Relatório de Estudos
MAC 5701 - Tópicos em Ciência da Computação

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

Aluno: Rafael Pereira Luna
Orientadora: Cristina Gomes Fernandes

São Paulo, 01 de Dezembro de 2003

Sumário

1. Introdução

Apresentaremos aqui o projeto desenvolvido durante o segundo semestre de 2003 dentro das atividades que fazem parte do escopo da disciplina MAC 5701 - Tópicos em Ciência da Computação. Tais atividades foram voltadas para o estudo de algumas das possíveis implementações do método de aproximação primal-dual aplicado ao problema da floresta de Steiner.

Inicialmente, definiremos o problema em questão e daremos uma descrição do algoritmo de aproximação primal-dual proposto por Goemans e Williamson para ele [2]. Após isso, apresentaremos três maneiras distintas de implementar este algoritmo, baseadas nas implementações sugeridas por Goemans e Williamson [2], Cole, Hariharan, Lewenstein e Porat [3], e Klein [5], exibindo ainda uma implementação em CWEB para as duas primeiras.

No programa CWEB correspondente as duas primeiras implementações, com o intuito de implementar de forma mais rápida a segunda fase do algoritmo de Goemans e Williamson, usamos uma técnica baseada em um algoritmo para encontrar o ancestral comum mais próximo de dois vértices em uma árvore enraizada. Apresentaremos esta técnica logo após a descrição da implementação proposta por Klein e, em seguida, exibiremos uma implementação em CWEB do algoritmo devido a Schieber e Vishkin no qual a ténica se baseia. Ao final, faremos algumas considerações finais sobre as atividades realizadas durante o semestre e sobre alguns planos para atividades futuras.

2. O Problema

Nesta e nas próximas seções, 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.

O **problema da floresta de Steiner** consiste do seguinte: dados um grafo G, uma função custo c de E_G em $Q_{\geq}$ e uma coleção R de subconjuntos de V_G, encontrar uma *R-floresta* F que minimize $c(F) = \sum_{e \in E_F} c_e$.

Cada conjunto da coleção R é chamado de *conjunto de terminais* e os vértices que não pertencem aos conjuntos de R são chamados de *vértices de Steiner*. Uma *R-floresta* F é uma floresta geradora de G que tem a seguinte propriedade: cada conjunto de terminais em R se encontra inteiramente contido no conjunto dos vértices de alguma das componentes de F. Quando R está implícito, dizemos que uma floresta satisfazendo tal propriedade é uma *floresta de Steiner* de G. Note que podemos assumir que os conjuntos de R são dois a dois disjuntos (caso contrário, bastaria substituir cada par de conjuntos que não satisfaz esta restrição pela união dos conjuntos no par).

Não se conhece um algoritmo eficiente (polinomial no tamanho da entrada (G, c, R)) para resolver o problema acima. Se a coleção R consiste em apenas um conjunto, o problema se reduz ao conhecido *problema de Steiner em grafos*, o qual é NP-difícil.

3. O Algoritmo

As implementações que estudaremos são todas baseadas num algoritmo de aproximação projetado por Goemans e Williamson [2] para o problema. Tal algoritmo utiliza o método de aproximação primal-dual e é uma 2-aproximação para o problema da floresta de Steiner (uma explicação detalhada sobre algoritmos de aproximação, incluindo o método de aproximação primal-dual, pode ser encontrada em [1]).

4. Idéia geral do algoritmo

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

Note que se F é uma 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. Assim, podemos enunciar o problema da floresta de Steiner na forma de um programa inteiro, como segue abaixo.

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

Neste programa, x é um vetor indexado por E_G e c é a função custo definida no problema. Dada uma floresta geradora F, 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.

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

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

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

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

O algoritmo parte de uma solução viável y do programa dual e de uma floresta geradora 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, 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 e tem início uma nova iteração do algoritmo. No momento em que nenhuma das componentes da floresta corrente for ativa o algoritmo termina, devolvendo uma sub–floresta de Steiner minimal contida na floresta corrente. 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, esse algoritmo.

5. 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. Na verdade, y não é armazenado explicitamente; ao invés disso, guarda-se o valor da função objetivo dual em y e, para cada vértice v do grafo, o número $d(v)$ que é a soma das posições de y correspondentes às componentes às quais v pertenceu nas florestas das iterações anteriores (incluindo a floresta corrente). A primeira iteração começa com uma floresta 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 e, assim, $d(v) = 0$ para todo $v \in V_G$.

Em cada iteração, uma *aresta externa* é incluída na floresta; uma *aresta externa* nada mais é do que uma aresta que une duas componentes distintas da floresta. A aresta externa escolhida pelo algoritmo para ser incluída na floresta é aquela que apresenta a menor *folga* dentre todas as outras (na verdade, pode haver mais de uma tal aresta). O valor da folga de uma aresta uv é calculado da seguinte forma:

- $(c_{uv} - d(u) - d(v))/2$, se u e v pertencem a componentes ativas;
- $c_{uv} - d(u) - d(v)$, se u ou v, mas não ambos, pertence a uma componente ativa;
- ∞, se u e v pertencem a componentes inativas.

Em caso de empate, é dada preferência para arestas que tenham cada um dos extremos em uma componente ativa.

Uma vez que foi escolhida uma aresta, atualiza-se o custo da floresta (somando-se o custo da aresta incluída), o valor da função objetivo dual (somando-se a folga da aresta incluída multiplicada pelo número de componentes ativas) e o valor de $d(v)$ para cada vértice v em uma componente ativa (somando-se a folga da aresta incluída).

O algoritmo pára quando todas as componentes da floresta estiverem inativas (o que ocorre após, no máximo, $n-1$ iterações), devolvendo uma floresta de Steiner minimal contida na floresta corrente.

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

Algoritmo *MinFS(G, R, c)*

$F \leftarrow (V_G, \emptyset)$
$E_{ext} \leftarrow E_G$
$d(v) \leftarrow 0$ para cada v em V_G
enquanto F tem alguma componente ativa faça
    escolha $\overline{a}$ tal que $\mathit{folga}(\overline{a}) = \min\{\mathit{folga}(a) \mid a \in E_{ext}\}$
    $d(v) \leftarrow d(v) + \mathit{folga}(\overline{a})$ para cada v em uma componente ativa
    inclua $\overline{a}$ em F
    $E_{ext} \leftarrow E_{ext} \setminus \{uv \in E_G \mid u$ e v pertencem a uma mesma componente de $F\}$
seja F' uma R-floresta minimal contida em F
devolva F'

6. Implementações

O algoritmo descrito na seção anterior pode ser implementado de diferentes maneiras. A implementação originalmente proposta por Goemans e Williamson [2] consome tempo $O(n^2 \log n)$, onde n é o número de vértices do grafo de entrada, e é bastante adeqüada para grafos densos. Subseqüentemente, a complexidade de tempo desta implementação foi melhorada por outras, como as de Gabow, Goemans e Williamson [4], Klein [5], e Cole, Hariharan, Lewenstein e Porat [3].

Aqui, iremos apresentar três dessas implementações: a original, de Goemans e Williamson, e as implementações de Cole, Hariharan, Lewenstein e Porat, e de Klein. Além disso, exibiremos ainda uma implementação em CWEB para essas duas primeiras. Deste forma, vamos antes apresentar a estrutura geral 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.

$\langle$ Header files 9 $\rangle$
$\langle$ Data structures 17 $\rangle$
$\langle$ Auxiliary functions 24 $\rangle$
$\langle$ Steiner forest construction functions 33 $\rangle$
int *main*(**int** *argc*, **char** $*$*argv*[ ])
{
  $\langle$ Local variables of function *main* 10 $\rangle$
  $\langle$ Process the comand line 7 $\rangle$
  $\langle$ Find a Steiner forest 108 $\rangle$
  $\langle$ Show the results 109 $\rangle$
}

7. 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. 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 [6], 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`.

```
⟨ Process the comand line 7 ⟩ ≡
  {
    FILE *file;
    if (argc < 2) ⟨ Print a help message and exit 8 ⟩
    if (¬strcmp(argv[1], "-sgb")) {
      if (argc < 4) ⟨ Print a help message and exit 8 ⟩
      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 18 ⟩
    }
    else {
      if (¬(file = fopen(argv[1], "r"))) {
        fprintf(stderr, "Cannot␣open␣file␣%s.\n", argv[2]);
        exit(1);
      }
      ⟨ Try to create a graph from the file; exit if unsuccessful 11 ⟩
    }
    fclose(file);
    ⟨ Initialize the field origin of each edge 16 ⟩
  }
```

Este código é usado no bloco 6.

8. Se o usuário digitar algo errado na linha de comando, será exibida uma mensagem explicando brevemente a forma correta de se executar o programa.

```
⟨ Print a help message and exit 8 ⟩ ≡
  {
    printf("\n␣sf␣[-sgb␣arquivo_gb]␣arquivo_de_entrada\n");
    exit(1);
  }
```

Este código é usado no bloco 7.

9. 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.

⟨ Header files 9 ⟩ ≡
#include `<stdio.h>`
#include `<stdlib.h>`
#include `<string.h>`
#include `"gb_graph.h"`
#include `"gb_save.h"`

Veja também blocos 104, 117, 119, 126 e 132.

Este código é usado nos blocos 6 e 116.

10. Vamos declarar aqui 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.

⟨ Local variables of function *main* 10 ⟩ ≡
Graph $*graph$;

Veja também blocos 12, 15 e 110.

Este código é usado no bloco 6.

11. 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 descrito grafo 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.

```
⟨ Try to create a graph from the file; exit if unsuccessful 11 ⟩ ≡
  {
    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 13 ⟩
    ⟨ Read the edges from the file; exit if unsuccessful 14 ⟩
    ⟨ Read the terminal sets from the file; exit if unsuccessful 18 ⟩
  }
```

Este código é usado no bloco 7.

12.
```
⟨ Local variables of function main 10 ⟩ +≡
  long num_vertices;
```

13.
```
⟨ Read the vertices from the file; exit if unsuccessful 13 ⟩ ≡
  {
    Vertex *v;
    char *name = (char *) malloc(30 * sizeof(char));
    for (v = graph→vertices; v < graph→vertices + num_vertices; v++) {
      if (¬fscanf(file, "␣%s", name)) {
        fprintf(stderr, "There␣are␣less␣than␣%ld␣vertices␣in␣file␣%s.\n", num_vertices,
            (argc ≡ 4) ? argv[3] : argv[2]);
        exit(1);
      }
      v→name = gb_save_string(name);
```

```
  }
  free(name);
}
```

Este código é usado no bloco 11.

14. ⟨ Read the edges from the file; exit if unsuccessful 14 ⟩ ≡

```
{
  long 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 11.

15. ⟨ Local variables of function *main* 10 ⟩ +≡

```
long num_edges;
```

16. 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 *origin* que contém um apontador para o vértice de origem do arco.

```
#define origin a.V
```

⟨ Initialize the field *origin* of each edge 16 ⟩ ≡

```
{
  Vertex *v;
  for (v = graph→vertices; v < graph→vertices + num_vertices; v++) {
    Arc *a;
    for (a = v→arcs; a; a = a→next) a→origin = v;
  }
}
```

Este código é usado no bloco 7.

17. Internamente, cada conjunto de terminais é representado por uma estrutura do tipo *TermSet*, a qual será descrita neste bloco. Esta estrutura possui um identificador *id* e um campo *connected*, o qual indica se 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). Além disso, esta 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* é um apontador para uma estrutura do mesmo tipo e pode ser utilizado, por exemplo, para a criação de listas ligadas de conjuntos de terminais.

```
#define next_terminal u.V    /* next vertex in the linked list vertices */
⟨ Data structures 17 ⟩ ≡
  typedef struct ts_struct {
    int id;
    Vertex *vertices;
    int num_vertices;
    int connected;
    struct ts_struct *next_tset;
  } TermSet;
```

Veja também blocos 19, 21, 22, 49, 50, 53 e 67.

Este código é usado no bloco 6.

18. A leitura da especificação dos conjuntos de terminais e a criação das estruturas que os representam são feitas no trecho de código abaixo. O campo *t_sets* da estrutura que representa o grafo da entrada é um apontador para um vetor contendo os conjuntos de terminais do grafo e o campo *num_ts* é o número de elementos deste vetor. Além disso, cada vértice do grafo contém um campo *termset* que guarda um apontador para o conjunto de terminais do qual o vértice faz parte (o valor deste campo será igual a Λ para os vértices que não pertencem aos conjuntos de terminais do grafo).

