
A.6	Função find_edge	63
A.7	Função insert_edge	64
A.8	Função inicializa_residual	64
A.9	Função delete	65
A.10	Função bfs	65
A.11	Função bfs_r	66
A.12	Função augment_path	67
A.13	Função flow	68
A.14	Função f	68
A.15	Função candidato	69
A.16	Função bfs_caminho	69
A.17	Função caminhos	70
A.18	Função viavel	70

Lista de Figuras

1	Exemplo de grafo	2
2	Solução do exemplo	3
3.1	Grafo inicial	20
3.2	Escolha da primeira aresta	21
3.3	Escolha da segunda aresta	21
3.4	Escolha da terceira aresta	22
3.5	Escolha da quarta aresta	23
3.6	Escolha da quinta aresta	24
3.7	Escolha da sexta aresta, início da segunda fase	25
3.8	Escolha da sétima aresta	26
4.1	Exemplo de a -conjuntos. As linhas pontilhadas representam os a -conjuntos, enquanto as linhas cheias representam os conjuntos ativos anteriores	37
4.2	Um exemplo do processo de inserção de arestas	43
4.3	Um exemplo do processo de limpeza	44
4.4	O resultado final	45
5.1	1ª fase: inserção de arestas	53
5.2	1ª fase: remoção de arestas	53
5.3	2ª fase: inserção de arestas	54
5.4	2ª fase: remoção de arestas	54
5.5	3ª fase: inserção de arestas	55
5.6	3ª fase: remoção de arestas	55

Introdução

Um problema freqüentemente encontrado em aplicações práticas é o de projetar redes seguras. Por exemplo, considere uma rede de interconexões (fibra óptica, telefonia, etc.) ligando várias cidades. Os requisitos de interconexão entre as cidades podem variar bastante de acordo com sua importância para a região: entre algumas cidades podemos querer que a rede tenha várias ligações disjuntas, enquanto para outros pares basta que haja uma conexão. Certamente estas redes devem ser projetadas de forma a minimizar o custo.

Uma rede pode ser protegida contra a falha de algumas ligações. Por exemplo, em uma rede em que sejam garantidos pelo menos dois caminhos disjuntos entre cada nó, se qualquer ligação falhar, ainda existe comunicação entre todos os nós. No problema mais geral, pode ser necessário que, entre um par de nós muito importantes, mesmo com a falha de muitos links ainda seja possível a conexão, enquanto que entre um par de nós sem muita importância não precisa nem ao menos haver uma conexão. O projeto de cada ligação pode acarretar em um custo maior ou menor. Mostrar maneiras de encontrar uma rede com essas características de conectividade e com o menor custo possível (ou um custo bem próximo do menor possível) é o objetivo deste trabalho.

Uma forma de modelar o problema é através do *Problema de Steiner Generalizado*, nosso objeto de estudo neste trabalho. O Problema de Steiner Generalizado é um problema de *Otimização Combinatória*. Problemas de otimização combinatória consistem em encontrar uma solução que maximiza ou minimiza um problema definido sobre um espaço discreto.

Muitos desses problemas têm um número grande de soluções possíveis, e muitas vezes não são conhecidos algoritmos que encontrem uma solução ótima eficientemente. O problema de Steiner Generalizado é um exemplo desse tipo de problema. Mesmo sendo difícil encontrar uma solução exata com eficiência, não podemos simplesmente descartar o estudo desse problema, devido a suas inúmeras aplicações práticas. Uma das maneiras de contornar essa dificuldade é trabalhar com *soluções aproximadas*, isto é, soluções que não são ótimas, mas que tem uma garantia de quanto é sua *distância* em relação à solução ótima. Algoritmos que produzem tais soluções são chamados *algoritmos de aproximação*.

O *Problema de Steiner Generalizado* (ou **SNDP** - *Survivable Network Design Problem*) consiste em, dados um grafo $G = (V, E)$, custos não negativos c_e para toda aresta $e \in E$, requisitos de conectividade r_{ij} para cada par de vértices $i, j \in V$, encontrar o subgrafo de G de custo mínimo no qual cada par de vértices i, j está ligado por no mínimo r_{ij} caminhos disjuntos nas arestas.

A seguir, damos um pequeno exemplo do problema.

Considere o grafo $G = (V, E)$ da figura 1 em que os custos c_{ij} das arestas estão indicados na figura e os requisitos de conectividade r_{ij} entre os pares de vértices são dados na tabela 1.

Figura 1: Exemplo de grafo

A solução ótima para o problema pode ser vista na figura 2.

Várias abordagens baseadas em algoritmos de aproximação foram propostas, como em [7], por Gabow, Goemans e Williamson e em [1], por Agrawal, Klein e Ravi.

Nosso objetivo neste trabalho é estudar o Problema de Steiner Generalizado neste contexto de projeto de redes seguras. Pretendemos analisar os algoritmos de aproximação propostos e suas implementações.

Este trabalho está organizado da seguinte maneira. No capítulo 1 definimos o *Problema de Steiner Generalizado* e suas variantes, mostrando que o problema é $\mathcal{NP}$ -difícil.

No capítulo 2 é dada uma pequena introdução a algoritmos de aproximação, com especial destaque ao método primal-dual.

No capítulo 3 apresentamos o algoritmo de Williamson, Goemans, Mihail e Vazirani [16].

r_{ij}	a	b	c	d	e	f
b	1					
c	1	2				
d	1	2	2			
e	1	0	1	1		
f	0	1	0	0	0	
g	1	0	0	1	1	0

Tabela 1: Requisitos de conectividade

Figura 2: Solução do exemplo

Esse foi o primeiro algoritmo desenvolvido para o problema, baseado no método primal-dual de aproximação.

No capítulo 4 apresentamos um algoritmo de Gabow, Goemans e Williamson [7], que utiliza a mesma lógica do algoritmo anterior, mas com estruturas diferentes na implementação. Tem a mesma razão de aproximação do anterior, mas com tempo de execução bem menor.

No capítulo 5 mostramos um algoritmo de Jain, Mandoiu, Vazirani e Williamson [13], que trabalha com uma variante do problema, na qual alguns nós também podem falhar. Utiliza o método primal-dual como os outros.

No capítulo 6 temos algumas conclusões sobre o trabalho e expectativas para o futuro.

No final existe um apêndice com o código da implementação do algoritmo do capítulo 3.

Capítulo 1

Definição do problema e notação

O *Problema de Steiner Generalizado* (ou **SNDP** - *Survivable Network Design Problem*) consiste em, dados um grafo $G = (V, E)$, custos não negativos c_e para toda aresta $e \in E$, requisitos de conectividade r_{ij} para cada par de vértices $i, j \in V$, encontrar o subgrafo de G de custo mínimo no qual cada par de vértices i, j está ligado por no mínimo r_{ij} caminhos disjuntos nas arestas.

Este problema é $\mathcal{NP}$ -difícil, já que o problema da árvore de Steiner é um caso particular dele. Existem algoritmos de aproximação, com razão dependente do tamanho da entrada. O primeiro algoritmo de aproximação foi desenvolvido por Williamson e outros [16], e melhorado em [7].

Antes do estudo de algoritmos de aproximação para este problema, já existiam heurísticas que tentavam resolvê-lo. A primeira heurística para este problema foi proposta por Steiglitz, Weiner e Kleitman [15], em 1969. A idéia apresentada foi baseada em buscas locais. O algoritmo para esta heurística primeiramente encontra uma solução viável para o problema para, em seguida, procurar por transformações locais que diminuam o custo da rede. Quando uma dessas transformações locais produz uma solução viável, essa nova rede é adotada e a busca por outras transformações locais continua, até que mais nenhuma seja possível.

Han e Lee [11] também descrevem várias heurísticas para o problema.

Um outro modo de trabalhar com este problema é sua modelagem através de poliedros, como a feita por Grötschel, Monma e Stoer [9]. Esta abordagem não será estudada neste trabalho.

Existe também uma outra variante do problema, na qual a solução pode ter arestas paralelas, estudada por Agrawal, Klain e Ravi [1]. Para resolver esse problema, são feitas buscas em profundidade no grafo, e guardadas as árvores formadas por essas buscas. Quando duas ou mais árvores se encontram em um vértice, elas se fundem em uma árvore, e a busca continua com essa nova árvore no lugar das anteriores. Quando as buscas terminam, temos a solução. Também este problema não será abordado neste trabalho.

1.1 Programação linear e programação inteira

Há problemas que podem ser formulados como maximização ou minimização de objetivos, dados recursos limitados e algumas restrições. Se esses objetivos puderem ser expressos como funções lineares de algumas variáveis, e os recursos e restrições como igualdades ou desigualdades nessas variáveis, teremos um problema de programação linear. Assim, o seguinte problema, onde A é

uma matriz e c, b e x são vetores, é um problema de programação linear:

$$\begin{array}{ll} \min & cx \\ \text{sujeito a} & Ax \geq b \\ & x \geq 0 \end{array}$$

Todo programa linear tem um dual correspondente. O programa original também é chamado de primal. Para o exemplo acima, o dual é o seguinte:

$$\begin{array}{ll} \max & yb \\ \text{sujeito a} & yA \leq c \\ & y \geq 0 \end{array}$$

Existem vários algoritmos para resolver esse problema. O mais conhecido é o simplex, que apesar de ser rápido na maioria dos casos, tem tempo de execução exponencial no pior caso. Existem, contudo, algoritmos polinomiais para esse problema, como o elipsóide, que no caso médio é mais lento que o simplex, e algoritmos baseados no método de pontos interiores, que para instâncias grandes conseguem ser tão ou até mais rápidos que o simplex no caso médio.

Há uma relação fundamental entre as soluções viáveis de um programa linear e as soluções viáveis de seu dual.

Lema 1.1.1 *Para todo x solução viável do primal e todo y solução viável do dual, $cx \geq yb$.*

Prova. Das restrições do primal e do dual, $cx \geq (yA)x$, pois $yA \leq c$, e $y(Ax) \geq yb$, pois $Ax \geq b$. Assim, $cx \geq (yA)x = y(Ax) \geq yb$. □

Teorema 1.1.2 *Se x e y são soluções ótimas do primal e do dual, respectivamente, então $cx = yb$.*

A prova deste teorema envolve a construção do método simplex, e está descrita em [8].

Um conceito que será bastante utilizado nos algoritmos de aproximação mostrados no próximo capítulo é o de *folgas complementares*.

Dois vetores x e y têm folgas complementares se

$$x_j = 0 \text{ ou } (yA)_j = c_j$$

e

$$y_i = 0 \text{ ou } (Ax)_i = b_i$$

No capítulo seguinte, esse conceito será bastante utilizado e definido com maiores detalhes.

Se acrescentarmos ao problema de programação linear a restrição de que as variáveis devem ser inteiras, teremos um problema de programação inteira. Essa mudança faz com que o problema seja $\mathcal{NP}$ -difícil.

Todos os algoritmos estudados envolvem conceitos de programação linear e programação inteira.

1.2 Funções próprias

Para alguns algoritmos é utilizado o conceito de função própria, que definimos a seguir. Uma função $f : 2^V \rightarrow N$ é chamada própria se:

- for simétrica, isto é, $f(S) = f(V \setminus S)$ para todo $S \subseteq V$;
- for maximal, isto é, se A e B são disjuntos, então $f(A \cup B) \leq \max\{f(A), f(B)\}$.

Vamos assumir que $f(\emptyset) = 0$ para todas as funções próprias f .

1.3 Funções não cruzáveis

Outro conceito utilizado é o de função não cruzável.

Uma função $h : 2^V \rightarrow \{0, 1\}$, com $h(V) = 0$ e tal que se $h(A) = h(B) = 1$, então ou $h(A \setminus B) = h(B \setminus A) = 1$ ou $h(A \cup B) = h(A \cap B) = 1$, é dita não cruzável.

Os algoritmos estudados se baseiam fortemente no conceito de função não cruzável, como será visto nos próximos capítulos.

1.4 Alguma notações

Durante todo este trabalho, n denotará o número de vértices do grafo, m denotará o número de arestas e k será o maior dos requisitos, isto é, $k = \max_{i,j \in V} r_{ij}$.

Capítulo 2

Algoritmos de aproximação

Existem muitos problemas de otimização que pertencem à classe $\mathcal{NP}$ -difícil, isto é, problemas para os quais não se conhece um algoritmo polinomial - e se for encontrado um algoritmo polinomial algum dia, então $\mathcal{P} = \mathcal{NP}$. Esses problemas, muitas vezes, são importantes na vida cotidiana, e por isso não podem simplesmente ser ignorados. Por esse motivo, foram desenvolvidas algumas técnicas que, se não resolvem o problema com exatidão, conseguem bons resultados.

Uma dessas técnicas é se restringir a uma classe de instâncias do problema e encontrar um algoritmo polinomial para esses casos. Em alguns problemas, por exemplo, se a entrada é um grafo planar, existe um algoritmo polinomial que resolve o problema. Por exemplo, o problema de se encontrar o corte máximo de um grafo geral é $\mathcal{NP}$ -completo, mas se restrito a grafos planares, o problema tem um algoritmo polinomial conhecido [10].

Outra maneira é tentar encontrar uma solução que garantidamente se aproxime da solução ótima, ao invés de tentar encontrar a própria solução ótima. São utilizados **algoritmos de aproximação**. Esses algoritmos não garantem que sua resposta é a solução ótima, mas garantem que não se afasta muito. O quanto a solução dada pelo algoritmo pode se afastar do ótimo é chamado **razão de aproximação**, que é definida, para um problema de minimização, da seguinte forma: seja $\text{opt}(I)$ o valor ótimo da instância I e $\text{aprx}(I)$ o valor devolvido por um algoritmo de aproximação para essa mesma instância. Se, para toda instância I , $\text{aprx}(I) \leq \alpha \text{opt}(I)$, então dizemos que esse algoritmo é uma α -aproximação para o problema.

2.1 Técnicas de aproximação

Existem várias técnicas de desenvolvimento de algoritmos de aproximação. Vamos descrever sucintamente as principais, que utilizam programação linear. Consistem em modelar o problema com programação inteira, e trabalhar com a relaxação linear deste.

- **arredondamento**: nesta técnica, a relaxação linear é resolvida, e os resultados são arredondados para um inteiro próximo que seja uma solução viável;
- **método dual**: neste caso, é resolvido o dual da relaxação linear, e a partir dessa solução, encontrar uma resposta para o problema original. Adiante daremos detalhes de como encontrar essa resposta;

- **método primal-dual:** neste método, é utilizado o primal do problema inteiro e o dual da relaxação linear. No início, há uma solução viável do dual, que a cada passo é modificada, até que ela corresponda a uma solução viável também para o primal.

Existem outras técnicas, como o uso de algoritmos probabilísticos e de programação semi-definida, que fogem do escopo deste trabalho. O leitor interessado pode estudar estas outras técnicas em [3] e [12].

Na sequência, vamos detalhar um pouco esses métodos, com atenção especial ao primal-dual.

2.1.1 Arredondamento

Vamos dar um exemplo de utilização desse método, extraído de [3], o problema da cobertura mínima por conjuntos. Uma coleção $\mathcal{S}$ de conjuntos cobre um conjunto E se todo elemento $e \in E$ estiver em pelo menos um conjunto de $\mathcal{S}$.

O problema é o seguinte: dados um conjunto finito E , uma coleção finita $\mathcal{S}$ de conjuntos finitos que cobre E e um custo c_S , racional e positivo, para cada $S \subseteq \mathcal{S}$, encontrar uma cobertura $\mathcal{T} \subseteq \mathcal{S}$ de E que minimize o custo de $\mathcal{T}$.

O programa linear que representa o problema é o seguinte:

$$\begin{aligned} \min \quad & cx \\ \text{sujeito a} \quad & \sum_{S:e \in S} x_S \geq 1 \quad \forall e \in E \\ & x_S \geq 0 \quad \forall S \in \mathcal{S} \end{aligned}$$

Neste programa, existe uma variável x_S para cada $S \subseteq \mathcal{S}$ e, para cada $e \in E$ uma restrição garantindo que e seja coberto. Esse programa tem uma solução ótima racional, e como qualquer cobertura de E é viável, temos que

$$\text{opt}(E, \mathcal{S}, c) \geq c\hat{x}$$

onde $\hat{x}$ é uma solução ótima do programa.

Se todos os componentes de uma solução $\hat{x}$ forem inteiros, então a coleção $\{S \in \mathcal{S} : \hat{x}_S > 0\}$ é uma solução ótima para o problema. Mas no caso geral teremos componentes fracionárias em $\hat{x}$. Assim, se arredondarmos esses valores de uma maneira correta, podemos obter uma solução viável para o problema. Vamos ver uma maneira de realizar esse arredondamento.

Para cada $e \in E$, vamos chamar de f_e o número de elementos de $\mathcal{S}$ em que e está presente. Seja β o maior dos f_e . Como $\mathcal{S}$ cobre E , temos que $\beta \geq 0$. Desse modo, uma maneira de obter uma solução para o problema seria escolher um conjunto $S \subseteq \mathcal{S}$ para fazer parte da solução se o valor da variável $\hat{x}_S$ for pelo menos $\frac{1}{\beta}$.

Esse algoritmo garante uma solução viável $\mathcal{T}$ (ou seja, uma cobertura), pois cada elemento e está em no máximo β conjuntos de $\mathcal{S}$, e como $\sum_{S:e \in S} \hat{x}_S \geq 1$, então $\hat{x}_S \geq \frac{1}{\beta}$ para pelo menos um $S \subseteq \mathcal{S}$ tal que $e \in S$. Além disso, esse algoritmo é uma β -aproximação para o problema, pois $c(\mathcal{T}) = \sum_{S \in \mathcal{T}} c_S \leq \beta \sum_{S \in \mathcal{T}} \hat{x}_S c_S \leq \beta c\hat{x} \leq \beta \text{opt}(E, \mathcal{S}, c)$, pois $\beta \hat{x}_S \geq 1$, para todo $S \in \mathcal{T}$.

2.1.2 O método dual

Aqui também vamos dar um exemplo da utilização do método, também extraído de [3], o problema da cobertura mínima por vértices.

O problema consiste no seguinte: dado um grafo $G = (V, E)$ com custos c_v nos vértices, encontrar um conjunto de vértices S que contenha pelo menos um extremo de cada aresta que minimize $c(S)$.

O método dual consiste em obter uma solução ótima do dual de uma relaxação linear do problema e, a partir desta solução, encontrar uma resposta para o problema. A relaxação linear do problema é a seguinte: encontrar um vetor x indexado por V que

$$\begin{aligned} \min \quad & cx \\ \text{sujeito a} \quad & x_u + x_v \geq 1 \quad \forall uv \in E \\ & x_v \geq 0 \quad \forall v \in V. \end{aligned}$$

O dual consiste em encontrar um vetor y indexado por E que

$$\begin{aligned} \max \quad & y(E) \\ \text{sujeito a} \quad & y(\delta(v)) \leq c_v \quad \forall v \in V \\ & y_e \geq 0 \quad \forall e \in E. \end{aligned}$$

Como qualquer cobertura por vértices é viável no primal, temos que

$$\text{opt}(G, c) \geq c\hat{x} = \hat{y}(E),$$

onde $\hat{x}$ é uma solução ótima do primal e $\hat{y}$ é uma solução ótima do dual.

