Implementações do Método de Aproximação Primal-Dual aplicado ao problema da floresta de Steiner

Rafael Pereira Luna

Orientadora: Profa. Dra. Cristina Gomes Fernandes

— São Paulo, dezembro de 2004 —

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

Sumário

Capítulo 1

Introdução

Problemas de otimização combinatória têm aplicações nas mais diversas áreas, incluindo, por exemplo, projeto de redes de telecomunicação, de circuitos VLSI, empacotamento de objetos em containers, escalonamento e roteamento de veículos, etc. Vários destes problemas são reconhecidamente difíceis — NP-difíceis — e algoritmos de aproximação [10, 18, 3] são uma forma de atacá-los que têm recebido bastante atenção dentro de otimização combinatória na última década.

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

O algoritmo de Bar-Yehuda e Even [2], de 1981, para o problema da cobertura por vértices, foi o primeiro uso (ainda que implícito) do método de aproximação primal-dual. Foram os trabalhos de Agrawal, Klein e Ravi [1] e de Goemans e Williamson [8], da década de 90, que retomaram e formalizaram o uso do método para o projeto de algoritmos de aproximação. Desde a publicação desses trabalhos, vários outros apareceram, tratando de problemas diferentes [4, 7, 9, 12, 11, 15] ou de implementações alternativas que tornam o método mais eficiente [5, 6, 13].

Apresentaremos aqui algumas das implementações do método de aproximação primal-dual aplicado ao problema da floresta de Steiner. Inicialmente, definiremos o problema em questão e daremos uma descrição do algoritmo de aproximação primal-dual proposto por Goemans e Williamson para ele [8]. Após isso, apresentaremos quatro maneiras distintas de implementar este algoritmo, baseadas nas implementações sugeridas por Goemans e Williamson [8], Cole et al [5], Klein [13] e Gabow, Goemans e Williamson [6], exibindo ainda uma implementação em CWEB para cada uma delas. Ao final, serão apresentados alguns resultados obtidos através da análise experimental dessas implementações.

Neste capítulo, iremos inicialmente definir a notação e alguns conceitos básicos que serão bastante úteis para o desenvolvimento e a compreensão de todo o resto do texto. Após isso, passaremos a abordar o problema da floresta de Steiner e alguns resultados de interesse relativos a este problema. Por fim, diremos algumas palavras sobre *literate programming*, CWEB e sobre a plataforma SGB [14], com o auxílio da qual foram realizadas as implementaçãoes apresentadas neste texto.

1.1 Notação e conceitos básicos

1.2 O problema da floresta de Steiner

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 $\mathcal{R}$ de subconjuntos de V_G, encontrar uma $\mathcal{R}$-*floresta* F que minimize $c(F) = \sum_{e \in E_F} c_e$.

Cada conjunto da coleção $\mathcal{R}$ é chamado de *conjunto de terminais* e os vértices que não pertencem aos conjuntos de $\mathcal{R}$ são chamados de *vértices de Steiner*. Uma R-*floresta* F é uma floresta geradora de G que tem a seguinte propriedade: cada conjunto de terminais em $\mathcal{R}$ se encontra inteiramente contido no conjunto dos vértices de alguma das componentes de F. Quando $\mathcal{R}$ está implícito, dizemos que uma floresta satisfazendo tal propriedade é uma *floresta de Steiner* de G. Note que podemos assumir que os conjuntos de $\mathcal{R}$ 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 $\mathcal{R}$ consiste em apenas um conjunto, o problema se reduz ao conhecido *problema de Steiner em grafos*, o qual é NP-difícil.

1.3 Literate programming e CWEB

1 ⟨ Header files of sf.c 71 ⟩
⟨ Data structures of sf.c 68 ⟩
⟨ Auxiliary functions 76 ⟩
⟨ Steiner forest construction functions 77 ⟩
⟨ The main program 209 ⟩

1.4 Stanford Graph Base

Capítulo 2

Estruturas de dados

Aqui descreveremos as estruturas de dados utilizadas nas implementações que veremos a partir do capítulo 4.

3 ⟨ Header files of item.c 4 ⟩
⟨ Data structures of item.c 5 ⟩
⟨ Global variables of item.c 7 ⟩
⟨ Internal functions of item.c 9 ⟩
⟨ Itens manipulation functions 6 ⟩

4 ⟨ Header files of item.c 4 ⟩ ≡

```
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
```

Este código é usado no bloco 3.

5 ⟨ Data structures of item.c 5 ⟩ ≡

```
struct item_struct {
  void *value;
  double(*key)(void *);
  void *min;
};
```

Este código é usado no bloco 3.

6 ⟨ Itens manipulation functions 6 ⟩ ≡

```
void allocItens(int n)
{
  int i;
  if (blocks ≡ 10) {
    fprintf(stderr, "\n␣Overflow!!!\n");
    exit(1);
  }
  if (¬blocks) {
    I = malloc((n + 1) * sizeof(struct item_struct));
    for (i = 0; i < n; i++) I[i].value = I + i + 1;
    I[n].value = I;
  }
  else {
    Item new_block = malloc(n * sizeof(struct item_struct));
    block[blocks − 1] = new_block;
    for (i = 0; i < n − 1; i++) new_block[i].value = new_block + i + 1;
    new_block[i].value = I→value;
    I→value = new_block;
  }
  blocks++;
}
```

Veja também blocos 8, 10, 11, 12, 13, 14 e 15.

Este código é usado no bloco 3.

7 ⟨ Global variables of item.c 7 ⟩ ≡

```
int blocks = 0;
Item I, block[10];
```

Veja também bloco 16.

Este código é usado no bloco 3.

8 ⟨ Itens manipulation functions 6 ⟩ +≡

```
void freeItens( )
{
  static int calls = 0;
```

```
    calls++;
    if (calls ≡ blocks) {
      while (blocks > 0) {
        blocks--;
        free(block[blocks]);
      }
      free(I);
    }
  }
```

9 ⟨ Internal functions of item.c 9 ⟩ ≡

```
  Item allocItem()
  {
    Item x = I→value;
    if (x ≡ I) {
      printf("Item: Overflow!!!\n");
      exit(1);
    }
    I→value = x→value;
    return x;
  }
```

Este código é usado no bloco 3.

10 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  void freeItem(Item a)
  {
    a→value = I→value;
    I→value = a;
  }
```

11 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  Item newItem(void *value, double(*key)(void *), void *min)
  {
    Item x = allocItem();
```

```
    x→value = value;
    x→key = key;
    x→min = min;
    return x;
  }
```

12 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  void *getValue(Item a)
  {
    return a→value;
  }
```

13 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  void setValue(Item a, void *value)
  {
    a→value = value;
  }
```

14 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  double key(Item a)
  {
    return a→key(a→value);
  }
```

15 ⟨ Itens manipulation functions 6 ⟩ +≡

```
  Item minItem(Item a)
  {
    Item min_item = &min;
    min_item→value = min_item→min = a→min;
    min_item→key = a→key;
    return min_item;
  }
```

16 ⟨ Global variables of item.c 7 ⟩ +≡

```
  struct item_struct min;
```

17 ⟨ item.h 17 ⟩ ≡

```
typedef struct item_struct *Item;
void allocItens(int n);
Item newItem(void *value, double(*key)(void *), void *min);
void *getValue(Item a);
void setValue(Item a, void *value);
double key(Item a);
Item minItem(Item a);
void freeItem(Item a);
void freeItens( );
```

2.1 Árvores binárias de busca balanceadas

18 ⟨ bbst.h 18 ⟩ ≡

```
typedef struct bbst_struct *BalBST;
void BBSTalloc(int n);
BalBST BBSTinit( );
BalBST BBSTinsert(BalBST r, Item x, Item *y);
void BBSTtraverse(BalBST r, void(*func)(Item item, void *args[]), void *args[]);
void BBSTdestroy(BalBST r);
void BBSTfree( );
```

19

```
⟨ Header files of bbst.c 20 ⟩
⟨ Data structures of bbst.c 21 ⟩
⟨ Global variables of bbst.c 23 ⟩
⟨ Internal functions of bbst.c 25 ⟩
⟨ Trees manipulation functions 22 ⟩
```

20 ⟨ Header files of bbst.c 20 ⟩ ≡

```
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
#include "bbst.h"
```

Este código é usado no bloco 19.

21 ⟨ Data structures of bbst.c 21 ⟩ ≡

```
struct bbst_struct {
  Item item;
  BalBST l;
  BalBST r;
  int bal;
};
```

Este código é usado no bloco 19.

22 ⟨ Trees manipulation functions 22 ⟩ ≡

```
void BBSTalloc(int n)
{
  int i;
  T = malloc((n + 1) * sizeof(struct bbst_struct));
  for (i = 0; i < n; i++) T[i].l = T + i + 1;
  T[n].l = T;
}
```

Veja também blocos 24, 28, 29, 32 e 33.

Este código é usado no bloco 19.

23 ⟨ Global variables of bbst.c 23 ⟩ ≡