```
#define num_ts uu.I    /* number of terminal sets */
#define t_sets vv.S    /* array of terminal sets */
#define termset v.S    /* pointer to the terminal set of the vertex */
⟨ Read the terminal sets from the file; exit if unsuccessful 18 ⟩ ≡
  {
    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 + num_vertices; v++) v→termset = Λ;
    ((TermSet *) graph→t_sets) = (TermSet *) malloc((graph→num_ts) * sizeof(TermSet));
    for (i = 0; i < graph→num_ts; i++) {
      TermSet *ts = ((TermSet *) graph→t_sets) + i;
      Vertex *t = ts→vertices = (Vertex *) malloc(sizeof(Vertex));
      char c;
      int n = 0;
```

```
        c = getc(file);
        ungetc(c, file);
        while (c ≠ EOF ∧ c ≠ '\n') {
          int v;
          if (fscanf(file, "%d%c", &v, &c) ∧ v ≥ 0 ∧ v < graph→n) {
            ((TermSet *)(graph→vertices + v)→termset) = ts;
            t→next_terminal = (graph→vertices + v);
            t = t→next_terminal;
            n++;
          }
          else {
            fprintf(stderr, "Invalid␣value␣for␣the␣element␣%d␣of␣terminal␣set␣%d.",
                n + 1, i + 1);
            exit(1);
          }
        }
        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);
        }
        t→next_terminal = Λ;
        t = ts→vertices;
        ts→vertices = t→next_terminal;
        free(t);
        ts→id = i;
        ts→num_vertices = n;
        ts→connected = ((n ≡ 1) ? 1 : 0);
      }
    }
```

Este código é usado nos blocos 7 e 11.

19. 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 bloco 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.

```
#define next_edge b.A

⟨ Data structures 17 ⟩ +≡
  typedef struct sf_struct {
    Arc *edges;
    double cost;
    double dual_cost;
  } SteinerForest;
```

20. Implementação de Goemans e Williamson

Quando o grafo dado no problema é denso, a fase de escolha das arestas que farão parte da floresta (linha 5 do algoritmo) pode ficar bastante custosa, tornando-se a parte do algoritmo que consome mais tempo no total. A implementação proposta por Goemans e Williamson é bastante adequada nesta situação. Ela utiliza uma estrutura de dados especial para reduzir o número de arestas examinadas nesta fase do algoritmo, diminuindo assim o gasto de tempo no total.

21. Estruturas de Dados

Neste bloco, iniciaremos a apresentação das estruturas de dados que são utilizadas na implementação. A estrutura de dados que representa uma componente da floresta é apresentada aqui. Ela é a principal responsável pela redução do número de arestas examinadas no passo executado na linha 5 do algoritmo.

Para decidir qual aresta incluir na floresta em uma dada iteração, o programa examina o campo *edges* de cada componente. O campo *edges* da estrutura *Component* guarda as arestas do grafo que ligam a componente às demais componentes da floresta corrente. Na verdade, nem todas essas arestas são guardadas, mas apenas uma aresta para cada componente da floresta: dada uma componente L, a aresta guardada é uma das que apresentam a menor folga dentre todas as arestas para L.

As arestas em *edges* estão divididas em dois grupos. As arestas para componentes inativas da floresta se encontram em *edges*[0] e as arestas para componentes ativas estão em *edges*[1]. Cada um desses grupos está organizado na forma de um *heap* de mínimo, onde a chave é dada pela folga das arestas. Desta forma, fica fácil saber qual aresta apresenta a menor folga dentro de cada grupo: basta examinar a primeira posição do heap correspondente. Assim, para decidir qual aresta incluir na floresta em cada iteração, o programa examina apenas duas arestas por componente.

Os vértices que fazem parte da componente se encontram organizados em uma lista ligada apontada pelo campo *vertices*. Dado um vértice v, o vértice que sucede v na lista é dado por $v\to$*next_vertex*. O número de vértices nesta lista (que é igual ao número de vértices na componente) é guardado no campo *num_vertices* da estrutura.

O campo *term_sets_size* é um vetor indexado pelos identificadores dos conjuntos de terminais do grafo. O valor guardado em *term_sets_size*[i] é igual ao número de vértices da componente que fazem parte do conjunto de terminais cujo o *id* é i. O número de conjuntos de terminais que têm intersecção não-vazia com a componente mas não estão propriamente contidos nela é guardado no campo *unconnected*. Note que se este número for maior que zero então a componente estará ativa e, neste caso, o valor do campo *active* deverá ser igual a 1; caso contrário, a componente estará inativa e, assim, *active* deverá ser igual a 0.

Durante toda a execução da fase iterativa do algoritmo, iremos manter as componentes da floresta corrente em uma lista duplamente ligada. Para cada componente L nesta lista, $L\to$*prev* e $L\to$*next* serão, respectivamente, apontadores para a componente que antecede e para a componente que precede L na lista.

Por fim, cada componente possui ainda um identificador *id*.

#define *next_vertex* *w.V*

⟨ Data structures 17 ⟩ +≡
```
typedef struct heap_struct Heap;
typedef struct component_struct {
  int id;
  int active;
  Vertex *vertices;
```

```
    long num_vertices;
    Heap *edges[2];
    int *term_sets_size;
    int unconnected;
    struct component_struct *prev;
    struct component_struct *next;
  } Component;
```

22. Cada um dos *heaps* de arestas das componentes da floresta é implementado através da estrutura *Heap*, descrita abaixo. O campo *heap* desta estrutura é o vetor de arestas que implementa o heap propriamente dito e N é o número de arestas contidas neste vetor. A estrurura também contém um campo *position*, o qual é um vetor indexado pelo identificador das componentes da floresta. Para cada i, o valor guardado em *position*$[i]$ indica a posição do *heap* na qual se encontra a aresta para a componente que tem *id* igual a i; caso não existam arestas para a componente i, *position*$[i]$ será igual a -1.

⟨ Data structures 17 ⟩ +≡
```
  struct heap_struct {
    Arc **heap;
    int N;
    int *position;
  };
```

23. Como já foi dito, a chave de uma aresta a do *heap*, dada por *slackness*(a), é definida como sendo a folga da aresta. Sendo assim, a propriedade do heap pode ser enunciada, para cada estrutura H e para todo i entre 2 e $H \to N$, da seguinte forma:

$$slackness(H \to heap[i/2]) \leq slackness(H \to heap[i])$$

Na definição de *slackness*( ) abaixo, $d(v)$ indica o valor da função d (definida no bloco 5) no vértice v. Note que esta definição não corresponde exatamente ao valor da folga para as arestas que possuem ambos os extremos em componentes ativas da floresta (se a é uma aresta deste tipo, *slackness*(a) é igual a duas vezes o valor da folga de a). A definição foi feita desta forma por motivos de conveniência. Note ainda que podemos definir a chave das arestas desta maneira devido ao modo como as arestas com extremo em uma componente estão organizadas (um heap para as arestas que vão para componentes ativas e outro para as arestas que vão para componentes inativas).

```
#define d(V)  (*((double *) V->x.S))
#define slackness(A)  (((double) A->len) - d(A->origin) - d(A->tip))
```

24. A maioria das funções que manipulam a estrutura *Heap*, descrita acima, utilizam as funções *fixUp* e *fixDown* que serão apresentadas nesta e na próxima seção. Basicamente, o que estas funções fazem é reorganizar a ordem dos elementos no heap após alguma alteração local que, possivelmente, tenha feito com que a propriedadade do heap não fosse mais satisfeita.

A função *fixUp* recebe como argumento a posição *pos* de um elemento do heap cuja chave é possivelmente menor que a chave do elemento na posição *pos*/2; a função realiza o trabalho de deslocar este elemento para alguma posição do heap de forma que a propriedade do heap seja satisfeita. A função *fixDown* realiza o mesmo trabalho, mas com a seguinte diferença: a chave do elemento cuja posição i é recebida como argumento pela função é posssivelmente maior que a chave de algum dos elementos nas posições $2 * i$ e $2 * i + 1$ do heap (note que, dependendo do valor de i, tais posições podem não fazer parte do heap).

Como veremos, daqui para frente, muitas vezes será necessário determinar a componente à qual um dado vértice pertence. Por este motivo, cada vértice guarda um apontador para a componente da qual faz parte, o qual é armazenado no campo *cmpt* do vértice.

#define *cmpt* *y.S*

⟨ Auxiliary functions 24 ⟩ ≡
```
void fixUp(Heap *H, int pos)
{
  Arc *t = H->heap[pos];
  while (pos > 1 ∧ slackness(t) < slackness(H->heap[pos/2])) {
    Arc *a = H->heap[pos/2];
    H->heap[pos] = a;
    H->position[((Component *) a->tip->cmpt)->id] = pos;
    pos /= 2;
  }
  H->heap[pos] = t;
  H->position[((Component *) t->tip->cmpt)->id] = pos;
}
```
Veja também blocos 25, 26, 27, 28, 29, 30, 31, 32, 54, 55, 56, 57, 60, 61, 63, 64, 65, 66, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 91, 99, 122, 124, 127 e 130.

Este código é usado nos blocos 6 e 116.

25.⟨ Auxiliary functions 24 ⟩ +≡
```
void fixDown(Heap *h, long i)
{
  long comp_id;
  Arc *a = h->heap[i];
  while (2 * i ≤ h->N) {
    long j = 2 * i;
    if (j + 1 ≤ h->N ∧ slackness(h->heap[j + 1]) < slackness(h->heap[j])) j++;
    if (slackness(a) ≤ slackness(h->heap[j])) break;
    h->heap[i] = h->heap[j];
    comp_id = ((Component *) h->heap[j]->tip->cmpt)->id;
    h->position[comp_id] = i;
    i = j;
  }
  h->heap[i] = a;
  comp_id = ((Component *) a->tip->cmpt)->id;
  h->position[comp_id] = i;
}
```

26. Aqui, será apresentada a função responsável pela criação de uma estrutura do tipo *Heap*. Ela recebe como argumentos um vetor de arestas $a[0..num-1]$ e um inteiro *capacity*, e devolve um apontador para uma estrutura do tipo *Heap* cujo campo *heap* é um heap construído com as arestas do vetor a e com capacidade para comportar no máximo *capacity* arestas.

$\langle$ Auxiliary functions 24 $\rangle$ $+\equiv$

Heap $*newHeap$(**Arc** $**a$, **int** num, **long** $capacity$)
{
  int i;
  Heap $*h =$ (**Heap** $*$) $malloc$(**sizeof**(**Heap**));
  $h\rightarrow N = num$;
  $h\rightarrow heap =$ (**Arc** $**$) $malloc(capacity *$ **sizeof**(**Arc** $*$));
  $h\rightarrow position =$ (**int** $*$) $malloc(capacity *$ **sizeof**(**int**));
  for $(i = 0;\ i < capacity;\ i{+}{+})\ h\rightarrow position[i] = -1$;
  for $(i = 0;\ i < num;\ i{+}{+})$ {
    Arc $*t = a[i]$;
    $h\rightarrow heap[i] = t$;
    $h\rightarrow position$[((**Component** $*$) $t\rightarrow tip\rightarrow cmpt)\rightarrow id] = i + 1$;
  }
  $(h\rightarrow heap){-}{-}$;
  for $(i = num/2;\ i > 0;\ i{-}{-})\ fixDown(h, i)$;
  return h;
}

27. A função *freeHeap*, apresentada neste bloco, libera a memória alocada para uma estrutura do tipo **Heap**. Para tanto, ela recebe como argumento um apontador H para uma tal estrutura.

$\langle$ Auxiliary functions 24 $\rangle$ $+\equiv$

void $freeHeap$(**Heap** $*H$)
{
  $free(H\rightarrow heap)$;
  $free(H\rightarrow position)$;
  $free(H)$;
}

28. A função abaixo devolve a aresta que posssui a menor folga dentre todas as arestas do heap H.

$\langle$ Auxiliary functions 24 $\rangle$ $+\equiv$

Arc $*findMin$(**Heap** $*H$)
{
  return $((H\rightarrow N < 1)\ ?\ \Lambda : H\rightarrow heap[1])$;
}

29. Agora vamos apresentar a função que remove um dado elemento do heap. Ela recebe como argumentos um heap H e um inteiro pos entre 1 e $H\rightarrow N$, e remove a aresta que se encontra na posição pos de $H\rightarrow heap$.

$\langle$ Auxiliary functions 24 $\rangle$ $+\equiv$

void delete (**Heap** $*H$, **int** pos)
{
  Arc $*a$;

