Implementações do Método de Aproximação Primal-Dual
Aplicado ao Problema da Floresta de Steiner

Rafael Pereira Luna

Orientadora: Profa. Dra. Cristina Gomes Fernandes

— São Paulo, março de 2006 —

– Durante o desenvolvimento deste trabalho, o aluno recebeu apoio financeiro da CAPES –

Sumário

1 Introdução **3**

1.1 O Problema da Floresta de Steiner 4

1.2 O algoritmo de Goemans e Williamson 5

1.3 Implementações 13

1.4 Programação letrada e CWEB 14

1.5 Stanford GraphBase 16

1.6 Estruturas para a representação de conjuntos de terminais e florestas de Steiner e algumas definições de macros 21

1.7 Heaps de Fibonacci, árvores AVL e union-find 24

2 Implementação de Goemans e Williamson **29**

2.1 Descrição da implementação 29

2.2 Implementação em CWEB 32

3 Implementação de Gabow, Goemans e Williamson **47**

3.1 Descrição da implementação 48

3.2 Implementação em CWEB 51

4 Implementação de Klein **73**

4.1 Descrição da implementação 74

4.2 Implementação em CWEB 81

5 Implementação de Cole, Hariharan, Lewenstein e Porat **115**

5.1 Descrição da implementação 116

5.2 Estruturas de dados 118
5.3 Função principal 122

6 Fase da Limpeza **135**

A Cálculo do ancestral comum mais próximo **145**
A.1 Dois casos especiais: cadeias e árvores binárias completas 145
A.2 O algoritmo 147
A.3 Implementação em CWEB 152

B Função *main* do programa **159**

Referências Bibliográficas **179**

Índice remissivo **183**

Índice remissivo para o código **185**

Capítulo 1

Introdução

Problemas de otimização combinatória têm aplicações nas mais diversas áreas, incluindo, por exemplo, projeto de redes de telecomunicação, de circuitos VLSI, empacotamento de objetos em containers, escalonamento e roteamento de veículos, etc. Vários destes problemas são reconhecidamente difíceis — NP-difíceis — e algoritmos de aproximação [3, 14, 27] são uma forma de atacá-los que têm recebido bastante atenção dentro de otimização combinatória na última década.

Dentre as várias técnicas de projeto de algoritmos de aproximação, uma se destaca por sua elegância e versatilidade: o método primal-dual de aproximação. Sofisticado, tal método deriva-se do método primal-dual clássico, que originou algoritmos exatos para problemas de fluxo em redes, de emparelhamento, de caminhos mais curtos.

O algoritmo de Bar-Yehuda e Even [2], de 1981, para o problema da cobertura por vértices, foi o primeiro uso (ainda que implícito) do método de aproximação primal-dual. Foram os trabalhos de Agrawal, Klein e Ravi [1] e de Goemans e Williamson [12], da década de 90, que retomaram e formalizaram o uso do método para o projeto de algoritmos de aproximação. Desde a publicação desses trabalhos, vários outros apareceram, tratando de problemas diferentes [4, 11, 13, 15, 16, 22] ou de implementações alternativas que tornam o método mais eficiente [5, 9, 17].

Apresentaremos aqui algumas das implementações do método de aproximação primal-dual aplicado ao Problema da Floresta de Steiner. Inicialmente, definiremos o problema em questão e daremos uma descrição de um algoritmo de aproximação primal-dual proposto por Goemans e Williamson [12] que pode ser aplicado a ele. Após isso, apresentaremos quatro maneiras distintas de implementar este algoritmo, sugeridas por Goemans e Williamson [12], Cole et al. [5], Klein [17] e Gabow, Goemans e Williamson [9], exibindo ainda uma implementação em CWEB para cada uma delas. Ao final, serão apresentados alguns resultados obtidos através da análise experimental dessas implementações.

Buscamos, em nosso estudo, implementar de maneira o mais eficiente possível a segunda fase do algoritmo de Goemans e Williamson. Nesse processo, criamos uma implementação dessa fase para o Problema da Floresta de Steiner que é tão rápida quanto a implementação descrita por Gabow, Goemans e Williamson [9]. Nossa implementação utiliza um algoritmo para encontrar o ancestral comum mais próximo de dois vértices em uma árvore enraizada. Apresentaremos esta técnica no capítulo correspondente à implementação proposta por Goemans e Williamson e, no apêndice A, exibiremos uma implementação em CWEB do algoritmo desenvolvido por Schieber e Vishkin [25] para o cáculo do ancestral comum mais próximo.

1.1 O Problema da Floresta de Steiner

Aqui e ao longo de todo o resto deste texto, denotaremos o conjunto dos vértices e o conjunto das arestas de um grafo G por V_G e E_G, respectivamente. Além disso, denotaremos por $Q_{\geq}$ o conjunto dos racionais não-negativos.

Seja G um grafo e seja $\mathcal{R}$ uma família de subconjuntos de V_G. Dizemos que uma floresta F do grafo G é uma *$\mathcal{R}$-floresta* de G se F é uma floresta geradora de G e cada um dos conjuntos da família $\mathcal{R}$ se encontra inteiramente contido no conjunto dos vértices de alguma componente de F. Quando $\mathcal{R}$ está implícito, dizemos que uma floresta satisfazendo tal propriedade é uma *floresta de Steiner* de G.

O **Problema da Floresta de Steiner** consiste no seguinte: dados um grafo G, uma função c de E_G em $Q_{\geq}$ e uma família $\mathcal{R}$ de subconjuntos de V_G, encontrar uma *$\mathcal{R}$-floresta* F que minimize $c(F) = \sum_{e \in E_F} c_e$.

Note que podemos supor que os conjuntos de $\mathcal{R}$ são dois a dois disjuntos, pois, caso contrário, bastaria substituir cada par de conjuntos que não satisfaz esta restrição pela união dos conjuntos no par para obtermos um problema equivalente.

Cada conjunto da família $\mathcal{R}$ é chamado de *conjunto de terminais* e os vértices que não pertencem aos conjuntos de $\mathcal{R}$ são chamados de *vértices de Steiner*. Além disso, diremos que c_e é o custo da aresta e e que $c(F)$ é o custo da floresta F. A figura 1.1 mostra uma instância do problema, juntamente com uma possível solução.

O Problema da Floresta de Steiner tem aplicações no roteamento de circuitos VLSI e no projeto de redes. Considere, por exemplo, o projeto de uma rede de computadores com as seguintes restrições de conectividade. Nesta rede, existem dois tipos de computadores: os computadores do *tipo 1* que são mais importantes e, por isso, devem estar conectados a determinados computadores do mesmo tipo, de forma que possam comunicar-se entre si, e os computadores do *tipo 2*, sobre os quais não há restrição de conectividade e eventualmente podem ser utilizados como nós intermediários para a conexão de computadores do *tipo 1*. Para cada computador do *tipo 1*, são

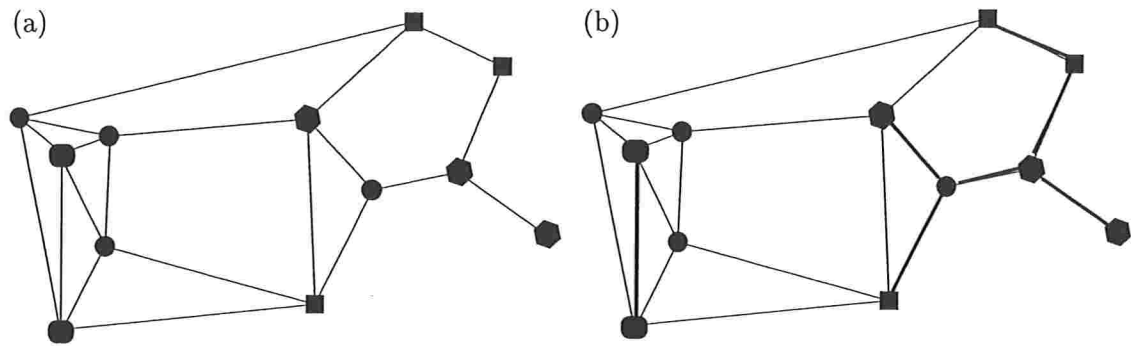

Figura 1.1: (a) Instância do problema: aqui, a família $\mathcal{R}$ é formada por três conjuntos de terminais: o conjunto dos quadrados, o conjunto dos retângulos com cantos arredondados e o conjunto dos hexágonos. Os círculos são vértices de Steiner. Além disso, o custo de cada aresta é dado pelo seu comprimento. (b) Uma possível solução, onde as arestas da floresta Steiner estão desenhadas com um traço mais escuro que as demais.

especificados os demais computadores do *tipo 1* aos quais ele deve estar conectado. Além disso, deve-se levar em conta o custo para se conectar diretamente dois computadores através de fibras óticas. Por fim, não é necessário que tal rede seja conexa. Podemos modelar esta situação através de um grafo onde cada grupo de computadores do *tipo 1* que precisam estar conectados é um conjunto de terminais, os computadores do *tipo 2* são vértices de Steiner e o custo de cada aresta reflete o custo da conexão direta, por fibra ótica, de seus extremos. Neste caso, o problema de se projetar uma rede de custo mínimo se reduz ao Problema da Floresta de Steiner.

Não se conhece um algoritmo eficiente (polinomial no tamanho da entrada $(G, c, \mathcal{R})$) para resolver o Problema da Floresta de Steiner. Se a família $\mathcal{R}$ é formada por um único conjunto, o problema se reduz ao conhecido *Problema de Steiner em Grafos*, o qual é NP-difícil [10]. Neste caso particular, o melhor algoritmo de aproximação que se conhece é uma 1,55-aproximação obtida por Robins e Zelikovsky [23]. O primeiro algoritmo de aproximação para o Problema da Floresta de Steiner foi projetado por Agrawal, Klein e Ravi [1].

1.2 O algoritmo de Goemans e Williamson

Goemans e Williamson [12] generalizaram o algoritmo de Agrawal, Klein e Ravi e projetaram um algoritmo de aproximação que se aplica a uma classe mais ampla de problemas em grafos. As implementações que apresentaremos aqui são todas baseadas neste algoritmo de Goemans e Williamson. Ele utiliza o método de aproximação primal-dual e é uma $\left(2 - \frac{2}{t}\right)$-aproximação para o Problema da Floresta de Steiner, onde $t = \sum_{T \in \mathcal{R}} |T|$.

Idéia geral do algoritmo

Um conceito fundamental no qual se apóia o algoritmo é o de *conjunto ativo*. Dados um grafo G e uma família $\mathcal{R}$ de conjuntos de terminais, um subconjunto S de V_G é dito *ativo* se existe um conjunto T em $\mathcal{R}$ tal que $T \cap S \neq \emptyset$ e $T \setminus S \neq \emptyset$. Quando o conjunto dos vértices de alguma componente de uma floresta F de G é ativo, dizemos que tal componente é uma componente *ativa* de F; caso contrário, a componente é dita *inativa*.

Note que se F é uma $\mathcal{R}$-floresta de G então, para todo conjunto S ativo, $E_F \cap \delta(S) \neq \emptyset$, onde $\delta(S)$ é o corte determinado por S no grafo G (ou seja, $\delta(S)$ é o conjunto das arestas de G que possuem exatamente uma ponta em S). Assim, podemos enunciar o Problema da Floresta de Steiner, na forma de um programa inteiro, como segue: encontrar um vetor x indexado por E_G que

$$\begin{array}{ll} \text{minimize } cx & \\ \text{sob as restrições} & \sum_{e\in\delta(S)} x_e \geq 1, \text{ para todo conjunto } S \text{ ativo} \\ & x_e \in \{0,1\}, \text{ para toda aresta } e \text{ em } E_G. \end{array}$$

Dado um subgrafo gerador F de G, o *vetor característico* de F é o vetor x tal que $x_e = 1$ para e em E_F e $x_e = 0$ para e em $E_G \setminus E_F$. Assim, toda floresta de Steiner do grafo G pode ser interpretada como uma solução viável do programa acima: dada uma floresta de Steiner F do grafo G, temos que o vetor característico de F é uma solução viável do programa. Além disso, podemos supor que não existem em G arestas de custo zero (caso contrário, bastaria somar uma constante positiva no custo das arestas de G) e, assim, é fácil ver que, se um vetor x é uma solução ótima do programa acima, então x é o vetor característico de alguma floresta de Steiner do grafo G: certamente, o subgrafo de G correspondente a x não tem circuitos, pois, caso contrário, poderíamos descartar uma aresta de um circuito deste subgrafo e obter uma solução viável de custo estritamente menor que x.

O programa linear abaixo é uma relaxação linear do programa inteiro:

$$\begin{array}{ll} \text{minimize } cx & \\ \text{sob as restrições} & \sum_{e\in\delta(S)} x_e \geq 1, \text{ para todo conjunto conjunto } S \text{ ativo} \\ & x_e \geq 0, \text{ para toda aresta } e \text{ em } E_G. \end{array}$$

O dual do programa acima é dado a seguir, onde y é um vetor indexado pela coleção de todos os conjuntos ativos de G:

$$\begin{array}{ll} \text{maximize } \sum_S y_S & \\ \text{sob as restrições} & \sum_{S:e\in\delta(S)} y_S \leq c_e, \text{ para toda aresta } e \text{ em } E_G \\ & y_S \geq 0, \text{ para todo conjunto } S \text{ ativo.} \end{array}$$

O algoritmo parte de uma solução viável y do programa dual e de uma floresta geradora F de G cujo vetor característico não satisfaz uma ou mais restrições no programa primal (note

que, deste modo, tal floresta tem uma ou mais componentes ativas). Então, em cada iteração, o algoritmo tenta "melhorar" a solução y, aumentando o valor das posições de y correspondentes às componentes ativas da floresta F, até que a restrição do programa dual correspondente a alguma das arestas do grafo seja satisfeita com igualdade. Quando isto ocorre, uma tal aresta é então incluída na floresta F e tem início uma nova iteração do algoritmo. No momento em que nenhuma das componentes de F for ativa, o algoritmo termina, devolvendo uma floresta de Steiner minimal contida em F. O custo da floresta devolvida pelo algoritmo nunca ultrapassa o dobro do custo de uma floresta de Steiner ótima.

A subseção abaixo descreve mais detalhadamente o algoritmo.

Descrição do algoritmo

Cada iteração do algoritmo começa com uma floresta geradora de G que contém pelo menos uma componente ativa e um vetor y, solução viável do programa dual. A primeira iteração começa com uma floresta geradora F que não contém arestas, sendo composta por n componentes, onde $n = |V_G|$ (ou seja, cada vértice do grafo é uma componente da floresta). Além disso, y é igual ao vetor nulo.

Dizemos que uma aresta de G é uma *aresta externa* a F se seus extremos estão em componentes distintas de F. Em cada iteração, uma aresta externa é incluída em F. A escolha da aresta a ser incluída é baseada na *folga* das arestas externas, sendo que a folga de uma aresta uv é definida como

- $(c_{uv} - \sum_{S:uv\in\delta(S)} y_S)/2$, se u e v pertencem a componentes ativas;
- $c_{uv} - \sum_{S:uv\in\delta(S)} y_S$, se u ou v, mas não ambos, pertence a uma componente ativa;
- ∞, se u e v pertencem a componentes inativas.

A aresta externa escolhida pelo algoritmo para ser incluída em F tem pelo menos um extremo em uma componente ativa e tem folga mínima dentre todas as arestas externas. Pode haver mais de uma aresta que satisfaz estas condições e, neste caso, o algoritmo escolhe uma delas arbitrariamente.

Uma vez que uma nova aresta foi incluída em F, atualiza-se o vetor y, somando-se a folga da aresta incluída ao valor nas posições de y correspondentes às componentes de F que eram ativas antes da inclusão da aresta. Tem início, então, uma nova iteração.

O algoritmo pára quando todas as componentes de F estiverem inativas (o que ocorre após, no máximo, $n-1$ iterações), devolvendo uma floresta de Steiner minimal contida em F. Se, ao final, quisermos não só a floresta, mas também uma delimitação inferior para o valor ótimo do

problema, podemos devolver o valor final de $\sum_S y_S$: pelo teorema fraco da dualidade, este valor é um limitante inferior para o custo de uma solução do Problema da Floresta de Steiner. Tal valor, obtido ao final de todas as iterações, pode ser usado no cálculo de um limitante superior para a razão entre o custo da floresta devolvida pelo algoritmo e o custo de uma solução para o Problema da Floresta de Steiner.

Abaixo, apresentamos em pseudo-código uma descrição não muito detalhada do algoritmo.

Algoritmo *MinFS(G, $\mathcal{R}$, c)*

1 $F \leftarrow (V_G, \emptyset)$
2 $E_{ext} \leftarrow E_G$
3 $y_S \leftarrow 0$ para todo conjunto S ativo
4 enquanto F tem alguma componente ativa faça
5 escolha $\overline{e}$ tal que $folga(\overline{e}) = \min\{folga(e) \mid e \in E_{ext}\}$
6 se $folga(\overline{e}) = \infty$
7 pare: "o problema não tem solução!"
8 $y_S \leftarrow y_S + folga(\overline{e})$ para cada componente ativa S de F
9 $E_F \leftarrow E_F \cup \{\overline{e}\}$
10 $E_{ext} \leftarrow E_{ext} \setminus \{uv \in E_G \mid u$ e v pertencem a uma mesma componente de $F\}$
11 seja F' uma $\mathcal{R}$-floresta minimal contida em F
12 devolva F'

Iremos nos referir aos passos que se encontram entre as linhas 4 e 10 como *fase iterativa* ou *primeira fase* do algoritmo. O passo 11 é conhecido como *fase da limpeza* ("clean-up stage") ou *segunda fase* do algoritmo. As figuras 1.2 e 1.3 ilustram a execução do algoritmo sobre a instância mostrada na figura 1.1.

Análise da razão de aproximação

Vamos mostrar aqui que o algoritmo de Goemans e Williamson que acabamos de ver nunca devolve uma $\mathcal{R}$-floresta de custo maior que 2 vezes o custo de uma solução do Problema da Floresta de Steiner. Mais precisamente, vamos mostrar que

$$c(F') \leq \left(2 - \frac{2}{t}\right) opt(G, c, \mathcal{R}), \tag{1.1}$$

onde F' é a floresta devolvida pelo algoritmo, $opt(G, c, \mathcal{R})$ é o custo de uma solução do Problema da Floresta de Steiner e $t = \sum_{T \in \mathcal{R}} |T|$.

Um dos invariantes fundamentais para a análise que faremos aqui é que, durante a fase iterativa, sempre que uma aresta uv é incluída na floresta F (linha 9 do algoritmo *MinFS*),

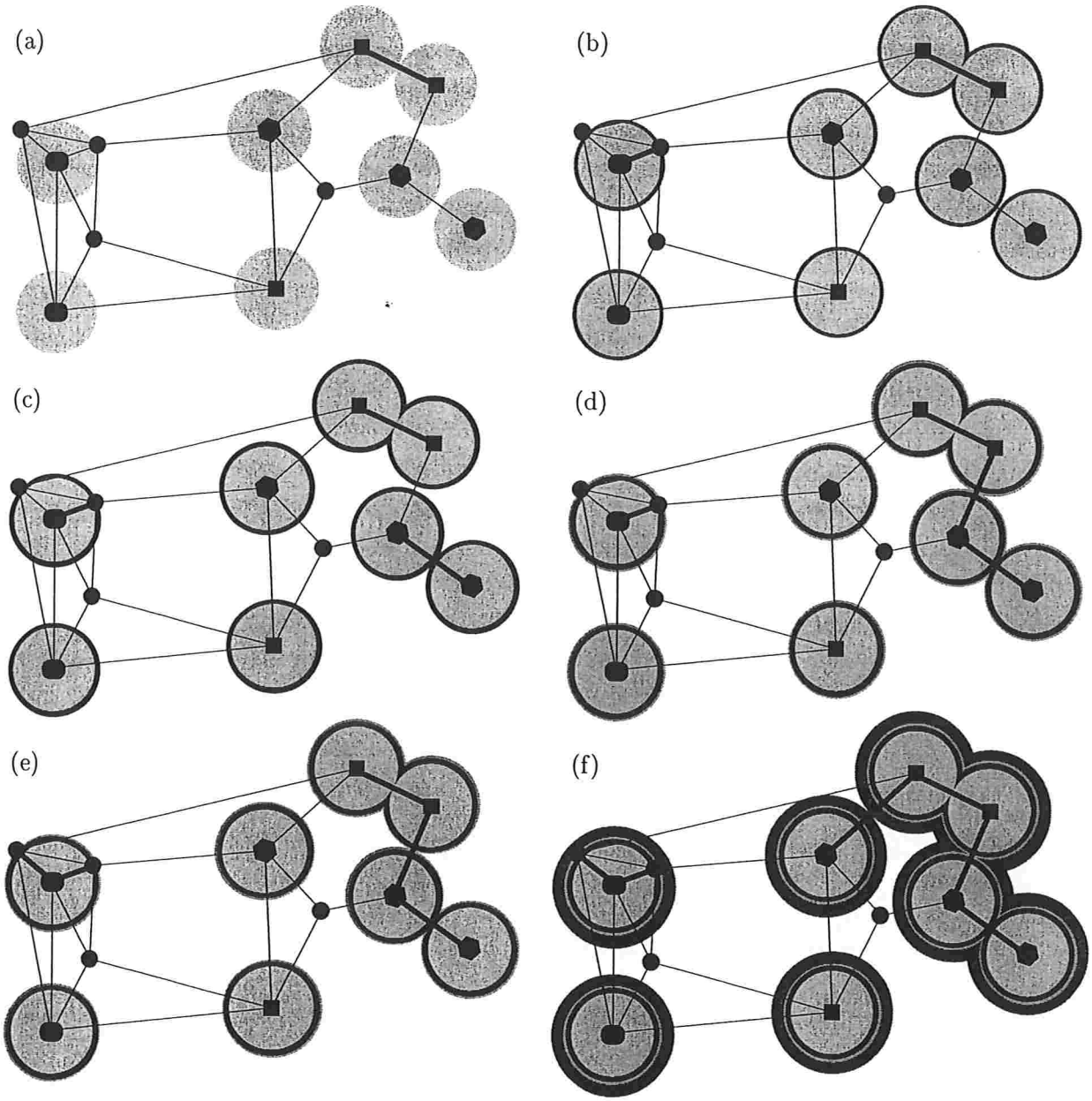

Figura 1.2: Execução do algoritmo sobre a instância da figura 1.1. (a) - (f) Situação ao final de cada uma das seis primeiras iterações da primeira fase do algoritmo. Em cada iteração, o raio do maior círculo em torno de cada vértice v em uma componente ativa S é incrementado do valor corrente da variável y_S (podemos pensar que, antes da primeira iteração, o maior círculo em torno de cada vértice do grafo tem raio 0). A folga de uma aresta uv com ao menos um extremo em uma componente ativa é igual ao comprimento de uv que se encontra na região externa aos círculos em torno de u e v. As arestas da floresta corrente estão desenhadas com um traço mais escuro que as demais.

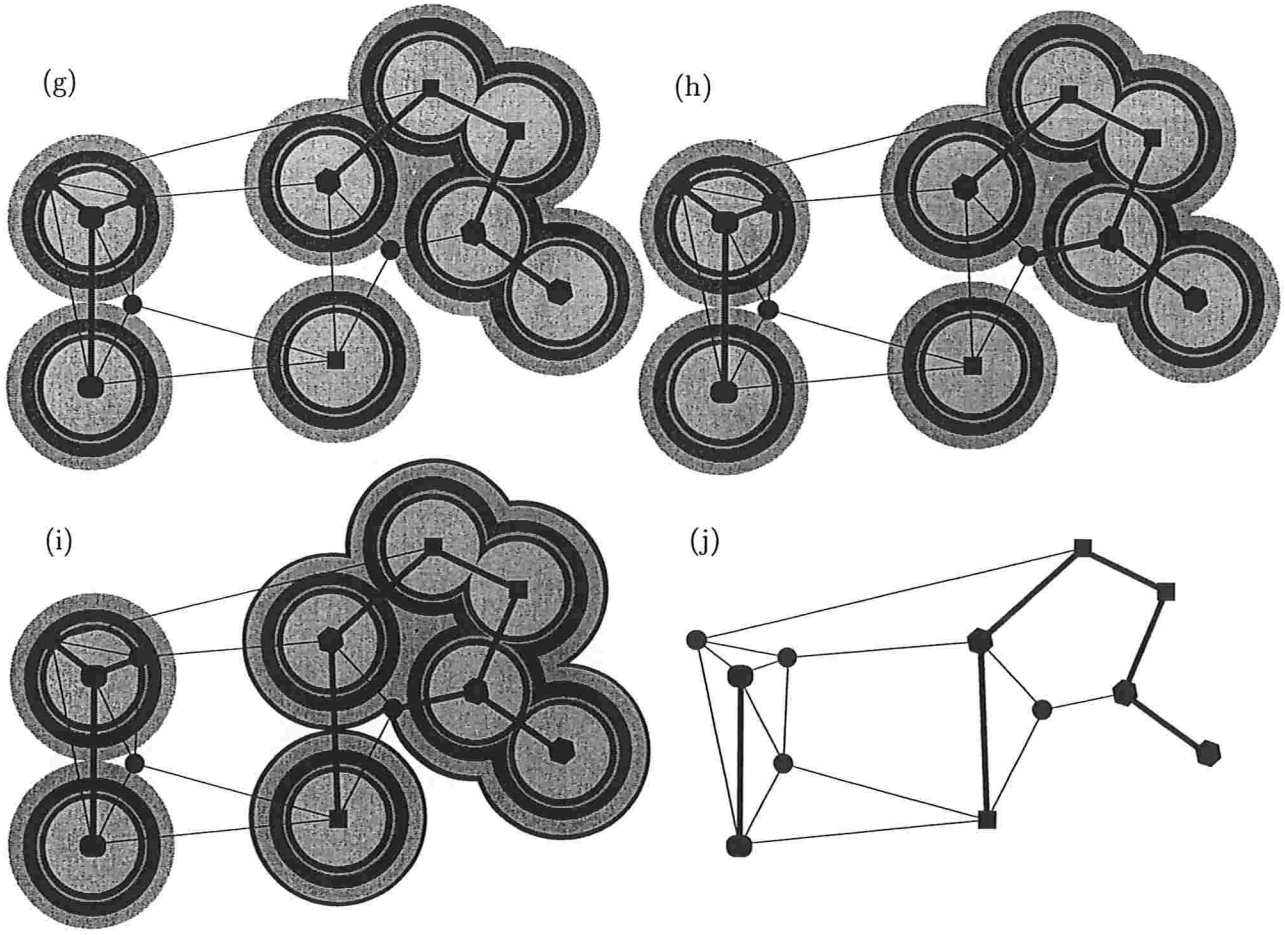

Figura 1.3: Execução do algoritmo sobre a instância da figura 1.1 (continuação). (g) - (i) Situação ao final da sétima, oitava e nona iterações da primeira fase do algoritmo. Note que, após a sétima iteração, a componente que contém os retângulos de cantos arredondados se torna inativa e, portanto, o raio do maior círculo em torno de seus vértices não será incrementado nas iterações seguintes. (j) Floresta devolvida pelo algoritmo após a execução da segunda fase, a qual tem custo 1.023 vezes maior que o custo de uma solução do problema.

temos que $c_{uv} = \sum_{S:uv\in\delta(S)} y_S$. Para ver que isto, de fato, ocorre, note que as únicas parcelas da somatória em questão que têm seu valor alterado, após a execução da linha 8 do algoritmo, são aquelas que correspondem a componentes ativas de F em que se encontram os extremos de uv. Assim, sejam S_u e S_v as componentes de F correspondentes a u e v, respectivamente, e seja y'_S o valor do vetor y na posição correspondente a S, imediatamente antes da execução da linha 8 do algoritmo. Se S_u e S_v são ativas, então apenas as posições de y correspondentes a S_u e S_v serão incrementadas de *folga*(uv) e, dessa forma, imediatamente após a execução da linha 8 do algoritmo,

$$\begin{aligned}\sum_{S:uv\in\delta(S)} y_S &= \sum_{S:uv\in\delta(S)} y'_S + \mathit{folga}(uv) + \mathit{folga}(uv)\\ &= \sum_{S:uv\in\delta(S)} y'_S + 2\,\frac{(c_{uv} - \sum_{S:uv\in\delta(S)} y'_S)}{2} = c_{uv}.\end{aligned}$$

Caso contrário, suponhamos, sem perda de generalidade, que apenas S_u é ativa. Assim, apenas a posição de y que corresponde a S_u será incrementada de *folga*(uv) e, deste modo, logo após a execução da linha 8,

$$\begin{aligned}\sum_{S:uv\in\delta(S)} y_S &= \sum_{S:uv\in\delta(S)} y'_S + \mathit{folga}(uv)\\ &= \sum_{S:uv\in\delta(S)} y'_S + \left(c_{uv} - \sum_{S:uv\in\delta(S)} y'_S\right) = c_{uv}.\end{aligned}$$

Note que, a partir do momento que uv é incluída em F, o valor da somatória $\sum_{S:uv\in\delta(S)} y_S$ não mais será alterado. Desta forma, ao final de todas as iterações da primeira fase, para toda aresta e em F, temos que $c_e = \sum_{S:e\in\delta(S)} y_S$. Em particular, o mesmo vale para toda aresta e na floresta F' devolvida pelo algoritmo e, desta forma,

$$c(F') = \sum_{e\in E_{F'}} c_e = \sum_{e\in E_{F'}} \sum_{S:e\in\delta(S)} y_S = \sum_S |\delta_{F'}(S)| y_S\,, \tag{1.2}$$

onde $\delta_{F'}(S)$ é o conjunto das arestas em F' que têm exatamente uma ponta em S.

Por outro lado, pelo teorema fraco da dualidade, sabemos que $\sum_S y_S \leq \mathit{opt}(G, c, \mathcal{R})$ e, assim,

$$\left(2 - \frac{2}{t}\right) \sum_S y_S \leq \left(2 - \frac{2}{t}\right) \mathit{opt}(G, c, \mathcal{R})\,. \tag{1.3}$$

Logo, tudo que precisamos fazer para provar (1.1) é mostrar que

$$\sum_S |\delta_{F'}(S)| y_S \leq \left(2 - \frac{2}{t}\right) \sum_S y_S\,. \tag{1.4}$$

Note que, no início da primeira iteração, esta desigualdade vale trivialmente, pois $y = 0$. Agora, suponhamos que (1.4) vale no início de uma dada iteração e sejam θ e $\mathcal{S}_F$, respectivamente, a

folga da aresta que será incluída em F nesta iteração e a família das componentes ativas de F antes da inclusão de tal aresta. Ao final da iteração, o lado esquerdo de (1.4) terá recebido um acréscimo igual a $\theta \sum_{S \in \mathcal{S}_F} |\delta_{F'}(S)|$, enquanto que o lado direito terá aumentado de $\left(2 - \frac{2}{t}\right) |\mathcal{S}_F| \theta$. Assim, para que (1.4) continue valendo ao final da iteração, basta que

$$\sum_{S \in \mathcal{S}_F} |\delta_{F'}(S)| \leq \left(2 - \frac{2}{t}\right) |\mathcal{S}_F|$$

e é exatamente isso que iremos provar no

Lema 1.1. *No início de cada iteração do algoritmo,*

$$\sum_{S \in \mathcal{S}_F} |\delta_{F'}(S)| \leq \left(2 - \frac{2}{t}\right) |\mathcal{S}_F| .$$

Prova. No que segue, diremos que duas componentes, digamos S e S', de uma floresta num grafo G são adjacentes em G se existe alguma aresta em G que tenha um extremo em S e o outro em S'.

Seja $\mathcal{Z}_F$ o conjunto das componentes inativas S da floresta F tais que $|\delta_{F'}(S)| \geq 1$ e seja F_0 a floresta obtida ao final de todas as iterações da primeira fase. Agora, considere o grafo H definido da seguinte forma:

- $V_H = \mathcal{S}_F \cup \mathcal{Z}_F$ e
- $E_H = \{\{S, S'\} \mid S \text{ e } S' \text{ são adjacentes em } F_0\}$.

Sabemos que não existem circuitos em H, pois, caso contrário, haveria algum circuito em F_0 e isto é absurdo, já que F_0 é uma floresta. Assim, H é uma floresta e, deste modo,

$$|E_H| \leq |V_H| - 1 = |\mathcal{S}_F| + |\mathcal{Z}_F| - 1.$$

Por outro lado, sabemos que $\sum_{S \in V_H} |\delta_H(S)| = 2|E_H|$ e, assim, como $|\delta_{F'}(S)| \leq |\delta_H(S)|$ para todo S em V_H, temos que

$$\begin{aligned}
&\sum_{S \in V_H} |\delta_{F'}(S)| \leq 2|E_H| \leq 2(|\mathcal{S}_F| + |\mathcal{Z}_F| - 1) \\
&\Rightarrow \sum_{S \in \mathcal{S}_F \cup \mathcal{Z}_F} |\delta_{F'}(S)| \leq 2(|\mathcal{S}_F| + |\mathcal{Z}_F| - 1) \\
&\Rightarrow \sum_{S \in \mathcal{S}_F} |\delta_{F'}(S)| \leq 2(|\mathcal{S}_F| + |\mathcal{Z}_F| - 1) - \sum_{S \in \mathcal{Z}_F} |\delta_{F'}(S)|.
\end{aligned}$$

Por fim, note que, como F' é uma $\mathcal{R}$-floresta minimal, não existe S em $\mathcal{Z}_F$ tal que $|\delta_{F'}(S)| = 1$ e, deste modo, $|\delta_{F'}(S)| \geq 2$ para todo S em $\mathcal{Z}_F$. Logo,

$$2(|\mathcal{S}_F| + |\mathcal{Z}_F| - 1) - \sum_{S \in \mathcal{Z}_F} |\delta_{F'}(S)| \leq 2(|\mathcal{S}_F| + |\mathcal{Z}_F| - 1) - 2|\mathcal{Z}_F| = 2|\mathcal{S}_F| - 2$$

e, portanto,

$$\sum_{S \in \mathcal{S}_F} |\delta_{F'}(S)| \leq 2|\mathcal{S}_F| - 2 = \left(2 - \frac{2}{|\mathcal{S}_F|}\right)|\mathcal{S}_F| \leq \left(2 - \frac{2}{t}\right)|\mathcal{S}_F|,$$

como queríamos provar. □

Assim, concluímos a prova de que a desigualdade (1.4) é sempre válida e, desta forma, juntando tudo, temos o seguinte

Teorema 1.1. *O algoritmo MinFS é uma* $\left(2 - \frac{2}{t}\right)$*-aproximação para o Problema da Floresta de Steiner.*

Prova. Segue diretamente de (1.2), (1.4) e (1.3) e do fato de que podemos implementá-lo de forma que ele consuma tempo polinomial para qualquer instância do problema, como veremos nos próximos capítulos. □

1.3 Implementações

A implementação mais simples do algoritmo de Goemans e Williamson para o Problema da Floresta de Steiner consome tempo $\Theta(n^3)$, onde n é o número de vértices do grafo dado. Goemans e Williamson [13] mencionam uma implementação direta do algoritmo que consome tempo $O(n^2 \log n)$. Tal implementação utiliza uma fila de prioridade para tornar mais rápida a determinação de uma aresta de folga mínima.

Klein [17] introduziu uma estrutura de dados que pode ser usada para obter-se uma implementação do algoritmo de Goemans e Williamson de complexidade $O(n\sqrt{m}\log n)$, que é mais rápida que a implementação $O(n^2 \log n)$ em grafos esparsos. Aqui, como de costume, m e n denotam, respectivamente, o número de arestas e de vértices do grafo.

Gabow, Goemans e Williamson [9] propuseram uma outra implementação do algoritmo, que organiza as filas de prioridade de uma forma diferente e tem complexidade $O(n(n+\sqrt{m \log \log n}))$.

A implementação de Cole et al. [5] usa uma idéia de subdividir as arestas dinamicamente e resulta em uma complexidade melhor, sob o custo de uma pequena degradação na razão de aproximação do algoritmo. Especificamente, tal implementação consome tempo $O(k(n+m)\log^2 n)$, onde k é um parâmetro extra que controla a degradação da razão de aproximação do algoritmo.

A partir do próximo capítulo, faremos uma exposição mais detalhada de cada uma dessas implementações e, para cada uma delas, exibiremos uma implementação em CWEB, um sistema de programação letrada (em inglês, "literate programming") que combina programação em C com documentação tipografada em LaTeX. O código em C correspondente a todas estas implementações se encontra no arquivo `sf.c` que está organizado de acordo com a seguinte estrutura:

1 ⟨ Header files of sf.c 2 ⟩
⟨ Data structures of sf.c 4 ⟩
⟨ Global variables of sf.c 257 ⟩
⟨ Auxiliary functions 20 ⟩
⟨ Steiner forest construction functions 10 ⟩
⟨ The main program 238 ⟩

Antes de dar prosseguimento à apresentação das seções em que se divide o arquivo sf.c, vamos redigir algumas palavras sobre programação letrada e CWEB.

1.4 Programação letrada e CWEB

O conceito de programação letrada foi introduzido por Knuth [19]. Em seu artigo "Literate Programming" [18], de 1984, ele escreve:

> Eu acredito que já é tempo de documentarmos nossos programas de uma forma significativamente melhor e a melhor maneira de alcançarmos isto é considerando programas como obras de literatura. Daí vem o título: "Literate Programming".
>
> Vamos mudar nossa atitude tradicional na construção de programas: ao invés de imaginar que nossa principal tarefa é dizer ao computador o que fazer, vamos nos concentrar em explicar aos seres humanos o que nós queremos que o computador faça.
>
> O praticante da programação letrada pode ser considerado como um ensaísta, cuja principal preocupação é com a exposição e a excelência de estilo. Tal autor, com um tesouro em mãos, escolhe os nomes das variáveis cuidadosamente e explica o que cada uma significa. Ele ou ela aspira por um programa compreensível porque seus conceitos foram introduzidos em uma ordem que é melhor para a compreensão humana, usando uma mistura de métodos formais e informais que se reforçam mutuamente.

Knuth desenvolveu o sistema WEB de programação letrada, o qual permite ao usuário desenvolver programas que estejam de acordo com as idéias expostas no trecho acima. Estes programas podem ser vistos como documentos hipertexto, como a World Wide Web (de fato, Knuth começou a usar a palavra "web" neste sentido muito antes do nascimento da rede WWW). Em 1987, Silvio Levy adaptou o sistema para a linguagem C, dando origem ao CWEB. Uma descrição detalhada do sistema pode ser encontrada no livro "The CWEB System of Structured Documentation" [21].

Um programa CWEB é uma sequência de seções numeradas, as quais são chamadas de blocos (em inglês, "chunks"). Um programa completo pode ser visto como uma teia ("web") ou rede

que consiste de pequenos pedaços de código e interconexões entre os pedaços. Cada bloco de um programa CWEB começa com um comentário sobre os objetivos ou sobre algum aspecto importante do bloco. Tais comentários são escritos em português (por exemplo), de maneira informal. O bloco termina com uma parte formal, expressa em linguagem C, e supõe-se que o código formal faz exatamente o que os comentários sugerem. Entre os comentários introdutórios e o código C também pode haver uma ou mais definições de macro. Assim, cada bloco consiste de três partes, nesta ordem: comentário informal, definições de macros e código C, sendo que uma ou mais partes, destas três, podem estar vazias.

Alguns blocos têm nome e outros são anônimos. Um exemplo de bloco anônimo é o bloco 1 acima, aquele que contém a estrutura geral do arquivo `sf.c` que foi exibida no final da seção anterior. O código C de um bloco com nome começa com

⟨ Nome do bloco *id* ⟩ ≡

(onde *id* indica o número do bloco) e é seguido por um *texto de substituição* em C. Um exemplo disso é o código C do bloco 15 (o qual irá aparecer no próximo capítulo):

```
⟨Some initializations 15⟩≡
  {
    last = malloc(g→n * sizeof(Vertex *));
    sizes = malloc((g→n) * (g→num_ts) * sizeof(int));
    if (¬last ∨ ¬sizes) {
      err = 2;
      return;
    }
  }
```

Como podemos ver no exemplo acima, os textos de substituição são formatados de uma maneira especial em CWEB, o que torna mais fácil a leitura do código C. Os seguintes símbolos especiais são utilizados para melhorar a legibilidade:

símbolo	significado
$\leq$	`<=`
$\geq$	`>=`
$\neq$	`!=`
$\equiv$	`==`
$\wedge$	`&&`
$\vee$	`||`
$\neg$	`!`
$\rightarrow$	`->`
&	`&`
$\oplus$	`^`
Λ	`NULL`

Quando o nome de um bloco x aparece em um outro bloco y (o que ocorre, por exemplo, no bloco 1, que faz o papel de y neste caso), isso significa que, no arquivo que contém o código C do programa, o texto de substituição correspondente ao bloco x aparece no exato lugar do bloco y em que se encontra o nome de x. Assim, por exemplo, no arquivo `sf.c`, o texto de substituição do bloco 2 aparece no exato lugar do bloco 1 em que se encontra seu nome, ⟨ Header files of `sf.c` 2 ⟩. Por este motivo, no programa CWEB, veremos que, no final do bloco 2, aparece uma nota de rodapé indicando que ele foi usado no bloco 1.

É possível definir novos blocos que acrescentem mais código C ao texto de substituição de um bloco que já foi definido anteriormente. Neste caso, o novo bloco terá o mesmo nome do bloco definido anteriormente, só que, após o nome, ao invés de aparecer o sinal "≡", irá aparecer o sinal "+≡". Os blocos 2 e 8, como veremos adiante, exemplificam esta situação. Neste caso, como o bloco 8 é definido de forma a acrescentar código C ao bloco 2, aparece, no final de 2, uma nota de rodapé com uma mensagem do tipo "Veja também o bloco 8".

Por fim, veremos também que, algumas vezes, o nome de um bloco é o nome de um arquivo, escrito em fonte de máquina-de-escrever, como é o caso de ⟨ `extra.h` 7 ⟩. Isso indica que o texto de substituição daquele bloco será gravado no arquivo especificado. Desta forma, um só programa CWEB pode descrever vários arquivos que estejam, de alguma forma, relacionados.

1.5 Stanford GraphBase

Para representar vértices, arestas e grafos, faremos uso das estruturas de dados definidas no *Stanford GraphBase*, ao qual iremos nos referir, de modo abreviado, por SGB, um pacote de programas e conjuntos de dados projetados por Knuth [20] para gerar e manipular grafos e redes. Sendo assim, um dos arquivos-cabeçalho que se encontram no bloco ⟨ Header files of `sf.c` 2 ⟩,

como vemos abaixo, é `gb_graph.h`, que é o módulo do SGB onde se encontram definidas todas estas estruturas.

2 ⟨ Header files of `sf.c` 2 ⟩ ≡
```
#include "gb_graph.h"
```

Veja também blocos 8, 9, 28, 50, 166, 182, 216, 242 e 255.

Este código é citado nos blocos 2 e 9.

Este código é usado no bloco 1.

No SGB, grafos, vértices e arcos são representados por **structs** dos tipos **Graph**, **Vertex** e **Arc**, respectivamente. Como, com freqüência, manipularemos estas estruturas através de apontadores para elas, muitas vezes diremos, por exemplo, que uma variável v é um vértice ou guarda um vértice, quando, na verdade, v guarda um apontador para uma estrutura do tipo **Vertex**. O mesmo se aplica a arcos e grafos.

Cada vértice v possui um nome e um apontador para uma lista ligada dos arcos que têm a ponta inicial em v. O campo $v\to name$ guarda um apontador para uma cadeia de caracteres contendo o nome de v e o campo $v\to arcs$ armazena um apontador para a estrutura **Arc** correspondente ao primeiro arco da lista dos arcos que têm a ponta inicial em v. Quando tal lista for vazia, o conteúdo de $v\to arcs$ será igual a Λ. Caso contrário, cada arco a da lista, com exceção do último, guardará em $a\to next$ um apontador para a estrutura **Arc** do arco que sucede a na lista. O conteúdo do campo *next* do último arco da lista será igual a Λ.

Além do campo *next*, cada arco a possui dois outros campos: $a\to tip$ e $a\to len$. O primeiro guarda um apontador para a estrutura **Vertex** correspondente ao vértice que é ponta final de a; o outro guarda um inteiro cujo valor, dependendo do contexto, corresponde ao comprimento ou custo de a.

O número de vértices e o número de arcos de um grafo g se encontram armazenados, respectivamente, em $g\to n$ e $g\to m$. O campo $g\to vertices$ é um vetor de estruturas do tipo **Vertex** que correspondem aos $g\to n$ vértices do grafo g. Cada grafo contém ainda um campo $g\to id$ que nada mais é do que uma cadeia de caracteres que identifica o grafo, informando quais parâmetros foram usados para a sua geração.

As figuras 1.4 e 1.5 ilustram as estruturas que acabamos de descrever e a representação de um grafo no SGB.

Além dos campos mencionados acima, as estruturas **Graph**, **Vertex** e **Arc** possuem ainda campos adicionais que são chamados de *utility fields*. Cada um desses campos é um **union** do tipo **util**, o qual exibimos abaixo.

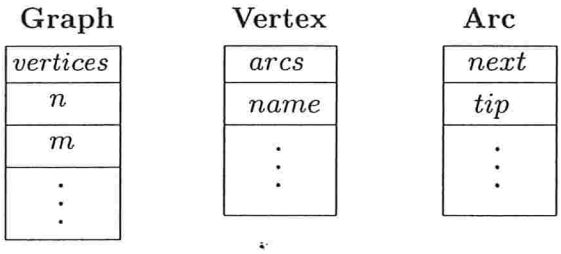

Figura 1.4: Ilustração das estruturas **Graph**, **Vertex** e **Arc**.