```
BalBST T;
```

Este código é usado no bloco 19.

24 ⟨ Trees manipulation functions 22 ⟩ +≡

```
void BBSTfree( )
{
  free(T);
}
```

25 ⟨ Internal functions of bbst.c 25 ⟩ ≡

```
BalBST allocNode( )
{
  BalBST x = T→l;
```

```
    if (x ≡ T) {
      printf("BBST: Overflow!!!\n");
      exit(1);
    }
    T→l = x→l;
    return x;
  }
```

Veja também blocos 26 e 27.

Este código é usado no bloco 19.

26 ⟨ Internal functions of bbst.c 25 ⟩ +≡

```
  void freeNode(BalBST x)
  {
    x→l = T→l;
    T→l = x;
  }
```

27 ⟨ Internal functions of bbst.c 25 ⟩ +≡

```
  BalBST newNode(Item item)
  {
    BalBST x = allocNode( );
    x→l = Λ;
    x→r = Λ;
    x→item = item;
    x→bal = 0;
    return x;
  }
```

28 ⟨ Trees manipulation functions 22 ⟩ +≡

```
  BalBST BBSTinit( )
  {
    return Λ;
  }
```

Recebe:

- um item x;
- o nó raiz r de uma árvore binária de busca AVL;
- um apontador y para um item.

E faz o que?

Tenta inserir x na árvore de raiz r. O item x somente é inserido caso não exista na árvore um item com a mesma chave que x. Caso já exista um item na árvore que possua a mesma chave de x, tal item será armazenado no endereço dado por y. Caso contrário, o conteúdo armazenado em y será Λ. Ao final de sua execução, a função devolve o novo nó raiz da árvore (note que a raiz de uma árvore AVL pode mudar após a inserção de um novo item).

29 ⟨ Trees manipulation functions 22 ⟩ +≡

```
BalBST BBSTinsert(BalBST r, Item x, Item *y)
{
  static int inc;
  if (¬r) {
    inc = 1;
    *y = Λ;
    return newNode(x);
  }
  else {
    if (key(x) ≡ key(r→item)) {
      inc = 0;
      *y = r→item;
      return r;
    }
    else {
      if (key(x) < key(r→item)) {
        r→l = BBSTinsert(r→l, x, y);
        if (inc) {
          switch (r→bal) {
          case −1: ⟨ Case 1 30 ⟩
          case 0: r→bal = −1;
            return r;
          default:      /* inc == 1 */
            r→bal = 0;
            inc = 0;
            return r;
```

```
            }
          }
          else return r;
        }
        else {
          r→r = BBSTinsert(r→r, x, y);
          if (inc) {
            switch (r→bal) {
            case −1: r→bal = 0;
              inc = 0;
              return r;
            case 0: r→bal = 1;
              return r;
            default:      /* inc == 1 */
              ⟨ Case 2 31 ⟩
            }
          }
          else return r;
        }
      }
    }
```

30 ⟨ Case 1 30 ⟩ ≡

```
{
  BalBST u = r→l;
  inc = 0;
  if (u→bal ≡ −1) {
    r→l = u→r;
    u→r = r;
    r→bal = 0;
    u→bal = 0;
    return u;
  }
  else {
    BalBST v = u→r;
    u→r = v→l;
    r→l = v→r;
```

```
    v↣l = u;
    v↣r = r;
    if (v↣bal ≡ −1) r↣bal = 1;
    else r↣bal = 0;
    if (v↣bal ≡ 1) u↣bal = −1;
    else u↣bal = 0;
    v↣bal = 0;
    return v;
  }
}
```

Este código é usado no bloco 29.

31 ⟨ Case 2 31 ⟩ ≡

```
{
  BalBST u = r↣r;
  inc = 0;
  if (u↣bal ≡ 1) {
    r↣r = u↣l;
    u↣l = r;
    r↣bal = 0;
    u↣bal = 0;
    return u;
  }
  else {
    BalBST v = u↣l;
    u↣l = v↣r;
    r↣r = v↣l;
    v↣l = r;
    v↣r = u;
    if (v↣bal ≡ 1) r↣bal = −1;
    else r↣bal = 0;
    if (v↣bal ≡ −1) u↣bal = 1;
    else u↣bal = 0;
    v↣bal = 0;
    return v;
  }
}
```

Este código é usado no bloco 29.

32 ⟨ Trees manipulation functions 22 ⟩ +≡

```
void BBSTtraverse(BalBST r, void(*func)(Item item, void *args[]), void *args[])
{
  if (r) {
    BBSTtraverse(r->l, func, args);
    BBSTtraverse(r->r, func, args);
    (*func)(r->item, args);
  }
}
```

33 ⟨ Trees manipulation functions 22 ⟩ +≡

```
void BBSTdestroy(BalBST r)
{
  if (r) {
    BBSTdestroy(r->l);
    BBSTdestroy(r->r);
    freeNode(r);
  }
}
```

2.2 Heaps de Fibonacci

34 ⟨ `fibheap.h` 34 ⟩ ≡

```
typedef struct fh *FibHeap;
typedef struct fh_node *FHnode;
void FHEAPalloc(int num_heaps, int num_elems);
FibHeap FHEAPinit();
int FHEAPempty(FibHeap H);
FHnode FHEAPinsert(FibHeap H, Item a);
Item FHEAPfindMin(FibHeap H);
void FHEAPdelMin(FibHeap H);
void FHEAPdecreaseKey(FibHeap H, FHnode x, Item a);
```

```
Item FHEAPdelete(FibHeap H, FHnode x);
void FHEAPtraverse(FibHeap H, void(*func)(Item item, void *args[]), void *args[]);
FibHeap FHEAPjoin(FibHeap H1, FibHeap H2);
void FHEAPdestroy(FibHeap H);
void FHEAPfree( );
```

35 ⟨ Header files of fibheap.c 36 ⟩
⟨ Global variables of fibheap.c 39 ⟩
⟨ Data structures of fibheap.c 37 ⟩
⟨ Internal functions of fibheap.c 41 ⟩
⟨ Heaps manipulation functions 38 ⟩

36 ⟨ Header files of fibheap.c 36 ⟩ ≡

```
#include <stdlib.h>
#include "item.h"
#include "fibheap.h"
```

Este código é usado no bloco 35.

37 ⟨ Data structures of fibheap.c 37 ⟩ ≡

```
struct fh {
  FHnode min;
};
struct fh_node {
  Item item;
  FHnode parent;
  FHnode child;
  FHnode right;
  FHnode left;
  long degree_mark;
};
```

Este código é usado no bloco 35.

38 ⟨ Heaps manipulation functions 38 ⟩ ≡

```
void FHEAPalloc(int num_heaps, int num_elems)
{
  int i;
```

```
    heaps = malloc((num_heaps + 1) * sizeof(struct fh));
    for (i = 0; i < num_heaps; i++) heaps[i].min = (FHnode)(heaps + i + 1);
    heaps[i].min = (FHnode) heaps;
    nodes = malloc((num_elems + 1) * sizeof(struct fh_node));
    for (i = 0; i < num_elems; i++) nodes[i].parent = nodes + i + 1;
    nodes[i].parent = nodes;
  }
```

Veja também blocos 40, 46, 47, 48, 49, 50, 51, 53, 54, 56 e 57.

Este código é usado no bloco 35.

39 ⟨ Global variables of fibheap.c 39 ⟩ ≡

```
  FibHeap heaps;
  FHnode nodes;
```

Este código é usado no bloco 35.

40 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
  void FHEAPfree()
  {
    free(heaps);
    free(nodes);
  }
```

41 ⟨ Internal functions of fibheap.c 41 ⟩ ≡

```
  FibHeap allocFibHeap()
  {
    FibHeap x = (FibHeap)(heaps→min);
    if (x ≡ heaps) {
      printf("FHEAP: Overflow!!! (heaps)\n");
      exit(1);
    }
    heaps→min = x→min;
    return x;
  }
```

Veja também blocos 42, 43, 44, 45, 52, 55 e 58.

Este código é usado no bloco 35.