```
    if (pos ≡ H→N) {
      (H→N)−−;
      return;
    }
    a = H→heap[H→N];
    H→heap[pos] = a;
    H→position[((Component *) a→tip→cmpt)→id] = pos;
    (H→N)−−;
    if (pos > 1 ∧ slackness(H→heap[pos]) < slackness(H→heap[pos/2])) fixUp(H, pos);
    else fixDown(H, pos);
  }
```

30. Para remover do heap H a aresta que apresenta a menor folga basta chamar a função abaixo.

⟨ Auxiliary functions 24 ⟩ +≡

```
  void delMin(Heap *H)
  {
    delete (H, 1);
  }
```

31. A inserção de uma aresta no heap é feita pela função a seguir. Ela recebe como argumentos o heap H e aresta a a ser inserida em H.

⟨ Auxiliary functions 24 ⟩ +≡

```
  void insert(Heap *H, Arc *a)
  {
    (H→N)++;
    H→heap[H→N] = a;
    H→position[((Component *) a→tip→cmpt)→id] = H→N;
    fixUp(H, H→N);
  }
```

32. Por fim, apresentamos a função responsável por diminuir a chave de uma das arestas do heap. Ela recebe como argumentos o heap H, um inteiro *pos* entre 1 e $H \to N$ e uma aresta a tal que $slackness(a) < slackness(H \to heap[pos])$, e substitui a aresta na posição *pos* do *heap* pela aresta a.

⟨ Auxiliary functions 24 ⟩ +≡

```
  void decreaseKey(Heap *H, int pos, Arc *a)
  {
    H→heap[pos] = a;
    fixUp(H, pos);
  }
```

33. Função principal

Vamos examinar, a partir de agora, a função que implementa a estratégia de Goemans e Williamson para a construção de uma floresta de Steiner. Abaixo, é apresentada uma visão panorâmica desta função.

```
⟨ Steiner forest construction functions 33 ⟩ ≡
  SteinerForest *sfgw(Graph *g)
  {
    ⟨ Local variables of sfgw 35 ⟩
    ⟨ Create a spanning forest with no edges 34 ⟩
    ⟨ Initialize d(v) for each vertex v of the graph 37 ⟩
    ⟨ While there is some active component include an edge in the forest 38 ⟩
    ⟨ Drop out unnecessary edges and return the resulting Steiner forest 46 ⟩
  }
```

Veja também bloco 80.

Este código é usado no bloco 6.

34. Este bloco implementa a criação da floresta geradora inicial, a qual não possui arestas (passo correspondente a linha 1 do algoritmo). Em cada iteração, a floresta corrente é representada através de uma lista ligada de componentes, sendo que, na primeira iteração, cada uma delas contém apenas um único vértice. A variável *components* é um apontador para o primeiro nó desta lista. Como já vimos, para percorrer tal lista devemos utilizar o campo *next* de cada componente.

```
⟨ Create a spanning forest with no edges 34 ⟩ ≡
  {
    int i;
    Component *t;
    t = components = (Component *) malloc(sizeof(Component));
    for (i = 0; i < g->n; i++) {
      int j;
      Vertex *v;
      TermSet *ts;
      Component *L = (Component *) malloc(sizeof(Component));
      L->id = i;
      v = g->vertices + i;
      L->vertices = v;
      v->next_vertex = Λ;
      ((Component *) v->cmpt) = L;
      L->num_vertices = 1;
      L->term_sets_size = (int *) malloc((g->num_ts) * sizeof(int));
      for (j = 0; j < g->num_ts; j++) L->term_sets_size[j] = 0;
      L->active = 0;
      L->unconnected = 0;
      ts = (TermSet *) v->termset;
      if (ts ∧ ¬(ts->connected)) {
        L->active = 1;
        num_actives++;
        L->term_sets_size[ts->id]++;
        (L->unconnected)++;
      }
      t->next = L;
      L->prev = t;
      t = t->next;
    }
```

```
t->next = components;
components->prev = t;
⟨ Construct a pair of edge heaps for each component 36 ⟩
}
```
Este código é usado no bloco 33.

35. Vamos declarar neste bloco duas das variáveis locais de *sfgw*: *components* e *num_actives*. A primeira já foi apresentada no bloco anterior; vamos agora apresentar a segunda. A variável *num_actives* guarda o número de componentes ativas na floresta corrente. Ela é utilizada, como veremos mais adiante, para determinar quando não é mais necessário acrescentar arestas na floresta corrente.

```
⟨ Local variables of sfgw 35 ⟩ ≡
  long num_actives = 0;
  Component *components;
```
Veja também bloco 39.

Este código é usado no bloco 33.

36. O trecho de código abaixo realiza a construção dos dois *heaps* de arestas de cada componente da floresta. Para cada componente, é feito o seguinte: as arestas incidentes no único vértice que faz parte da componente são divididas em arestas para componentes ativas e arestas para componentes inativas; por fim, é criado um heap de mínimo para cada grupo.

```
⟨ Construct a pair of edge heaps for each component 36 ⟩ ≡
  {
    for (t = components->next; t ≠ components; t = t->next) {
      Vertex *v = t->vertices;
      Arc **active_links = (Arc **) malloc(g->n * sizeof(Arc *));
      Arc **inactive_links = (Arc **) malloc(g->n * sizeof(Arc *));
      Arc *a;
      int n = 0, m = 0;
      for (a = v->arcs; a; a = a->next) {
        Component *L = (Component *) a->tip->cmpt;
        if (L->active) active_links[n++] = a;
        else inactive_links[m++] = a;
      }
      t->edges[0] = newHeap(inactive_links, m, g->n);
      t->edges[1] = newHeap(active_links, n, g->n);
      free(active_links);
      free(inactive_links);
    }
  }
```
Este código é usado no bloco 34.

37. Aqui iremos fazer com que $d(v)$ seja zero para todo vértice v do grafo. Como $d(v)$ pode assumir valores que não são inteiros, não podemos utizar os *utility fields* de v para guardar o valor de $d(v)$. Assim, teremos de alocar espaço extra para isto.

⟨ Initialize $d(v)$ for each vertex v of the graph 37 ⟩ ≡

```
{
  Vertex *v;
  double *a = (double *) malloc(g->n * sizeof(double));
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    int i = v - g->vertices;
    a[i] = 0;
    &(d(v)) = a + i;
  }
}
```

Este código é usado no bloco 33.

38. Este bloco 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). A variável *sf* é um apontador para a estrutura que representa esta floresta.

Em cada iteração, uma aresta externa é incluída na floresta 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, além disso, é realizada a união das componentes `L1` e `L2` às quais pertencem, respectivamente, os vértices *a→origin* e *a→tip*, extremos de a. Note que, após ser incluída na floresta, a é removida dos *heaps* de arestas de `L1` e `L2`, já que deixou de ser uma aresta externa.

⟨ While there is some active component include an edge in the forest 38 ⟩ ≡

```
{
  Arc *t;
  sf = (SteinerForest *) malloc(sizeof(SteinerForest));
  sf->cost = sf->dual_cost = 0;
  t = sf->edges = (Arc *) malloc(sizeof(Arc));
  while (num_actives > 0) {
    double inc;
    Arc *a;
    Component *L1, *L2;
    ⟨ Set a to point to an edge with the smallest slackness 40 ⟩
    t->next_edge = a;
    t = t->next_edge;
    sf->cost += a->len;
    L1 = (Component *) a->origin->cmpt;
    L2 = (Component *) a->tip->cmpt;
    if (L1->active ∧ L2->active) {
      delMin(L1->edges[1]);
      delMin(L2->edges[1]);
      inc = slackness(a)/2;
      sf->dual_cost += inc * num_actives;
      num_actives--;
    }
    else {
      if (L1->active) {
        delMin(L1->edges[0]);
        delMin(L2->edges[1]);
```

```
      }
      else {
        delMin(L1->edges[1]);
        delMin(L2->edges[0]);
      }
      inc = slackness(a);
      sf->dual_cost += inc * num_actives;
    }
    ⟨ Increment d(v) for each vertex v in some active component 41 ⟩
    ⟨ Merge the components L1 and L2 42 ⟩
  }
  t->next_edge = Λ;
  t = sf->edges;
  sf->edges = sf->edges->next_edge;
  free(t);
}
```

Este código é usado no bloco 33.

39.⟨ Local variables of *sfgw* 35 ⟩ +≡

```
SteinerForest *sf;
```

40. Aqui, é feita a escolha da aresta a a ser incluída na floresta corrente (passo correspondente a linha 5 do algoritmo). Para encontrar uma aresta que apresenta a menor folga, são examinadas apenas duas arestas por componente ativa.

⟨ Set a to point to an edge with the smallest slackness 40 ⟩ ≡

```
{
  double min;
  Arc *b;
  Component *L;

  a = Λ;
  for (L = components->next; L ≠ components; L = L->next)
    if (L->active) {
      b = findMin(L->edges[0]);
      if (¬b) {
        a = findMin(L->edges[1]);
        min = slackness(a)/2;
      }
      else {
        a = findMin(L->edges[1]);
        if (a ∧ (slackness(a)/2 < slackness(b))) min = slackness(a)/2;
        else {
          a = b;
          min = slackness(b);
        }
      }
      L = L->next;
      break;
```

```
    }
  for ( ; L ≠ components; L = L→next) {
    if (L→active) {
      b = findMin(L→edges[0]);
      if (b ∧ (slackness(b) < min)) {
        min = slackness(b);
        a = b;
      }
      b = findMin(L→edges[1]);
      if (b ∧ (slackness(b)/2 < min)) {
        min = slackness(b)/2;
        a = b;
      }
    }
  }
}
```

Este código é usado no bloco 38.

41. O seguinte trecho de código atualiza o valor de $d(v)$ para cada vértice v em uma componente ativa, após a escolha da aresta a ser incluída na floresta (passo que corresponde a linha 6 do algoritmo). A variável *inc* guarda o valor do incremento a ser feito em $d(v)$.

⟨ Increment $d(v)$ for each vertex v in some active component 41 ⟩ ≡

```
{
  Component *L;
  for (L = components→next; L ≠ components; L = L→next)
    if (L→active) {
      Vertex *v;
      for (v = L→vertices; v; v = v→next_vertex) d(v) += inc;
    }
}
```

Este código é usado no bloco 38.

42. Como já vimos, cada vez que uma aresta é incluída na floresta, ocorre a união das duas componentes das quais fazem parte os extremos da aresta. Isto é feito em dois estágios: inicialmente, faz-se a união dos conjuntos de vértices das duas componentes e, a seguir, é feita a atualização dos *heaps* de arestas.

Aqui, `L1` e `L2` são apontadores para as estruturas das duas componentes a serem unidas. Após a união, `L1` estará apontando para a estrutura da componente resultante.

⟨ Merge the components `L1` and `L2` 42 ⟩ ≡

```
{
  int active_ = 1;
  if (L1→num_vertices < L2→num_vertices) {
    Component *t = L1;
    L1 = L2;
    L2 = t;
  }
```

```
⟨ Insert each vertex of L2 in L1 and set active_ to the new state of L1 43 ⟩
⟨ Alter the place of the edges to L1 if the state of L1 has changed 44 ⟩
⟨ Merge the edge heaps of L1 and L2 45 ⟩
L1→active = active_;
L2→next→prev = L2→prev;
L2→prev→next = L2→next;
freeHeap(L2→edges[0]);
freeHeap(L2→edges[1]);
free(L2→term_sets_size);
free(L2);
}
```

Este código é usado no bloco 38.

43. A união dos conjuntos de vértices das duas componentes é feita incluíndo cada vértice de L2 na lista dos vértices de L1. A cada vértice incluído, verificamos se ocorreu alguma alteração no estado da componente. Ao final deste trecho de código, a variável *active_* estará guardando o estado da componente resultante da união (se *active_* $\equiv 1$ a componente estará ativa e, caso contrário, estará inativa).