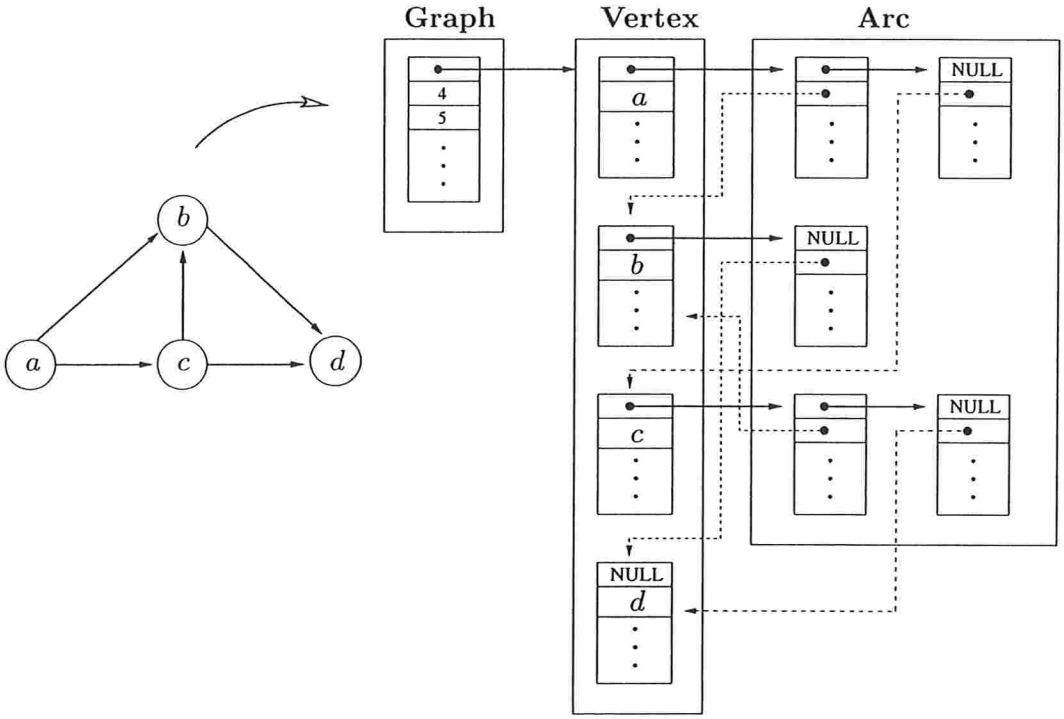

Figura 1.5: Um grafo e sua representação no SGB (para melhor visualização, as estruturas **Vertex** não foram desenhadas de forma contígua no vetor de vértices do grafo).

```
typedef union {
  struct vertex_struct *V;
  struct arc_struct *A;
  struct graph_struct *G;
  char *S;
  long I;
} util;
```

Um campo do tipo **util** pode ser usado para armazenar ou um vértice, ou um arco, ou um grafo, ou um apontador para uma cadeia de caracteres ou um inteiro, bastando, para isto, que adicionemos o sufixo adequado (". V", ". A", ". G", ". S" ou ". I", respectivamente) ao nome do campo. Uma prática muito comum e que também será aqui adotada é o uso de diretivas do tipo # **define** para renomear estes campos, dando nomes mais sugestivos a eles, de acordo com a forma como eles são utilizados na aplicação. Veremos vários exemplos desta prática ao longo deste e dos próximos capítulos.

Cada estrutura do tipo **Vertex** possui seis utility fields: u, v, w, x, y e z. Exibimos, abaixo, a definição completa de uma estrutura do tipo **Vertex**:

```
typedef struct vertex_struct {
  struct arc_struct *arcs;
  char *name;
  util u, v, w, x, y, z;
} Vertex;
```

As estruturas do tipo **Arc** possuem dois utility fields: a e b. Abaixo, é mostrada a definição completa de uma estrutura do tipo **Arc**:

```
typedef struct arc_struct {
  struct vertex_struct *tip;
  struct arc_struct *next;
  long len;
  util a, b;
} Arc;
```

Por fim, cada estrutura do tipo **Graph** possui seis utility fields: uu, vv, ww, xx, yy e zz. Abaixo, segue a definição completa de uma estrutura do tipo **Graph**.

```
typedef struct graph_struct {
  Vertex *vertices;
  long n;
  long m;
```

```
    char id[ID_FIELD_SIZE];
    char util_types[15];
    Area data;
    Area aux_data;
    util uu, vv, ww, xx, yy, zz;
  } Graph;
```

Note que a definição acima contém ainda três campos que não foram mencionados na descrição que fizemos até aqui: *util_types*, *data* e *aux_data*. Como estes campos não serão utilizados em nossas implementações, não iremos descrevê-los em detalhes. Diremos apenas que

- *util_types* é usado para armazenar uma cadeia de caracteres que descreve de que forma estão sendo utilizados cada um dos 14 utility fields correspondentes às estruturas que descrevem o grafo (isto é, qual dos cinco tipos de dados está armazenado em cada um desses 14 campos);
- *data* guarda um apontador para a área usada para armazenar arcos e cadeias de caracteres.
- *aux_data* guarda um apontador para a área usada para armazenar informação auxiliar relacionada ao grafo.

No SGB, não existe um **struct** para representar arestas, ao contrário do que acontece com arcos. Em vez disso, arestas são definidas em termos de arcos. Mais especificamente, uma aresta com extremos nos vértices u e v é representada através de um único par de arcos: um arco com a ponta inicial em u e a ponta final em v e um outro arco que tem a ponta inicial em v e a ponta final em u. Além disso, tendo em mãos um dos arcos do par que corresponde a uma dada aresta e o vértice que é ponta inicial deste arco, podemos facilmente (em tempo constante) determinar o outro arco. Em nossas implementações, usaremos a macro *inverse*(a), que definimos abaixo, para realizar esta tarefa. Usaremos também um dos utility fields de a, o qual renomearemos como *from*, para guardar um apontador para a estrutura **Vertex** do vértice que é ponta inicial de a.

```
3 #define from  a.V
  #define inverse(a)  ((a→from < a→tip) ? a + 1 : a − 1)
```

Assim, a partir de um arco, podemos determinar o par de arcos que representa a aresta. Em razão disso e por conveniência, muitas vezes iremos nos referir a um apontador a para uma estrutura do tipo **Arc** como "a aresta a" ou a uma lista de estruturas do tipo **Arc** como "uma lista de arestas". Assim, a menos que digamos o contrário, sempre que nos referirmos a uma aresta a, quando a, na verdade, for uma estrutura do tipo **Arc** ou um apontador para uma estrutura

deste tipo, estaremos nos referindo à aresta que é representada pelo par de arcos determinado pelo arco correspondente à estrutura ou apontador a.

Além da definição das estruturas usadas no SGB para a representação de um grafo, o arquivo `gb_graph.h` contém ainda o protótipo de funções básicas para a criação e manipulação de grafos. Algumas delas são utilizadas em nossas implementações e, abaixo, listamos estas funções.

- **Graph** $*gb_new_graph$(**long** n): cria um grafo com n vértices e nenhuma aresta e devolve um apontador para a estrutura **Graph** correspondente a este grafo;

- **char** $*gb_save_string$(**char** $*s$): recebe um apontador para uma cadeia de caracteres (com 0 demarcando o final da cadeia), guarda uma cópia dessa cadeia em uma área especialmente reservada para este propósito e devolve um apontador para o lugar onde se encontra a nova cópia;

- **void** gb_new_edge(**Vertex** $*u$, **Vertex** $*v$, **long** len): cria uma aresta com extremos nos vértices u e v e com comprimento (ou custo) igual a len. Se $u \neq v$ então, logo após a chamada a esta função, $u\to arcs$ guardará um apontador para a estrutura **Arc** de um dos arcos do par que representa a aresta, enquanto que $v\to arcs$ guardará um apontador para a estrutura **Arc** do outro;

- **void** $gb_recycle$(**Graph** $*g$): libera a memória ocupada pelo grafo g.

1.6 Estruturas para a representação de conjuntos de terminais e florestas de Steiner e algumas definições de macros

Dentre as estruturas de dados que estão especificadas na seção ⟨Data structures of `sf.c` 4⟩, encontram-se as que criamos para representar conjuntos de terminais e florestas de Steiner, as quais vamos apresentar agora. Cada conjunto de terminais é representado por uma estrutura do tipo **TermSet**. O campo *id* desta estrutura guarda um número inteiro entre 0 e $|\mathcal{R}| - 1$ que identifica de maneira única o conjunto de terminais que ela representa ($\mathcal{R}$, como antes, é a família de conjuntos de terminais especificada na entrada). Diremos que tal número é o identificador do conjunto de terminais. Um outro campo, *connected*, indica se todos os vértices do conjunto estão ou não conectados através das arestas de uma dada floresta do grafo (*connected* $\equiv 1$ ou *connected* $\equiv 0$, respectivamente). A estrutura contém ainda um campo *vertices*, o qual guarda um apontador para uma lista ligada dos vértices que fazem parte do conjunto de terminais, e um campo *num_vertices* que guarda o número de elementos nesta lista. Para percorrer a lista *vertices*, devemos utilizar o campo *next_terminal* de cada vértice, o qual guarda um apontador para o próximo vértice na lista. Por fim, o campo *next_tset* da estrutura é um apontador para

uma outra estrutura do tipo **TermSet** e é utilizado para a criação de listas ligadas de conjuntos de terminais. Abaixo, segue a especificação em C da estrutura que acabamos de descrever:

4 #**define** *next_terminal* *u.V*

⟨Data structures of `sf.c` 4⟩ ≡

```
typedef struct ts_struct {
  int id;
  Vertex *vertices;
  int num_vertices;
  int connected;
  struct ts_struct *next_tset;
} TermSet;
```

Veja também blocos 6, 11, 46, 47, 183 e 186.

Este código é citado no bloco 4.

Este código é usado no bloco 1.

Em nossas implementações, suporemos que cada um dos vértices do grafo possui um campo *termset_id* que guarda ou o identificador do conjunto de terminais ao qual o vértice pertence, ou -1, caso o vértice não esteja em nenhum dos conjuntos de $\mathcal{R}$.

5 #**define** *termset_id* *v.I*

A estrutura de dados que representa uma floresta de Steiner é apresentada abaixo. Cada uma das funções correspondentes às implementações que apresentaremos a partir do próximo capítulo devolverá um apontador para uma estrutura deste tipo. O campo *edges* da estrutura é um apontador para uma lista ligada contendo as arestas da floresta. Para percorrer esta lista, deve-se utilizar o campo *next_edge* de cada aresta, o qual guarda um apontador para a próxima aresta na lista. O custo da floresta é guardado no campo *cost*, enquanto que o valor da função objetivo dual se encontra no campo *dual_cost* da estrutura.

6 #**define** *next_edge* *b.A*

⟨Data structures of `sf.c` 4⟩ +≡

```
typedef struct sf_struct {
  Arc *edges;
  long cost;
  double dual_cost;
} SteinerForest;
```

Como vimos na seção anterior, cada estrutura do tipo **Arc** possui apenas dois utility fields. Por outro lado, veremos que, nas implementações que apresentaremos a partir do próximo capítulo, será necessário armazenar, em uma estrutura deste tipo, mais informação do que é possível armazenar usando apenas estes dois campos. Em particular, numa dessas implementações, o mesmo problema irá ocorrer também com as estruturas do tipo **Vertex**. Para contornar esta situação, criamos a estrutura **ExtraFields**, que apresentamos abaixo, e usamos um dos utility fields dos arcos do grafo para guardar um apontador para uma estrutura deste tipo. Quando necessário, o mesmo é feito com os vértices. Renomearemos os utility fields que guardarão estes apontadores nas estruturas **Arc** e **Vertex** para *afields* e *vfields*, respectivamente. Além disso, utilizaremos as macros `ARC_FIELDS`(A) e `VERTEX_FIELDS`(V) para acessar o conteúdo do campo *afields* de um arco A e o conteúdo do campo *vfields* de um vértice V. Guardaremos todas estas definições em um arquivo-cabeçalho, o arquivo `extra.h`, para que elas possam ser utilizadas também por funções que não se encontram em `sf.c`.

Cada estrutura do tipo **ExtraFields** possui oito campos do tipo **extra**, um **union** que pode armazenar três tipos de dados diferentes: ou um apontador (". P"), ou número em ponto flutuante de precisão dupla (". D"), ou um inteiro (". I").

7 ⟨`extra.h` 7⟩ ≡

```
typedef union {
  void *P;
  double D;
  long I;
} extra;
typedef struct extra_fields_struct {
  extra e1, e2, e3, e4, e5, e6, e7, e8;
} ExtraFields;
#define afields b.S
#define ARC_FIELDS(A) ((ExtraFields *)((A)→afields))
#define vfields x.S
#define VERTEX_FIELDS(V) ((ExtraFields *)((V)→vfields))
```

Este código é citado no bloco 2.

8 ⟨Header files of `sf.c` 2⟩ +≡

```
#include "extra.h"
```

Na próxima seção, veremos algumas outras estruturas de dados, de propósito mais geral que estas, bastante úteis às nossas implementações.

1.7 Heaps de Fibonacci, árvores AVL e union-find

Dentre os arquivos-cabeçalho que se encontram em ⟨Header files of sf.c 2⟩, estão aqueles correspondentes a estruturas de dados que são utilizadas em nossas implementações, tais como heaps de Fibonacci, árvores binárias balanceadas de busca (mais especificamente, árvores AVL) e uma estrutura de dados para a manipulação de conjuntos disjuntos usualmente chamada de union-find. Para implementar estas estruturas de dados, nos baseamos nas implementações que se encontram descritas em [6, 26].

9 ⟨Header files of sf.c 2⟩ +≡

```
#include "fibheap.h"
#include "intersection.h"
#include "bbst.h"
#include "uf.h"
```

Vamos descrever a seguir cada uma das operações que manipulam todas estas estruturas de dados, informando também a complexidade de tempo de cada uma delas.

Heaps de Fibonacci

As funções para a criação e manipulação de heaps de Fibonacci se encontram todas prototipadas no arquivo `fibheap.h`. Utilizaremos heaps para armazenar mais de um tipo de dados e, por este motivo, os elementos que são manipulados em nossa implementação de heaps de Fibonacci são apontadores do tipo "**void** $*$". Assim, por exemplo, podemos ter heaps de arestas, heaps de vértices e até heaps de heaps. Cada heap é representado por uma estrutura do tipo **FibHeap** e é composto por nós. Cada um desses nós é responsável por armazenar um único elemento do heap e é representado por uma estrutura do tipo **FHnode**. Abaixo, descrevemos cada uma das funções que manipulam estas estruturas.

- **int** *FHEAPalloc*(**int** *num_heaps*, **int** *num_elems*)

  Descrição: aloca o espaço necessário para a criação de *num_heaps* heaps capazes de armazenar, no total, *num_elems* elementos. Além disso, organiza, em listas livres, todas as estruturas que foram alocadas. Devolve 1, se a alocação do espaço for realizada com sucesso, e devolve 0, caso contrário.

  Complexidade: $O(num_heaps + num_elems)$;

- **FibHeap** *FHEAPinit*(**int**($*less$)(**void** $*$, **void** $*$))

Descrição: cria um heap, inicialmente vazio, e devolve o heap criado. A função *less* recebida como argumento é usada para comparar elementos do heap com relação às suas chaves. Aqui, adotaremos a convenção de que Λ é um elemento cuja chave é mínima. Assim, *less* deve ser tal que uma chamada do tipo $less(a, b)$ devolva 1 caso $a \equiv \Lambda \neq b$ ou a chave de a seja menor que a chave de b, e devolva 0 caso $b \equiv \Lambda$ ou a chave de a seja maior ou igual à chave de b.

É necessário que os heaps manipulados pelas funções que seguem abaixo tenham sido criados através de uma chamada à função *FHEAPinit*.

Complexidade: $O(1)$;

- **int** *FHEAPempty*(**FibHeap** H)

  Descrição: Caso o heap H esteja vazio, devolve 1; caso contrário, devolve 0.

  Complexidade: $O(1)$;

- **int** *FHEAPnum*(**FibHeap** H)

  Descrição: devolve o número de elementos no heap H.

  Complexidade: $O(1)$;

- **FHnode** *FHEAPinsert*(**FibHeap** H, **void** $*a$)

  Descrição: insere a no heap H e devolve o nó de H em que a foi inserido.

  Complexidade: $O(1)$;

- **void** $*$*FHEAPfindMin*(**FibHeap** H)

  Descrição: devolve um elemento cuja chave seja mínima no heap H.

  Complexidade: $O(1)$;

- **void** *FHEAPdelMin*(**FibHeap** H)

  Descrição: remove, do heap H, um elemento cuja chave seja mínima em H.

  Complexidade amortizada: $O(\lg n)$, onde n é o número de elementos em H;

- **void** *FHEAPdecreaseKey*(**FibHeap** H, **FHnode** x, **void** $*a$)

  Descrição: Substitui, por a, o elemento que se encontra no nó x do heap H caso a chave de a seja menor ou igual à chave deste elemento. Além disso, reposiciona x no heap se necessário.

  Complexidade amortizada: $O(1)$;

- **void** $*$*FHEAPdelete*(**FibHeap** H, **FHnode** x)

Descrição: remove, do heap H, o elemento que se encontra no nó x de H e devolve o elemento removido.

Complexidade amortizada: $O(\lg n)$, onde n é o número de elementos no heap H;

- **void** *FHEAPtraverse*(**FibHeap** H, **void**(**func*)(**void** **item*, **void** **args*[]), **void** **args*[])

  Descrição: aplica a função *func*, com os argumentos que se encontram no vetor *args*, a cada elemento *item* do heap H.

  Complexidade: $O(n)$ vezes a complexidade de pior caso da função *func*, onde n é o número de elementos em H;

- **FibHeap** *FHEAPjoin*(**FibHeap** *h1*, **FibHeap** *h2*)

  Descrição: realiza a união dos heaps *h1* e *h2* e devolve o heap resultante da união.

  Complexidade: $O(1)$;

- **void** *FHEAPdestroy*(**FibHeap** H)

  Descrição: devolve às listas livres toda a memória ocupada pelo heap H e por seus nós para que, deste modo, esta memória possa ser usada futuramente.

  Complexidade: $O(n)$, onde n é o número de elementos em H;

- **void** *FHEAPfree*( )

  Descrição: libera toda a memória previamente alocada pela última chamada feita à função *FHEAPalloc*.

  Complexidade: $O(1)$.

Árvores AVL

Cada árvore é representada por uma estrutura do tipo **BalBST** e os elementos que armazenaremos nos nós dessas árvores são do tipo **Intersection**, que se encontra definido no arquivo-cabeçalho `intersection.h`. Mais adiante, no capítulo 4, apresentaremos a definição deste tipo de dados. O arquivo `bbst.h` contém o protótipo das funções responsáveis pela criação e manipulação de nossas árvores. Segue, abaixo, a descrição de cada uma dessas funções.

- **int** *BBSTalloc*(**int** n);

  Descrição: aloca o espaço necessário para a criação de n nós. Além disso, organiza, em uma lista livre, todas as estruturas alocadas. Devolve 1, se a alocação do espaço for realizada com sucesso, e devolve 0, caso contrário.

Complexidade: $O(n)$;

- **BalBST** *BBSTinit*( )

  Descrição: cria e devolve uma árvore inicialmente vazia.

  Complexidade: $O(1)$;

- **BalBST** *BBSTinsert*(**BalBST** r, **Intersection** x, **Intersection** $*y$)

  Descrição: recebe uma árvore r, um objeto x do tipo **Intersection** e um apontador y para uma variável do tipo **Intersection** e, se r não possui um elemento com a mesma chave de x, insere x em r, devolvendo, ao final, a árvore resultante e alterando para Λ o conteúdo da variável apontada por y; caso r já possua um elemento com a mesma chave de x, x não é inserido em r e a variável apontada por y passa a guardar o elemento de r que possui a mesma chave de x.

  Complexidade: $O(\lg n)$;

- **void** *BBSTtraverse*(**BalBST** r, **void**($*func$)(**Intersection**, **void** $*args$[ ]), **void** $*args$[ ])

  Descrição: aplica, para cada elemento da árvore r, a função *func* com os argumentos fornecidos no vetor *args*.

  Complexidade: $O(n)$ vezes a complexidade de pior caso da função *func*, onde n é o número de elementos em r;

- **void** *BBSTdestroy*(**BalBST** r)

  Descrição: devolve à lista livre toda a memória ocupada pela árvore r, de modo que esta memória possa ser usada futuramente.

  Complexidade: $O(n)$, onde n é o número de elementos em r;

- **void** *BBSTfree*( )

  Descrição: desaloca toda a memória previamente alocada pela última chamada à função *BBSTalloc*.

  Complexidade: $O(1)$.

Union-find

As funções para a manipulação de conjuntos disjuntos se encontram prototipadas no arquivo `uf.h`. Segue, abaixo, a descrição de cada uma delas.

- **void** *UFinit*(**Vertex** $*vertices$, **int** n)

Descrição: cria uma partição, em n partes unitárias, do conjunto dos vértices que se encontram nas posições de 0 a $n-1$ do vetor *vertices*. A cada conjunto da partição sempre estará associado um *representante*, o qual é um dos vértices do conjunto. O representante de cada conjunto após a chamada a esta função é, obviamente, o seu único elemento.

- **Vertex** $*$*UFfind*(**Vertex** $*u$)

  Descrição: recebe um vértice u e devolve o representante do conjunto ao qual u pertence na partição.

- **void** *UFunion*(**Vertex** $*u$, **Vertex** $*v$)

  Descrição: realiza a união dos conjuntos cujos representantes são u e v. Para representante do conjunto resultante da união é escolhido o representante do maior dos dois conjuntos. Como nossa intenção é manter uma partição dos vértices, os conjuntos cujos representantes eram u e v antes da chamada a esta função são substituídos em nossa coleção de conjuntos pelo resultado de sua união.

Qualquer sequência de m chamadas formada por uma chamada inicial a *UFinit* seguida de $m-1$ chamadas a *UFunion* e *UFfind* consome, no pior caso, tempo $O((m+n)\alpha(m+n,n))$, onde n é o argumento que foi passado na chamada a *UFinit* e $\alpha(x,y)$ é a inversa da função de Ackermann [6]. O crescimento desta função é extremamente lento, de forma que, para valores práticos de x e y, temos que $\alpha(x,y) \leq 4$.

Capítulo 2

Implementação de Goemans e Williamson

Uma forma bem simples e direta de implementar o passo correspondente à linha 5 do algoritmo *MinFS*, isto é, a escolha da aresta que será incluída na floresta corrente em uma dada iteração, é examinar, uma a uma, todas as arestas do grafo G. Para cada aresta a, seria preciso testar se a é uma aresta externa e, neste caso, calcular o valor da folga de a, atualizando, se necessário, a variável que guarda uma aresta que possui o menor valor de folga dentre as arestas que foram examinadas até o momento na iteração corrente; após examinarmos todas as arestas de G, a aresta guardada nesta variável seria incluída na floresta corrente. Poderíamos implementar as componentes da floresta corrente usando a estrutura de dados para conjuntos disjuntos que vimos no capítulo anterior e, desta forma, verificar se uma aresta é externa levaria tempo amortizado $O(\alpha(m,n))$, onde $m = |E_G|$ e $n = |V_G|$. Considerando que podemos calcular a folga de uma aresta em tempo constante, cada execução deste passo levaria, no total, tempo $O(m\alpha(m,n))$.

Quando o grafo dado no problema é denso (ou seja, $m = \Theta(n^2)$), isto seria bastante custoso, pois, no pior caso (que ocorre, por exemplo, quando $\mathcal{R} = \{V_G\}$), as execuções deste passo consumiriam, ao todo, tempo $\Omega(n^3)$, tornando este passo a parte do algoritmo que consome mais tempo no total. A implementação proposta por Goemans e Williamson [13] é bastante adequada nesta situação. Ela utiliza uma fila de prioridades para reduzir o número de arestas examinadas neste passo do algoritmo, consumindo, no total, tempo $O(n^2 \log n)$.

2.1 Descrição da implementação

Tal fila de prioridades, a qual denotaremos por H, é implementada através de um heap de mínimo e é utilizada para armazenar arestas externas com relação à floresta da iteração corrente.

Para cada par (não-orientado) $\{S, S'\}$ de componentes distintas da floresta corrente, é guardada no máximo uma única aresta em H: a aresta guardada é uma das que apresentam a menor folga dentre todas as arestas com um extremo em S e o outro em S'. Note que as demais arestas que ligam S a S' não precisam ser levadas em consideração para a determinação de uma aresta externa de folga mínima no grafo G. A chave de cada uma das arestas que se encontram em H é definida como sendo igual à sua folga. Desta forma, como H é um heap de mínimo, podemos determinar, em tempo constante, uma aresta externa de folga mínima em G.

Para o cálculo da folga das arestas, utilizaremos a seguinte estratégia. Ao invés de armazenar explicitamente o vetor y, que pode ter um número exponencial de componentes, guardaremos, para cada vértice v do grafo, o número $d(v) = \sum_{S:v\in S} y_S$. Assim, a folga de uma aresta externa uv passa a ser calculada da seguinte forma:

$$folga(uv) = \begin{cases} \frac{c_{uv}-d(u)-d(v)}{2} & \text{se } u \text{ e } v \text{ pertencem a componentes ativas} \\ c_{uv}-d(u)-d(v) & \text{se } u \text{ ou } v\text{, mas não ambos, pertence a uma} \\ & \text{componente ativa} \\ \infty & \text{se } u \text{ e } v \text{ pertencem a componentes inativas.} \end{cases}$$

Inicialmente, $d(v) = 0$ para todo v e, em cada iteração, incrementaremos $d(v)$, para cada v em uma componente ativa, com o valor da folga da aresta que será incluída na floresta. Note que, desta forma, em cada iteração, gastaremos tempo $O(n)$ com a atualização dos valores de $d(v)$.

Após a determinação de uma aresta de folga mínima no grafo G, iremos incluir tal aresta na floresta corrente. Note que, após a inclusão desta nova aresta, as componentes que contêm os seus extremos terão se unido para formar uma nova componente na floresta resultante da inclusão. Deste modo, é interessante que possamos determinar de maneira eficiente as componentes em que se encontram os extremos de uma dada aresta e, ao mesmo tempo, poder realizar de maneira rápida a união dos conjuntos de vértices de quaisquer duas componentes da floresta. Para isto, iremos utilizar a estrutura de dados para conjuntos disjuntos que vimos no capítulo anterior, a qual permite que cada uma destas duas operações sejam realizadas em tempo amortizado $O(\alpha(n, n))$, considerando que, no pior caso, $O(n)$ destas operações serão realizadas neste passo ao longo de toda a execução do algoritmo.

O próximo passo, agora, é determinar se a nova componente é ativa ou não. Para realizar esta tarefa, iremos utilizar a seguinte estratégia. A cada componente S da floresta corrente, estará associado um vetor $I(S)$ indexado por $\mathcal{R}$, que irá armazenar, para cada T em $\mathcal{R}$, o número de elementos na intersecção de T com S (sempre que nos referirmos a uma componente S, deve-se interpretar que estamos nos referindo à componente da floresta corrente cujo conjunto de vértices é S). Assim, dadas duas componentes S_1 e S_2, tudo o que temos que fazer é calcular, para cada T em $\mathcal{R}$, o valor de $I(S_1 \cup S_2)[T]$, o qual é igual a $I(S_1)[T] + I(S_2)[T]$; caso $0 < I(S_1 \cup S_2)[T] < |T|$ para algum T em $\mathcal{R}$, a componente $S_1 \cup S_2$ será ativa e, caso contrário, será inativa. Desta forma,

levaremos tempo $O(|\mathcal{R}|)$, em cada iteração, para decidir se a componente formada pela inclusão de uma nova aresta é ativa ou não. Note que, como estamos supondo que os conjuntos em $\mathcal{R}$ são dois a dois disjuntos, sabemos que $|\mathcal{R}| \leq n$ e, assim, todas as execuções deste passo irão consumir, no pior caso, tempo $O(n^2)$ no total.

Por fim, é necessário atualizar o heap H. Para isto, devemos inicialmente remover de H a aresta incluída na floresta na iteração corrente. Agora, sejam S_1 e S_2 as componentes que contêm os extremos desta aresta e seja f o valor de sua folga. Observe que se a é uma aresta externa cujos extremos não se encontram em S_1 e nem em S_2 e tal que ao menos um de seus extremos se encontra em uma componente ativa, então, ao final da iteração corrente, o novo valor da folga de a será igual ao valor de sua folga no início da iteração menos f. Além disso, se os extremos de a não se encontram em S_1, nem em S_2 e ambos pertencem a componentes inativas, a folga de a permanece inalterada (e igual a ∞). Logo, se a e a' são arestas externas cujos extremos não se encontram em S_1 e nem em S_2, então, se $folga(a) \leq folga(a')$ no início da iteração corrente, esta desigualdade permanece inalterada ao final da iteração. Por outro lado, se uma dessas duas arestas tem um dos extremos em S_i, para algum i em $\{1, 2\}$, a desigualdade pode deixar de valer ao final da iteração. Mais especificamente, a desigualdade pode não mais valer ao final da iteração corrente se S_i é ativa, mas $S_1 \cup S_2$ é inativa, ou se S_i é inativa (e, conseqüentemente, $S_1 \cup S_2$ é ativa). Portanto, apenas as arestas com algum extremo em S_1 ou S_2 precisam ser reposicionadas em H, pois podem estar violando a condição do heap. Isso é feito da seguinte maneira. Inicialmente, cada uma das arestas de H com algum extremo em S_1 ou S_2 é removida do heap. Diremos, no que segue, que duas componentes distintas da floresta são *adjacentes* se existe ao menos uma aresta do grafo com um extremo em cada uma destas componentes. Agora, o que precisamos fazer é determinar, para cada componente S adjacente a S_1 ou a S_2 e diferente de S_1 e de S_2, a aresta que será inserida no heap H, a qual deverá ter folga menor ou igual à folga de cada uma das arestas que ligam S a $S_1 \cup S_2$. Se S é adjacente a S_1, mas não é adjacente a S_2, é claro que basta tomar a aresta com um extremo em S e o outro em S_1 que se encontrava em H. O caso em que S é adjacente a S_2, mas não é adjacente a S_1 é análogo. Resta, agora, tratar o caso em que S é adjacente a ambas, S_1 e S_2. Neste caso, para i em $\{1, 2\}$, seja e_i a aresta com um extremo em S e o outro em S_i que se encontrava em H. Se $folga(e_1) \leq folga(e_2)$, basta tomar e_1 para fazer parte de H; caso contrário, é suficiente escolher e_2. Como cada componente é adjacente a no máximo $n - 1$ componentes da floresta, ao final de todo este processo, teremos executado $O(n)$ remoções e inserções no heap H e, portanto, para atualizar H, levaremos ao todo tempo $O(n \log n)$ por iteração. Logo, no pior caso, o tempo gasto em todas as execuções deste passo é $O(n^2 \log n)$.

Todas as demais operações do algoritmo podem ser implementadas de modo a consumir tempo $O(n^2)$ no total e, deste modo, a complexidade de tempo da implementação de Goemans e Williamson é $O(n^2 \log n)$. Na próxima seção, iremos apresentar uma implementação em CWEB

dos procedimentos aqui descritos, explicitando alguns detalhes subjacentes à descrição, que demos acima, da implementação.

2.2 Implementação em CWEB

Vamos examinar, a partir de agora, a função que implementa a estratégia de Goemans e Williamson para a construção de uma floresta de Steiner. A função *sf_gw*, que apresentamos abaixo, recebe como argumentos um grafo g simples (ou seja, sem laços e sem arestas paralelas) e um vetor *term_sets*, que se encontra indexado de 0 a g→*num_ts* − 1 e cujos elementos são conjuntos de terminais, e devolve uma floresta de Steiner do grafo g. Caso não exista uma tal floresta, a função devolve Λ.

```
10 #define num_ts  uu.I
⟨Steiner forest construction functions 10⟩ ≡
  SteinerForest *sf_gw(Graph *g, TermSet *term_sets)
  {
    ⟨Local variables of sf_gw 14⟩
    ⟨Create a spanning forest sf with no edges 13⟩
    ⟨Initialize d(v) for each vertex v of the graph 18⟩
    ⟨Create a heap that contains the incident edges of each component 25⟩
    ⟨While there is some active component include an edge in the forest sf 30⟩
    ⟨Drop out unnecessary edges from sf and return a minimal Steiner forest 43⟩
  }
```

Veja também blocos 45, 167 e 187.

Este código é usado no bloco 1.

Antes de exibirmos o código de cada uma das seções em que a função *sf_gw* se divide, iremos descrever a estrutura de dados que usaremos para representar cada uma das componentes de uma floresta, a qual é uma estrutura do tipo **GWComponent**.

Dada uma floresta, cada uma de suas componentes é identificada de maneira única através de um número inteiro entre 0 e $n-1$ que fica armazenado no campo *id* da estrutura. Um outro campo, *active*, é responsável pelo armazenamento de um inteiro que indica se a componente é ou não ativa: se *active* $\equiv 0$, então a componente representada pela estrutura é inativa e, se *active* $\equiv 1$, então tal componente é ativa. Os vértices que fazem parte da componente ficam todos armazenados em uma lista ligada. O campo *vertices* guarda um apontador para o primeiro vértice desta lista e, para cada vértice v na lista, v→*next_vertex* é um apontador para o vértice

que sucede v na lista. O campo *num_vertices* armazena o número de vértices da componente e o campo *ts_intersect_size* é um vetor que armazena em *ts_intersect_size*[i] o número de vértices da componente que fazem parte do conjunto de terminais cujo identificador é i. Também guardaremos em uma lista algumas das arestas externas que incidem na componente. Mais precisamente, dada uma componente S, armazenaremos na lista de arestas correspondentes a S, para cada componente S' adjacente a S, uma aresta que possua folga mínima dentre todas as arestas com um extremo em S e outro em S'. Tal lista é implementada através de uma lista circular duplamente ligada com cabeça e o campo *edges* da estrutura que representa S guardará um apontador para o nó-cabeça desta lista. Cada estrutura do tipo **GWComponent** contém ainda campos *prev* e *next* que são utilizados para guardar apontadores para outras estruturas do tipo **GWComponent**. Durante toda a execução da fase iterativa do algoritmo, iremos manter as estruturas que representam as componentes da floresta corrente em uma lista duplamente ligada. Os campos *prev* e *next* de cada estrutura C nesta lista guardarão, respectivamente, apontadores para a estrutura que antecede e para a estrutura que sucede C na lista.

11 **#define** *next_vertex* *y.V*

```
⟨Data structures of sf.c 4⟩ +≡
  typedef struct gw_component {
    int id;
    int active;
    Vertex *vertices;
    long num_vertices;
    int *ts_intersect_size;
    Arc *edges;
    struct gw_component *prev;
    struct gw_component *next;
  } GWComponent;
```

Inicialmente, iremos armazenar, no campo x de cada vértice V do grafo, um apontador C para a estrutura que representa a componente à qual V pertence e, para isto, utilizaremos a macro SET_COMPONENT(V, C). Ao longo das iterações, conforme inserimos novas arestas na floresta, a componente de V eventualmente irá mudar e, conseqüentemente, o conteúdo do campo x de V ficará desatualizado. Por uma questão de economia de tempo, ao invés de atualizarmos o conteúdo deste campo toda vez que a componente de V muda, utilizaremos (como será visto adiante) a estrutura de dados para conjuntos disjuntos que mencionamos no capítulo 1. A macro COMPONENT(V) que definimos abaixo é utilizada (como veremos, não somente nesta implementação) para determinar a componente à qual um dado vértice V pertence.

12 **#define** SET_COMPONENT(V, C) $((V)\to x.S = (\textbf{char} *)(C))$

```
#define COMPONENT(V) ((UFfind(V))→x.S)
```

O trecho a seguir implementa a criação da floresta geradora inicial, a qual não possui arestas (passo correspondente a linha 1 do algoritmo *MinFS*). Durante toda a execução da fase iterativa do algoritmo, a floresta corrente é representada através de uma estrutura do tipo **Graph** e a variável *sf* guarda um apontador para esta estrutura. Além disso, as estruturas que representam as componentes da floresta corrente são mantidas em uma lista circular duplamente ligada com cabeça. A variável *components* guarda um apontador para o nó-cabeça desta lista.

```
13 ⟨Create a spanning forest sf with no edges 13⟩ ≡
   {
     int i;
     GWComponent *t;
     sf = gb_new_graph(g→n);
     components = t = malloc((g→n + 1) * sizeof(GWComponent));
     if (¬sf ∧ ¬components) { err = 2; return Λ; }
     ⟨Some initializations 15⟩
     for (i = 0; i < g→n; i++) {
       Vertex *v = g→vertices + i;
       C = components + i + 1;
       ⟨Initialize some fields of the component C of v 16⟩
       t→next = C;
       C→prev = t;
       t = t→next;
     }
     t→next = components;
     components→prev = t;
     UFinit(g→vertices, g→n);
   }
```

Este código é usado no bloco 10.

```
14 ⟨Local variables of sf_gw 14⟩ ≡
     Graph *sf;
     GWComponent *components, *C;
```

Veja também blocos 17, 19, 23, 26, 31 e 39.

Este código é usado no bloco 10.

Neste bloco, vamos inicializar algumas das variáveis locais de *sf_gw*. A variável *num_actives* é usada para guardar o número de componentes ativas na floresta corrente. O vetor *last* guarda, em *last*[*id*], o último vértice na lista de vértices da componente cujo identificador é *id* e é usado para auxiliar a concatenação das listas de vértices de duas componentes no momento de sua união. A variável *sizes* guarda um apontador para o espaço que utilizaremos para armazenar todos os vetores acessados a partir do campo *ts_intersect_size* de alguma componente.

15 ⟨ Some initializations 15 ⟩ ≡

```
num_actives = 0;
last = malloc(g→n * sizeof(Vertex *));
sizes = malloc((g→n) * (g→num_ts) * sizeof(int));
if (¬last ∨ ¬sizes) {
  err = 2;
  return Λ;
}
```

Veja também blocos 22 e 38.

Este código é citado no bloco 2.

Este código é usado nos blocos 13 e 52.

16 ⟨ Initialize some fields of the component C of v 16 ⟩ ≡

```
{
  int j;
  if (¬(v→arcs)) {
    if (v→termset_id ≥ 0) { err = 1; return Λ; }
    continue;
  }
  C→id = i;
  SET_COMPONENT(v, C);
  C→vertices = last[C→id] = v;
  v→next_vertex = Λ;
  C→num_vertices = 1;
  C→ts_intersect_size = sizes + i * (g→num_ts);
  for (j = 0; j < g→num_ts; j++) C→ts_intersect_size[j] = 0;
  if (v→termset_id ≥ 0) {
    C→active = 1;
    num_actives++;
    C→ts_intersect_size[v→termset_id]++;
  }
```

```
  else C→active = 0;
}
```

Este código é usado nos blocos 13 e 53.

17 ⟨Local variables of *sf_gw* 14⟩ +≡

```
long num_actives;
Vertex **last;
int *sizes;
```

Aqui é feita a inicialização de $d(v)$ para todo vértice v do grafo. Observe que $d(v)$ pode assumir valores que não são inteiros e, deste modo, não podemos utizar os *utility fields* de v para guardar o valor de $d(v)$. Por este motivo, é necessário alocar espaço extra, o qual poderá ser acessado através do vetor *d_value*.

18 #define $d(v)$ $*((\textbf{double} *)(v)\to z.S)$

⟨Initialize $d(v)$ for each vertex v of the graph 18⟩ ≡

```
{
  int i;
  if (¬(d_value = malloc(g→n * sizeof(double)))) {
    err = 2; return Λ;
  }
  for (i = 0; i < g→n; i++) {
    Vertex *v = g→vertices + i;
    v→z.S = (char *)(d_value + i);
    d(v) = 0;
  }
}
```

Este código é usado nos blocos 10 e 45.

19 ⟨Local variables of *sf_gw* 14⟩ +≡

```
double *d_value;
```

Apresentamos abaixo as funções que utilizamos para inserir e remover arestas na lista $C\to edges$ de alguma componente C da floresta corrente. Caso uma aresta a se encontre em uma tal lista, utilizaremos as macros $prev(a)$ e $next(a)$, que definimos abaixo, para acessar os campos de a que

guardam apontadores, respectivamente, para a aresta que precede e para aquela que sucede a na lista. A função *insert_edge* recebe duas arestas, e e l, e insere e na lista circular duplamente ligada em que l se encontra, de modo que e suceda l na lista. A função *delete_edge* recebe uma aresta e e a remove da lista circular duplamente ligada em que ela se encontra.

20
```
#define next(a)  (ARC_FIELDS(a)→e1.P)
#define prev(a)  (ARC_FIELDS(a)→e2.P)
⟨Auxiliary functions 20⟩ ≡
  void insert_edge(Arc *l, Arc *e)
  {
    Arc *next = next(l);
    next(e) = next;
    prev(next) = e;
    prev(e) = l;
    next(l) = e;
  }
  void delete_edge(Arc *e)
  {
    Arc *prev = prev(e);
    Arc *next = next(e);
    next(prev) = next;
    prev(next) = prev;
  }
```

Veja também blocos 27, 29, 51, 59, 64, 65, 69, 84, 91, 176, 180, 181, 184, 185, 202, 204, 211, 259 e 260.

Este código é usado no bloco 1.

21
```
⟨list.h 21⟩ ≡
  void insert_edge(Arc *, Arc *);
  void delete_edge(Arc *);
```

A alocação dos nós-cabeça das listas de arestas externas incidentes nas componentes da floresta inicial é feita abaixo e o vetor *head* é usado para guardar estes nós. Pelo modo como definimos as macros $next(a)$ e $prev(a)$, será necessário alocar também uma estrutura do tipo **ExtraFields** para cada um desses nós-cabeça. Usaremos o vetor *ef* para guardar estas estruturas.

22
```
⟨Some initializations 15⟩ +≡
  head = malloc(g→n * sizeof(Arc));
```

```
ef = malloc(g→n * sizeof(ExtraFields));
if (¬head ∨ ¬ef) {
  err = 2;
  return Λ;
}
```

23 ⟨Local variables of *sf_gw* 14⟩ +≡

```
Arc *head;
ExtraFields *ef;
```

No trecho abaixo, é feita a construção da lista de arestas externas incidentes na componente C correspondente a um dado vértice v na floresta geradora inicial. Além disso, vamos aproveitar a ocasião para preencher, para cada aresta a nesta lista, o campo que usaremos para guardar o número de extremos de a que se encontram em uma componente ativa. Para acessar este campo, usaremos a macro $act(a)$ que definimos abaixo.

24 #define $act(a)$ (ARC_FIELDS(a)→$e4.I$)

⟨Construct the list *C→edges* 24⟩ ≡

```
{
  static int i = 0, j = 0;
  C→edges = head + j++;
  C→edges→afields = (char *)(ef + i++);
  prev(C→edges) = next(C→edges) = C→edges;
  for (a = v→arcs; a; a = a→next) {
    insert_edge(C→edges, a);
    act(a) = C→active;
    if (a→tip→termset_id ≥ 0) act(a)++;
  }
}
```

Este código é usado no bloco 25.

Aqui, é feita a construção do heap de arestas externas candidatas a serem incluídas na floresta geradora inicial. Usaremos a variável *edges* para guardar tal heap. É importante ressaltar que na implementação sugerida por Goemans e Williamson não é necessário que usemos um heap de Fibonacci e isso só é feito aqui por uma questão de conveniência (já que as demais implementações que veremos adiante exigem que este tipo de heap seja usado). A função *EDGEless* será

apresentada nos próximos blocos. Cada aresta a inserida em *edges* guardará, em um de seus campos, o nó do heap *edges* em que ela se encontra. Usaremos a macro $node(a)$ para acessar este campo.

```
25 #define node(a) (ARC_FIELDS(a)→e3.P)
   ⟨Create a heap that contains the incident edges of each component 25⟩ ≡
     if (¬FHEAPalloc(1, g→m)) { err = 2; return Λ; }
     edges = FHEAPinit(EDGEless);
     for (C = components→next; C ≠ components; C = C→next) {
       Vertex *v = C→vertices;
       Arc *a;
       ⟨Construct the list C→edges 24⟩
       for (a = v→arcs; a; a = a→next) {
         if (v < a→tip) node(a) = FHEAPinsert(edges, a);
         else {
           Arc *a_ = a − 1;
           node(a) = node(a_);
         }
       }
     }
```

Este código é usado no bloco 10.

```
26 ⟨Local variables of sf_gw 14⟩ +≡
     FibHeap edges;
```

Como vimos na seção anterior, a chave de cada aresta a no heap *edges* é dada pela folga de a. Usamos a função *key*, abaixo, para calcular a folga de uma aresta externa. Dada uma aresta externa a, $key(a)$ devolve o valor da folga de a com relação à floresta corrente. Por outro lado, caso $a \equiv \Lambda$, então $key(a)$ devolve -1. É importante observar que o valor da chave (folga) de uma aresta externa é sempre não-negativo e, por este motivo, $key(\Lambda) < key(a)$ para toda aresta externa a.

```
27 ⟨Auxiliary functions 20⟩ +≡
     double key(Arc *a)
     {
       if (¬a) return −1;
       if (¬act(a)) return DBL_MAX;      /* ∞ */
       return (((double) a→len) − d(a→from) − d(a→tip))/act(a);
     }
```

28 ⟨ Header files of sf.c 2 ⟩ +≡
```
#include <float.h>
```

A função abaixo é usada para comparar duas arestas, a e b, no heap *edges*. Caso $a \equiv \Lambda \neq b$ ou a chave de a seja menor que a chave de b, ela devolve 1 e, caso $b \equiv \Lambda$ ou a chave de a seja maior ou igual a chave de b, ela devolve 0.