Depois de obtida uma solução ótima $\hat{y}$ do dual, basta escolher todos os vértices de G cuja restrição no dual é satisfeita com igualdade. Essa estratégia é justificada a partir das folgas complementares: se $\hat{x}$ é uma solução ótima do primal então todo vértice v tal que $\hat{x}_v > 0$ é escolhido. Isto é, o algoritmo devolve uma cobertura C tal que $C = \{v \in V : \hat{y}(\delta(v)) = c_v\}$.

Essa cobertura C é uma 2-aproximação para o problema, pois

$$c(C) = \sum_{v \in C} c_v = \sum_{v \in C} \hat{y}(\delta(v)) \leq 2\hat{y}(E) \leq \text{opt}(G, c).$$

A primeira desigualdade vale pois, para cada aresta $e = (i, j)$, a variável $\hat{y}_e$ aparece no máximo duas vezes na somatória: uma vez se i está em C e outra se j está em C . A segunda desigualdade vale pois $\text{opt}(G, c) \geq c\hat{x} = \hat{y}(E)$.

2.2 O método primal-dual

Começamos explicando o método clássico, para resolver problemas de programação linear, e depois uma modificação deste, aplicável como algoritmo de aproximação.

2.2.1 Primal-dual clássico

Considere o problema de programação linear abaixo, onde A é uma matriz com os conjuntos $M = \{1, \dots, m\}$ e $N = \{1, \dots, n\}$ como índices de linhas e colunas, respectivamente, c e x são vetores indexados por N e b é um vetor indexado por M .

$$\begin{aligned} \min \quad & cx \\ \text{sujeito a} \quad & Ax \geq b \\ & x \geq 0 \end{aligned}$$

E o dual do problema, onde y é um vetor indexado por M .

$$\begin{aligned} \max \quad & yb \\ \text{sujeito a} \quad & yA \leq c \\ & y \geq 0 \end{aligned}$$

Os vetores x e y têm folgas complementares se $x_j = 0$ ou $(yA)_j = c_j$, para todo índice j em N , e $y_i = 0$ ou $(Ax)_i = b_i$, para todo índice i em M .

O método primal-dual é iterativo, começando com um vetor y viável no dual. É fácil notar que o vetor nulo é viável no dual, e por isso a primeira iteração começa com y nulo. Em cada iteração, procura-se um vetor x viável no primal com folgas complementares às de y . Se esse vetor é encontrado, o método pára, pois já encontrou as soluções ótimas. Se não encontrou tal vetor x , o método modifica o vetor y e começa uma nova iteração.

Cada iteração consiste no seguinte. Sejam

$$I(y) = \{i \in M : y_i = 0\} \text{ e } J(y) = \{j \in N : (yA)_j = c_j\}.$$

Considere o problema a seguir, onde se quer encontrar um vetor x indexado por N :

$$\begin{aligned} (Ax)_i &\geq b_i & \forall i \in I(y), \\ (Ax)_i &= b_i & \forall i \in M \setminus I(y), \\ x_j &\geq 0 & \forall j \in J(y), \\ x_j &= 0 & \forall j \in N \setminus J(y). \end{aligned}$$

Se esse problema for viável, tomemos uma solução x . Esse vetor x tem folgas complementares às de y , e portanto é solução para o problema de programação linear, e portanto o método pára devolvendo x e y .

Se o problema for inviável, então o seguinte problema tem solução: encontrar um vetor y' indexado por M tal que

$$\begin{aligned} y'b &> 0, \\ (y'A)_j &\leq 0 & \forall j \in J(y), \\ y'_i &\geq 0 & \forall i \in I(y). \end{aligned}$$

Seja y' uma solução desse problema. Se, para todo número positivo θ , $y + \theta y'$ for solução viável do dual, então o dual é ilimitado e o método devolve y' , vetor de inviabilidade do primal. Caso contrário, uma nova iteração é iniciada com $y + \theta y'$ no lugar de y , onde θ é o maior número tal que $y + \theta y'$ é viável no dual.

2.2.2 Aproximação primal-dual

No caso aproximado, o método é semelhante. A diferença é que são procuradas soluções suficientemente próximas, e não exatas. Essa proximidade é medida pelas folgas aproximadas.

Sejam α e β dois números tais que $0 < \alpha \leq 1 \leq \beta$. Os vetores x e y têm folgas α -aproximadas no primal se $x_j = 0$ ou $(yA)_j \geq \alpha c_j$, para todo índice j de x , e folgas β -aproximadas no dual se $y_i = 0$ ou $(Ax)_i \leq \beta b_i$, para todo índice i de y .

Assim, temos que $\alpha cx \leq (yA)x = y(Ax) \leq \beta yb$, pois x e y são não-negativos.

Esse método vai gerar uma solução x , se existir, tal que $cx \leq \frac{\beta}{\alpha} c\hat{x}$, onde $\hat{x}$ é a solução ótima do problema. Portanto, o método devolve uma solução $\frac{\beta}{\alpha}$ -aproximada.

Como no caso clássico, cada iteração começa com um vetor y viável no dual. Sejam

$$I(y) = \{i \in M : y_i = 0\} \text{ e } J(y, \alpha) = \{j \in N : (yA)_j \geq \alpha c_j\}.$$

Da mesma forma que no método clássico, considere o problema a seguir, onde se quer encontrar um vetor x indexado por N :

$$\begin{aligned} (Ax)_i &\geq b_i & \forall i \in I(y), \\ b_i &\leq (Ax)_i \leq \beta b_i & \forall i \in M \setminus I(y), \\ x_j &\geq 0 & \forall j \in J(y, \alpha), \\ x_j &= 0 & \forall j \in N \setminus J(y, \alpha). \end{aligned}$$

Se esse problema é viável, então x é uma solução viável do primal tal que $\alpha cx \leq \beta yb$ e o método devolve x e y .

Se o problema for inviável, então o seguinte problema tem solução: encontrar um vetor y' indexado por M tal que

$$\begin{aligned} y'b &> 0, \\ (y'A)_j &\leq 0 & \forall j \in J(y, \alpha), \\ y'_i &\geq 0 & \forall i \in I(y). \end{aligned}$$

Seja y' uma solução desse problema. Se, para todo número positivo θ , $y + \theta y'$ for solução viável do dual, então o dual é ilimitado e o método devolve y' , vetor de inviabilidade do primal. Caso contrário, uma nova iteração é iniciada com $y + \theta y'$ no lugar de y , onde θ é o maior número tal que $y + \theta y'$ é solução do dual.

Sejam x e y os vetores devolvidos pelo método. É fácil verificar que são soluções viáveis do primal e do dual, respectivamente, e que tem folgas α -aproximadas no primal e β -aproximadas no dual. Pelo lema das folgas aproximadas, $\alpha cx \leq \beta yb$. Assim

$$cx \leq \frac{\beta}{\alpha}yb \leq \frac{\beta}{\alpha}\hat{y}b = \frac{\beta}{\alpha}c\hat{x},$$

onde $\hat{x}$ é uma solução ótima do primal e $\hat{y}$ é uma solução ótima do dual. Assim, x é uma solução $\frac{\beta}{\alpha}$ -aproximada.

Capítulo 3

Uma $2k$ -aproximação para o SNDP com custo de tempo $O(n^4k^3)$

Este foi o primeiro algoritmo de aproximação desenvolvido para o SNDP, no início da década de 90, por Williamson, Goemans, Mihail e Vazirani. Devolve uma solução viável $2k$ -aproximada para o problema.

O algoritmo aproxima a solução de um problema inteiro, baseado numa função própria f , definida como $f(S) = \max_{i \in S, j \notin S} r_{ij}$, para todo $S \subseteq V$.

O seguinte programa inteiro modela o problema (IP):

$$\begin{array}{ll} \min & \sum_{e \in E} c_e x_e \\ \text{sujeito a} & x(\delta(S)) \geq f(S) \quad S \subset V \\ & x_e \in \{0, 1\} \quad e \in E \end{array}$$

onde $\delta(S)$ representa as arestas com exatamente uma ponta em S .

Vamos mostrar que esse problema é equivalente ao SNDP.

Lema 3.0.1 *x satisfaz (IP) se e somente se existe G' subgrafo de G que satisfaz os requisitos r_{ij} e x representa as arestas de G' .*

Prova: ($\Rightarrow$) Sejam l e m vértices de G , e $\mathcal{S}$ a coleção de todos os conjuntos $S \subset V$ tais que $l \in S$ e $m \notin S$. Como x satisfaz (IP), então, para todo $S \in \mathcal{S}$, $x(\delta(S)) \geq r_{lm}$, ou seja, o corte mínimo entre l e m tem pelo menos r_{lm} arestas. Assim, pelo teorema de Menger [2], existem pelo menos r_{lm} caminhos disjuntos entre l e m .

($\Leftarrow$) Seja $S \subset V$, e suponhamos que $x(\delta(S)) < f(S)$. Tomemos $l \in S$ e $m \notin S$, tais que $f(S) = r_{lm}$. Como não existem pelo menos r_{lm} arestas ligando S e $V \setminus S$, não é possível haver esse número de caminhos disjuntos ligando l e m .

□

Este é um algoritmo de aproximação dividido em k fases. Em cada uma das fases é resolvida uma parte do problema, utilizando-se método primal-dual para a aproximação. Ao final da fase p , todos os requisitos $r_{ij} \leq p$ devem estar satisfeitos. Assim, após a k -ésima fase, teremos uma solução viável.

Na fase p , definimos o p -ésimo truncamento da função f como $g_p(S) = \min(f(S), p)$. Note que g_p também é própria, pois:

$$\begin{aligned} g_p(V \setminus S) &= \min(f(V \setminus S), p) \\ &= \min(f(S), p) \\ &= g_p(S) \end{aligned}$$

$$\begin{aligned} g_p(A \cup B) &= \min(f(A \cup B), p) \\ &\leq \min(\max\{f(A), f(B)\}, p) \\ &= \max\{\min(f(A), p), \min(f(B), p)\} \\ &= \max\{g_p(A), g_p(B)\} \end{aligned}$$

Falta apenas mostrar que $\min(\max\{f(A), f(B)\}, p) = \max\{\min(f(A), p), \min(f(B), p)\}$. Vamos supor, sem perda de generalidade, que $f(A) \geq f(B)$. Temos 3 casos:

1) $p > f(A)$. Neste caso

$$\begin{aligned} \min(\max\{f(A), f(B)\}, p) &= \min(f(A), p) = f(A) \text{ e} \\ \max\{\min(f(A), p), \min(f(B), p)\} &= \max\{f(A), f(B)\} = f(A) \end{aligned}$$

2) $f(A) \geq p \geq f(B)$. Agora

$$\begin{aligned} \min(\max\{f(A), f(B)\}, p) &= \min(f(A), p) = p \text{ e} \\ \max\{\min(f(A), p), \min(f(B), p)\} &= \max\{p, f(B)\} = p \end{aligned}$$

3) $f(B) > p$. Neste último

$$\begin{aligned} \min(\max\{f(A), f(B)\}, p) &= \min(f(A), p) = p \text{ e} \\ \max\{\min(f(A), p), \min(f(B), p)\} &= \max\{p, p\} = p \end{aligned}$$

O algoritmo deve assegurar que, ao final da fase p , a união das arestas escolhidas nas primeiras p fases, F_p , satisfaz g_p , isto é, para todo $S \subset V$, $|\delta_{F_p}(S)| \geq g_p(S)$, onde $\delta_A(S) = \delta(S) \cap A$. Para conseguirmos satisfazer essa condição, devemos cobrir todos os cortes $\delta(S)$ para os quais $g_p(S) = p$ e $|\delta_{F_{p-1}}(S)| = p - 1$ com arestas de $E - F_{p-1}$. Definimos $h : 2^V \rightarrow \{0, 1\}$ como a função em que $h(S) = 1$ se e somente se $g_p(S) = p$ e $|\delta_{F_{p-1}}(S)| = p - 1$. Essa função é não cruzável. Então, na fase p , o algoritmo escolhe arestas em $E_h = E - F_{p-1}$ que satisfazem h .

O problema a ser resolvido em cada fase pode ser modelado através do seguinte programa inteiro, (IP_h) :

$$\begin{aligned} \min \quad & \sum_{e \in E_h} c_e x_e \\ \text{sujeito a} \quad & \sum_{e \in \delta(S)} x_e \geq h(S) \quad S \subset V \\ & x_e \in \{0, 1\} \quad e \in E_h \end{aligned}$$

Na p -ésima fase, o algoritmo encontra uma solução aproximada para (IP_h) e um empacotamento de cortes (*packing of cuts*) correspondente para o dual da relaxação linear de (IP_h) :

$$\begin{aligned} \max \quad & \sum_S h(S) y_S \\ \text{sujeito a} \quad & \sum_{S: e \in \delta(S)} y_S \leq c_e \quad e \in E_h \\ & y_S \geq 0 \quad \emptyset \neq S \neq V \end{aligned}$$

Lema 3.0.2 *Sejam F' o conjunto de arestas e y a solução dual viável construída pelo algoritmo para (IP_h) . Então*

$$\sum_{e \in F'} c_e \leq (2 - \frac{2}{l}) \sum_{S \subset V} h(S) y_S,$$

onde l é o número máximo de conjuntos disjuntos S de V para os quais $h(S) = 1$.

A prova deste lema consiste em mostrar que a solução dual encontrada pode ser transformada em uma solução dual viável para a relaxação linear de (IP) de pelo menos o mesmo valor. Além disso, as arestas escolhidas na fase p , F' , têm peso entre $2 \sum_{S \subset V} h(S) y_S \leq 2 \times Z_{IP}^*$, onde Z_{IP}^* é o custo da solução ótima para (IP) . Somando todas as fases, obtemos uma solução para (IP) que tem peso, no máximo, $2k \times Z_{IP}^*$. Faremos a prova desse lema após descrever o algoritmo e sua implementação.

3.1 Algoritmo de aproximação para funções não cruzáveis

Vamos descrever um algoritmo de aproximação para (IP_h) para qualquer função não cruzável h . Dadas uma função não cruzável h e um conjunto de arestas E_h , o algoritmo funciona em dois passos. No primeiro, começa com um conjunto de arestas F , vazio, e adiciona arestas a F até que F seja viável. No segundo passo, remove arestas redundantes.

A função utilizada deve ser não cruzável para garantir que o algoritmo não se torne exponencial. Sendo não cruzável a função h , todos os conjuntos de vértices utilizados no algoritmo (violados minimais) são disjuntos e formam uma família de conjuntos laminares. Dessa forma, o número total de conjuntos não pode ser muito grande. Quando dizemos que um conjunto é violado “minimal”, queremos dizer que nenhum subconjunto próprio dele é violado.

Lema 3.1.1 $|\delta_F(S)|$ é submodular.

Prova. Vamos chamar de $|\delta_F(A : B)|$ o número de arestas de F que ligam A e B , e seja $C = (V \setminus B) \setminus A$. Temos que $|\delta_F(A \cup B)| = |\delta_F(A)| + |\delta_F(B)| - |\delta_F(A \cap B)_{B \setminus A}| - |\delta_F(A \cap B : A \setminus B)| - |\delta_F(A \cap B : C)| - 2|\delta_F(A \setminus B : B \setminus A)|$, e também que $|\delta_F(A \cap B)| = |\delta_F(A \cap B : B \setminus A)| + |\delta_F(A \cap B : A \setminus B)| + |\delta_F(A \cap B : C)|$. Portanto, $|\delta_F(A)| + |\delta_F(B)| \geq |\delta_F(A \cup B)| + |\delta_F(A \cap B)| = |\delta_F(A)| + |\delta_F(B)| - 2|\delta_F(A \setminus B : B \setminus A)|$.

□

Teorema 3.1.2 *Sejam h uma função não cruzável e F qualquer subconjunto de E_h . Então os conjuntos violados minimais com relação a h são disjuntos.*

Prova. Suponha que A e B são conjuntos violados minimais e que A e B não são disjuntos. Como A e B são violados, $h(A) = h(B) = 1$. Pela definição de função não cruzável, ou $h(A \setminus B) = h(B \setminus A) = 1$ ou $h(A \cup B) = h(A \cap B) = 1$. Suponha que o segundo caso vale (a prova para o outro é semelhante). Pela submodularidade de $|\delta_F(S)|$, temos $|\delta_F(A)| + |\delta_F(B)| \geq |\delta_F(A \cap B)| + |\delta_F(A \cup B)|$. Como A e B são violados, então $|\delta_F(A)| = |\delta_F(B)| = 0$. Assim, $|\delta_F(A \cap B)| = |\delta_F(A \cup B)| = 0$ e, portanto, $A \cap B$ e $A \cup B$ são violados, o que contradiz a minimalidade de A e B e, portanto, A e B devem ser disjuntos. $\square$

Vamos agora dar mais detalhes do algoritmo. Ele começa com uma solução não viável para o primal, $F = \emptyset$, e uma solução viável dual, $y_S = 0$ para todo S . Enquanto a solução do primal não é viável, o algoritmo executa um passo de incremento primal-dual. Nesse passo, o algoritmo identifica os conjuntos violados minimais (*ativos*). Vamos denotar por $\mathcal{C}$ a coleção de conjuntos ativos em uma iteração. O algoritmo uniformemente aumenta as variáveis y_C correspondentes aos conjuntos $C \in \mathcal{C}$ até que alguma das restrições duais para alguma aresta $e \in E_h$ fique justa, ou seja,

$$c_e = \sum_{S: e \in \delta(S)} y_S.$$

Então e é acrescentada a F . Quando F se torna viável para o primal, ocorre o segundo passo, em que as arestas são olhadas na ordem inversa em que foram inseridas. Seja e a aresta sendo analisada. Se $F - e$ ainda é viável, então e é removida de F . Detalhes estão na próxima seção, onde é descrita a implementação deste algoritmo.

Para que o algoritmo funcione, a função h precisa ser não cruzável. A seguir, demonstraremos que isso de fato é verdade.

Observação 3.1.3 *Seja f uma função própria, e seja A, B, C uma partição de V . Então o máximo de $f(A)$, $f(B)$ e $f(C)$ não é unicamente atingido.*

Prova. Seja C o conjunto que atinge o máximo. Então, como f é simétrica, $f(C) = f(V \setminus C)$. Como f é maximal e $V \setminus C = A \cup B$, $f(C) \leq \max\{f(A), f(B)\}$. $\square$

Corolário 3.1.4 *Seja f uma função própria. Para conjuntos disjuntos A e B , o máximo de $f(A)$, $f(B)$ e $f(A \cup B)$ não é unicamente atingido.*

Lema 3.1.5 *Sejam f uma função própria e A e B conjuntos de vértices tais que $f(A) \geq p$ e $f(B) \geq p$. Então, ou $f(A \setminus B) \geq p$ e $f(B \setminus A) \geq p$, ou $f(A \cap B) \geq p$ e $f(A \cup B) \geq p$.*

Prova. Assuma, por exemplo, que $f(A \setminus B) < p$. Então, aplicando o corolário acima para $A \setminus B$, $A \cap B$ e A , vemos que $f(A \cap B) \geq p$. Aplicando o corolário para $A \setminus B$, B e $A \cup B$ temos que $f(A \cup B) \geq p$. Os outros casos são similares. $\square$

Teorema 3.1.6 *Seja F_{p-1} um conjunto de arestas tal que $|\delta_{F_{p-1}}(S)| \geq g_{p-1}(S)$, para todo $S \subseteq V$, sendo $g_p(S) = \min(f(S), p)$. Seja $h(S) = 1$ se $g_p(S) = p$ e $|\delta_{F_{p-1}}(S)| = p - 1$, e $h(S) = 0$ caso contrário. Então, h é não cruzável.*

Prova. É fácil ver que $h(V) = 0$, já que $g_p(V) = 0$. Sejam A e B conjuntos de vértices tais que $h(A) = h(B) = 1$. Como g_p é própria, então ou $g_p(A \cup B) = g_p(A \cap B) = p$ ou $g_p(A \setminus B) = g_p(B \setminus A) = p$. Suponha que vale a primeira. Como F_{p-1} satisfaz g_{p-1} , devemos ter $|\delta_{F_{p-1}}(A \cup B)| \geq p - 1$ e $|\delta_{F_{p-1}}(A \cap B)| \geq p - 1$. Mas, pela submodularidade de $|\delta_{F_{p-1}}(S)|$, temos que

$$|\delta_{F_{p-1}}(A)| + |\delta_{F_{p-1}}(B)| \geq |\delta_{F_{p-1}}(A \cup B)| + |\delta_{F_{p-1}}(A \cap B)|.$$

Assim, devemos ter $|\delta_{F_{p-1}}(A \cap B)| = |\delta_{F_{p-1}}(A \cup B)| = p - 1$. Então $h(A \cap B) = h(A \cup B) = 1$. O outro caso é idêntico. □

3.2 Conjuntos ativos

Para que o algoritmo tenha tempo de execução polinomial, é necessário encontrar conjuntos ativos e realizar a remoção de arestas em tempo polinomial. Como a remoção de arestas se reduz a encontrar conjuntos ativos, vamos nos concentrar nesse caso. Mostraremos que podemos encontrar conjuntos ativos utilizando fluxos em rede para funções não cruzáveis.