```
⟨ Insert each vertex of L2 in L1 and set active_ to the new state of L1 43 ⟩ ≡
{
  Vertex *v = L2→vertices;
  while (v) {
    TermSet *ts;
    Vertex *z = v→next_vertex;
    v→next_vertex = L1→vertices;
    L1→vertices = v;
    L1→num_vertices++;
    ((Component *) v→cmpt) = L1;
    ts = (TermSet *) v→termset;
    if (ts ∧ ¬(ts→connected)) {
      L1→term_sets_size[ts→id]++;
      if (L1→term_sets_size[ts→id] ≡ 1) {
        (L1→unconnected)++;
        if (L1→unconnected ≡ 1) active_ = 1;
      }
      else if (L1→term_sets_size[ts→id] ≡ ts→num_vertices) {
        ts→connected = 1;
        (L1→unconnected)−−;
        if (¬(L1→unconnected)) active_ = 0;
      }
    }
    v = z;
  }
  if (¬active_) num_actives−−;
}
```

Este código é usado no bloco 42.

44. A organização dos *heaps* de arestas da componente resultante da união de `L1` e `L2` divide-se em duas partes: a troca de heap, se necessária, para as arestas que ligam `L1` às componentes da floresta que não são *adjacentes* a `L2` no grafo e a união dos *heaps* de arestas de `L1` e `L2`. Iremos tratar da primeira parte aqui e da segunda parte no próximo bloco.

Dizemos que duas componentes da floresta são *adjacentes* se existe, no grafo, uma aresta externa com extremos em cada uma delas. Se o estado da componente `L1` foi alterado após a inclusão dos vértices de `L2` em seu conjunto de vértices, será necessário trocar de heap as arestas para `L1` que se encontram nos *heaps* de cada componente L não adjacente a `L2` (também será necessário fazer isto para as arestas nos *heaps* de componentes adjacentes a `L2`, mas este caso será tratado no próximo bloco). E é exatamente isto que é feito pelo trecho de código abaixo.

⟨ Alter the place of the edges to `L1` if the state of `L1` has changed 44 ⟩ ≡

```
{
  if (L1→active ≠ active_) {
    int j;
    for (j = 0; j < 2; j++) {
      int k;
      Heap *h1 = L1→edges[j];
      for (k = 1; k ≤ h1→N; k++) {
        Arc *a = h1→heap[k];
        Component *L = (Component *) a→tip→cmpt;
        Heap *h2 = L2→edges[L→active];
        if (h2→position[L→id] ≡ −1) {      /* L isn't adjacent to L2 */
          Heap *h = L→edges[L1→active];
          int pos = h→position[L1→id];
          a = h→heap[pos];
          delete (h, pos);
          insert(L→edges[active_], a);
        }
      }
    }
  }
}
```

Este código é usado no bloco 42.

45. A união dos *heaps* das componentes `L1` e `L2` é feita da seguinte forma. Cada um dos *heaps* de arestas de `L2` é percorrido e, para cada aresta a, verifica-se se a componente L a qual a liga `L2` é adjacente a `L1`. Se L não for adjacente a `L1`, a é incluída em um dos *heaps* de `L1` (de acordo com o estado de L); caso contrário, já existe uma aresta b ligando `L1` a L e, desta forma, devemos decidir qual aresta, a ou b, guardar em um dos heaps de `L1`: se $slackness(a) < slackness(b)$ então b é substituída por a e, caso contrário, b é mantida no heap. Em cada um dos casos, são feitas as atualizações necessárias nos *heaps* da componente L.

⟨ Merge the edge heaps of `L1` and `L2` 45 ⟩ ≡

```
{
  int j;
  for (j = 0; j < 2; j++) {
    int k;
    Heap *h1 = L1→edges[j];
    Heap *h2 = L2→edges[j];
```

```
for (k = 1; k ≤ h2→N; k++) {
  Arc *a = h2→heap[k];
  Component *L = (Component *) a→tip→cmpt;
  int pos = h1→position[L→id];
  Heap *h;
  if (pos ≡ −1) {    /* L isn't adjacent to L1, so we need to include a in h1 */
    insert(h1, a);
    if (L2→active ≡ active_) {
      h = L→edges[active_];
      h→position[L1→id] = h→position[L2→id];
    }
    else {
      h = L→edges[L2→active];
      pos = h→position[L2→id];
      a = h→heap[pos];
      delete (h, pos);
      insert(L→edges[active_], a);
    }
  }
  else {
    Arc *b = h1→heap[pos];    /* b is the best edge linking L1 to L */
    if (slackness(a) < slackness(b)) {    /* let's substitute b for a */
      decreaseKey(h1, pos, a);
      h = L→edges[L1→active];
      pos = h→position[L1→id];
      delete (h, pos);
      if (L2→active ≡ active_) {
        h = L→edges[active_];
        h→position[L1→id] = h→position[L2→id];
      }
      else {
        h = L→edges[L2→active];
        pos = h→position[L2→id];
        a = h→heap[pos];
        delete (h, pos);
        insert(L→edges[active_], a);
      }
    }
    else {    /* let's keep edge b in the heap */
      h = L→edges[L2→active];
      pos = h→position[L2→id];
      delete (h, pos);
      if (L1→active ≠ active_) {
        h = L→edges[L1→active];
        pos = h→position[L1→id];
        a = h→heap[pos];
        delete (h, pos);
        insert(L→edges[active_], a);
      }
```

```
            }
        }
        h = L→edges[L2→active];
        h→position[L2→id] = −1;
      }
    }
  }
```

Este código é usado no bloco 42.

46. Ao final do processo iterativo descrito nos blocos anteriores, 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 subfloresta de Steiner minimal contida na floresta apontada por esta variável. A floresta encontrada é então devolvida como resposta pela função *sfgw*. A operação implementada por *edgePrunning* é conhecida como *limpeza das arestas* e será vista com detalhes mais adiante.

⟨ Drop out unnecessary edges and return the resulting Steiner forest 46 ⟩ ≡

```
  return edgePrunning(sf, g);
```

Este código é usado nos blocos 33 e 80.

47. 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 acima. Um exemplo é a implementação proposta por Cole, Hariharan, Lewenstein e Porat em [3], a qual passaremos a descrever agora.

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 (note que, desta forma, cada componente precisa manter apenas um heap de arestas) . Para isto, cada aresta do grafo é particionada dinamicamente ("dynamic edge spliting") através da inclusão de um ou mais vértices no grafo. A idéia aqui é a seguinte: sempre que uma aresta tem ambos os extremos em componentes ativas da floresta, divide-se esta aresta em duas arestas de igual comprimento através da inclusão de um novo vértice no grafo, o qual será considerado como uma nova componente inativa da floresta. Denominaremos um tal vértice de *s-vértice*. Durante toda execução da fase iterativa do algoritmo, uma aresta (do grafo original) é particionada no total $O(k \log n)$ vezes, onde k é uma dada constante fornecida como parâmetro. Se após isso alguma das arestas geradas através de uma dessas divisões da aresta original tiver ambos os extremos em componentes ativas, tal aresta é tratada 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.

48. 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 chamada *aresta terminal* (mais adiante, ficará clara a motivação por trás desta terminologia), é tal que ou u' ou v' faz parte de uma componente inativa da floresta e, além isto, tal componente consiste de um único s-vértice. Assim, toda aresta externa $u'v'$ que não é terminal, é tal que

$$folga(u'v') = \min\{c_{u'v'} - d(u'), c_{u'v'} - d(v')\}$$

pois $d(w) = 0$ para todo s-vértice w que faz parte de uma componente com um único vértice. 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, serão examinadas as arestas que se encontram na primeira posição de cada heap associado a uma componente ativa e, dentre elas, será selecionada aquela que possuir a menor chave.

Vamos apresentar agora uma descrição não muito detalhada do algoritmo correspondente a esta implementação. Inicialmente, cada aresta do grafo é subdividida em duas arestas de igual comprimento através da inclusão de um novo vértice no grafo, sendo que cada vértice incluído corresponde a uma nova componente inativa da floresta (lembre-se que cada vértice em um conjunto da coleção 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, é criado o heap associado a cada componente.

Em cada uma das iterações seguintes, o algoritmo executa 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 $\lceil \frac{c_{vw}}{2} \rceil$, $\lceil \frac{c_{vw}}{4} \rceil$, $\lceil \frac{c_{vw}}{8} \rceil$, ..., $\lceil \frac{c_{vw}}{2^i} \rceil$ de v, onde i é o menor índice tal que uma das seguintes condições é satisfeita:

   - $c_{vw} - \lceil \frac{c_{vw}}{2^i} \rceil > d(w)$.
   - $\lceil \frac{c_{vw}}{2^i} \rceil \leq \frac{c_{v'w'}}{n^k}$, 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, todas as arestas geradas como descrito acima, com excessão da última aresta (cujo comprimento 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 às quais nos referimos anteriormente).

Agora, suponha que i seja tal que a primeira condição é satisfeita e seja z o vértice adicionado que se encontra a distância $\lceil \frac{c_{vw}}{2^i} \rceil$ 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.

49. 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*, *unconnected*, *prev* e *next* são como aqueles que se encontram na estrutura **Component** descrita no bloco 21.

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 reprsentar 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 realzada 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 $O(n)$, 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 a 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.

⟨ Data structures 17 ⟩ +≡

```
typedef Arc *FibHeap;
typedef struct es_component_struct {
  int active;
  int d1;
  int *d2;
  Vertex *vertices;
  long num_vertices;
  FibHeap edges;
  int pos;
  int *term_sets_size;
  int unconnected;
  struct es_component_struct *prev;
  struct es_component_struct *next;
} ES_Component;
```

50. 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 na estrutura.

Vimos no bloco anterior que cada aresta uv do grafo estará associada a uma parcela do valor de $d(u)$ e a 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 49), 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 aresta terminal é gerada, representaremos cada uma das arestas geradas através dos arcos correspondentes a aresta original. Para isto, cada arco passa a ter um campo *current_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 *current_len* de cada um deles corresponderá ao custo dessa aresta terminal. 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 *current_len* que descrevemos acima. Para cada arco a, *fields*(a) será um apontador para uma estrutura deste tipo.

```
#define fields(A) ((EdgeFields *)(A→a.S))
```

⟨ Data structures 17 ⟩ +≡

```
typedef struct edge_fields_struct {
  int current_len;
  int d;
} EdgeFields;
```

51. 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 de um arco.

```
#define slackness1(A) (fields(A)->current_len - fields(A)->d - A->origin->cmpt->d1)
#define slackness2(A) (slackness1(A) - *(A->origin->cmpt->d2))
#define is_active(V) (V->cmpt->active)
#define key(A) (is_active(A->origin) ? slackness2(A) : slackness1(A))
```

52. Um *Fibonacci heap* de arestas é um conjunto de arborescências que respeitam as seguintes propriedades:

1. *ordem*: para cada par (p, f), em uma mesma arborescência, onde p é o pai da aresta f, $key(p) \leq key(f)$
2. *descendentes*: para cada aresta p, os filhos de p podem ser ordenados de maneira que o i-ésimo filho tenha pelo menos $i - 2$ filhos.

Dizemos que o *grau* de uma aresta é o seu número de filhos no heap. Como conseqüência da propriedade 2 pode-se notar que toda aresta de grau k possui pelo menos F_{k+2} descendentes (incluindo ela), onde F_k é o número de Fibonacci,

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

ou

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

sendo esta a motivação do nome Fibonacci heap.

A implementação de um Fibonacci heap é feita através do uso de listas duplamente ligadas, como veremos a seguir.

53. A listas que implementam um Fibonacci heap de arestas podem ser divididas em dois tipos: (1) (*root*) uma lista formada pelas raízes das arborescências; e (2) (*child*) lista formada por filhos de uma dada aresta.

A cada aresta num fibonacci heap associaremos cinco campos: ponteiros *parent*, *child*, *right*, *left*, que representam as listas duplamente ligadas circulares (lista *root* e *child*), e um inteiro *grau_ marca*, que guarda o número $2 * grau + marca$, onde o *grau* é o grau da aresta e *marca* tem valor 1 se a aresta não se encontra na lista *root* e já perdeu algum filho, e 0 caso contrário.

A estrutura **Node** que apresentamos abaixo armazena os campos associados a uma aresta em um Fibonacci heap. Para cada aresta a em um Fibonacci heap, *node*(a) será um apontador para a estrutura **Node** correspondente a a.