42 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
void freeFibHeap(FibHeap H)
{
  H→min = heaps→min;
  heaps→min = (FHnode) H;
}
```

43 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
FHnode allocFHnode( )
{
  FHnode x = nodes→parent;
  if (x ≡ nodes) {
    printf("FHEAP: Overflow!!! (nodes)\n");
    exit(1);
  }
  nodes→parent = x→parent;
  return x;
}
```

44 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
void freeFHnode(FHnode x)
{
  x→parent = nodes→parent;
  nodes→parent = x;
}
```

45 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
static void LISTinsert(FHnode a, FHnode b)
{
  FHnode c = a→right;
  b→right = c;
  c→left = b;
  a→right = b;
  b→left = a;
}
```

```
static void LISTdelete(FHnode b)
{
  FHnode a = b→left;
  FHnode c = b→right;
  a→right = c;
  c→left = a;
}
static void LISTconcatenate(FHnode a, FHnode d)
{
  FHnode b = a→right;
  FHnode c = d→left;
  a→right = d;
  d→left = a;
  b→left = c;
  c→right = b;
}
```

46 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
FibHeap FHEAPinit( )
{
  FibHeap H = allocFibHeap( );
  H→min = Λ;
  return H;
}
```

47 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
int FHEAPempty(FibHeap H)
{
  return (H→min ≡ Λ);
}
```

48 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
FHnode FHEAPinsert(FibHeap H, Item a)
{
  FHnode x = allocFHnode( );
```

```
    x→item = a;
    x→parent = x→child = Λ;
    x→degree_mark = 0;
    if (¬(H→min)) {
      x→left = x→right = x;
      H→min = x;
    }
    else {
      LISTinsert(H→min, x);
      if (key(a) < key(H→min→item)) H→min = x;
    }
    return x;
  }
```

49 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
  Item FHEAPfindMin(FibHeap H)
  {
    return H→min→item;
  }
```

50 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
  void FHEAPdelMin(FibHeap H)
  {
    FHnode min = H→min, v;
    if (¬min) return;
    if (¬(min→child) ∧ (min ≡ min→right)) {
      H→min = Λ;
      freeFHnode(min);
      return;
    }
    if (min→child) {
      FHnode x = min→child;
      while (x ≠ x→right) {
        FHnode y = x→right;
        LISTdelete(x);
        LISTinsert(min, x);
        x→parent = Λ;
```

```
      x = y;
    }
    min→child = Λ;
    LISTinsert(min, x);
    x→parent = Λ;
  }
  LISTdelete(min);      /* delete min from the root list */
  v = min→right;
  freeFHnode(min);
  {
    FHnode A[1 + 8 * sizeof(long)];      /* A[d] keeps all roots which have degree d */
    int i, Dn = 1 + 8 * sizeof(long);      /* 1 + log(n): maximum number of trees */

    for (i = 0; i < Dn; i++) A[i] = Λ;
    do {
      int d;
      FHnode x = v;

      v = v→right;
      if (x ≡ v) v = Λ;
      else LISTdelete(x);      /* delete x from the root list */
      d = x→degree_mark/2;      /* d is the degree of x */
      while (d < Dn ∧ A[d]) {
        FHnode y = A[d];

        if (key(x→item) > key(y→item)) {
          FHnode aux = x;

          x = y;
          y = aux;
        }
        {
          FHnode f = x→child;

          if (f) LISTinsert(f, y);
          else x→child = y→left = y→right = y;
          y→parent = x;
          x→degree_mark += 2;      /* increment by 1 the degree of x */
          y→degree_mark = (y→degree_mark ≫ 1) ≪ 1;      /* set to 0 the mark of y */
        }
        A[d] = Λ;
        d++;
```

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

51 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
void FHEAPdecreaseKey(FibHeap H, FHnode x, Item a)
{
  FHnode p;
  if (¬a) a = x→item;
  else {
    if (key(x→item) < key(a)) return;
    x→item = a;
  }
  p = x→parent;
  if (p ∧ key(a) < key(p→item)) {
    if (x→right ≡ x) p→child = Λ;
    else {
      if (p→child ≡ x) p→child = x→right;
      LISTdelete(x);      /* delete x from the child list of p */
    }
    p→degree_mark −= 2;      /* decrement by 1 the degree of p */
    LISTinsert(H→min, x);      /* insert x in the root list */
    x→parent = Λ;
    x→degree_mark = (x→degree_mark ≫ 1) ≪ 1;      /* set to 0 the mark of x */
    cascading_cut(H, p);
  }
  if (key(a) < key(H→min→item)) H→min = x;
}
```

52 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
void cascading_cut(FibHeap H, FHnode x)
{
  FHnode p;
  p = x→parent;
  if (p) {
    if (x→degree_mark % 2 ≡ 0) x→degree_mark++;      /* set to 1 the mark of x */
    else {
      if (x→right ≡ x) p→child = Λ;
      else {
        if (p→child ≡ x) p→child = x→right;
        LISTdelete(x);      /* delete x from the child list of p */
      }
      p→degree_mark -= 2;      /* decrement by 1 the degree of p */
      LISTinsert(H→min, x);      /* insert x in the root list */
      x→parent = Λ;
      x→degree_mark = (x→degree_mark ≫ 1) ≪ 1;      /* set to 0 the mark of x */
      cascading_cut(H, p);
    }
  }
}
```

53 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
Item FHEAPdelete(FibHeap H, FHnode x)
{
  Item item = x→item;
  FHEAPdecreaseKey(H, x, minItem(item));
  FHEAPdelMin(H);
  return item;
}
```

54 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
void FHEAPtraverse(FibHeap H, void(*func)(Item item, void *args[]), void *args[])
{
  traverse(H→min, func, args);
}
```

55 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
void traverse(FHnode r, void(*func)(Item item, void *args[]), void *args[])
{
  FHnode x;
  if (r) {
    (*func)(r→item, args);
    traverse(r→child, func, args);
    for (x = r→right; x ≠ r; x = x→right) {
      (*func)(x→item, args);
      traverse(x→child, func, args);
    }
  }
}
```

56 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
FibHeap FHEAPjoin(FibHeap H1, FibHeap H2)
{
  if (¬(H1→min)) {
    freeFibHeap(H1);
    return H2;
  }
  if (¬(H2→min)) {
    freeFibHeap(H2);
    return H1;
  }
  LISTconcatenate(H1→min, H2→min);
  if (key(H1→min→item) < key(H2→min→item)) {
    freeFibHeap(H2);
    return H1;
  }
  else {
    freeFibHeap(H1);
    return H2;
  }
}
```

57 ⟨ Heaps manipulation functions 38 ⟩ +≡

```
void FHEAPdestroy(FibHeap H)
{
  destroy(H→min);
  freeFibHeap(H);
}
```

58 ⟨ Internal functions of fibheap.c 41 ⟩ +≡

```
void destroy(FHnode r)
{
  FHnode x;
  if (r) {
    destroy(r→child);
    x = r→right;
    while (x ≠ r) {
      FHnode z = x→right;
      destroy(x→child);
      LISTdelete(x);
      freeFHnode(x);
      x = z;
    }
    freeFHnode(r);
  }
}
```

2.3 Union-Find

59 ⟨ Header files of uf.c 60 ⟩
⟨ Data structures of uf.c 61 ⟩
⟨ Global variables of uf.c 62 ⟩
⟨ Disjoint sets manipulation functions 63 ⟩

60 ⟨ Header files of uf.c 60 ⟩ ≡

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

Este código é usado no bloco 59.

61 ⟨ Data structures of uf.c 61 ⟩ ≡

```
typedef struct set_node {
  int p;
  int size;
} SetNode;
```

Este código é usado no bloco 59.

62 ⟨ Global variables of uf.c 62 ⟩ ≡

```
SetNode *set;
```

Este código é usado no bloco 59.

63 ⟨ Disjoint sets manipulation functions 63 ⟩ ≡

```
void UFinit(int n)
{
  int i;
  set = malloc(n * sizeof(SetNode));
  for (i = 0; i < n; i++) {
    set[i].p = i;
    set[i].size = 1;
  }
}
```

Veja também blocos 64, 65 e 66.

Este código é usado no bloco 59.

64 ⟨ Disjoint sets manipulation functions 63 ⟩ +≡

```
int UFfind(int key)
{
  int i = key, j;
  while (set[i].p ≠ i) i = set[i].p;
  for (j = key; set[j].p ≠ j; j = set[j].p) set[j].p = i;
  return i;
}
```

65 ⟨ Disjoint sets manipulation functions 63 ⟩ +≡

```
void UFunion(int key1, int key2)
{
  if (set[key1].size < set[key2].size) {
    int tmp = key1;
    key1 = key2;
    key2 = tmp;
  }
  set[key2].p = key1;
  set[key1].size += set[key2].size;
}
```

66 ⟨ Disjoint sets manipulation functions 63 ⟩ +≡