Os conjuntos violados minimais têm propriedades estruturais especiais. Considere uma fase p do algoritmo e seja F o conjunto das arestas já selecionadas. Vamos provar que para qualquer conjunto ativo S existem vértices $u \in S$ e $v \in \bar{S}$ tais que o fluxo máximo de u a v no grafo $(V, F \cup F_{p-1})$, com capacidades unitárias nas arestas, ajuda a identificar S ; S será o corte mínimo entre u e v que contém u . Assim, a implementação utiliza técnicas já conhecidas de fluxo máximo.

Lema 3.2.1 *Seja S um conjunto violado minimal com relação ao conjunto de arestas $H = F_{p-1} \cup F$. Então existe algum $u \in S$, $f(u) \geq p$, tal que não existe $T \subset S, u \in T$, com $|\delta_H(T)| \leq p - 1$.*

Prova. Por contradição. Suponha que para todo $u_i \in S$, com $f(u_i) \geq p$ existe um conjunto $T_i \subset S$ tal que $|\delta_H(T_i)| \leq p - 1$. Vamos supor que $f(\bigcup T_i) \leq p - 1$. Se for verdadeira, então como $f(S) \geq p$, devemos ter $f(S \setminus \bigcup T_i) = f(S) \geq p$ pelo corolário 3.1.4. Mas então, por maximalidade, deve existir algum $u \in (S \setminus \bigcup T_i)$ com $f(u) \geq p$, uma contradição.

Vamos provar a suposição. Por submodularidade, para qualquer par T_i e T_j na coleção de conjuntos $\{T_i\}$, temos $|\delta_H(T_i)| + |\delta_H(T_j)| \geq |\delta_H(T_i \setminus T_j)| + |\delta_H(T_j \setminus T_i)|$. Como tanto $|\delta_H(T_i)| \leq p - 1$ quanto $|\delta_H(T_j)| \leq p - 1$, então ou $|\delta_H(T_i \setminus T_j)| \leq p - 1$ ou $|\delta_H(T_j \setminus T_i)| \leq p - 1$. Sem perda de generalidade, suponha que o primeiro é verdadeiro. Então, substitua T_i por $T_i \setminus T_j$ na coleção. Note que $\bigcup T_i$ continua o mesmo. Continue o processo até que $\{T'_i\}$ seja uma coleção de conjuntos disjuntos dois a dois com $\bigcup T'_i = \bigcup T_i$. Como cada conjunto é substituído por um menor o processo termina. Cada T'_i tem $|\delta_H(T'_i)| \leq p - 1$, e como S é um conjunto violado minimal, devemos ter $f(T'_i) \leq p - 1$. Então, pela propriedade de maximalidade da função própria f , $f(\bigcup T'_i) \leq p - 1$, o que implica que $f(\bigcup T_i) \leq p - 1$. □

Lema 3.2.2 *Seja S um conjunto violado com relação ao conjunto de arestas $H = F_{p-1} \cup F$. Então existe algum $u \in S$, $f(u) \geq p$, tal que não existe $T \subset S, u \in T$, com $|\delta_H(T)| \leq p - 2$.*

Prova. Como acima. Na prova acima usamos a minimalidade de S para mostrar que se $|\delta_h(T'_i)| \leq p - 1$, então $f(T'_i) \leq p - 1$. Nesta prova, quando $|\delta_h(T'_i)| < p - 1$, o fato de que H satisfaz os requerimentos de todo corte S com $f(S) \leq p - 1$ implica que $f(T'_i) < p - 1$. $\square$

Teorema 3.2.3 *Dado qualquer conjunto violado minimal S com relação ao conjunto de arestas $H = F_{p-1} \cup F$, existe um par de vértices $u \in S$, $v \notin S$ tais que o fluxo máximo $u - v$ tem valor $p - 1$ e S é o corte mínimo $u - v$ minimal contendo u .*

Prova. Seja u o vértice de S especificado no lema 3.2.1. Seja v o vértice de $V \setminus S$ especificado no lema 3.2.2 (note que $V \setminus S$ é violado porque $f(V \setminus S) = f(S) \geq p$). Portanto, não há $T \subset S$ tal que $u \in T$ e $|\delta_H(T)| \leq p - 1$, e nenhum $T \subset V \setminus S$ tal que $v \in T$ e $|\delta_H(T)| \leq p - 2$. Considere o fluxo máximo de u a v . Como $|\delta_H(S)| = p - 1$, o fluxo não pode ser maior que $p - 1$. Suponha que o fluxo é $L \leq p - 1$, e que C é o menor corte contendo u . Por hipótese, não podemos ter $C \subset S$. Se $C \cap S \neq S$, então como $L + (p - 1) = |\delta_H(C)| + |\delta_H(S)| \geq |\delta_H(C \cap S)| + |\delta_H(C \cup S)|$, e $|\delta_H(C \cap S)| \geq p$, segue que $|\delta_H(C \cup S)| < L$. Mas $C \cup S$ separa u e v ; isto contradiz o fato que C é um corte mínimo. Então devemos ter $S \subseteq C$. Como $V \setminus C \subseteq V \setminus S$ e $v \in V \setminus C$, lema 3.2.2 implica que $|\delta_H(C)| = p - 1$. Como o fluxo máximo é $p - 1$ e $S \subseteq C$, S deve ser o corte mínimo minimal contendo u . $\square$

Assim, podemos identificar um conjunto ativo S com uma computação de fluxo máximo, se escolhermos adequadamente u e v ; depois de computar o fluxo, escolhemos os vértices do grafo residual acessíveis a partir de u . Para identificar todos os conjuntos ativos, achamos o fluxo máximo $u - v$ para todos os pares de vértices u, v , e mantemos os minimais com valor $p - 1$. Desses, utilizamos os conjuntos minimais por inclusão. Então, achamos quais dos conjuntos S tem $f(S) \geq p$. Esses são os conjuntos ativos da fase p .

Esse processo é feito no início de cada iteração numa fase. Como uma fase tem, no máximo, n iterações, em cada fase são necessárias, no máximo, $O(n^3)$ computações de fluxo máximo.

Na seção sobre implementação, os números de iterações e custos de tempo serão mostrados.

3.3 Implementação

Este algoritmo foi implementado, em linguagem C, utilizando a biblioteca SGB (Stanford Graph Base), de Donald Knuth, para os grafos. Utilizamos sempre grafos completos, fazendo as arestas que não existem na entrada terem custo 0. As arestas *acrescentadas* são utilizadas para armazenar o valor do requisito entre suas pontas, e para guardar o corte mínimo entre elas, como descrito a seguir.

Na descrição da implementação, quando mencionarmos as arestas do grafo, estaremos sempre nos referindo ao grafo completo, e não ao grafo original.

3.3.1 Estruturas de dados

As principais estruturas de dados utilizadas são as fornecidas pela biblioteca SGB (*Stanford Graph Base*), de Donald Knuth, e são a base de todas as manipulações com grafos. Maiores referências sobre o SGB estão em [14].

Além do SGB, foram utilizadas mais duas estruturas auxiliares. A primeira é a estrutura **Aresta**, utilizada para armazenar a lista de arestas escolhidas. É uma estrutura simples, que mantém um ponteiro para o primeiro vértice da aresta, um para o arco equivalente a ela na estrutura do SGB (que por sua vez tem um ponteiro para o outro vértice, o custo da aresta, e um ponteiro para outra **Aresta**, formando assim uma lista ligada.

A outra estrutura é a **Corte**. Essa estrutura é formada por um vetor de bits, no qual a posição i indica se o vértice i pertence ou não ao corte, e um inteiro, indicando o tamanho do corte. Cada aresta do grafo completo tem uma estrutura **Corte**, representando o corte mínimo, naquele momento, entre os vértices que são pontas da aresta.

Um grafo auxiliar é utilizado, servindo para serem encontrados os cortes mínimos entre cada par de vértices.

3.3.2 Função principal

Esta função recebe um grafo completo, com custos nas arestas, onde custo zero significa que a aresta não faz parte do grafo original, e um conjunto de requisitos entre cada par de vértices. Antes de ser chamada esta função, há um pré-processamento no qual o grafo e os requisitos são lidos e a estrutura do grafo no SGB é construída.

Primeiramente, é encontrado o maior requisito dado na entrada, k , e as estruturas Corte de cada aresta são inicializadas.

Em seguida, a função *uncrossable*, descrita a seguir, é chamada k vezes, representando as fases do algoritmo. Ao final, é verificado se a resposta dada é realmente viável, e esta é impressa.

Pseudo-código da função:

Entrada: Um grafo não orientado $G = (V, E)$, custos $c_e \geq 0$ nas arestas, e um conjunto de requisitos r .

Saída: Um conjunto de arestas F viável para (IP)

```
01   Para toda aresta  $a \in E$  inicializa o grafo residual
02    $k = \max r_{ij}, i, j \in V$ 
03    $F = \emptyset$ 
04   Para  $p$  de 1 até  $k$ 
05        $F = \text{uncrossable}(G, c, p, F)$ 
06   Imprime  $F$ 
```

3.3.3 Função *uncrossable*

Esta função recebe um grafo completo, com custos nas arestas, sendo que as arestas que não fazem parte do grafo original têm custo zero, o número da fase e o conjunto de arestas já

escolhidas nas fases anteriores. As arestas de custo zero não podem ser escolhidas pelo algoritmo. Esse conjunto de arestas escolhidas é representado por uma lista ligada da estrutura **Aresta**.

A função começa inicializando os conjuntos ativos, que é a parte mais custosa, como veremos em seguida. Após essas inicializações começa a laço principal da função.

Dentro do laço, é encontrada a próxima aresta que deve ser acrescentada à solução. Essa aresta é marcada, colocada na lista com as arestas escolhidas, e os conjuntos ativos são atualizados, seguindo o mesmo procedimento utilizado para sua inicialização. O laço principal é executado enquanto houver algum conjunto ativo.

Finalizado o laço principal, é executado o processo chamado de *reverse delete*. As arestas são analisadas na ordem contrária de sua escolha, e é verificado se elas realmente são necessárias. Uma aresta é necessária se, ao ser retirada da solução, a torna inviável. As que não são necessárias, são eliminadas da solução.

A seguir, um pseudo-código da função, seguida da análise do custo de tempo de execução.

Exemplo de execução do programa:

Figura 3.1: Grafo inicial

Na figura 1 vemos o grafo inicial, com os números próximos às arestas representando o custo da aresta. Os requisitos são os seguintes: $r(0, 3) = 2$, $r(1, 4) = 2$, $r(1, 5) = 2$ e $r(2, 4) = 1$.

Na figura 2, vemos a escolha da primeira aresta. Cada “bola” representa um conjunto ativo. Vemos que em dois pontos duas bolas se tocam, e são essas arestas que podem ser escolhidas num primeiro passo. Neste ponto é escolhida uma das arestas, e na figura 3 a outra.

Na figura 4, vemos que há 5 arestas onde as bolas se tocam, e que podem ser a próxima escolha. Três delas são escolhidas, como podemos ver nas figuras de 4 a 6, finalizando a primeira fase, pois na figura temos uma árvore formada com as arestas escolhidas.

Figura 3.2: Escolha da primeira aresta

Figura 3.3: Escolha da segunda aresta

Na figura 7, começa a segunda fase. Aqui, acontece apenas um toque entre duas bolas, indicando que apenas uma aresta pode ser escolhida.

Figura 3.4: Escolha da terceira aresta

Na figura 8, é escolhida a última aresta necessária, finalizando assim o programa. A solução final tem custo 14, enquanto o ótimo para o problema tem custo 12.

Entrada: Um grafo não orientado $G = (V, E)$, custos $c_e \geq 0$ nas arestas, o número p da fase e um conjunto de arestas F_p já escolhidas

Saída: Um conjunto de arestas F' viável para (IP_h)

```

01   $F \leftarrow \emptyset$ 
02   $i \leftarrow 0$ 
03   $d(v) \leftarrow 0$  para todo  $v \in V$ 
04  Comentário: Implicitamente fazendo  $y_S \leftarrow 0$  para todo  $S \subset V$ 
05  Para todo  $v \in V$ 
06      Para todo  $w \in V, w \neq v$ 
07          Inicializa o grafo residual entre  $v$  e  $w$ 
08          Encontra o corte mínimo de  $v$  a  $w$  (minimal_mincut)
09  Encontra, para cada vértice, o menor corte mínimo que o contém (minimal_sets)
10  Enquanto houver conjuntos ativos (mark_active)
11       $i \leftarrow i + 1$ 

```

Figura 3.5: Escolha da quarta aresta

- 12 Para todo vértice v de um conjunto ativo, incremente $d(v)$ uniformemente de ε até que alguma aresta $e_i = (u, v) \in E_h$ satisfaça $d(u) + d(v) = c_{uv}$ para $e_i \in \delta(C)$ de algum conjunto ativo C
- 13 *Comentário: Implicitamente fazendo $y_C \leftarrow y_C + \varepsilon$, para todo conjunto ativo C*
- 14 $F \leftarrow F \cup \{e_i\}$
- 15 Atualize os conjuntos ativos
- 16 *Comentário: Passo de limpeza*
- 17 $F' \leftarrow F$
- 18 Para $j \leftarrow i$ até 1
- 19 Se $F' - \{e_j\}$ satisfaz h
- 20 $F' \leftarrow F' - \{e_j\}$
- 21 Devolva F'

A linha 3 é executada em tempo $O(n)$, pois um vetor de tamanho n é percorrido.

A linha 7 é executada em tempo $O(nk)$. Nessa inicialização, é criado um novo grafo, que no SGB tem tempo de execução $O(n)$. Depois disso, todas as arestas já escolhidas são acrescentadas ao grafo. Como cada acréscimo de aresta tem tempo de execução $O(1)$, e, ao final do algoritmo, são escolhidas no máximo nk arestas, o custo de acrescentá-las é $O(nk)$. Assim, o custo da linha 7 é $O(n) + O(nk) = O(nk)$.

Figura 3.6: Escolha da quinta aresta

A linha 8 é executada em tempo $O(nk^2)$, pois é encontrado um fluxo entre os vértices, que tem esse tempo, utilizando o algoritmo de Ford e Fulkerson. Esse algoritmo tem tempo de execução $O(ml)$, onde m é o número de arestas e l o maior fluxo possível. Como neste caso $m = nk$ (arestas da solução) e $l = k$, concluímos que o custo é $O(nk^2)$.

Como as linhas 5 e 6 indicam um loop a ser executado $O(n^2)$, uma vez para cada aresta, o custo total das linhas 5 a 8 é $O(n^3k^2)$.

A linha 9 é executada em tempo $O(n^3)$. Temos $O(n^2)$ grafos residuais e, em cada um deles, todos os vértices são verificados, o que tem custo $O(n)$.

Na linha 10, encontrar os conjuntos ativos custa tempo $O(n^3)$. Para cada vértice, é verificado o valor da função f para o conjunto marcado como o menor que contém esse vértice, que tem tempo de execução $O(n^2)$. Assim, para verificar todos os vértices, o tempo é $O(n^3)$. Depois disso, para cada par de vértices, é verificado se o menor conjunto violado que contém um está contido ou tem intersecção própria com o menor conjunto violado que contém o outro. Essa verificação custa $O(n)$, totalizando também $O(n^3)$ para cada par. Esse laço é executado $O(n)$ vezes, pois em uma fase, o máximo que podemos encontrar é uma árvore, que aumentaria em 1 a conexão entre qualquer par de vértices.

A linha 12 é executada em tempo $O(n^2)$. As arestas ainda não escolhidas são verificadas. Essa verificação é feita em tempo constante.

Na linha 15, todo o processo de inicialização feito nas linhas 5 a 9 é refeito. Portanto, o

Figura 3.7: Escolha da sexta aresta, início da segunda fase

custo dessa linha é $O(n^3k^2)$.

O laço das linhas 10 a 15 é executado no máximo n vezes, pois esse é o número máximo de conjuntos ativos na primeira iteração, e, a cada iteração, o número de conjuntos ativos decresce de pelo menos 1. Assim, o custo total desse laço é de $O(n^4k^2)$.

Para a linha 19, é preciso executar todos os passos das linhas 5 a 9, ou seja, o custo da linha 19 é $O(n^3k^2)$. Já a linha 20 pode ser executada em tempo constante ($O(1)$). Assim, o laço das linhas 18 a 20 tem tempo de execução $O(n^4k^2)$.

Dessa forma, concluímos que a função *uncrossable* tem custo de tempo total $O(n^4k^2)$.

Função *minimal_mincut*:

Esta função recebe dois vértices, u e v , um grafo residual G , possivelmente com alguns fluxos entre u e v já marcados, e o número p da fase atual.

Tenta-se encontrar um fluxo de tamanho $p - 1$. Caso seja encontrado, o corte mínimo é marcado e devolvido.

Entrada: Um grafo residual $G = (V, E)$, dois vértices v e u de G e a fase p que está sendo

Figura 3.8: Escolha da sétima aresta

executada.

- 01 Encontra o fluxo máximo F de u para v em G
- 02 Se $|F| == p - 1$
- 03 Encontra o corte determinado por F .

A linha 1 tem custo de tempo $O(nk^2)$, utilizando o algoritmo de Ford e Fulkerson, sendo o maior fluxo possível exatamente k . A linha 3 consiste simplesmente de uma busca em largura, que tem custo $O(nk)$ [4], resultando num custo de $O(nk^2)$ para a função.

Função *minimal_sets*

Recebe um grafo $G = (V, E)$, completo, e o número p da fase. Para cada vértice $v \in V$, guarda o menor corte entre os encontrados na função acima que contém v .

Cada corte é olhado, e se for menor que o corte atual de v , atualiza o corte mínimo contendo v .

Entrada: Um grafo não orientado $G = (V, E)$ e a fase p que está sendo executada.

01 Inicializa a variável onde fica o conjunto ativo em que está v , para todo $v \in V$.

02 Para todo par de vértices v, u de V

03 Se o corte do grafo residual de v a u é o menor que contém v , faça as variáveis de v apontarem esse corte como o conjunto violado em que está v

A linha 1 tem custo de tempo $O(n)$. A linha 3 tem custo $O(1)$, pois é olhado apenas o tamanho do corte. Portanto, as linhas 2 e 3 têm custo total $O(n^2)$, que é o custo total da função.