```
#define node(A)  ((Node *)(A→b.S))
```

⟨ Data structures 17 ⟩ +≡

```
typedef struct node_struct {
   Arc *parent;
   Arc *child;
   Arc *right;
   Arc *left;
   int grau_marca;
} Node;
```

54. As funções a seguir são responsáveis por inserir e remover arestas em uma lista duplamente ligada, e concatenar duas listas de arestas duplamente ligadas. A função *inserList* insere na lista a a aresta b e a função *deleteList* remove a aresta b de uma lista. Por fim, a função *concatenateList* concatena as listas de arestas às quais pertencem as arestas b e c.

⟨ Auxiliary functions 24 ⟩ +≡

```
void insertList(Arc *a, Arc *b)
{
   Arc *c = node(a)→right;
   node(b)→right = c;
   node(c)→left = b;
   node(b)→left = a;
   node(a)→right = b;
}
void deleteList(Arc *b)
{
   Arc *a = node(b)→left;
   Arc *c = node(b)→right;
   node(a)→right = c;
   node(c)→left = a;
}
void cocatenateList(Arc *b, Arc *c)
{
   Arc *a = node(b)→right;
   Arc *d = node(c)→left;
   node(b)→right = c;
   node(c)→left = b;
   node(a)→left = d;
```

```
    node(d)→right = a;
  }
```

55. Inserir uma aresta no Fibonacci heap significa colocá-la na lista *root*. A função abaixo insere a aresta a no Fibonacci heap H e devolve o Fibonacci heap resultante desta operação.

⟨ Auxiliary functions 24 ⟩ +≡

```
  FibHeap insertFibHeap(FibHeap H, Arc *a)
  {
    if (¬H) return a;
    cocatenateList(H, a);
    if (key(a) < key(H)) return a;
    else return H;
  }
```

56. Para criar o Fibonacci heap é utilizada a função *newFibHeap*. Ela recebe um vetor de arestas a contendo n elementos e devolve um Fibonacci heap contendo as arestas em a.

⟨ Auxiliary functions 24 ⟩ +≡

```
  FibHeap newFibHeap(Arc **a, int n)
  {
    H = Λ;
    Node *v = (Node *) malloc(n * sizeof(Node));
    int i;
    for (i = 0; i < n; i++) {
      node(a[i]) = (char *)(v + i);
      v[i].right = v[i].left = a[i];
      v[i].grau_marca = 0;
      v[i].parent = v[i].child = Λ;
      H = insertFibHeap(H, a[i]);
    }
    return H;
  }
```

57. A aresta com a menor chave no Fibonacci heap se encontra na primeira posição, ou seja, H aponta para a aresta com a menor chave. Remover essa aresta implica em adicionar seus filhos à lista *root*, e em seguida, é necessário utilizar o bloco ⟨ Consolidate 58 ⟩, para atualizar o número de arborescências e encontrar a nova aresta com chave mínima. Ao final de sua execução, a função devolve o Fibonacci heap resultante.

⟨ Auxiliary functions 24 ⟩ +≡

```
  FibHeap delMinFibHeap(FibHeap H)
  {
    Arc *min, *f, *v;
    min = H;
    if (min) {
      f = node(min)→child;
      while (f) {
```

```
        v = f;      /* for each child v of min */
        f = node(f)→right;
        if (v ≡ f) f = Λ;
        deleteList(v);      /* delete v from the child list */
        insertList(H, v);      /* insert v in the root list */
        node(v)→parent = Λ;
      }
      deleteList(min);      /* delete min from the root list */
      if (node(min)→right ≡ min) H = Λ;
      else {
        H = node(H)→right;
        ⟨ Consolidate 58 ⟩
      }
    }
    return H;
  }
```

58. O bloco ⟨ Consolidate 58 ⟩ é responsável por reduzir o número de raízes de H, ou seja, reduzir o número de arborescências. Ele consiste em executar repetidamente os seguintes passos, até que todas as arestas da lista *root* tenham graus distintos:

1. encontre duas raízes x e y na lista *root* com o mesmo grau. Supõe-se que $key(x) \le key(y)$; e

2. junte y a x, removendo y da lista *root*, e adicionando y como um filho de x.

A operação (2) é feita em ⟨ Insert y as a child of x 59 ⟩

⟨ Consolidate 58 ⟩ ≡

```
  {
    Arc *x, *y, *aux, **A;
    long i, d, Dn;
    Dn = 1 + 8 * sizeof(long);      /* 1+log(n): maximum number of trees */
    A = (Arc **) malloc(Dn * sizeof(Arc *));
    for (i = 0; i < Dn; i++)      /* A[d] keeps all vertices wich have degree d */
      A[i] = Λ;
    v = H;
    do {
      x = v;
      v = node(v)→right;
      if (x ≡ v) v = Λ;
      deleteList(x);      /* delete x from the root list */
      d = (int) node(x)→grau_marca/2;      /* 2 * grau + marca */
      while ((d < Dn) ∧ A[d]) {
        y = A[d];
        if (key(x) > key(y)) {
          aux = x;
          x = y;
          y = aux;
        }
        ⟨ Insert y as a child of x 59 ⟩
```

```
      A[d] = Λ;
      d++;
    }
    A[d] = node(x)→right = node(x)→left = x;
  } while (v);
  H = Λ;
  for (i = 0; i < Dn; i++) {
    if (A[i]) {
      if (¬H) H = A[i];
      else {
        insertList(H, A[i]);      /* insert A[i] in the root list */
        if (key(A[i]) < key(H)) H = A[i];
      }
    }
  }
  free(A);
}
```

Este código é citado nos blocos 57 e 58.

Este código é usado no bloco 57.

59. ⟨ Insert y as a child of x 59 ⟩ ≡

```
deleteList(y);      /* delete y from the root list */
f = node(x)→child;
if (f) insertList(f, y);      /* insert y in the children list of x */
else node(x)→child = y;
node(y)→parent = x;
node(x)→grau_marca += 2;
node(y)→grau_marca = (node(y)→grau_marca ≫ 1) ≪ 1;
```

Este código é citado no bloco 58.

Este código é usado no bloco 58.

60. A função abaixo devolve um elemento que possui a menor chave no Fibonacci heap H.

⟨ Auxiliary functions 24 ⟩ +≡

```
Arc *findMinFibHeap(FibHeap H)
{
  return H;
}
```

61. Diminuir a chave de uma aresta v pode tornar necessário reposicioná-la em H, de modo que as propriedades do Fibonacci heap H continuem sendo respeitadas. Isso é feito chamando-se ⟨ Delete v from the children list of p 62 ⟩, onde p é o pai de v. Caso p esteja perdendo um segundo filho, ainda é necessário chamar a função *cascading_cut* para garantir a segunda propriedade descrita no bloco 52. Ao final de sua execução, a função abaixo retornará o Fibonacci heap resultante dessa operação.

⟨ Auxiliary functions 24 ⟩ +≡

```
void cascading_cut(FibHeap, Arc *);
```

```
FibHeap decreaseKeyFibHeap(FibHeap H, Arc *v)
{
  Arc *p;
  p = node(v)→parent;
  if (p ∧ (key(v) < key(p))) {
    ⟨ Delete v from the children list of p 62 ⟩
    cascading_cut(H, p);
  }
  if (key(v) < key(H)) H = v;
  return H;
}
```

62. ⟨ Delete v from the children list of p 62 ⟩ ≡

```
  if (node(v)→right ≡ v) node(p)→child = Λ;
  else {
    if (node(p)→child ≡ v) node(p)→child = node(v)→right;
    deleteList(v);    /* remove v da lista de filhos de p */
  }
  node(p)→grau_marca −= 2;
  insertList(H, v);    /* adiciona v na lista root */
  node(v)→parent = Λ;
  node(v)→grau_marca = (node(v)→grau_marca ≫ 1) ≪ 1;
```

Este código é citado nos blocos 61 e 63.

Este código é usado nos blocos 61 e 63.

63. A função *cascading_cut* repete o processo de ⟨ Delete v from the children list of p 62 ⟩ até encontrar um aresta que ainda não tenha perdido nenhum filho.

⟨ Auxiliary functions 24 ⟩ +≡

```
void cascading_cut(FibHeap H, Arc *v)
{
  Arc *p;
  p = node(v)→parent;
  if (p) {
    if (node(v)→grau_marca % 2 ≡ 0) node(v)→grau_marca++;
    else {
      ⟨ Delete v from the children list of p 62 ⟩
      cascading_cut(H, p);
    }
  }
}
```

64. A função abaixo remove a aresta a do Fibonacci heap H. Ao final, o Fibonacci heap resultante desta operação é devolvido pela função.

⟨ Auxiliary functions 24 ⟩ +≡
```
FibHeap deleteNodeFibHeap(FibHeap H, Arc *a)
{
  fields(a)->current_cost = -1;
  H = decreaseKeyFibHeap(H, a);
  return delMinFibHeap(H);
}
```

65. A função *mergeFibHeap* recebe dois Fibonacci heaps `H1` e `H2` e devolve o Fibonacci heap resultante da união de `H1` e `H2`.

⟨ Auxiliary functions 24 ⟩ +≡
```
FibHeap *mergeFibHeap(FibHeap H1, FibHeap H2)
{
  if (¬H1) return H2;
  if (¬H2) return H1;
  cocatenateList(H1, H2);
  if (key(H1) < key(H2)) return H1;
  else return H2;
}
```

66. A função *traverseFibHeap* toma como argumentos um Fibonacci heap H, uma função *func* e um inteiro n. Então, para cada aresta a em H, *traverseFibHeap* executa uma chamada a $func(a, n)$.

⟨ Auxiliary functions 24 ⟩ +≡
```
void traverseFibHeap(FibHeap H, void(*func)(Arc *, int), int n)
{
  Arc *a;
  if (H) {
    (*func)(H, n);
    traverseFibHeap(node(H)->child, func, n);
    for (a = node(H)->right; a ≠ H; a = node(a)->right) traverseFibHeap(a, func, n);
  }
}
```

67. Como mencionamos anteriormente iremos manter os heaps de arestas das componentes ativas da floresta como elementos de um heap. A estrutura *BinHeap* apresentada abaixo implementa este heap. O campo heap desta estrutura é o vetor de elementos do tipo **FibHeap** e implementa o heap binário propriamente dito; N é o número de elementos contidos neste vetor.

⟨ Data structures 17 ⟩ +≡
```
typedef struct binary_heap_struct {
  FibHeap *heap;
  int N;
} BinHeap;
```

68. A chave de um Fibonacci heap h no heap, dada por $min(h)$, é definida como sendo a chave de menor valor dentre todas aquelas das arestas que se encontram em h. Sendo assim, a propriedade do heap binário pode ser enunciada, para cada estrutura H e para todo i entre 2 e $H{\to}N$, da seguinte forma:

$$min(H{\to}heap[i/2]) \leq min(H{\to}heap[i])$$

Abaixo apresentamos a definição de $min(\,)$.