```
void UFdestroy()
{
  free(set);
}
```

67 ⟨ `uf.h` 67 ⟩ ≡

```
void UFinit(int n);
int UFfind(int key);
void UFunion(int key1, int key2);
void UFdestroy();
```

Capítulo 3

Método primal-dual de aproximação

As implementações que estudaremos são todas baseadas num algoritmo de aproximação projetado por Goemans e Williamson [8] para o problema. Tal algoritmo utiliza o método de aproximação primal-dual e é uma 2-aproximação para o problema da floresta de Steiner.

3.1 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 $\mathcal{R}$ de conjuntos de terminais, um subconjunto S de V_G é dito *ativo* se, e somente se, existe um conjunto T em $\mathcal{R}$ 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 $\mathcal{R}$-floresta de G então, para todo conjunto S ativo, $E_F \cap \delta(S) \neq \emptyset$, onde $\delta(S)$ é o corte determinado por S no grafo G. 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.

3.2 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, $\mathcal{R}$, c)*

1 $F \leftarrow (V_G, \emptyset)$
2 $E_{ext} \leftarrow E_G$
3 $d(v) \leftarrow 0$ para cada v em V_G
4 enquanto F tem alguma componente ativa faça
5 escolha $\overline{a}$ tal que $folga(\overline{a}) = \min\{folga(a) \mid a \in E_{ext}\}$
6 $d(v) \leftarrow d(v) + folga(\overline{a})$ para cada v em uma componente ativa
7 inclua $\overline{a}$ em F
8 $E_{ext} \leftarrow E_{ext} \setminus \{uv \in E_G \mid u$ e v pertencem a uma mesma componente de $F\}$
9 seja F' uma R-floresta minimal contida em F
10 devolva F'

3.3 Implementações

A implementação mais simples do algoritmo de Goemans e Williamson para o problema da floresta de Steiner consome tempo $O(n^3)$, onde n é o número de vértices do grafo dado.

Goemans e Williamson [9] mencionam uma implementação direta do algoritmo que consome tempo $O(n^2 \log n)$. Tal implementação utiliza várias filas de prioridade para tornar mais rápida a determinação de uma aresta de folga mínima.

Klein [13] introduziu uma estrutura de dados que pode ser usada para obter-se uma implementação do algoritmo de Goemans e Williamson de complexidade $O(n\sqrt{m}\log n)$, que é mais rápida que a implementação $O(n^2 \log n)$ em grafos esparsos. Aqui, como de costume, m e n denotam, respectivamente, o número de arestas e de vértices do grafo.

Gabow, Goemans e Williamson [6] propuseram uma outra implementação do algoritmo, que organiza as filas de prioridade de uma forma diferente e tem complexidade $O(n(n+\sqrt{m \log\log n}))$.

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

A partir do próximo capítulo, faremos uma exposição mais detalhada de cada uma dessas implementações e, para cada uma delas, exibiremos uma implementação em CWEB. Desta forma, vamos apresentar aqui as estruturas de dados que iremos utilizar para representar conjuntos de terminais e florestas de Steiner nestas implementações.

Cada conjunto de terminais é representado por uma estrutura do tipo **TermSet**. 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.

```
68 #define next_terminal  u.V      /* next vertex in the linked list vertices */
   #define termset_id  v.I      /* id of terminal set of the vertex */
   #define num_ts  uu.I      /* number of terminal sets */

⟨ Data structures of sf.c 68 ⟩ ≡
  typedef struct ts_struct {
    int id;
    Vertex *vertices;
    int num_vertices;
    int connected;
```

```
  struct ts_struct *next_tset;
} TermSet;
```

Veja também blocos 70, 73, 106, 110, 112 e 114.

Este código é usado no bloco 1.

No que segue, veremos que algumas vezes será necessário determinar o conjunto de terminais ao qual um dado vértice pertence. Desta forma, iremos definir aqui a macro `TERMSET( )` que nos será bastante útil nestas situações. Se v é um vértice terminal, a operação correspondente a `TERMSET`(v) devolve um apontador para o conjunto de terminais do qual v faz parte; caso contrário, o valor correpondente a `TERMSET`(v) será Λ.

69
```
#define TERMSET(V)  ((V→termset_id < 0) ? Λ : term_sets + V→termset_id)
```

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

70
```
#define next_edge  b.A
⟨ Data structures of sf.c 68 ⟩ +≡
  typedef struct sf_struct {
    Arc *edges;
    long cost;
    double dual_cost;
  } SteinerForest;
```

71
```
⟨ Header files of sf.c 71 ⟩ ≡
#include "gb_graph.h"
```

Veja também blocos 74, 75, 102, 109, 113, 183, 213 e 222.

Este código é usado no bloco 1.

Capítulo 4

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.

4.1 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 C, a aresta guardada é uma das que apresentam a menor folga dentre todas as arestas para C.

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 ativa.

Para que consigamos manter nos heaps de arestas de cada componente apenas a aresta que tem a menor folga dentre todas as arestas que a ligam a uma determinada componente da floresta, precisaremos ter acesso direto ao nó do heap que guarda tal aresta. Para alcançar este objetivo, a estrutura de cada componente possui um campo *adjacent* que consiste de um vetor com a seguinte propriedade: se existe uma aresta ligando a componente a uma dada componente da floresta cujo o identificador (guardado no campo *id* da estrutura de cada componente) é igual a i, então $adjacent[i]$ guarda o nó do heap de arestas da primeira que corresponde a aresta de menor folga entre as duas componentes; caso não existam arestas entre as duas componentes, $adjacent[i]$ será igual a Λ.

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 $v\rightarrowtail next\_vertex$. 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 *ts_intersect_size* é um vetor indexado pelos identificadores dos conjuntos de terminais do grafo. O valor guardado em $ts\_intersect\_size[i]$ 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 *disconnected*. 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 C nesta lista, $C\rightarrowtail prev$ e $C\rightarrowtail next$ serão, respectivamente, apontadores para a componente que antecede e para a componente que precede C na lista.

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

73 ⟨ Data structures of sf.c 68 ⟩ +≡

```
typedef struct gw_component {
  int id;
  int active;
  Vertex *vertices;
  long num_vertices;
```

```
    FibHeap edges[2];
    FHnode *adj;
    int *ts_ intersect_ size;
    int disconnected;
    struct gw_component *prev;
    struct gw_component *next;
  } GWComponent;
```

74 ⟨ Header files of sf.c 71 ⟩ +≡
```
#include "item.h"
#include "fibheap.h"
```

Como veremos, daqui para frente, muitas vezes será necessário determinar a componente a qual um dado vértice pertence. Por este motivo, cada vértice guardará o identificador da componente da qual faz parte, o qual ficará armazenado no campo *cmpnt_ id* da sua estrutura.

75 ```
#define ind w.I
#define REP(V) ((V) - (V)→ind + UFfind((V)→ind))
#define COMPONENT(V) (REP(V)→x.S)
#define SET_COMPONENT(V, C) ((V)→x.S = (char *)(C))
```
⟨ Header files of sf.c 71 ⟩ +≡
```
#include "uf.h"
```

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 $H{\rightarrow}N$, da seguinte forma:

$$slackness(H{\rightarrow}heap[i/2]) \leq slackness(H{\rightarrow}heap[i])$$

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 $a$), mas podemos definir a chave desta maneira devido ao modo como as arestas nos heaps de uma componente estão organizadas (um heap para as arestas que vão para componentes ativas e outro para as arestas que vão para componentes inativas).
```

76 ⟨ Auxiliary functions 76 ⟩ ≡

```
double reduced_len(void *edge)
{
  Arc *a = edge;
  if (¬(a→tip)) return −1;
  return ((double) a→len) − d(a→from) − d(a→tip);
}
```

Veja também blocos 92, 93, 95, 98, 108, 111, 115, 132, 134, 189, 225 e 226.

Este código é usado no bloco 1.

4.2 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. A função *sf_gw* recebe como argumentos um grafo g e um vetor *term_sets* de conjuntos de terminais, cujos elementos se encontram indexados de 0 a g→*num_ts*, e devolve um floresta de Steiner do grafo g. Abaixo, é apresentada uma visão panorâmica desta função.

77 ⟨ Steiner forest construction functions 77 ⟩ ≡

```
SteinerForest *sf_gw(Graph *g, TermSet *term_sets)
{
  ⟨ Local variables of sf_gw 79 ⟩
  ⟨ Create a spanning forest with no edges 78 ⟩
  ⟨ Initialize d(v) for each vertex v of the graph 82 ⟩
  ⟨ Construct a pair of edge heaps for each component 80 ⟩
  ⟨ While there is some active component include an edge in the forest 84 ⟩
  ⟨ Drop out unnecessary edges and return the resulting Steiner forest 96 ⟩
}
```

Veja também blocos 116 e 184.

Este código é usado no bloco 1.

Este bloco implementa a criação da floresta geradora inicial, a qual não possui arestas (passo correspondente a linha 1 do algoritmo). Durante a execução da fase iterativa do algoritmo (linhas

de 4 a 8), a floresta corrente será representada através da variável *sf*, a qual armazena um apontador para uma estrutura do tipo **Graph**, e 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.

Como muitas vezes, a partir de um dado vértice, precisaremos acessar a componente da qual ele faz parte, iremos definir aqui a macro `COMPONENT`(V), a qual associa a cada vértice V um apontador para a componente a qual V pertence.

78 #**define** *next_vertex* *y.V*

⟨ Create a spanning forest with no edges 78 ⟩ ≡