29 ⟨ Auxiliary functions 20 ⟩ +≡
```
int EDGEless(void *a, void *b)
{
  Arc *e1 = a, *e2 = b;
  return (key(e1) < key(e2));
}
```

O trecho abaixo é o responsável pela construção iterativa de uma floresta de Steiner do grafo de entrada (passo correspondente às linhas de número 4 a 8 do algoritmo *MinFS*). Em cada iteração, uma aresta externa é incluída na floresta *sf* até que não existam mais componentes ativas, o que é detectado através da variável *num_actives*. Cada vez que uma aresta a é incluída na floresta, atualiza-se o valor de $d(v)$ para cada vértice v em uma componente ativa e o valor da função objetivo dual que fica armazenado na variável *dual_cost*. Além disso, é realizada a atualização do heap *edges* e a união das componentes C1 e C2 às quais pertencem, respectivamente, os vértices *a→from* e *a→tip*, extremos de a.

30 ⟨ While there is some active component include an edge in the forest *sf* 30 ⟩ ≡
```
while (num_actives > 0) {
  double inc;
  Arc *a, *a_;
  GWComponent *C1, *C2;
  ⟨ Let a be an edge with the smallest slackness inc 32 ⟩
  C1 = (GWComponent *) COMPONENT(a→from);
  C2 = (GWComponent *) COMPONENT(a→tip);
  if (¬(C1→active) ∧ ¬(C2→active)) { err = 1; return Λ; }
  a_ = inverse(a);
  ⟨ Include a in the forest sf 33 ⟩
  ⟨ For each vertex v in some active component, do d(v) += inc 34 ⟩
  dual_cost += inc * num_actives;
  FHEAPdelete(edges, node(a));
  delete_edge(a);
```

```
    delete_edge(a_);
    if (C1→active ∧ C2→active) num_actives —;
    ⟨ Merge the components C1 and C2 35 ⟩
  }
```

Este código é usado no bloco 10.

31 ⟨ Local variables of *sf_gw* 14 ⟩ += **double** *dual_cost* = 0;

32 ⟨ Let *a* be an edge with the smallest slackness *inc* 32 ⟩ ≡

```
  if (FHEAPempty(edges)) { err = 1; return Λ; }
  a = FHEAPfindMin(edges);
  inc = key(a);
```

Este código é usado no bloco 30.

Aqui é feita a inclusão, na floresta *sf*, da aresta *a* escolhida no passo anterior. Os arcos que representam tal aresta na floresta *sf* guardarão, no campo *original_arc*, um apontador para os arcos *a* e *a_* que representam a aresta no grafo *g*.

33 #**define** *original_arc* *a.A*

⟨ Include *a* in the forest *sf* 33 ⟩ ≡

```
  {
    Vertex *u, *v;
    u = sf→vertices + (a→from − g→vertices);
    v = sf→vertices + (a→tip − g→vertices);
    gb_new_edge(u, v, a→len);
    u→arcs→original_arc = a;
    v→arcs→original_arc = a_;
  }
```

Este código é usado nos blocos 30, 76 e 167.

34 ⟨ For each vertex *v* in some active component, do *d*(*v*) += *inc* 34 ⟩ ≡

```
  {
    for (C = components→next; C ≠ components; C = C→next)
      if (C→active) {
        Vertex *v;
```

```
        for (v = C→vertices; v; v = v→next_vertex) d(v) += inc;
    }
}
```

Este código é usado nos blocos 30 e 76.

A união das componentes C1 e C2 é implementada no trecho de código que apresentamos a seguir. É importante que realizemos o teste inicial para descobrir qual dessas duas é a maior componente em número de vértices. O motivo é que, após a operação *UFunion*(C1→*vertices*, C2→*vertices*) realizada abaixo, sempre que v for um vértice que se encontra na componente resultante da união de C1 e C2, a macro COMPONENT(v) devolverá um apontador para a estrutura que representa a maior componente dentre estas duas. Desta forma, iremos manter esta estrutura na lista de componentes da floresta e alterar o conteúdo de alguns de seus campos de modo que ela passe a representar a componente C1 ∪ C2. Ao final da execução do trecho abaixo, a estrutura correspondente à menor componente em número de vértices é removida da lista de componentes da floresta corrente.

```
35 #define exchange(A, B) { void *t = A; A = B; B = t; }
⟨Merge the components C1 and C2 35⟩ ≡
  if (C1→num_vertices < C2→num_vertices) exchange(C1, C2)
  ⟨Concatenate the lists C1→vertices and C2→vertices 36⟩
  ⟨Update C1→active 37⟩
  if (¬(C1→active)) num_actives—;
  ⟨Update the heap edges 40⟩
  UFunion(C1→vertices, C2→vertices);
  C2→next→prev = C2→prev;
  C2→prev→next = C2→next;
```

Este código é usado no bloco 30.

```
36 ⟨Concatenate the lists C1→vertices and C2→vertices 36⟩ ≡
  last[C1→id]→next_vertex = C2→vertices;
  last[C1→id] = last[C2→id];
  C1→num_vertices += C2→num_vertices;
```

Este código é usado nos blocos 35 e 79.

O trecho abaixo implementa o procedimento descrito na seção anterior para decidir se a componente resultante da união é ativa ou inativa.

```
37 ⟨Update C1→active 37⟩ ≡
   {
     int i;
     C1→active = 0;
     for (i = 0; i < g→num_ts; i++) {
       TermSet *ts = term_sets + i;
       if (ts→connected) continue;
       C1→ts_intersect_size[i] += C2→ts_intersect_size[i];
       if (C1→ts_intersect_size[i] > 0) {
         if (C1→ts_intersect_size[i] < ts→num_vertices) C1→active = 1;
         else ts→connected = 1;
       }
     }
   }
```

Este código é usado nos blocos 35 e 79.

Por fim, vamos mostrar como é implementada a atualização do heap *edges* e a construção da lista de arestas externas incidentes na componente C1 ∪ C2. Na implementação destes passos, utilizamos dois vetores auxiliares. O vetor *component_id* é utilizado para guardar os identificadores das componentes adjacentes à componente C1. Por outro lado, *adj* é um vetor cujas posições serão preenchidas da seguinte forma: para cada i, $adj[i]$ guardará uma aresta de folga mínima dentre aquelas que possuem um extremo em C1 e o outro na componente cujo identificador é i; caso a componente cujo identificador é i não seja adjacente a C1, $adj[i]$ guardará Λ.

```
38 ⟨Some initializations 15⟩ +≡
   adj = malloc(g→n * sizeof(Arc *));
   component_id = malloc(g→n * sizeof(int));
   if (¬adj ∨ ¬component_id) {
     err = 2; return Λ;
   }
   for (i = 0; i < g→n; i++) adj[i] = Λ;
```

```
39 ⟨Local variables of sf_gw 14⟩ +≡
   Arc **adj;
   int *component_id;
```

A atualização do heap *edges* e a construção da lista de arestas externas incidentes na componente C1 ∪ C2 são implementadas no trecho abaixo. Inicialmente, removeremos, do heap *edges*, as arestas externas incidentes em C1 e aquelas que são incidentes em C2. Ao mesmo tempo, atualizaremos o valor do campo *act* dessas arestas, preencheremos os vetores *adj* e *component_id* e, com o auxílio de *adj*, construiremos a lista de arestas externas incidentes na componente C1 ∪ C2, a qual ficará acessível a partir do nó-cabeça C1→*edges* (já que, após a execução do trecho abaixo, a estrutura correspondente à variável C1 irá representar a componente C1 ∪ C2). Por fim, iremos inserir, no heap *edges*, cada uma das arestas presentes na nova lista C1→*edges*.

```
40 ⟨Update the heap edges 40⟩ ≡
   {
     int i = 0;
     ⟨Delete from edges the edges in C1→edges 41⟩
     ⟨Do the same for C2→edges and merge the lists C1→edges and C2→edges 42⟩
     for (a = next(C1→edges); a ≠ C1→edges; a = next(a)) {
       Arc *a_ = inverse(a);
       node(a) = node(a_) = FHEAPinsert(edges, a);
     }
     while (i > 0) adj[component_id[—i]] = Λ;
   }
```

Este código é usado no bloco 35.

```
41 ⟨Delete from edges the edges in C1→edges 41⟩ ≡
   for (a = next(C1→edges); a ≠ C1→edges; a = next(a)) {
     C = (GWComponent *) COMPONENT(a→tip);
     FHEAPdelete(edges, node(a));
     adj[C→id] = a;
     component_id[i++] = C→id;
     act(a) = act(inverse(a)) = C1→active + C→active;
   }
```

Este código é usado no bloco 40.

```
42 ⟨Do the same for C2→edges and merge the lists C1→edges and C2→edges 42⟩ ≡
   for (a = next(C2→edges); a ≠ C2→edges; a = next(a)) {
     C = (GWComponent *) COMPONENT(a→tip);
     FHEAPdelete(edges, node(a));
     a_ = inverse(a);
```

```
    act(a) = act(a_) = C1→active + C→active;
    if (¬adj[C→id]) {
      Arc *t = prev(a);
      delete_edge(a);
      insert_edge(prev(C1→edges), a);
      a = t;
    }
    else {
      Arc *b = adj[C→id];
      if (key(a) < key(b)) {
        Arc *t = prev(a);
        delete_edge(inverse(b));
        delete_edge(a);
        insert_edge(prev(b), a);
        delete_edge(b);
        a = t;
      }
      else delete_edge(a_);
    }
  }
```

Este código é usado no bloco 40.

Ao final do processo iterativo descrito nos trechos anteriormente vistos, a variável *sf* estará guardando um apontador para uma floresta de Steiner do grafo *g*. No trecho de código abaixo, a função *edgePrunning* é executada sobre *sf* para determinar uma floresta de Steiner minimal contida na floresta correspondente a esta variável. A floresta encontrada é então devolvida como resposta pela função *sf_gw*. A operação implementada por *edgePrunning* corresponde à segunda fase do algoritmo de Goemans e Williamson e será vista com detalhes no capítulo 6.

43 ⟨Drop out unnecessary edges from *sf* and return a minimal Steiner forest 43⟩ ≡

```
  {
    SteinerForest *min_sf = edgePruning(sf, g, term_sets);
    min_sf→dual_cost = dual_cost;
    ⟨Free the memory allocated by sf_gw 44⟩
    return min_sf;
  }
```

Este código é usado no bloco 10.

44 ⟨ Free the memory allocated by *sf_gw* 44 ⟩ ≡

```
gb_recycle(sf);
free(sizes);
free(components);
free(ef);
free(head);
free(d_value);
free(adj);
free(component_id);
FHEAPfree();
```

Este código é usado no bloco 43.

Capítulo 3

Implementação de Gabow, Goemans e Williamson

No capítulo anterior, vimos que, durante a fase iterativa do algoritmo, podemos desconsiderar algumas das arestas do grafo, pois elas jamais serão escolhidas para serem incluídas na floresta. Mais especificamente, vimos que basta levar em conta uma única aresta e para cada par não-ordenado $\{S, S'\}$ de componentes distintas da floresta corrente, desde que escolhamos e de modo que todas as demais arestas com um extremo em S e o outro em S' tenham folga maior ou igual à folga de e. Seja I um conjunto formado por arestas e do grafo escolhidas desta forma. Veremos no lema abaixo que também é possível desconsiderar, por um certo número de iterações e sob uma determinada condição, algumas das arestas que se encontram em I. No que segue, iremos supor que as arestas se encontram totalmente ordenadas de acordo com a sua folga: caso existam duas arestas com a mesma folga, alguma numeração fixa das arestas, feita a priori, irá determinar qual delas é a menor aresta. Desta forma, a k-ésima menor aresta, por exemplo, é única.

Lema 3.1. *Seja uv uma aresta em I que tem um extremo na componente S_u e o outro na componente S_v e tal que, em uma dada iteração i, uv não se encontra entre as $2k$ menores arestas de I que possuem um extremo em S_u. Então, se S_u e S_v não sofrerem alterações, uv não será incluída em F antes que pelo menos k iterações do algoritmo sejam executadas.*

Prova. Considere uma iteração na qual uv é a aresta que será incluída na floresta. Agora, seja $u'v'$ uma aresta que, na iteração i, se encontrava entre as $2k$ menores arestas de I que possuem um extremo em S_u. Como S_u não sofreu alterações, sabemos que $u'v'$ não foi incluída na floresta. Logo, a folga de uv teve um decréscimo maior que o decréscimo na folga de $u'v'$ desde a iteração i (um aumento na folga corresponderia a um decréscimo "negativo"). Deste modo, como nem S_u e nem S_v sofreram alterações desde a iteração i, certamente a componente $S_{v'}$, que contém o extremo de $u'v'$ que não se encontra em S_u, sofreu alguma alteração decorrente de sua união com

uma outra componente da floresta. Por outro lado, devido ao modo como definimos I, sabemos que às $2k$ arestas $u'v'$ correspondem $2k$ componentes distintas $S_{v'}$. Logo, desde a iteração i, foram realizadas pelo menos k uniões de componentes da floresta (as uniões envolvendo as $2k$ componentes $S_{v'}$) e, portanto, como em cada iteração da primeira fase ocorre uma única união, temos que pelo menos k iterações foram executadas desde a iteração i. □

Gabow, Goemans e Williamson [9] sugeriram modificações na implementação que vimos no capítulo anterior que tiram proveito do resultado provado no lema acima, obtendo, com isto, uma implementação de complexidade $O(n(n + \sqrt{m \log \log n}))$. Uma delas é a de dividir a primeira fase do algoritmo em subfases e ignorar, ao longo de cada subfase, determinadas arestas do grafo. Além disso, eles utilizam a idéia de estruturas de dados chamadas de *pacotes*, devida a Gabow et al. [8], para organizar as arestas do grafo que são levadas em conta na determinação de uma aresta de folga mínima.

3.1 Descrição da implementação

A primeira fase do algoritmo é dividida em subfases de r iterações cada uma, onde r é um parâmetro cujo valor definiremos adiante. No início de cada subfase, para cada componente S, escolheremos as $2r$ menores arestas dentre aquelas que se encontram em I (onde I é definido como acima) e possuem exatamente um extremo em S. Seja uv uma aresta em I e sejam S_u e S_v as componentes em que se encontram os seus extremos. Então, uv é considerada *acordada* se ela se encontra entre as $2r$ menores arestas quando consideramos $S = S_u$ e se o mesmo ocorre quando consideramos $S = S_v$. Uma aresta que não for considerada como acordada será completamente ignorada durante a subfase, a menos que alguma das componentes em que se encontram os seus extremos seja unida a uma outra componente da floresta.

As arestas consideradas acordadas no início de uma subfase são organizadas em estruturas de dados chamadas *pacotes*. A cada componente S da floresta corresponderá um conjunto de pacotes onde ficarão armazenadas as arestas acordadas que possuem um extremo em S. Cada pacote tem capacidade para armazenar no máximo $\log n$ arestas e é implementado através de um heap de mínimo. Dentre os pacotes que correspondem a uma componente, existe um que é especial, o qual iremos chamar de *pacote em crescimento*. Os demais pacotes são chamados de *pacotes comuns*. Sempre que uma nova aresta deve ser armazenada nos pacotes de uma componente S, ela é inserida no pacote em crescimento de S. Quando o pacote em crescimento atinge o limite de sua capacidade (ou seja, $\log n$ arestas), ele se torna um pacote comum e um novo pacote em crescimento é criado para a componente.

Sempre que uma aresta é a menor aresta nos dois pacotes em que ela se encontra e estes pacotes são pacotes comuns, ela é chamada de *duplo mínimo*. Tais arestas, além de serem

mantidas nos pacotes das componentes que contêm seus extremos, ficarão armazenadas também em um heap de mínimo que denotaremos por D. Este heap é utilizado para que possamos determinar rapidamente a aresta mínima dentre aquelas que se encontram em pacotes comuns.

Após a inicialização das estruturas descritas acima, são executadas as r iterações da subfase. De acordo com a forma com que as estruturas estão definidas, a determinação de uma aresta de folga mínima em cada iteração é feita comparando-se o mínimo do heap D com o mínimo de cada um dos pacotes em crescimento. Uma vez que tal aresta é escolhida, ela e todas as arestas acordadas incidentes nas componentes que contêm os extremos dela são removidas dos pacotes em que estão armazenadas e também do heap D. Note que a remoção destas arestas dos pacotes em que elas estavam pode ter gerado novos duplos mínimos e, caso isto ocorra, deve-se atualizar D inserindo-se os duplos mínimos gerados. Após a inclusão na floresta da aresta escolhida, devemos decidir quais das arestas incidentes na nova componente gerada pela inclusão serão consideradas acordadas. Seja S a nova componente da floresta. Na seleção dessas arestas, devemos considerar como candidata apenas uma aresta para cada componente S' adjacente a S no grafo; a aresta candidata é a menor dentre aquelas que possuem um extremo em S e o outro em S'. Dentre estas arestas candidatas, as $2r$ menores serão declaradas acordadas. Então, é criado o pacote em crescimento de S e, uma a uma, cada aresta e, dentre essas $2r$, é inserida em dois pacotes: no pacote em crescimento de S e no pacote em crescimento da componente que contém o outro extremo de e. Sempre que a capacidade destes pacotes atinge seu limite, é criado um novo pacote em crescimento e o antigo se torna um pacote comum. Note que o surgimento de novos pacotes comuns através deste processo pode gerar novos duplos mínimos e, neste caso, o heap D deve ser devidamente atualizado. Após a execução de todos estes passos, tem início uma nova iteração.

Análise da complexidade de tempo

Note que, no início de cada subfase, o número de arestas que se encontram nos pacotes de uma componente S é limitado por $2r$. Além disso, note que, em cada uma das r iterações de uma subfase, no máximo uma aresta é adicionada aos pacotes de S. Assim, durante a execução de uma subfase, o número de arestas que se encontram nos pacotes de uma componente nunca ultrapassa $3r$. Deste modo, podemos concluir também que, durante a execução de uma subfase, o número de pacotes associados a uma componente nunca é maior que $\lceil 3r/\log n \rceil$. Por outro lado, observe que, como cada pacote contém no máximo $\log n$ arestas, o tempo gasto com a remoção de uma aresta de um pacote é $O(\log\log n)$ e, além disso, como o número de arestas no heap D nunca ultrapassa m, o tempo gasto com a remoção de uma aresta de D é $O(\log m) = O(\log n)$. Logo, em cada iteração, o tempo gasto com a remoção de arestas dos pacotes em que elas se encontram é $O(r \log\log n)$ e o tempo gasto com a remoção de arestas do heap D é $O((r/\log n)\log n) = O(r)$, já que o número de duplos mínimos incidentes em uma componente S é limitado pelo número

de pacotes de S. Portanto, ao longo de todas as iterações da primeira fase, o tempo gasto com a remoção de arestas dos heaps usados na implementação é $O(nr \log\log n)$. Agora, vamos analisar o tempo gasto com inserções nos heaps. Observe que, em cada iteração, o número de inserções de arestas em pacotes é igual a $4r$, já que cada aresta acordada é inserida em dois pacotes. Além disso, o número de inserções no heap D em cada iteração é $O(r)$. A razão disso é que as inserções feitas em D, devido ao surgimento de novos duplos mínimos, podem ocorrer em dois momentos em uma iteração:

- quando são removidas arestas de pacotes comuns e, neste caso, o número de inserções feitas em D é limitado pelo número de remoções feitas nesses pacotes e, portanto, nunca ultrapassa $6r$;
- quando surgem novos pacotes comuns em razão da inserção, em pacotes em crescimento, das arestas acordadas incidentes na componente gerada pela inclusão de uma nova aresta na floresta; neste caso, o número de inserções em D não ultrapassa $2r$.

Assim, em cada iteração, o número de inserções feitas em heaps é $O(r)$ e, deste modo, considerando que todos os heaps são implementados através de heaps de Fibonacci, o tempo gasto com estas inserções em cada iteração é $O(r)$. Logo, ao longo de todas as iterações da primeira fase, o tempo gasto com inserções em heaps é $O(nr)$.

Observe ainda que, para determinar uma aresta externa de folga mínima, o tempo gasto em cada iteração é $O(n)$. A atualização dos valores de $d(v)$ para cada vértice v em uma componente ativa da floresta, a união das componentes em que se encontram os extremos da aresta externa de folga mínima escolhida e as operações para decidir se a componente resultante desta união é ou não ativa podem ser executadas em tempo $O(n)$ amortizado, como vimos na implementação anterior (note que, em decorrência do uso da estrutura union-find para implementar a união das componentes, é necessário falar em tempo amortizado). Além disso, para selecionar as $2r$ menores arestas que incidem na componente gerada pela inclusão da nova aresta na floresta, podemos utilizar um algoritmo de seleção em tempo linear e, assim, o tempo gasto com este passo em cada iteração também é $O(n)$. Portanto, durante todas as iterações da primeira fase, o tempo gasto com todos estes passos é $O(n^2)$.

Resta agora determinar o tempo gasto com a inicialização do heap D e dos pacotes das componentes da floresta no início de cada subfase. Note que, usando um algoritmo de seleção em tempo linear, podemos construir os pacotes das componentes da floresta e, simultaneamente, construir D em tempo $O(m)$. Como o número total de subfases é $\lceil n/r \rceil$, o tempo gasto com estas inicializações durante toda a primeira fase é, portanto, $O(nm/r)$.

Logo, por tudo o que vimos acima, a complexidade de tempo desta implementação, em função de r, é $O(nr \log\log n + n^2 + nm/r)$. Portanto, escolhendo $r = \sqrt{m/\log\log n}$, como sugerem

Gabow, Goemans e Williamson, temos que a complexidadade de tempo da implementação é $O(n(n + \sqrt{m \log \log n}))$.

3.2 Implementação em CWEB

A função *sf_ggw*, que apresentamos abaixo, recebe um grafo g simples, um vetor *term_sets* de conjuntos de terminais, que se encontram indexados de 0 a *g→num_ts* − 1 no vetor, e um inteiro *random* em $\{0, 1\}$. Ela implementa os passos descritos na seção anterior para construir uma floresta de Steiner de g, devolvendo, ao final de sua execução, a floresta construída. Caso não exista uma floresta de Steiner em g, a função devolve Λ. Se *random* $\equiv 0$, o algoritmo de seleção usado por *sf_ggw* será um algoritmo cujo consumo de tempo no pior caso é linear no tamanho da entrada, como especificado na implementação de Gabow, Goemans e Williamson. Caso contrário, ou seja, se *random* $\equiv 1$, a função usará um algoritmo de seleção mais simples, aleatorizado, que tem consumo esperado de tempo linear.

45 ⟨ Steiner forest construction functions 10 ⟩ +≡

```
SteinerForest *sf_ggw(Graph *g, TermSet *term_sets, int random)
{
  ⟨ Local variables of sf_ggw 49 ⟩
  ⟨ Memory allocation and parameter settings 48 ⟩
  ⟨ Initialize the structures corresponding to the components of the initial forest 52 ⟩
  ⟨ Initialize d(v) for each vertex v of the graph 18 ⟩
  ⟨ Initialize the packets of each component 57 ⟩
  ⟨ Include an edge in the forest while there is some active component 76 ⟩
  ⟨ Find and return a minimal Steiner forest 92 ⟩
}
```

Como na implementação que vimos no capítulo anterior, manteremos em uma lista, durante toda a execução da fase iterativa, as estruturas que representam as componentes da floresta corrente. Cada componente da floresta é representada através de uma estrutura do tipo **GGWComponent**. Esta estrutura é bastante parecida com aquela que utilizamos para respresentar as componentes no capítulo anterior: a única diferença é que esta possui dois campos adicionais, *gr_packet* e *packets*, correspondentes aos pacotes que guardam as arestas acordadadas incidentes na componente. Dada uma componente C, o pacote em crescimento de C será guardado em *C→gr_packet*, enquanto que os pacotes comuns de C serão armazenados em uma lista ligada e *C→packets* guardará um apontador para o primeiro nó desta lista.

46 ⟨ Data structures of `sf.c` 4 ⟩ +≡

```
typedef struct packnode *PackList;
typedef struct ggw_component {
  int id;
  int active;
  Vertex *vertices;
  int num_vertices;
  int *ts_intersect_size;
  Arc *edges;
  PackList packets;
  FibHeap gr_packet;
  struct ggw_component *prev;
  struct ggw_component *next;
} GGWComponent;
```

Os nós da lista ligada de pacotes comuns de uma componente são implementados através de estruturas do tipo **struct packnode**, que é apresentado abaixo.

47 ⟨Data structures of sf.c 4⟩ +≡

```
struct packnode {
  FibHeap packet;
  struct packnode *next;
};
```

Alocaremos, de uma vez só, a memória ocupada por todos os nós deste tipo e organizaremos todos eles em uma lista livre. A variável *nodes* guardará um apontador para o nó-cabeça desta lista. Como vimos, durante toda a execução de uma subfase, o número de pacotes associados a uma componente nunca ultrapassa $\lceil 3r/\log n \rceil$ e, portanto, como utilizaremos os nós apenas para guardar pacotes comuns, o número de nós na lista livre, sem contar o nó-cabeça, nunca será maior que $3nr/\log n$.

48 ⟨Memory allocation and parameter settings 48⟩ ≡

```
{
  int i, n = g.n;
  num_ordpack = (3 * n * r)/log_n;
  nodes = malloc((num_ordpack + 1) * sizeof(struct packnode));
  for (i = 0; i < num_ordpack; i++) nodes[i].next = nodes + i + 1;
  nodes[i].next = nodes;
}
```

Veja também blocos 56, 62 e 73.

Este código é usado no bloco 45.

Para calcular o valor das variáveis *log_n* e r, utilizamos a função *msb* que apresentaremos adiante. Observe que a expressão $g\to m/2$, que aparece no cáculo de r, é igual ao número de arestas do grafo g.

49 ⟨ Local variables of *sf_ggw* 49 ⟩ ≡

```
int log_n = msb(g→n), r = sqrt((g→m/2)/msb(log_n)), num_ordpack;
PackList nodes;
```

Veja também blocos 54, 58, 63, 74, 77 e 86.

Este código é usado no bloco 45.

50 ⟨ Header files of `sf.c` 2 ⟩ +≡

```
#include <math.h>
```

A função *msb*, que apresentamos abaixo, recebe um inteiro x e devolve, em tempo $O(\log x)$, o índice do bit 1 mais à esquerda, o bit mais significativo, na representação binária de x, ou equivalentemente, o chão do logaritmo na base 2 de x.

51 ⟨ Auxiliary functions 20 ⟩ +≡

```
static int msb(int x)
{
  int j = 0;
  while (x > 1) {
    x = x/2; j++;
  }
  return j;
}
```

Como mencionamos há pouco, durante toda a fase iterativa, as estruturas que representam as componentes da floresta corrente ficarão todas armazenadas em uma lista. Tal lista é implementada através de uma lista circular duplamente ligada com cabeça e a variável *components* guardará um apontador para o nó-cabeça desta lista. A floresta corrente será representada por uma estrutura do tipo **Graph** e a variável *sf* guardará um apontador para tal estrutura. No trecho de código abaixo, é feita a inicialização de todas estas estruturas, de modo que elas correspondam à floresta geradora inicial, a qual não possui arestas.

52 ⟨ Initialize the structures corresponding to the components of the initial forest 52 ⟩ ≡

```
{
  int i;
  GGWComponent *t;
  sf = gb_new_graph(g→n);
  components = t = malloc((g→n + 1) * sizeof(GGWComponent));
  if (¬sf ∧ ¬components) {
    err = 2; return Λ;
  }
  ⟨ Some initializations 15 ⟩
  for (i = 0; i < g→n; i++) {
    Vertex *v = g→vertices + i;
    C = components + i + 1;
    ⟨ Initialize the fields of the component C of v 53 ⟩
    t→next = C;
    C→prev = t;
    t = t→next;
  }
  t→next = components;
  components→prev = t;
  UFinit(g→vertices, g→n);
}
```

Este código é usado no bloco 45.

53 ⟨ Initialize the fields of the component C of v 53 ⟩ ≡

```
⟨ Initialize some fields of the component C of v 16 ⟩
⟨ Create a double linked list of edges incident in C 55 ⟩
```

Este código é usado no bloco 52.

No trecho abaixo, declaramos algumas das variáveis locais da função *sf_ggw*. Note que elas possuem o mesmo nome que algumas das variáveis que usamos na implementação da função *sf_gw* no capítulo anterior. E isto não é à toa, já que aqui elas desempenham o mesmo papel desempenhado por suas variáveis homônimas em *sf_gw*. Veremos que o mesmo irá acontecer com outras variáveis de *sf_ggw* que serão apresentadas mais adiante. Desta forma, para não sermos repetitivos, não descreveremos aqui qual é a função de cada uma destas variáveis e pedimos que o leitor consulte o capítulo anterior sempre que necessário.

54 ⟨ Local variables of *sf_ggw* 49 ⟩ +≡

```
Graph *sf;
GGWComponent *components, *C;
int num_actives, *sizes;
Vertex **last;
double *d_value;
Arc *head;
ExtraFields *ef;
```

A construção da lista de arestas $C\to edges$ é feita abaixo. Aproveitaremos a ocasião para inicializar alguns dos campos presentes nas estruturas correspondentes às arestas que farão parte da lista. Como na implementação anterior, utilizaremos as macros $prev(a)$ e $next(a)$ para acessar os campos da aresta a que guardam apontadores, respectivamente, para a aresta que precede e para aquela que sucede a na lista *edges* de alguma componente da floresta corrente, caso a se encontre em uma tal lista. Também, como na implementação anterior, utilizaremos a macro $act(a)$ para acessar o campo de a que guarda o número de componentes ativas dentre aquelas em que se encontram os extremos da aresta a. Em adição a estas macros, definiremos aqui outras duas: $rank(a)$ e $D_node(a)$. A primeira é usada para acessar o campo de a que guarda um inteiro entre 0 e $g\to m/2-1$ que identifica de maneira única a aresta a. Usaremos este número para decidir qual, dentre duas arestas, é a menor, caso elas tenham a mesma folga. A segunda macro, $D_node(a)$, é usada para acessar o campo de a que guarda ou o nó do heap D em que a se encontra, ou Λ, caso a não esteja em D.

55

```
#define rank(a) (ARC_FIELDS(a)→e5.I)
#define D_node(a) (ARC_FIELDS(a)→e7.P)
⟨ Create a double linked list of edges incident in C 55 ⟩ ≡
  {
    static int p = 0, q = 0, rank = 0;
    Arc *a;
    C→edges = head + p++;
    C→edges→afields = (char *)(ef + q++);
    prev(C→edges) = next(C→edges) = C→edges;
    for (a = v→arcs; a; a = a→next) {
      if (v < a→tip) rank(a) = rank++;
      else rank(a) = rank(inverse(a));
      D_node(a) = Λ;
      act(a) = C→active;
      if (a→tip→termset_id ≥ 0) act(a)++;
```

```
    insert_edge(C→edges, a);
  }
}
```

Este código é usado no bloco 53.

No trecho é abaixo, é feita a alocação do espaço para armazenar o heap D e os pacotes de todas as componentes da floresta. Note que o número total de pacotes em crescimento em cada iteração nunca é superior a $g{\to}n$.

56 ⟨Memory allocation and parameter settings 48⟩ +≡

```
if (¬FHEAPalloc(1 + num_ordpack + g→n, g→m + g→m/2)) {
  err = 2; return Λ;
}
```

Agora, vamos mostrar como é feita a construção dos pacotes associados a cada componente e também do heap D. Cada pacote, assim como o heap D, é implementado através de um heap de Fibonacci. A função *EDGE_less*, usada para comparar as arestas presentes nestes heaps, será apresentada adiante.

57 ⟨Initialize the packets of each component 57⟩ ≡

```
{
  int i;
  for (C = components→next; C ≠ components; C = C→next) {
    C→gr_packet = FHEAPinit(EDGE_less);
    C→packets = Λ;
  }
  D = FHEAPinit(EDGE_less);
  ⟨Other initializations 60⟩
  for (C1 = components→next; C1 ≠ components; C1 = C1→next)
    ⟨Select the 2r smallest edges in the list C1→edges 61⟩
}
```

Este código é usado nos blocos 45 e 89.

58 ⟨Local variables of *sf_ggw* 49⟩ +≡

```
GGWComponent *C1;
FibHeap D;
```

A função abaixo é usada para comparar duas arestas, a e b, em cada um dos heaps usados na implementação. Caso $a \equiv \Lambda \neq b$ ou a seja menor que b, ela devolve 1 e, caso $b \equiv \Lambda$ ou b seja menor que a, ela devolve 0.

59 ⟨ Auxiliary functions 20 ⟩ +≡

```
int EDGE_less(void *a, void *b)
{
  Arc *e1 = a, *e2 = b;
  double s1, s2;
  if (¬e2) return 0;
  if (¬e1) return 1;
  s1 = key(e1);
  s2 = key(e2);
  if (s1 < s2) return 1;
  else {
    if (s2 < s1) return 0;
    else return (rank(e1) < rank(e2));
  }
}
```

Para determinar as arestas que serão consideradas acordadas no início de cada subfase, contaremos com o auxílio do vetor *selected*, que é usado para "marcar" as arestas externas que foram selecionadas dentre as $2r$ menores que estão na lista *edges* de alguma componente da floresta. Mais especificamente, para cada i entre 0 e $g\to m/2 - 1$, $select[i] \equiv 1$ se a aresta a tal que $rank(a) \equiv i$ se encontra entre as $2r$ menores arestas externas que estão na lista $C\to edges$ de alguma componente C da floresta; caso contrário, $select[i] \equiv 0$.

60 ⟨ Other initializations 60 ⟩ ≡

```
for (i = 0; i < g→m/2; i++) selected[i] = 0;
```

Veja também bloco 72.

Este código é usado no bloco 57.

A seleção das $2r$ menores arestas que se encontram na lista *edges* da componente `C1` é feita abaixo. A função usada para fazer a seleção dessas arestas é guardada na variável *selection*. Mais adiante, apresentaremos as duas funções dentre as quais escolheremos, de acordo com o valor do parâmetro *random*, a que realizará esta tarefa. Como, durante a seleção, precisaremos consultar mais de uma vez a folga de uma mesma aresta e como sabemos que, durante toda a execução deste

passo, não há variação na folga das arestas, usaremos um dos campos da estrutura de cada aresta a para armazenar a folga de a e, assim, evitar que ficássemos recalculando desnecessariamente esse valor seguidas vezes. Para acessar o campo que guarda a folga de a, usaremos a macro *slackness*(a).

```
61 #define slackness(a) (ARC_FIELDS(a)→e8.D)
⟨Select the 2r smallest edges in the list C1→edges 61⟩ ≡
  {
    int n = 0;
    Arc *a;
    for (a = next(C1→edges); a ≠ C1→edges; a = next(a)) {
      slackness(a) = key(a);
      aux[n++] = a;
    }
    if (2 * r < n) {
      selection(aux, 0, n − 1, 2 * r);
      n = 2 * r;
    }
    for (i = 0; i < n; i++) {
      a = aux[i];
      if (¬selected[rank(a)]) selected[rank(a)] = 1;
      else ⟨Make a an awake edge 71⟩
    }
  }
```

Este código é usado no bloco 57.

```
62 ⟨Memory allocation and parameter settings 48⟩ +≡
    aux = malloc(g→n * sizeof(Arc *));
    selected = malloc((g→m/2) * sizeof(int));
    if (¬aux ∨ ¬selected) {
      err = 2; return Λ;
    }
    if (random) selection = randomized_select;
    else selection = select;
```

```
63 ⟨Local variables of sf_ggw 49⟩ +≡
    Arc **aux;
    Arc *(*selection)(Arc **, int, int, int);
```

```
int *selected;
```

A função *_EDGE_less* recebe duas arestas externas, a e b, tais que $a \neq \Lambda \neq b$. Caso a seja menor que b, ela devolve 1 e, caso contrário, ela devolve 0. Note que *_EDGE_less* realiza a mesma tarefa que a função *EDGE_less* que apresentamos anteriormente. A diferença é que a função abaixo tira proveito do fato de termos guardado a folga de cada aresta a no campo de a acessado pela macro *slackness*(a).

64 ⟨ Auxiliary functions 20 ⟩ +≡

```
int _EDGE_less(Arc *a, Arc *b)
{
  double s1 = slackness(a), s2 = slackness(b);
  if (s1 < s2) return 1;
  else {
    if (s2 < s1) return 0;
    else return (rank(a) < rank(b));
  }
}
```

Abaixo, apresentamos uma das funções que poderemos usar para fazer a seleção das $2r$ menores arestas na lista *edges* de uma componente da floresta. A função *select* recebe um vetor a de arestas, índices l e r desse vetor tais que $l \leq r$ e um inteiro $j \leq r - l + 1$. Ela devolve o j-ésimo menor elemento dentre aqueles que se encontram entre as posições l e r do vetor a. Além disso, ela reorganiza os elementos de a que se encontram entre as posições l e r de modo que os j menores elementos dentre eles se encontrem nas j primeiras posições de a a partir de l. O algoritmo implementado pela função *select* é de Blum, Floyd, Pratt, Rivest e Tarjan [6, 7] e, no pior caso, leva tempo $O(n)$ para realizar a sua tarefa, onde $n = r - l + 1$.

65 ⟨ Auxiliary functions 20 ⟩ +≡

```
⟨ Edge selection auxiliary functions 66 ⟩
Arc *select(Arc **a, int l, int r, int j)
{
  if (l ≡ r) return a[l];
  else {
    int i;
    Arc *x;
    ⟨ Choose an adequate element x to partitioning a 67 ⟩
    i = partition(a, l, r, x);
```

```
    if (j ≡ i − l + 1) return a[i];
    if (j < i − l + 1) return select(a, l, i − 1, j);
    else return select(a, i + 1, r, j − (i − l + 1));
  }
}
```

A função *partition* é uma peça essencial na construção de algoritmos de seleção. Ela recebe um vetor a de arestas, índices l e r desse vetor e um elemento x de a que se encontra entre as posições l e r do vetor, ou seja, $x \equiv a[i]$ para algum i tal que $l \leq i \leq r$. Então, ela rearranja os elementos de a de modo que, para todo j entre l e r, se $a[j]$ é menor que x, então $j < i$ e, se $a[j]$ é maior que x, então $j > i$.

66 ⟨Edge selection auxiliary functions 66⟩ ≡

```
int partition(Arc **a, int l, int r, Arc *x)
{
  int i = l − 1, j = r + 1;
  while (1) {
    do j−−; while (_EDGE_less(x, a[j]));
    do i++; while (_EDGE_less(a[i], x));
    if (i < j) exchange(a[i], a[j])
    else return j;
  }
}
```

Veja também blocos 68 e 70.

Este código é usado no bloco 65.

A escolha do elemento x adequado para particionar o vetor a é fundamental para que a complexidade de tempo do algoritmo no pior caso seja linear. No caso deste algoritmo, x é escolhido dividindo-se a em grupos de 5 elementos (eventualmente, um dos grupos pode ter menos de 5 elementos) e tomando-se x como sendo a mediana do conjunto formado pelas medianas dos grupos em que a foi dividido.

67 ⟨Choose an adequate element x to partitioning a 67⟩ ≡

```
{
  int i, n = r − l + 1, m = (n % 5) ? n/5 + 1 : n/5;
  for (i = 0; i < m − 1; i++) {
    insertionSort(a, l + 5 * i, l + 5 * i + 4);
```

```
    exchange(a[l + i], a[l + 5 * i + 2])
  }
  insertionSort(a, l + 5 * m - 5, r);
  i = (n - 5 * m + 5);
  if (i % 2) exchange(a[l + m - 1], a[(r + l + 5 * m - 4)/2])
  else {
    Arc *m1 = a[(r + l + 5 * m - 5)/2];
    Arc *m2 = a[(r + l + 5 * m - 5)/2 + 1];
    if (_EDGE_less(m2, m1)) exchange(a[l + m - 1], a[(r + l + 5 * m - 5)/2])
    else exchange(a[l + m - 1], a[(r + l + 5 * m - 5)/2 + 1])
  }
  i = (m % 2) ? m/2 + 1 : m/2;
  x = select(a, l, l + m - 1, i);
}
```

Este código é usado no bloco 65.

Apresentamos abaixo a função *insertionSort*, usada acima para ordenar cada um dos grupos em que a foi dividido. Ela recebe um vetor a de arestas e índices l e r desse vetor e rearranja a de modo que, para quaisquer dois índices i e j entre l e r tais que $a[i]$ é menor que $a[j]$, tenhamos $i < j$.

68 ⟨Edge selection auxiliary functions 66⟩ +≡

```
void insertionSort(Arc **a, int l, int r)
{
  int i;
  for (i = l + 1; i ≤ r; i++) {
    int j;
    Arc *x = a[i];
    for (j = i; j > 0 ∧ _EDGE_less(x, a[j - 1]); j—) a[j] = a[j - 1];
    a[j] = x;
  }
}
```

A outra função que poderemos usar na escolha das $2r$ menores arestas na lista *edges* de uma componente é *randomized_select*. Ela recebe um vetor a de arestas, índices l e r desse vetor tais que $l \leq r$ e um inteiro $j \leq r - l + 1$ e devolve o j-ésimo menor elemento dentre aqueles que se encontram entre as posições l e r do vetor a. Além disso, ela também rearranja os elementos que

estão entre as posições l e r de a de modo que os j menores elementos dentre eles se encontrem nas j primeiras posições de a a partir de l. A função *randomized_select* implementa um algoritmo de seleção aleatorizado que tem consumo esperado de tempo $O(n)$, onde $n = r - l + 1$ (veja [6]).

69 ⟨ Auxiliary functions 20 ⟩ +≡

```
Arc *randomized_select(Arc **a, int l, int r, int j)
{
  int q, k;
  if (l ≡ r) return a[l];
  q = randomized_partition(a, l, r);
  k = q − l + 1;
  if (j ≡ k) return a[q];
  if (j < k) return randomized_select(a, l, q − 1, j);
  else return randomized_select(a, q + 1, r, j − k);
}
```

A aleatoriedade presente em *randomized_selection* é proveniente da função que apresentamos abaixo. A função *randomized_partition* recebe um vetor a de arestas e índices l e r deste vetor tais que $l \leq r$. Ela, então, escolhe aleatoriamente um índice i entre l e r e rearranja a de modo que, para todo j entre l e r, se $a[j]$ é menor que $a[i]$, então $j < i$ e, se $a[j]$ é maior $a[i]$, então $j > i$.

70 ⟨ Edge selection auxiliary functions 66 ⟩ +≡

```
int randomized_partition(Arc **a, int l, int r)
{
  int i = l + (rand() % (r − l + 1));
  return partition(a, l, r, a[i]);
}
```

Uma vez que uma aresta é considerada acordada, devemos inseri-la nos pacotes em crescimento das componentes que contêm seus dois extremos. É exatamente isso que é feito no trecho de código abaixo. Seja a um apontador para a estrutura do tipo **Arc** que representa a nova aresta acordada e sejam `C1` e `C2` as componentes em que se encontram, respectivamente, *a→from* e *a→tip*. No heap em crescimento de `C1`, guardaremos a e, no heap em crescimento de `C2`, guardaremos *inverse*(a). Usaremos um dos campos da estrutura apontada por a para guardar o pacote em que a se encontra e, para acessar tal campo, será usada a macro *packet*(a). Além disso, usaremos um outro campo desta estrutura para armazenar o nó de *packet*(a) que contém a e, para acessar este campo, usaremos a macro *node*(a). É claro que também faremos tudo isso para *inverse*(a).

Para detectar o surgimento de novos duplos mínimos, usaremos, como abaixo, o vetor *min*. Para cada i entre 0 e $g\to m/2-1$, $min[i] \equiv 1$ se a aresta a tal que $rank(a) \equiv i$ é mínima em $packet(a)$ e $packet(a)$ é um pacote comum; caso contrário, $min[i] \equiv 0$.

```
71 #define packet(a) (ARC_FIELDS(a)→e6.P)
⟨Make a an awake edge 71⟩ ≡
  {
    int j;
    Arç *b[2], *m;
    b[0] = a; b[1] = inverse(a);
    C2 = (GGWComponent *) COMPONENT(a→tip);
    for (C = C1, j = 0; j ≤ 1; C = C2, j++) {
      node(b[j]) = FHEAPinsert(C→gr_packet, b[j]);
      packet(b[j]) = C→gr_packet;
      if (FHEAPnum(C→gr_packet) ≡ log_n) {
        ⟨Insert C→gr_packet in the list C→packets 75⟩
        m = FHEAPfindMin(C→gr_packet);
        if (¬min[rank(m)]) min[rank(m)] = 1;
        else D_node(m) = D_node(inverse(m)) = FHEAPinsert(D, m);
        C→gr_packet = FHEAPinit(EDGE_less);
      }
    }
  }
```

Este código é usado nos blocos 61 e 85.

```
72 ⟨Other initializations 60⟩ +≡ for (i = 0; i < g→m/2; i++) min[i] = 0;
```

```
73 ⟨Memory allocation and parameter settings 48⟩ +≡
    min = malloc((g→m/2) * sizeof(int));
    if (¬min) { err = 2; return Λ; }
```

```
74 ⟨Local variables of sf_ggw 49⟩ +≡
    int *min; GGWComponent *C2;
```

Se o pacote em crescimento $C\to gr_packet$ atingiu o limite de sua capacidade, iremos inseri-lo na lista $C\to packets$ de pacotes comuns da componente C. Para isto, devemos remover, da lista livre *nodes*, o nó que guardará o novo pacote comum e, após isso, inserir tal nó em $C\to packets$.