```
#define min(A)  key(A)
```

69. A seguir apresentamos uma lista de funções que serão úteis para a manipulação de uma estrutura do tipo **BinHeap**. Todas estas funções são semelhantes às funções definidas a partir do bloco 28 para a manipulação de uma estrutura do tipo **Heap**.

70.⟨ Auxiliary functions 24 ⟩ +≡

```
void fixUpBinHeap(BinHeap *H, int pos)
{
  FibHeap t = H→heap[pos];
  while (pos > 1 ∧ min(t) < min(H→heap[pos/2])) {
    FibHeap a = H→heap[pos/2];
    H→heap[pos] = a;
    ((ES_Component *) a→origin→cmpt)→pos = pos;
    pos /= 2;
  }
  H→heap[pos] = t;
  ((ES_Component *) t→origin→cmpt)→pos = pos;
}
```

71.⟨ Auxiliary functions 24 ⟩ +≡

```
void fixDownBinHeap(BinHeap *h, int i)
{
  long comp_id;
  FibHeap a = h→heap[i];
  while (2 * i ≤ h→N) {
    long j = 2 * i;
    if (j + 1 ≤ h→N ∧ min(h→heap[j + 1]) < min(h→heap[j])) j++;
    if (min(a) ≤ min(h→heap[j])) break;
    h→heap[i] = h→heap[j];
    ((ES_Component *) h→heap[j]→origin→cmpt)→pos = i;
    i = j;
  }
  h→heap[i] = a;
  ((ES_Component *) a→origin→cmpt)→pos = i;
}
```

72. ⟨ Auxiliary functions 24 ⟩ +≡

```
BinHeap *newBinHeap(FibHeap *a, int num, long capacity)
{
  int i;
  BinaryHeap *h = (BinHeap *) malloc(sizeof (BinaryHeap));
  h->N = num;
  h->heap = (FibHeap *) malloc(capacity * sizeof(FibHeap));
  for (i = 0; i < num; i++) {
    FibHeap t = a[i];
    h->heap[i] = t;
    ((ES_Component *) t->origin->cmpt)->pos = i + 1;
  }
  (h->heap)--;
  for (i = num/2; i > 0; i--) fixDown(h, i);
  return h;
}
```

73. ⟨ Auxiliary functions 24 ⟩ +≡

```
void freeBinHeap(BinHeap *H)
{
  free(H->heap);
  free(H);
}
```

74. ⟨ Auxiliary functions 24 ⟩ +≡

```
FibHeap findMinBinHeap(BinHeap *H)
{
  return ((H->N < 1) ? Λ : H->heap[1]);
}
```

75. ⟨ Auxiliary functions 24 ⟩ +≡

```
void deleteBinHeap(BinHeap *H, int pos)
{
  FibHeap a;
  if (pos ≡ H->N) {
    (H->N)--;
    return;
  }
  a = H->heap[H->N];
  H->heap[pos] = a;
  ((ES_Component *) a->origin->cmpt)->pos = pos;
  (H->N)--;
  if (pos > 1 ∧ min(H->heap[pos]) < min(H->heap[pos/2])) fixUp(H, pos);
  else fixDown(H, pos);
}
```

76.⟨ Auxiliary functions 24 ⟩ +≡

void *delMinBinHeap*(**BinHeap** ∗*H*)
{
  deleteBinHeap(*H*, 1);
}

77.⟨ Auxiliary functions 24 ⟩ +≡

void *insertBinHeap*(**BinHeap** ∗*H*, **FibHeap** *a*)
{
  (*H*→*N*)++;
  H→*heap*[*H*→*N*] = *a*;
  ((**ES_Component** ∗) *a*→*origin*→*cmpt*)→*pos* = *H*→*N*;
  fixUp(*H*, *H*→*N*);
}

78.⟨ Auxiliary functions 24 ⟩ +≡

void *decreaseKeyBinHeap*(**BinHeap** ∗*H*, **int** *pos*, **FibHeap** *a*)
{
  H→*heap*[*pos*] = *a*;
  fixUp(*H*, *pos*);
}

79. A função abaixo substitui o elemento na posição *pos* do heap *H*→*heap* por *a*, onde o Fibonacci heap *a* é tal que $min(H\rightarrow heap[pos]) \le min(a)$, e rearranja a posição dos elementos em *H*→*heap* de forma que a propriedade do heap seja satisfeita.

⟨ Auxiliary functions 24 ⟩ +≡

void *increaseKeyBinHeap*(**BinHeap** ∗*H*, **int** *pos*, **FibHeap** *a*)
{
  H→*heap*[*pos*] = *a*;
  fixDown(*H*, *pos*);
}

80. Função principal

A função abaixo é responsável por aplicar ao grafo *g* o método de Cole, Hariharan, Lewenstein e Porat 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*→*n*.

⟨ Steiner forest construction functions 33 ⟩ +≡

SteinerForest ∗*sf_chlp*(**Graph** ∗*g*, **int** *k*)
{
  ⟨ Local variables of *sf_chlp* 83 ⟩
  ⟨ For each edge *a* in *g* split *a* in two pieces 81 ⟩
  ⟨ Create the initial spanning forest 82 ⟩
  ⟨ Include an edge piece in the forest while there is some active component 86 ⟩
  ⟨ Drop out unnecessary edges and return the resulting Steiner forest 46 ⟩
}

81. 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 arcos correspondentes 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 custo da aresta original a.

⟨ For each edge a in g split a in two pieces 81 ⟩ ≡

```
{
  Arc *a;
  Vertex *v;
  EdgeFields *ef = (EdgeFields *) malloc(g→m * sizeof(EdgeFields));
  int i = 0, j;

  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next) {
      fields(a) = ef + i;
      fields(a)→current_len = −1;
      fields(a)→d = 0;
      i++;
    }
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next)
      if (fields(a)→current_len ≡ −1) {
        if (a→len % 2) {
          fields(a)→current_len = (a→len)/2 + 1;
          fields(a + 1)→current_len = (a→len)/2;
        }
        else fields(a)→current_len = (a→len)/2;
      }
}
```

Este código é usado no bloco 80.

82. Aqui são inicializadas as estruturas correspondentes às componentes da floresta geradora inicial (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 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 50, o campo *d2* de cada componente será um apontador para esta variável.

⟨ Create the initial spanning forest 82 ⟩ ≡

```
{
  Vertex *v;
  ES_Component *comp, *t;
  TermSet *ts;
  int i = 0, j;

  num_actives = 0;
  active_d = 0;
  t = (ES_Component *) malloc(sizeof(ES_Component));
  components = t;
  for (v = g→vertices; v < g→vertices + g→n; v++) {
```

$comp = (\mathbf{ES\_Component}\ *)\ malloc(\mathbf{sizeof}(\mathbf{ES\_Component}));$
$comp \rightarrow pos = -1;$
$comp \rightarrow d1 = 0;$
$comp \rightarrow d2 = \&active\_d;$
$comp \rightarrow vertices = v;$
$v \rightarrow next\_vertex = \Lambda;$
$v \rightarrow cmpt = comp;$
$comp \rightarrow num\_vertices = 1;$
$comp \rightarrow term\_sets\_size = (\mathbf{int}\ *)\ malloc((g \rightarrow num\_ts) * \mathbf{sizeof}(\mathbf{int}));$
$\mathbf{for}\ (j = 0;\ j < g \rightarrow num\_ts;\ j{+}{+})\ \ comp \rightarrow term\_sets\_size[j] = 0;$
$comp \rightarrow active = 0;$
$comp \rightarrow unconnected = 0;$
$ts = (\mathbf{TermSet}\ *)\ v \rightarrow termset;$
$\mathbf{if}\ (ts \wedge \neg(ts \rightarrow connected))\ \{$
$\quad comp \rightarrow active = 1;$
$\quad num\_actives{+}{+};$
$\quad comp \rightarrow term\_sets\_size[ts \rightarrow id]{+}{+};$
$\quad (comp \rightarrow unconnected){+}{+};$
$\}$
$t \rightarrow next = comp;$
$comp \rightarrow prev = t;$
$t = comp;$
$\}$
$t \rightarrow next = components;$
$components \rightarrow prev = t;$
⟨ Create an edge heap for each component 84 ⟩
$\}$

Este código é usado no bloco 80.

83. ⟨ Local variables of *sf_chlp* 83 ⟩ ≡
$\mathbf{int}\ num\_actives;$
$\mathbf{int}\ active\_d;$
$\mathbf{ES\_Component}\ *components;$

Veja também blocos 85 e 96.
Este código é usado no bloco 80.

84. 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 à componentes ativas da floresta será mantido em um heap de mínimo, o qual será apontado pelo valor guardado na variável $active_heaps$.

⟨ Create an edge heap for each component 84 ⟩ ≡
$\{$
$\quad \mathbf{Arc}\ **edges = (\mathbf{Arc}\ **)\ malloc(g \rightarrow m * \mathbf{sizeof}(\mathbf{Arc}\ *));$
$\quad \mathbf{FibHeap}\ *h = (\mathbf{FibHeap}\ *)\ malloc(g \rightarrow n * \mathbf{sizeof}(\mathbf{FibHeap}));$
$\quad \mathbf{ES\_Component}\ comp;$
$\quad \mathbf{int}\ m = 0;$

```
for (comp = components->next; comp ≠ components; comp = comp->next) {
  Arc *a;
  Vertex *v = comp->vertices;
  int i, n = 0;
  for (a = v->arcs; a; a = a->next) edges[n++] = a;
  comp->edges = newFibHeap(edges, n);
  if (comp->active) h[m++] = comp->edges;
}
active_heaps = newBinHeap(h, m, m);
free(edges);
free(h);
}
```

Este código é usado no bloco 82.

85. ⟨ Local variables of *sf_chlp* 83 ⟩ +≡

```
BinHeap *active_heaps;
```

86. 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. O vetor *selected* guarda apenas as arestas terminais selecionadas, já que uma aresta do grafo original passa a fazer parte da floresta apenas quando uma fração terminal dela é selecionada.

⟨ Include an edge piece in the forest while there is some active component 86 ⟩ ≡

```
{
  Arc *uv, **selected;
  int n = 0, nk = 1, dual_cost = 0;
  for (i = 1; i ≤ k; i++) nk = nk * g->n;
  selected = (Arc **) malloc((g->n − 1) * sizeof(Arc *));
  while (num_actives > 0) {
    ⟨ set uv to point to an edge with the smallest slackness 87 ⟩
    active_d += key(uv);
    dual_cost += num_actives * key(uv);
    if (fields(uv)->current_len ≤ (uv->len)/nk)    /* uv is a terminal edge */
    {
      ES_Component *U, *V;
      Arc *vu;
      selected[n++] = uv;
      vu = (uv->origin < uv->tip) ? uv + 1 : uv − 1;
      U = uv->origin->cmpt;
      V = vu->origin->cmpt;
      U->edges = delMinFibHeap(U->edges);    /* delete uv from U->edges */
      delMinBinHeap(active_heaps);    /* delete U->edges from the heap */
      V->edges = deleteNodeFibHeap(V->edges, vu);
      if (V->active) {
        deleteBinHeap(active_heaps, V->pos);
        num_actives −−;
      }
```

```
      ⟨ unite the components U and V 88 ⟩
    }
    else {
      Arc *wv;
      ES_Component *U;
      U = (ES_Component *) uv→origin→cmpt;
      U→edges = delMinFibHeap(U→edges);      /* delete uv from U->edges */
      delMinBinHeap(active_heaps);      /* delete U->edges from the heap */
      wv = (uv→origin < uv→tip) ? uv + 1 : uv − 1;
      ⟨ split the edge wv 92 ⟩
    }
  }
  ⟨ construct a Steiner forest with the selected edges 95 ⟩
}
```

Este código é usado no bloco 80.

87. A aresta escolhida para fazer para ser incluída na floresta corrente é uma aresta externa que apresenta o menor 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 da componente contenha arestas que não são externas. Deste modo, o laço **while** no trecho de código abaixo itera, removendo do heap de alguma componente ativa uma aresta que apresenta o menor valor de *key*( ) dentre todas as outras nos heaps de componentes ativas, até que uma aresta externa seja encontrada.

```
⟨ set uv to point to an edge with the smallest slackness 87 ⟩ ≡
{
  while (1) {
    Arc *vu;
    FibHeap h;
    ES_Component *U, *V;
    h = findMinBinHeap(active_heaps);
    uv = findMinFibHeap(h);
    U = (ES_Component *) uv→origin→cmpt;
    V = (ES_Component *) uv→tip→cmpt;
    if (U ≠ V) break;
    U→edges = delMinFibHeap(U→edges);      /* delete uv from U->edges */
    vu = (uv→origin < uv→tip) ? uv + 1 : uv − 1;
    U→edges = deleteNodeFibHeap(U→edges, vu);
        /* actualize the positision of U->edges in the heap */
    increaseKeyBinHeap(active_heaps, 1, U→edges);
  }
}
```

Este código é usado no bloco 86.

88. Sempre que uma aresta terminal 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 são 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 do 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*.

```
⟨ unite the components U and V 88 ⟩ ≡
  {
    int active_ = 1;
    if (U→num_vertices < V→num_vertices) {
      ES_Component *t = U;
      U = V;
      V = t;
    }
    ⟨ Include each vertex of V in U and set active_ to the new state of U 89 ⟩
    ⟨ Unite the edge heaps of U and V 90 ⟩
    U→active = active_;
    if (U→active) insertBinHeap(active_heaps, U→edges);
    else {
      U→d1 += *(U→d2);
      num_actives−−;
    }
    V→next→prev = V→prev;
    V→prev→next = V→next;
    free(V→term_sets_size);
    free(V);
  }
```

Este código é usado no bloco 86.

89. 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.