```
{
  int i, *sizes;
  GWComponent *t;
  Vertex *v;
  sf = gb_new_graph(g->n);
  for (v = sf->vertices; v < sf->vertices + sf->n; v++) {
    Vertex *w = g->vertices + (v - sf->vertices);
    v->name = gb_save_string(w->name);
  }
  components = t = malloc((g->n + 1) * sizeof(GWComponent));
  sizes = malloc((g->n) * (g->num_ts) * sizeof(int));
  for (i = 0; i < g->n; i++) {
    int j;
    GWComponent *C = components + i + 1;
    v = g->vertices + i;
    C->id = v->ind = i;
    SET_COMPONENT(v, C);
    C->vertices = v;
    v->next_vertex = Λ;
    C->num_vertices = 1;
    C->ts_intersect_size = sizes + i * (g->num_ts);
    for (j = 0; j < g->num_ts; j++) C->ts_intersect_size[j] = 0;
    C->active = C->disconnected = 0;
    if (v->termset_id ≥ 0) {
      C->active = 1;
      num_actives++;
      C->ts_intersect_size[v->termset_id]++;
```

```
        C→disconnected++;
      }
      t→next = C;
      C→prev = t;
      t = t→next;
    }
    t→next = components;
    components→prev = t;
    UFinit(g→n);
  }
```

Este código é usado no bloco 77.

Vamos declarar neste bloco algumas das variáveis locais de *sf_gw*: *sf*, *components* e *num_actives*. 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.

79 ⟨ Local variables of *sf_gw* 79 ⟩ ≡

```
  Graph *sf;
  GWComponent *components;
  long num_actives = 0;
```

Veja também blocos 81, 83 e 85.

Este código é usado no bloco 77.

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.

80 ⟨ Construct a pair of edge heaps for each component 80 ⟩ ≡

```
  {
    int j;
    GWComponent *t;
    FHnode *adj;
    FHEAPalloc(2 * g→n, g→m + g→n);
    allocItens(g→m);
    adj = malloc((g→n) * (g→n) * sizeof(FHnode));
```

```
    min_edge = malloc(sizeof(Arc));
    min_edge→tip = Λ;
    for (t = components→next, j = 0; t ≠ components; t = t→next, j++) {
      int i;
      Vertex *v;
      Arc *a;
      t→adj = adj + j * (g→n);
      for (i = 0; i < g→n; i++) t→adj[i] = Λ;
      t→edges[0] = FHEAPinit( );
      t→edges[1] = FHEAPinit( );
      v = t→vertices;
      for (a = v→arcs; a; a = a→next) {
        GWComponent *C = (GWComponent *) COMPONENT(a→tip);
        if (C→active)
          t→adj[C→id] = FHEAPinsert(t→edges[1], newItem(a, reduced_len, min_edge));
        else t→adj[C→id] = FHEAPinsert(t→edges[0], newItem(a, reduced_len, min_edge));
      }
    }
  }
```

Este código é usado no bloco 77.

81 ⟨ Local variables of *sf_gw* 79 ⟩ +≡

```
  Arc *min_edge;
```

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. O vetor global *d_value* guarda os valores de $d(v)$ para cada vértice v do grafo. Para acessar, neste vetor, o valor correspondente a um dado vértice v do grafo, utilizaremos o conteúdo de *v→d_index*, o qual corresponde ao índice de v no vetor *g→vertices*.

82 #**define** $d(V)$ *((**double** *)(V)→z.S)

⟨ Initialize $d(v)$ for each vertex v of the graph 82 ⟩ ≡

```
  {
    int i;
    d_value = malloc(g→n * sizeof(double));
    for (i = 0; i < g→n; i++) {
      Vertex *v = g→vertices + i;
```

```
    v→z.S = (char *)(d_value + i);
    d(v) = 0;
  }
}
```

Este código é usado no bloco 77.

83 ⟨ Local variables of *sf_gw* 79 ⟩ +≡
```
double *d_value;
```

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 *sf* até que não existam mais componentes ativas, o que é detectado através da variável *num_actives*. Cada vez que uma aresta *a* é incluída na floresta, atualiza-se o valor de $d(v)$ para cada vértice v em uma componente ativa e, além disso, é realizada a união das componentes `C1` e `C2` às quais pertencem, respectivamente, os vértices *a→from* e *a→tip*, extremos de *a*. Note que, após ser incluída na floresta, *a* é removida dos *heaps* de arestas de `C1` e `C2`, já que deixou de ser uma aresta externa.

Cada aresta incluída na floresta *sf* guardará um apontador para a aresta correspondente a ela no grafo *g*. Para isso, faremos uso do *utility field a* destas arestas, ao qual daremos o nome de *original_arc*.

84 ⟨ While there is some active component include an edge in the forest 84 ⟩ ≡
```
{
  dual_cost = 0;
  while (num_actives > 0) {
    double inc;
    Arc *a;
    GWComponent *C1, *C2;
    ⟨ Let a be an edge with the smallest slackness inc 86 ⟩
    ⟨ Include a in the forest sf 87 ⟩
    ⟨ Increment d(v) for each vertex v in some active component 88 ⟩
    dual_cost += inc * num_actives;
    C1 = (GWComponent *) COMPONENT(a→from);
    C2 = (GWComponent *) COMPONENT(a→tip);
    FHEAPdelete(C2→edges[1], C2→adj[C1→id]);
    C2→adj[C1→id] = Λ;
```

```
    if (C2→active) {
      FHEAPdelMin(C1→edges[1]);
      num_actives--;
    }
    else FHEAPdelMin(C1→edges[0]);
    C1→adj[C2→id] = Λ;
    ⟨ Merge the components C1 and C2 89 ⟩
  }
}
```

Este código é usado no bloco 77.

85 ⟨ Local variables of *sf_gw* 79 ⟩ +≡

```
double dual_cost;
```

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.

86 ⟨ Let a be an edge with the smallest slackness *inc* 86 ⟩ ≡

```
{
  Arc *b;
  GWComponent *C;
  for (C = components→next; C ≠ components; C = C→next) {
    if (C→active) {
      if (FHEAPempty(C→edges[0])) {
        if (FHEAPempty(C→edges[1])) return Λ;      /* o problema é inviável */
        a = getValue(FHEAPfindMin(C→edges[1]));
        inc = reduced_len(a)/2;
      }
      else {
        b = getValue(FHEAPfindMin(C→edges[0]));
        if (FHEAPempty(C→edges[1])) {
          a = b;
          inc = reduced_len(b);
        }
        else {
          a = getValue(FHEAPfindMin(C→edges[1]));
```

```
            if (reduced_len(a)/2 < reduced_len(b)) inc = reduced_len(a)/2;
            else {
              a = b;
              inc = reduced_len(b);
            }
          }
        }
        C = C→next;
        break;
      }
    }
    for ( ; C ≠ components; C = C→next) {
      if (C→active) {
        int empty = 0;
        if (¬FHEAPempty(C→edges[0])) {
          b = getValue(FHEAPfindMin(C→edges[0]));
          if (reduced_len(b) < inc) {
            inc = reduced_len(b);
            a = b;
          }
        }
        else empty++;
        if (¬FHEAPempty(C→edges[1])) {
          b = getValue(FHEAPfindMin(C→edges[1]));
          if (reduced_len(b)/2 < inc) {
            inc = reduced_len(b)/2;
            a = b;
          }
        }
        else if (empty) return Λ;      /* o problema é inviável */
      }
    }
  }
```

Este código é usado no bloco 84.

87 `#define original_arc a.A`

⟨ Include a in the forest sf 87 ⟩ ≡

```
{
  Vertex *u, *v;
  u = sf->vertices + (a->from - g->vertices);
  v = sf->vertices + (a->tip - g->vertices);
  gb_new_edge(u, v, a->len);
  u->arcs->original_arc = a;
  v->arcs->original_arc = (u < v) ? a + 1 : a - 1;
}
```

Este código é usado no bloco 84.

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 $d(v)$.

88 ⟨ Increment $d(v)$ for each vertex v in some active component 88 ⟩ ≡

```
{
  GWComponent *C;
  for (C = components->next; C ≠ components; C = C->next)
    if (C->active) {
      Vertex *v;
      for (v = C->vertices; v; v = v->next_vertex) d(v) += inc;
    }
}
```

Este código é usado no bloco 84.

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, `C1` e `C2` são apontadores para as estruturas das duas componentes a serem unidas. Após a união, `C1` estará apontando para a estrutura da componente resultante.

89 ⟨ Merge the components `C1` and `C2` 89 ⟩ ≡

```
{
  int active_ = 1;
  if (C1->num_vertices < C2->num_vertices) {
    GWComponent *t = C1;
```

```
    C1 = C2;
    C2 = t;
  }
  ⟨ Insert each vertex of C2 in C1 and set active_ to the new state of C1 90 ⟩
  ⟨ Alter the place of the edges incident to C1 if the state of C1 has changed 91 ⟩
  ⟨ Merge the edge heaps of C1 and C2 94 ⟩
  UFunion(C1→id, C2→id);
  C1→active = active_;
  C2→next→prev = C2→prev;
  C2→prev→next = C2→next;
}
```

Este código é usado no bloco 84.

A união dos conjuntos de vértices das duas componentes é feita incluíndo cada vértice de C2 na lista dos vértices de C1. 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).

90 ⟨ Insert each vertex of C2 in C1 and set *active_* to the new state of C1 90 ⟩ ≡