Função *mark_active*

Recebe um grafo $G = (V, E)$, completo, e o número p da fase. Nesta função, os conjuntos minimais encontrados anteriormente são verificados, e deles são retirados os conjuntos ativos.

Primeiro, os conjuntos com o valor de f menor do que p são descartados. Depois, apenas os minimais sob inclusão são mantidos.

Entrada: Um grafo não orientado $G = (V, E)$ e a fase p que está sendo executada.

01 Inicializa os vértices $v \in V$, marcando-os como não pertencentes a nenhum conjunto ativo.

02 Para cada vértice $v \in V$, se v tiver um conjunto violado, verifica o valor da função f para esse conjunto.

03 Se o valor de f for $< p$, desconsidera esse conjunto.

04 Senão, marca, no grafo residual onde está esse conjunto, os vértices que estão e que não estão no corte.

05 Para cada vértice $v \in V$, se ele tem um conjunto violado

06 Para cada vértice $w \in V$, verifica se o conjunto de w intersecta ou contém o conjunto de v

07 Se não contém, marca os vértices do conjunto de v como sendo desse conjunto ativo.

A linha 1 tem custo de tempo $O(n)$. A linha 3 tem custo $O(n^2)$, como será visto abaixo. A linha 4 tem custo $O(n)$. Portanto, as linhas 2 a 4 têm custo total $O(n^3)$. A linha 6 tem custo $O(n)$, assim como a linha 7. Então, as linhas 5 a 7 têm custo total $O(n^2)$. Assim, o custo total da função é $O(n^3)$.

Função *find_edge*

Esta função recebe um grafo G completo. As arestas *verdadeiras* de G , (as que existiam originalmente e que tem custo positivo) ainda não utilizadas são analisadas, e a que elevará menos o custo total é escolhida.

Entrada: Um grafo não orientado $G = (V, E)$, com custos c nas arestas

01 Para cada aresta $a \in E$ ainda não escolhida

- 02 Se a aresta liga dois vértices v e u que estão em conjuntos ativos diferentes, o peso da aresta é calculado como $\frac{c-d(v)-d(u)}{2}$
- 03 Se a aresta liga um vértice v em um conjunto ativo a outro que não pertence a nenhum conjunto ativo, então o peso da aresta é $c - d(v)$
- 04 Atualiza as variáveis $d(v)$ para todo vértice v que está em algum conjunto ativo, acrescentando o peso da aresta de menor peso
- 05 Devolve a aresta de menor peso.

As linhas 2 e 3 têm custo de tempo constante. Assim, as linhas de 1 a 3 têm custo total $O(n^2)$. A linha 4 tem custo $O(n)$ e, portanto, a função tem custo $O(n^2)$.

Função inicializa_residual

Entrada: Um grafo residual G e uma aresta f que acabou de ser acrescentada

- 01 Faz todas as capacidades das arestas de G iguais a 1
- 02 Insere a aresta f em G .

A linha 1 tem custo de tempo $O(nk)$, pois esse é o número máximo de arestas que podem ter sido escolhidas. Já a linha 2 tem custo de tempo $O(1)$. Assim, o custo total da função é $O(nk)$.

Função zera_aresta

Entrada: Um grafo residual G e uma aresta f que será zerada.

- 01 Encontra a aresta no grafo
- 02 Coloca a capacidade da aresta para zero

A linha 1 tem custo $O(n)$, pois são varridas as arestas da ponta indicada por f . A linha 2 tem custo $O(1)$, pois simplesmente o arco encontrado e o arco inverso são zerados.

Função remove_aresta

Recebe o grafo residual G e a aresta f , que liga os vértices v e u . Procura o arco de v para u na lista de arcos de v , e o remove. Procura o arco de u para v na lista de arcos de u , e o remove.

Entrada: Um grafo residual G e uma aresta f que será removida.

- 01 Encontra o primeiro arco no grafo e o remove
- 02 Encontra o segundo arco no grafo e o remove

Tanto a linha 1 quanto a linha 2 tem custo $O(n)$, que, portanto, é o custo total da função.

Função f

Entrada: Um grafo não orientado $G = (V, E)$ e um conjunto R de vértices

- 01 Para cada vértice $u \in R$, verifica os requisitos entre u e todos os outros vértices v
- 02 Se $v \notin R$, guarda o requisito de u a v
- 03 Devolve o valor do maior requisito

A linha 2 é executada em tempo constante e, portanto, a função toda tem custo de tempo $O(n^2)$.

3.3.4 Testes

Os testes realizados dividem-se em dois tipos: testes pequenos, para os quais é fácil encontrar o valor ótimo para comparações, e testes grandes, onde é possível observar se o tempo de execução cresce apropriadamente. Infelizmente, não encontramos teste grandes na literatura para o problema geral, por isso nos limitamos a testes de casos específicos, como o da árvore de Steiner, onde os requisitos são sempre 0 ou 1.

Os testes seguintes foram tirados da biblioteca *Steinlib* (<http://elib.zib.de/steinlib/steinlib.php>). Todos eles têm requisitos máximos de valor 1. Os valores ótimos foram obtidos do *site* da *Steinlib*. Foram realizados em um computador com processador Pentium 4 2.4Ghz e 512Mb de memória.

Problema	V	tempo	resultado	ótimo	razão
b01	50	0,5s	82	82	1
b06	50	0,9s	128	122	1,05
b08	75	2,0s	104	104	1
b11	75	2,2s	92	88	1,05
b14	100	10,6s	238	235	1,01
b18	100	12,4s	224	218	1,03
p609	100	6,5s	18972	18263	1,04
p612	100	13,1s	30353	30258	1,003
p615	200	4min23,8s	43953	42474	1,03
p616	200	4min17,5s	64240	62263	1,03
mc13	150	1min32,5s	111	92	1,2
mc3	97	12,8s	49	47	1,04
mc7	400	1h25min38,3s	3533	3417	1,03
mc11	400	1h33min22,1s	11923	11689	1,02
diw313	468	26min42,5s	397	397	1
diw250	353	20min0,2s	355	350	1,01

Os testes mostram resultados esperados, tanto com relação ao resultado obtido, quanto com relação ao tempo de execução. Como nestes casos temos $k = 1$, a complexidade de tempo do algoritmo é $O(n^4)$, e o crescimento do tempo de execução parece seguir bem essa complexidade.

Alguns desvios acontecem, como por exemplo no teste *p609* e no teste *diw313*, que tem tempo de execução um pouco abaixo do esperado, se levarmos em conta os tempos dos outros testes. No caso do *p609*, a diferença entre ele e o *p612* é o número de requisitos positivos. Enquanto o *p609* tem requisitos positivos entre 20 vértices, o *p612* tem requisitos entre 50 vértices. Esse número equivale ao número de terminais no problema da árvore de Steiner. A causa da diferença de tempo entre o *mc11* e o *diw313* é a mesma. Apesar do *diw313* ter mais vértices, tem requisitos entre 14 vértices apenas, enquanto o *mc11* tem requisitos entre 213. Outro fator que pode ter influenciado é o fato do *diw313* ser uma grade.

Além desses, foram feitos alguns testes com requisitos maiores, mas com poucos vértices.

Problema	V	k	tempo	resultado	ótimo	razão
teste0	6	2	0,002s	24	24	1
teste1	19	3	0,05s	108	100	1,08
teste2	7	2	0,002s	10	9	1,11

3.3.5 Prova da garantia de $2k$ -aproximação e do lema 3.0.2

Vamos provar o lema 3.3.2, e mostrar que o algoritmo encontra uma $2k$ - aproximação para o problema.

Como o algoritmo para (IP_h) mantém as folgas complementares primais, sabemos que para qualquer aresta $e \in F$, $\sum_{S:e \in \delta(S)} y_S = c_e$. Assim, o custo da solução F' é

$$\sum_{e \in F'} c_e = \sum_{e \in F'} \sum_{S:e \in \delta(S)} y_S$$

. Podemos reescrever o lado direito como $\sum_{S \subset V} y_S \cdot |\delta_{F'}(S)|$. A prova do lema será feita por indução no loop principal do algoritmo (while), para mostrar que

$$\sum_{S \subset V} y_S \cdot |\delta_{F'}(S)| \leq (2 - \frac{2}{l}) \sum_{S \subset V} h(S) y_S$$

. Como $y_S > 0$ somente quando $h(S) = 1$, a soma do lado direito se reduz a $\sum_S y_S$. A desigualdade é verdadeira antes da primeira iteração do loop, pois inicialmente todo $y_S = 0$. Considere o coleção $\mathcal{C}$ de conjuntos ativos no início de alguma iteração do loop. O lado esquerdo da desigualdade crescerá de $\sum_{C \in \mathcal{C}} \epsilon \cdot |\delta_{F'}(C)|$ nessa iteração, enquanto o lado direito cresce $(2 - \frac{2}{l})\epsilon \cdot |\mathcal{C}|$.

A prova do lema 3.3.2 virá da prova de que, em qualquer iteração,

$$\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq (2 - \frac{2}{l}) |\mathcal{C}|$$

. Em outras palavras, mostraremos que o grau médio dos conjuntos ativos com relação a F' não é maior que $2 - \frac{2}{l}$. A prova pode ser vista como um “esquema de troca”, na qual mostramos que há tantos conjuntos ativos de grau 1 que compensam os conjuntos ativos de grau alto. Daqui em diante, vamos falar de alguma iteração do algoritmo.

Definimos $H = \bigcup_{C \in \mathcal{C}} \delta_{F'}(C)$, ou seja, todas as arestas de F' que vieram de conjuntos ativos.

Lema 3.3.1 *Para cada aresta $e \in H$ existe um conjunto testemunha (witness) $S_e \subset V$ tal que*

1. $h(S_e) = 1$,
2. $\delta_{F'}(S_e) = e$,
3. Para todo $C \in \mathcal{C}$, ou $C \subseteq S_e$ ou $C \cap S_e = \emptyset$.

Prova. Qualquer aresta $e \in H$ também está em F' , e portanto durante o estágio de remoção de arestas (*reverse delete*), a remoção de e faz com que h seja violada para algum S . Ou seja, não existe nenhum outro $e' \in F'$ que está também em $\delta(S)$. Esse conjunto S será o conjunto testemunha (*witness*) S_e para e , e claramente satisfaz (1) e (2). Seja F_b o conjunto de todas as arestas acrescentadas antes da iteração corrente. Note que quando consideramos a aresta e no estágio de *reverse delete*, nenhuma aresta de F_b foi ainda removida. Então S_e é violado mesmo se todas as arestas de F_b estão incluídas; isto é, S_e é violado na iteração corrente. Assim, (3) segue da minimalidade dos conjuntos ativos C . □

Considere uma coleção de conjuntos S_e satisfazendo as condições do lema anterior, cobrindo todas as arestas $e \in H$. Uma coleção dessas é chamada de família testemunha (*witness family*). Qualquer coleção de conjuntos é *laminar* se para qualquer par de conjuntos A e B na coleção ou $A \subseteq B$ ou $B \subseteq A$ ou $A \cap B = \emptyset$.

Lema 3.3.2 *Existe uma família testemunha laminar.*

Prova. De acordo com o lema anterior, existe uma família testemunha. Dessa coleção de conjuntos formamos uma coleção laminar da seguinte forma. Sabemos que todos os conjuntos da coleção tem $h(S) = 1$. Se a coleção não for laminar, existe um par de conjuntos A, B tal que $A \not\subseteq B$, $B \not\subseteq A$ e $A \cap B \neq \emptyset$. Dizemos que A e B *cruzam*. Como $h(A) = h(B) = 1$, ou $h(A \setminus B) = h(B \setminus A) = 1$ ou $h(A \cup B) = h(A \cap B) = 1$. Se a última afirmação é verdadeira, “descruzamos” substituindo A e B na coleção por $A \cup B$ e $A \cap B$ (o outro caso é análogo). O processo termina com uma coleção laminar pois sempre que dois conjuntos são descruzados, o número total de pares de conjuntos que cruzam diminui. Para provar isso, note que se um conjunto X na coleção cruza tanto A quanto B , substituir A e B por $A \setminus B$ e $B \setminus A$, ou $A \cup B$ e $A \cap B$ não pode aumentar o número de conjuntos que X cruza. Se X cruza somente A e não está contido em B , então não pode cruzar $B \setminus A$ nem $A \cap B$. Se X cruza somente A e está contido em B , então não pode cruzar $A \setminus B$ nem $A \cup B$, e novamente descruzar A e B não pode aumentar o número total de conjuntos que cruzam X . Assim, descruzar A e B não aumenta o número total de pares que cruzam, e de fato diminui esse número em pelo menos um, já que A não cruza B mais.

Vamos agora mostrar que a coleção laminar resultante forma uma família testemunha. Vamos mostrar isso por indução no processo de descruzar. A propriedade (3) continua válida quando quaisquer dois conjuntos são descruzados. Suponha que temos dois conjuntos testemunha S_1 e S_2 correspondendo às arestas e_1 e e_2 tais que S_1 e S_2 se cruzam. Como h é não cruzável, ou $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$ ou $h(S_1 \setminus S_2) = h(S_2 \setminus S_1) = 1$. Sem perda de generalidade, suponha que $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$. Pela submodularidade, $2 = |\delta_{F'}(S_1)| + |\delta_{F'}(S_2)| \geq |\delta_{F'}(S_1 \cup S_2)| + |\delta_{F'}(S_1 \cap S_2)|$. Como $h(S_1 \cup S_2) = 1$, não podemos ter $S_1 \cup S_2 = V$. Então, se $h(S_1 \cup S_2) = h(S_1 \cap S_2) = 1$, temos $|\delta_{F'}(S_1 \cup S_2)| = |\delta_{F'}(S_1 \cap S_2)| = 1$. Portanto, ou $S_1 \cap S_2$ é um conjunto testemunha para e_1 e $S_1 \cup S_2$ é um conjunto testemunha para e_2 , ou vice versa. □

Seja $\mathcal{L}$ uma família laminar testemunha. Aumente a família com o conjunto V de vértices. A família pode ser vista como uma árvore P com um vértice v_S para cada $S \in \mathcal{L}$ e aresta (v_S, v_T) se T é o menor elemento de $\mathcal{L}$ que contém S propriamente. A cada conjunto ativo $C \in \mathcal{C}$ correspondemos o menor conjunto $S \in \mathcal{L}$ que o contém. Chamaremos um vértice v_S de ativo se S está associado a algum conjunto ativo C .

Lema 3.3.3 *A árvore P não tem folhas inativas.*

Prova. Apenas V e os conjuntos testemunha minimais sob inclusão podem ser folhas. Qualquer conjunto testemunha minimal é um conjunto violado, e portanto deve ter um conjunto ativo correspondente. Seja S um conjunto testemunha maximal. Tanto S quanto $V \setminus S$ são conjuntos violados, e portanto correspondem a conjuntos ativos C . Assim, v_V não pode ser simultaneamente uma folha e inativo. $\square$

Lema 3.3.4 *O grau de um vértice ativo em P é pelo menos a soma de $|\delta_{F'}(C)|$ de $C \in \mathcal{C}$ a que ele corresponde.*

Prova. Note que o mapeamento um a um entre as arestas de H e os conjuntos testemunha implica em um mapeamento de um para um entre as arestas de H e as de P : cada conjunto testemunha S define uma única aresta (v_S, v_T) de P , onde T contém S . Considere qualquer aresta $e \in \delta_{F'}(C)$ para algum $C \in \mathcal{C}$. Seja (v_S, v_T) a aresta definida pelo conjunto testemunha S_e . Ou v_S ou v_T deve ser o vértice ativo correspondente a C . Assumindo isto sobre todas as arestas $e \in \delta_{F'}(C)$ para todos os conjuntos ativos C correspondentes a um vértice ativo de P , obtemos o lema. $\square$

Seja P_a o conjunto de vértices ativos de P e seja d_v o grau do vértice v . Então, como mostrado em [16],

$$\begin{aligned} \sum_{v \in P_a} d_v &= \sum_{v \in P} d_v - \sum_{v \in P \setminus P_a} d_v \\ &\leq 2(|P| - 1) - 2(|P| - |P_a|) \\ &= 2|P_a| - 2. \end{aligned}$$

Esta desigualdade é válida pois P é uma árvore com $|P| - 1$ arestas e cada vértice em $P \setminus P_a$ tem grau mínimo 2. O lema acima implica que $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq \sum_{v \in P_a} d_v$, enquanto claramente $|P_a| \leq |\mathcal{C}|$. Assim $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq 2|\mathcal{C}| - 2$. Como $|\mathcal{C}| \leq 1$, temos que $\sum_{C \in \mathcal{C}} |\delta_{F'}(C)| \leq (2 - \frac{2}{l})|\mathcal{C}|$, como desejado.

Capítulo 4

Uma $2k$ -aproximação para o SNDP com custo de tempo $O(k^2n^2 + kn^2\sqrt{\log \log n})$

Este algoritmo é uma versão aprimorada do algoritmo apresentado no capítulo anterior. Tem a mesma razão de aproximação, mas consome um tempo menor. Para alcançar esse resultado, mantém a mesma estrutura básica, mudando detalhes de como a implementação é feita, resultando em um menor consumo de tempo. Foi descrito por Gabow, Goemans e Williamson [7].

Relembrando o problema, o SNDP consiste em, dados um grafo $G = (V, E)$, custos não negativos c_e para toda aresta $e \in E$, requisitos de conectividade r_{ij} para cada par de vértices $i, j \in V$, encontrar o subgrafo de G de custo mínimo no qual cada par de vértices i, j está ligado por no mínimo r_{ij} caminhos disjuntos nas arestas.

Toda a descrição do algoritmo anterior, com a modelagem por programação inteira, continuam valendo para este algoritmo. Temos, como no caso anterior, uma função $f(S) = \max_{i \in S, j \notin S} r_{ij}$ e o seguinte problema inteiro (IP) modela o problema:

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{sujeito a} \quad & x(\delta(S)) \geq f(S) \quad S \subset V \\ & x_e \in \{0, 1\} \quad e \in E, \end{aligned}$$

onde $\delta(S)$ denota as arestas da borda de S , isto é, arestas que tem uma ponta em S e outra fora.

Seja $k = \max_S f(S)$. Como o algoritmo anterior, este também é dividido em k fases. Em cada fase é encontrado um aumento na solução atual. Seja F_{p-1} as arestas escolhidas nas primeiras $p-1$ fases, e vamos denotar $A \cup \delta(S)$, com $A \subseteq E$ por $\delta_A(S)$. Assim, na fase p encontramos um conjunto de arestas $F \subseteq E - F_{p-1}$ tal que se $f(S) \geq p$ e $|\delta_{F_{p-1}}(S)| = p-1$, então $|\delta_F(S)| \geq 1$. Então fazemos $F_p = F_{p-1} \cup F$, para mantermos o invariante de que $|\delta_{F_p}(S)| \geq \min(f(S), p)$ para todos os subconjuntos S . Assim, ao final da última fase, o conjunto F_k é uma solução viável para (IP).