75 ⟨ Insert $C\to gr_packet$ in the list $C\to packets$ 75 ⟩ ≡

```
{
  PackList x = nodes→next;
  nodes→next = x→next;
  x→packet = C→gr_packet;
  x→next = C→packets;
  C→packets = x;
}
```

Este código é usado no bloco 71.

Após a inicialização de todas as estruturas envolvidas na implementação, começa a fase iterativa. Em cada iteração, uma nova aresta é incluída na floresta. Desta forma, usaremos o número de arestas na floresta corrente, que será guardado na variável *num_edges*, para detectar o momento em que deverá ter início uma nova subfase. A variável *dual_cost* exerce aqui o mesmo papel que desempenha na implementação do capítulo anterior.

76 ⟨ Include an edge in the forest while there is some active component 76 ⟩ ≡

```
{
  int num_edges = 0;
  while (num_actives > 0) {
    double inc;
    Arc *a, *a_;
    ⟨ Select an edge a with the smallest slackness inc 78 ⟩
    C1 = (GGWComponent *) COMPONENT(a→from);
    C2 = (GGWComponent *) COMPONENT(a→tip);
    if (¬(C1→active) ∧ ¬(C2→active)) { err = 1; return Λ; }
    a_ = inverse(a);
    ⟨ Include a in the forest sf 33 ⟩
    num_edges++;
    ⟨ For each vertex v in some active component, do d(v) += inc 34 ⟩
    dual_cost += inc * num_actives;
    if (C1→active ∧ C2→active) num_actives—;
    ⟨ Join the components C1 and C2 79 ⟩
    if (num_edges % r ≡ 0) ⟨ Update the structures for the new subphase 89 ⟩
  }
}
```

Este código é usado no bloco 45.

77 ⟨ Local variables of *sf_ggw* 49 ⟩ +≡
double *dual_cost* = 0;

Aqui é feita a escolha da próxima aresta a ser incluída na floresta *sf*. Note que a aresta escolhida é uma aresta externa de folga mínima, mas não é necessariamente a menor aresta externa no grafo *g*.

78 ⟨ Select an edge *a* with the smallest slackness *inc* 78 ⟩ ≡

```
if (FHEAPempty(D)) { a = Λ; inc = DBL_MAX; }
else { a = FHEAPfindMin(D); inc = key(a); }
for (C = components→next; C ≠ components; C = C→next) {
  if (¬FHEAPempty(C→gr_packet)) {
    Arc *b = FHEAPfindMin(C→gr_packet);
    double key_b = key(b);
    if (key_b < inc) { a = b; inc = key_b; }
  }
}
if (¬a) { err = 1; return Λ; }
```

Este código é usado no bloco 76.

A união das componentes C1 e C2, onde se encontram os extremos da aresta *a* escolhida no bloco anterior, é feita abaixo. Do mesmo modo que no bloco 35 da implementação do capítulo anterior, manteremos na lista *components* a estrutura correspondente à maior dessas duas componentes (em número de vértices) e alteraremos os campos desta estrutura para que ela represente a componente C1 ∪ C2. A estrutura correspondente à menor componente terá sido removida de *components* ao final da execução do trecho abaixo.

79 ⟨ Join the components C1 and C2 79 ⟩ ≡

```
⟨ Delete a from the lists C1→edges and C2→edges 80 ⟩
⟨ Delete all awake edges incident to C1 or C2 from their packets and from D 81 ⟩
if (C1→num_vertices < C2→num_vertices) exchange(C1, C2)
⟨ Concatenate the lists C1→vertices and C2→vertices 36 ⟩
⟨ Update C1→active 37 ⟩
if (¬C1→active) num_actives—;
UFunion(C1→vertices, C2→vertices);
C2→next→prev = C2→prev;
C2→prev→next = C2→next;
⟨ Choose the awake edges incident to the new component 85 ⟩
```

Este código é usado no bloco 76.

80 ⟨Delete a from the lists C1→*edges* and C2→*edges* 80⟩ ≡

```
delete_edge(a);
delete_edge(a_);
```

Este código é usado no bloco 79.

No trecho abaixo, as arestas acordadas incidentes em C1 e em C2 são removidas de todos os heaps em que elas se encontram. A função *deleteEdge*, que veremos mais adiante, irá nos auxiliar nesta tarefa.

81 ⟨Delete all awake edges incident to C1 or C2 from their packets and from D 81⟩ ≡

```
{
  int i;
  void *args[2];
  args[0] = D; args[1] = min;
  for (C = C1, i = 0; i < 2; C = C2, i++) {
    PackList x = C→packets;
    while (x) {
      PackList next = x→next;
      ⟨Delete from their packets and from D the edges in x→packet 83⟩
      FHEAPdestroy(x→packet);
      ⟨Free the node pointed to by x 82⟩
      x = next;
    }
    FHEAPtraverse(C→gr_packet, deleteEdge, args);
    FHEAPdestroy(C→gr_packet);
  }
}
```

Este código é usado no bloco 79.

82 ⟨Free the node pointed to by x 82⟩ ≡

```
x→next = nodes→next;
nodes→next = x;
```

Este código é usado nos blocos 81 e 90.

A remoção das arestas que estão em $x\to packet$ dos heaps em que elas se encontram é feita abaixo. Observe que, se o mínimo de $x\to packet$ não for um duplo mínimo, a remoção deste mínimo dos dois pacotes em que ele se encontra não gera um novo duplo mínimo. Por este motivo, iremos tratar este caso separadamente.

83 ⟨ Delete from their packets and from D the edges in $x\to packet$ 83 ⟩ ≡

```
if (¬FHEAPempty(x→packet)) {
  Arc *a = FHEAPfindMin(x→packet);
  if (D_node(a)) {
    FHEAPdelete(D, D_node(a));
    D_node(a) = D_node(inverse(a)) = Λ;
  }
  else {
    Arc *a_ = inverse(a);
    FHEAPdelete(packet(a_), node(a_));
    FHEAPdelete(packet(a), node(a));
    min[rank(a)] = 0;
  }
  FHEAPtraverse(x→packet, deleteEdge, args);
}
```

Este código é usado no bloco 81.

A função *deleteEdge* recebe um vetor *args* tal que *args*[0] guarda o heap D e *args*[1] guarda o vetor *min* e também recebe uma aresta *item* tal que ou *item* é um duplo mínimo, ou *packet*(*item*) é um pacote em crescimento, ou *packet*(*item*) é um pacote comum, mas *item* não é mínimo em *packet*(*item*). Ela remove $b \equiv inverse(item)$ de $h \equiv packet(b)$, atualiza o vetor *min* se necessário e, caso tenha surgido um novo duplo mínimo após a remoção de b do pacote h, ela insere em D o novo duplo mínimo gerado.

84 ⟨ Auxiliary functions 20 ⟩ +≡

```
void deleteEdge(void *item, void *args[])
{
  Arc *a = item, *b = inverse(a);
  FibHeap h = packet(b);
  int *min = args[1];
  FHEAPdelete(h, node(b));
  if (min[rank(b)]) {
    min[rank(b)] = 0;
```

```
    if (¬FHEAPempty(h)) {
      Arc *m = FHEAPfindMin(h);
      if (¬min[rank(m)]) min[rank(m)] = 1;
      else {
        FibHeap D = args[0];
        D_node(m) = D_node(inverse(m)) = FHEAPinsert(D, m);
      }
    }
  }
}
```

Após a união das componentes C1 e C2, é necessário decidir quais das arestas incidentes na nova componente da floresta, C1 ∪ C2, serão acordadas. Esta tarefa é realizada pelo trecho de código abaixo. Inicialmente, é construída a lista *edges* da nova componente e, para isto, contaremos com o auxílio dos vetores *adj* e *component_id*, que são usados aqui exatamente da mesma forma que na implementação do capítulo anterior. Ao mesmo tempo, para cada aresta a nas listas C1→*edges* e C2→*edges*, atualizaremos o conteúdo do campo acessado pela macro $act(a)$. A lista construída ficará acessível a partir de C1→*edges*, já que a estrutura apontada por C1 irá representar a nova componente da floresta. Por fim, iremos tornar acordadas as $2r$ menores arestas na lista C1→*edges*.

85 ⟨ Choose the awake edges incident to the new component 85 ⟩ ≡

```
{
  Arc *a;
  int i = 0, n = 0;
  ⟨ Update the field act of the edges in C1→edges 87 ⟩
  ⟨ Do the same for C2→edges and merge C1→edges and C2→edges 88 ⟩
  while (i > 0) adj[component_id[—i]] = Λ;
  C1→packets = Λ;
  C1→gr_packet = FHEAPinit(EDGE_less);
  for (a = next(C1→edges); a ≠ C1→edges; a = next(a)) {
    slackness(a) = key(a);
    aux[n++] = a;
  }
  if (2 * r < n) {
    selection(aux, 0, n − 1, 2 * r);
    n = 2 * r;
  }
```

```
    for (i = 0; i < n; i++) {
      a = aux[i];
      ⟨Make a an awake edge 71⟩
    }
  }
```

Este código é usado no bloco 79.

86 ⟨Local variables of *sf_ggw* 49⟩ +≡

```
  Arc **adj; int *component_id;
```

87 ⟨Update the field *act* of the *edges* in C1→*edges* 87⟩ ≡

```
  for (a = next(C1→edges); a ≠ C1→edges; a = next(a)) {
    C = (GGWComponent *) COMPONENT(a→tip);
    act(a) = act(inverse(a)) = C1→active + C→active;
    adj[C→id] = a;
    component_id[i++] = C→id;
  }
```

Este código é usado no bloco 85.

88 ⟨Do the same for C2→*edges* and merge C1→*edges* and C2→*edges* 88⟩ ≡

```
  for (a = next(C2→edges); a ≠ C2→edges; a = next(a)) {
    Arc *a_ = inverse(a), *t;
    C = (GGWComponent *) COMPONENT(a→tip);
    act(a) = act(a_) = C1→active + C→active;
    if (¬adj[C→id]) {
      t = prev(a);
      delete_edge(a);
      insert_edge(C1→edges, a);
      a = t;
    }
    else {
      Arc *b = adj[C→id];
      if (EDGE_less(a, b)) {
        delete_edge(b);
        delete_edge(inverse(b));
        t = prev(a);
```

```
        delete_edge(a);
        insert_edge(C1→edges, a);
        a = t;
      }
      else delete_edge(a_);
    }
  }
```

Este código é usado no bloco 85.

No início de cada nova subfase, é necessário reconstruir o heap D e os pacotes associados a cada uma das componentes da floresta corrente. Assim, antes de dar início a essa reconstrução, toda memória previamente ocupada pelas estruturas correspondentes a estes pacotes e ao heap D é devolvida às listas livres.

89 ⟨ Update the structures for the new subphase 89 ⟩ ≡

```
  {
    ⟨ Free the memory allocated to the old structures 90 ⟩
    ⟨ Initialize the packets of each component 57 ⟩
  }
```

Este código é usado no bloco 76.

90 ⟨ Free the memory allocated to the old structures 90 ⟩ ≡

```
  FHEAPtraverse(D, resetDnode, Λ);
  FHEAPdestroy(D);
  for (C = components→next; C ≠ components; C = C→next) {
    PackList x = C→packets;
    while (x) {
      PackList next = x→next;
      FHEAPdestroy(x→packet);
      ⟨ Free the node pointed to by x 82 ⟩
      x = next;
    }
    FHEAPdestroy(C→gr_packet);
  }
```

Este código é usado no bloco 89.

Como iremos reconstruir D, é necessário reinicializar o campo das arestas que se encontram em D que é acessado pela macro *D_node*. A função *resetDnode* recebe uma aresta *item* e reinicializa os campos acessados por *D_node*(*item*) e *D_node*(*inverse*(*item*)).

91 ⟨Auxiliary functions 20⟩ +≡

```
void resetDnode(void *item, void *args[])
{
  Arc *a = item;
  D_node(a) = D_node(inverse(a)) = Λ;
}
```

Ao final da primeira fase, a floresta correspondente à variável *sf* será uma floresta de Steniner do grafo *g* e, como no capítulo anterior, usaremos a função *edgePruning* para determinar uma floresta de Steiner minimal contida na floresta *sf*.

92 ⟨Find and **return** a minimal Steiner forest 92⟩ ≡

```
{
  SteinerForest *min_sf = edgePruning(sf, g, term_sets);
  min_sf→dual_cost = dual_cost;
  ⟨Free the memory allocated by sf_ggw 93⟩
  return min_sf;
}
```

Este código é usado no bloco 45.

93 ⟨Free the memory allocated by *sf_ggw* 93⟩ ≡

```
gb_recycle(sf);
free(components);
free(sizes);
free(d_value);
free(head);
free(ef);
free(nodes);
free(aux);
free(selected);
free(min);
free(adj);
free(component_id);
FHEAPfree();
```

Este código é usado no bloco 92.

Capítulo 4

Implementação de Klein

Como vimos, em nenhum momento do algoritmo de Goemans e Williamson, as arestas que possuem os seus dois extremos em uma mesma componente são examinadas, já que, para decidir qual aresta incluir na floresta, são levadas em consideração apenas arestas externas à floresta corrente. Podemos dizer então que o algoritmo "enxerga" o grafo como se as arestas com os dois extremos em uma mesma componente não existissem, ou ainda, como se cada componente da floresta corrente fosse um vértice e não existissem laços no grafo. Desta forma, cada vez que uma aresta é incluída na floresta, tudo se passa como se as componentes que contêm os extremos dessa aresta se fundissem, dando origem a um novo vértice e eliminando do grafo as arestas que possuiam ambos os extremos nelas. Observe ainda que podemos dividir em dois grupos as arestas candidatas a serem incluídas na floresta em uma dada iteração: o daquelas que possuem os dois extremos em componentes ativas e o daquelas que possuem apenas um dos extremos em uma componente ativa. Toda vez que incluímos uma nova aresta $\overline{e}$ na floresta corrente, o numerador da expressão correspondente à folga de cada aresta e em um destes dois grupos, isto é, $c_e - \sum_{S:e\in\delta(S)} y_S$, que antes da primeira iteração é igual a c_e, sofre um decréscimo, que varia de acordo com o grupo ao qual e pertence. Se e é do grupo das arestas que possuem apenas um dos extremos em uma componente ativa, este valor sofre um decréscimo de $folga(\overline{e})$ e, caso contrário, o decréscimo é de $2folga(\overline{e})$.

Baseando-se em observações como estas, Klein [17] definiu uma nova estrutura de dados que captura certas operações chave que são realizadas em cada iteração da primeira fase do algoritmo e propôs uma implementação que redefine o algoritmo de Goemans e Williamson em termos das operações sobre esta estrutura, reduzindo, desta forma, o problema de implementar de maneira eficiente o algoritmo ao problema de implementar de maneira eficiente as operações sobre tal estrutura de dados. Essa estrutura de dados é definida sob um conjunto V de elementos, que está particionado em "categorias" e permite manipular de maneira eficiente pares de elementos de V. Os pares de elementos de V estão particionados em "bicategorias", dependendo da categoria

dos elementos de V que os formam. A implementação proposta por Klein tem complexidade $O(n\sqrt{m}\log n)$ e, desta forma, é mais rápida que a implementação de Goemans e Williamson quando o grafo de entrada é esparso.

4.1 Descrição da implementação

A estrutura de dados que mencionamos acima é definida da seguinte forma. Dado um grafo orientado G, com custos nos arcos, e um número inteiro C fixo, suporemos que, a cada vértice de G, estará atribuído um número do conjunto $\{1, \ldots, C\}$. Para cada vértice v em V_G, denotaremos por $cat(v)$ o número atribuído a v, ao qual iremos nos referir como a *categoria* de v. Note que, desta forma, cada arco uv do grafo G está associado a um par ordenado $(cat(u), cat(v))$. Daremos o nome de *bicategoria* ao conjunto dos arcos associados a um dado par ordenado de categorias. Observe que as bicategorias definidas deste modo formam uma partição de E_G. Para manipular o conjunto das bicategorias em que E_G se encontra particionado, são utilizadas as seguintes operações:

- *decreaseCost*(b, δ), onde b é uma bicategoria e δ é um número racional. Esta operação diminui de δ o custo de cada arco pertencente à bicategoria b.

- *findMin*(b), onde b é uma bicategoria. Esta operação encontra um arco que apresente o menor custo dentre todos aqueles pertencentes à bicategoria b.

- *changeCategory*(v, c), onde $v \in V_G$ e c é uma das C categorias. Esta operação atribui a categoria c ao vértice v, mudando implicitamente de bicategoria todos os arcos incidentes em v.

- *contractEdge*(a, c), onde $a \in E_G$ e c é uma das C categorias. Esta operação contrai o arco a, transformando-o em um novo vértice do grafo e atribuindo, a este vértice, a categoria c. Tal operação remove, de E_G, a e todos os arcos paralelos a ele.

Estas operações podem ser implementadas de modo que o tempo gasto no total para se executar uma seqüência qualquer de k dessas operações seja $O(k\sqrt{m}\log n + m\log n)$, onde $m = |E_G|$ e $n = |V_G|$. A seguir, descreveremos uma forma de implementar as operações definidas acima dentro deste limite de tempo e, logo após, mostraremos como elas podem ser usadas na implementação do algoritmo de Goemans e Williamson.

Implementação da estrutura de dados correspondente às bicategorias

Seja G um grafo orientado, tal que $m = |E_G|$ e $n = |V_G|$, e considere uma atribuição de categorias aos vértices de G e as correspondentes bicategorias em que E_G fica particionado. Agora, suponhamos que uma seqüência $p_1, \ldots, p_k$ de operações, tomadas dentre as quatro operações definidas acima, seja executada e, para cada i entre 1 e k, seja G_i o grafo resultante após a execução da operação p_i. Além disso, seja $G_0 = G$.

A cada bicategoria em G_i, para todo i entre 0 e k, iremos associar um heap de mínimo de dois níveis. Mais especificamente, a cada bicategoria b, estará associado um heap $H(b)$ cujos elementos serão heaps da forma $H^-(v,b)$ e $H^+(v,b)$, associados a cada vértice v de G_i. Para cada v, os elementos de $H^-(v,b)$ e $H^+(v,b)$ serão, respectivamente, arcos do grafo pertencentes à bicategoria b que têm v como ponta inicial e arcos do grafo pertencentes à bicategoria b que têm v como ponta final. Note que um destes dois heaps sempre estará vazio quando $b = (c, c')$ e $c \neq c'$. Nem todos os arcos da bicategoria b que são incidentes em v serão mantidos nos heaps $H^-(v,b)$ e $H^+(v,b)$, mas apenas aqueles que estiverem *atribuídos* a v. Cada arco de G_i estará atribuído a no máximo um dos seus extremos e tal atribuição pode mudar caso alguma operação *contractEdge* venha a ser executada. Descreveremos o critério usado para a atribuição de arcos aos vértices mais adiante, quando estivermos discutindo a implementação da operação *changeCategory*. Para que possamos, de maneira rápida, ter acesso aos arcos que não estão atribuídos a um dado vértice, associaremos, a cada vértice v do grafo, duas listas, $extra^-(v)$ e $extra^+(v)$, contendo, respectivamente, os arcos do grafo não atribuídos a v que possuem a ponta inicial em v e os arcos do grafo não atribuídos a v que possuem a ponta final em v. Abaixo, descrevemos a implementação de cada uma das quatro operações definidas sobre o conjunto das bicategorias:

- ***decreaseCost***(b, δ): a fim de realizar de maneira rápida esta operação, associaremos, a cada um dos heaps e a cada um dos arcos correspondentes a uma bicategoria b, um número real, de maneira que o custo de cada arco a em b seja o resultado de uma soma de três parcelas: o número real associado a a, o número real associado ao heap em $H(b)$ que contém a e o número real associado a $H(b)$. Denotaremos por Δ_x o número real associado a x, onde x pode ser um arco ou um heap. Note que, de fato, o uso desta estratégia nos permite, de maneira rápida, reduzir de um dado valor δ o custo de todos os arcos que se encontram em uma bicategoria b: para isto, basta subtrair δ de $\Delta_{H(b)}$. Assim, o tempo gasto em cada execução da operação *decreaseCost* é $O(1)$.

- ***findMin***(b): vamos definir a chave dos elementos de cada um dos dois tipos de heaps associados a uma bicategoria b de modo a encontrarmos rapidamente um arco de custo mínimo em b. A chave de um arco a em um heap h será dada pelo valor de Δ_a e a chave de

um heap h em $H(b)$ será dada por $\Delta_h + \Delta_{min(h)}$, onde $min(h)$ é um arco de chave mínima dentre todos aqueles que se encontram em h. Note que, desta forma, existe um meio bem rápido de encontrarmos um arco de custo mínimo em b : primeiro devemos encontrar um heap h que possua a menor chave em $H(b)$ e, em seguida, encontrar um arco que possua a menor chave em h; pela forma como definimos os números Δ_x, onde x é um arco ou um heap, o arco escolhido desta maneira será um arco de custo mínimo em b. Cada um desses passos pode ser executado em tempo $O(1)$ e, portanto, o tempo gasto por cada operação *findMin* é $O(1)$.

- ***changeCategory***(v, c): como já vimos, esta operação atribui uma nova categoria c ao vértice v, alterando, desta forma, a bicategoria em que cada um dos arcos incidentes em v se encontra. Assim, se c_0 era a categoria anteriormente atribuída a v, para cada categoria c', tudo que precisamos fazer é mover os arcos que têm a ponta final em v e que se encontram na bicategoria (c', c_0) para a bicategoria (c', c) e, simetricamente, mover os arcos que têm a ponta inicial em v e que estão em (c_0, c') para (c, c').

  Como já mencionamos, os arcos incidentes em v que estão em uma dada bicategoria b se encontram divididos em dois grupos: o grupo dos arcos que estão atribuídos a v e aquele dos que não estão. Mover este primeiro grupo de arcos é fácil: já que todos eles se encontram agrupados em $H^-(v, b)$ e $H^+(v, b)$, basta mover cada um desses heaps, de uma só vez, para o heap da bicategoria adequada, só tomando o cuidado de atualizar os valores de $\Delta_{H^-(v,b)}$ e $\Delta_{H^+(v,b)}$ para que os custos dos arcos nestes heaps não sofram alteração devido à diferença entre os valores de $\Delta_{H(b)}$ e $\Delta_{H(b')}$, onde b' é a nova bicategoria dos arcos. Por outro lado, mover os arcos do segundo grupo já não é tão simples assim, já que eles se encontram espalhados pelos heaps de $H(b)$ correspondentes a outros vértices do grafo corrente G_i. Para contornar esta dificuldade, faremos uso das listas $extra^-(v)$ e $extra^+(v)$. Desta forma, para mover de bicategoria os arcos incidentes em v que não estão atribuídos a v, basta percorrer cada uma dessas listas e, para cada arco a visitado no percurso, remover a do heap em que ele se encontra em sua bicategoria atual e inseri-lo no heap adequado de sua nova bicategoria (e aqui devemos tomar o cuidado de atualizar o valor de Δ_a de modo que o custo de a não sofra alteração devido à sua mudança de heap). Note que o tempo gasto no total com esse processo é $O(\log n)$ vezes o número de arcos contidos nas listas $extra^-(v)$ e $extra^+(v)$ e, desta forma, é fundamental que adotemos uma estratégia de atribuição de arcos aos vértices que garanta que o número de arcos nestas duas listas não seja muito grande.

  No que segue, diremos que um vértice w tem *grau de saída alto* se o número de arcos que possuem a ponta inicial em w é pelo menos $\sqrt{m}$. Para calcular o grau de saída, estaremos considerando até mesmo arcos que já foram contraídos através de uma operação *contractEdge*, ou seja, o grau de saída de cada vértice não corresponderá ao seu grau de

saída no grafo corrente G_i; em vez disso, ele será calculado em um grafo G'_i ligeiramente diferente, o qual possui os mesmos vértices e arcos de G_i e, além disso, possui todos os arcos do grafo G original que foram contraídos até o momento, de tal forma que, em G'_i, para cada vértice w resultante de uma seqüência de contrações de arcos $a_1, \ldots, a_n$, os arcos $a_1, \ldots, a_n$ têm a ponta inicial e a ponta final em w. A estratégia que iremos adotar para a atribuição de arcos aos vértices é a seguinte: para cada arco uv, se u tem grau de saída alto, iremos atribuir uv a u; caso contrário, iremos atribuir uv a v. Note que, deste modo, para cada vértice w, o número de arcos em $extra^-(w)$ é sempre inferior a $\sqrt{m}$: se w tem grau de saída alto, $extra^-(w)$ estará vazia, já que todos os arcos com a ponta inicial em w estarão atribuídos a w; por outro lado, se w não tem grau de saída alto, o número de arcos com a ponta inicial em w é estritamente menor que $\sqrt{m}$ e, portanto, o mesmo vale para o número de arcos em $extra^-(w)$. Ademais, no máximo $\sqrt{m}$ dos arcos em $extra^+(w)$ são realmente relevantes; os demais podem ser descartados, pois nunca serão escolhidos por uma operação *findMin*. Isto se deve ao seguinte fato: todo vértice que é ponta inicial de algum arco em $extra^+(w)$ tem grau de saída alto. Logo, como no máximo $\sqrt{m}$ dos vértices de G_i tem grau de saída alto, no máximo $\sqrt{m}$ vértices de G_i podem ser ponta inicial de algum arco em $extra^+(w)$. Por outro lado, note que, embora, em $extra^+(w)$, possam existir dois ou mais arcos que possuam a mesma ponta inicial, podemos descartar quase todos estes deixando apenas um único arco que possua custo menor ou igual ao dos arcos descartados.

De acordo com tudo que discutimos acima, para cada operação $changeCategory(v, c)$, o tempo gasto para mover de bicategoria arcos incidentes em v é $O(\sqrt{m} \log n)$ (já que podemos mover os arcos atribuídos a v em tempo $O(\log n)$ e o número de arcos que não estão atribuídos a v a serem movidos é menor que $2\sqrt{m}$). Entretanto, para calcular o tempo gasto no total pela operação, temos de levar em conta ainda o tempo gasto para descartar arcos paralelos em $extra^+(v)$. Iremos fazer isto de maneira amortizada. Como cada arco é descartado no máximo uma única vez (já que não voltaremos a incluir, nas estruturas de dados, arcos que já foram descartados), durante toda a vida das estruturas de dados correspondentes às bicategorias (desde a sua criação até o final de sua utilização), serão executados no máximo m descartes. Cada descarte de um arco consiste em removê-lo de uma lista $extra^+(v)$, para algum v, e do heap da bicategoria em que ele se encontra e, portanto, cada descarte pode ser realizado em tempo $O(\log n)$. Logo, o tempo gasto no total para descartar arcos em qualquer seqüência de operações *changeCategory* é $O(m \log n)$. Deste modo, para executar uma seqüência de r operações *changeCategory*, o tempo gasto no total é $O(r\sqrt{m} \log n + m \log n)$.

- ***contractEdge***(uv, c): inicialmente, mudamos a categoria dos vértices u e v para c, através de duas chamadas à operação *changeCategory*. Em seguida, o arco uv a ser contraído e todos os arcos paralelos a ele são removidos dos heaps em que se encontram e das listas

$extra^+$ ou $extra^-$ às quais pertencem. Por fim, para cada bicategoria b, fazemos a união dos heaps $H^+(u,b)$ e $H^+(v,b)$ e dos heaps $H^-(u,b)$ e $H^-(v,b)$ e, feito isso, realizamos a união das listas $extra^+(u)$ e $extra^+(v)$ e das listas $extra^-(u)$ e $extra^-(v)$.

Note que, ao realizarmos a união de dois heaps no heap correspondente a uma dada bicategoria, teremos de definir o valor de Δ_h, onde h é o heap resultante da união. Para isso, iremos usar a seguinte estratégia. Sejam h_1 e h_2 os heaps a serem unidos e suponhamos que h_1 seja o heap correspondente ao vértice de maior *peso* (dentre u e v), onde o *peso* de um vértice z é definido através da seguinte recorrência:

$$peso(z) = \begin{cases} 1 & \text{, se } z \text{ é um vértice do grafo } G \text{ original,} \\ peso(x) + peso(y) & \text{, se } z \text{ é resultante da contração do arco } xy. \end{cases}$$

Definiremos o valor de Δ_h como sendo igual a Δ_{h_1} e, além disso, atualizaremos adequadamente o valor de Δ_a, para cada arco a em h_2, de modo que o custo dos arcos em h_2 não sofra alteração após a união. Agora, vamos mostrar que o uso desta estratégia garante que, em qualquer seqüência de operações *contractEdge*, o número de vezes que atualizaremos Δ_a, para cada arco a, é $O(\log n)$ e, desta forma, o tempo gasto com essas atualizações, em qualquer seqüência de operações *contractEdge*, é $O(m \log n)$. Primeiramente, considere o caso em que, durante toda a seqüência de operações *contractEdge*, o vértice z ao qual o arco a está atribuído ou é sempre ponta inicial de a, ou é sempre ponta final de a. Neste caso, a pior situação que poderíamos ter é aquela em que, toda vez que o heap correspondente a z em que a se encontra é unido ao heap correspondente a um vértice z', o peso de z é sempre menor que o peso de z'. Observe que, assim, cada vez que Δ_a é atualizado, o extremo a que a está atribuído tem seu peso multiplicado por um fator maior ou igual a dois. Desta forma, como o peso de qualquer vértice é limitado por n, podemos concluir que o número de vezes que Δ_a será atualizado nunca é superior a $\log n$. Por fim, vamos considerar o caso mais geral, em que o extremo ao qual a está atribuído pode mudar ao longo da seqüência de operações *contractEdge*. Note que, de acordo com o modo como definimos o grau de saída de um vértice, sabemos que o grau de saída da ponta inicial de a nunca diminui e, desta forma, a partir do momento em que a é atribuído à sua ponta inicial, a jamais será atribuído ao seu outro extremo. Logo, durante toda a seqüência de operações *contractEdge*, o extremo ao qual a se encontra atribuído é alterado no máximo uma única vez e, desta forma, podemos concluir que Δ_a é atualizado não mais que $2 \log n$ vezes.

Além dos passos que enumeramos acima, note que, durante uma operação *contractEdge*, também pode haver a necessidade de alterarmos os vértices aos quais arcos que possuíam a ponta inicial em u ou em v se encontram atribuídos. Isso irá acontecer quando o vértice w resultante da contração da aresta uv tiver grau de saída alto e o mesmo não ocorrer para pelo menos um vértice dentre u e v. Note que, de acordo com o modo como definimos o grau de saída de um vértice, o grau de saída de w é dado pela soma dos graus de saída

de u e de v. Neste caso, para cada vértice x em $\{u, v\}$ que não possuir grau de saída alto, deveremos percorrer a lista $extra^-(x)$ e, para cada arco xy visitado neste percurso, deveremos remover xy do heap $H^+(y, b)$, onde b é a bicategoria a que xy pertence, inserir xy em $extra^+(y)$, inserir xy no heap $H^-(w, b)$ e, por fim, remover xy de $extra^-(w)$. Como o número de arcos para os quais executaremos o procedimento descrito acima é $O(\sqrt{m})$ (pois, como já vimos, para todo vértice x, o número de arcos em $extra^-(x)$ nunca ultrapassa $\sqrt{m}$), o tempo gasto no total por cada operação *contractEdge* para mudar a atribuição de arcos aos vértices é $O(\sqrt{m} \log n)$. Deste modo, por tudo que vimos acima, podemos concluir que o tempo gasto no total para se executar uma seqüência de r operações *contractEdge* é $O(r\sqrt{m} \log n + m \log n)$.

Implementação do algoritmo de GW usando bicategorias

Observe que o grafo G da entrada do algoritmo de Goemans e Williamson é não-orientado e, dessa forma, como a definição da estrutura de dados correspondente às bicategorias é feita a partir de um grafo orientado, inicialmente, deve-se atribuir, de maneira arbitrária, uma orientação às arestas de G. O algoritmo que descreveremos aqui independe da orientação adotada. Para que nossa exposição fique mais simples, continuaremos a chamar os elementos em E_G de arestas, em vez de chamá-los de arcos, como deveríamos devido à orientação.

Durante toda a execução da primeira fase do algoritmo, os vértices de G ficarão particionados em duas categorias, às quais nos referiremos como INATIVO e ATIVO. Inicialmente, a categoria INATIVO será formada pelos vértices de Steiner de G e a categoria ATIVO pelos demais vértices do grafo. Assim, E_G ficará particionado em quatro bicategorias: (INATIVO, INATIVO), (ATIVO, INATIVO), (INATIVO, ATIVO) e (ATIVO, ATIVO).

Em cada iteração, para decidir qual aresta incluir na floresta, apenas as arestas nas bicategorias (ATIVO, INATIVO), (INATIVO, ATIVO) e (ATIVO, ATIVO) serão levadas em consideração. Para cada uma destas bicategorias, é tomada uma aresta candidata a ser incluída na floresta através de uma chamada a *findMin*. Nesta e nas demais operações sobre bicategorias, o custo que será considerado para cada aresta e será o valor $\hat{c}_e = c_e - \sum_{S:e\in\delta(S)} y_S$, que chamaremos aqui de *custo reduzido* de e. Observe que, inicialmente, $\hat{c}_e = c_e$ para cada e em E_G. Note ainda que, quando e é uma aresta que se encontra em (ATIVO, INATIVO) ou em (INATIVO, ATIVO) ou em (ATIVO, ATIVO), $\hat{c}_e$ corresponde ao numerador da expressão da folga de e. Assim, se $e_1 = findMin$(ATIVO, INATIVO), $e_2 = findMin$(INATIVO, ATIVO) e $e_3 = findMin$(ATIVO, ATIVO), a aresta que será incluída na floresta é escolhida, comparando-se $\hat{c}_{e_1}$, $\hat{c}_{e_2}$ e $\hat{c}_{e_3}/2$ e tomando-se uma aresta correspondente ao menor destes três valores.

Após a escolha da aresta $\bar{e}$ que será incluída na floresta corrente, atualizaremos o custo reduzido das arestas nas bicategorias (ATIVO, INATIVO), (INATIVO, ATIVO) e (ATIVO, ATIVO). De

acordo com a observação que fizemos no início deste capítulo, o custo reduzido de cada aresta que está em uma das duas primeiras bicategorias acima sofrerá um decréscimo de $folga(\overline{e})$, enquanto que o custo reduzido de cada aresta na terceira delas terá um decréscimo de $2folga(\overline{e})$. Para isto, serão executadas três chamadas a *decreaseCost*.

Por fim, a aresta $\overline{e}$ é incluída na floresta e é feita a determinação do estado da nova componente da floresta, criada após a inclusão de $\overline{e}$. Então, $\overline{e}$ é contraída no grafo através de uma operação *contractEdge*. Caso a nova componente da floresta seja ativa, a categoria do vértice resultante da contração será ATIVO e, caso contrário, será INATIVO. Abaixo, apresentamos em pseudo-código o algoritmo correspondente à descrição que fizemos acima. Para simplificar a apresentação, estamos supondo que a instância $(G, \mathcal{R}, c)$ recebida pelo algoritmo é sempre viável.

Algoritmo *MinFSAdaptado(G, $\mathcal{R}$, c)*

1 atribua uma orientação arbitrária às arestas em E_G
2 $\hat{c}_a \leftarrow c_a$ para cada a em E_G
3 para cada $v \in V_G$ faça
4 se v é um vértice de Steiner
5 $cat(v) \leftarrow$ INATIVO
6 senão
7 $cat(v) \leftarrow$ ATIVO
8 $F \leftarrow (V_G, \emptyset)$
9 enquanto G tem algum vértice v tal que $cat(v) =$ ATIVO faça
10 $e_1 \leftarrow$ *findMin*(INATIVO, ATIVO)
11 $e_2 \leftarrow$ *findMin*(ATIVO, INATIVO)
12 $e_3 \leftarrow$ *findMin*(ATIVO, ATIVO)
13 escolha $\overline{e} \in \{e_1, e_2, e_3\}$ tal que $folga(\overline{e}) = \min\{folga(e_j) \mid j \in \{1, 2, 3\}\}$
14 *decreaseCost*((INATIVO, ATIVO), $folga(\overline{e})$)
15 *decreaseCost*((ATIVO, INATIVO), $folga(\overline{e})$)
16 *decreaseCost*((ATIVO, ATIVO), $2 \cdot folga(\overline{e})$)
17 $E_F \leftarrow E_F \cup \{\overline{e}\}$
18 seja C a nova componente de F, criada após a inclusão de $\overline{e}$ em E_F
19 se C é ativa
20 *contractEdge*($\overline{e}$, ATIVO)
21 senão
22 *contractEdge*($\overline{e}$, INATIVO)
23 seja F' uma $\mathcal{R}$-floresta minimal contida em F
24 devolva F'

Como, em cada iteração, o algoritmo executa um número constante de operações sobre bicategorias e no máximo são realizadas $n - 1$ iterações, temos que, no total, são executadas $O(n)$

operações sobre bicategorias. Desta forma, o tempo gasto no total pelo algoritmo com estas operações é $O(n\sqrt{m}\log n + m\log n)$, ou seja, $O(n\sqrt{m}\log n)$. As demais operações podem ser realizadas dentro deste limite de tempo (como veremos na próxima seção) e, portanto, o tempo gasto no total pelo algoritmo é $O(n\sqrt{m}\log n)$.

4.2 Implementação em CWEB

A implementação em CWEB que apresentamos nesta seção está dividida em duas partes. A primeira apresenta o conteúdo dos arquivos `bicat.c` e `bicat.h` que implementam as bicategorias e as funções utilizadas para manipulá-las. A segunda parte apresenta a função de `sf.c` que implementa o algoritmo *MinFSAdaptado* que vimos há pouco, no final da seção anterior.

Bicategorias

O arquivo `bicat.c` se encontra organizado de acordo com a estrutura que exibimos abaixo. As funções utilizadas para a manipulação das bicategorias estão todas agrupadas em ⟨ External functions of `bicat.c` 96 ⟩.

95 ⟨ Header files of `bicat.c` 98 ⟩
⟨ Data structures of `bicat.c` 106 ⟩
⟨ Global variables of `bicat.c` 100 ⟩
⟨ Internal functions of `bicat.c` 111 ⟩
⟨ External functions of `bicat.c` 96 ⟩

Uma delas é *initBicategories*. Ela recebe um inteiro c e um grafo não-orientado g cujos vértices se encontram particionados em c categorias que são identificadas através de números inteiros entre 0 e $c-1$. Ela supõe ainda que, para cada vértice v de g, o número que identifica a categoria à qual v pertence se encontra armazenado em $v\to category$. A função então atribui uma orientação às arestas de g, particionando-as, assim, em bicategorias, e inicializa estruturas que serão utilizadas pelas funções que manipularão estas bicategorias. Ela devolve 0, caso ocorra algum problema na alocação da memória que será usada por estas estruturas e, caso contário, devolve 1.

96 ⟨ External functions of `bicat.c` 96 ⟩ ≡
```
int initBicategories(Graph *g, int c)
{
  int n = g→n; C = c;
  ⟨ Assign a direction to each edge of g and initialize the fields of arcs and vertices 101 ⟩
```

```
  ⟨ Allocate space for C^2 bicategories 109 ⟩
  ⟨ Assign each arc to one of its two endpoints 114 ⟩
  UFinit(g→vertices, n);
  return 1;
}
```

Veja também blocos 99, 126, 127, 130, 145 e 164.

Este código é citado nos blocos 95 e 97.

Este código é usado no bloco 95.

O arquivo-cabeçalho `bicat.h` é utilizado para que outros programas possam ter acesso às funções que se encontram em ⟨ External functions of `bicat.c` 96 ⟩. Além do protótipo dessas funções, ele também guarda a definição de macros importantes, como é o caso de *category*.

97 ⟨ `bicat.h` 97 ⟩ ≡

```
#define category z.I
```

Veja também blocos 102, 129 e 165.

Assim, um dos arquivos-cabeçalho que estão incluídos em `bicat.c` é `bicat.h`. Além dele, também incluímos em `bicat.c` o arquivo `uf.h`. Cada vez que um arco uv é contraído, será escolhido, para representar o vértice resultante da contração, a estrutura **Vertex** do vértice de maior peso dentre u e v. Para associar o novo vértice a esta estrutura e para determinar as estruturas do tipo **Vertex** que correspondem aos extremos de um dado arco do grafo, utilizamos a estrutura de dados union-find que tem as funções para a sua manipulação prototipadas no arquivo `uf.h`.

98 ⟨ Header files of `bicat.c` 98 ⟩ ≡

```
#include <stdlib.h>
#include "gb_graph.h"
#include "extra.h"
#include "bicat.h"
#include "uf.h"
```

Veja também blocos 107, 112, 116 e 123.

Este código é usado no bloco 95.

Abaixo, definimos outras duas funções que se encontram prototipadas em `bicat.h`. A função *From* recebe um arco a e devolve o vértice que é ponta inicial de a no grafo. A função *Tip* recebe um arco a e devolve o vértice que é ponta final de a no grafo.

99 #define *from* *a.V*

⟨ External functions of bicat.c 96 ⟩ +≡

```
Vertex *From(Arc *a)
{
  return UFfind(a→from);
}
Vertex *Tip(Arc *a)
{
  return UFfind(a→tip);
}
```

Uma das variáveis globais de bicat.c é a variável C que guarda o número de categorias em que os vértices do grafo g estão particionados.

100 ⟨ Global variables of bicat.c 100 ⟩ ≡

```
int C;
```

Veja também blocos 103, 110, 115, 119, 128, 136 e 158.

Este código é usado no bloco 95.

Orientaremos cada aresta a do grafo g de forma que a ponta inicial de a seja o extremo de a que possui o menor índice no vetor *g→vertices*. Levando em conta essa convenção, o trecho de código abaixo inicializa, para cada vértice v de g, o campo de v que guarda o seu grau de saída. Para acessar este campo, é usada a macro *out_degree*(v). Aproveitamos a ocasião para também inicializar os outros campos dos vértices e alguns dos campos dos arcos de g. Usaremos a macro *id*(v) para acessar o campo de v que guarda um inteiro que identifica, de maneira única, o vértice v e usaremos a macro *weight*(v) para acessar o campo do vértice v responsável por armazenar o peso de v. Além disso, para cada arco a, utilizaremos a macro *Delta*(a) para acessar o campo de a que guarda o valor de Δ_a.

101 #define *Delta*(a) (ARC_FIELDS(a)→*e3.D*)

#define *id*(v) (VERTEX_FIELDS(v)→*e1.I*)

#define *out_degree*(v) (VERTEX_FIELDS(v)→*e2.I*)

⟨ Assign a direction to each edge of g and initialize the fields of arcs and vertices 101 ⟩ ≡

```
{
  int i, j = 0;
  ⟨ Initialize some auxiliary variables 104 ⟩
  if (¬(min_cost_arc = malloc(n * sizeof(Arc *)))) return 0;
```

```
for (i = 0; i < n; i++) {
  Arc *a;
  Vertex *v = g→vertices + i;
  id(v) = i;
  out_degree(v) = 0;
  for (a = v→arcs; a; a = a→next) {
    if (v < a→tip) {
      Delta(a) = a→len;
      out_degree(v)++;
    }
  }
  weight(v) = 1;
  ⟨Initialize the lists extra⁻(v) and extra⁺(v) 105⟩
  min_cost_arc[i] = Λ;
}
}
```

Este código é usado no bloco 96.

A definição da macro *weight*(v) se encontra no arquivo `bicat.h` e, deste modo, esta macro pode ser utilizada por todos os programas que incluem `bicat.h`.

102 ⟨`bicat.h` 97⟩ +≡
#**define** *weight*(v) (`VERTEX_FIELDS`(v)→*e5.I*)

Além de inicializar os campos de vertices e arcos de g, o código correspondente ao bloco 101 também inicializa o vetor *min_cost_arc*, que é outra das variáveis globais de `bicat.c`. Dado um vértice v, *min_cost_arc*$[i]$ será usado para guardar, para cada i entre 0 e g→$n - 1$, um arco de custo mínimo dentre aqueles que têm a ponta inicial no vértice u tal que $id(u) \equiv i$ e a ponta final em v. Caso não haja arcos entre u e v, *min_cost_arc*$[i] \equiv \Lambda$. Como veremos mais adiante, esse vetor será utilizado quando for necessário descartar arcos de alguma lista $extra^+(v)$.

103 ⟨Global variables of `bicat.c` 100⟩ +≡ **Arc** ***min_cost_arc*;

Para cada vértice v, cada uma das listas de arcos $extra^-(v)$ e $extra^+(v)$ é implementada através de uma lista circular duplamente ligada com cabeça. Usaremos as macros *extra_out*(v) e *extra_in*(v) para acessar os campos de v que guardam apontadores para os nós-cabeça das listas que implementam, respectivamente, $extra^-(v)$ e $extra^+(v)$. O vetor *head*, que inicializamos no

trecho de código abaixo, é usado para guardar todos os nós-cabeça de listas deste tipo. Para cada arco a em uma lista destas, usaremos as macros $prev(a)$ e $next(a)$ para acessar os campos de a que guardam apontadores, respectivamente, para o arco que antecede a e para aquele que sucede a na lista. Pelo modo como estas macros são definidas, devemos alocar um estrutura do tipo **ExtraFields** para cada um dos nós-cabeça das listas. Tais estruturas, ficarão guardadas no vetor af.

```
104 #define extra_in(v)  (VERTEX_FIELDS(v)→e3.P)
    #define extra_out(v)  (VERTEX_FIELDS(v)→e4.P)
    #define prev(a)  (ARC_FIELDS(a)→e1.P)
    #define next(a)  (ARC_FIELDS(a)→e2.P)
    ⟨Initialize some auxiliary variables 104⟩ ≡
      Arc *head = malloc(2 * n * sizeof(Arc));
      ExtraFields *af = malloc(2 * n * sizeof(ExtraFields));
      if (¬head ∨ ¬af) return 0;
```

Este código é usado no bloco 101.