```
⟨ Include each vertex of V in U and set active_ to the new state of U 89 ⟩ ≡
  {
    Vertex *v = V→vertices;
    while (v) {
      TermSet *ts;
      Vertex *z = v→next_vertex;
      v→next_vertex = U→vertices;
      U→vertices = v;
      U→num_vertices++;
      ((ES_Component *) v→cmpt) = U;
      ts = (TermSet *) v→termset;
      if (ts ∧ ¬(ts→connected)) {
        U→term_sets_size[ts→id]++;
        if (U→term_sets_size[ts→id] ≡ 1) {
          (U→unconnected)++;
          if (U→unconnected ≡ 1) active_ = 1;
```

```
        }
        else if (U→term_sets_size[ts→id] ≡ ts→num_vertices) {
          ts→connected = 1;
          (U→unconnected)−−;
          if (¬(U→unconnected)) active_ = 0;
        }
      }
      v = z;
    }
  }
```
Este código é usado no bloco 88.

90. 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.

⟨ Unite the edge heaps of U and V 90 ⟩ ≡
```
  {
    int dU = U→active ? U→d1 + *(U→d2) : U→d1;
    int dV = V→active ? V→d1 + *(V→d2) : V→d1;
    traverse(V→edges, inc_field_d, dV − dU);
    U→edges = mergeFibHeap(U→edges, V→edges);
  }
```
Este código é usado no bloco 88.

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

⟨ Auxiliary functions 24 ⟩ +≡
```
  void inc_field_d(Arc *a, int inc)
  {
    fields(a)→d += inc;
  }
```

92. Quando a aresta uv incluída na floresta corrente não é uma aresta terminal, a aresta original, da qual uv é uma fração, 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 não faz parte da floresta corrente. Esta fração será subdividida até obtermos uma aresta terminal ou duas arestas externas. No primeiro caso, uv e wv estão representando a mesma aresta terminal; no segundo caso, uv e wv representarão as arestas externas que possuem um s-vértice v como extremo comum.

⟨ split the edge wv 92 ⟩ ≡
```
  {
    ES_Component *W;
    int cur_len, new_len, d_w, inc, p;
```

```
    W = (ES_Component *) wv→origin→cmpt;
    inc = W→active ? W→d1 + *(W→d2) : W→d1;
    d_w = fields(wv)→d + inc;
    new_len = cur_len = fields(wv)→current_len;
    do {
      p = new_len % 2;
      new_len = p ? new_len/2 + 1 : new_len = new_len/2;
    } while (new_len > (vw→len)/nk ∧ cur_len − new_len ≤ d_w);
    if (new_len ≤ (wv→len)/nk) ⟨ set uv and wv to represent the same terminal edge 93 ⟩
    else {
      ⟨ set uv and wv to represent two external edges 94 ⟩
      W→edges = decreaseKeyFibHeap(W→edges, wv);
      decreaseKeyBinHeap(active_heaps, W→pos, W→edges);
    }
    U→edges = insertFibHeap(U→edges, uv);
    insertBinHeap(active_heaps, U→edges);
  }
```
Este código é usado no bloco 86.

93. 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 *current_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.

```
⟨ set uv and wv to represent the same terminal edge 93 ⟩ ≡
  {
    fields(uv)→current_len = fields(wv)→current_len = new_len;
    fields(wv)→d = d_w − (cur_len − new_len) − inc;
    fields(uv)→d = d_w − (cur_len − new_len) − (U→d1 + *(U→d2));
  }
```
Este código é usado no bloco 92.

94. 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.

```
⟨ set uv and wv to represent two external edges 94 ⟩ ≡
  {
    fields(uv)→current_len = new_len;
    fields(wv)→current_len = p ? new_len − 1 : new_len;
    fields(uv)→d = −(U→d1 + *(U→d2));
    fields(wv)→d = fields(wv)→current_len − inc − (cur_len − new_len − d_w);
  }
```
Este código é usado no bloco 92.

95. Aqui é construída uma estrutura que representa a floresta de Steiner do grafo g que possui as arestas selecionadas durante o processo iterativo descrito no bloco 86. A variável *sf* armazena um ponteiro para esta estrutura.

$\langle$ construct a Steiner forest with the selected edges 95 $\rangle \equiv$

```
{
  int i;
  sf = (SteinerForest *) malloc(sizeof(SteinerForest));
  sf→edges = Λ;
  sf→cost = 0;
  for (i = 0; i < n; i++) {
    selected[i]→next_edge = sf→edges;
    sf→edges = selected[i];
    sf→cost += selected[i]→len;
  }
  sf→dual_cost = dual_cost;
}
```

Este código é usado no bloco 86.

96. $\langle$ Local variables of *sf_chlp* 83 $\rangle +\equiv$

```
SteinerForest *sf;
```

97. Implementação de Klein

Do mesmo modo que a implementação de Cole, Hariharan, Lewenstein e Porat apresentada acima, a implementação proposta por Klein tem um melhor desempenho que a implementação de Goemans e Williamson quando o grafo de entrada é esparso. A idéia central desta implementação consiste em definir uma estrutura de dados a partir do grafo de entrada e de uma atribuição de categorias aos vértices deste grafo e, então, remodelar o algoritmo de Goemans e Williamson em termos de certas operações sobre esta estrutura de dados.

Mais especificamente, dado um grafo orientado G com custos nos arcos e um número constante C de categorias, atribui-se a cada vértice de G uma das C categorias. Dado um vértice $v \in V_G$, denotaremos por $c(v)$ a categoria atribuída a v. Desta forma, cada arco uv do grafo G fica associado a uma *bicategoria*, dada pelo par ordenado $(c(u), c(v))$. A intenção aqui é implementar de modo eficiente o seguinte conjunto de operações sobre as bicategorias definidas desta forma:

- $decreaseCost(b, \delta)$, onde b é uma bicategoria e δ é um número real. Esta operação diminui de δ o custo de cada arco pertencente a 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 um arco a, transformando-o em um novo vértice e atribuindo a este vértice a categoria c. Tal operação remove a de E_G.

Estas operações podem ser implementadas de modo a consumir tempo amortizado $O(\sqrt{m}\log n)$ cada uma, desde que uma seqüência de ao menos $\sqrt{m}$ dessas operações seja executada. A seguir, mostraremos como as operações definidas acima podem ser usadas na implementação do algoritmo de Goemans e Williamson.

98. Descrição da implementação

Como o grafo G de entrada no algoritmo de Goemans e Williamson é não-orientado e 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.

As categorias que serão atribuidas a cada vértice v de G estarão relacionadas ao estado de atividade da componente da floresta corrente a que v pertence, ou seja, se v se encontra em uma componente ativa da floresta corrente então a categoria associada a v será ATIVO e, caso contrário, tal categoria será INATIVO. Deste modo, E_G ficará particionado em quatro bicategorias: (ATIVO, ATIVO), (ATIVO, INATIVO), (INATIVO, ATIVO) e (INATIVO, INATIVO).

Para cada aresta uv em E_G, definiremos o *custo reduzido* de uv, denotado por $\hat{c}_{uv}$, como sendo igual a $c_{uv} - d(u) - d(v)$. Note que, assim, a folga de cada aresta a pertencente à bicategoria (ATIVO, ATIVO) é igual a $\hat{c}_a/2$ e a folga de cada aresta a em (ATIVO, INATIVO) ou em (INATIVO, ATIVO) é igual a $\hat{c}_a$.

Agora, estamos aptos para descrever o algoritmo de Goemans e Williamson em termos das operações definidas no bloco anterior. No algoritmo abaixo, sempre que uma aresta for incluída na floresta iremos contrair tal aresta. Desta forma, em cada iteração, a floresta corrente não mais será composta por componentes ativas e inativas, mas sim por vértices ativos e inativos.

Algoritmo *MinFSAdaptado(G, R, c)*

1 $F \leftarrow (V_G, \emptyset)$
2 $\hat{c}_a \leftarrow c_a$ para cada a em E_G
3 $c(v) \leftarrow$ ATIVO para cada v em algum $T \in R$
4 $c(v) \leftarrow$ INATIVO para cada vértice de Steiner v
5 enquanto F tem algum vértice v tal que $c(v) =$ ATIVO faça
6     $a1 \leftarrow$ *findMin*(ATIVO, ATIVO)
7     $a2 \leftarrow$ *findMin*(ATIVO, INATIVO)
8     $a3 \leftarrow$ *findMin*(INATIVO, ATIVO)
9   escolha $\overline{a}$ tal que $folga(\overline{a}) = \min\{folga(a) \mid a \in \{a1, a2, a3\}\}$
10   inclua $\overline{a}$ em F
11   *contractEdge*$(\overline{a}, c)$ onde c é a categoria do vértice resultante
12   *decreaseCost*$(b, 2folga(\overline{a}))$ onde $b =$ (ATIVO, ATIVO)
13   *decreaseCost*$(b, folga(\overline{a}))$ para $b =$ (INATIVO, ATIVO)
14   *decreaseCost*$(b, folga(\overline{a}))$ para $b =$ (ATIVO, INATIVO)
15 seja F' uma R-floresta minimal contida em F
16 devolva F'

No algoritmo acima, as operações *findMin* e *decreaseCost* são realizadas considerando-se o custo reduzido de cada aresta. Assim, por exemplo, após a execução da linha 6 do algoritmo, $a1$ será uma aresta tal que $\hat{c}_{a1}$ é mínimo dentre todas as arestas em (ATIVO, ATIVO) e, após a execução da linha 12, o custo reduzido de todas as arestas em (ATIVO, ATIVO) sofrerá um decréscimo de $2folga(\overline{a})$. Na linha 11 do algoritmo, a operação *contractEdge* inicialmente muda para c a categoria dos vértices que são extremos de a, utilizando para isto duas chamadas a operação *changeCategory*.

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. Assim, como neste caso cada operação tem custo amortizado $O(\sqrt{m}\log n)$, a complexidade da implementação acima é $O(n\sqrt{m}\log n)$.

99. Limpeza das arestas

Após a construção iterativa da floresta de Steiner inicial, tem início a segunda fase do algoritmo, conhecida como fase de *limpeza das arestas*. 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 um apontador g para uma estrutura do tipo **Graph** e um apontador *sf* para uma estrutura que representa uma floresta de Steiner no grafo g e devolve um apontador para uma estrutura correspondente a uma floresta de Steiner minimal contida em *sf*.

```
⟨ Auxiliary functions 24 ⟩ +≡
  ⟨ More auxiliary functions 101 ⟩
  SteinerForest *edgePrunning(SteinerForest *sf, Graph *g)
  {
    Graph *sf_graph;
    ⟨ Construct sf_graph, a spanning subgraph of g with the edges of sf 100 ⟩
    ⟨ Preprocess the trees in sf_graph for the lca queries 102 ⟩
    ⟨ Find the lca of each terminal set 103 ⟩
    ⟨ Determine the necessary edges with the aid of the lcas 105 ⟩
  }
```

100. Aplicaremos o algoritmo para o cálculo dos ancestrais em cada uma das árvores da floresta. Como para isto é necessário que essas árvores sejam enraizadas, determinaremos, para cada árvore, uma raiz e uma relação de descendência entre seus vértices. Para facilitar esta tarefa, construiremos uma estrutura do tipo **Graph** contendo as arestas da floresta *sf*. O trecho de código abaixo é responsável pela construção desta estrutura. A variável *sf_graph* guardará um apontador para a estrutura construída.

```
⟨ Construct sf_graph, a spanning subgraph of g with the edges of sf 100 ⟩ ≡
  {
    Arc *a;
    Vertex *v;
    sf_graph = gb_new_graph(g→n);
    for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
      Vertex *w = g→vertices + (v − sf_graph→vertices);
      v→name = gb_save_string(w→name);
    }
    for (a = sf→edges; a; a = a→next_edge) {
      Vertex *u = sf_graph→vertices + (a→origin − g→vertices);
      Vertex *v = sf_graph→vertices + (a→tip − g→vertices);
```

```
        gb_new_edge(u, v, a→len);
      }
    }
```

Este código é usado no bloco 99.

101. A função *initialize_parent*, aqui apresentada, realiza uma busca em largura a partir de um dado vértice r de um grafo conexo não-orientado G, utilizando a estrutura de dados do grafo para construir a árvore enraizada de raiz r definida pela busca. Tal árvore é construída preenchendo-se o conteúdo do campo *parent_edge* de cada vértice v do grafo da seguinte forma:

- se $v \equiv r$, o campo *parent_edge* de v guarda um apontador para um arco a tal que $a \notin E_G$ e $a \to tip \equiv r$;
- se $v \neq r$, o campo *parent_edge* de v guarda um apontador para o arco (v, w) do grafo indicando que w é o pai de v na árvore; tal arco é removido da lista de adjacências de v no grafo.