```
{
  Vertex *v = C2→vertices;
  while (v) {
    TermSet *ts;
    Vertex *z = v→next_vertex;
    v→next_vertex = C1→vertices;
    C1→vertices = v;
    C1→num_vertices++;
    ts = TERMSET(v);
    if (ts ∧ ¬(ts→connected)) {
      C1→ts_intersect_size[ts→id]++;
      if (C1→ts_intersect_size[ts→id] ≡ 1) {
        (C1→disconnected)++;
        if (C1→disconnected ≡ 1) active_ = 1;
      }
      else if (C1→ts_intersect_size[ts→id] ≡ ts→num_vertices) {
        ts→connected = 1;
        (C1→disconnected)--;
```

```
        if (¬(C1→disconnected)) active_ = 0;
      }
    }
    v = z;
  }
  if (¬active_) num_actives−−;
}
```

Este código é usado no bloco 89.

A organização dos *heaps* de arestas da componente resultante da união de `C1` e `C2` divide-se em duas partes: a troca de heap, se necessária, para as arestas que ligam `C1` às componentes da floresta que não são *adjacentes* a `C2` no grafo e a união dos *heaps* de arestas de `C1` e `C2`. 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 `C1` foi alterado após a inclusão dos vértices de `C2` em seu conjunto de vértices, será necessário trocar de heap as arestas para `C1` que se encontram nos *heaps* de cada componente C não adjacente a `C2` (também será necessário fazer isto para as arestas nos *heaps* de componentes adjacentes a `C2`, mas este caso será tratado no próximo bloco). E é exatamente isto que é feito pelo trecho de código abaixo.

91 ⟨ Alter the place of the edges incident to `C1` if the state of `C1` has changed 91 ⟩ ≡

```
if (C1→active ≠ active_) {
  int i;
  void *args[3];
  args[0] = C1;
  args[1] = C2;
  for (i = 0; i < 2; i++) FHEAPtraverse(C1→edges[i], flipIfNotAdjacent, args);
}
```

Este código é usado no bloco 89.

92 ⟨ Auxiliary functions 76 ⟩ +≡

```
void flipEdge(GWComponent *C1, GWComponent *C2)
{
  FibHeap cur = C1→edges[C2→active];
  FHnode node = C1→adj[C2→id];
  Item item = FHEAPdelete(cur, node);
  FibHeap new = C1→edges[1 − C2→active];
```

```
    C1→adj[C2→id] = FHEAPinsert(new, item);
  }
```

93 ⟨ Auxiliary functions 76 ⟩ +≡

```
  void flipIfNotAdjacent(Item item, void *args[])
  {
    GWComponent *C1 = args[0];
    GWComponent *C2 = args[1];
    Arc *a = getValue(item);
    GWComponent *C = (GWComponent *) COMPONENT(a→tip);
    if (¬(C2→adj[C→id])) flipEdge(C, C1);
  }
```

A união dos *heaps* das componentes `C1` e `C2` é feita da seguinte forma. Cada um dos *heaps* de arestas de `C2` é percorrido e, para cada aresta a, verifica-se se a componente C a qual a liga `C2` é adjacente a `C1`. Se C não for adjacente a `C1`, a é incluída em um dos *heaps* de `C1` (de acordo com o estado de C); caso contrário, já existe uma aresta b ligando `C1` a C e, desta forma, devemos decidir qual aresta, a ou b, guardar em um dos heaps de `C1`: 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 C.

94 ⟨ Merge the edge heaps of `C1` and `C2` 94 ⟩ ≡

```
  {
    int j;
    void *args[4];
    args[0] = C1;
    args[1] = C2;
    args[2] = &active_;
    for (j = 0; j < 2; j++) {
      FHEAPtraverse(C2→edges[j], insertIfBetter, args);
      FHEAPdestroy(C2→edges[j]);
    }
  }
```

Este código é usado no bloco 89.

95 ⟨ Auxiliary functions 76 ⟩ +≡

```
  void insertIfBetter(Item item, void *args[])
```

```
{
  GWComponent *C1 = args[0];
  GWComponent *C2 = args[1];
  int active_ = *((int *) args[2]);
  Arc *a = getValue(item);
  GWComponent *C = (GWComponent *) COMPONENT(a->tip);
  FibHeap h = C1->edges[C->active];
  if (¬(C1->adj[C->id])) {
    C1->adj[C->id] = FHEAPinsert(h, item);
    if (C2->active ≠ active_) flipEdge(C, C2);
    C->adj[C1->id] = C->adj[C2->id];
  }
  else {
    FHnode node = C1->adj[C->id];
    Item item_b = FHEAPdelete(h, node);
    Arc *b = getValue(item_b);
    if (reduced_len(a) < reduced_len(b)) {
      C1->adj[C->id] = FHEAPinsert(h, item);
      FHEAPdelete(C->edges[C1->active], C->adj[C1->id]);
      if (C2->active ≠ active_) flipEdge(C, C2);
      C->adj[C1->id] = C->adj[C2->id];
    }
    else {
      C1->adj[C->id] = FHEAPinsert(h, item_b);
      FHEAPdelete(C->edges[C2->active], C->adj[C2->id]);
      if (C1->active ≠ active_) flipEdge(C, C1);
    }
  }
  C->adj[C2->id] = Λ;
}
```

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 representada por esta variável. A floresta encontrada é então devolvida como resposta pela função *sf_gw*. A operação implementada por *edgePrunning* é conhecida como *limpeza das arestas* e será vista com detalhes mais adiante.

96 ⟨ Drop out unnecessary edges and return the resulting Steiner forest 96 ⟩ ≡

```
{
  SteinerForest *min_sf;
  min_sf = edgePruning(sf, g, term_sets);
  min_sf->dual_cost = dual_cost;
  ⟨ Free all the auxiliary memory allocated 97 ⟩
  return min_sf;
}
```

Este código é usado no bloco 77.

97 ⟨ Free all the auxiliary memory allocated 97 ⟩ ≡

```
{
  gb_recycle(sf);
  free((components + 1)->ts_intersect_size);
  free((components + 1)->adj);
  free(components);
  free(min_edge);
  free(d_value);
  UFdestroy( );
  FHEAPfree( );
  freeItens( );
}
```

Este código é usado no bloco 96.

4.3 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 apontadores *sf_graph* e *g* para estruturas do tipo **Graph**, onde *sf_graph* representa uma floresta de Steiner no grafo *g*, e um vetor *term_sets* de conjuntos de terminais do grafo *g*, cujos elementos se encontram indexados de 0 a *g→num_ts*, e devolve um apontador para uma estrutura **SteinerForest** correspondente a uma floresta de Steiner minimal contida em *sf_graph*.

98 ⟨ Auxiliary functions 76 ⟩ +≡

```
⟨ More auxiliary functions 99 ⟩
SteinerForest *edgePruning(Graph *sf_graph, Graph *g, TermSet *term_sets)
{
  SteinerForest *sf;
  ⟨ Find the lca of each terminal set 100 ⟩
  ⟨ Determine the necessary edges with the aid of the lcas 103 ⟩
}
```

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

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

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

```
99 #define parent_edge u.A /* v→parent_edge→tip is the parent of v */
   #define next_item v.V /* next vertex in the FIFO queue */
   #define IN_LIST(TS_ID) (f→vertices + TS_ID)→x.I
   ⟨ More auxiliary functions 99 ⟩ ≡
     TermSet *initialize_parent(Vertex *r, TermSet *term_sets, Graph *g, Graph *f)
     {
       Arc *a;
       Vertex *head, *tail;
       TermSet x, *l;
       int i;
       l = &x;
       l→next_tset = Λ;
       for (i = 0; i < g→num_ts; i++) IN_LIST(i) = 0;
       a = (Arc *) malloc(sizeof(Arc));
       a→tip = r;
       r→parent_edge = a;
       r→next_item = Λ;
       head = tail = r;
       while (head) {
         Arc *t = Λ;
         Vertex *v = head, *v_;
         TermSet *ts;
         for (a = v→arcs; a; a = a→next) {
           Vertex *w = a→tip;
           if (w→parent_edge) { /* w is the parent of v */
             if (t) t→next = a→next;
             else v→arcs = a→next;
             continue;
           }
           w→parent_edge = (v < w) ? (a + 1) : (a − 1);
           w→next_item = Λ;
           tail→next_item = w;
           tail = tail→next_item;
           t = a;
         }
         v_ = g→vertices + (v − f→vertices);
         ts = TERMSET(v_);
         if (ts ∧ ¬IN_LIST(v_→termset_id)) {
```

```
        ts→next_tset = l→next_tset;
        l→next_tset = ts;
        IN_LIST(v_→termset_id) = 1;
      }
      head = head→next_item;
    }
    return l→next_tset;
  }
```

Veja também bloco 104.

Este código é usado no bloco 98.

Aqui, inicialmente é feito o pré-processamento das árvores em *sf_graph* para o cálculo dos vértices ancestrais. Para cada árvore T da floresta, aplicamos a função *initialize_parent* apresentada no bloco anterior, tornando T uma árvore enraizada. Feito isso, podemos, então, utilizar a função *lca_preprocessing* que implementa a fase de pré-processamento do algoritmo para o cálculo do ancestral commum mais próximo. Esta função pré-processa uma dada árvore enraizada de tal forma que as consultas pelo ancestral comum mais próximo de dois vértices quaisquer na árvore podem ser respondidas de maneira rápida (em tempo constante). Por fim, obtemos o ancestral comum mais próximo de cada conjunto de terminais que se encontra em T. Após a execução do trecho de código abaixo, para cada vértice v no grafo g, v→*head_id* conterá o identificador do conjunto de terminais que se encontra na primeira posição da lista ligada de conjuntos de terminais dos quais v é ancestral comum mais próximo; caso não existam conjuntos de terminais que tenham v como ancestral comum mais próximo, v→*head_id* será igual a -1.