O trecho abaixo inicializa os campos de v correspondentes às macros $extra_in(v)$ e $extra_in(v)$. Note que, logo após a inicialização, as listas correspondentes a estes campos estarão vazias.

```
105 ⟨Initialize the lists extra⁻(v) and extra⁺(v) 105⟩ ≡
      a = head + i;
      a→afields = (char *)(af + j++);
      prev(a) = next(a) = a;
      extra_in(v) = a;
      a = head + n + i;
      a→afields = (char *)(af + j++);
      prev(a) = next(a) = a;
      extra_out(v) = a;
```

Este código é usado no bloco 101.

A cada um dos heaps utilizados na implementação, corresponderá uma estrutura do tipo **struct labeled_heap**. Cada estrutura lh deste tipo contém um campo *heap* que guarda o heap h correspondente a lh e um campo *Delta* que armazena o valor corrente de Δ_h. Os apontadores para estruturas deste tipo são objetos do tipo **LbHeap**.

```
106 ⟨Data structures of bicat.c 106⟩ ≡
      struct labeled_heap {
```

```
  double Delta;
  FibHeap heap;
};
typedef struct labeled_heap *LbHeap;
```

Veja também bloco 108.

Este código é usado no bloco 95.

107 ⟨Header files of bicat.c 98⟩ +≡

```
#include "fibheap.h"
```

Cada uma das bicategorias é implementada através de uma estrutura do tipo **Bicategory**. Dada uma bicategoria b, o campo H da estrutura correspondente a b implementa o heap $H(b)$ e os campos *in* e *out* são tais que, para cada vértice v do grafo, $in[id(v)]$ e $out[id(v)]$ guardam os nós de $H\to heap$ onde se encontram os heaps que implementam, respectivamente, $H^+(v,b)$ e $H^-(v,b)$.

108 ⟨Data structures of bicat.c 106⟩ +≡

```
typedef struct bicategory {
  LbHeap H;
  FHnode *in;
  FHnode *out;
} Bicategory;
```

O trecho abaixo aloca memória para as estruturas correspondentes às C^2 bicategorias em que os arcos de g se encontram particionados e também inicializa cada um dos campos dessas estruturas. A função *HEAPless* fornecida como argumento na inicialização dos heaps no trecho abaixo será exibida adiante. As estruturas do tipo **Bicategory** alocadas ficarão todas armazenadas em um vetor e a variável global *b0* guardará um apontador para a estrutura que se encontra na primeira posição desse vetor. Tal estrutura corresponderá à bicategoria $(0,0)$. As estruturas em cada uma das $C-1$ posições seguintes do vetor corresponderão às bicategorias $(0,1),\ldots,(0,C-1)$ nesta ordem. Aquelas que se encontram nas C posições seguintes corresponderão às bicategorias $(1,0),\ldots,(1,C-1)$ e assim por diante. A figura 4.1 ilustra a disposição das estruturas correspondentes às bicategorias no vetor.

De acordo com esta disposição, dada uma bicategoria (x,y), a posição do vetor em que se encontra a estrutura correspondente a ela é $x\cdot C+y$. Usaremos a macro $bicat_id(x,y)$ para determinar tal posição. Como veremos mais adiante, este número também será usado como um identificador da bicategoria (x,y).

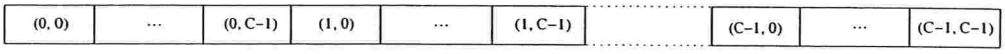

Figura 4.1: Ilustração da disposição das estruturas **Bicategory** no vetor.

```
109 #define bicat_id(x, y)  (x * C + y)
    ⟨ Allocate space for C^2 bicategories 109 ⟩ ≡
      {
        int i;
        LbHeap lh = malloc(C * C * sizeof(struct labeled_heap));
        FHnode *nodes = malloc(2 * n * C * C * sizeof(FHnode));
        b0 = malloc(C * C * sizeof(Bicategory));
        if (¬lh ∨ ¬nodes ∨ ¬b0) return 0;
        if (¬FHEAPalloc(2 * n * C * C + C * C, 2 * n * C * C + g→m/2)) return 0;
        for (i = 0; i < C * C; i++) {
          Bicategory *B = b0 + i;
          B→H = lh + i;
          B→H→heap = FHEAPinit(HEAPless);
          B→H→Delta = 0;
          B→in = nodes + 2 * n * i;
          B→out = B→in + n;
        }
        ⟨ Allocate memory to other variables 118 ⟩
      }
```

Este código é usado no bloco 96.

```
110 ⟨ Global variables of bicat.c 100 ⟩ +≡
      Bicategory *b0;
```

A função *heap_key* que exibimos abaixo recebe um heap h do tipo **LbHeap** e, caso $h \neq \Lambda$, devolve a chave de h no heap $H(b)$ em que ele se encontra; caso contrário, isto é, se $h \equiv \Lambda$, ela devolve −DBL_MAX.

```
111 ⟨ Internal functions of bicat.c 111 ⟩ ≡
      double heap_key(LbHeap h)
      {
        if (h) {
```

```
    if (¬FHEAPempty(h→heap)) {
      Arc *min = FHEAPfindMin(h→heap);
      return h→Delta + Delta(min);
    }
    else return DBL_MAX;
  }
  return −DBL_MAX;
}
```

Veja também blocos 113, 120, 121, 155 e 162.

Este código é usado no bloco 95.

112 ⟨Header files of bicat.c 98⟩ +≡

```
#include <float.h>
```

A função abaixo é usada para comparar dois heaps *h1* e *h2* do tipo **LbHeap** no heap $H(b)$ em que eles se encontram. Caso $h1 \equiv \Lambda \neq h2$ ou a chave de *h1* seja menor que a chave de *h2*, a função *HEAPless* devolve 1 e, caso $h2 \equiv \Lambda$ ou a chave de *h1* seja maior ou igual a chave de *h2*, ela devolve 0.

113 ⟨Internal functions of bicat.c 111⟩ +≡

```
static int HEAPless(void *h1, void *h2)
{
  return (heap_key(h1) < heap_key(h2));
}
```

De um certo modo, podemos dizer que a orientação das arestas do grafo g é feita, de fato, no trecho abaixo. Aqui, para cada aresta a de g, apenas um dos dois arcos que representam a em g, aquele cuja orientação está de acordo com a que definimos no bloco 101, é inserido nas estruturas de dados usadas para manter o rastro dos arcos do grafo. Os arcos que ficarem de fora dessas estruturas serão completamente ignorados pelas operações sobre as bicategorias e, desta forma, do ponto de vista destas operações, tudo se passa como se apenas os arcos cuja orientação está de acordo com a definida exitissem no grafo, ou seja, como se tivéssemos orientado as arestas de g de acordo com orientação desses arcos. No trecho de código abaixo, atribuiremos cada um dos arcos obtidos através dessa orientação das arestas de g a um de seus dois extremos.

114 ⟨Assign each arc to one of its two endpoints 114⟩ ≡

```
{
  Vertex *v;
```

```
    sqrt_m = sqrt(g→m/2);
    for (v = g→vertices + n − 1; v ≥ g→vertices; v—) {
      Arc *a;
      ⟨For each bicategory b, create the heaps H⁻(v,b) and H⁺(v,b) 117⟩
      for (a = v→arcs; a; a = a→next) {
        int bicat_id;
        Vertex *w = a→tip;
        if (v > w) continue;
        bicat_id = bicat_id(v→category, w→category);
        if (out_degree(v) ≥ sqrt_m) ⟨Assign a to v 122⟩
        else ⟨Assign a to w 124⟩
      }
    }
    ⟨For each v and each b insert H⁻(v,b) and H⁺(v,b) in H(b) 125⟩
  }
```

Este código é usado no bloco 96.

115 ⟨Global variables of `bicat.c` 100⟩ +≡

```
double sqrt_m;
```

116 ⟨Header files of `bicat.c` 98⟩ +≡

```
#include <math.h>
```

A inicialização dos heaps $H^-(v,b)$ e $H^+(v,b)$ para cada bicategoria b é feita pelo trecho de código abaixo. A função *ARCless* usada nessas inicializações será exibida mais adiante. A variável global `LH` é o vetor onde se encontram todas as estruturas do tipo **struct labeled_heap** que correspondem a estes heaps. Particionaremos este vetor em n fatias correspondentes a cada um dos n vértices de g. Para cada vértice v, a fatia correspondente a v será constituída de $2C^2$ posições adjacentes. As primeiras C^2 posições da fatia corresponderão aos C^2 heaps $H^+(v,b)$ e as C^2 posições seguintes, aos C^2 heaps $H^-(v,b)$. Estes heaps estarão dispostos de acordo com a posição das bicategorias no vetor da figura 4.1, isto é, se a posição de $b1$ no vetor da figura antecede a de $b2$, $H^-(v,b1)$ antecederá $H^-(v,b2)$ no vetor `LH`.

117 ⟨For each bicategory b, create the heaps $H^-(v,b)$ and $H^+(v,b)$ 117⟩ ≡

```
int i;
LbHeap h_in_v = LH + (v − g→vertices) * 2 * C * C, h_out_v = h_in_v + C * C;
```

```
for (i = 0; i < C * C; i++) {
  h_in_v[i].heap = FHEAPinit(ARCless);
  h_out_v[i].heap = FHEAPinit(ARCless);
  h_in_v[i].Delta = h_out_v[i].Delta = 0;
}
```

Este código é usado no bloco 114.

118 ⟨Allocate memory to other variables 118⟩ ≡

```
if (¬(LH = malloc(2 * C * C * n * sizeof(struct labeled_heap)))) return 0;
```

Veja também bloco 157.

Este código é usado no bloco 109.

119 ⟨Global variables of bicat.c 100⟩ +≡ **LbHeap** LH;

A função *arc_key* que exibimos abaixo recebe um arco a e, se $a \neq \Lambda$, devolve a chave de a no heap em que ele se encontra, isto é, Δ_a; caso contrário, ela devolve −DBL_MAX.

120 ⟨Internal functions of bicat.c 111⟩ +≡

```
double arc_key(Arc *a)
{
  if (¬a) return −DBL_MAX;
  return Delta(a);
}
```

Para comparar dois arcos a e b que estão em algum dos heaps é usada a função *ARCless*. Ela recebe os arcos a e b e, caso $a \equiv \Lambda \neq b$ ou a chave de a seja menor que a chave de b, ela devolve 1; caso $b \equiv \Lambda$ ou a chave de a seja maior ou igual a chave de b, ela devolve 0.

121 ⟨Internal functions of bicat.c 111⟩ +≡

```
int ARCless(void *a, void *b)
{
  return (arc_key(a) < arc_key(b));
}
```

Para cada arco a, usaremos um dos campos de a para armazenar o nó que contém a num dos heaps correspondentes ao vértice ao qual a está atribuído. A macro $node(a)$ será usada para acessar tal campo. Para inserir e remover arcos das listas *extra*, usaremos as funções *insert_edge* e *delete_edge* que se encontram prototipadas em `list.h`.

122 #define $node(a)$ (ARC_FIELDS(a)→$e4.P$)

⟨ Assign a to v 122 ⟩ ≡

```
{
  node(a) = FHEAPinsert(h_out_v[bicat_id].heap, a);
  insert_edge(extra_in(w), a);
}
```

Este código é usado no bloco 114.

123 ⟨ Header files of `bicat.c` 98 ⟩ +≡

```
#include "list.h"
```

124 ⟨ Assign a to w 124 ⟩ ≡

```
{
  LbHeap h_in_w = LH + (w - g→vertices) * 2 * C * C;
  node(a) = FHEAPinsert(h_in_w[bicat_id].heap, a);
  insert_edge(extra_out(v), a);
}
```

Este código é usado no bloco 114.

125 ⟨ For each v and each b insert $H^-(v,b)$ and $H^+(v,b)$ in $H(b)$ 125 ⟩ ≡

```
{
  int i, j;
  for (i = 0; i < n; i++) {
    LbHeap h_in = LH + i * 2 * C * C, h_out = h_in + C * C;
    for (j = 0; j < C * C; j++) {
      Bicategory *b = b0 + j;
      b→in[i] = FHEAPinsert(b→H→heap, h_in + j);
      b→out[i] = FHEAPinsert(b→H→heap, h_out + j);
    }
  }
}
```

Este código é usado no bloco 114.

A função *decreaseCost* recebe um número racional *delta* e dois inteiros *cat1* e *cat2* tais que $0 \leq cat1 \leq C-1$ e $0 \leq cat2 \leq C-1$ e diminui de *delta* o custo de cada um dos arcos que se encontram na bicategoria $(cat1, cat2)$.

126 ⟨External functions of `bicat.c` 96⟩ +≡

```
void decreaseCost(int cat1, int cat2, double delta)
{
  Bicategory *b = b0 + bicat_id(cat1, cat2);
  b→H→Delta −= delta;
}
```

A função *findMin* recebe dois inteiros *cat1* e *cat2* tais que $0 \leq cat1 \leq C-1$ e $0 \leq cat2 \leq C-1$ e devolve um arco de custo mínimo dentre aqueles que se encontram na bicategoria $(cat1, cat2)$. Além disso, o custo do arco devolvido pela função é guardado na variável global *min_cost*. Caso não existam arcos na bicategoria $(cat1, cat2)$, *findMin* devolve Λ.

127 ⟨External functions of `bicat.c` 96⟩ +≡

```
Arc *findMin(int cat1, int cat2)
{
  Bicategory *b = b0 + bicat_id(cat1, cat2);
  LbHeap h_min = FHEAPfindMin(b→H→heap);
  if (¬FHEAPempty(h_min→heap)) {
    Arc *min = FHEAPfindMin(h_min→heap);
    min_cost = Delta(min) + h_min→Delta + b→H→Delta;
    return min;
  }
  return Λ;
}
```

128 ⟨Global variables of `bicat.c` 100⟩ +≡

```
double min_cost;
```

A declaração da variável *min_cost* também se encontra em `bicat.h` para que os programas que incluam este arquivo possam ter acesso ao conteúdo desta variável.

129 ⟨`bicat.h` 97⟩ +≡

```
extern double min_cost;
```

A função *changeCategory*, apresentada abaixo, recebe um vértice v e um inteiro *new_cat* tal que $0 \leq new_cat \leq C - 1$ e $new_cat \neq v\rightarrow category$ e altera para *new_cat* a categoria de v. A bicategoria de cada um dos arcos incidentes em v é implicitamente alterada pela função.

```
130 ⟨External functions of bicat.c 96⟩ +≡
    void changeCategory(Vertex *v, int new_cat)
    {
      int old_cat = v→category;
      ⟨Other local variables of changeCategory 138⟩
      ⟨Change the bicategory of the arcs assigned to v 131⟩
      ⟨Discard parallel arcs in the extra_in list of v 134⟩
      ⟨Change the bicategory of each arc in the extra_in list of v 137⟩
      ⟨Change the bicategory of each arc in the extra_out list of v 142⟩
      v→category = new_cat;
    }
```

```
131 ⟨Change the bicategory of the arcs assigned to v 131⟩ ≡
    {
      int id_v = id(v), c;
      for (c = 0; c < C; c++) {
        Bicategory *old, *new;
        old = b0 + bicat_id(c, old_cat);
        new = b0 + bicat_id(c, new_cat);
        ⟨Move to the bicategory new the arcs in H+(v, old) 132⟩
        old = b0 + bicat_id(old_cat, c);
        new = b0 + bicat_id(new_cat, c);
        ⟨Move to the bicategory new the arcs in H−(v, old) 133⟩
      }
    }
```

Este código é usado no bloco 130.

No trecho abaixo, o heap *h1*, que corresponde a $H^+(v, old)$, é removido do heap da bicategoria *old* e inserido no heap da bicategoria *new*. Para que o custo dos arcos em *h1* não seja alterado devido a esta mudança, o valor de *h1→Delta* é devidamente atualizado. Note que o heap guardado na variável *h2*, que corresponde a $H^+(v, new)$, está vazio, pois, como $new \equiv (c, new_cat)$ e $new_cat \neq old_cat$, nenhum dos arcos na bicategoria *new* possuía a ponta final em v.

132 ⟨Move to the bicategory *new* the arcs in $H^+(v, old)$ 132⟩ ≡

```
{
  LbHeap h1 = FHEAPdelete(old→H→heap, old→in[id_v]);
  LbHeap h2 = FHEAPdelete(new→H→heap, new→in[id_v]);
  h1→Delta += old→H→Delta − new→H→Delta;
  new→in[id_v] = FHEAPinsert(new→H→heap, h1);
  h2→Delta = 0;
  old→in[id_v] = FHEAPinsert(old→H→heap, h2);
}
```

Este código é usado no bloco 131.

Os comentários feitos no bloco anterior se aplicam, de forma similar, ao trecho de código abaixo.

133 ⟨Move to the bicategory *new* the arcs in $H^-(v, old)$ 133⟩ ≡

```
{
  LbHeap h1 = FHEAPdelete(old→H→heap, old→out[id_v]);
  LbHeap h2 = FHEAPdelete(new→H→heap, new→out[id_v]);
  h1→Delta += old→H→Delta − new→H→Delta;
  new→out[id_v] = FHEAPinsert(new→H→heap, h1);
  h2→Delta = 0;
  old→out[id_v] = FHEAPinsert(old→H→heap, h2);
}
```

Este código é usado no bloco 131.

Aqui são descartados da lista *extra_in*(v) arcos que possuem a ponta inicial em comum (e que, desta forma, são paralelos). Isto é feito de modo que apenas um destes arcos, cujo custo é mínimo, permaneça na lista. Removeremos estes arcos não somente da lista *extra_in*(v), mas também dos heaps onde eles estão guardados. Para realizar esta tarefa, contaremos com o auxílio do vetor *min_cost_arc*, que, imediatamente antes da execução deste código, tem o conteúdo de cada uma de suas posições preenchido com Λ.

134 ⟨Discard parallel arcs in the *extra_in* list of v 134⟩ ≡

```
{
  head = extra_in(v); a = next(head);
  while (a ≠ head) {
    Arc *next = next(a);
```

```
      Vertex *u = From(a);
      int id_u = id(u);
      if (¬min_cost_arc[id_u]) min_cost_arc[id_u] = a;
      else {
        Arc *b = min_cost_arc[id_u];
        ⟨ if b's cost is greater than a's cost, discard b; else discard a 135 ⟩
      }
      a = next;
    }
    _min_[indx++ % 2] = min_cost_arc[pos];
  }
```

Este código é usado nos blocos 130 e 145.

No trecho abaixo, a e b são arcos da lista $extra_in(v)$ que têm a ponta inicial em u. Por terem a ponta inicial em comum, eles estão guardados em um mesmo heap e, assim, para saber qual dos dois tem menor custo, basta comparar $Delta(a)$ e $Delta(b)$.

```
135 ⟨ if b's cost is greater than a's cost, discard b; else discard a 135 ⟩ ≡
  {
    Bicategory *B = b0 + bicat_id(u→category, v→category);
    LbHeap h_out_u = FHEAPdelete(B→H→heap, B→out[id_u]);
    if (Delta(a) < Delta(b)) {
      min_cost_arc[id_u] = a;
      delete_edge(b);
      FHEAPdelete(h_out_u→heap, node(b));
    }
    else {
      delete_edge(a);
      FHEAPdelete(h_out_u→heap, node(a));
    }
    B→out[id_u] = FHEAPinsert(B→H→heap, h_out_u);
  }
```

Este código é usado no bloco 134.

Para cada vértice u que tem grau de saída alto, se existe algum arco no grafo que tenha a ponta inicial em u e a ponta final em v, $min_cost_arc[id(u)]$ guardará, ao final da execução do **while**

acima, o único arco deste tipo que não foi descartado; caso contrário, $min_cost_arc[id(u)] \equiv \Lambda$. Após a execução de *changeCategory*, a função *contractEdge*, que veremos mais adiante, eventualmente precisará da informação contida em alguma das posições do vetor *min_cost_arc*. Por outro lado, antes do final da execução de *changeCategory*, será necessário reinicializar o conteúdo das posições de *min_cost_arc*, deixando o vetor preenchido da mesma forma que ele se encontrava antes da execução do **while**. Para resolver este impasse, usamos as variáveis globais que declaramos abaixo para guardar a informação que poderá ser usada por *contractEdge*.

136 ⟨ Global variables of `bicat.c` 100 ⟩ +≡

```
Arc *_min_[2];
int indx = 0, pos = 0;
```

No momento em que o trecho abaixo é executado, *head* guarda um apontador para o nó cabeça da lista *extra_in*(*v*). Cada um dos arcos nesta lista é removido do heap $H^-(u,old)$ e inserido no heap $H^-(u,new)$. Vamos aproveitar a ocasião para reinicializar o conteúdo das posições do vetor *min_cost_arc*.

137 ⟨ Change the bicategory of each arc in the *extra_in* list of *v* 137 ⟩ ≡

```
for (a = next(head); a ≠ head; a = next(a)) {
  Vertex *u = From(a);
  int id_u = id(u);
  Bicategory *old = b0 + bicat_id(u→category, old_cat);
  Bicategory *new = b0 + bicat_id(u→category, new_cat);
  ⟨ Remove the uv-arc a from the bicategory old 139 ⟩
  ⟨ Put the uv-arc a in the bicategory new 140 ⟩
  min_cost_arc[id_u] = Λ;
}
```

Este código é usado no bloco 130.

138 ⟨ Other local variables of *changeCategory* 138 ⟩ ≡

```
Arc *head, *a;
```

Veja também bloco 141.

Este código é usado no bloco 130.

139 ⟨ Remove the *uv*-arc *a* from the bicategory *old* 139 ⟩ ≡

```
h_old = FHEAPdelete(old→H→heap, old→out[id_u]);
FHEAPdelete(h_old→heap, node(a));
```

```
old→out[id_u] = FHEAPinsert(old→H→heap, h_old);
```

Este código é usado no bloco 137.

Note que, antes de inserir o arco a em seu novo heap, devemos alterar adequadamente o conteúdo do campo acessado por $Delta(a)$ para que o custo de a não seja alterado por esta operação.

140 ⟨ Put the uv-arc a in the bicategory *new* 140 ⟩ ≡

```
h_new = FHEAPdelete(new→H→heap, new→out[id_u]);
Delta(a) += h_old→Delta + old→H→Delta − h_new→Delta − new→H→Delta;
node(a) = FHEAPinsert(h_new→heap, a);
new→out[id_u] = FHEAPinsert(new→H→heap, h_new);
```

Este código é usado no bloco 137.

141 ⟨ Other local variables of *changeCategory* 138 ⟩ +≡

```
LbHeap h_old, h_new;
```

No trecho de código abaixo, cada um dos arcos na lista $extra_out(v)$ é removido do heap $H^+(w,old)$ e inserido no heap $H^+(w,new)$.

142 ⟨ Change the bicategory of each arc in the *extra_out* list of v 142 ⟩ ≡

```
head = extra_out(v);
for (a = next(head); a ≠ head; a = next(a)) {
  Vertex *w = Tip(a);
  int id_w = id(w);
  Bicategory *old = b0 + bicat_id(old_cat, w→category);
  Bicategory *new = b0 + bicat_id(new_cat, w→category);
  ⟨ Remove the vw-arc a from the bicategory old 143 ⟩
  ⟨ Put the vw-arc a in the bicategory new 144 ⟩
}
```

Este código é usado no bloco 130.

143 ⟨ Remove the vw-arc a from the bicategory *old* 143 ⟩ ≡

```
h_old = FHEAPdelete(old→H→heap, old→in[id_w]);
FHEAPdelete(h_old→heap, node(a));
old→in[id_w] = FHEAPinsert(old→H→heap, h_old);
```

Este código é usado no bloco 142.

144 ⟨Put the vw-arc a in the bicategory *new* 144⟩ ≡

```
h_new = FHEAPdelete(new→H→heap, new→in[id_w]);
Delta(a) += h_old→Delta + old→H→Delta − h_new→Delta − new→H→Delta;
node(a) = FHEAPinsert(h_new→heap, a);
new→in[id_w] = FHEAPinsert(new→H→heap, h_new);
```

Este código é usado no bloco 142.

A função *contractEdge* recebe um arco a e um inteiro *cat* tal que $0 \leq cat \leq C - 1$. Ela contrai a, removendo do grafo todos os arcos que têm uma das pontas em $From(a)$ e a outra em $Tip(a)$, e atribui ao vértice resultante da contração a categoria *cat*. Aqui, "remover do grafo" um arco deve ser interpretado como removê-lo da lista *extra* e do heap nos quais ele se encontra. Após a execução de *contractEdge*, o vértice resultante da contração será representado pela estrutura **Vertex** de um vértice de peso máximo dentre $From(a)$ e $Tip(a)$ (em caso de empate, a estrutura de $From(a)$ é usada para representar o novo vértice).

145 ⟨External functions of `bicat.c` 96⟩ +≡

```
void contractEdge(Arc *a, int cat)
{
  ⟨Local variables of contractEdge 146⟩
  x = From(a); y = Tip(a);
  id[0] = id(x); id[1] = id(y); indx = 0;
  for (v = x, i = 0; i ≤ 1; i++, v = y) {
    pos = id[1 − i];
    if (v→category ≠ cat) changeCategory(v, cat);
    else {
      ⟨Discard parallel arcs in the extra_in list of v 134⟩
      ⟨Clear the positions of the vector min_cost_arc 147⟩
    }
  }
  ⟨Discard all xy-arcs from the graph 149⟩
  ⟨Discard all yx-arcs from the graph 152⟩
  if (weight(x) < weight(y)) {
    Vertex *tmp = x; x = y; y = tmp;
  }
  ⟨For each bicategory b, join the heaps of x and y in b 153⟩
  ⟨Change the assignment of the arcs incident to x or y if necessary 159⟩
  concat(extra_in(x), extra_in(y)); concat(extra_out(x), extra_out(y));
  ⟨Set x to represent the vertex resulting from the contraction of a 163⟩
```

}

Aqui, declaramos algumas das variáveis locais de *contractEdge*. Como pode ser visto acima, inicialmente, a categoria de cada um dos vértices x e y, que são extremos de a, é alterada, se necessário, através de uma chamada a *changeCategory*. Caso a categoria de um destes vértices já seja igual a *cat*, não será necessário fazer uma chamada a *changeCategory* para ele, mas ainda assim é preciso eliminar arcos paralelos em sua lista *extra_in*. Em ambos os casos, as variáveis globais *indx* e *pos* serão alteradas de modo a guardar em *_min_*[0] e em *_min_*[1] o conteúdo das posições $id(x)$ e $id(y)$ do vetor *min_cost_arc* antes que este seja reinicializado.

146 ⟨ Local variables of *contractEdge* 146 ⟩ ≡

```
int i, id[2];
Vertex *v, *x, *y;
```

Veja também blocos 148 e 151.

Este código é usado no bloco 145.

Abaixo, o vetor *min_cost_arc* é reinicializado após o descarte de arcos pararelos em *extra_in*(v). A variável *head* guarda um apontador para o nó cabeça desta lista.

147 ⟨ Clear the positions of the vector *min_cost_arc* 147 ⟩ ≡

```
{
  Arc *b;
  for (b = next(head); b ≠ head; b = next(b)) min_cost_arc[id(From(b))] = Λ;
}
```

Este código é usado no bloco 145.

148 ⟨ Local variables of *contractEdge* 146 ⟩ +≡

```
Arc *head;
```

No trecho abaixo, todo arco com a ponta inicial em x e a ponta final em y será removido da lista e do heap nos quais se encontra guardado. Se x tem grau de saída alto, sabemos que todos estes arcos se encontram na lista *extra_in*(y). Observe que, em tal lista, só pode existir um único arco deste tipo, já que, no início de *contractEdge*, todos os arcos paralelos de *extra_in*(y) foram descartados. Observe ainda que, nesta ocasião, tal arco foi armazenado em *_min_*[1]. Por fim, note que existe pelo menos um tal arco, já que a é um xy-arco. Assim, neste caso, basta remover o arco que se encontra em *_min_*[1] da lista *extra_in*(y) e do heap $H^-(x, B)$, onde B é a bicategoria do arco.

149 ⟨Discard all xy-arcs from the graph 149⟩ ≡

```
{
  B = b0 + bicat_id(x→category, y→category);
  id_x = id(x); id_y = id(y);
  if (out_degree(x) < sqrt_m) {
    Vertex *u = x; v = y;
    ⟨Discard all uv-arcs, knowing that this arcs are assigned to v 150⟩
  }
  else {
    LbHeap h_out_x = FHEAPdelete(B→H→heap, B→out[id_x]);
    Arc *xy = _min_[1];
    delete_edge(xy);
    FHEAPdelete(h_out_x→heap, node(xy));
    B→out[id_x] = FHEAPinsert(B→H→heap, h_out_x);
  }
}
```

Este código é usado no bloco 145.

Por outro lado, caso x não tenha grau de saída alto, todos os xy-arcos se encontram em *extra_out*(x) e será necessário percorrer esta lista para descartar tais arcos. Lembre-se de que tal lista contém no máximo $\sqrt{m}$ arcos e, assim, o tempo gasto pelo código abaixo para executar esta tarefa é $O(\sqrt{m}\log n)$ amortizado (a amortização vem do uso das funções *FHEAPdelete* e *Tip*). No trecho abaixo, os vértices u e v estão nos papéis de x e y respectivamente.

150 ⟨Discard all uv-arcs, knowing that this arcs are assigned to v 150⟩ ≡

```
{
  int id_v = id(v);
  LbHeap h_in_v = FHEAPdelete(B→H→heap, B→in[id_v]);
  Arc *head = extra_out(u), *b = next(head);
  while (b ≠ head) {
    Arc *next = next(b);
    if (Tip(b) ≡ v) {
      delete_edge(b);
      FHEAPdelete(h_in_v→heap, node(b));
    }
    b = next;
  }
  B→in[id_v] = FHEAPinsert(B→H→heap, h_in_v);
```

```
}
```

Este código é usado nos blocos 149 e 152.

151 ⟨ Local variables of *contractEdge* 146 ⟩ +≡

```
int id_x, id_y;
Bicategory *B;
```

No trecho abaixo, todo arco com a ponta inicial em y e a ponta final em x será removido da lista e do heap nos quais se encontra. A única diferença do código abaixo para aquele no bloco 149 é que aqui há a possibilidade de não existirem yx-arcos a serem descartados e, desta forma, o conteúdo de _min_[0] pode ser igual a Λ.

152 ⟨ Discard all yx-arcs from the graph 152 ⟩ ≡

```
B = b0 + bicat_id(y→category, x→category);
if (out_degree(y) < sqrt_m) {
  Vertex *u = y; v = x;
  ⟨ Discard all uv-arcs, knowing that this arcs are assigned to v 150 ⟩
}
else {
  LbHeap h_out_y = FHEAPdelete(B→H→heap, B→out[id_y]);
  Arc *yx = _min_[0];
  if (yx) {
    delete_edge(yx);
    FHEAPdelete(h_out_y→heap, node(yx));
  }
  B→out[id_y] = FHEAPinsert(B→H→heap, h_out_y);
}
```

Este código é usado no bloco 145.

Uma vez que todos os arcos que têm um dos extremos em x e o outro em y foram descartados, é o momento de criar o vértice resultante da contração de a. Para representar este vértice, usaremos a estrutura **Vertex** de um vértice cujo peso é máximo dentre x e y. Como pode ser visto no bloco 145, arranjamos as coisas de modo que a variável x guarde um apontador para tal estrutura. Neste e nos próximos blocos, alteraremos os campos desta e das demais estruturas relacionadas a x de modo que elas representem o novo vértice do grafo. Abaixo, é feita a união dos heaps correspondentes a x e a y para cada uma das C^2 bicategorias. Note que, como o

conteúdo das variáveis x e y pode ter sido trocado, precisamos preencher novamente o conteúdo das variáveis id_x e id_y.

153 ⟨For each bicategory b, join the heaps of x and y in b 153⟩ ≡

```
{
  id_x = id(x); id_y = id(y);
  for (B = b0; B < b0 + C * C; B++) {
    ⟨Join the heaps H+(x, B) and H+(y, B) 154⟩
    ⟨Join the heaps H−(x, B) and H−(y, B) 156⟩
  }
}
```

Este código é usado no bloco 145.

Aqui é feita a união dos heaps $H^+(x, B)$ e $H^+(y, B)$ e o resultado desta união é guardado na estrutura correspondente a $H^+(x, B)$. Como, mesmo após a união, manteremos inalterado o campo *Delta* desta estrutura, devemos modificar adequadamente *Delta*(a) para cada arco a no heap $H^+(y, B)$, de modo que o custo destes arcos não seja afetado por esta operação. Para isto, contaremos com o auxílio da função *inc_field_delta* que veremos no próximo bloco.

154 ⟨Join the heaps $H^+(x, B)$ and $H^+(y, B)$ 154⟩ ≡

```
{
  void *args[1];
  LbHeap h_in_x = FHEAPdelete(B→H→heap, B→in[id_x]);
  LbHeap h_in_y = FHEAPdelete(B→H→heap, B→in[id_y]);
  double inc = h_in_y→Delta − h_in_x→Delta;
  args[0] = &inc;
  FHEAPtraverse(h_in_y→heap, inc_field_delta, args);
  h_in_x→heap = FHEAPjoin(h_in_x→heap, h_in_y→heap);
  B→in[id_x] = FHEAPinsert(B→H→heap, h_in_x);
}
```

Este código é usado no bloco 153.

A função *inc_field_delta* recebe um arco *item* e um apontador para **double**, guardado em *args*[0], e incrementa, de um valor igual ao conteúdo da variável apontada por *args*[0], o conteúdo do campo acessado por *Delta*(*item*).

155 ⟨Internal functions of `bicat.c` 111⟩ +≡

```
static void inc_field_delta(void *item, void *args[])
```

```
{
  Arc *a = item;
  double inc = *((double *) args[0]);
  Delta(a) += inc;
}
```

A união dos heaps $H^-(x, B)$ e $H^-(y, B)$ é feita abaixo. As operações realizadas neste trecho são similares àquelas feitas no código do bloco 154, com exceção da última linha, na qual, ao invés de guardarmos o heap resultante da união no heap da bicategoria correspondente, o guardamos no vetor H_out_x, na posição correspondente ao identificador da bicategoria. Faremos isto, pois talvez ainda seja necessário inserir mais alguns arcos neste heap, como veremos adiante.

156 ⟨ Join the heaps $H^-(x, B)$ and $H^-(y, B)$ 156 ⟩ ≡

```
{
  void *args[1];
  LbHeap h_out_x = FHEAPdelete(B→H→heap, B→out[id_x]);
  LbHeap h_out_y = FHEAPdelete(B→H→heap, B→out[id_y]);
  double inc = h_out_y→Delta − h_out_x→Delta;
  args[0] = &inc;
  FHEAPtraverse(h_out_y→heap, inc_field_delta, args);
  h_out_x→heap = FHEAPjoin(h_out_x→heap, h_out_y→heap);
  H_out_x[B − b0] = h_out_x;
}
```

Este código é usado no bloco 153.

157 ⟨ Allocate memory to other variables 118 ⟩ +≡

```
if (¬(H_out_x = malloc(C * C * sizeof(LbHeap)))) return 0;
```

158 ⟨ Global variables of `bicat.c` 100 ⟩ +≡

```
LbHeap *H_out_x;
```

Caso o vértice resultante da contração de a tenha grau de saída alto, mas o mesmo não seja verdade para algum extremo v de a, deve-se mudar a atribuição dos arcos que têm a ponta inicial em v, os quais se encontram em $extra_out(v)$. É exatamente isto que é feito pelo trecho de código abaixo. Ao final de todo este processo, iremos reinserir, no heap da bicategoria adequada, cada um dos heaps que se encontram no vetor H_out_x.

159 ⟨ Change the assignment of the arcs incident to x or y if necessary 159 ⟩ ≡

```
{
  int out_degree_x = out_degree(x), out_degree_y = out_degree(y);
  if (out_degree_x + out_degree_y ≥ sqrt_m)
    for (i = 1, v = x; i ≤ 2; i++, v = y)
      if (out_degree(v) < sqrt_m) {
        Arc *head = extra_out(v), *b = next(head);
        while (b ≠ head) {
          Arc *next = next(b);
          Vertex *w = Tip(b);
          int bicat_id = bicat_id(v→category, w→category);
          B = b0 + bicat_id;
          ⟨ Change the endpoint which b is assigned to 160 ⟩
          b = next;
        }
      }
  ⟨ For each bicategory B, reinsert in H(B) the heaps in H_out_x 161 ⟩
}
```

Este código é usado no bloco 145.

Aqui, o arco b, que antes se encontrava atribuído ao vértice w, será atribuído ao vértice resultante da contração de a. Note que, antes de inserir b em $H_out_x[B - b0]$, o heap que corresponde ao vértice resultante da contração de a na bicategoria B a que b pertence, devemos alterar adequadamente o valor de $Delta(b)$ para que o custo de b não seja afetado devido à sua mudança de heap.

160 ⟨ Change the endpoint which b is assigned to 160 ⟩ ≡

```
{
  int id_w = id(w);
  LbHeap h_w = FHEAPdelete(B→H→heap, B→in[id_w]);
  FHEAPdelete(h_w→heap, node(b));
  B→in[id_w] = FHEAPinsert(B→H→heap, h_w);
  Delta(b) += h_w→Delta − H_out_x[bicat_id]→Delta;
  node(b) = FHEAPinsert(H_out_x[bicat_id]→heap, b);
  delete_edge(b);
  insert_edge(extra_in(w), b);
}
```

Este código é usado no bloco 159.

161 ⟨ For each bicategory B, reinsert in $H(B)$ the heaps in H_out_x 161 ⟩ ≡

```
for (i = 0; i < C * C; i++) {
  B = b0 + i;
  B→out[id_x] = FHEAPinsert(B→H→heap, H_out_x[i]);
}
```

Este código é usado no bloco 159.

Como vimos no bloco 145, para concatenar as listas *extra* dos vértices x e y, foi usada a função *concat*, que apresentamos abaixo. Ela recebe apontadores *head1* e *head2* para os nós-cabeça de duas listas circulares duplamente ligadas e concatena estas duas listas de modo que a lista resultante possa ser acessada a partir de *head1*.

162 ⟨ Internal functions of `bicat.c` 111 ⟩ +≡

```
void concat(Arc *head1, Arc *head2)
{
  if (next(head2) ≠ head2) {
    Arc *last1 = prev(head1);
    Arc *first2 = next(head2);
    Arc *last2 = prev(head2);
    next(last1) = first2;
    prev(first2) = last1;
    next(last2) = head1;
    prev(head1) = last2;
  }
}
```

Por fim, os campos da estrutura **Vertex** apontada por x são alterados de modo que ela passe a representar o novo vértice do grafo. Para associar esta estrutura ao vértice resultante da contração de a é usada a função *UFunion*.

163 ⟨ Set x to represent the vertex resulting from the contraction of a 163 ⟩ ≡

```
out_degree(x) += out_degree(y);
weight(x) += weight(y);
UFunion(x, y);
```

Este código é usado no bloco 145.

A última função que apresentaremos dentre as que se encontram prototipadas em `bicat.h` é *freeBicatMem*. Ela recebe o grafo g fornecido como argumento na última chamada que foi feita a *initBicategories* e libera toda a memória que foi alocada, na ocasião, por esta função.

164 ⟨External functions of `bicat.c` 96⟩ +≡

```
void freeBicatMem(Graph *g)
{
  Vertex *v = g→vertices;
  Arc *head = extra_in(v);
  free(head→afields);
  free(head);
  free(b0→H);
  free(b0→in);
  free(b0);
  free(LH);
  free(min_cost_arc);
  free(H_out_x);
  FHEAPfree();
}
```

165 ⟨`bicat.h` 97⟩ +≡

```
int initBicategories(Graph *g, int C);
Vertex *From(Arc *a);
Vertex *Tip(Arc *a);
void decreaseCost(int cat1, int cat2, double delta);
Arc *findMin(int cat1, int cat2);
void changeCategory(Vertex *v, int new_cat);
void contractEdge(Arc *a, int cat);
void freeBicatMem();
```

Implementação do algoritmo de Goemans e Williamson

Nesta seção, apresentaremos a função de `sf.c` que implementa o algoritmo de Goemans e Williamson utilizando a estrutura de dados correspondente às bicategorias. Como vimos, o arquivo-cabeçalho que contém o protótipo das funções utilizadas para manipular bicategorias é `bicat.h` e, desta forma, devemos incluí-lo em `sf.c`.

166 ⟨Header files of sf.c 2⟩ +≡
```
#include "bicat.h"
```

A função de sf.c à qual nos referíamos no parágrafo anterior é *sf_klein*, que apresentamos abaixo. Ela recebe como argumentos um grafo g simples (ou seja, sem laços e sem arestas paralelas) e um vetor *term_sets* de conjuntos de terminais, que se encontram indexados de 0 a g→*num_ts* − 1 no vetor. Ela devolve uma floresta de Steiner do grafo g, caso exista uma tal floresta, e devolve Λ, caso contrário.

167
```
#define INACTIVE 0
#define ACTIVE 1
⟨Steiner forest construction functions 10⟩ +≡
  SteinerForest *sf_klein(Graph *g, TermSet *term_sets)
  {
    ⟨Local variables of sf_klein 168⟩
    ⟨Assign one category to each vertex of g and make some initializations 174⟩
    if (¬initBicategories(g, 2) ∨ ¬(sf = gb_new_graph(g→n))) {
      err = 2; return Λ;
    }
    while (num_actives > 0) {
      Arc *a = Λ, *a_;
      double delta;
      int cat;
      ⟨Select an edge a with the smallest slackness delta 169⟩
      decreaseCost(INACTIVE, ACTIVE, delta);
      decreaseCost(ACTIVE, INACTIVE, delta);
      decreaseCost(ACTIVE, ACTIVE, 2 * delta);
      a_ = inverse(a);
      ⟨Include a in the forest sf 33⟩
      ⟨Set cat to the state of the component in sf that contains a 175⟩
      contractEdge(a, cat);
    }
    ⟨Find and return a subgraph of sf that is a minimal Steiner forest 177⟩
  }
```

Para representar a floresta geradora à qual iremos adicionar uma aresta de g em cada iteração da primeira fase do algoritmo, usaremos uma estrutura do tipo **Graph**. A variável *sf* guardará

um apontador para esta estrutura e a variável *num_actives* guardará o número de componentes ativas presentes na floresta.

168 ⟨Local variables of *sf_klein* 168⟩ ≡

```
Graph *sf;
int num_actives;
```

Veja também blocos 170 e 173.

Este código é usado no bloco 167.

Para selecionar a aresta de g que será incluída em sf em cada iteração, são feitas três chamadas a *findMin*. Após cada chamada, a variável global *min_cost* (declarada em `bicat.h`) guardará o custo reduzido da aresta devolvida pela função, caso a bicategoria fornecida como argumento não seja vazia. Com base nestes valores, é escolhida uma aresta de folga mínima dentre as três candidatas selecionadas por *findMin*. Tal aresta é guardada na variável a e a sua folga, na variável *delta*.

169 ⟨Select an edge a with the smallest slackness *delta* 169⟩ ≡

```
{
  Arc *a1 = findMin(INACTIVE, ACTIVE);
  double d1 = (¬a1) ? DBL_MAX : min_cost;
  Arc *a2 = findMin(ACTIVE, INACTIVE);
  double d2 = (¬a2) ? DBL_MAX : min_cost;
  Arc *a3 = findMin(ACTIVE, ACTIVE);
  double d3 = (¬a3) ? DBL_MAX : min_cost/2;
  if (d1 ≤ d2) { delta = d1; a = a1; }
  else { delta = d2; a = a2; }
  if (d3 < delta) {
    delta = d3; a = a3; dual_cost += delta * num_actives--;
  }
  else dual_cost += delta * num_actives;
  if (¬a) { err = 1; return Λ; }
}
```

Este código é usado no bloco 167.

Como nas implementações dos capítulos anteriores, usaremos uma variável *dual_cost* para guardar o valor corrente da função objetivo do programa dual.

170 ⟨Local variables of *sf_klein* 168⟩ +≡

```
double dual_cost = 0;
```

Aqui, usaremos uma estratégia diferente da que usamos nos capítulos anteriores para decidir se uma dada componente é ou não ativa. Nos capítulos anteriores, a cada componente S da floresta, estava associado um vetor $I(S)$ que, para cada T em $\mathcal{R}$, guardava em $I(S)[T]$ o número de vértices do conjunto de terminais T que se encontram na componente S, isto é, $|S \cap T|$. Com base na informação contida neste vetor, decidíamos se a componente resultante da inclusão de uma nova aresta na floresta era ou não ativa. Só que o tempo gasto na inicialização destes vetores é $\Theta(nr)$, onde $r = |\mathcal{R}|$, e assim é possível que, só para inicializar tais vetores, gastemos mais tempo do que o limite permitido para esta implementação, isto é, $O(n\sqrt{m}\log n)$ (para ver isto, basta, por exemplo, tomar $m = r = n/2$).