```
#define parent_edge u.A
#define next_item v.V
#define selected a.I

⟨ More auxiliary functions 101 ⟩ ≡
  void initialize_parent(Vertex *r)
  {
    Arc *a = (Arc *) malloc(sizeof(Arc));
    Vertex *head, *tail;

    a→tip = r;
    r→parent_edge = a;
    r→next_item = Λ;
    head = tail = r;
    while (head) {
      Arc *t = Λ;
      Vertex *v = head;

      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→parent_edge→selected = 0;
        w→next_item = Λ;
        tail→next_item = w;
        tail = tail→next_item;
        t = a;
      }
      head = head→next_item;
    }
```

```
  }
```

Veja também bloco 106.

Este código é usado no bloco 99.

102. Aqui é feito o pré-processamento das árvores em *sf_graph* para o cálculo dos vértices ancestrais. Para cada árvore da floresta, inicialmente, aplicamos a função *initialize_parent*, apresentada no bloco anterior, obtendo assim 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. Esta função pré-processa uma dada árvore enraizada de tal forma que as consultas pelo ancestral comum de dois vértices quaisquer na árvore podem ser respondidas de maneira rápida (em tempo constante).

⟨ Preprocess the trees in *sf_graph* for the lca queries 102 ⟩ ≡

```
{
  Vertex *v;
  for (v = sf_graph->vertices; v < sf_graph->vertices + sf_graph->n; v++) v->parent_edge = Λ;
  for (v = sf_graph->vertices; v < sf_graph->vertices + sf_graph->n; v++) {
    if (v->parent_edge) continue;
    initialize_parent(v);
    lca_preprocessing(v);
  }
}
```

Este código é usado no bloco 99.

103. O trecho de código abaixo é responsável pelo cálculo do ancestral comum mais próximo de cada conjunto de terminais do grafo. 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, para cada vértice w no grafo g, *w->ts_list* guardará 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, *w->ts_list* será igual a Λ.

```
#define ts_list w.S
```

⟨ Find the lca of each terminal set 103 ⟩ ≡

```
{
  TermSet *ts;
  Vertex *v;
  for (v = g->vertices; v < g->vertices + g->n; v++) v->ts_list = Λ;
  for (ts = (TermSet *) g->t_sets; ts < ((TermSet *) g->t_sets) + g->num_ts; ts++) {
    Vertex *w = ts->vertices;
    Vertex *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 = (TermSet *) w->ts_list;
    ((TermSet *) w->ts_list) = ts;
```

```
    }
  }
```

Este código é usado no bloco 99.

104. 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.

⟨ Header files 9 ⟩ +≡

```
#include "lca.h"
```

105. 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.

```
#define visit y.I
#define next_lca x.V
```

⟨ Determine the necessary edges with the aid of the lcas 105 ⟩ ≡

```
  {
    Arc *t = sf->edges = (Arc *) malloc(sizeof(Arc));
    Vertex *l, *v, *w;

    sf->cost = 0;
    for (v = sf_graph->vertices; v < sf_graph->vertices + sf_graph->n; v++) {
      v->visit = 0;
      v->parent_edge->selected = 0;
    }
    for (v = g->vertices; v < g->vertices + g->n; v++) v->next_lca = Λ;
    l = w = (Vertex *) malloc(sizeof(Vertex));
    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 = (TermSet *) w->ts_list; 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 107 ⟩
      }
    }
    w = l;
  }
  free(l);
  t→next_edge = Λ;
  t = sf→edges;
  sf→edges = t→next_edge;
  free(t);
  return sf;
}
```

Este código é usado no bloco 99.

106. 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* $\neq \Lambda$, 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).

```
⟨ More auxiliary functions 101 ⟩ +≡
  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_→ts_list) {
      (*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);
```

```
    }
  }
```

107. 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 *selected* da aresta, e incluída na floresta *sf*.

⟨ select the edges in the path from z to w in *sf_graph* 107 ⟩ ≡

```
  {
    Vertex *w_ = sf_graph→vertices + (w − g→vertices);
    Vertex *z_ = sf_graph→vertices + (z − g→vertices);
    while (z_ ≠ w_ ∧ ¬(z_→parent_edge→selected)) {
      t→next_edge = z_→parent_edge;
      sf→cost += z_→parent_edge→len;
      z_→parent_edge→selected = 1;
      t = t→next_edge;
      z_ = z_→parent_edge→tip;
    }
  }
```

Este código é usado no bloco 105.

108. Execução das implementações e impressão dos resultados

⟨ Find a Steiner forest 108 ⟩ ≡

```
  {
    if (¬(sf = sfgw(graph))) {
      exit(1);
    }
  }
```

Este código é usado no bloco 6.

109. 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.

⟨ Show the results 109 ⟩ ≡

```
  {
    Arc *a;
    fprintf(stdout, "Edges:\n");
    for (a = sf→edges; a; a = a→next_edge)
      fprintf(stdout, "(%s,␣%s)\n", a→origin→name, a→tip→name);
    fprintf(stdout, "\nRate:␣%f\n", (sf→cost)/(sf→dual_cost));
    return 0;
  }
```

Este código é usado no bloco 6.

110. ⟨ Local variables of function *main* 10 ⟩ +≡
SteinerForest $*sf$;

111. 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 [7]. Uma descrição mais detalhada pode ser encontrada em [8].

112. 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)*

$i_x \leftarrow$ índice do bit 1 mais à direita de in(x)
$i_y \leftarrow$ índice do bit 1 mais à direita de in(y)
$i_{max} \leftarrow max\{i_x, i_y\}$
se os $l - i_{max}$ primeiros bits de $in(x)$ são iguais aos $l - i_{max}$ primeiros bits de $in(y)$
então
    se $i_x = i_{max}$
    então devolva x
    senão devolva y
senão
    $i \leftarrow$ índice do bit mais à esquerda em que $in(x)$ e $in(y)$ diferem
    seja b o número formado pelos $l-i$ primeiros bits de $in(x)$ seguidos por um bit 1 e i bits 0
    seja z o vértice de T tal que $in(z) = b$
    devolva z

113. 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.

114. 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.

115. 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, $z = \bar{y}$.

116. Implementação

A partir de agora, apresentaremos uma implementação do algoritmo que acabamos de descrever. A estrutura geral da implementação é dada abaixo.

⟨ `lca.c` 116 ⟩ ≡
  ⟨ Header files 9 ⟩
  ⟨ Global variables 121 ⟩
  ⟨ Auxiliary functions 24 ⟩
  ⟨ External functions 120 ⟩

117. 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.

⟨ Header files 9 ⟩ +≡
#include `"gb_graph.h"`

118. 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.

⟨ `lca.h` 118 ⟩ ≡
  void *lca_preprocessing*(**Vertex** ∗*r*);
  Vertex ∗*lca*(**Vertex** ∗*u*, **Vertex** ∗*v*);

119. ⟨ Header files 9 ⟩ +≡
#include `"lca.h"`

120. 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: $u\to level$, $u\to inlabel$ e $u\to ascendant$, 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, $r\to parent\_edge\to tip \equiv r$ e, se $v \neq r$, $v\to parent\_edge\to tip$ será um ponteiro para o pai de v na árvore. Por fim, para todo vértice v, o arco $v\to parent\_edge$ não deve fazer parte da lista de adjacência de v na árvore.

#define *parent_edge* *u.A*
#define *level* *v.I*
#define *inlabel* *x.I*
#define *ascendant* *y.I*

⟨ External functions 120 ⟩ ≡

```
void lca_preprocessing(Vertex *r)
{
  ⟨ Compute the level and the inlabel number of each vertex 123 ⟩
  ⟨ Compute the ascendant number of each vertex and construct the head table 125 ⟩
}
```

Veja também bloco 128.

Este código é usado no bloco 116.

121. ⟨ Global variables 121 ⟩ ≡

```
Vertex **head;
```

Este código é usado no bloco 116.

122. 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.

⟨ Auxiliary functions 24 ⟩ +≡

```
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);
}
```

123. 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.

⟨ Compute the level and the inlabel number of each vertex 123 ⟩ ≡

```
{
  r→level = −1;
  label1(r);
}
```

Este código é usado no bloco 120.

124. 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*.

```
#define preorder y.I
#define size z.I
```

⟨ Auxiliary functions 24 ⟩ +≡

```
void label1(Vertex *v)
{
  Arc *a;
  Vertex *w;
  int i, x;
  static long p = 1;
  w = v->parent_edge->tip;
  v->level = w->level + 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;
}
```

125. Aqui é feito o cálculo do valor do rótulo *ascendant* de cada um dos vértices na árvore e, além disso, a tabela *head*[0..(*r→inlabel*)] é criada e preenchida. A maior parte desse serviço é feita pela função *label2*, a qual descreveremos logo adiante.

⟨ Compute the ascendant number of each vertex and construct the *head* table 125 ⟩ ≡

```
{
  Arc *a;
  head = (Vertex **) malloc((r->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 120.

126.⟨ Header files 9 ⟩ +≡

```
#include <stdlib.h>
```

127. 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.

$\langle$ Auxiliary functions 24 $\rangle$ +≡

```
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);
}
```

128. 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*.

$\langle$ External functions 120 $\rangle$ +≡

```
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 129 ⟩
  ⟨ Find and return the lca of x and y 131 ⟩
}
```

129. 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.

$\langle$ Find *lca_inlabel*, the inlabel number of the lca of x and y 129 $\rangle$ ≡

```
{
  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 128.

130. 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.

⟨ Auxiliary functions 24 ⟩ +≡

```
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);
}
```

131. 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.

⟨ Find and return the lca of x and y 131 ⟩ ≡

```
{
  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((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((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 128.

132.⟨ Header files 9 ⟩ +≡

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

133. Considerações finais

Aqui, faremos algumas considerações sobre atividades realizadas durante o decorrer deste semestre e sobre alguns planos para atividades futuras.

Inicialmente, as atividades foram voltadas para a conclusão do programa CWEB correspondente a implementação de Goemans e Williamson e do código correspondente ao algoritmo de Schieber e Vishkin para o cálculo do ancestral comum mais próximo. As versões aqui apresentadas são praticamente definitivas e, deste modo, não mais sofrerão alterações substanciais.

A versão atual do código relativo a implementação de Cole, Hariharan, Lewenstein e Porat que foi aqui apresentada ainda necessita de alguns ajustes. Por este motivo, não foi possível realizar um estudo experimental comparativo entre esta e a implementação de Goemans e Williamson como havíamos previsto no plano de estudos apresentado no início do semestre.

Os próximos passos dentro deste projeto serão tomados em direção a obtenção de uma versão definitiva do código referente a implementação de Cole, Hariharan, Lewenstein e Porat. Após sua conclusão, daremos inicío aos testes de comparação de desempenho entre as implementações já codificadas. A seguir, iremos codificar a implementação sugerida por Klein.

134. Bibliografia

Referências

[1] M.H. Carvalho, M.R. Cerioli, R. Dahab, P. Feofiloff, C.G. Fernandes, C.E. Ferreira, K.S. Guimarães, F.K. Miyazawa, J.C. Pina Jr., J. Soares e Y. Wakabayashi, *Uma Introdução Sucinta a Algoritmos de Aproximação*, Publicações Matemáticas do IMPA, 2001.

[2] M. Goemans e D. Williamson, *A General Approximation Technique for Constrained Forest Problems*, SIAM Journal on Computing 24 (2), pp. 296–317, 1995.

[3] R. Cole, R. Hariharan, M. Lewenstein e E. Porat, *A Faster Implementation of the Goemans-Williamson Clustering Algorithm*, SODA 2001.

[4] H. Gabow, M. Goemans e D. Williamson, *An Efficient Approximation Algorithm for the Survivable Network Design Problem*, IPCO 1993.

[5] P.N. Klein, *A Data Structure for Bicategories with Applications to Speeding Up an Approximation Algorithm*, Information Processing Letters 52, pp. 303–307, 1994.

[6] D.E.Knuth, *The Stanford GraphBase: a platform for combinatorial computing*, ACM Press, 1993.

[7] B. Schieber, U. Vishkin, *On Finding Lowest Common Ancestors: Simplification and Parallelization*, SIAM Journal on Computing, 17, 6, 1984, pp. 1253-1262.

[8] J.H. Reif, *Syntesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 259-271.

Índice Remissivo

Lista de Refinamentos