Cada fase pode ser vista como encontrar uma solução de baixo custo para o seguinte programa inteiro:

$$\begin{aligned} \min \quad & \sum_{e \in E_h} c_e x_e \\ \text{sujeito a} \quad & x(\delta(S)) \geq h(S) \quad S \subset V \\ & x_e \in \{0, 1\} \quad e \in E_h, \end{aligned}$$

onde $E_h = E - F_{p-1}$ e $h(S) = 1$ se e somente se $f(S) \geq p$ e $|\delta_{F_{p-1}}(S)| = p - 1$, e $h(S) = 0$ caso contrário.

O algoritmo em 3.3.2 continua válido. Relembrando:

Entrada: Um grafo não orientado $G = (V, E)$, custos $c_e \geq 0$ nas arestas, e um conjunto de requisitos r .

Saída: Um conjunto de arestas F viável para (IP)

```

01   Para toda aresta  $a \in E$  inicializa o grafo residual
02    $k = \max r_{ij}, i, j \in V$ 
03    $F = \emptyset$ 
04   Para  $p$  de 1 até  $k$ 
05      $F = \text{uncrossable}(G, c, p, F)$ 
06   Imprime  $F$ 

```

A função *uncrossable* sofre modificações na sua implementação, em relação ao algoritmo anterior. São essas mudanças que tornam o algoritmo mais rápido.

Entrada: Um grafo não orientado $G = (V, E)$, custos $c_e \geq 0$ nas arestas, o número p da fase e um conjunto de arestas F_p já escolhidas

Saída: Um conjunto de arestas F' viável para (IP_{*h*})

```

01    $F \leftarrow \emptyset$ 
02    $i \leftarrow 0$ 
03    $d(v) \leftarrow 0$  para todo  $v \in V$ 
04   Comentário: Implicitamente fazendo  $y_S \leftarrow 0$  para todo  $S \subset V$ 
05    $\mathcal{C} \leftarrow$  todos os conjuntos ativos  $C$ 
06   Enquanto houver conjuntos ativos ( $|\mathcal{C}| > 0$ )
07      $i \leftarrow i + 1$ 
08     Para todo vértice  $v$  de um conjunto ativo, incremente  $d(v)$  uniformemente de  $\varepsilon$ 
    até que alguma aresta  $e_i = (u, w) \in E_h$  satisfaça  $d(u) + d(w) = c_{uw}$  para  $e_i \in \delta(C)$  de algum
    conjunto ativo  $C$ 
09     Comentário: Implicitamente fazendo  $y_C \leftarrow y_C + \varepsilon$ , para todo conjunto ativo  $C$ 
10      $F \leftarrow F \cup \{e_i\}$ 
11     Atualize os conjuntos ativos
12   Comentário: Passo de limpeza
13    $F' \leftarrow F$ 
14   Desmarque todos os conjuntos que foram ativos nesta fase. Marque  $V$ 
15   Para  $j \leftarrow i$  até 1
16     Se  $e_j$  é especial e  $C(e_j)$  está desmarcado e  $\delta_{F'}(A(e_j)) \supseteq \delta_{F'}(C(e_j))$ 

```

```

17       $F' \leftarrow F' - \{e_j\}$ 
18      Marque  $C(e_j)$ 
19      Devolva  $F'$ 

```

Conjuntos ativos serão definidos abaixo, mas têm o mesmo significado que tinham no algoritmo anterior. Arestas especiais, $C(e)$ e $A(e)$ serão definidos posteriormente no texto.

O algoritmo Uncrossable tem dois passos. O primeiro produz iterativamente um conjunto de arestas F . Em cada iteração, há um passo de seleção, e outro de adição de arestas. O segundo passo é um passo de limpeza, em que arestas são removidas de F .

O primeiro passo inicializa F como vazio. Em qualquer ponto, um conjunto S é chamado de violado se $|\delta_F(S)| \leq h(S)$. Um conjunto S é *ativo* se for violado minimal (com relação a inclusão). Os conjuntos violados têm a seguinte propriedade:

Lema 4.0.5 *Se A e B são conjuntos violados em algum ponto do algoritmo, então ou $A \cap B$ e $A \cup B$ são violados, ou $A \setminus B$ e $B \setminus A$ são violados.*

A prova deste lema está em 3.1.2

Dizemos que o conjunto A cruza B se $A \cap B \neq \emptyset$, mas $A \not\subseteq B$ e $B \not\subseteq A$. Vamos agora definir um corolário do teorema 3.1.2. Uma família laminar de conjuntos $\mathcal{A}$ é tal que se $A, B \in \mathcal{A}$ e $A \cap B \neq \emptyset$, então ou $A \subseteq B$ ou $B \subseteq A$, ou seja, A e B não se cruzam.

Corolário 4.0.6 *Seja $\mathcal{UC}$ o conjunto de todos os conjuntos ativos formados em todas as iterações. Então $\mathcal{UC}$ é uma família laminar.*

4.1 Conjuntos ativos

Para encontrar os conjuntos ativos, é criado um grafo em que a aresta (i, j) tem custo r_{ij} . Nesse grafo, é encontrada uma árvore geradora máxima T .

Lema 4.1.1 *Qualquer conjunto que satisfaça os requerimentos de conectividade das arestas em T , satisfaz todos os requerimentos r_{ij} .*

Prova. Para qualquer requerimento r_{pq} que não está em T , existe um caminho em T ligando p a q . Seja $p, k, \dots, l, q$ o caminho de p a q em T . Como T é uma árvore geradora máxima, $r_{pq} \leq \min(r_{pk}, \dots, r_{lq})$ e, portanto, há pelo menos r_{pq} caminhos disjuntos em G entre p e q , se unirmos os caminhos entre $p, k, \dots, l, q$. □

Sejam S_e e $V \setminus S_e$ a partição do conjunto de vértices induzida pela remoção da aresta e da árvore T . Os conjuntos violados da fase p devem corresponder aos cortes mínimos de valor $p - 1$ associados a uma aresta $e = (s, t)$ de T tendo $f(S_e) = r_{st} \geq p$. Nesse caso, temos também que cada corte mínimo deve corresponder a um conjunto violado. Assim, qualquer conjunto ativo deve ser um corte mínimo *minimal* (s, t) ou (t, s) para algum $(s, t) \in T$.

Esses cortes mínimos minimais são mantidos durante todo o algoritmo. Para isso é mantido, para cada aresta $(s, t) \in T$, um fluxo $s - t$ de valor não maior que r_{st} . No início da fase p , começamos com os fluxos computados na fase anterior e encontramos caminhos de aumento

para cada aresta e de T até o valor p , se possível. Assim, na fase p , podemos detectar qualquer aresta (s, t) de T para a qual o corte vale $p - 1$, mas $r_{st} \geq p$. Durante todo o algoritmo, são encontrados no máximo k caminhos de aumento para cada aresta de T , resultando em um tempo de execução, em todas fases, de $O(km'n)$ para manter esses fluxos, onde $m' = \min(kn, m)$.

Suponha que para uma aresta $e = (s, t) \in T$, o valor do corte é $p - 1$ e $r_{st} \geq p$. Inicialmente, o corte mínimo minimal (s, t) é formado pelos vértices alcançáveis a partir de s no grafo residual de um fluxo máximo de s a t . O corte mínimo minimal (t, s) é equivalente. Esses cortes formam uma família de $O(n)$ conjuntos violados que contém todos os conjuntos ativos. Esses conjuntos ativos podem ser obtidos dessa família em tempo $O(n^2)$, encontrando os conjuntos minimais desta família. Isto pode ser feito mantendo-se, para cada vértice, o conjunto de menor cardinalidade que o contém. Sempre que uma iteração adiciona uma aresta $e = (u, v)$ ao conjunto F , cada corte mínimo minimal é atualizado. Por exemplo, considere o corte mínimo minimal (s, t) . Se u é alcançável a partir de s então v também é, e todos os vértices acessíveis a partir de v por arestas residuais. Se t se tornar alcançável, então desconsideramos a aresta (s, t) até o fim da fase. O tempo total em uma fase para atualizar um corte é o tempo de uma busca no grafo residual, e utiliza tempo $O(m')$. Assim, o tempo total para atualizar todas as arestas de T é $O(nm')$. Para encontrar o novo conjunto ativo, se existir, que resultar da inclusão de uma aresta (u, v) , é feita uma busca em todos os $O(n)$ candidatos a menor conjunto violado contendo u . Esse conjunto será ativo se não contiver nenhum outro conjunto ativo. Esse busca tem custo de tempo $O(n)$. Assim, o tempo total para o processo de adição de arestas em uma fase é $O(nm')$, o que nos dá um tempo total de $O(knm')$ para este processo durante todo o algoritmo.

4.2 Seleção de arestas

Uma das maiores diferenças em relação ao algoritmo anterior é o processo de seleção de arestas. A idéia deste procedimento é sempre manter uma solução viável y para o dual da relaxação linear de (IP_h) :

$$\begin{aligned} \max \quad & \sum_{S \subset V} h(S) y_S \\ \text{sujeito a} \quad & \sum_{S: e \in \delta(S)} y_S \leq c_e \quad e \in E_h \\ & y_S \geq 0 \quad S \subset V. \end{aligned}$$

O processo se inicia com $y = 0$. Em cada iteração, y_C , onde C é um conjunto ativo, é aumentado em um valor ϵ que seja o maior possível sem violar a desigualdade $\sum_{S: e \in \delta(S)} y_S \leq c_e$. Com isso, a desigualdade será satisfeita com igualdade para alguma aresta $e \in E_h$ na borda de um conjunto ativo. Essa aresta é acrescentada à solução.

Como no algoritmo anterior, não é mantida a solução dual y , mas sim uma variável $d(u) = \sum_{S: u \in S} y_S$ para cada $u \in V$. Assim, incrementar y_C para cada conjunto ativo C pelo maior valor possível ϵ , se torna equivalente a incrementar as variáveis $d(u)$ para $u \in C \in \mathcal{C}$, onde $\mathcal{C}$ é a coleção dos conjuntos ativos, pelo maior ϵ possível sem violar $d(u) + d(v) \leq c_{uv}$ para $e = (u, v) \in E_h$, e na fronteira de algum conjunto ativo. Portanto, sendo $a(u) = 1$ se u está em um conjunto ativo e $a(u) = 0$ caso contrário, temos

$$\epsilon = \min_{(i,j) \in E_h: a(i)+a(j)>0} \frac{c_{ij} - d(i) - d(j)}{a(i) + a(j)}.$$

As arestas acrescentadas são chamadas de 1-aresta, se elas têm uma ponta em um conjunto ativo e a outra fora de um conjunto ativo, e de 2-aresta se têm as duas pontas em conjuntos ativos.

A seleção da próxima aresta a ser inserida na solução utiliza uma estrutura de pacotes descrita em [6], e um lema que permite que algumas arestas sejam ignoradas.

É utilizada também a noção de “hora” para as arestas, sendo que a hora é zero no início da função Uncrossable, e é incrementada em ϵ a cada iteração. É mantido um *tempo de adição* em que uma aresta seria selecionada se os conjuntos ativos não mudassem. O tempo de adição de uma aresta (u, v) que liga b conjuntos ativos no tempo t é definido como $t(u, v) = t + \lfloor c_{uv} - d(u) - d(v) \rfloor / b$, sendo que $d(u)$ e $d(v)$ são os valores no tempo t e $b = 0, 1$ ou 2 ; o tempo de adição é ∞ se $b = 0$. O tempo de adição não muda a não ser que o conjunto ativo contendo u ou v mude. Assim, no tempo t , a aresta com menor tempo de adição é acrescentada a F .

É mantida uma partição dos vértices em a -conjuntos. Os a -conjuntos são definidos da seguinte maneira: Contraia todos os conjuntos ativos atuais e anteriores (anteriores na fase atual) em super-vértices. Agora temos como vértices esses super-vértices mais os vértices que nunca fizeram parte de conjuntos ativos. Um a -conjunto corresponde a uma árvore na floresta induzida pelas arestas de F nesses vértices. Todo vértice que nunca esteve em um conjunto ativo e não é incidente a uma aresta de F é um a -conjunto “singleton”. Em qualquer momento, todo conjunto ativo é um a -conjunto. Um a -conjunto que não é singleton nem um conjunto atualmente ativo é uma coleção maximal de conjuntos anteriormente ativos, mais possivelmente um singleton.

Figura 4.1: Exemplo de a -conjuntos. As linhas pontilhadas representam os a -conjuntos, enquanto as linhas cheias representam os conjuntos ativos anteriores

Notemos que entre as arestas ligando os mesmos dois a -conjuntos, podemos descartar todas menos a de menor tempo de adição. Seja $I \subseteq E$ o subgrafo de G com essas arestas. Assuma que na escolha da próxima aresta a adicionar, o processo de seleção quebra empates com uma numeração fixa das arestas. Portanto, qualquer conjunto de arestas está totalmente ordenado pelo tempo de adição, ou seja, a k -ésima menor aresta é única.

Seja $A(u)$ o a -conjunto contendo u , e seja a o número atual de a -conjuntos. O algoritmo é baseado no seguinte lema.

Lema 4.2.1 *Fixe uma iteração no loop while de Uncrossable. Considere uma aresta $(u, v) \in \delta_I(A(u))$ que não está entre as $2k$ menores arestas de $\delta_I(A(u))$. Então (u, v) não é adicionada a F até que $A(u)$ tenha mudado ou $A(v)$ tenha mudado ou a tenha diminuído de k .*

Prova. Suponha que (u, v) é adicionada a F numa iteração em que nem $A(u)$ nem $A(v)$ mudaram. Considere uma aresta (u', v') , uma das $2k$ menores arestas de $\delta_I(A(u))$. Quando (u, v) é adicionada a F , (u', v') não foi adicionada, pois $A(u)$ não mudou. Portanto o tempo de adição de (u', v') aumentou, implicando que $A(v')$ mudou e, portanto, os $2k$ conjuntos distintos $A(v')$ mudaram. Essas mudanças são resultado de unir vários a -conjuntos incluindo os $2k$ conjuntos $A(v')$. Portanto a deve ter diminuído de k . □

Graças a esse lema, podemos ignorar algumas arestas durante parte do loop while de Uncrossable. Para isso, o loop principal é dividido em subfases. Seja r um parâmetro a ser escolhido. Toda vez que a decresce por r ou mais desde o início da subfase atual, uma nova subfase começa. Vamos chamar algumas arestas de “acordadas” de tal modo que o lema 4.2.1 implica que uma aresta (u, v) que não está acordada nesta fase pode ser desconsiderada a não ser que $A(u)$ ou $A(v)$ mudem. No começo de uma subfase, escolhemos as $2r$ menores arestas na borda de todos os a -conjuntos. Toda aresta escolhida pelos a -conjuntos de suas duas pontas será inicialmente uma aresta acordada.

4.2.1 Estrutura de dados para a seleção de arestas

Para manter a lista de arestas acordadas, serão utilizadas filas de prioridade. A entrada de uma fila de prioridade é uma aresta acordada que liga dois a -conjuntos, e a prioridade dessa aresta é seu tempo de adição. As arestas acordadas incidentes a cada a -conjunto são particionadas em “pacotes” de no máximo $\log n$ arestas cada. Um dos pacotes é chamado de “pacote crescente” e os outros de “pacotes comuns”. Cada pacote é uma fila de prioridade. Cada aresta acordada ligando dois a -conjuntos está em dois pacotes, um para cada ponta. Além dos pacotes, é mantida uma outra fila de prioridade D . Uma aresta é colocada em D se for o mínimo de seus dois pacotes, e esses pacotes são comuns. Tais arestas são chamadas de “duplamente mínimas”. Isso implica que a aresta com menor tempo de adição é a aresta mínima de D ou a aresta mínima de um pacote crescente.

4.2.2 Funcionamento do algoritmo

No começo de cada subfase, são escolhidas as $2r$ menores arestas na borda de cada a -conjunto. Uma aresta que é escolhida nas suas duas pontas está acordada. Essas arestas são organizadas em pacotes, e as duplamente mínimas são colocadas em D . Para selecionar uma nova aresta e para F , é escolhida a menor aresta entre o mínimo de D e dos pacotes crescentes. Seja A o novo a -conjunto criado pela adição de e a F . As arestas incidentes aos vértices de A são excluídas dos seus pacotes e de D . Se essa remoção criar novas arestas duplamente mínimas, estas são acrescentadas a D . Com isso todos os pacotes correspondentes a A ficam vazios, então um novo pacote crescente é inicializado. Para escolher as arestas acordadas incidentes a A , examinamos todas as arestas incidentes a A . As arestas paralelas entre A e outros a -conjuntos são descartadas, mantendo sempre a menor. As $2r$ menores arestas incidentes a A que não foram

descartadas são marcadas como acordadas. Essas arestas são adicionadas aos pacotes crescentes das suas pontas (uma delas sendo o próprio A). Quando um pacote crescente chegar a $\log n$ arestas, ele se torna ordinário, um novo pacote crescente é criado, e uma entrada em D é criada se tivermos uma aresta duplamente mínima. O lema 4.2.1 justifica ignorarmos as arestas não acordadas.

4.2.3 Custo de tempo do algoritmo

Para qualquer a -conjunto A no máximo $3r$ arestas de $\delta(A)$ são acordadas. Note que no máximo $2r$ arestas são acordadas quando um a -conjunto é inicializado. Em cada iteração, no máximo uma aresta de $\delta(A)$ é acordada. Como temos no máximo r iterações, então no máximo mais r arestas são acrescentadas após a inicialização. Assim, qualquer a -conjunto tem no máximo $\frac{3r}{\log n}$ pacotes em qualquer ponto de uma subfase.

Pela observação acima, uma iteração que decrece o número a de a -conjuntos em j remove no máximo $O(jr)$ arestas de pacotes e no máximo $O(\frac{jr}{\log n})$ arestas de D . Assim, todos os passos de adição removem um total de $O(nr)$ arestas de pacotes e $O(\frac{nr}{\log n})$ de D , para um tempo total de $O(nr \log \log n + nr) = O(nr \log \log n)$ para todas remoções, lembrando que os pacotes têm no máximo $\log n$ arestas e D tem no máximo n arestas. O tempo para as inserções é igual, pois no máximo $O(nr)$ arestas entram em pacotes, e no máximo $O(\frac{nr}{\log n})$ arestas entram em D . Assim, o tempo total para inserir arestas deve ser também $O(nr \log \log n)$.

Por fim, temos no máximo $\frac{n}{r}$ subfases. Com a seleção em tempo linear, podemos montar os pacotes e D com tempo adicional $O(m)$ no início de qualquer subfase. É necessário tempo amortizado $O(n)$ sempre que uma aresta é selecionada para o descarte de arestas paralelas, examinando as arestas incidentes a A e atualizando $A(u)$. Portanto o tempo total é $O(n^2 + \frac{nm}{r} + nr \log \log n)$. Escolhendo $r = \sqrt{\frac{m}{\log \log n}}$, temos um tempo total de $O(n(n + \sqrt{m \log \log n}))$ para a seleção de arestas.

4.3 Processo de limpeza

Para descrever o novo método de limpeza, precisamos de algumas definições preliminares. Vamos começar supondo que a função não-cruzável h considerada é tal que nunca são criados novos conjuntos ativos durante uma fase. Assim que uma aresta na fronteira de algum conjunto ativo C é selecionada, então C não é mais ativo. Neste caso, haverá no máximo $|C|$ iterações no passo de adição de arestas.