Observe que a maior parte deste tempo é gasta para preencher com zeros posições destes vetores, já que, no momento da inicialização, cada componente da floresta é composta por um único vértice. Então, para resolver este problema, uma possível saída é guardar, na estrutura $I(S)$ destinada a armazenar o tamanho das intersecções de S com os conjuntos de terminais, apenas a informação associada a conjuntos de terminais que têm intersecção não-vazia com S. Assim, o tempo gasto com estas inicializações seria $O(n)$. Por outro lado, note que assim, caso implementássemos $I(S)$ através de um vetor (como antes), já não poderíamos mais, de maneira trivial, determinar o tamanho da intersecção de S com um dado conjunto de terminais, pois não haveria uma forma trivial de saber se uma dada posição deste vetor tem seu conteúdo atualizado, isto é, se uma dada posição deste vetor corresponde a um conjunto de terminais que tem intersecção não-vazia com S. Em outras palavras, não mais seria possível implementar $I(S)$ através de um vetor a menos que tivéssemos o auxílio de uma estrutura de dados que informasse eficientemente se uma dada posição do vetor corresponde ou não a um conjunto de terminais que tem intersecção não-vazia com S.

Para contornar este problema, implementamos $I(S)$ através de uma árvore binária balanceada de busca que guarda, para cada conjunto de terminais T que tem intersecção não-vazia com S, um apontador para uma estutura do tipo **struct intersection**. Esta estrutura contém dois campos: um campo *ts_id* que armazena o identificador do conjunto de terminais T correspondente à estrutura e um campo *size* que guarda o tamanho da intersecção de T com S. Os apontadores para estruturas deste tipo são objetos do tipo **Intersection**. A chave de cada objeto X do tipo **Intersection** em uma árvore $I(S)$ é dada por X→*ts_id*. Assim, podemos determinar o tamanho da intersecção de S com um dado conjunto de terminais em tempo $O(\log r)$ e, deste modo, como veremos adiante, há uma forma de garantir que, em tempo $O(n\sqrt{m}\log n)$, possamos determinar se a componente resultante da inclusão de uma nova aresta na floresta é ou não ativa.

```
171 ⟨intersection.h 171⟩ ≡
    struct intersection {
      int ts_id;
      int size;
```

```
};
typedef struct intersection *Intersection;
```

Abaixo, é feita a alocação de todas as estruturas do tipo **struct intersection** que serão utilizadas ao longo de toda a execução de *sf_klein* e também dos nós das árvores que guardarão apontadores para estas estruturas. Todas as estruturas do tipo **struct intersection** ficarão armazenadas no vetor *intersect*. Iremos também alocar espaço para o vetor *ts_disconnect* que guardará, para cada componente S da floresta, o número de conjuntos de terminais T cujos vértices não se encontram conectados pelas arestas da floresta e tais que $T \cap S \neq \emptyset$.

172 ⟨ Initializations 172 ⟩ ≡

```
num_actives = 0;
for (i = 0; i < g→num_ts; i++) num_actives += term_sets[i].num_vertices;
intersect = malloc(num_actives * sizeof(struct intersection));
ts_disconnect = malloc(g→n * sizeof(int));
if (¬intersect ∨ ¬ts_disconnect ∨ ¬BBSTalloc(num_actives + g→num_ts)) {
  err = 2; return Λ;
}
```

Este código é usado no bloco 174.

173 ⟨ Local variables of *sf_klein* 168 ⟩ +≡

```
Intersection intersect;
int *ts_disconnect;
```

Como, a cada vez que incluímos uma aresta na floresta, tal aresta é contraída no grafo g, durante toda a primeira fase, cada componente S da floresta corresponde a um vértice v no grafo g e vice-versa. Assim, acessaremos a posição de *ts_disconnect* associada a uma dada componente S da floresta através do índice, no vetor *g→vertices*, do vértice v que corresponde a S. Dado um vértice v, o índice de v no vetor *g→vertices* pode ser obtido através do uso da macro $ind(v)$ que definimos abaixo. Além disso, usaremos a macro *ts_intersect* para acessar o campo de v que guarda a árvore $I(S)$, onde S é a componente correspondente a v na floresta.

174 #define *ts_intersect* *y.S*

```
#define ind(v) (v − g→vertices)
```

⟨ Assign one category to each vertex of g and make some initializations 174 ⟩ ≡

```
{
  int i;
  Vertex *v;
```

```
    ⟨ Initializations 172 ⟩
    for (v = g→vertices, i = 0; v < g→vertices + g→n; v++) {
      v→ts_intersect = (char *) BBSTinit();
      if (v→termset_id < 0) {
        v→category = INACTIVE;
        ts_disconnect[ind(v)] = 0;
      }
      else {
        Intersection X = intersect + i++, Y;
        if (¬(v→arcs)) { err = 1; return Λ; }
        X→ts_id = v→termset_id, X→size = 1;
        v→ts_intersect = (char *) BBSTinsert((BalBST) v→ts_intersect, X, &Y);
        ts_disconnect[ind(v)] = 1;
        v→category = ACTIVE;
      }
    }
  }
```

Este código é usado no bloco 167.

Para decidir se a componente S formada através da inclusão de uma nova aresta a na floresta é ou não ativa e, ao mesmo tempo, construir a árvore $I(S)$, seguiremos os seguintes passos. Inicialmente, determinaremos a estrutura **Vertex** que irá representar o vértice resultante da contração de a após a chamada a *contractEdge*. Observe que o vértice resultante da contração de a será o vértice correspondente a S no grafo g. No trecho abaixo, ajustamos as coisas de modo que a variável u guarde um apontador para esta estrutura e, desta forma, após a sua construção, $I(S)$ ficará acessível a partir de *u→ts_intersect*. Agora, sejam S_u e S_v as componentes correspondentes aos vértices $From(a)$ e $Tip(a)$ do grafo g antes de inclusão de a na floresta e considere que S_u corresponde ao vértice, dentre $From(a)$ e $Tip(a)$, cuja estrutura **Vertex** é aquela apontada por u. Para construir $I(S)$, partiremos da árvore $I(S_u)$ e tentaremos, através da função *updateIfNotConnected*, inserir em $I(S_u)$ cada objeto X do tipo **Intersection** que se encontra em $I(S_v)$. Note que nem sempre conseguiremos inserir um tal X em $I(S_u)$, pois $I(S_u)$ pode já conter algum objeto que tenha a mesma chave de X, o que significaria que S_u tem intersecção não-vazia com o conjunto de terminais correspondente a X. Neste caso, tudo o que precisamos fazer é incrementar de *X→size* o conteúdo do campo *size* do objeto de $I(S_u)$ que tem a mesma chave de X. Ao mesmo tempo que construímos $I(S)$, decidiremos se S é ou não ativa e, de acordo com esta decisão, preencheremos a variável *cat* com a categoria adequada.

Observe que S_u corresponde ao um vértice de peso máximo dentre $From(a)$ e $Tip(a)$ e, desta

forma, o peso do vértice correspondente a S no grafo (que é a soma dos pesos de $From(a)$ e $Tip(a)$) é pelo menos duas vezes maior que o peso do vértice correspondente a S_v. Assim, a cada vez que um objeto do tipo **Intersection** é inserido em uma árvore, a sua nova árvore corresponde a um vértice cujo peso é pelo menos duas vezes maior que o peso do vértice correspondente à sua antiga árvore. Logo, como o peso de qualquer vértice é limitado por n, o número de inserções realizadas para cada objeto do tipo **Intersection** é $O(\log n)$. Ademais, ocorre no máximo uma inserção mal sucedida para cada um desses objetos, pois, uma vez que tentamos inserir um objeto X do tipo **Intersection** em uma árvore $I(S)$ e não conseguimos, tal objeto perde a sua utilidade após calcularmos o tamanho da intersecção de S com o conjunto de terminais correspondente a X. Desta forma, como o tempo gasto em cada inserção (mal sucedida ou não) é $O(\log r)$ e o número de estruturas do tipo **Intersection** é igual a $t = \sum_{T \in \mathcal{R}} |T|$, concluímos que o tempo gasto ao longo de toda a primeira fase com estas inserções é $O(t \log n \log r)$. Caso $m \geq t/2$, não é difícil concluir que esse tempo é $O(n\sqrt{m} \log n)$, pois $n \geq t$ e $r \leq t/2$. Caso contrário, isto é, caso $m < t/2$, existe algum vértice terminal no qual não incidem arestas de g (assim, a instância dada é inviável) e detectaríamos isto no início da execução de *sf_klein*, em tempo $O(n)$, como vimos no código do bloco anterior. Portanto, ao longo de toda a primeira fase, o tempo gasto para determinar qual categoria atribuir ao vértice resultante da contração de cada nova aresta inserida na floresta é $O(n\sqrt{m} \log n)$.

175 ⟨ Set *cat* to the state of the component in *sf* that contains *a* 175 ⟩ ≡

```
{
  void *args[4];
  Vertex *u = From(a), *v = Tip(a);
  cat = ACTIVE;
  if (weight(u) < weight(v)) exchange(u, v)
  args[0] = term_sets;
  args[1] = u;
  args[2] = &cat;
  args[3] = ts_disconnect + ind(u);
  BBSTtraverse((BalBST)(v→ts_intersect), updateIfNotConnected, args);
  if (cat ≡ INACTIVE) num_actives —;
  BBSTdestroy((BalBST) v→ts_intersect);
}
```

Este código é usado no bloco 167.

A função *updateIfNotConnected* recebe um objeto X do tipo **Intersection** e um vetor *args* preenchido da seguinte forma: *args*[0] guarda o vetor *term_sets*, *args*[1] guarda um vértice do grafo g, *args*[2] guarda um apontador para a variável *cat* e *args*[3] guarda um apontador

para a posição do vetor *ts_disconnect* correspondente ao vértice em *args*[1]. Caso o conjunto de terminais $T \equiv$ *term_sets*[*X*→*id*] tenha seus vértices conectados pelas arestas da floresta, absolutamente nada é feito. Caso contrário, a função atualiza o tamanho da intersecção de T com a componente S da floresta correspondente ao vértice u em *args*[1], tentando inserir X na árvore *u*→*ts_intersect*. Se tal inserção for mal sucedida, o campo *size* do objeto com a mesma chave de X em *u*→*ts_intersect* é incrementado de *X*→*size*. Além disso, se necessário, *updateIfNotConnected* atualiza adequadamente *ts_disconnect*[*id*(*u*)] e o conteúdo da variável *cat* e do campo *connected* de T.

176 ⟨ Auxiliary functions 20 ⟩ +≡

```
void updateIfNotConnected(Intersection X, void *args[])
{
  TermSet *term_sets = args[0], *ts;
  Vertex *u = args[1];
  int *cat = args[2], *disconnected = args[3];
  Intersection Y;
  ts = term_sets + X→ts_id;
  if (ts→connected) return;
  u→ts_intersect = (char *) BBSTinsert((BalBST)(u→ts_intersect), X, &Y);
  if (Y ≠ Λ) {
    Y→size += X→size;
    if (Y→size ≡ ts→num_vertices) {
      ts→connected = 1;
      (*disconnected)—;
      if (¬(*disconnected)) *cat = INACTIVE;
    }
  }
  else {
    (*disconnected)++;
    if ((*disconnected) ≡ 1) *cat = ACTIVE;
  }
}
```

No trecho de código abaixo, determinamos, através da função *edgePrunning* uma floresta de Steiner minimal contida na floresta de Steiner representada pela variável *sf*. Além disso, liberamos toda a memória previamente alocada por *sf_klein*.

177 ⟨ Find and **return** a subgraph of *sf* that is a minimal Steiner forest 177 ⟩ ≡

```
{
```

```
    SteinerForest *min_sf;
    freeBicatMem(g);
    min_sf = edgePruning(sf, g, term_sets);
    min_sf→dual_cost = dual_cost;
    ⟨Free the memory allocated by sf_klein 178⟩
    return min_sf;
  }
```

Este código é usado no bloco 167.

178 ⟨Free the memory allocated by *sf_klein* 178⟩ ≡

```
  {
    gb_recycle(sf);
    free(intersect);
    free(ts_disconnect);
    BBSTfree();
  }
```

Este código é usado no bloco 177.

Capítulo 5

Implementação de Cole, Hariharan, Lewenstein e Porat

Para grafos não muito densos, existem implementações que apresentam um melhor desempenho que a implementação de Goemans e Williamson descrita no capítulo 2. Um exemplo é a implementação proposta por Cole, Hariharan, Lewenstein e Porat [5].

Esta implementação difere da implementação de Goemans e Williamson por tentar garantir que, durante a fase iterativa do algoritmo, cada aresta externa do grafo tenha no máximo um dos extremos em uma componente ativa da floresta. Para isto, cada aresta do grafo é subdividida dinamicamente (*dynamic edge spliting*) através da inclusão de um ou mais vértices no grafo. A idéia aqui é a seguinte: para impedir que uma aresta externa tenha ambos os extremos em componentes ativas da floresta, toda aresta externa é dividida ao meio, com seu custo distribuído igualmente entre as duas arestas resultantes da subdivisão. O novo vértice, inserido na subdivisão, será considerado como uma nova componente inativa da floresta. Denominaremos tais vértices de *s-vértices*. Durante toda execução da fase iterativa do algoritmo, cada aresta do grafo original é subdividida no total $O(k \log n)$ vezes, onde k é uma dada constante fornecida como parâmetro. Após o número máximo de subdivisões de uma aresta original ter sido atingido, ao menos uma (e no máximo duas) das arestas resultantes destas subdivisões será uma aresta externa. Chamaremos uma tal aresta de *aresta terminal*. Note que uma aresta terminal não pode ser subdividida, pois isto resultaria em um número de subdivisões maior que o permitido para a aresta original correspondente. Desta forma, é possível que, em algum momento, existam arestas terminais com ambos os extremos em componentes ativas da floresta. Mesmo que isto ocorra, tais arestas serão tratadas como se apenas um de seus extremos fizesse parte de uma componente ativa. Isso introduz um erro no algoritmo, o qual é responsável por uma degradação de $\frac{1}{n^k}$ no fator de aproximação.

Para cada constante k, a implementação consome no total tempo $O(k(n+m)\log^2 n)$, onde n e m são respectivamente o número de vértices e o número de arestas do grafo, e devolve uma floresta de Steiner de custo menor ou igual a $2(1+\frac{1}{n^k})$ vezes o custo de uma floresta de Steiner ótima.

5.1 Descrição da implementação

Como na implementação de Goemans e Williamson, faremos uso de heaps de arestas para reduzir o número de arestas examinadas em cada iteração do algoritmo, com a diferença de que aqui utilizaremos apenas um único heap por componente. A cada componente S da floresta estará associado um heap de mínimo $heap(S)$ contendo as arestas externas que possuem um dos extremos em S. A chave de uma dada aresta uv em $heap(S)$ é definida como segue:

$$chave(uv) = \begin{cases} c_{uv} - d(u) & \text{, se } u \in S \\ c_{uv} - d(v) & \text{, caso contrário.} \end{cases}$$

Esta definição é motivada pela seguinte propriedade garantida pela implementação: dada uma aresta uv do grafo original, cada nova aresta $u'v'$ gerada a partir de uv, com excessão de uma única aresta, a qual será uma aresta terminal, é tal que ou u' ou v' faz parte de uma componente inativa da floresta e, além disto, tal componente consiste de um único s-vértice. Assim, como $d(w) = 0$ para todo s-vértice w que constitui uma componente com um único vértice, toda aresta externa $u'v'$ que não é terminal é tal que

$$folga(u'v') = \begin{cases} c_{u'v'} - d(u') & \text{, se } u' \text{ está em uma componente ativa} \\ c_{u'v'} - d(v') & \text{, se } v' \text{ está em uma componente ativa} \\ \infty & \text{, caso contrário.} \end{cases}$$

As arestas terminais podem possuir ambos os extremos em componentes ativas ou então podem ser tais que $d(u') > 0$ e $d(v') > 0$ e, nestes casos, a folga de tais arestas não pode ser calculada deste modo. Apesar disto, também consideraremos a folga de uma aresta terminal como sendo dada pela igualdade acima. Logo, em cada iteração, para determinar a aresta externa que será incluída na floresta, basta considerar as arestas que se encontram na primeira posição de cada heap associado a uma componente ativa e, dentre elas, selecionar uma que apresente o valor mínimo para a chave.

Vamos apresentar agora uma descrição não muito detalhada do algoritmo correspondente a esta implementação. Inicialmente, cada aresta do grafo é subdividida ao meio em duas arestas de igual custo através da inclusão de um novo vértice no grafo, sendo que cada vértice incluído será considerado como uma nova componente inativa da floresta (lembre-se que, inicialmente, cada vértice em um conjunto da coleção $\mathcal{R}$ é considerado uma componente ativa e que cada vértice

de Steiner do grafo original é considerado uma componente inativa). Além disso, a partir das arestas geradas através desta operação, são criados os heaps associados a cada componente.

Em cada uma das iterações seguintes e enquanto existirem componentes ativas na floresta, o algoritmo irá executar os passos descritos abaixo:

1. *Escolha da próxima aresta*: inicialmente, é escolhida uma aresta externa que apresente a menor folga dentre todas as outras. Seja uv a aresta escolhida neste passo. Para cada vértice z pertencente a uma componente ativa da floresta, $d(z)$ é incrementado de $folga(uv)$. Além disso, uv é incluída na floresta. Por fim, é realizada a união dos conjuntos de vértices das componentes da floresta correspondentes aos vértices u e v.

2. *Divisão de arestas*: note que, no início desta iteração, no máximo uma das componentes correspondentes aos vértices u e v era inativa. Este passo só é executado pelo algoritmo se uma dessas duas componentes era inativa e consistia de um único s-vértice. Neste caso, vamos supor, sem perda de generalidade, que tal vértice seja v. Como v é um s-vértice, existe apenas uma única aresta incidente em v, além de uv. Seja w o vértice que se encontra na outra extremidade desta aresta. Vamos dividir a aresta vw em pedaços subseqüentes através da adição de novos vértices ao grafo, os quais deverão se encontrar às distâncias $\frac{c_{vw}}{2}, \frac{c_{vw}}{4}, \frac{c_{vw}}{8}, \ldots, \frac{c_{vw}}{2^i}$ de v, onde i é o menor índice tal que uma das seguintes condições é satisfeita:

   - $c_{vw} - \frac{c_{vw}}{2^i} > d(w)$.
   - $\frac{c_{vw}}{2^i} \leq \frac{c_{v'w'}}{n^k}$, onde $v'w'$ é a aresta do grafo original a partir da qual vw foi gerada.

   A segunda condição acima é usada para garantir que cada aresta do grafo original seja dividida não mais que $O(k \log n)$ vezes. Sempre que tal condição for satisfeita e o mesmo não cocrrer com a primeira condição, todas as arestas geradas como descrito acima, com excessão da última aresta (cujo custo é menor ou igual a $\frac{c_{v'w'}}{n^k}$), passarão a fazer parte da floresta. Note que esta última aresta não pode mais ser subdivida, mesmo que, em alguma iteração futura, ela possa ter ambos os extremos em componentes ativas (estas são as tais arestas terminais cuja folga não pode ser calculada pela fórmula dada no início desta seção).

   Agora, suponha que i seja tal que a primeira condição é satisfeita e seja z o vértice adicionado que se encontra a distância $\frac{c_{vw}}{2^i}$ de v. Note que, neste caso, todos os novos vértices adicionados ao grafo, com excessão de z, estarão a uma distância de w menor ou igual a $d(w)$. Deste modo, todos eles passarão a fazer parte da componente que contém w, ao passo que z irá formar uma nova componente inativa da floresta (para a qual será criado um heap contendo inicialmente duas arestas). Além disso, todas as novas arestas que foram criadas neste passo, com excessão daquelas incidentes no vértice z, serão incluídas na floresta. Por

fim, as duas arestas incidentes em z serão inseridas em $heap(S_v)$ e $heap(S_w)$, onde S_v e S_w são, respectivamente, as componentes que contém v e w, ao passo que a aresta vw será removida destes dois heaps.

3. *Fusão de heaps*: sejam S_u e S_v as componentes de u e v respectivamente. Se tanto S_u quanto S_v não têm a propriedade necessária para que o passo anterior seja executado (e, portanto, uv é uma aresta terminal) então $heap(S_u)$ e $heap(S_v)$ são combinados para formar um único heap e, além disso, o estado (com relação à atividade) da nova componente formada pela união de S_u e S_v é determinado.

5.2 Estruturas de dados

Para representar as componentes da floresta utilizaremos uma estrutura semelhante a que vimos na implementação anterior. Na estrutura *ES_ Component* apresentada neste bloco, os campos *active*, *vertices*, *num_ vertices*, *term_ sets_ size*, *disconnected*, *prev* e *next* são como aqueles que se encontram na estrutura *Component* descrita no bloco 20.

O campo *edges* é o heap de arestas da componente. Com o intuito de implementar de maneira mais rápida o passo que chamamos na descrição acima de *Fusão de heaps*, optamos por representar o heap de arestas de cada componente através da estrutura de dados conhecida como *Fibonacci heap*, a qual permite que a união de dois *heaps* seja realizada em tempo $O(1)$. Mais adiante, descreveremos com mais detalhes esta estrutura.

Para que o tempo gasto pela implementação seja $O(k(n+m)\log^2 n)$, como citamos anteriormente, não podemos mais gastar tempo $\Theta(n)$, como na implementação anterior, para escolher a aresta que será incluída na floresta em cada iteração. É necessário que esta escolha seja realizada em tempo $O(\log n)$. Para isto, iremos utilizar um heap cujos elementos serão os heaps de arestas das componentes ativas da floresta. O campo *pos* da estrutura *ES_ Component* guarda a posição do heap de arestas da componente neste heap de heaps, caso a componente seja ativa.

Por fim, vamos explicar a utilidade dos campos *d1* e *d2*. Para evitar que em cada iteração tenhamos que atualizar o valor de $d(v)$ para cada v em uma componente ativa (o que nos levaria a gastar tempo $\Theta(n)$ por iteração, como vimos na implementação anterior), iremos dividir o valor de $d(v)$ em parcelas: uma parcela estará associada a estrutura de cada aresta saindo de v, outra estará associada a estrutura da componente da floresta à qual v pertence e, caso v se encontre em uma componente ativa, há ainda uma outra parcela associada a todas as componentes ativas. A parcela associada a componente a qual v pertence é armazenada no campo *d1* da estrutura dessa componente. O campo *d2* é um apontador para a variável que guarda a parcela correspondente a todas as componentes ativas.

Como na implementação anterior, muitas vezes será preciso acessar a componente à qual um dado vértice pertence. Por este motivo, cada vértice v do grafo guardará, em um de seus *utility fields*, um apontador para a componente da qual faz parte. No que segue, utilizaremos a macro COMPONENT(v) para acessar a componente a qual um dado vértice v pertence e a macro SET_COMPONENT(v, S) para fazer com que S seja a nova componente de v.

Precisaremos associar às arestas do grafo, ou mais precisamente, aos arcos correspondentes a elas, um número de campos superior ao número de *utility fields* existentes na estrutura de um arco. Para contornar este problema, iremos criar para cada arco uma estrutura de dados para armazenar alguns destes campos e utilizar um dos *utility fields* do arco como um apontador para esta estrutura. Antes de apresentar a definição desta estrutura, iremos apresentar, a seguir, cada um dos campos presentes nela.

Vimos a pouco que a cada aresta uv do grafo estará associada uma parcela do valor de $d(u)$ e uma parcela do valor de $d(v)$. Como no SGB cada aresta uv é representada através dos arcos uv e vu, vamos associar tais parcelas às estruturas de dados correspondentes a estes arcos. Para isto, criaremos para cada arco do grafo g um campo d. Assim, dada uma aresta uv, a parcela correspondente a $d(u)$ ficará armazenada no campo d do arco uv e a parcela correspondente a $d(v)$ ficará armazenada no campo d do arco vu.

A divisão das arestas, descrita no passo 2 (bloco 48), será feita apenas de modo simulado. Como em cada iteração, após uma sequência de operações de subdivisão de uma aresta, são geradas exatamente duas arestas externas, com exceção do caso em que uma única aresta terminal é gerada, representaremos cada uma das arestas geradas através de um dos arcos correspondentes a aresta original do grafo. Para isto, cada arco passa a ter um campo *curr_ len*, o qual guardará o custo da aresta representada pelo arco. No caso em que apenas uma única aresta for gerada, a qual será terminal, os dois arcos estarão representando a mesma aresta e o valor do campo *curr_ len* de cada um deles corresponderá ao custo dessa aresta terminal. Para detectar esta situação, criaremos o campo *bad_ piece* que permanecerá com valor 0 enquanto os dois arcos estiverem representando arestas distintas e assumirá valor 1 a partir do momento em que eles estiverem representando a mesma aresta. Vale a pena ressaltar que, como a divisão de arestas será apenas simulada, em nenhum instante serão adicionados novos vértices a estrutura do grafo original.

A estrutura *EdgeFields* apresentada abaixo armazena os campos d e *curr_ len* que descrevemos acima. Além disso, tal estrutura possui ainda um campo *origin*, o qual armazenará o valor contido no campo *from* do arco (isto é necessário, pois iremos utilizar o *utility field a* do arco, no qual o valor do campo *from* está armazenado, para guardar um apontador para a estrutura *EdgeFields* correspondente ao arco). COMENTAR SOBRE A MACRO *node*(a).

179 **#define** *curr_ len*(a) (ARC_FIELDS(a)→*e1*.*I*)

```
#define delta(a) (ARC_FIELDS(a)→e2.I)
#define duplicate(a) (ARC_FIELDS(a)→e4.I)
```

Como mencionamos anteriormente, a chave de uma aresta uv no heap de arestas de uma dada componente S é definida como

$$chave(uv) = \begin{cases} c_{uv} - d(u) & , \text{ se } u \in S \\ c_{uv} - d(v) & , \text{ caso contrário} \end{cases}$$

Como estaremos representando as arestas do grafo através de arcos, teremos de definir a chave de cada aresta $\alpha = uv$ em um heap em termos do arco ou dos arcos (no caso de uma aresta terminal) a ela correpondentes. Assim, dado um arco a iremos definir a chave de a no heap da componente de seu vértice de origem u, denotada por $key(a)$, como sendo dada pelo custo da aresta representada por a menos $d(u)$. É importante ressaltar aqui que cada arco será sempre mantido no heap da componente à qual pertence seu vértice de origem. Deste modo, se $\alpha = uv$ é uma aresta terminal, como α é representada pelos arcos uv e vu, a chave de α no heap da componente de u será dada pela chave $key(uv)$ do arco uv e, do mesmo modo, a chave da aresta α no heap da componente de v será dada por $key(vu)$. Caso α não seja terminal, ela é representada por um único arco e desta forma a chave de α será igual a chave deste arco. Apresentamos abaixo a definição da chave $key(a)$ de um arco a.

180 ⟨Auxiliary functions 20⟩ +≡

```
double edge_piece_key(void *p)
{
  Arc *a = p;
  if (¬a) return -1;
  return curr_len(a) - delta(a);
}
```

181 ⟨Auxiliary functions 20⟩ +≡

```
int PIECEless(void *a, void *b)
{
  return (edge_piece_key(a) < edge_piece_key(b));
}
```

182 ⟨Header files of sf.c 2⟩ +≡

```
#include <limits.h>
```

Como mencionamos anteriormente iremos manter os heaps de arestas das componentes ativas da floresta como elementos de um heap. Definiremos a chave de um **FibHeap** h em um **FibHeap** como sendo a chave de menor valor dentre todas aquelas das arestas que se encontram em h. Ou seja, a chave de h será igual a $key(h)$, já que, como vimos, h é um apontador para uma aresta cujo valor da chave é mínimo em h.

183 ⟨Data structures of sf.c 4⟩ +≡

```
typedef struct labeled_heap *LbHeap;
struct labeled_heap {
  int d;
  FibHeap heap;
};
```

184 ⟨Auxiliary functions 20⟩ +≡

```
double lb_heap_key(void *p)
{
  LbHeap LH = p;
  if (¬LH) return −1;
  if (FHEAPempty(LH→heap)) return DBL_MAX;
  return edge_piece_key(FHEAPfindMin(LH→heap)) − LH→d;
}
```

185 ⟨Auxiliary functions 20⟩ +≡

```
int HEAPless(void *a, void *b)
{
  return (lb_heap_key(a) < lb_heap_key(b));
}
```

186 ⟨Data structures of sf.c 4⟩ +≡

```
typedef struct chlp_component {
  int active;
  int num_vertices;
  LbHeap edges;
  BalBST ts_intersect_size;
  int disconnected;
} CHLPComponent;
```

5.3 Função principal

A função abaixo é responsável por aplicar ao grafo g o método de Cole et al para a construção de uma floresta de Steiner cujo custo não é superior a $2(1+\frac{1}{n^k})$ vezes o custo de uma floresta de Steiner ótima, onde n é igual a $g\to n$. Além de g e k, ela recebe um vetor *term_sets* de conjuntos de terminais de g, os quais se encontram indexados de 0 até $g\to num_ts$. Ao final de sua execução, *sf_chlp* irá devolver um apontador para a estrutura **SteinerForest** que representa a floresta de Steiner construída.

187 ⟨Steiner forest construction functions 10⟩ +≡

```
SteinerForest *sf_chlp(Graph *g, TermSet *term_sets, int k)
{
  ⟨Local variables of sf_chlp 190⟩
  ⟨Multiply by n^k the cost of each edge in g 188⟩
  ⟨For each edge a in g split a in two pieces 191⟩
  ⟨Create the initial spanning forest 192⟩
  ⟨Include an edge piece in the forest while there is some active component 196⟩
  ⟨Restore the original costs of the edges of g 189⟩
  ⟨Discard from sf unnecessary edges and return the resulting Steiner forest 208⟩
}
```

188 ⟨Multiply by n^k the cost of each edge in g 188⟩ ≡

```
{
  int i;
  Vertex *v;
  nk = 1;
  for (i = 1; i ≤ k; i++) nk = nk * g→n;
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    Arc *a;
    for (a = v→arcs; a; a = a→next) a→len *= nk;
  }
}
```

Este código é usado no bloco 187.

189 ⟨Restore the original costs of the edges of g 189⟩ ≡

```
{
  Vertex *v;
```

```
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      Arc *a;
      for (a = v→arcs; a; a = a→next) a→len /= nk;
    }
    dual_cost /= nk;
  }
```

Este código é usado no bloco 187.

190 ⟨Local variables of *sf_chlp* 190⟩ ≡

```
  unsigned long int nk;
```

Veja também blocos 193, 195 e 197.

Este código é usado no bloco 187.

O trecho de código abaixo inicializa os campos correspondentes a cada arco do grafo g. Cada aresta do grafo original é dividida ao meio, sendo que cada metade é representada pelos arcos na estrutura de g que correspondem à aresta. Como estaremos trabalhando apenas com inteiros, uma das metades terá custo $\lfloor \frac{c_a}{2} \rfloor$ e a outra, $\lceil \frac{c_a}{2} \rceil$, onde c_a é o valor que se encontra guardado em *a→len*.

191 ⟨For each edge a in g split a in two pieces 191⟩ ≡

```
  {
    Arc *a;
    Vertex *v;
    for (v = g→vertices; v < g→vertices + g→n; v++)
      for (a = v→arcs; a; a = a→next) {
        curr_len(a) = −1;
        delta(a) = 0;
        duplicate(a) = 0;
      }
    for (v = g→vertices; v < g→vertices + g→n; v++)
      for (a = v→arcs; a; a = a→next)
        if (curr_len(a) ≡ −1) {
          Arc *a_ = a + 1;
          if (a→len % 2) {
            curr_len(a) = (a→len)/2 + 1;
            curr_len(a_) = (a→len)/2;
```

```
        }
        else curr_len(a) = curr_len(a_) = (a→len)/2;
      }
}
```

Este código é usado no bloco 187.

Aqui é criada a floresta geradora inicial *sf* e são inicializadas as estruturas correspondentes às componentes desta floresta (note que não são criadas estruturas do tipo *ES_Component* para representar as componentes correpondentes a s-vértices). Todas estas estuturas serão mantidas em uma lista circular duplamente ligada contendo um nó cabeça. A variável *components* guardará um apontador para este nó. O número de componentes ativas nesta floresta e também nas florestas em cada uma das iterações posteriores será guardado na variável *num_actives*. Por fim, a variável *active_d* guardará uma parcela do valor de $d(v)$ para cada v em uma componente ativa da floresta. Como vimos no bloco 49, o campo *d2* de cada componente será um apontador para esta variável.

192 ⟨ Create the initial spanning forest 192 ⟩ ≡

```
{
  int i, j = 0;
  sf = gb_new_graph(g→n);
  num_actives = active_d = 0;
  for (i = 0; i < g→num_ts; i++) num_actives += term_sets[i].num_vertices;
  intersect = malloc(num_actives * sizeof(struct intersection));
  if (¬BBSTalloc(num_actives + g→num_ts − 1)) {
    err = 2; return Λ;
  }
  components = malloc(g→n * sizeof(CHLPComponent));
  for (i = 0; i < g→n; i++) {
    CHLPComponent *C = components + i;
    Vertex *v = g→vertices + i;
    SET_COMPONENT(v, C);
    C→num_vertices = 1;
    C→ts_intersect_size = BBSTinit();
    C→active = C→disconnected = 0;
    if (v→termset_id ≥ 0) {
      Intersection I = intersect + j++, J;
      I→ts_id = v→termset_id, I→size = 1;
      C→ts_intersect_size = BBSTinsert(C→ts_intersect_size, I, &J);
```

```
        C→active = 1;
        C→disconnected++;
      }
    }
    UFinit(g→vertices, g→n);
    ⟨Create an edge heap for each component 194⟩
  }
```

Este código é usado no bloco 187.

193 ⟨Local variables of *sf_chlp* 190⟩ +≡

```
  Graph *sf;
  CHLPComponent *components;
  Intersection intersect;
  int num_actives;
  int active_d;
```

O trecho de código abaixo é responsável por criar um heap de arestas para cada componente da floresta inicial. Como mencionamos antes, cada um dos heaps correspondentes a componentes ativas da floresta será mantido em um heap de mínimo, o qual será apontado pelo valor guardado na variável $active_heaps$.

194 ⟨Create an edge heap for each component 194⟩ ≡

```
  {
    int i;
    if (¬FHEAPalloc(g→n + 1, g→n + g→m)) {
      err = 2;
      return Λ;
    }
    active_heaps = FHEAPinit(HEAPless);
    h = malloc(g→n * sizeof(struct labeled_heap));
    node = malloc(g→n * sizeof(FHnode));
    for (i = 0; i < g→n; i++) {
      Arc *a;
      CHLPComponent *C = components + i;
      Vertex *v = g→vertices + i;
      h[i].d = 0;
      h[i].heap = FHEAPinit(PIECEless);
```

```
      for (a = v→arcs; a; a = a→next) node(a) = FHEAPinsert(h[i].heap, a);
      C→edges = h + i;
      if (C→active) node[i] = FHEAPinsert(active_heaps, C→edges);
    }
  }
```

Este código é usado no bloco 192.

195 ⟨Local variables of *sf_chlp* 190⟩ +≡

```
  FibHeap active_heaps;
  LbHeap h;
  FHnode *node;
```

Neste bloco, é construída de modo iterativo uma floresta de Steiner do grafo g. Em cada iteração, é selecionada uma aresta, correspondente a uma fração de alguma aresta do grafo original, para ser incluída na floresta geradora corrente. Note que incluímos uma aresta em *sf* apenas no momento em que a última fração terminal de alguma aresta a do grafo original é selecionada, pois somente neste instante é que a passa a fazer parte da floresta. Para detectar o momento em que esta situação ocorre fazemos uso do campo *bad_piece* da aresta selecionada.

196 #define RLEN(*uv*) (*edge_piece_key*(*uv*) − *U→edges→d* − *active_d*)

#define ID(*C*) ((*C*) − *components*)

⟨Include an edge piece in the forest while there is some active component 196⟩ ≡

```
  {
    dual_cost = 0;
    while (num_actives > 0) {
      Arc *uv;
      CHLPComponent *U, *V;
      ⟨Set uv to point to an edge with the smallest slackness 199⟩
      dual_cost += num_actives * RLEN(uv);
      active_d += RLEN(uv);
      FHEAPdelMin(U→edges→heap);      /* delete uv from U→edges */
      FHEAPdelMin(active_heaps);      /* delete U→edges from the active_heaps */
      if (curr_len(uv) ≤ (uv→len)/nk ∧ duplicate(uv)) {
          /* uv is a terminal edge and RLEN(uv) != slackness(uv) */
        Arc *vu = inverse(uv);
        ⟨Include uv in the spanning forest sf 198⟩
        FHEAPdelete(V→edges→heap, node(vu));
```

```
        if (V→active) {
          FHEAPdelete(active_heaps, node[ID(V)]);
          num_actives —;
        }
        ⟨Unite the components U and V 200⟩
      }
      else {
        Arc *wv = inverse(uv);
        ⟨Split the edge wv 205⟩
      }
    }
  }
```

Este código é usado no bloco 187.

197 ⟨Local variables of *sf_chlp* 190⟩ +≡

```
double dual_cost;
```

198 ⟨Include *uv* in the spanning forest *sf* 198⟩ ≡

```
{
  Vertex *u = sf→vertices + (uv→from − g→vertices);
  Vertex *v = sf→vertices + (uv→tip − g→vertices);
  gb_new_edge(u, v, (uv→len)/nk);
  u→arcs→original_arc = uv;
  v→arcs→original_arc = vu;
}
```

Este código é usado no bloco 196.

Em cada iteração, a aresta escolhida para ser incluída na floresta corrente é uma aresta externa que minimiza o valor de *key*( ). Como iremos permitir que o heap de arestas de cada componente possua mais de uma aresta para cada outra componente da floresta, é possível que, em algum momento, o heap de arestas de uma componente contenha arestas que não sejam externas. Deste modo, no trecho de código abaixo, o laço **while** itera até que uma aresta externa seja encontrada, sendo que, em cada iteração, é removida de seu heap uma aresta cujo valor de *key*( ) seja mínimo dentre todas as outras nos heaps de componentes ativas.

199 ⟨Set *uv* to point to an edge with the smallest slackness 199⟩ ≡

```
{
  while (1) {
    Arc *vu;
    LbHeap H_u = FHEAPfindMin(active_heaps);
    if (FHEAPempty(H_u→heap)) exit(1);      /* the problem is unfeasible */
    uv = FHEAPfindMin(H_u→heap);
    U = (CHLPComponent *) COMPONENT(uv→from);
    V = (CHLPComponent *) COMPONENT(uv→tip);
    if (U ≠ V) break;
    FHEAPdelMin(H_u→heap);      /* delete uv from U→edges */
    vu = inverse(uv);
    FHEAPdelete(H_u→heap, node(vu));      /* delete vu from U→edges */
      /* actualize the positision of U→edges in the heap */
    FHEAPdelMin(active_heaps);      /* delete H_u from active_heaps */
    node[ID(U)] = FHEAPinsert(active_heaps, H_u);
  }
}
```

Este código é citado no bloco 205.

Este código é usado no bloco 196.

Sempre que a última fração terminal de uma aresta do grafo original é selecionada para fazer parte da floresta, é necessário realizar a união de duas componentes da floresta. Como na implementação anterior, manteremos sempre a estrutura da componente que possui o maior de número de vértices para representar a componente resultante da união, garantindo assim que os campos de todos os vértices e arcos do grafo sejam atualizados no máximo $O(\log n)$ vezes durante toda a fase iterativa. Se a componente resultante da união for inativa, o campo *d1* da estrutura que a representa é incrementado pelo valor guardado na variável apontada por *d2*, já que, a partir deste momento, a parcela guardada nesta variável não mais será usada no cáculo da chave das arestas no heap da componente, até que ela eventualmente seja unida a uma componente ativa. Caso a componente resultante seja ativa, seu heap de arestas é inserido em *active_heaps*.

200 ⟨Unite the components U and V 200⟩ ≡

```
{
  int active_ = 1;
  if (U→num_vertices < V→num_vertices) {
    CHLPComponent *t = U;
```

```
    U = V;
    V = t;
  }
  ⟨Include each vertex of V in U and set active_ to the new state of U 201⟩
  ⟨Unite the edge heaps of U and V 203⟩
  if (¬(U→active) ∧ active_) U→edges→d −= active_d;
  U→active = active_;
  if (U→active) node[ID(U)] = FHEAPinsert(active_heaps, U→edges);
  else {    /* U and V were active components */
    U→edges→d += active_d;
    num_actives −−;
  }
}
```

Este código é usado no bloco 196.

No trecho de código abaixo, é realizada a união dos conjuntos de vértices das componentes U e V. O modo como isto é feito é semelhante àquele descrito no bloco 43.

201 ⟨Include each vertex of V in U and set *active_* to the new state of U 201⟩ ≡

```
{
  void *args[3];
  UFunion(UFfind(uv→from), UFfind(uv→tip));
  U→num_vertices += V→num_vertices;
  V→num_vertices = −1;
  args[0] = term_sets;
  args[1] = U;
  args[2] = &active_;
  BBSTtraverse(V→ts_intersect_size, insertIfDisconnected, args);
  BBSTdestroy(V→ts_intersect_size);
}
```

Este código é usado no bloco 200.

202 ⟨Auxiliary functions 20⟩ +≡

```
void insertIfDisconnected(Intersection I, void *args[])
{
  TermSet *term_sets = args[0];
  CHLPComponent *C = args[1];
```

```
    int *active_ = args[2];
    Intersection J;
    TermSet *ts;
    ts = term_sets + I→ts_id;
    if (ts→connected) return;
    C→ts_intersect_size = BBSTinsert(C→ts_intersect_size, I, &J);
    if (J ≠ Λ) {
      J→size += I→size;
      if (J→size ≡ ts→num_vertices) {
        ts→connected++;
        C→disconnected--;
        if (¬(C→disconnected)) *active_ = 0;
      }
    }
    else {
      C→disconnected++;
      if (C→disconnected ≡ 1) *active_ = 1;
    }
  }
```

Ao contrário do que fizemos na implementação anterior, aqui não iremos exigir que o heap de arestas de cada componente guarde no máximo uma aresta para cada outra componente. Assim, o heap de arestas da componente resultante da união de U e V é obtido através da união dos heaps das duas componentes, utilizando-se para isto a função *mergeFibHeap*. Antes da união dos heaps, atualizamos o valor do campo d de cada aresta no heap da componente V para que o valor da chave destas arestas no novo heap seja igual ao valor de sua chave no heap de V. Para isto, é utilizada a função *traverseFibHeap* que aplica a função *in_field_d*, apresentada no próximo bloco, a cada aresta no heap de V.

203 ⟨Unite the edge heaps of U and V 203⟩ ≡

```
  {
    int dU = U→active ? U→edges→d + active_d : U→edges→d;
    int dV = V→active ? V→edges→d + active_d : V→edges→d;
    int inc = dV - dU;
    void *args[1];
    args[0] = &inc;
    FHEAPtraverse(V→edges→heap, inc_field_delta, args);
    U→edges→heap = FHEAPjoin(U→edges→heap, V→edges→heap);
```

```
}
```

Este código é usado no bloco 200.

A função *inc_field_d* abaixo incrementa de *inc* o valor guardado no campo *d* da aresta *a*.

```
204 ⟨Auxiliary functions 20⟩ +≡
  void inc_field_delta(void *item, void *args[])
  {
    Arc *a = item;
    int inc = *((int *) args[0]);
    delta(a) += inc;
  }
```

Enquanto a aresta *uv* escolhida no bloco ⟨Set *uv* to point to an edge with the smallest slackness 199⟩ não for a última fração terminal da aresta original, esta poderá ser subdividida mais algumas vezes. É exatamente isto que é feito pelo trecho de código abaixo. A aresta *wv* corresponde a fração da aresta original que ainda não foi selecionada. Se possível, esta fração será subdividida até obtermos duas arestas externas ou uma única aresta terminal. No primeiro caso, *uv* e *wv* representarão as duas arestas externas geradas, as quais possuem um s-vértice virtual *v* como extremo comum; no segundo caso, *uv* e *wv* estarão representando a mesma aresta terminal.

```
205 ⟨Split the edge wv 205⟩ ≡
  {
    CHLPComponent *W;
    int cur_len, new_len, d_w, inc, p;
    W = (CHLPComponent *) COMPONENT(wv→from);
    inc = W→active ? W→edges→d + active_d : W→edges→d;
    d_w = delta(wv) + inc;
    new_len = cur_len = curr_len(wv);
    if (cur_len ≤ (wv→len)/nk)      /* wv is a terminal edge */
      ⟨Set uv and wv to represent the same terminal edge 206⟩
    else {
      do {
        p = new_len % 2;
        new_len = p ? new_len/2 + 1 : new_len/2;
      } while (new_len > (wv→len)/nk ∧ cur_len − new_len ≤ d_w);
      if (cur_len − new_len > d_w) {
        ⟨Set uv and wv to represent two external edges 207⟩
```

```
        FHEAPdecreaseKey(W→edges→heap, node(wv), wv);
        if (W→active) FHEAPdecreaseKey(active_heaps, node[ID(W)], W→edges);
      }
      else ⟨Set uv and wv to represent the same terminal edge 206⟩
    }
    node(uv) = FHEAPinsert(U→edges→heap, uv);
    node[ID(U)] = FHEAPinsert(active_heaps, U→edges);
  }
```

Este código é usado no bloco 196.

Aqui, atribuímos valores aos campos dos arcos *uv* e *vw* de modo que eles representem a mesma aresta terminal gerada no bloco anterior. Desta forma, o valor do campo *curr_len* dos dois arcos deve ser o mesmo e o valor do campo *d* de cada um deles é calculado de modo a fazer com que o valor da chave destes arcos seja igual a folga da aresta terminal.