```
100 #define head_id  w.I
⟨ Find the lca of each terminal set 100 ⟩ ≡
  {
    Vertex *v;
    for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
      Vertex *u;
      v→parent_edge = Λ;
      u = g→vertices + (v − sf_graph→vertices);
      u→head_id = −1;
    }
    for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
      TermSet *ts_list;
      if (v→parent_edge) continue;
```

```
    ts_list = initialize_parent(v, term_sets, g, sf_graph);
    if (¬ts_list) continue;
    lca_preprocessing(v);
    ⟨ find the lca of each terminal set in the list ts_list 101 ⟩
  }
}
```

Este código é usado no bloco 98.

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

```
101 #define TS_LIST(V)  ((V→head_id < 0) ? Λ : (term_sets + V→head_id))
⟨ find the lca of each terminal set in the list ts_list 101 ⟩ ≡
  {
    TermSet *ts = ts_list;
    while (ts) {
      TermSet *t = ts→next_tset;
      Vertex *v, *w, *w_;
      w = ts→vertices;
      w_ = sf_graph→vertices + (w − g→vertices);
      for (v = w→next_terminal; v; v = v→next_terminal) {
        Vertex *v_ = sf_graph→vertices + (v − g→vertices);
        w_ = lca(w_, v_);
      }
      w = g→vertices + (w_ − sf_graph→vertices);
      ts→next_tset = TS_LIST(w);
      w→head_id = ts→id;
      ts = t;
    }
  }
```

Este código é usado no bloco 100.

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.

102 ⟨ Header files of sf.c 71 ⟩ +≡
```
#include "lca.h"
```

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

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

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

103
```
#define visit  y.I
#define next_lca  x.V
#define selected  b.I
```
⟨ Determine the necessary edges with the aid of the lcas 103 ⟩ ≡
```
{
  Arc *t, x;
  Vertex l, *v, *w;
  sf = (SteinerForest *) malloc(sizeof(SteinerForest));
  sf→edges = t = &x;
  sf→cost = 0;
  for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
    v→visit = 0;
    v→parent_edge→selected = 0;
  }
  w = &l;
  w→next_lca = Λ;
  for (v = sf_graph→vertices; v < sf_graph→vertices + sf_graph→n; v++) {
```

```
    if (v→parent_edge→tip ≠ v) continue;
    lca_preorderlist(v, g, sf_graph, &w);
    for (w = l.next_lca; w; w = w→next_lca) {
      TermSet *ts;
      for (ts = TS_LIST(w); ts; ts = ts→next_tset) {
        Vertex *z;
        for (z = ts→vertices; z; z = z→next_terminal)
          ⟨ select the edges in the path from z to w in sf_graph 105 ⟩
      }
    }
    w = &l;
    w→next_lca = Λ;
  }
  t→next_edge = Λ;
  sf→edges = sf→edges→next_edge;
  return sf;
}
```

Este código é usado no bloco 98.

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 $u\text{→}ts\_list \neq \Lambda$, 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).

104 ⟨ More auxiliary functions 99 ⟩ +≡

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

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

105 ⟨ select the edges in the path from z to w in *sf_graph* 105 ⟩ ≡

```
{
  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->original_arc;
    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 103.

Capítulo 5

Implementação de Cole, Hariharan, Lewenstein e Porat

Para grafos não muito densos, existem implementações que apresentam um melhor desempenho que a implementação de Goemans e Williamson descrita acima. Um exemplo é a implementação proposta por Cole, Hariharan, Lewenstein e Porat [5], 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 precisaria 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: para impedir que uma aresta externa tenha ambos os extremos em componentes ativas da floresta, toda aresta externa é dividida ao meio em duas arestas de igual custo, 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. Após o número máximo de subdivisões de uma aresta original ter sido atingido, ao menos uma (e no máximo duas) das arestas resultantes destas subdivisões será uma aresta externa. Chamaremos uma tal aresta de *aresta terminal*. Note que uma aresta terminal não pode ser subdividida, pois isto resultaria em um número de subdivisões maior que o permitido para a aresta original correspondente. Desta forma, é possível que, em algum momento, existam arestas terminais com ambos os extremos em componentes ativas da floresta. Mesmo que isto ocorra, tais arestas serão tratadas como se apenas um de seus extremos fizesse parte de uma componente ativa. Isso introduz um erro no algoritmo, o qual é responsável por uma degradação de $\frac{1}{n^k}$ no fator de aproximação.

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

5.1 Descrição da implementação

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Estruturas de dados

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

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

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

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

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

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

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

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

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

106 ⟨ Data structures of sf.c 68 ⟩ +≡

```
typedef struct edge_fields_struct {
  int curr_len;
  int d;
  int duplicate;
  FHnode node;
} EdgeFields;
```

Para cada arco a, `FIELDS`(a) será um apontador para a estrutura **EdgeFields** que armazena os campos de a. Utilizaremos também a macro `SET_FIELDS`(a, ef), a qual nos será útil para associar ao arco a os campos correspodentes a estrutura **EdgeFields** apontada por ef.

107
```
#define FIELDS(A)  ((EdgeFields *)((A)->b.S))
#define SET_FIELDS(A,Y)  ((A)->b.S = (char *)(Y))
```

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

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

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

108 ⟨ Auxiliary functions 76 ⟩ +≡
```
double edge_piece_key(void *p)
{
  Arc *a = p;
  if (¬(a->tip)) return -1;
  return FIELDS(a)->curr_len - FIELDS(a)->d;
}
```

109 ⟨ Header files of sf.c 71 ⟩ +≡
```
#include <limits.h>
```

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

110 ⟨ Data structures of sf.c 68 ⟩ +≡
```
typedef struct labeled_heap *LbHeap;
struct labeled_heap {
  int d;
  FibHeap heap;
};
```

111 ⟨ Auxiliary functions 76 ⟩ +≡
```
double lb_heap_key(void *p)
{
  LbHeap LH = p;
  if (¬FHEAPempty(LH→heap)) {
    Item item = FHEAPfindMin(LH→heap);
    Arc *a = getValue(item);
    return edge_piece_key(a) − LH→d;
  }
  return −1;
}
```

112 ⟨ Data structures of sf.c 68 ⟩ +≡
```
typedef struct chlp_component {
  int active;
  int num_vertices;
  LbHeap edges;
  BalBST ts_intersect_size;
  int disconnected;
} CHLPComponent;
```

113 ⟨ Header files of sf.c 71 ⟩ +≡
```
#include "bbst.h"
```

114 ⟨ Data structures of sf.c 68 ⟩ +≡
```
typedef struct i_struct *Intersection;
struct i_struct {
  int ts_id;
  int n;
};
```

115 ⟨ Auxiliary functions 76 ⟩ +≡
```
double I_key(void *p)
{
  Intersection I = p;
  return I->ts_id;
}
```

5.3 Função principal

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

116 ⟨ Steiner forest construction functions 77 ⟩ +≡
```
SteinerForest *sf_chlp(Graph *g, TermSet *term_sets, int k)
{
  ⟨ Local variables of sf_chlp 119 ⟩
  ⟨ Multiply by n^k the cost of each edge in g 117 ⟩
  ⟨ For each edge a in g split a in two pieces 120 ⟩
  ⟨ Create the initial spanning forest 122 ⟩
  ⟨ Include an edge piece in the forest while there is some active component 126 ⟩
  ⟨ Restore the original costs of the edges of g 118 ⟩
```

```
    ⟨ Discard unnecessary edges and return the resulting Steiner forest 138 ⟩
  }
```

117 ⟨ Multiply by n^k the cost of each edge in g 117 ⟩ ≡

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

Este código é usado no bloco 116.

118 ⟨ Restore the original costs of the edges of g 118 ⟩ ≡

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

Este código é usado no bloco 116.

119 ⟨ Local variables of *sf_chlp* 119 ⟩ ≡

```
  unsigned long int nk;
```

Veja também blocos 121, 123, 125 e 127.

Este código é usado no bloco 116.