Seja F o conjunto de arestas adicionadas. F é uma floresta em um grafo com cada conjunto ativo contraído em um nó, e mais alguns nós. Essa floresta “cresce” quando novas arestas são acrescentadas a F . Um novo nó em uma árvore já existente é criado quando é acrescentada uma 1-aresta que tem um ponta em um conjunto ativo, que será o novo nó, e a outra ponta em um conjunto que já foi ativo, ou seja, um nó já existente. Se acrescentarmos uma 1-aresta que tem uma ponta em um conjunto ativo e a outra não está em um conjunto que já foi ou ainda é ativo, é criada uma nova árvore. O mesmo ocorre com uma 2-aresta, que tem as duas pontas em conjuntos ativos.

As arestas que interessam no processo de limpeza são as 1-aresta e tais que as arestas acrescentadas após e formam uma árvore geradora exatamente dos conjuntos que estavam ativos

imediatamente antes da adição de e . Essas arestas são chamadas *arestas especiais*.

O conceito de arestas especiais pode ser generalizado para o caso em que novos conjuntos ativos são criados. Para isso, as arestas e os conjuntos ativos são particionados. Seja $\mathcal{UC}$ a coleção de todos os conjuntos ativos criados em todas as iterações do algoritmo, mais V . De acordo com o corolário 4.0.6 $\mathcal{UC}$ é laminar. Definimos uma árvore T baseada em $\mathcal{UC}$, com um vértice v_C para cada $C \in \mathcal{UC}$. Então fazemos v_C ser pai de v_D na árvore T se C é o menor conjunto em $\mathcal{UC}$ que propriamente contém D . Seja $\mathcal{D}(C)$ a coleção de conjuntos correspondentes aos filhos de v_C em T . A coleção $\mathcal{D}(C)$ pode ser pensada como uma classe de equivalência de conjuntos ativos. Para cada aresta e , seja $C(e)$ o menor conjunto $C \in \mathcal{UC}$ que contém ambas as pontas de e . O conjunto de todas as arestas de F para os quais $C(e) = C$ é denotado por F_C . As arestas em F_C podem ser pensadas como uma classe de equivalência de F . O comportamento das arestas de F_C nos conjuntos ativos em $\mathcal{D}(C)$ será como no caso acima em que nenhum novo conjunto ativo é criado. Seja $\mathcal{A}(e)$ a coleção dos conjuntos em $\mathcal{D}(C(e))$ que estão ativos logo antes de e ser selecionada.

Observação 4.3.1 *Para qualquer vértice v_C em T , todas as arestas em F_C devem ser selecionadas antes que uma aresta em $\delta(C)$ seja selecionada.*

Prova. Suponha que a afirmação seja falsa, e que em alguma interação uma aresta em $\delta(C)$ é selecionada antes de alguma aresta em F_C . Como nem todas as arestas em F_C foram selecionadas, algum $C' \subset C$ deve estar ativo. Mas como uma aresta em $\delta(C)$ foi selecionada, C nunca se tornará um conjunto ativo, uma contradição. $\square$

Digamos que um conjunto de arestas H forma uma árvore geradora em uma família de conjuntos de vértices disjuntos $\{C_i\}_{i=1}^k$ se cada ponta de cada aresta de H está em algum C_i e H forma uma árvore geradora no grafo com cada C_i contraído a um nó. Então e é especial se é uma 1-aresta e se as arestas adicionadas a $F_{C(e)}$ depois de e formam uma árvore geradora nos conjuntos em $\mathcal{A}(e)$.

O processo de limpeza consiste em olhar as arestas de F na ordem inversa em que foram selecionadas. Seja $A(e) = \bigcup_{C \in \mathcal{A}(e)} C$. O processo de limpeza remove a aresta e de F se e for especial, nenhuma outra aresta de $F_{C(e)}$ já tenha sido removida, e todas as arestas restantes de F em $\delta(C(e))$ também estejam em $\delta(A(e))$. A aresta e não é removida se $C(e) = V$.

4.3.1 Implementação do processo de remoção

Para a implementação, serão utilizadas uma árvore $\bar{T}$ e arborescências τ_C . A árvore $\bar{T}$ é uma modificação da árvore T . Para montar $\bar{T}$, são criados nós filhos adicionais para cada nó interno v_C da árvore representando o conjunto de vértices em C que não estão em outros filhos de v_C .

Para cada nó v_C em $\bar{T}$ é construída uma arborescência τ_C . Um “branching” é uma floresta direcionada na qual todo nó tem grau de entrada no máximo 1, e uma arborescência é um “branching” conectado. Com as arestas de F_C , é formado um “branching” β_C nos nós correspondentes aos filhos de C em $\bar{T}$. Um arco (v_A, v_B) é criada para cada aresta $e \in F_C$ quando e está na borda dos conjuntos A e B . Para as 2-arestas, a direção do arco pode ser qualquer uma, enquanto para as 1-arestas, o arco é direcionado para o conjunto ativo que o define. Esses arcos formam um “branching” pois duas arestas de F não podem ser direcionadas para o mesmo conjunto ativo.

Ele pode ser convertido a uma arborescência τ_C se adicionarmos um nó conectado a todas as raízes do “branching”. Tanto a árvore $\bar{T}$ quanto as arborescências τ_C podem ser facilmente construídas no passo de adição de arestas.

Observação 4.3.2 *Uma 1-aresta e em F_C (com $C = C(e)$) é especial se e somente se todas as arestas de F_C adicionadas a τ_C depois de e formam uma arborescência com a cabeça de e como raiz.*

Prova. A prova segue da definição de aresta especial. Uma 1-aresta e é especial se e somente se todas as arestas de F_C adicionadas depois de e formarem uma árvore geradora nos conjuntos ativos correspondentes aos filhos de v_C , e essa árvore geradora faz com que τ_C seja uma arborescência direcionada da cabeça de e já que sempre que uma 1-aresta é direcionada para um nó correspondente a um conjunto ativo, todas as arestas adicionadas após ela devem ser direcionadas para fora desse nó. □

Para encontrar as arestas especiais em tempo $O(n)$, cada conjunto ativo $C \in UC$ é considerado na sua vez. Suponha que F_C contenha as arestas $e_1, \dots, e_t$, onde e_i foi adicionada antes de e_j se $i < j$. Seja l_i o primeiro ancestral comum das cabeças de $e_i, \dots, e_t$ em τ_C . Assim, uma 1-aresta e_i é especial se e somente se l_i é a cabeça da aresta e_i em τ_C . Todos os l_i podem ser encontrados em tempo linear se as arestas forem processadas em ordem reversa, marcando os nós ano caminho até o ancestral e parando no primeiro nó já marcado.

O processo de limpeza é implementado com uma busca de cima para baixo na árvore $\bar{T}$. Seja F'' o conjunto de arestas o conjunto de arestas que sobraram após um processo de limpeza que funciona com uma busca de cima para baixo. Em cada nó v_C de $\bar{T}$, as arestas de F_C são processadas em ordem reversa à que elas aparecem em F , detectando e possivelmente removendo uma aresta especial de F_C , de acordo com as condições da seção anterior.

Lema 4.3.3 $F'' = F'$

Prova. A remoção de uma aresta e de F_C depende apenas das arestas em F_C e $\delta_F(C)$, e afeta somente a remoção de arestas em F_C e F_D , para nós D com $e \in \delta(D)$. Tal conjunto D deve ser um descendente de v_C em $\bar{T}$, e qualquer nó v_A tal que todas as arestas de $\delta_F(C)$ estão em F_A deve ser um ancestral de C . Pela observação 4.3.1, todas as arestas de F_C ocorrem em F antes de qualquer aresta em $\delta(C)$ e após qualquer aresta em F_D , sendo v_D um descendente de v_C . Portanto o conjunto de arestas F'' será o mesmo que F' . □

Chamamos um filho v_B de v_C *marcado* se existe uma arestas já processada mas ainda não removida que está nas bordas de B e C .

Lema 4.3.4 *Uma aresta especial $e \in F_C$ é removida na busca de cima para baixo em $\bar{T}$ se e somente se ela for a aresta especial de maior profundidade em τ_C cuja cabeça é um ancestral em τ_C de todos os filhos marcados de v_C .*

Prova. Lembrando que, ao considerarmos as arestas de F_C em ordem reversa à ordem em que foram adicionadas, uma aresta especial e é removida de F_C se e somente se nenhuma outra aresta especial tenha sido removida e todas as arestas restantes em $\delta(C)$ também estão em

$\delta(A(e))$, onde $A(e)$ é a união dos conjuntos em $\mathcal{A}(e)$. Para qualquer aresta especial e de F_C , os conjuntos em $\mathcal{A}(e)$ correspondem exatamente aos descendentes da cabeça da aresta e em τ_C . Assim todas as arestas restantes em $\delta(C)$ também estão em $\delta(A(e))$ para a aresta especial e se e somente se a cabeça de e for um ancestral em τ_C de todos os filhos marcados de v_C . Como todas as arestas em F_C adicionadas depois de uma aresta especial e devem ser arestas em τ_C entre os descendentes da cabeça de e , a profundidade das arestas especiais em τ_C é estritamente crescente com relação à ordem em que as arestas especiais foram escolhidas para a solução. Portanto o aresta especial em τ_C cuja cabeça é um ancestral em τ_C de todas os filhos marcados de v_C é exatamente a aresta que será removida. $\square$

O procedimento funciona da forma descrita a seguir. É assumido que quando o nó v_C for visitado, todos os filhos de v_C que devem foram corretamente marcados. Assumindo isso, o lema acima permite que uma aresta especial que será removida possa ser detectada enquanto as arestas especiais de F_C são determinadas. Assim que for decidido qual aresta especial de F_C deve ser removida (se houver alguma) e quais arestas manter, para cada aresta $e = (u, v) \in F_C$ que é mantida, os nós nos caminhos em $\bar{T}$ das folhas u e v até o seu ancestral comum v_C (sem incluir o próprio v_C) são marcados. Este procedimento pode marcar erradamente alguns filhos de v_C , mas isso não importa, pois as arestas de F_C já foram processadas. Para que este processo de marcação de nós tenha tempo linear, a marcação de um caminho para antes de alcançar v_C se for encontrado um nó já marcado. Essa estratégia é válida pois a busca é feita de cima para baixo em $\bar{T}$, e se um nó v_D no caminho para v_C está marcada, então também estão todos os ancestrais de v_D até v_C . Essa busca de cima para baixo também garante que quando um nó v_D é alcançado, todos os seus filhos que devem ser marcados estão marcados. Assim, o tempo de execução de todo o processo de limpeza é $O(n)$.

4.4 Custo total do algoritmo

Somando o custo da escolha dos conjuntos ativos, $O(nm') = O(kn^2)$ ao custo do processo de seleção e adição de arestas, $O(n(n + \sqrt{m \log \log n}))$, com $m = O(n^2)$, ao custo do processo de limpeza, $O(n)$, temos que o custo total do algoritmo Uncrossable, isto é, o custo para uma fase, é $O(n(kn + n\sqrt{\log \log n}))$. Multiplicando pelas k fases, temos um custo total do algoritmo de $O(kn(kn + n\sqrt{\log \log n})) = O(k^2n^2 + kn\sqrt{\log \log n})$.

4.5 Exemplo de funcionamento do algoritmo

A seguir, um exemplo de como o algoritmo funciona, primeiro o processo de escolha de arestas, e por fim o processo de limpeza.

Na figura 4.2 vemos um exemplo de inserção de arestas. Nessa figura, as caixas correspondem aos conjuntos ativos, e as linhas são as arestas de F . Apenas as arestas e_4 e e_8 são 2-arestas. Todas as outras são 1-arestas. Vamos analisá-las uma a uma.

- e_1 : $C(e_1)$ é o conjunto da esquerda no quadro 7. Assim, $\mathcal{A}(e_1)$ corresponde a todos os conjuntos ativos do quadro 1 que estão “dentro” de $C(e_1)$. Note que $F_{C(e_1)} = \{e_1, e_3, e_5, e_6\}$,

Figura 4.2: Um exemplo do processo de inserção de arestas

sendo que e_3, e_5 e e_6 foram adicionadas após e_1 , e formam uma árvore geradora nos conjuntos em $\mathcal{A}(e_1)$. Portanto, e_1 é especial.

- e_2 : $C(e_2)$ é o conjunto da direita no quadro 5. Assim, $\mathcal{A}(e_2)$ corresponde a todos os conjuntos ativos do quadro 2 que estão “dentro” de $C(e_2)$. Note que $F_{C(e_2)} = \{e_2, e_4\}$, sendo que e_4 foi adicionada após e_2 , e não forma uma árvore geradora nos conjuntos em $\mathcal{A}(e_2)$. Portanto, e_2 não é especial.
- e_3 : $C(e_3)$ é o conjunto da esquerda no quadro 7. Assim, $\mathcal{A}(e_3)$ corresponde a todos os conjuntos ativos do quadro 3 que estão “dentro” de $C(e_3)$. Note que $F_{C(e_3)} = \{e_1, e_3, e_5, e_6\}$, sendo que e_5 e e_6 foram adicionadas após e_3 , e não formam uma árvore geradora nos conjuntos em $\mathcal{A}(e_3)$. Portanto, e_3 não é especial.
- e_5 : $C(e_5)$ é o conjunto da esquerda no quadro 7. Assim, $\mathcal{A}(e_5)$ corresponde a todos os conjuntos ativos do quadro 5 que estão “dentro” de $C(e_5)$. Note que $F_{C(e_5)} = \{e_1, e_3, e_5, e_6\}$, sendo que e_6 foi adicionada após e_1 , e forma uma árvore geradora nos conjuntos em $\mathcal{A}(e_5)$. Portanto, e_5 é especial.

- e_6 : $C(e_6)$ é o conjunto da esquerda no quadro 7. Assim, $\mathcal{A}(e_6)$ corresponde a todos os conjuntos ativos do quadro 6 que estão “dentro” de $C(e_6)$. Note que $F_{C(e_6)} = \{e_1, e_3, e_5, e_6\}$. Como nenhuma delas é adicionada após e_6 , e $\mathcal{A}(e_6)$ tem apenas um conjunto, temos uma árvore geradora. Portanto, e_6 é especial.
- e_7 : $C(e_7)$ é o conjunto da esquerda no quadro 8. Assim, $\mathcal{A}(e_7)$ corresponde a todos os conjuntos ativos do quadro 7 que estão “dentro” de $C(e_7)$. Note que $F_{C(e_7)} = \{e_1, e_3, e_5, e_6, e_7\}$. Como nenhuma delas é adicionada após e_7 , e $\mathcal{A}(e_7)$ tem apenas um conjunto, temos uma árvore geradora. Portanto, e_7 é especial.

Figura 4.3: Um exemplo do processo de limpeza

Na figura 4.3, é mostrado o processo de limpeza, após a seleção de arestas mostrada na figura 4.2. Como o processo de limpeza trabalha com as arestas especiais, na ordem inversa em que foram inseridas, a possível remoção se dá nesta ordem: e_7, e_6, e_5, e_1 . Os quadros de 1 a 4 correspondem à situação logo antes da possível remoção de e_7, e_6, e_5, e_1 , respectivamente.

No quadro 1, a caixa mostra $C(e_7)$, e os vértices brancos representam os vértices em $\mathcal{A}(e_7)$. Do mesmo modo estão representados $C(e_6)$ e $\mathcal{A}(e_6)$ no quadro 2, $C(e_5)$ e $\mathcal{A}(e_5)$ no quadro 3 e $C(e_1)$ e $\mathcal{A}(e_1)$ no quadro 4.

A aresta e_7 é removida, pois nenhuma outra aresta de $F_{C(e_7)}$ já foi removida, e e_8 , a única aresta em $\delta(C(e_7))$ também está em $\delta(\mathcal{A}(e_7))$.

Já a aresta e_6 não é removida, pois $e_8 \in \delta(C(e_6))$, mas $e_8 \notin \delta(\mathcal{A}(e_6))$.

A aresta e_5 é removida, pois nenhuma outra aresta de $F_{C(e_5)}$ já foi removida, e e_8 , a única aresta em $\delta(C(e_5))$ também está em $\delta(\mathcal{A}(e_5))$.

A aresta e_1 não é removida, pois $e_5 \in F_{C(e_1)}$ já foi removida.

A figura 4.4 mostra o resultado do algoritmo.

Figura 4.4: O resultado final

Capítulo 5

Uma variante do problema

Vamos agora apresentar uma variante do problema. Até agora, sempre procuramos caminhos que fossem disjuntos nas arestas. Nesta variante será exigido que esses caminhos sejam disjuntos nas arestas e num subconjunto dos vértices. Esse problema é conhecido como *element connectivity survivable network design problem* (ELC-SNDP) [13]. Nesta versão, os vértices são separados em dois tipos, *terminais* e *vértices de Steiner*. Os vértices de Steiner, assim como as arestas, são os que podem falhar, e portanto os caminhos devem ser disjuntos neles; esses são os *elementos*. Para cada par de terminais, u e v , temos um requerimento de conectividade r_{uv} , e o problema consiste em encontrar um subgrafo de custo mínimo tal que, para cada par de terminais, u e v , mesmo com a falha de $r_{uv} - 1$ elementos, ainda exista um caminho entre u e v .

Um exemplo poderia ser a ligação elétrica de uma cidade, na qual alguns pontos importantes como hospitais devem ser tolerantes a falhas (seriam os terminais), enquanto residências podem sofrer falhas (vértices de Steiner).

O problema é claramente $\mathcal{NP}$ -difícil, pois se o conjunto de vértices de Steiner for vazio temos exatamente o problema dos capítulos anteriores.

O algoritmo que obtém uma solução aproximada para este problema deriva dos algoritmos vistos anteriormente. Uma outra forma de abordar o problema é utilizar um método de arredondamento das soluções para a relaxação linear do problema, abordado por Fleischer, Jain e Williamson [5].

Vamos dar uma definição mais formal do problema.

Seja $G = (V, E)$ um grafo não-direcionado com custos não negativos c_e para toda aresta $e \in E$. O conjunto de vértices V é particionado em dois conjuntos disjuntos S e R . S é o conjunto de vértices de Steiner, e R é o conjunto de terminais. Para quaisquer dois vértices $u, v \in R$ temos também um requerimento de conectividade r_{uv} , ou seja, mesmo que falhem $r_{uv} - 1$ elementos de $S \cup E$, ainda existe pelo menos um caminho ligando u a v .

O problema pode ser modelado pelo seguinte programa inteiro:

$$\begin{array}{ll} \min & \sum_{e \in E} c_e x_e \\ \text{sujeito a} & \sum_{e \in \delta_{G \setminus A}(B)} x_e \geq f(S) - |A| \quad \forall A \subseteq S \cup E, B \subseteq V \setminus A \\ & x_e \in \{0, 1\} \quad e \in E, \end{array}$$

onde $\delta_{G \setminus A}(B)$ representa as arestas de $G \setminus A$ com uma ponta em B e outra fora de B e $f(S) = \max_{i \in S, j \notin S} r_{ij}$.

A relaxação linear é obtida relaxando-se a condição $x_e \in \{0, 1\}$ para $x_e \geq 0$. Assim, o dual da relaxação linear passa a ser o seguinte:

$$\begin{aligned} \max \quad & \sum_{A \subseteq S \cup E, B \subseteq V \setminus A} (f(B) - |A|) y_{BA} \\ \text{sujeito a} \quad & \sum_{BA: A \subseteq S \cup E, B \subseteq V \setminus A, e \in \delta_{G \setminus A}(B)} y_{BA} \leq c_e \quad \forall e \in E \\ & y_{BA} \geq 0 \quad \forall A \subseteq S \cup E, B \subseteq V \setminus A \end{aligned}$$

Seja OPT o valor da solução ótima do programa inteiro, ou seja, a solução ótima do problema. Pelo teorema fraco da dualidade, qualquer solução deste dual é um limitante inferior para OPT.