206 ⟨Set *uv* and *wv* to represent the same terminal edge 206⟩ ≡

```
  {
    curr_len(uv) = curr_len(wv) = new_len;
    delta(wv) = d_w − (cur_len − new_len) − inc;
    delta(uv) = d_w − (cur_len − new_len) − (U→edges→d + active_d);
    duplicate(wv) = duplicate(uv) = 1;
  }
```

Este código é usado no bloco 205.

No trecho de código abaixo, os campos dos arcos *uv* e *wv* são preenchidos de modo que cada um deles represente uma das arestas externas geradas no processo descrito no bloco 92.

207 ⟨Set *uv* and *wv* to represent two external edges 207⟩ ≡

```
  {
    curr_len(uv) = new_len;
    curr_len(wv) = p ? new_len − 1 : new_len;
    delta(uv) = −(U→edges→d + active_d);
    delta(wv) = curr_len(wv) − inc − (cur_len − new_len − d_w);
  }
```

Este código é usado no bloco 205.

208 ⟨ Discard from *sf* unnecessary edges and **return** the resulting Steiner forest 208 ⟩ ≡

```
{
  SteinerForest *min_sf;
  min_sf = edgePruning(sf, g, term_sets);
  min_sf→dual_cost = dual_cost;
  ⟨ Free the memory allocated by sf_chlp 209 ⟩
  return min_sf;
}
```

Este código é usado no bloco 187.

209 ⟨ Free the memory allocated by *sf_chlp* 209 ⟩ ≡

```
{
  gb_recycle(sf);
  free(node);
  free(h);
  free(components);
  free(intersect);
  BBSTfree();
  FHEAPfree();
}
```

Este código é usado no bloco 208.

Capítulo 6

Fase da Limpeza

Após a construção iterativa da floresta de Steiner inicial, tem início a segunda fase do algoritmo, conhecida como fase de *fase da limpeza*. O objetivo desta fase é encontrar uma floresta de Steiner minimal contida na floresta construída na primeira fase. Ao final desta fase, a floresta de Steiner resultante é devolvida como saída do algoritmo.

A estratégia que usaremos aqui se baseia em um algoritmo para encontrar o ancestral comum mais próximo (em inglês, "nearest common ancestor" ou "least common ancestor") de dois nós em uma árvore enraizada. Usaremos este algoritmo para determinar, para cada conjunto de terminais do grafo, o ancestral comum mais próximo dos vértices que fazem parte do conjunto. Por fim, com auxílio dos vértices ancestrais encontrados, determinaremos quais as arestas da floresta são realmente essenciais (essenciais no seguinte sentido: se uma dessas arestas for removida da floresta, a floresta resultante possuirá pelo menos dois vértices de um mesmo conjunto de terminais em componentes distintas).

A função *edgePrunning* apresentada aqui implementa essa estratégia. Ela recebe apontadores *sf_graph* e *g* para estruturas do tipo **Graph**, onde *sf_graph* representa uma floresta de Steiner no grafo *g*, e um vetor *term_sets* de conjuntos de terminais do grafo *g*, cujos elementos se encontram indexados de 0 a *g→num_ts*, e devolve um apontador para uma estrutura **SteinerForest** correspondente a uma floresta de Steiner minimal contida em *sf_graph*.

```
211 ⟨Auxiliary functions 20⟩ +≡
    ⟨More auxiliary functions 212⟩

    SteinerForest *edgePruning(Graph *sf_graph, Graph *g, TermSet *term_sets)
    {
      SteinerForest *sf;

      ⟨Find the lca of each terminal set 213⟩
      ⟨Determine the necessary edges with the aid of the lcas 217⟩
```

}

Aplicaremos o algoritmo para o cálculo dos ancestrais em cada uma das árvores da floresta *sf_graph*. Como para isto é necessário que essas árvores sejam enraizadas, determinaremos, para cada árvore, uma raiz e uma relação de descendência entre seus vértices. A função *initialize_parent*, aqui apresentada, realiza uma busca em largura a partir de um dado vértice r de uma floresta f em um grafo não-orientado g, preenchendo o conteúdo do campo *parent_edge* de cada vértice v na componente de r em f da seguinte forma:

- se $v \equiv r$, o campo *parent_edge* de v guarda um apontador para um arco a tal que $a\to tip \equiv r$ e a não pertence a f;
- se $v \neq r$, o campo *parent_edge* de v guarda um apontador para o arco (v, w) de f, indicando que w é o pai de v na árvore enraizada de raiz r determinada pela busca; tal arco é removido da lista de adjacências de v em f.

Além de r, f e g, a função recebe ainda um vetor *term_sets* de conjuntos de terminais de g, cujos elementos se encontram indexados de 0 a $g\to num_ts$. Ao final de sua execução, a função devolve um apontador para uma lista ligada dos conjuntos de terminais em *term_sets* que se encontram na componente de r em f.

```
212 #define parent_edge  u.A      /* v→parent_edge→tip is the parent of v */
    #define next_item  v.V      /* next vertex in the FIFO queue */
    #define IN_LIST(TS_ID)  (f→vertices + TS_ID)→x.I
    #define TERMSET(V)  ((V→termset_id < 0) ? Λ : term_sets + V→termset_id)
    ⟨More auxiliary functions 212⟩ ≡
      TermSet *initialize_parent(Vertex *r, TermSet *term_sets, Graph *g, Graph *f)
      {
        Arc *a;
        Vertex *head, *tail;
        TermSet x, *l;
        int i;
        l = &x;
        l→next_tset = Λ;
        for (i = 0; i < g→num_ts; i++) IN_LIST(i) = 0;
        a = (Arc *) malloc(sizeof(Arc));
        a→tip = r;
        r→parent_edge = a;
```

```
    r→next_item = Λ;
    head = tail = r;
    while (head) {
      Arc *t = Λ;
      Vertex *v = head, *v_;
      TermSet *ts;
      for (a = v→arcs; a; a = a→next) {
        Vertex *w = a→tip;
        if (w→parent_edge) {      /* w is the parent of v */
          if (t) t→next = a→next;
          else v→arcs = a→next;
          continue;
        }
        w→parent_edge = (v < w) ? (a + 1) : (a − 1);
        w→next_item = Λ;
        tail→next_item = w;
        tail = tail→next_item;
        t = a;
      }
      v_ = g→vertices + (v − f→vertices);
      ts = TERMSET(v_);
      if (ts ∧ ¬IN_LIST(v_→termset_id)) {
        ts→next_tset = l→next_tset;
        l→next_tset = ts;
        IN_LIST(v_→termset_id) = 1;
      }
      head = head→next_item;
    }
    return l→next_tset;
  }
```

Veja também bloco 218.

Este código é usado no bloco 211.

Aqui, inicialmente é feito o pré-processamento das árvores em *sf_graph* para o cálculo dos vértices ancestrais. Para cada árvore T da floresta, aplicamos a função *initialize_parent* apresentada no bloco anterior, tornando T uma árvore enraizada. Feito isso, podemos, então, utilizar a função *lca_preprocessing* que implementa a fase de pré-processamento do algoritmo para o cálculo do

ancestral commum mais próximo. Esta função pré-processa uma dada árvore enraizada de tal forma que as consultas pelo ancestral comum mais próximo de dois vértices quaisquer na árvore podem ser respondidas de maneira rápida (em tempo constante). Por fim, obtemos o ancestral comum mais próximo de cada conjunto de terminais que se encontra em T. Após a execução do trecho de código abaixo, para cada vértice v no grafo g, $v\to head_id$ conterá o identificador do conjunto de terminais que se encontra na primeira posição da lista ligada de conjuntos de terminais dos quais v é ancestral comum mais próximo; caso não existam conjuntos de terminais que tenham v como ancestral comum mais próximo, $v\to head_id$ será igual a -1.

```
213 #define head_id  w.I
⟨Find the lca of each terminal set 213⟩ ≡
  {
    Vertex *v;
    ⟨Construct the list of terminals of each terminal set 214⟩
    for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
      Vertex *u;
      v→parent_edge = Λ;
      u = g→vertices + (v − sf_graph→vertices);
      u→head_id = −1;
    }
    for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
      TermSet *ts_list;
      if (v→parent_edge) continue;
      ts_list = initialize_parent(v, term_sets, g, sf_graph);
      if (¬ts_list) continue;
      lca_preprocessing(v);
      ⟨find the lca of each terminal set in the list ts_list 215⟩
    }
  }
```

Este código é usado no bloco 211.

```
214 ⟨Construct the list of terminals of each terminal set 214⟩ ≡
  {
    Vertex *v;
    TermSet *ts;
    for (ts = term_sets; ts < term_sets + g→num_ts; ts++) ts→vertices = Λ;
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      ts = TERMSET(v);
```

```
      if (¬ts) continue;
      v→next_terminal = ts→vertices;
      ts→vertices = v;
    }
  }
```

Este código é usado no bloco 213.

O trecho de código abaixo é responsável pelo cálculo do ancestral comum mais próximo de cada conjunto de terminais que se encontra na lista *ts_list*. Para isto, ele conta com o auxílio da função *lca* que devolve o ancestral comum mais próximo de dois vértices em uma árvore pré-processada pela função *lca_preprocessing*. Ao final da execução deste trecho, TS_LIST(w) associará a cada vértice w do grafo g um apontador para uma lista ligada dos conjuntos de terminais dos quais w é ancestral comum mais próximo. Se não existirem conjuntos de terminais que tenham w como ancestral comum mais próximo, TS_LIST(w) será igual a Λ.

```
215 #define TS_LIST(V)  ((V→head_id < 0) ? Λ : (term_sets + V→head_id))
⟨find the lca of each terminal set in the list ts_list 215⟩ ≡
  {
    TermSet *ts = ts_list;
    while (ts) {
      TermSet *t = ts→next_tset;
      Vertex *v, *w, *w_;
      w = ts→vertices;
      w_ = sf_graph→vertices + (w − g→vertices);
      for (v = w→next_terminal; v; v = v→next_terminal) {
        Vertex *v_ = sf_graph→vertices + (v − g→vertices);
        w_ = lca(w_, v_);
      }
      w = g→vertices + (w_ − sf_graph→vertices);
      ts→next_tset = TS_LIST(w);
      w→head_id = ts→id;
      ts = t;
    }
  }
```

Este código é usado no bloco 213.

Para que possamos utilizar as funções *lca_preprocessing* e *lca* como acima, precisamos incluir o arquivo cabeçalho `lca.h`, no qual elas se encontram prototipadas.

216 ⟨Header files of `sf.c` 2⟩ +≡
```
#include "lca.h"
```

Tendo identificado o ancestral comum mais próximo de cada conjunto de terminais, tudo o que precisamos fazer agora é selecionar as arestas da floresta no caminho entre cada vértice do conjunto e o ancestral encontrado. Note que a floresta *sf* construída com as arestas selecionadas desta forma é uma floresta de Steiner minimal, pois:

- dados dois vértices u e v de um mesmo conjunto de terminais, sempre existe um caminho entre u e v que contém somente arestas em *sf*;
- se removermos alguma das arestas em *sf* obteremos uma floresta que possui pelo menos dois vértices de um mesmo conjunto de terminais em componentes distintas.

Devido ao modo como selecionaremos as arestas que farão parte da nova floresta, será necessário calcular, para cada árvore, uma lista contendo os vértices ancestrais em pré-ordem e processá-los na ordem em que eles aparecem nesta lista. Tal lista é calculada abaixo pela função *lca_preorderlist*, a qual será apresentada mais adiante.

217
```
#define visit  y.I
#define next_lca  x.V
#define marked  b.I
```
⟨Determine the necessary edges with the aid of the lcas 217⟩ ≡
```
{
  Arc *t, x;
  Vertex l, *v, *w;
  sf = (SteinerForest *) malloc(sizeof(SteinerForest));
  sf→edges = t = &x;
  sf→cost = 0;
  for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
    v→visit = 0;
    v→parent_edge→marked = 0;
  }
  w = &l;
  w→next_lca = Λ;
  for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
```

```
      if (v→parent_edge→tip ≠ v) continue;
      lca_preorderlist(v, g, sf_graph, &w);
      for (w = l.next_lca; w; w = w→next_lca) {
        TermSet *ts;
        for (ts = TS_LIST(w); ts; ts = ts→next_tset) {
          Vertex *z;
          for (z = ts→vertices; z; z = z→next_terminal)
            ⟨select the edges in the path from z to w in sf_graph 219⟩
        }
      }
      w = &l;
      w→next_lca = Λ;
    }
    t→next_edge = Λ;
    sf→edges = sf→edges→next_edge;
    return sf;
  }
```

Este código é usado no bloco 211.

Apresentamos aqui a função *lca_preorderlist*, utilizada no bloco anterior. Ela recebe os seguintes argumentos:

- *graph*: um grafo;
- *v*: um dos vértices de *sf_graph*;
- *sf_graph*: apontador para uma estrutura do tipo **Graph** que representa uma floresta geradora do grafo *graph* da qual uma árvore de raiz *v* faz parte; além disso, para cada vértice *w* de *graph*, se *w* encontra-se na posição *i* do vetor *graph→vertices* então o vértice correspondente a *w* em *sf_graph* encontra-se na posição *i* do vetor *sf_graph→vertices*;
- *t*: apontador para uma estrutura do tipo **Vertex**.

Ao final de sua execução, a função terá construído uma lista ligada de vértices *u* de *graph*, onde *u→ts_list* ≠ Λ, que apresenta a seguinte propriedade: quando visitamos os vértices na ordem em que eles aparecem na lista, os correspondentes vértices em *sf_graph* são visitados em um percurso em pré-ordem na árvore de raiz *v*. Além disso, o campo *next_lca* do vértice que era apontado por *t* no momento em que a função foi evocada conterá um apontador para o primeiro

vértice de tal lista (aqui, é importante ressaltar que, ao final da execução da função, t não estará mais apontando para o mesmo vértice para o qual apontava inicialmente, antes da execução da função).

218 ⟨More auxiliary functions 212⟩ +≡

```
void lca_preorderlist(Vertex *v, Graph *graph, Graph *sf_graph, Vertex **t)
{
  Arc *a;
  Vertex *v_;
  v→visit = 1;
  v_ = graph→vertices + (v − sf_graph→vertices);
  if (v_→head_id ≥ 0) {
    v_→next_lca = Λ;
    (*t)→next_lca = v_;
    *t = v_;
  }
  for (a = v→arcs; a; a = a→next) {
    if (a→tip→visit) continue;
    lca_preorderlist(a→tip, graph, sf_graph, t);
  }
}
```

Para selecionar as arestas no caminho em *sf_graph* entre o vértice terminal z e o ancestral w do conjunto de terminais de z, basta percorrermos as arestas da árvore que os contém, seguindo os apontadores *parent_edge* dos vértices a partir de z até que encontremos ou o vértice w ou uma aresta já selecionada. Cada aresta visitada neste percurso é marcada, atribuindo-se valor 1 para o campo *marked* da aresta, e incluída na floresta *sf*.

219 ⟨select the edges in the path from z to w in *sf_graph* 219⟩ ≡

```
{
  Vertex *w_ = sf_graph→vertices + (w − g→vertices);
  Vertex *z_ = sf_graph→vertices + (z − g→vertices);
  while (z_ ≠ w_ ∧ ¬(z_→parent_edge→marked)) {
    t→next_edge = z_→parent_edge→original_arc;
    sf→cost += z_→parent_edge→len;
    z_→parent_edge→marked = 1;
    t = t→next_edge;
    z_ = z_→parent_edge→tip;
  }
```

```
}
```

Este código é usado no bloco 217.

Apêndice A

Cálculo do ancestral comum mais próximo

Seja T uma árvore enraizada e seja x um vértice qualquer de T. Dizemos que todo vértice que ocorre no caminho que vai de x até a raiz de T é um *ancestral* de x em T. Agora, sejam u e v dois vértices de T e P_u e P_v os caminhos que vão, respectivamente, de u até a raiz e de v até a raiz. O *ancestral comum mais próximo* (em inglês, *nearest common ancestor* ou *least common ancestor*) de u e v em T é o primeiro vértice que ocorre tanto em P_u quanto em P_v.

Considere então o seguinte

> **Problema**: Dados uma árvore enraizada T e dois vértices u e v de T, encontrar o ancestral comum mais próximo de u e v em T.

Apresentaremos aqui uma implementação de um algoritmo que preprocessa T em tempo linear no seu número de vértices e, após isso, responde a consultas pelo ancestral comum mais próximo de quaisquer dois vértices de T em tempo constante. Este algoritmo, concebido por Schieber e Vishkin, foi originalmente apresentado em [25]. Uma descrição mais detalhada pode ser encontrada em [24].

A.1 Dois casos especiais: cadeias e árvores binárias completas

O algoritmo para o cálculo do ancestral comum mais próximo em árvores enraizadas arbitrárias se baseia em algoritmos para dois casos especiais: cadeias e árvores binárias completas.

Quando a árvore T do problema é uma cadeia (ou seja, cada vértice interno de T contém apenas um único filho) é facil determinar o ancestral comum mais próximo de u e v: basta

determinar qual dos dois vértices se encontra mais próximo da raiz e este será o vértice ancestral procurado. Em outras palavras, o ancestral comum mais próximo de u e v será aquele, dentre u e v, que se encontra no nível de menor valor (isto é, no nível mais alto) da árvore (lembrando que o nível de um vértice na árvore é dado por sua distância até a raiz).

Quando T é uma árvore binária completa (ou seja, uma árvore onde cada vértice interno possui exatamente dois filhos e todas as folhas se encontram em um mesmo nível) precisamos inicialmente calcular a ordem em que cada vértice de T é visitado em um percurso de T em inordem. Isto é necessário, pois a representação binária do número $in(v) \geq 1$ associado a ordem em que cada vértice v é visitado neste percurso apresenta propriedades bastante úteis. Assumindo que os bits da representação binária de $in(v)$ estão indexados de 0 (bit menos significativo) a l (bit mais significativo), onde l é o nível em que se encontram as folhas de T, e que i é o índice do bit 1 mais à direita (ou seja, menos significativo), essas propriedades são as seguintes:

- $l - i$ indica o nível em que o vértice v se encontra em T;
- se v_e é o filho esquerdo do vértice v, então a representação binária de $in(v_e)$ consiste dos $l - i$ primeiros bits de $in(v)$ (ou seja, os $l - i$ bits mais significativos), seguidos de um bit 0, um bit 1 e $i - 1$ bits 0;
- se v_d é o filho direito do vértice v, então a representação binária de $in(v_d)$ consiste dos $l - i$ primeiros bits de $in(v)$, seguidos de dois bits 1 e $i - 1$ bits 0.

A partir dessas propriedades, pode-se deduzir facilmente que um vértice v é ancestral de um vértice w se, e somente se, os números $in(v)$ e $in(w)$ satisfazem as seguintes condições, onde i é o índice do bit 1 mais à direita na representação binária de $in(v)$:

1. os $l - i$ primeiros bits de $in(w)$ correspondem exatamente aos $l - i$ primeiros bits de $in(v)$;
2. o índice do bit 1 mais à direita de $in(w)$ é menor ou igual a i.

Desta forma, dados dois vértices x e y, um vértice z é ancestral comum de x e y se os $l - i$ primeiros bits de $in(z)$ (onde i é, como antes, o índice do bit 1 mais à direita na representação binária de $in(z)$) são iguais aos $l - i$ primeiros bits de $in(x)$ e de $in(y)$ e, além disso, o índice do bit 1 mais à direita em ambos, $in(x)$ e $in(y)$, é menor ou igual a i. Note que, assim, o ancestral comum mais próximo de x e y é aquele para o qual i é mínimo (pois este é o ancestral que se encontra num nível mais baixo — ou seja, de maior valor — que os demais).

O algoritmo abaixo se baseia nestas idéias para encontrar o ancestral comum mais próximo de dois vértices x e y em uma árvore binária completa T.

Algoritmo *ACMP-BC(T, x, y)*

1 $i_x \leftarrow$ índice do bit 1 mais à direita de in(x)
2 $i_y \leftarrow$ índice do bit 1 mais à direita de in(y)
3 $i_{max} \leftarrow max\{i_x, i_y\}$
4 se os $l - i_{max}$ primeiros bits de $in(x)$ são iguais aos $l - i_{max}$ primeiros bits de $in(y)$
5 então
6 se $i_x = i_{max}$
7 então devolva x
8 senão devolva y
9 senão
10 $i \leftarrow$ índice do bit mais à esquerda em que $in(x)$ e $in(y)$ diferem
11 seja b o número formado pelos $l - i$ primeiros bits de $in(x)$ seguidos por um bit 1 e i bits 0
12 seja z o vértice de T tal que $in(z) = b$
13 devolva z

A.2 O algoritmo

Como já foi mencionado, o algoritmo para árvores enraizadas arbitrárias se baseia nos algoritmos descritos no bloco anterior para dois casos especiais do problema. A idéia do algoritmo é particionar os vértices da árvore T do problema em cadeias disjuntas e mapear, através de uma função injetiva Ψ, este conjunto de cadeias no conjunto de vértices de uma árvore binária completa B. Tal mapeamento tem a seguinte propriedade (*preservação da ascendência*): se v é um ancestral de u em T então $\Psi(C_v)$ é um ancestral de $\Psi(C_u)$ em B, onde C_v e C_u são as cadeias em T às quais pertencem, respectivamente, v e u.

Assim, dados dois vértices x e y de T, se x e y pertencem a uma mesma cadeia, então é fácil determinar o ancestral comum mais próximo de x e y, como vimos no bloco anterior (caso especial em que a árvore do problema é uma cadeia). Caso contrário, sejam C_x e C_y as cadeias às quais x e y pertencem. Para descobrir o ancestral comum mais próximo de x e y em T, o algoritmo inicialmente encontra o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B (através do algoritmo *ACMP-BC* visto no bloco anterior). Então, usando algumas informações adicionais, como veremos adiante, o algoritmo identifica a cadeia em T que contém o ancestral comum mais próximo de x e y e, por fim, determina e devolve tal vértice.

Pré-processamento

O objetivo da fase de pré-processamento do algoritmo é fazer com que seja possível realizar em tempo constante consultas pelo ancestral comum mais próximo de quaisquer dois vértices da árvore T. Para isto, a árvore T é pré-processada de modo que, ao final, três rótulos estejam associados a cada vértice v de T: $level(v)$, $inlabel(v)$ e $ascendant(v)$. Além disso, é construída uma tabela auxiliar, a qual chamaremos de *head*.

Para cada v, o rótulo $level(v)$ indica o nível em que o vértice v se encontra em T. O nível de um vértice na árvore indica sua distância até a raiz e, desta forma, podemos defini-lo recursivamente da seguinte forma, onde $p(v)$ indica o pai do vértice v em T:

$$level(v) = \begin{cases} 0 & \text{se } v \text{ é a raiz de } T \\ level(p(v)) + 1 & \text{caso contrário.} \end{cases}$$

Note que, assim, $level(v)$ pode ser facilmente calculado para todo v, percorrendo-se a árvore T (uma única vez) em pré-ordem.

O rótulo *inlabel* é usado para particionar o conjunto de vértices de T em cadeias disjuntas. Assim, dois vértices u e v pertencem a uma mesma cadeia em T se e somente se $inlabel(u) = inlabel(v)$. Para calcular o rótulo *inlabel*, é necessário primeiramente calcular, para cada vértice v em T, os rótulos $pre(v)$ e $size(v)$ que indicam, respectivamente, a ordem em que v é visitado em um percurso em pré-ordem de T e o número de vértices na sub-árvore de T com raiz em v. O percurso de T em pré-ordem apresenta a seguinte propriedade: para cada vértice w na sub-árvore de T com raiz em v, $pre(v) \leq pre(w) \leq pre(v) + size(v) - 1$. Chamaremos de *intervalo de* v o intervalo fechado $[pre(v), pre(v) + size(v) - 1]$. Desta forma, vamos definir o rótulo $inlabel(v)$, para cada v, como sendo igual ao inteiro no intervalo de v cujo o índice do bit 1 mais à direita em sua representação binária é máximo.

Por que o rótulo *inlabel* definido desta maneira particiona os vértices de T em cadeias disjuntas? Primeiramente, note que, dados dois vértices u e v de T, se $inlabel(u) = inlabel(v) = p$, então o vértice w tal que $pre(w) = p$ se encontra tanto na sub-árvore de T com raiz em u quanto na sub-árvore de T com raiz em v. Desta forma, u e v são ancestrais de w e, portanto, u e v se encontram em uma mesma cadeia em T. Além disso, se dois vértices u e v se encontram em uma cadeia cujos extremos superior e inferior, s e t, respectivamente, são tais que $inlabel(s) = inlabel(t)$, então $inlabel(u) = inlabel(v)$. Vamos mostrar que isso é verdade. Sejam i_u, i_s e i_t os índices do bit 1 mais à direita em $inlabel(u)$, em $inlabel(s)$ e em $inlabel(t)$, respectivamente. Como t está na sub-árvore de T com raiz em u, temos que o intervalo de t está contido no intervalo de u e, desta forma, $i_u \geq i_t$. Seguindo o mesmo raciocínio, temos que $i_s \geq i_u$. Assim, se $inlabel(u) \neq inlabel(t)$ então $i_s \geq i_u > i_t$ e, portanto, $i_s > i_t$ o que é absurdo já que $inlabel(s) = inlabel(t)$. Logo, $inlabel(u) = inlabel(t)$. Do mesmo modo, concluímos que

$inlabel(v) = inlabel(t)$ e, portanto, $inlabel(u) = inlabel(v)$.

O rótulo *inlabel* também é usado para definir uma função injetiva Ψ que mapeia as cadeias em que T foi particionada nos vértices de uma árvore binária completa B com $2^{\lfloor \log |V_T| \rfloor + 1} - 1$ vértices (note que esta é a menor árvore que contém pelo menos $|V_T|$ vértices). Seja C_i a cadeia cujo valor do rótulo *inlabel* de cada um dos vértices é igual a i. Vamos definir $\Psi(C_i)$ da seguinte forma:

$$\Psi(C_i) = v \in V_B \text{ tal que } in(v) = i.$$

A função Ψ definida desta maneira satisfaz a *propriedade de preservação da ascendência*, ou seja, se u e v são dois vértices de T tais que v é ancestral de u, então $\Psi(C_v)$ é ancestral de $\Psi(C_u)$, onde C_v e C_u são as cadeias às quais pertencem v e u, respectivamente. Para enxergar isto, note primeiramente que, como v é ancestral de u, sabemos que u se encontra na sub-árvore de T com raiz em v e, deste modo, o intervalo de u está contido no intervalo de v. Assim, $inlabel(u)$ pertence ao intervalo de v e, portanto, $i_v \geq i_u$, onde i_u e i_v são os índices do bit 1 mais à direita em $inlabel(u)$ e em $inlabel(v)$, respectivamente. Além disso, pelo mesmo motivo, os $l - i_v$ bits mais à esquerda de $inlabel(u)$ são iguais aos $l - i_v$ bits mais à esquerda de $inlabel(v)$. Logo, $inlabel(v) = in(\Psi(C_v))$ e $inlabel(u) = in(\Psi(C_u))$ satisfazem as condições 1 e 2 descritas no bloco 2 e, portanto, $\Psi(C_v)$ é um ancestral de $\Psi(C_u)$ em B.

A recíproca da propriedade acima nem sempre é verdadeira, ou seja, é possível que, embora v não seja ancestral de u em T, $\Psi(C_v)$ seja um ancestral de $\Psi(C_u)$ em B. Esse fato motiva a utilização do rótulo *ascendant*. Para cada vértice u, $ascendant(u)$ guarda todos os ancestrais de $\Psi(C_u)$ em B que estão associados a ancestrais de u em T. Mais precisamente, $ascendant(u)$ guarda todos os números $inlabel(v)$ para os quais v é um ancestral de u em T (note que $inlabel(v) = in(\Psi(C_v))$ e que, pela propriedade da preservação da ascendência, $\Psi(C_v)$ é ancestral de $\Psi(C_u)$ em B). Podemos guardar todos estes números em apenas um único inteiro no intervalo $[0, 2^{\lfloor \log |V_T| \rfloor + 1} - 1]$. Isso segue do fato de que, para cada w ancestral de $\Psi(C_u)$ em B, podemos determinar $in(w)$ a partir do índice i do bit 1 mais à direita em sua representação binária, já que os $l - i$ primeiros bits de $in(w)$ são iguais aos $l - i$ primeiros bits de $in(\Psi(C_u)) = inlabel(u)$.

Desta forma, vamos definir $ascendant(u)$ como sendo o inteiro cuja representação binária é formada pelos bits A_j, $0 \leq j \leq \lfloor \log |V_T| \rfloor$, que satisfazem a seguinte propriedade:

$$A_j = 1 \Leftrightarrow \begin{array}{l} j \text{ é o índice do bit 1 mais à direita em } inlabel(v) \\ \text{para algum vértice } v \text{ ancestral de } u \text{ em } T. \end{array}$$

Note que o único ancestral da raiz de T é ela própria e que o valor do rótulo *inlabel* da raiz é 2^l. Além disso, se u é um vértice de T diferente da raiz, o único ancestral de u que não é ancestral

de $p(u)$ é o próprio u. Assim, podemos definir recursivamente o rótulo $ascendant(u)$, para cada vértice u em T, da seguinte forma, onde i é o índice do bit 1 mais à direita na representação binária de $inlabel(u)$:

$$ascendant(u) = \begin{cases} 2^l & \text{se } u \text{ é a raiz de } T \\ ascendant(p(u)) & \text{se } inlabel(u) = inlabel(p(u)) \\ ascendant(p(u)) + 2^i & \text{caso contrário.} \end{cases}$$

Note que a definição acima sugere uma estratégia simples para o cálculo do rótulo *ascendant* de cada vértice de T: percorrer T (uma única vez) em pré-ordem, testando para cada vértice u visitado as condições da definição acima e atribuindo o valor correspondente a $ascendant(u)$.

Para concluir a nossa descrição sobre a fase de pré-processamento do algoritmo, vamos falar agora sobre a tabela auxiliar *head*. Esta tabela nada mais é do que um vetor indexado pelo valor dos rótulos *inlabel* dos vértices da árvore T (note que o tamanho deste vetor é linear em $|V_T|$). Para cada i, $head[i]$ guarda o vértice da árvore T que se encontra mais próximo da raiz e cujo valor do rótulo *inlabel* é igual a i. Note que, desta forma,

$$head[i] = \begin{cases} \text{raiz de } T & \text{se } i = 2^l \\ v \text{ tal que } i = inlabel(v) \neq inlabel(p(v)) & \text{caso contrário.} \end{cases}$$

Assim, podemos facilmente preencher a tabela *head* em um percurso em pré-ordem da árvore T.

Consultas

Após realizar o pré-processamento da árvore T, como foi descrito no bloco anterior, é possível responder a consultas pelo ancestral comum mais próximo de quaisquer dois vértices de T em tempo constante. Dados dois vértices x e y de T, vamos descrever a partir de agora o algoritmo para encontrar o ancestral comum mais próximo de x e y.

Se $inlabel(x) = inlabel(y)$, então x e y se encontram em uma mesma cadeia em T e, desta forma, o ancestral comum mais próximo de x e y será x, se $level(x) \leq level(y)$, e será y, caso contrário. Agora, se $inlabel(x) \neq inlabel(y)$, então x e y se encontram em cadeias distintas em T e, neste caso, iremos encontrar o ancestral mais próximo de x e y em quatro passos:

1. Sejam C_x e C_y as cadeias em T em que se encontram, respectivamente, x e y. Este primeiro passo consiste em encontrar o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B. Mais precisamente, estamos interessados em encontrar a ordem em que tal vértice é visitado em um percurso em inordem de B. Note que, para isto, basta utilizar uma

versão levemente modificada do algoritmo *ACMP-BC* (visto no bloco 2) com B, $\Psi(C_x)$ e $\Psi(C_y)$ como argumentos, já que B é uma árvore binária completa (na implementação deste passo, a árvore B será dada implicitamente e o algoritmo receberá como argumentos $inlabel(x) = in(\Psi(C_x))$ e $inlabel(y) = in(\Psi(C_y))$).

2. Seja v o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B. A partir de $in(v)$, vamos descobrir a cadeia em T em que se encontra z, o ancestral comum mais próximo de x e y. Mais especificamente, vamos descobrir o valor de $inlabel(z)$ o qual sabemos que é igual a $in(\Psi(C_z))$, onde C_z é a cadeia em T em que se encontra z. Note que, como nem todo ancestral comum de $\Psi(C_x)$ e $\Psi(C_y)$ em B está associado à cadeia de um ancestral comum de x e y em T, v não é necessariamente igual a $\Psi(C_z)$. Entretanto, não é difícil de concluir que $\Psi(C_z)$ é o ancestral mais próximo de v que está associado à cadeia de um ancestral comum de x e y em T (tal fato segue da propriedade da preservação da ascendência).

   Então, sejam i o índice do bit 1 mais à direita em $in(v)$, j o índice do bit 1 mais à direta em $inlabel(z)$ e $ascendant_{[l-i]}(x)$ e $ascendant_{[l-i]}(y)$ os números formados pelos $l-i$ primeiros bits de $ascendant(x)$ e $ascendant(y)$, respectivamente. Como $\Psi(C_z)$ é ancestral de v e $in(\Psi(C_z)) = inlabel(z)$, temos que $j \geq i$ e, desta forma, j é o índice de algum dos bits 1 de $ascendant_{[l-i]}(x) \wedge ascendant_{[l-i]}(y)$. Como o nível de $\Psi(C_z)$ é o mais baixo possível, sabemos que j é o índice do bit 1 mais à direita em $ascendant_{[l-i]}(x) \wedge ascendant_{[l-i]}(y)$. Por fim, note que uma vez que conseguimos determinar j, determinamos também o valor de $inlabel(z)$. Isso segue do fato que $inlabel(z)$ é o número formado pelos $l-j$ primeiros bits de $inlabel(x)$ seguidos por um bit 1 e j bits 0, já que $\Psi(C_z)$ é ancestral de $\Psi(C_x)$ em B, $in(\Psi(C_z)) = inlabel(z)$ e $in(\Psi(C_x)) = inlabel(x)$.

3. Agora que já sabemos em que cadeia de T o vértice z se encontra, vamos descobrir, neste passo, os vértices $\bar{x}$ e $\bar{y}$ que são, respectivamente, o ancestral mais próximo de x em C_z e o ancestral mais próximo de y em C_z. Aqui, descreveremos apenas como determinar $\bar{x}$. O procedimento para determinar $\bar{y}$ é análogo.

   Note que, se $inlabel(x) = inlabel(z)$, então x se encontra em C_z e, desta forma, $\bar{x} = x$. Caso contrário, seja w o ancestral de x cujo pai é $\bar{x}$. Como $\bar{x}$ é o ancestral mais próximo de x que se encontra em C_z, temos que $inlabel(w) \neq inlabel(\bar{x})$ e, desta forma, w é o vértice que se encontra no nível mais alto em sua cadeia. Logo, $w = head[inlabel(w)]$. Assim, tudo que precisamos fazer para determinar w e, desta forma, determinar $\bar{x}$, é descobrir o valor de $inlabel(w)$. Observe que, para isto, basta que calculemos i_w, o índice do bit 1 mais à direita em $inlabel(w)$, já que $inlabel(w)$ é o número formado pelos $l - i_w$ primeiros bits de $inlabel(x)$ seguidos por um bit 1 e i_w bits 0. É exatamente isso que vamos fazer agora. Seja $i_{\bar{x}}$ o índice do bit 1 mais à direita em $inlabel(\bar{x})$. Como w está na sub-árvore de T com raiz em $\bar{x}$ sabemos que o intervalo de w está contido no intervalo de $\bar{x}$ e, portanto, $i_{\bar{x}} > i_w$. Do mesmo modo, podemos concluir que para todo w' tal que w é ancestral de w', $i_w \geq i_{w'}$,

onde $i_{w'}$ é o índice do bit 1 mais à direita em $inlabel(w')$. Logo, i_w é o índice do bit 1 mais à esquerda em $ascendant_{i_{\bar{x}}}(x)$, número formado pelos $i_{\bar{x}}$ últimos bits de $ascendant(x)$.

4. Por fim, note que podemos determinar z facilmente a partir de $\bar{x}$ e $\bar{y}$. Para isto, basta observar que z é aquele, dentre estes dois vértices, que se encontra em um nível mais alto em T. Esta observação segue do fato que, como $\bar{x}$ e $\bar{y}$ se encontram em uma mesma cadeia em T, um dos dois (o que está no nível mais alto) é ancestral do outro e, portanto, é ancestral comum de x e y. Assim, pela maneira como definimos $\bar{x}$ e $\bar{y}$, temos que se $level(\bar{x}) > level(\bar{y})$ então $z = \bar{x}$ e, caso contrário, $z = \bar{y}$.

A.3 Implementação em CWEB

A partir de agora, apresentaremos uma implementação do algoritmo que acabamos de descrever. A estrutura geral da implementação é dada abaixo.

221 ⟨ Header files of lca.c 222 ⟩
⟨ Global variables of lca.c 226 ⟩
⟨ Internal functions of lca.c 227 ⟩
⟨ External functions of lca.c 225 ⟩

Para representar vértices, arestas e a árvore de entrada, faremos uso das estruturas de dados definidas no SGB. Para isto, vamos incluir o arquivo cabeçalho `gb_graph.h` que contém as definições dessas estruturas.

222 ⟨ Header files of lca.c 222 ⟩ ≡
```
#include "gb_graph.h"
```

Veja também blocos 224, 231 e 237.

Este código é usado no bloco 221.

O algoritmo para o cálculo do ancestral comum mais próximo será implementado através de duas funções: *lca_preprocessing*, responsável pelo pré-processamento da árvore de entrada, e *lca*, a qual responde à consultas sobre o ancestral comum de quaisquer dois vértices da árvore pré-processada. Aqui, vamos gerar um arquivo `lca.h` que servirá de interface para programas que porventura queiram utilizar estas duas funções.

223 ⟨ `lca.h` 223 ⟩ ≡
```
void lca_preprocessing(Vertex *r);
Vertex *lca(Vertex *u, Vertex *v);
```

224 ⟨ Header files of lca.c 222 ⟩ +≡
```
#include "lca.h"
```

A função *lca_preprocessing* recebe um ponteiro r para a raiz de uma árvore e preprocessa esta árvore, associando a cada vértice u três rótulos inteiros: u→*level*, u→*inlabel* e u→*ascendant*, os quais são armazenados nos *utility fields* v, x e y do vértice, respectivamente. Além disso, também é criada uma tabela auxiliar *head*. Os rótulos e a tabela são como aqueles descritos no algoritmo visto no bloco 4 e serão utilizados posteriormente pela função *lca*, sempre que esta for chamada.

A árvore deve estar representada através de uma estrutura **Graph**. Além disso, cada um de seus vértices, com exceção da raiz r, deve possuir um apontador para o arco leva ao seu pai na árvore e tal apontador deve estar armazenado no *utility field* u do vértice. No caso da raiz, o *utility field* u deve armazenar um ponteiro para um arco que conduz a própria raiz. Aqui, renomearemos este *utility field*, através de um `define`, para *parent_edge*. Assim, r→*parent_edge*→*tip* $\equiv r$ e, se $v \neq r$, v→*parent_edge*→*tip* será um ponteiro para o pai de v na árvore. Por fim, para todo vértice v, o arco v→*parent_edge* não deve fazer parte da lista de adjacência de v na árvore.

225 **#define** *parent_edge* $u.A$
#define *level* $v.I$
#define *inlabel* $x.I$
#define *ascendant* $y.I$
⟨ External functions of lca.c 225 ⟩ ≡
```
void lca_preprocessing(Vertex *r)
{
  ⟨ Compute the level and the inlabel number of each vertex 228 ⟩
  ⟨ Compute the ascendant number of each vertex and construct the head table 230 ⟩
}
```
Veja também bloco 233.

Este código é usado no bloco 221.

226 ⟨ Global variables of lca.c 226 ⟩ ≡
```
Vertex **head;
int max_inlabel = 0;
```
Este código é usado no bloco 221.

Nos próximos blocos, freqüentemente precisaremos descobrir o índice do bit 1 mais significativo de um número. A função abaixo recebe um inteiro x e calcula o índice do bit 1 mais à esquerda (o bit mais significativo) na representação binária de x.

227 ⟨Internal functions of lca.c 227⟩ ≡

```
static int msb(unsigned long x)
{
  register unsigned long b;
  register float f;
  if (x ≡ 0) return 0;
  f = (float) x;
  __asm__("mov␣␣␣%1,␣%0\n\t"
      "sar␣␣␣$0x17,␣%0\n\t"
      "sub␣␣␣$0x7F,␣%0\n\t"
      : "=r"(b)
      :"r"(f)
      ) ;
  return ((int) b);
}
```

Veja também blocos 229, 232 e 235.

Este código é usado no bloco 221.

Este bloco é responsável pelo preenchimento dos campos *level* e *inlabel* de cada vértice da árvore. Na verdade, todo este trabalho é feito pela função *label1* , a qual será descrita no próximo bloco.

228 ⟨Compute the level and the inlabel number of each vertex 228⟩ ≡

```
{
  r→level = −1;
  label1 (r);
}
```

Este código é usado no bloco 225.

A função *label1* recebe como argumento um vértice v de uma árvore e calcula, para todo vértice w na sub-árvore com raiz em v, o valor de w→*level* e w→*inlabel*.

229 #define *preorder* *y.I*

#define *size* *z.I*

⟨Internal functions of lca.c 227⟩ +≡

```
void label1 (Vertex *v)
{
```

```
  Arc *a;
  Vertex *w;
  int i, x;
  static long p;
  w = v→parent_edge→tip;
  v→level = w→level + 1;
  if (w ≡ v) p = 1;
  v→preorder = p++;
  v→size = 0;
  for (a = v→arcs; a; a = a→next) {
    label1(a→tip);
    v→size += a→tip→size;
  }
  (v→size)++;
  x = (v→preorder − 1) ⊕ (v→preorder + v→size − 1);
  i = msb(x);
  v→inlabel = ((v→preorder + v→size − 1) ≫ i) ≪ i;
  if (v→inlabel > max_inlabel) max_inlabel = v→inlabel;
}
```

Aqui é feito o cálculo do valor do rótulo *ascendant* de cada um dos vértices na árvore e, além disso, a tabela *head*[0..*max_inlabel* + 1] é criada e preenchida. A maior parte desse serviço é feita pela função *label2*, a qual descreveremos logo adiante.

230 ⟨ Compute the ascendant number of each vertex and construct the *head* table 230 ⟩ ≡

```
{
  Arc *a;
  head = (Vertex **) malloc((max_inlabel + 1) * sizeof(Vertex *));
  r→ascendant = r→inlabel;
  head[r→inlabel] = r;
  for (a = r→arcs; a; a = a→next) label2(a→tip);
}
```

Este código é usado no bloco 225.

231 ⟨ Header files of lca.c 222 ⟩ +≡

```
#include <stdlib.h>
```

A função *label2* recebe um vértice v de uma árvore, da qual v não é raiz, e calcula o valor do rótulo *ancendant* de todo vértice que se encontra na sub-árvore com raiz em v. Além disso, esta função também preenche a tabela *head* como especificado no algoritmo descrito no bloco 4.

232 ⟨Internal functions of lca.c 227⟩ +≡

```
void label2(Vertex *v)
{
  Arc *a;
  Vertex *w = v→parent_edge→tip;
  if (v→inlabel ≡ w→inlabel) v→ascendant = w→ascendant;
  else {
    int x = v→inlabel;
    v→ascendant = w→ascendant + (x − (x & (x − 1)));
    head[x] = v;
  }
  for (a = v→arcs; a; a = a→next) label2(a→tip);
}
```

A função *lca* recebe como argumentos dois vértices x e y de uma árvore e devolve o ancestral como mais próximo de x e y. Para tanto, tal árvore deve ter sido pré-processada anteriormente através de uma chamada à função *lca_preprocessing*.

233 ⟨External functions of lca.c 225⟩ +≡

```
Vertex *lca(Vertex *x, Vertex *y)
{
  int lca_inlabel, j;
  if (x→inlabel ≡ y→inlabel) return ((x→level < y→level) ? x : y);
  ⟨Find lca_inlabel, the inlabel number of the lca of x and y 234⟩
  ⟨Find and return the lca of x and y 236⟩
}
```

O trecho de código abaixo calcula o valor do rótulo *inlabel* do ancestral comum mais próximo de x e y. Para isto, ele conta com o auxílio da função *cbt_lca*, a qual será apresentada no próximo bloco. Note que este trecho de código nada mais é do que uma implementação dos passos 1 e 2 do algoritmo visto no bloco 5.

234 ⟨Find *lca_inlabel*, the inlabel number of the lca of x and y 234⟩ ≡

```
{
  int b = cbt_lca(x→inlabel, y→inlabel);
```

```
  int i = msb(b - (b & (b - 1)));
  int common = (x→ascendant) & (y→ascendant);
  int common_i = (common >> i) << i;
  j = msb(common_i - (common_i & (common_i - 1)));
  lca_inlabel = (((x→inlabel) >> (j + 1)) << (j + 1)) + (1 << j);
}
```

Este código é usado no bloco 233.

A função *cbt_lca* recebe dois inteiros x e y tais que $x \equiv in(u)$, $y \equiv in(v)$ e u e v são vértices de uma árvore binária completa. A partir de x e y, esta função calcula e devolve $in(z)$, onde z é o ancestral comum mais próximo de u e v na árvore. Note que esta função é uma implementação do algoritmo *ACMP-BC* visto no bloco 2.