5.1 Descrição do algoritmo

A descrição em alto nível do algoritmo é semelhante à dos algoritmos anteriores. Seja H uma solução parcial para o problema. Uma solução parcial é o conjunto de arestas já selecionadas para fazer parte da solução. A deficiência de uma restrição é definida como a diferença entre o lado direito e o lado esquerdo. Somente restrições não satisfeitas têm deficiências positivas. A deficiência de um conjunto $B \subseteq V$ é definida como a maior deficiência de uma restrição na qual B está envolvido.

Assim como nos algoritmos anteriores, este também está dividido em k fases. Se numerarmos as fases de k a 1, no início da fase p a deficiência de um conjunto pode ser no máximo p , e ao final da fase, pode ser no máximo $p - 1$. Assim, ao final da fase 1 temos uma solução viável.

Seja H a solução parcial no início da fase p . A função h é definida como $h(B) = 1$, se a deficiência de B é igual a p , e $h(B) = 0$, caso contrário.

Seja $\Gamma_H(S)$ o conjunto dos vértices de Steiner que não estão em S mas têm um vizinho em S , considerando as arestas H . Seja $\delta_H(S)$ o conjunto de arestas de H que têm uma ponta em S e outra em $R \setminus S$, onde R é o conjunto de terminais. Por fim, a vizinhança de elementos do conjunto S com relação ao grafo H como $\varepsilon_H(S) = \Gamma_H(S) \cup \delta_H(S)$.

O programa inteiro (IP_h) que é resolvido na fase p é o seguinte:

$$\begin{aligned} \min \quad & \sum_{e \in E_h} c_e x_e \\ \text{sujeito a} \quad & \sum_{e \in \delta_{G \setminus \varepsilon_H(S)}} x_e \geq h(S) \quad S \subseteq V \\ & x_e \in \{0, 1\} \quad e \in E_h \end{aligned}$$

E o dual da relaxação linear desse programa inteiro é o seguinte:

$$\begin{aligned} \max \quad & \sum_{S \subseteq V} h(S) y_S \\ \text{sujeito a} \quad & \sum_{S: e \in \delta_{G \setminus \varepsilon_H(S)}} y_S \leq c_e \quad \forall e \in E_h \\ & y_S \geq 0 \quad \forall S \subseteq V \end{aligned}$$

5.1.1 O algoritmo para uma fase

Em cada fase, é executado o algoritmo abaixo. Note que ele é muito semelhante aos algoritmos dos capítulos anteriores, com pequenas adaptações.

```

01    $F \leftarrow \emptyset$ 
02    $i \leftarrow 0$ 
03    $d(v) \leftarrow 0$  para todo  $v \in V$ 
04   Comentário: Implicitamente fazendo  $y_S \leftarrow 0$  para todo  $S \subset V$ 
05   Enquanto houver conjuntos ativos
06      $i \leftarrow i + 1$ 
07     Para todo vértice  $v$  de um conjunto ativo, incremente  $d(v)$  uniformemente de  $\varepsilon$ 
    até que alguma aresta  $e_i = (u, w) \in E_h - (\varepsilon_H(S) \cup F)$  satisfaça  $d(u) + d(w) = c_{uw}$  para  $e_i \in \delta(C)$ 
    de algum conjunto ativo  $C$ 
08     Comentário: Implicitamente fazendo  $y_C \leftarrow y_C + \varepsilon$ , para todo conjunto ativo  $C$ 
09      $F \leftarrow F \cup \{e_i\}$ 
10     Atualize os conjuntos ativos
11   Comentário: Passo de limpeza
12    $F' \leftarrow F$ 
13   Para  $j \leftarrow i$  até 1
14     Se  $F' - \{e_j\}$  satisfaz  $h$ 
15        $F' \leftarrow F' - \{e_j\}$ 
16   Devolva  $F'$ 

```

Lema 5.1.1 *Seja $H_k = \frac{1}{k} + \frac{1}{k-1} + \dots + 1$. O algoritmo é uma $2H_k$ -aproximação para o problema.*

Para provar este lema, precisamos de algumas definições, dadas a seguir. Após essas definições, teremos a prova deste lema.

A partir deste ponto, H é uma solução parcial obtida ao final da fase $p + 1$, e i uma iteração do loop while do algoritmo, que será chamada de iteração corrente. Seja I qualquer solução para a fase p . Com isso, algumas definições são dadas a seguir.

Dois conjuntos $A, B \subseteq V$ se cruzam em relação a $\varepsilon_{H \cup I}$ se $A \not\subseteq B$, $B \not\subseteq A$, $A \cap (B \cup \varepsilon_{H \cup I}(B)) \neq \emptyset$ e $B \cap (A \cup \varepsilon_{H \cup I}(A)) \neq \emptyset$.

Uma família de subconjuntos de V é laminar em relação a $\varepsilon_{H \cup I}$ se nenhum par de conjuntos dessa família se cruzarem em relação a $\varepsilon_{H \cup I}$.

Essa definição é mais forte que a definição usual de família laminar, dada no capítulo 3, pela qual uma família de conjuntos é laminar se para quaisquer conjuntos A e B da família ou $A \subseteq B$ ou $B \subseteq A$ ou $A \cap B = \emptyset$.

Sejam F o conjunto de arestas devolvido pelo algoritmo na fase p , $\mathcal{C}$ a coleção de conjuntos ativos na iteração corrente, e Y o conjunto de arestas $e \in F$ para as quais existe um $C \in \mathcal{C}$ tal que $e \in \delta_{G \setminus \varepsilon_H(B)}(B)$. Para cada aresta $e \in Y$, é definido um *conjunto testemunha* $W_e \subseteq V$, que segue as seguintes condições:

- $|\varepsilon_{H \cup I \cup F}(W_e)| = f(W_e) - p + 1$;
- $|\varepsilon_{H \cup I \cup F \setminus \{e\}}(W_e)| = f(W_e) - p$.

Cada aresta $e \in Y$ deve ter um conjunto testemunha, pois pela construção do algoritmo, $I \cup F \setminus \{e\}$ não é uma solução viável, e portanto deve existir algum conjunto violado W_e que é um conjunto testemunha. Uma *família testemunha* para Y é uma família de subconjuntos de V que contém exatamente um conjunto testemunha para cada aresta em Y .

Há três condições que devem ser mantidas durante qualquer iteração:

- Nenhum conjunto ativo em relação a I cruza (com relação a $\varepsilon_{H \cup I}$) nenhum outro conjunto ativo em relação a I .
- Os conjuntos ativos em relação a I podem ser calculados em tempo polinomial.
- Existe uma família testemunha para Y .

Para provar que estas condições são válidas, precisamos de algumas definições e teoremas.

Seja $\varphi : 2^V \rightarrow \mathbf{Z}^+$. φ é dita $\varepsilon_{H \cup I}$ -submodular se $\varphi(V) = 0$ e, para todo $A, B \subseteq V$, as seguintes condições valem:

- $\varphi(A) + \varphi(B) \geq \varphi(A \cup B) + \varphi(A \cap B)$
- $\varphi(A) + \varphi(B) \geq \varphi(A \setminus B \setminus \varepsilon_{H \cup I}(B)) + \varphi(B \setminus A \setminus \varepsilon_{H \cup I}(A))$

Lema 5.1.2 $|\varepsilon_{H \cup I}|$ é $\varepsilon_{H \cup I}$ -submodular.

Prova. É preciso provar que as duas desigualdades que definem $\varepsilon_{H \cup I}$ -submodularidade são válidas. A primeira vale pois a contribuição de qualquer elemento para o lado esquerdo da desigualdade é pelo menos a mesma contribuição desse elemento para o lado direito. Para a segunda desigualdade a prova é semelhante, exceto quando existir uma aresta uv , $u \in R$, $v \in S \cap A \cap B$ e ou $u \in A \setminus B \setminus \varepsilon_{H \cup I}(B)$ ou $u \in B \setminus A \setminus \varepsilon_{H \cup I}(A)$. Neste caso, v contribui somente para o lado direito da desigualdade. Mas, para compensar, a aresta uv contribui apenas para o lado esquerdo. □

Seja $\varphi : 2^V \rightarrow \mathbf{Z}^+$. φ é dita fracamente $\varepsilon_{H \cup I}$ -supermodular se $\varphi(V) = 0$ e, para todo $A, B \subseteq V$, pelo menos uma das seguintes condições vale:

- $\varphi(A) + \varphi(B) \leq \varphi(A \cup B) + \varphi(A \cap B)$
- $\varphi(A) + \varphi(B) \leq \varphi(A \setminus B \setminus \varepsilon_{H \cup I}(B)) + \varphi(B \setminus A \setminus \varepsilon_{H \cup I}(A))$

Lema 5.1.3 $f(B) = \max_{i \in B, j \notin B} r_{ij}$ é fracamente $\varepsilon_{H \cup I}$ -supermodular.

Prova. Como $\varepsilon_{H \cup I}(B)$ não contém um vértice requerido $f(A \setminus B \setminus \varepsilon_{H \cup I}(B)) = f(A \setminus B)$ e $f(B \setminus A \setminus \varepsilon_{H \cup I}(A)) = f(B \setminus A)$. Isto reduz o problema a mostrar que f é fracamente supermodular. Como f é uma função própria, então valem as seguintes desigualdades:

- $\max\{f(A \setminus B), f(A \cap B)\} \geq f(A)$
- $\max\{f(B \setminus A), f(A \cup B)\} = \max\{f(B \setminus A), f(V \setminus (A \cup B))\} \geq f(V \setminus A) = f(A)$
- $\max\{f(B \setminus A), f(A \cap B)\} \geq f(B)$

- $\max\{f(A \setminus B), f(A \cup B)\} = \max\{f(A \setminus B), f(V \setminus (A \cup B))\} \geq f(V \setminus B) = f(B)$

Se o mínimo entre $f(A \setminus B)$, $f(A \cap B)$, $f(B \setminus A)$ e $f(A \cup B)$ for $f(A \setminus B)$, somamos a primeira e a quarta desigualdades, obtendo $f(A) + f(B) \leq \max\{f(A \setminus B), f(A \cap B)\} + \max\{f(A \setminus B), f(A \cup B)\} = f(A \cap B) + f(A \cup B)$. De forma análoga, se algum dos outros conjuntos for o mínimo, somamos duas das desigualdades, sempre obtendo uma das condições desejadas. $\square$

Teorema 5.1.4 *Se $A, B \subseteq V$ são violados com relação a I , então ou $A \cap B$ e $A \cup B$ também são violados, ou $A \setminus B \setminus \varepsilon_{H \cup I}(B)$ e $B \setminus A \setminus \varepsilon_{H \cup I}(A)$ também são violados.*

Prova. Como A e B são conjuntos violados, suas deficiências são iguais a p , que é a deficiência máxima nesta fase. Assim:

$$\begin{aligned} 2p &= (f(A) - |\varepsilon_{H \cup I}(A)|) + (f(B) - |\varepsilon_{H \cup I}(B)|) \\ &= (f(A) + f(B)) - (|\varepsilon_{H \cup I}(A)| + |\varepsilon_{H \cup I}(B)|) \end{aligned}$$

Como F é fracamente $\varepsilon_{H \cup I}$ -supermodular, ou $f(A) - f(B) \leq f(A \cap B) + f(A \cup B)$ ou $f(A) - f(B) \leq f(A \setminus B \setminus \varepsilon_{H \cup I}(B)) - f(B \setminus A \setminus \varepsilon_{H \cup I}(A))$. Suponha que a primeira desigualdade é válida. Por $\varepsilon_{H \cup I}$ -submodularidade, temos que $|\varepsilon_{H \cup I}(A)| + |\varepsilon_{H \cup I}(B)| \geq |\varepsilon_{H \cup I}(A \cap B)| + |\varepsilon_{H \cup I}(A \cup B)|$. Assim,

$$\begin{aligned} 2p &\leq (f(A \cap B) + f(A \cup B)) - (|\varepsilon_{H \cup I}(A \cap B)| + |\varepsilon_{H \cup I}(A \cup B)|) \\ &= (f(A \cap B) - |\varepsilon_{H \cup I}(A \cap B)|) + (f(A \cup B) - |\varepsilon_{H \cup I}(A \cup B)|) \\ &\leq 2p. \end{aligned}$$

A última desigualdade resulta do fato de nenhum conjunto ter deficiência maior que p . Como $(f(A \cap B) - |\varepsilon_{H \cup I}(A \cap B)|) + (f(A \cup B) - |\varepsilon_{H \cup I}(A \cup B)|)$ é no máximo e no mínimo $2p$, então é exatamente $2p$. E isso só é possível se $A \cap B$ e $A \cup B$ forem violados.

Da mesma forma, se $f(A) - f(B) \leq f(A \setminus B \setminus \varepsilon_{H \cup I}(B)) - f(B \setminus A \setminus \varepsilon_{H \cup I}(A))$, então $A \setminus B \setminus \varepsilon_{H \cup I}(B)$ e $B \setminus A \setminus \varepsilon_{H \cup I}(A)$ são violados. $\square$

O teorema implica o seguinte corolário, que prova a primeira condição.

Corolário 5.1.5 *Nenhum conjunto violado cruza com relação a $\varepsilon_{H \cup I}$ algum conjunto ativo.*

Prova. Se um conjunto violado A cruzar um conjunto ativo B , então, pelo teorema, ou $A \cap B$ ou $B \setminus A \setminus \varepsilon_{H \cup I}(A)$ é violado também, o que contradiz a minimalidade de B . $\square$

Desse corolário pode-se concluir que os conjuntos ativos são disjuntos e, portanto, cada vértice está em no máximo um conjunto ativo. Considere uma rede em $H \cup I$, onde as capacidades de cada aresta e de cada vértice de Steiner é 1 e a capacidade de cada terminal é infinita. Um vértice u estará em um conjunto ativo se existir um vértice v tal que o corte mínimo $u - v$ tem capacidade $r_{uv} - p$. O conjunto ativo no qual u está é o corte mínimo minimal (sob inclusão) $u - v$. Tal corte pode ser encontrado em tempo polinomial utilizando uma rotina de fluxo máximo. Isto prova a segunda condição.

Agora falta mostrar a terceira condição.

Teorema 5.1.6 *Há uma família testemunha para Y*

Prova. Dada uma família testemunha, suponha que dois conjuntos W_e e W_f cruzam, e cruzam em relação a $\varepsilon_{H \cup I \cup F}$. Sejam X e Y dois conjuntos obtidos do teorema 5.1.4. W_e e W_f serão substituídos por conjuntos X_e e Y_f , e esses conjuntos serão as testemunhas para as arestas e e f . Quando o primeiro caso do teorema 5.1.4 é válido, $X_e = X$ e $Y_f = Y$. Quando o segundo caso é válido, $X_e = W_e \setminus W_f$ e $Y_f = W_f \setminus W_e$. É preciso mostrar que X_e e Y_f são testemunhas para e e f .

Como W_e e W_f são testemunhas, temos que:

$$|\varepsilon_{H \cup I \cup F}(W_e)| + |\varepsilon_{H \cup I \cup F}(W_f)| = f(W_e) + f(W_f) - 2p + 2.$$

Pela fraca $\varepsilon_{H \cup I \cup F}$ -supermodularidade de f e pela $\varepsilon_{H \cup I \cup F}$ -submodularidade de $|\varepsilon_{H \cup I \cup F}|$, temos que

$$|\varepsilon_{H \cup I \cup F}(X)| - |\varepsilon_{H \cup I \cup F}(Y)| = f(X) + f(Y) - 2p + 2.$$

Como $F \cup I$ é uma solução viável para esta fase, a equação acima implica que $|\varepsilon_{H \cup I \cup F}(X)| = f(X) - p + 1$ e $|\varepsilon_{H \cup I \cup F}(Y)| = f(Y) - p + 1$.

Considere a aresta e . Aplicando a definição de testemunha para e temos que

$$|\varepsilon_{H \cup I \cup F \setminus \{e\}}(W_e)| + |\varepsilon_{H \cup I \cup F \setminus \{e\}}(W_f)| \leq f(W_e) + f(W_f) - 2p + 2.$$

Pela fraca $\varepsilon_{H \cup I \cup F \setminus \{e\}}$ -supermodularidade de f e pela $\varepsilon_{H \cup I \cup F \setminus \{e\}}$ -submodularidade de $|\varepsilon_{H \cup I \cup F \setminus \{e\}}|$, temos que

$$|\varepsilon_{H \cup I \cup F \setminus \{e\}}(X')| + |\varepsilon_{H \cup I \cup F \setminus \{e\}}(Y')| \leq f(X') + f(Y') - 2p + 2,$$

sendo que X' e Y' são obtidos da mesma forma que X e Y , trocando $\varepsilon_{H \cup I \cup F}$ -supermodularidade por $\varepsilon_{H \cup I \cup F \setminus \{e\}}$ -supermodularidade.

Pela viabilidade de H , $|\varepsilon_{H \cup I \cup F \setminus \{e\}}(X')| \geq f(X') - p$ e $|\varepsilon_{H \cup I \cup F \setminus \{e\}}(Y')| \geq f(Y') - p$. Então pelo menos um entre X' e Y' , digamos X' , $|\varepsilon_{H \cup I \cup F \setminus \{e\}}(X')| = f(X') - p$. No caso de X e X' serem o mesmo conjunto, então X é uma testemunha para e , e temos $X_e = X = X'$.

No caso de X e X' serem diferentes, temos duas possibilidades:

- $X = W_e - W_f - \varepsilon_{H \cup I \cup F}(W_f)$ e $X' = W_e - W_f - \varepsilon_{H \cup I \cup F \setminus \{e\}}(W_f)$;
- $X = W_f - W_e - \varepsilon_{H \cup I \cup F}(W_e)$ e $X' = W_f - W_e - \varepsilon_{H \cup I \cup F \setminus \{e\}}(W_e)$;

Qualquer que seja o caso, X' é uma testemunha para e , e então $X_e = X'$. O mesmo processo é aplicado com f para obter uma testemunha Y_f .

□

5.2 Prova do lema 5.1.1

Precisamos de mais um teorema e um lema para provar o lema 5.1.1.

Teorema 5.2.1 *É possível encontrar uma solução viável I para (IP_h) e uma solução viável y para o dual de (IP_h) tal que $c(I) \leq 2 \sum_{B \subseteq V} h(B)y_B$.*

Prova. Dadas as condições da seção anterior, a prova segue da prova do lema 3.0.2. □

Seja y como no teorema acima. Seja $\bar{y}_{BA} = y_B$, se $A = \varepsilon_H(B)$ e $\bar{y}_{BA} = 0$, caso contrário.

Lema 5.2.2 $c(I) \leq \frac{2}{p} OPT$.

Prova. Pela definição, $h(B) = 1$ se e somente se $f(B) - |\varepsilon_H(B)| = p$. Então,

$$c(I) \leq 2 \sum_{B \subseteq V} h(B)y_B = 2 \sum_{B \subseteq V} h(B)\bar{y}_{B\varepsilon_H(B)} = \frac{2}{p} \sum_{B \subseteq V} (f(B) - |\varepsilon_H(B)|)\bar{y}_{B\varepsilon_H(B)} \leq \frac{2}{p} OPT$$

□

O lema 5.1.1 segue diretamente do lema anterior, somando-se todas as fases.

5.3 Exemplo de funcionamento do algoritmo

As figuras a seguir mostram um exemplo do algoritmo funcionando. Os círculos preenchidos representam os terminais, enquanto os não preenchidos são os vértices de Steiner. As linhas tracejadas representam os conjuntos ativos. As arestas mais cheias são as que foram escolhidas. Os números sobre as arestas representam o seu custo. Os requisitos r_{ij} entre os terminais são dados na tabela abaixo:

r_{ij}	a	b	c
b	1		
c	2	3	
d	3	1	2

Tabela 5.1: Requisitos de conectividade

Note que o maior requisito é 3, e portanto o algoritmo tem 3 fases.

Na figura 5.1 vemos o processo de inserção de arestas da 1ª fase do algoritmo. Note que a primeira escolha poderia ser tanto a aresta bc quanto a aresta ad . Na escolha da terceira aresta também há várias arestas candidatas, todas as de custo unitário que saem dos conjuntos ativos. A quinta aresta escolhida liga dois conjuntos ativos, por isso é a escolhida.

Na figura 5.2 temos o processo de remoção de arestas da 1ª fase do algoritmo. Apenas uma aresta é removida.

Na figura 5.3 é mostrado o processo de inserção de arestas da 2ª fase. Note que agora os conjuntos ativos são os minimais que contém terminais e têm no máximo uma aresta saindo

Figura 5.1: 1ª fase: inserção de arestas

Figura 5.2: 1ª fase: remoção de arestas

deles. Por isso c e d não são ativos. Aqui também há várias arestas que podem ser escolhidas entre as de custo 1 que saem de um conjunto ativo.

Na figura 5.4 é mostrado o processo de remoção de arestas da 2ª fase. Também agora apenas uma aresta é removida.

Na figura 5.5 é mostrado o processo de inserção de arestas da 3ª fase. Note que agora os conjuntos ativos são os minimais que contêm terminais e têm no máximo duas arestas saindo deles. Neste caso, todos os terminais começam como conjuntos ativos.

Figura 5.3: 2ª fase: inserção de arestas

Figura 5.4: 2ª fase: remoção de arestas

Finalmente, a figura 5.6 mostra a única aresta removida nesta fase, e a solução final que o algoritmo devolve.

Figura 5.5: 3ª fase: inserção de arestas

Figura 5.6: 3ª fase: remoção de arestas

Capítulo 6

Conclusão

Neste trabalho estudamos o problema de construir redes seguras, com tolerância a falhas, pelo menor custo possível. Mostramos que em geral esse é um problema difícil de ser resolvido com exatidão, obtendo o menor custo, mas que é possível encontrar soluções bastante próximas daquela de menor custo.

Foram citados diversos algoritmos que produzem soluções aproximadas para o problema, e três deles foram estudados mais a fundo. Fizemos a implementação de um, que se mostrou, em casos práticos, bastante eficiente, obtendo soluções sempre muito próximas do ótimo.

Os dois primeiros algoritmos trabalham com o problema no qual apenas as ligações são suscetíveis a falhas, enquanto o terceiro enfoca o caso no qual alguns nós da rede também podem falhar.

Os três algoritmos estudados seguem a mesma linha, são algoritmos de aproximação baseados no método primal-dual. Esse método mostrou-se muito adequado ao problema. O algoritmo implementado foi o primeiro deles, para o caso em que apenas as ligações podem falhar. Ele é mais simples que o segundo, obtendo um resultado com a mesma razão de aproximação, mas com uma complexidade de tempo maior. Os testes realizados mostraram que o algoritmo funciona de uma maneira adequada, mas para instâncias grandes ainda tem um tempo de execução elevado, mesmo esse tempo tendo um crescimento polinomial.

A seguir, uma tabela comparando os três algoritmos.

Algoritmo do capítulo:	Razão de aproximação	Complexidade de tempo de execução
3	$2k$	$O(n^4 k^3)$
4	$2k$	$O(n^2 k^2 + n^2 k \sqrt{\log \log n})$
5	$2H_k$	$O(n^4 k^3)$

Tabela 6.1: Comparação entre os algoritmos

A continuação deste trabalho pode seguir dois caminhos:

- **implementação do algoritmo do capítulo 4:** implementar um algoritmo mais eficiente e comparar, em uso real, qual o ganho de tempo que ele proporciona;

- **novos algoritmos:** estudar e projetar outros algoritmos para o problema, possivelmente utilizando outras técnicas, tentando encontrar uma melhor razão de aproximação e um custo de tempo menor.

Apêndice A

Código da implementação

Implementação do algoritmo do capítulo 3, escrito em linguagem C.

A.1 Função principal

```
void main_alg (Graph *g) {  
  
    Vertex *v;  
    long k = 0, p, i, custo = 0;  
    Aresta *Fp = NULL, *f;  
  
    for (v = g->vertices; v < g->vertices + g->n; v++) {  
        Arc *a;  
        for (a = v->arcs; a; a = a->next) {  
            if (a->r > k) k = a->r;  
            a->residual = gb_new_graph(g->n);  
        }  
    }  
  
    for (p = 1; p <= k; p++)  
        Fp = uncrossable (g, p, Fp);  
  
    printf ("Arestas escolhidas (em ordem contrária à de escolha):\n");  
    for (f = Fp, i = 1; f; f = f->next, i++) {  
        printf ("(%d,%d)%s",  
            f->first - g->vertices,  
            f->a->tip - g->vertices,  
            (f->next && i%5 ? ", " : "\n"));  
        custo += f->len;  
    }  
}
```

```

printf ("O custo total é de %ld.\n", custo);

if (!viavel(g,Fp))
    printf ("Erro no algortimo!!!!\n");
    exit(1);
}

```

A.2 Função uncrossable

```

Aresta *uncrossable (Graph *g, long p, Arêsta *Fp) {

    Vertex *v;
    Aresta *F = NULL;

    // inicializa as variáveis duais
    for (v = g->vertices; v < g->vertices + g->n; v++) {
        v->d = (int) malloc (sizeof(float));
        *((float *) v->d) = 0.0;
    }

    // primeiros conjuntos violados
    for (v = g->vertices; v < g->vertices + g->n; v++) {
        Arc *a;
        for (a = v->arcs; a; a = a->next) {
            inicializa_residual (g, a, Fp, NULL);
            minimal_mincut (a->residual,
                           a->residual->vertices + (v - g->vertices),
                           a->residual->vertices + (a->tip - g->vertices), p);
        }
    }

    minimal_sets (g, p);

    // enquanto houver conjuntos ativos
    while (mark_active (g, p)) {
        Aresta *next;

        // procura e insere a próxima aresta
        next = find_edge (g);

        insert_edge (g, next);

        next->next = F;
    }
}

```

```

F = next;

// encontra os conjuntos violados
for (v = g->vertices; v < g->vertices + g->n; v++) {
    Arc *a;
    for (a = v->arcs; a; a = a->next) {
        inicializa_residual (g, a, Fp, F);
        minimal_mincut (a->residual,
                        a->residual->vertices + (v - g->vertices),
                        a->residual->vertices + (a->tip - g->vertices), p);
    }
}

minimal_sets (g, p);

}

// reverse delete
{
    Aresta *f1, *anterior;

    anterior = NULL;

    for (f1 = F; f1; f1 = f1->next) {
        if (anterior) anterior->next = f1->next;
        else F = f1->next;

        if (!delete (g, p, Fp, F)) {
            if (anterior) anterior->next = f1;
            else F = f1;
            anterior = f1;
        }
        else {
            Arc *b = f1->a;
            b->len = f1->len;
            if (f1->first < b->tip) b++;
            else b--;
            b->len = f1->len;
        }
    }
}

f1 = F;
if (f1) {
    while (f1->next) f1 = f1->next;
}

```

```

        f1->next = Fp;
        Fp = F;
    }

}

return Fp;

}

```

A.3 Função `minimal_mincut`

```

void minimal_mincut (Graph *g, Vertex *u, Vertex *v, long p) {

    if (flow (g, u, v) == p-1) {
        g->min_val = bfs_r(g, u, v);
        g->min = u;
    }
    else {
        g->min_val = 0;
        g->min = NULL;
    }

}

```

A.4 Função `minimal_sets`

```

void minimal_sets (Graph *g, long p) {

    Vertex *v;

    for (v = g->vertices; v < g->vertices + g->n; v++) {
        v->mini_val = MAXLONG;
        v->minimal = NULL;
    }

    for (v = g->vertices; v < g->vertices + g->n; v++) {
        register Vertex *u;
        register Arc *a;

        for (a = v->arcs; a; a = a->next) {
            Graph *rs = a->residual;

```



```

    }
    if (w == g->vertices + g->n) {
        rs = v->minimal;
        for (u = rs->min; u; u = u->nmin) {
            w = g->vertices + (u - rs->vertices);
            w->ativo = count;
        }
        count++;
    }
}

return count;
}

```

A.6 Função find_edge

```

Aresta *find_edge (Graph *g) {

    Vertex *v;
    Aresta *next = (Aresta *) malloc (sizeof(Aresta));
    Arc *nt;
    float peso = MAXFLOAT, aresta;

    for (v = g->vertices; v < g->vertices + g->n; v++) {
        Arc *a;
        for (a = v->arcs; a; a = a->next) {
            if (a->len && v->ativo != a->tip->ativo) {
                if (v->ativo == -1)
                    aresta = a->len - *((float *)a->tip->d);
                else if (a->tip->ativo == -1)
                    aresta = a->len - *((float *)v->d);
                else aresta = (a->len - *((float *)v->d) - *((float *)a->tip->d))/2;
                if (aresta < peso) {
                    peso = aresta;
                    next->first = v;
                    next->a = a;
                    next->len = a->len;
                    nt = a;
                }
            }
        }
    }

    if (!nt) {
        printf ("Não existe solução!!!\n");
    }
}

```

```

    exit (0);
}
nt->len = 0;
if (next->first < next->a->tip) nt++;
else nt--;
nt->len = 0;

for (v = g->vertices; v < g->vertices + g->n; v++)
    if (v->ativo >= 0) *((float *)v->d) += peso;

return next;
}

```

A.7 Função insert_edge

```

void insert_edge (Graph *g, Aresta *aresta) {

    Vertex *v;
    Graph *rs;

    for (v = g->vertices; v < g->vertices + g->n; v++) {
        Arc *a;

        for (a = v->arcs; a; a = a->next) {
            rs = a->residual;
            switch_to_graph (NULL);
            switch_to_graph (rs);
            gb_new_edge (a->residual->vertices + (aresta->first - g->vertices),
                        a->residual->vertices + (aresta->a->tip - g->vertices), 1);
        }
    }
}

```

A.8 Função inicializa_residual

```

void inicializa_residual (Graph *g, Arc *a, Aresta *Fp, Aresta *F) {

    Aresta *aresta;

    gb_recycle (a->residual);

    a->residual = gb_new_graph (g->n);
}

```

```

for (aresta = Fp; aresta; aresta = aresta->next)
    gb_new_edge (a->residual->vertices + (aresta->first - g->vertices),
                a->residual->vertices + (aresta->a->tip - g->vertices), 1);

for (aresta = F; aresta; aresta = aresta->next)
    gb_new_edge (a->residual->vertices + (aresta->first - g->vertices),
                a->residual->vertices + (aresta->a->tip - g->vertices), 1);

}

```

A.9 Função delete

```

int delete (Graph *g, long p, Aresta *Fp, Aresta *F) {

    Vertex *v;

    for (v = g->vertices; v < g->vertices + g->n; v++) {
        Arc *a;
        for (a = v->arcs; a; a = a->next) {
            inicializa_residual (g, a, Fp, F);
            minimal_mincut (a->residual,
                           a->residual->vertices + (v - g->vertices),
                           a->residual->vertices + (a->tip - g->vertices), p);
        }
    }

    minimal_sets (g, p);

    return (!mark_active (g, p));
}

```

A.10 Função bfs

```

int bfs (Graph *g, Vertex *s, Vertex *t) {

    int existe = 0;
    register Vertex *v;
    register Vertex *u;
    Vertex *sentinel = g->vertices + g->n;

    for (v = g->vertices; v < sentinel; v++)
        v->link = v->back = NULL;
}

```

```

v = s;
v->link = sentinel;
u = v;

for (; v != sentinel; v = v->link) {
    register Arc *a;

    for (a = v->arcs; a; a = a->next) {
        register Vertex *w;
        if (a->capacity == 0) continue;
        w = a->tip;
        if (w == t) existe = 1;
        if (!w->link) {
            w->link = sentinel;
            w->back = v;
            u->link = w;
            u = w;
        }
    }
}

return existe;
}

```

A.11 Função bfs_r

```

int bfs_r (Graph *g, Vertex *s, Vertex *t) {

    int tamanho = 1;
    register Vertex *v;
    register Vertex *u;
    register Vertex *minimo;
    Vertex *sentinel = g->vertices + g->n;

    for (v = g->vertices; v < sentinel; v++)
        v->link = v->back = NULL;

    v = s;
    v->link = sentinel;
    u = v;
    v->nmin = NULL;
    minimo = v;
    v->mincut = 1;

```

```

for (; v != sentinel; v = v->link) {
    register Arc *a;

    for (a = v->arcs; a; a = a->next) {
        register Vertex *w;
        if (a->capacity == 0) continue;
        w = a->tip;
        if (!w->link) {
            tamanho++;
            minimo->nmin = w;
            w->nmin = NULL;
            w->mincut = 1;
            minimo = w;
            w->link = sentinel;
            w->back = v;
            u->link = w;
            u = w;
        }
    }
}

return tamanho;
}

```

A.12 Função `augment_path`

```

int augment_path (Graph *g, Vertex *s, Vertex *t) {

    int min_cap = INT_MAX;
    Vertex *v;

    if (!bfs(g,s,t)) return 0;

    for (v = t; v != s; v = v->back) {
        Arc *a;
        for (a = v->back->arcs; a->tip != v; a = a->next);

        if (a->capacity < min_cap) min_cap = a->capacity;
    }

    for (v = t; v != s; v = v->back) {
        Arc *a, *b;
        for (a = v->back->arcs; a->tip != v; a = a->next);

```

```

    if (v < v->back) b = a-1;
    else b = a+1;

    a->capacity -= min_cap;
    b->capacity += min_cap;

}

return min_cap;

}

```

A.13 Função flow

```

int flow (Graph *g, Vertex *s, Vertex *t) {

    int flux, total = 0;

    while ((flux = augment_path (g,s,t)))
        total += flux;

    return total;
}

```

A.14 Função f

```

int f (Graph *g, Graph *rs) {

    Vertex *u, *w, *z;
    int max = 0;

    if (!rs) return 0;

    for (u = rs->min; u; u = u->nmin) {
        Arc *a;
        w = g->vertices + (u - rs->vertices);
        for (a = w->arcs; a; a = a->next) {
            z = rs->vertices + (a->tip - g->vertices);
            if (!z->mincut && a->r > max) max = a->r;
        }
    }

    return max;
}

```

A.15 Função candidato

```
int candidato (Graph *a, Graph *b) {

    int inter = 1;
    int ninter = 1;
    Vertex *u, *v;

    if (!b) return 1;

    for (u = a->vertices, v = b->vertices; u < a->vertices + a->n; u++, v++) {
        if (u->nocorte && v->nocorte) inter = 0;
        else if (u->nocorte && !v->nocorte) ninter = 0;
    }

    return (inter || ninter);
}
```

A.16 Função bfs_caminho

```
int bfs_caminho (Graph *g, Vertex *s, Vertex *t, long i) {

    int existe = 0;
    register Vertex *v;
    register Vertex *u;
    Vertex *sentinel = g->vertices + g->n;

    for (v = g->vertices; v < sentinel; v++)
        v->link = v->back = NULL;

    v = s;
    v->link = sentinel;
    u = v;

    for (; v != sentinel; v = v->link) {
        register Arc *a;

        for (a = v->arcs; a; a = a->next) {
            register Vertex *w;
            if (a->len >= i) continue;
            w = a->tip;
            if (w == t) existe = 1;
            if (!w->link) {
                w->link = sentinel;
            }
        }
    }
}
```

```

        if (v < w) w->back_a = a+1;
        else w->back_a = a-1;
        u->link = w;
        u = w;
    }
}

return existe;
}

```

A.17 Função caminhos

```

int caminhos (Graph *g, long v1, long v2, long n, long i) {
    long j;
    Vertex *v, *w;

    for (j = 0; j < n; j++) {
        if (!bfs_caminho (g, g->vertices + v1, g->vertices + v2, i)) return 0;
        v = g->vertices + v2;
        while (v != g->vertices + v1) {
            Arc *a = v->back_a;

            w = a->tip;
            a->len = i;
            if (v < w) a++;
            else a--;
            a->len = i;
            v = w;
        }
    }

    return 1;
}

```

A.18 Função viavel

```

int viavel (Graph *g, Aresta *F) {

    Graph *gv;
    Vertex *v;
    Aresta *f;
    long i = 1;

```



```

gv = gb_new_graph(g->n);

for (f = F; f; f = f->next)
    gb_new_edge (gv->vertices + (f->first - g->vertices),
                gv->vertices + (f->a->tip - g->vertices), 0);

for (v = g->vertices; v < g->vertices + g->n; v++) {
    Arc *a;
    for (a = v->arcs; a; a = a->next) {
        if (a->r)
            if (!caminhos (gv, v - g->vertices, a->tip - g->vertices, a->r, i++))
                return 0;
    }
}
return 1;
}

```

Referências Bibliográficas

- [1] A. Agrawal, P. Klein, and R. Ravi. When tres collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24:440–456, 1995.
- [2] T. Böhme, F. Göring, and J. Harant. Menger’s theorem. *Journal of Graph Theory*, 37:35–36, 2001.
- [3] M.H. Carvalho, M.R. Cerioli, R. Dahab, P. Feofiloff, C.G. Fernandes, C.E. Ferreira, K.S. Guimarães, F.K. Miyazawa, J.C. Pina Jr., J.A.R. Soares, and Y. Wakabayashi. Uma introdução sucinta a algoritmos de aproximação, 2001.
- [4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [5] L. Fleischer, K. Jain, and D. Williamson. An iterative rounding 2-approximation algorithm for the element connectivity problem. In *42nd IEEE symposium on Foundations of Computer Science*, page 339, 2001.
- [6] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning tress in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.
- [7] H.N. Gabow, M.X. Goemans, and D.P. Williamson. An efficient approximation algorithm for the survivable design problem. *Mathematical Programming*, 82:13–40, 1998.
- [8] H.W. Gale, D. Kuhn and A.W. Tucker. Linear programming and the theory of games. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 317–329. Wiley, 1951.
- [9] M. Grötschel, C.L. Monma, and M. Stoer. Facet for polyhedra arising in the design of communication networks with low-connectivity constraints. *SIAM Journal of Optimization*, 2:474–504, 1992.
- [10] F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4:221–225, 1975.
- [11] C.G. Han and T.J. Lee. Heuristics for the survivable network design problem. In *International Conference on Internet Computing*, pages 173–178, 2000.

- [12] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing, 1996.
- [13] K. Jain, I. Mandoiu, V.V. Vazirani, and D.P. Williamson. A primal-dual schema based approximation algorithm for the element connectivity problem. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.
- [14] D.E. Knuth. *The Stanford GraphBase*. ACM Press, 1993.
- [15] K. Steiglitz, P. Weiner, and D. Kleitman. The design of minimum-cost survivable networks. *IEEE Transactions on Circuits and Systems*, 16:455–460, 1969.
- [16] D.P. Williamson, M.X. Goemans, M. Mihail, and V.V. Vazirani. A primal-dual approximation algorithm for generalized Steiner network problems. *Combinatorica*, 15:435–454, 1995.