235 ⟨ Internal functions of lca.c 227 ⟩ +≡

```
int cbt_lca(int x, int y)
{
  int i, j, k;
  i = msb(x - (x & (x - 1)));
  j = msb(y - (y & (y - 1)));
  k = ((i > j) ? i : j) + 1;
  if ((x >> k) ≠ (y >> k)) {
    k = msb(x ⊕ y);
    return ((x >> (k + 1)) << (k + 1)) + (1 << k);
  }
  else return ((i ≥ j) ? x : y);
}
```

Por fim, aqui é calculado o ancestral comum mais próximo de x e y na árvore. O trecho de código abaixo é uma implementação dos passos 3 e 4 do algoritmo descrito no bloco 5.

236 ⟨ Find and return the lca of x and y 236 ⟩ ≡

```
{
  Vertex *x_, *y_, *w;
  int l = msb(INT_MAX);      /* index of the leftmost bit of an int variable */
  int i_w, w_inlabel;
  if (x→inlabel ≡ lca_inlabel) x_ = x;
  else {
```

```
    i_w = msb(((unsigned)(x→ascendant ≪ (l + 1 − j))) ≫ (l + 1 − j));
    w_inlabel = ((x→inlabel ≫ (i_w + 1)) ≪ (i_w + 1)) + (1 ≪ i_w);
    w = head[w_inlabel];
    x_ = w→parent_edge→tip;
  }
  if (y→inlabel ≡ lca_inlabel) y_ = y;
  else {
    i_w = msb(((unsigned)(y→ascendant ≪ (l − j + 1))) ≫ (l − j + 1));
    w_inlabel = ((y→inlabel ≫ (i_w + 1)) ≪ (i_w + 1)) + (1 ≪ i_w);
    w = head[w_inlabel];
    y_ = w→parent_edge→tip;
  }
  return ((x_→level ≤ y_→level) ? x_ : y_);
}
```

Este código é usado no bloco 233.

237 ⟨Header files of lca.c 222⟩ +≡

```
#include <limits.h>
```

Apêndice B

Função *main* do programa

Vamos apresentar aqui a função *main* do programa que será responsável por executar, sobre a entrada, as funções correspondentes a cada implementação. Ao final de sua execução, o programa exibirá, para cada implementação, as arestas da floresta de Steiner construída, juntamente com um valor que limita superiormente a razão entre o custo desta floresta e o custo de uma solução ótima do problema.

```
238 ⟨The main program 238⟩ ≡
    int main(int argc, char *argv[])
    {
      ⟨Local variables of function main 241⟩
      ⟨Process the comand line 239⟩
      ⟨Initialize the field from of each arc 247⟩
      ⟨Allocate the extra fields of the arcs 248⟩
      ⟨Find a Steiner forest 254⟩
      ⟨Show the results 258⟩
      return 0;
    }
```

Este código é usado no bloco 1.

Basicamente, há duas formas de executar o programa. Uma delas é digitando o nome do programa ("`sf`") seguido pelo nome do arquivo que contém a especificação do grafo de entrada e dos conjuntos de terminais neste grafo (a qual deve seguir um formato que descreveremos mais a diante). Eis um exemplo de como executar o programa desta forma:

```
sf nome_do_arquivo_de_entrada
```

A segunda forma é utilizando a opção "-sgb". Esta opção permite que o usuário especifique o grafo de entrada através de um arquivo contendo a descrição de um grafo do SGB, isto é, através de um arquivo gerado pela função *save_graph* do módulo GB_SAVE do SGB. Neste caso, para executar o programa o usuário deve digitar algo semelhante ao seguinte, na linha de comando:

```
sf -sgb nome_do_arquivo_sgb nome_do_arquivo_de_entrada
```

onde `nome_do_arquivo_de_entrada` é o nome de um arquivo que contém a especificação dos conjuntos de terminais do grafo descrito no arquivo `nome_do_arquivo_sgb`.

É possível ainda especificar a implementação que o usuário deseja aplicar sobre a entrada. Para aplicar apenas a implementação sugerida por Goemans e Wiliamson, basta acrescentar "-gw" após o nome do arquivo ou dos arquivos que compõem a entrada. Da mesma forma, se o usuário deseja aplicar apenas a implementação proposta por Cole et al., ele deve acrescentar "-chlp" a linha de comando (exatamente como no caso anterior) e, em seguida, fornecer o valor do parâmetro k tomado por esta implementação (um inteiro maior ou igual a 1). Caso nenhuma dessas opções tenham sido especificadas, o programa irá aplicar cada uma das implementações sobre a entrada.

Como já mencionamos, ao final de sua execução, o programa irá exibir, para cada implementação aplicada sobre a entrada, uma lista das arestas que fazem parte da floresta de Steiner construída, juntamente com o custo desta floresta e um limitante superior para a razão entre este custo e o custo de uma floresta de Steiner ótima. Se o usuário já tiver conhecimento prévio do custo de uma floresta de Steiner ótima, é possível informa-lo ao programa através da opção "-opt". Para isto, deve-se digitar "-opt cost", onde "cost" é o custo de uma floresta de Steiner ótima, após o nome do arquivo ou dos arquivos que fazem parte da entrada do programa.

239 ⟨Process the comand line 239⟩ ≡

```
{
  FILE *file;
  int next_arg;
  if (argc < 2) ⟨Print a help message and exit 240⟩
  if (¬strcmp(argv[1], "-sgb")) {
    if (argc < 4) ⟨Print a help message and exit 240⟩
    if (¬(graph = restore_graph(argv[2]))) {
      fprintf(stderr, "Cannot␣restore␣graph␣in␣file␣%s.\n", argv[2]);
      exit(1);
    }
    if (¬(file = fopen(argv[3], "r"))) {
      fprintf(stderr, "Cannot␣open␣file␣%s.\n", argv[3]);
```

```
      exit(1);
    }
    ⟨Read the terminal sets from the file; exit if unsuccessful 252⟩
    next_arg = 4;
  }
  else {
    if (¬(file = fopen(argv[1], "r"))) {
      fprintf(stderr, "Cannot␣open␣file␣%s.\n", argv[1]);
      exit(1);
    }
    ⟨Try to create a graph from the file; exit if unsuccessful 244⟩
    next_arg = 2;
  }
  fclose(file);
  opt = −1;
  show_edges = 0;
  imp = ALL;
  random = 0;
  while (argc − 1 ≥ next_arg) {
    if (¬strcmp(argv[next_arg], "-gw")) imp = GW;
    else {
      if (¬strcmp(argv[next_arg], "-chlp") ∧ argc − 1 ≥ next_arg + 1) {
        imp = CHLP;
        k = atoi(argv[++next_arg]);
        if (k < 1) {
          fprintf(stderr, "\n␣Invalid␣value␣for␣parameter\n");
          exit(1);
        }
      }
      else {
        if (¬strcmp(argv[next_arg], "-klein")) imp = KLEIN;
        else {
          if (¬strcmp(argv[next_arg], "-ggw")) {
            imp = GGW;
            if (argc − 1 ≥ next_arg + 1) {
              if (¬strcmp(argv[next_arg + 1], "-r")) {
                random = 1;
                next_arg++;
```

```
                }
              }
            }
            else {
              if (¬strcmp(argv[next_arg], "-opt") ∧ argc − 1 ≥ next_arg + 1)
                opt = atoi(argv[++next_arg]);
              else {
                if (¬strcmp(argv[next_arg], "-e")) show_edges = 1;
                else {
                  if (¬strcmp(argv[next_arg], "-t")) test = 1;
                  else ⟨ Print a help message and exit 240 ⟩
                }
              }
            }
          }
        }
      }
      next_arg++;
    }
  }
```

Este código é usado no bloco 238.

Se o usuário digitar algo errado na linha de comando, será exibida uma mensagem explicando brevemente a forma correta de se executar o programa.

240 ⟨ Print a help message and exit 240 ⟩ ≡

```
{
  fprintf(stderr, "\n␣sf␣[-sgb␣filename_gb]␣filename""␣[-gw␣|␣-chlp␣k␣|␣-\
      klein␣|␣-ggw␣|␣-opt␣cost]\n\n");
  exit(1);
}
```

Este código é usado no bloco 239.

Se não ocorrerem erros, após o processamento da linha de comando do programa, o valor guardado na variável *imp* irá indicar qual ou quais das implementações deverão ser aplicadas sobre a entrada. Caso este valor seja igual a CHLP, então a variável k estará guardando o valor correspondente ao parâmetro tomado pela implementação de Cole et al. Vale a pena mencionar

também que a variável *opt* será usada para guardar o custo de uma floresta de Steiner ótima, sempre que este custo for informado pelo usuário.

241 ⟨ Local variables of function *main* 241 ⟩ ≡

```
enum {
  ALL, GW, CHLP, KLEIN, GGW
} imp;
int k = 1, show_edges, test = 0, random;
long opt;
```

Veja também blocos 243, 249, 253 e 256.

Este código é usado no bloco 238.

Para representar vértices, arestas e grafos faremos uso das estruturas de dados definidas no SGB. Desta forma, devemos incluir o arquivo cabeçalho `gb_graph.h`, o qual contém as definições dessas estruturas e o protótipo das funções utilizadas para cria-las. Além disso, vamos incluir também o arquivo `gb_save.h`, o qual contém o protótipo da função *restore_graph*, usada para restaurar o grafo descrito no arquivo de entrada quando a opção `-sgb` estiver presente na linha de comando.

242 ⟨ Header files of `sf.c` 2 ⟩ +≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_save.h"
```

Vamos declarar aqui mais uma das variáveis locais da função *main* que foi usada acima: *graph*. Se não ocorrer nenhum erro durante o processamento da linha de comando ou leitura dos arquivos de entrada, esta variável guardará um apontador para a estrutura que representa o grafo da entrada.

243 ⟨ Local variables of function *main* 241 ⟩ +≡

```
Graph *graph;
```

Para fornecer a especificação do grafo de entrada através de um arquivo, o usuário deve seguir as seguintes convenções. Na primeira linha, deve-se informar, o número n de vértices do grafo e, nas n linhas seguintes, digitar o rótulo ou nome (com no máximo 30 caracteres) de cada um dos vértices, um vértice por linha. Na linha seguinte, vem o número m de arestas do grafo; nas próximas m linhas do arquivo, o usuário deve digitar, para cada aresta uv, as posições (entre 0 e

$n-1$) em que os vértices de rótulo u e v aparecem na lista dos vértices no arquivo e, em seguida, o custo da aresta. Cada uma dessas m linhas deve corresponder a definição de uma, e somente uma, aresta.

Os conjuntos de terminais do grafo devem ser especificados no arquivo de entrada do seguinte modo. Inicialmente, é necessário fornecer o número t de conjuntos de terminais no grafo. As próximas t linhas do arquivo devem informar os vértices que fazem parte de cada um dos conjuntos de terminais, sendo que cada linha deve corresponder a um, e somente um, conjunto. Se a opção `-sgb` estiver presente na linha de comando, cada vértice deverá ser especificado através de seu índice no vetor de vértices do grafo descrito no arquivo de entrada; caso contrário, os vértices deverão ser informados através de números inteiros (a partir de 0) correspondendo a ordem em que foram listados no arquivo de entrada.

Eis um exemplo de um arquivo de entrada satisfazendo as convenções acima:

```
5
A
B
C
D
E
7
0 1 15
2 3 12
3 4 18
2 0 24
0 4 30
1 3 28
4 1 32
2
0 3
1 4
```

O grafo especificado neste exemplo contém cinco vértices, sete arestas e dois conjuntos de terminais. Os vértices são *A, B, C, D* e *E*; as arestas são *(A, B)* (custo 15), *(C, D)* (custo 12), *(D, E)* (custo 18), *(C, A)* (custo 24), *(A, E)* (custo 30), *(D, B)* (custo 28) e *(E, B)* (custo 32); por fim, os conjuntos de terminais são *{A, D}* e *{B, E}*.

O trecho de código abaixo é responsável pela criação do grafo especificado no arquivo de entrada.

244 ⟨ Try to create a graph from the file; exit if unsuccessful 244 ⟩ ≡

```
{
  long num_vertices;
  if (¬fscanf(file, "␣%ld", &num_vertices)) {
    fprintf(stderr, "Cannot␣read␣the␣number␣of␣vertices.\n");
    exit(1);
  }
  if (¬(graph = gb_new_graph(num_vertices))) {
    fprintf(stderr, "Cannot␣create␣a␣graph␣with␣%ld␣vertices.\n", num_vertices);
    exit(1);
  }
  ⟨ Read the vertices from the file; exit if unsuccessful 245 ⟩
  ⟨ Read the edges from the file; exit if unsuccessful 246 ⟩
  ⟨ Read the terminal sets from the file; exit if unsuccessful 252 ⟩
}
```

Este código é usado no bloco 239.

245 ⟨ Read the vertices from the file; exit if unsuccessful 245 ⟩ ≡

```
{
  Vertex *v;
  char name[30];
  for (v = graph→vertices; v < graph→vertices + graph→n; v++) {
    if (¬fscanf(file, "␣%s", name)) {
      fprintf(stderr, "There␣are␣less␣than␣%ld␣vertices␣in␣file␣%s.\n", graph→n,
          (argc ≡ 4) ? argv[3] : argv[2]);
      exit(1);
    }
    v→name = gb_save_string(name);
  }
}
```

Este código é usado no bloco 244.

246 ⟨ Read the edges from the file; exit if unsuccessful 246 ⟩ ≡

```
{
  long num_edges, i;
```

```
    if (¬fscanf(file, "␣%ld", &num_edges)) {
      fprintf(stderr, "Cannot␣read␣the␣number␣of␣edges.\n");
      exit(1);
    }
    for (i = 0; i < num_edges; i++) {
      long p, q, l;
      if (¬fscanf(file, "␣%ld␣%ld␣%ld", &p, &q, &l)) {
        fprintf(stderr, "Error␣in␣definition␣of␣edge␣%ld.\n", i + 1);
        exit(1);
      }
      gb_new_edge(graph→vertices + p, graph→vertices + q, l);
    }
  }
```

Este código é usado no bloco 244.

Nas implementações que iremos descrever, será necessário ter acesso, de maneira rápida, aos extremos de uma dada aresta. É importante ressaltar que, no SGB, cada aresta uv é representada através dos arcos uv e vu. Como a estrutura que representa um arco contém apenas um apontador para o vértice de destino (ou seja, o vértice para o qual o arco aponta), será preciso pré-processar o grafo, inicializando, para cada arco, o campo *from* que contém um apontador para o vértice de origem do arco.

247 ⟨Initialize the field *from* of each arc 247⟩ ≡

```
  {
    Vertex *v;
    for (v = graph→vertices; v < graph→vertices + graph→n; v++) {
      Arc *a;
      for (a = v→arcs; a; a = a→next) a→from = v;
    }
  }
```

Este código é usado no bloco 238.

248 ⟨Allocate the extra fields of the arcs 248⟩ ≡

```
  {
    if (¬(af = malloc(graph→m * sizeof(ExtraFields)))) {
      printf("\n␣Overflow\n");
      exit(1);
```

```
    }
    ⟨ Initialize the extra fields of the arcs 250 ⟩
  }
```

Este código é usado no bloco 238.

249 ⟨ Local variables of function *main* 241 ⟩ +≡

```
  ExtraFields *af;
```

250 ⟨ Initialize the extra fields of the arcs 250 ⟩ ≡

```
  {
    int i = 0;
    Vertex *v;
    for (v = graph→vertices; v < graph→vertices + graph→n; v++) {
      Arc *a;
      for (a = v→arcs; a; a = a→next) a→afields = (char *)(af + i++);
    }
  }
```

Este código é usado nos blocos 248 e 254.

251 ⟨ Allocate the extra fields of the vertices 251 ⟩ ≡

```
  {
    int i = 0;
    Vertex *v;
    ExtraFields *ef;
    if (¬(ef = malloc(graph→n * sizeof(ExtraFields)))) {
      printf("\n␣Overflow\n");
      exit(1);
    }
    for (v = graph→vertices; v < graph→vertices + graph→n; v++)
      v→vfields = (char *)(ef + i++);
  }
```

Este código é usado no bloco 254.

A leitura da especificação dos conjuntos de terminais e a criação das estruturas que os representam são feitas no trecho de código abaixo. Tais estruturas serão guardadas no vetor *term_sets* e o número de elementos deste vetor será armazenado no campo *num_ts* da estrutura que representa o grafo da entrada. Faremos com que o campo *id* de cada conjunto de terminais corresponda ao seu índice no vetor *term_sets*. Além disso, para cada vértice v do grafo, v→*termset_id* guardará o identificador do conjunto de terminais do qual o vértice faz parte (o valor deste campo será igual a -1 para os vértices que não pertencem aos conjuntos de terminais do grafo).

252 ⟨Read the terminal sets from the file; exit if unsuccessful 252⟩ ≡

```
{
  int i;
  Vertex *v;
  if (¬fscanf(file, "␣%ld", &(graph→num_ts))) {
    fprintf(stderr, "Cannot␣read␣the␣number␣of␣terminal␣sets.\n");
    exit(1);
  }
  getc(file);
  for (v = graph→vertices; v < graph→vertices + graph→n; v++) v→termset_id = −1;
  term_sets = (TermSet *) malloc((graph→num_ts) * sizeof(TermSet));
  for (i = 0; i < graph→num_ts; i++) {
    TermSet *ts = term_sets + i;
    char c;
    int n = 0;
    ts→id = i;
    do c = getc(file); while (c ≡ '␣');
    while (c ≠ EOF ∧ c ≠ '\n') {
      int v;
      ungetc(c, file);
      if (fscanf(file, "%d", &v) ∧ v ≥ 0 ∧ v < graph→n) {
        (graph→vertices + v)→termset_id = ts→id;
        n++;
      }
      else {
        fprintf(stderr, "Invalid␣value␣for␣the␣element␣%d␣of␣terminal␣set␣%d.",
            n + 1, i + 1);
        exit(1);
      }
      do c = getc(file); while (c ≡ '␣');
    }
```

```
      if (n ≡ 0) {
        fprintf(stderr, "There␣are␣less␣than␣%ld␣terminal␣sets␣in␣file␣%s.\n",
            graph→num_ts, (argc ≡ 4) ? argv[3] : argv[1]);
        exit(1);
      }
      ts→num_vertices = n;
      ts→connected = ((n ≡ 1) ? 1 : 0);
    }
  }
```

Este código é usado nos blocos 239 e 244.

253 ⟨ Local variables of function *main* 241 ⟩ +≡

```
  TermSet *term_sets;
```

254 ⟨ Find a Steiner forest 254 ⟩ ≡

```
  {
    int i;
    long nk = graph→n;
    Vertex *v;
    TermSet *ts;
    if (graph→id) fprintf(stdout, "\nName:␣%s", graph→id);
    fprintf(stdout, "\nnumber␣of␣vertices:␣%ld", graph→n);
    fprintf(stdout, "\nnumber␣of␣edges:␣%ld", (graph→m)/2);
    fprintf(stdout, "\nnumber␣of␣terminal␣sets:␣%ld\n\n", graph→num_ts);
    switch (imp) {
    case GW: fprintf(stdout, "*****␣Goemans␣and␣Williamson␣*****\n");
      start = clock();
      sf = sf_gw(graph, term_sets);
      end = clock();
      break;
    case CHLP:
      fprintf(stdout, "*****␣Cole,␣Hariharan,␣Lewenstein␣and␣Porat␣*****\n");
      for (i = 1; i < k; i++) {
        if (LONG_MAX/nk < graph→n) {
          fprintf(stderr, "\nToo␣high␣value␣for␣parameter.␣");
          break;
        }
```

```
    nk = nk * graph→n;
  }
  for (v = graph→vertices; v < graph→vertices + graph→n; v++) {
    Arc *a;

    for (a = v→arcs; a; a = a→next)
      if (LONG_MAX/nk < a→len) {
        if (LONG_MAX/nk ≥ graph→n)
          fprintf(stderr, "\nTooUhighUvalueUforUparameter.\n\n");
        else fprintf(stderr, "\n\n");
        exit(1);
      }
  }
  if (i < k) fprintf(stderr, "UsingUkU=U%d.\n", i);
  k = i;
  start = clock();
  sf = sf_chlp(graph, term_sets, k);
  end = clock();
  break;
case KLEIN: ⟨ Allocate the extra fields of the vertices 251 ⟩
  fprintf(stdout, "*****UKleinU*****\n");
  start = clock();
  sf = sf_klein(graph, term_sets);
  end = clock();
  break;
case GGW: fprintf(stdout, "*****UGabow,UGoemansUandUWilliamsonU*****\n");
  start = clock();
  sf = sf_ggw(graph, term_sets, random);
  end = clock();
  break;
case ALL: fprintf(stdout, "*****UGoemansUandUWilliamsonU*****\n");
  start = clock();
  sf = sf_gw(graph, term_sets);
  end = clock();
  ⟨ Show the results 258 ⟩
  for (ts = term_sets; ts < term_sets + graph→num_ts; ts++) ts→connected = 0;
  ⟨ Initialize the extra fields of the arcs 250 ⟩
  fprintf(stdout,
      "*****U""Cole,UHariharan,ULewensteinUandUPoratU[kU=U%d]""U*****\n", k);
```

```
      start = clock();
      sf = sf_chlp(graph, term_sets, k);
      end = clock();
      ⟨Show the results 258⟩
      for (ts = term_sets; ts < term_sets + graph→num_ts; ts++) ts→connected = 0;
      ⟨Initialize the extra fields of the arcs 250⟩
      fprintf(stdout, "*****␣Gabow,␣Goemans␣and␣Williamson␣*****\n");
      start = clock();
      sf = sf_ggw(graph, term_sets, random);
      end = clock();
      ⟨Show the results 258⟩
      for (ts = term_sets; ts < term_sets + graph→num_ts; ts++) ts→connected = 0;
      ⟨Initialize the extra fields of the arcs 250⟩
      ⟨Allocate the extra fields of the vertices 251⟩
      fprintf(stdout, "*****␣Klein␣*****\n");
      start = clock();
      sf = sf_klein(graph, term_sets);
      end = clock();
      break;
    }
  }
```

Este código é usado no bloco 238.

255 ⟨Header files of sf.c 2⟩ +≡

```
#include <time.h>
#include <limits.h>
```

256 ⟨Local variables of function *main* 241⟩ +≡

```
  SteinerForest *sf;
  clock_t start, end;
```

257 ⟨Global variables of sf.c 257⟩ ≡

```
  int err;
```

Este código é usado no bloco 1.

O trecho de código abaixo é responsável por imprimir as arestas da floresta de Steiner sf contruída por uma das implementações, informando, além disso, a razão de aproximação desta floresta com relação a solução ótima.

258 ⟨ Show the results 258 ⟩ ≡

```
{
  Arc *a;
  double r;
  if (show_edges) {
    fprintf(stdout, "\nEdges:\n");
    for (a = sf→edges; a; a = a→next_edge)
      fprintf(stdout, "(%s,␣%s)\n", a→from→name, a→tip→name);
  }
  if (opt ≡ −1) r = ((double) sf→cost)/(sf→dual_cost);
  else r = ((double) sf→cost)/opt;
  fprintf(stdout, "\nCost:␣%ld␣[<=␣%.4f␣opt]\n", sf→cost, r);
  fprintf(stdout, "\nTime:␣%.3fs\n\n", ((double)(end − start))/CLOCKS_PER_SEC);
  if (test) {
    if (¬isSteinerForest(sf, graph, term_sets))
      printf("#␣Something␣bad␣has␣ocurred...\n\n");
    else {
      if (¬isMinimal(sf, graph, term_sets))
        printf("#␣Something␣bad␣has␣ocurred...\n\n");
      else printf("#␣It's␣everithing␣all␣right!\n\n");
    }
  }
}
```

Este código é usado nos blocos 238 e 254.

259

```
#define mark a.I
#define next_elem w.V
#define _from_(i) ((g→vertices + i)→x.V)
```

⟨ Auxiliary functions 20 ⟩ +≡

```
int isSteinerForest(SteinerForest *sf, Graph *g, TermSet *term_sets)
{
  Arc *a;
  Vertex *v;
  TermSet *ts;
```

```
int i;
for (a = sf→edges, i = 0; a; a = a→next_edge, i++) _from_(i) = a→from;
for (v = g→vertices; v < g→vertices + g→n; v++) {
  v→visit = 0;
  for (a = v→arcs; a; a = a→next) a→mark = 0;
}
for (a = sf→edges, i = 0; a; a = a→next_edge, i++) {
  Arc *a_ = (_from_(i) < a→tip) ? a + 1 : a − 1;
  a→mark = a_→mark = 1;
}    /* os terminais estão todos conectados? */
for (ts = term_sets; ts < term_sets + g→num_ts; ts++) {
  Vertex *head, *tail;
  v = ts→vertices;
  v→visit = 1;
  v→next_elem = Λ;
  head = tail = v;
  while (head) {
    Vertex *u = head;
    for (a = u→arcs; a; a = a→next) {
      Vertex *w = a→tip;
      if (¬a→mark ∨ w→visit) continue;
      w→visit = 1;
      w→next_elem = Λ;
      tail→next_elem = w;
      tail = w;
    }
    head = head→next_elem;
  }
  for (v = ts→vertices; v; v = v→next_terminal)
    if (¬v→visit) {
#ifdef VERBOSE
      printf("----->␣There␣are␣disconnected␣terminal␣sets\n");
#endif    /* reestabelecendo o valor do campo from dos arcos */
      for (v = g→vertices; v < g→vertices + g→n; v++) {
        Arc *a;
        for (a = v→arcs; a; a = a→next) a→from = v;
      }
```

```
        return 0;
      }
    for (v = g→vertices; v < g→vertices + g→n; v++) v→visit = 0;
  }     /* as arestas em sf são arestas de uma floresta? */
#ifdef DEFEITO2
  markAllEdges(g);
#endif
  for (v = g→vertices; v < g→vertices + g→n; v++) v→visit = 0;
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    Vertex *head, *tail;
    if (v→visit) continue;
    v→visit = 1;
    v→next_elem = Λ;
    head = tail = v;
    while (head) {
      Vertex *u = head;
      int n = 0;
      for (a = u→arcs; a; a = a→next) {
        Vertex *w = a→tip;
        if (¬a→mark) continue;
        if (w→visit) {
          n++;
          continue;
        }
        w→visit = 1;
        w→next_elem = Λ;
        tail→next_elem = w;
        tail = w;
      }
      if (n ≥ 2) {
#ifdef VERBOSE
        printf("----->␣There␣is␣a␣cycle␣in␣sf!\n");
#endif    /* reestabelecendo o valor do campo from dos arcos */
        for (v = g→vertices; v < g→vertices + g→n; v++) {
          Arc *a;
          for (a = v→arcs; a; a = a→next) a→from = v;
        }
        return 0;
```

```
        }
        head = head→next_elem;
      }
    }    /* reestabelecendo o valor do campo from dos arcos */
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      Arc *a;
      for (a = v→arcs; a; a = a→next) a→from = v;
    }
    return 1;
  }
```

260 ⟨Auxiliary functions 20⟩ +≡ /* teste de minimalidade */

```
  int isMinimal(SteinerForest *sf, Graph *graph, TermSet *term_sets)
  {
    int i, j, num_edges;
    Arc *a, **edges;
    printf("␣(Teste␣de␣Minimalidade)\n\n");
    num_edges = 0;
    for (a = sf→edges; a; a = a→next_edge) num_edges++;
#ifdef VERBOSE
    printf("\nnum_edges␣=␣%d\n\n", num_edges);
#endif
    edges = malloc(num_edges * sizeof(Arc *));
    for (i = 0, a = sf→edges; a; i++, a = a→next_edge) edges[i] = a;
    j = -1;
    do {
      j++;
      sf→edges = Λ;
      for (i = 0; i < j; i++) {
        edges[i]→next_edge = sf→edges;
        sf→edges = edges[i];
      }
      for (i = j + 1; i < num_edges; i++) {
        edges[i]→next_edge = sf→edges;
        sf→edges = edges[i];
      }
#ifdef VERBOSE
      printf("%d:␣", j);
```

```
#endif
        if (isSteinerForest(sf, graph, term_sets)) {
          free(edges);
          return 0;
        }
      } while (j < num_edges - 1);
      free(edges);
      return 1;
    }
```

Referências Bibliográficas

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: an approximation algorithm for the generalizad Steiner problem on networks. *SIAM Journal on Computing*, 24(2):440–456, 1995.

[2] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.

[3] M.H. Carvalho, M.R. Cerioli, R. Dahab, P. Feofiloff, C.G. Fernandes, C.E. Ferreira, F.K. Miyazawa, J.C. de Pina, J. Soares, and Y. Wakabayashi. *Uma Introdução Sucinta a Algoritmos de Aproximação*. Publicações Matemàticas do IMPA, 2001.

[4] F. Chudak, T. Roughgarden, and D.P. Williamson. Approximate k-MSTs and k-Steiner trees via the primal-dual method and Lagrangean relaxation. In *Proc. 8th Conference on Integer Programming and Combinatorial Optimization Conference (IPCO)*, volume 2081 of *Lecture Notes in Computer Science*, page 60 ff. Springer, 2001.

[5] R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. A faster implementation of the Goemans-Williamson clustering algorithm. In *Symposium on Discrete Algorithms*, pages 17–25, 2001.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[7] J.C. de Pina. Seleção em tempo linear. In *Notas de aula de Análise de Algoritmos*. 2004. Disponível em http://www.ime.usp.br/~coelho/algoritmos/aulas/material/aula14.pdf.

[8] Harold N. Gabow, Zvi Galil, Thomas H. Spencer, and Robert Endre Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[9] H.N. Gabow, M.X. Goemans, and D.P. Williamson. An efficient approximation algorithm for the survivable network design problem. *Mathematical Programming*, 82(1-2, Ser. B):13–40, 1998.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, 1979.

[11] N. Garg, V.V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18:3–20, 1997.

[12] M.X. Goemans and D.P. Williamson. A general approximation technique for constrained forest problems. In G. Frederickson, R.L. Graham, D.S. Hochbaum, E. Johnson, S.R. Kosaraju, M. Luby, N. Megiddo, B. Schieber, P. Vaidya, and F. Yao, editors, *Proceedings of the Third ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 307–316. SIAM, 1992.

[13] M.X. Goemans and D.P. Williamson. Primal-dual approximation algorithms for feedback problems in planar graphs. Manuscript, 1995.

[14] D.S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, 1997.

[15] K. Jain, I. Măndoiu, V.V. Vazirani, and D.P. Williamson. A primal-dual schema based approximation algorithm for the element connectivity problem. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 484–489, 1999.

[16] K. Jain and V.V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. *17th International Symposium on Mathematical Programming (ISMP)*, 2000. URL: `http://www.cc.gatech.edu/people/home/kjain/`.

[17] P. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Information Processing Letters*, 52(6):303–307, 1994.

[18] D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[19] D.E. Knuth. *Literate Programming.* CSLI, 1992.

[20] D.E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, 1993.

[21] D.E. Knuth and S. Levy. *The* CWEB *System of Structured Documentation.* Addison-Wesley, 1994.

[22] S. Rajagopalan and V.V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM Journal on Computing*, 28(2):526–541, 1999.

[23] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. In *Proceedings of the Tenth ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 770–779, 2000.

[24] B. Schieber. *Synthesis of Parallel Algorithms*, chapter 6, pages 259–273. Morgan Kaufmann Publishers, Inc., 1993.

[25] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[26] J. Szwarcfiter and L. Markenzon. *Estruturas de Dados e Seus Algoritmos.* LTC Editora, 2. edition, 1994.

[27] V.V. Vazirani. *Approximation Algorithms.* Springer, 2001.

Lista de Refinamentos

⟨ Allocate memory to other variables 118, 157 ⟩ Usado no bloco 109.

⟨ Allocate space for C^2 bicategories 109 ⟩ Usado no bloco 96.

⟨ Allocate the extra fields of the arcs 248 ⟩ Usado no bloco 238.

⟨ Allocate the extra fields of the vertices 251 ⟩ Usado no bloco 254.

⟨ Assign a direction to each edge of g and initialize the fields of arcs and vertices 101 ⟩ Usado no bloco 96.

⟨ Assign each arc to one of its two endpoints 114 ⟩ Usado no bloco 96.

⟨ Assign one category to each vertex of g and make some initializations 174 ⟩ Usado no bloco 167.

⟨ Assign a to v 122 ⟩ Usado no bloco 114.

⟨ Assign a to w 124 ⟩ Usado no bloco 114.

⟨ Auxiliary functions 20, 27, 29, 51, 59, 64, 65, 69, 84, 91, 176, 180, 181, 184, 185, 202, 204, 211, 259, 260 ⟩ Usado no bloco 1.

⟨ Change the assignment of the arcs incident to x or y if necessary 159 ⟩ Usado no bloco 145.

⟨ Change the bicategory of each arc in the *extra_in* list of v 137 ⟩ Usado no bloco 130.

⟨ Change the bicategory of each arc in the *extra_out* list of v 142 ⟩ Usado no bloco 130.

⟨ Change the bicategory of the arcs assigned to v 131 ⟩ Usado no bloco 130.

⟨ Change the endpoint which b is assigned to 160 ⟩ Usado no bloco 159.

⟨ Choose an adequate element x to partitioning a 67 ⟩ Usado no bloco 65.

⟨ Choose the awake edges incident to the new component 85 ⟩ Usado no bloco 79.

⟨ Clear the positions of the vector *min_cost_arc* 147 ⟩ Usado no bloco 145.

⟨ Compute the ascendant number of each vertex and construct the *head* table 230 ⟩ Usado no bloco 225.

⟨ Compute the level and the inlabel number of each vertex 228 ⟩ Usado no bloco 225.

⟨ Concatenate the lists `C1`→*vertices* and `C2`→*vertices* 36 ⟩ Usado nos blocos 35 e 79.

⟨ Construct the list of terminals of each terminal set 214 ⟩ Usado no bloco 213.

⟨ Construct the list C→*edges* 24 ⟩ Usado no bloco 25.

⟨ Create a double linked list of edges incident in C 55 ⟩ Usado no bloco 53.

⟨ Create a heap that contains the incident edges of each component 25 ⟩ Usado no bloco 10.

⟨ Create a spanning forest *sf* with no edges 13 ⟩ Usado no bloco 10.

⟨Create an edge heap for each component 194⟩ Usado no bloco 192.

⟨Create the initial spanning forest 192⟩ Usado no bloco 187.

⟨Data structures of `bicat.c` 106, 108⟩ Usado no bloco 95.

⟨Data structures of `sf.c` 4, 6, 11, 46, 47, 183, 186⟩ Citado no bloco 4. Usado no bloco 1.

⟨Delete all awake edges incident to `C1` or `C2` from their packets and from D 81⟩ Usado no bloco 79.

⟨Delete from their packets and from D the edges in x→*packet* 83⟩ Usado no bloco 81.

⟨Delete from *edges* the edges in `C1`→*edges* 41⟩ Usado no bloco 40.

⟨Delete a from the lists `C1`→*edges* and `C2`→*edges* 80⟩ Usado no bloco 79.

⟨Determine the necessary edges with the aid of the lcas 217⟩ Usado no bloco 211.

⟨Discard all uv-arcs, knowing that this arcs are assigned to v 150⟩ Usado nos blocos 149 e 152.

⟨Discard all xy-arcs from the graph 149⟩ Usado no bloco 145.

⟨Discard all yx-arcs from the graph 152⟩ Usado no bloco 145.

⟨Discard from *sf* unnecessary edges and **return** the resulting Steiner forest 208⟩ Usado no bloco 187.

⟨Discard parallel arcs in the *extra_in* list of v 134⟩ Usado nos blocos 130 e 145.

⟨Do the same for `C2`→*edges* and merge the lists `C1`→*edges* and `C2`→*edges* 42⟩ Usado no bloco 40.

⟨Do the same for `C2`→*edges* and merge `C1`→*edges* and `C2`→*edges* 88⟩ Usado no bloco 85.

⟨Drop out unnecessary edges from *sf* and return a minimal Steiner forest 43⟩ Usado no bloco 10.

⟨Edge selection auxiliary functions 66, 68, 70⟩ Usado no bloco 65.

⟨External functions of `bicat.c` 96, 99, 126, 127, 130, 145, 164⟩ Citado nos blocos 95 e 97. Usado no bloco 95.

⟨External functions of lca.c 225, 233⟩ Usado no bloco 221.

⟨Find a Steiner forest 254⟩ Usado no bloco 238.

⟨Find and return the lca of x and y 236⟩ Usado no bloco 233.

⟨Find and **return** a minimal Steiner forest 92⟩ Usado no bloco 45.

⟨Find and **return** a subgraph of *sf* that is a minimal Steiner forest 177⟩ Usado no bloco 167.

⟨Find the lca of each terminal set 213⟩ Usado no bloco 211.

⟨Find *lca_inlabel*, the inlabel number of the lca of x and y 234⟩ Usado no bloco 233.

⟨For each bicategory b, create the heaps $H^-(v,b)$ and $H^+(v,b)$ 117⟩ Usado no bloco 114.

⟨For each bicategory B, reinsert in $H(B)$ the heaps in *H_out_x* 161⟩ Usado no bloco 159.

⟨For each bicategory b, join the heaps of x and y in b 153⟩ Usado no bloco 145.

⟨For each edge a in g split a in two pieces 191⟩ Usado no bloco 187.

⟨For each vertex v in some active component, do $d(v) \mathrel{+}= inc$ 34⟩ Usado nos blocos 30 e 76.

⟨For each v and each b insert $H^-(v,b)$ and $H^+(v,b)$ in $H(b)$ 125⟩ Usado no bloco 114.

⟨Free the memory allocated by *sf_chlp* 209⟩ Usado no bloco 208.

⟨Free the memory allocated by *sf_ggw* 93⟩ Usado no bloco 92.

⟨Free the memory allocated by *sf_gw* 44⟩ Usado no bloco 43.

⟨ Free the memory allocated by *sf_klein* 178 ⟩ Usado no bloco 177.
⟨ Free the memory allocated to the old structures 90 ⟩ Usado no bloco 89.
⟨ Free the node pointed to by x 82 ⟩ Usado nos blocos 81 e 90.
⟨ Global variables of `bicat.c` 100, 103, 110, 115, 119, 128, 136, 158 ⟩ Usado no bloco 95.
⟨ Global variables of `sf.c` 257 ⟩ Usado no bloco 1.
⟨ Global variables of lca.c 226 ⟩ Usado no bloco 221.
⟨ Header files of `bicat.c` 98, 107, 112, 116, 123 ⟩ Usado no bloco 95.
⟨ Header files of `sf.c` 2, 8, 9, 28, 50, 166, 182, 216, 242, 255 ⟩ Citado nos blocos 2 e 9. Usado no bloco 1.
⟨ Header files of lca.c 222, 224, 231, 237 ⟩ Usado no bloco 221.
⟨ Include an edge in the forest while there is some active component 76 ⟩ Usado no bloco 45.
⟨ Include an edge piece in the forest while there is some active component 196 ⟩ Usado no bloco 187.
⟨ Include each vertex of V in U and set *active_* to the new state of U 201 ⟩ Usado no bloco 200.
⟨ Include a in the forest *sf* 33 ⟩ Usado nos blocos 30, 76 e 167.
⟨ Include uv in the spanning forest *sf* 198 ⟩ Usado no bloco 196.
⟨ Initializations 172 ⟩ Usado no bloco 174.
⟨ Initialize some auxiliary variables 104 ⟩ Usado no bloco 101.
⟨ Initialize some fields of the component C of v 16 ⟩ Usado nos blocos 13 e 53.
⟨ Initialize the extra fields of the arcs 250 ⟩ Usado nos blocos 248 e 254.
⟨ Initialize the field *from* of each arc 247 ⟩ Usado no bloco 238.
⟨ Initialize the fields of the component C of v 53 ⟩ Usado no bloco 52.
⟨ Initialize the lists $extra^-(v)$ and $extra^+(v)$ 105 ⟩ Usado no bloco 101.
⟨ Initialize the packets of each component 57 ⟩ Usado nos blocos 45 e 89.
⟨ Initialize the structures corresponding to the components of the initial forest 52 ⟩ Usado no bloco 45.
⟨ Initialize $d(v)$ for each vertex v of the graph 18 ⟩ Usado nos blocos 10 e 45.
⟨ Insert $C\to gr_packet$ in the list $C\to packets$ 75 ⟩ Usado no bloco 71.
⟨ Internal functions of `bicat.c` 111, 113, 120, 121, 155, 162 ⟩ Usado no bloco 95.
⟨ Internal functions of lca.c 227, 229, 232, 235 ⟩ Usado no bloco 221.
⟨ Join the components `C1` and `C2` 79 ⟩ Usado no bloco 76.
⟨ Join the heaps $H^+(x, B)$ and $H^+(y, B)$ 154 ⟩ Usado no bloco 153.
⟨ Join the heaps $H^-(x, B)$ and $H^-(y, B)$ 156 ⟩ Usado no bloco 153.
⟨ Let a be an edge with the smallest slackness *inc* 32 ⟩ Usado no bloco 30.
⟨ Local variables of function *main* 241, 243, 249, 253, 256 ⟩ Usado no bloco 238.
⟨ Local variables of *contractEdge* 146, 148, 151 ⟩ Usado no bloco 145.
⟨ Local variables of *sf_chlp* 190, 193, 195, 197 ⟩ Usado no bloco 187.
⟨ Local variables of *sf_ggw* 49, 54, 58, 63, 74, 77, 86 ⟩ Usado no bloco 45.
⟨ Local variables of *sf_gw* 14, 17, 19, 23, 26, 31, 39 ⟩ Usado no bloco 10.

⟨Local variables of *sf_klein* 168, 170, 173⟩ Usado no bloco 167.
⟨Make a an awake edge 71⟩ Usado nos blocos 61 e 85.
⟨Memory allocation and parameter settings 48, 56, 62, 73⟩ Usado no bloco 45.
⟨Merge the components C1 and C2 35⟩ Usado no bloco 30.
⟨More auxiliary functions 212, 218⟩ Usado no bloco 211.
⟨Move to the bicategory *new* the arcs in $H^+(v, old)$ 132⟩ Usado no bloco 131.
⟨Move to the bicategory *new* the arcs in $H^-(v, old)$ 133⟩ Usado no bloco 131.
⟨Multiply by n^k the cost of each edge in g 188⟩ Usado no bloco 187.
⟨Other initializations 60, 72⟩ Usado no bloco 57.
⟨Other local variables of *changeCategory* 138, 141⟩ Usado no bloco 130.
⟨Print a help message and exit 240⟩ Usado no bloco 239.
⟨Process the comand line 239⟩ Usado no bloco 238.
⟨Put the uv-arc a in the bicategory *new* 140⟩ Usado no bloco 137.
⟨Put the vw-arc a in the bicategory *new* 144⟩ Usado no bloco 142.
⟨Read the edges from the file; exit if unsuccessful 246⟩ Usado no bloco 244.
⟨Read the terminal sets from the file; exit if unsuccessful 252⟩ Usado nos blocos 239 e 244.
⟨Read the vertices from the file; exit if unsuccessful 245⟩ Usado no bloco 244.
⟨Remove the uv-arc a from the bicategory *old* 139⟩ Usado no bloco 137.
⟨Remove the vw-arc a from the bicategory *old* 143⟩ Usado no bloco 142.
⟨Restore the original costs of the edges of g 189⟩ Usado no bloco 187.
⟨Select an edge a with the smallest slackness *delta* 169⟩ Usado no bloco 167.
⟨Select an edge a with the smallest slackness *inc* 78⟩ Usado no bloco 76.
⟨Select the $2r$ smallest edges in the list C1→*edges* 61⟩ Usado no bloco 57.
⟨Set *cat* to the state of the component in *sf* that contains a 175⟩ Usado no bloco 167.
⟨Set uv and wv to represent the same terminal edge 206⟩ Usado no bloco 205.
⟨Set uv and wv to represent two external edges 207⟩ Usado no bloco 205.
⟨Set uv to point to an edge with the smallest slackness 199⟩ Citado no bloco 205. Usado no bloco 196.
⟨Set x to represent the vertex resulting from the contraction of a 163⟩ Usado no bloco 145.
⟨Show the results 258⟩ Usado nos blocos 238 e 254.
⟨Some initializations 15, 22, 38⟩ Citado no bloco 2. Usado nos blocos 13 e 52.
⟨Split the edge wv 205⟩ Usado no bloco 196.
⟨Steiner forest construction functions 10, 45, 167, 187⟩ Usado no bloco 1.
⟨The main program 238⟩ Usado no bloco 1.
⟨Try to create a graph from the file; exit if unsuccessful 244⟩ Usado no bloco 239.
⟨Unite the components U and V 200⟩ Usado no bloco 196.
⟨Unite the edge heaps of U and V 203⟩ Usado no bloco 200.
⟨Update the field *act* of the *edges* in C1→*edges* 87⟩ Usado no bloco 85.

⟨ Update the heap *edges* 40 ⟩ Usado no bloco 35.
⟨ Update the structures for the new subphase 89 ⟩ Usado no bloco 76.
⟨ Update `C1→`*active* 37 ⟩ Usado nos blocos 35 e 79.
⟨ While there is some active component include an edge in the forest *sf* 30 ⟩ Usado no bloco 10.
⟨ `bicat.h` 97, 102, 129, 165 ⟩
⟨ `extra.h` 7 ⟩ Citado no bloco 2.
⟨ find the lca of each terminal set in the list *ts_list* 215 ⟩ Usado no bloco 213.
⟨ `intersection.h` 171 ⟩
⟨ `lca.h` 223 ⟩
⟨ `list.h` 21 ⟩
⟨ select the edges in the path from z to w in *sf_graph* 219 ⟩ Usado no bloco 217.
⟨ **if** b's cost is greater than a's cost, discard b; **else** discard a 135 ⟩ Usado no bloco 134.