O trecho de código abaixo inicializa os campos correspondentes a cada arco do grafo g. Cada aresta do grafo original é dividida ao meio, sendo que cada metade é representada pelos arcos

na estrutura de g que correspondem à aresta. Como estaremos trabalhando apenas com inteiros, uma das metades terá custo $\lfloor \frac{c_a}{2} \rfloor$ e a outra, $\lceil \frac{c_a}{2} \rceil$, onde c_a é o valor que se encontra guardado em $a\rightarrow len$.

120 ⟨ For each edge a in g split a in two pieces 120 ⟩ ≡

```
{
  Arc *a;
  Vertex *v;
  int i = 0;

  ef = malloc(g→m * sizeof (EdgeFields));
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next) {
      ef[i].curr_len = −1;
      ef[i].d = 0;
      ef[i].duplicate = 0;
      SET_FIELDS(a, ef + i);
      i++;
    }
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next)
      if (FIELDS(a)→curr_len ≡ −1) {
        Arc *a_ = a + 1;

        if (a→len % 2) {
          FIELDS(a)→curr_len = (a→len)/2 + 1;
          FIELDS(a_)→curr_len = (a→len)/2;
        }
        else {
          FIELDS(a)→curr_len = (a→len)/2;
          FIELDS(a_)→curr_len = (a→len)/2;
        }
      }
}
```

Este código é usado no bloco 116.

121 ⟨ Local variables of *sf_chlp* 119 ⟩ +≡

```
EdgeFields *ef;
```

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

122 ⟨ Create the initial spanning forest 122 ⟩ ≡

```
{
  int i, j = 0;
  Vertex *v;

  sf = gb_new_graph(g→n);
  for (v = sf→vertices; v < sf→vertices + sf→n; v++) {
    Vertex *w = g→vertices + (v − sf→vertices);

    v→name = gb_save_string(w→name);
  }
  num_actives = active_d = 0;
  for (i = 0; i < g→num_ts; i++) num_actives += term_sets[i].num_vertices;
  intersect = malloc(num_actives * sizeof(struct i_struct));
  allocItens(g→m + g→n + num_actives);
  BBSTalloc(num_actives + g→num_ts − 1);
  components = malloc(g→n * sizeof(CHLPComponent));
  for (i = 0; i < g→n; i++) {
    CHLPComponent *C = components + i;
    Vertex *v = g→vertices + i;

    v→ind = i;
    SET_COMPONENT(v, C);
    C→num_vertices = 1;
    C→ts_intersect_size = BBSTinit( );
    C→active = C→disconnected = 0;
    if (v→termset_id ≥ 0) {
      Item y;
      BalBST t = C→ts_intersect_size;
      Intersection I = intersect + j++;

      I→ts_id = v→termset_id, I→n = 1;
```

```
        C→ts_intersect_size = BBSTinsert(t, newItem(I, I_key, Λ), &y);
        C→active = 1;
        C→disconnected++;
      }
    }
    UFinit(g→n);
    ⟨ Create an edge heap for each component 124 ⟩
  }
```

Este código é usado no bloco 116.

123 ⟨ Local variables of *sf_chlp* 119 ⟩ +≡

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

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

124 ⟨ Create an edge heap for each component 124 ⟩ ≡

```
  {
    int i;
    FHEAPalloc(g→n + 2, g→n + g→m);
    active_heaps = FHEAPinit( );
    h = malloc(g→n * sizeof(struct labeled_heap));
    node = malloc(g→n * sizeof(FHnode));
    min_edge_piece = malloc(sizeof(Arc));
    min_edge_piece→tip = Λ;
    min_lb_heap = malloc(sizeof(struct labeled_heap));
    min_lb_heap→heap = FHEAPinit( );
    for (i = 0; i < g→n; i++) {
      Arc *a;
      CHLPComponent *C = components + i;
      Vertex *v = g→vertices + i;
```

```
    h[i].d = 0;
    h[i].heap = FHEAPinit( );
    for (a = v→arcs; a; a = a→next) FIELDS(a)→node = FHEAPinsert(h[i].heap, newItem(a,
        edge_piece_key, min_edge_piece));
    C→edges = h + i;
    if (C→active)
      node[i] = FHEAPinsert(active_heaps, newItem(C→edges, lb_heap_key, min_lb_heap));
  }
}
```

Este código é usado no bloco 122.

125 ⟨ Local variables of *sf_chlp* 119 ⟩ +≡

```
FibHeap active_heaps;
LbHeap h, min_lb_heap;
FHnode *node;
Arc *min_edge_piece;
```

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

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

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

⟨ Include an edge piece in the forest while there is some active component 126 ⟩ ≡

```
{
  dual_cost = 0;
  while (num_actives > 0) {
    Arc *uv;
    CHLPComponent *U, *V;
    Item item_uv, item_h_u;
    ⟨ Set uv to point to an edge with the smallest slackness 129 ⟩
    dual_cost += num_actives * RLEN(uv);
    active_d += RLEN(uv);
    FHEAPdelMin(U→edges→heap);      /* delete uv from U→edges */
```

```
      FHEAPdelMin(active_heaps);      /* delete U→edges from the active_heaps */
      if (FIELDS(uv)→curr_len ≤ (uv→len)/nk ∧ FIELDS(uv)→duplicate) {
          /* uv is a terminal edge and RLEN(uv) != slackness(uv) */
        Arc *vu = (uv→from < uv→tip) ? uv + 1 : uv − 1;
        ⟨ Include uv in the spanning forest sf 128 ⟩
        FHEAPdelete(V→edges→heap, FIELDS(vu)→node);
        if (V→active) {
          FHEAPdelete(active_heaps, node[ID(V)]);
          num_actives−−;
        }
        ⟨ Unite the components U and V 130 ⟩
      }
      else {
        Arc *wv = (uv→from < uv→tip) ? uv + 1 : uv − 1;
        ⟨ Split the edge wv 135 ⟩
      }
    }
  }
```

Este código é usado no bloco 116.

127 ⟨ Local variables of *sf_chlp* 119 ⟩ +≡

```
  double dual_cost;
```

128 ⟨ Include *uv* in the spanning forest *sf* 128 ⟩ ≡

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

Este código é usado no bloco 126.

Em cada iteração, a aresta escolhida para ser incluída na floresta corrente é uma aresta externa que minimiza o valor de *key*( ). Como iremos permitir que o heap de arestas de cada componente

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

129 ⟨ Set uv to point to an edge with the smallest slackness 129 ⟩ ≡

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

Este código é citado no bloco 135.

Este código é usado no bloco 126.

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

das arestas no heap da componente, até que ela eventualmente seja unida a uma componente ativa. Caso a componente resultante seja ativa, seu heap de arestas é inserido em *active_heaps*.

```
130 ⟨ Unite the components U and V 130 ⟩ ≡
  {
    int active_ = 1;
    if (U→num_vertices < V→num_vertices) {
      CHLPComponent *t = U;
      U = V;
      V = t;
    }
    ⟨ Include each vertex of V in U and set active_ to the new state of U 131 ⟩
    ⟨ Unite the edge heaps of U and V 133 ⟩
    if (¬(U→active) ∧ active_) U→edges→d −= active_d;
    U→active = active_;
    if (U→active) node[ID(U)] = FHEAPinsert(active_heaps, item_h_u);
    else {      /* U and V were active components */
      U→edges→d += active_d;
      num_actives−−;
    }
  }
```

Este código é usado no bloco 126.

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.

```
131 ⟨ Include each vertex of V in U and set active_ to the new state of U 131 ⟩ ≡
  {
    void *args[3];
    UFunion(UFfind(uv→from→ind), UFfind(uv→tip→ind));
    U→num_vertices += V→num_vertices;
    V→num_vertices = −1;
    args[0] = term_sets;
    args[1] = U;
    args[2] = &active_;
    BBSTtraverse(V→ts_intersect_size, insertIfDisconnected, args);
    BBSTdestroy(V→ts_intersect_size);
  }
```

Este código é usado no bloco 130.

132 ⟨ Auxiliary functions 76 ⟩ +≡

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

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

133 ⟨ Unite the edge heaps of U and V 133 ⟩ ≡

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

Este código é usado no bloco 130.

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

134 ⟨ Auxiliary functions 76 ⟩ +≡

```
void inc_field_d(Item item, void *args[])
{
  Arc *a = getValue(item);
  int inc = *((int *) args[0]);
  FIELDS(a)→d += inc;
}
```

Enquanto a aresta uv escolhida no bloco ⟨ Set uv to point to an edge with the smallest slackness 129 ⟩ não for a última fração terminal da aresta original, esta poderá ser subdividida mais algumas vezes. É exatamente isto que é feito pelo trecho de código abaixo. A aresta wv corresponde a fração da aresta original que ainda não foi selecionada. Se possível, esta fração será subdividida até obtermos duas arestas externas ou uma única aresta terminal. No primeiro caso, uv e wv representarão as duas arestas externas geradas, as quais possuem um s-vértice virtual v como extremo comum; no segundo caso, uv e wv estarão representando a mesma aresta terminal.

135 ⟨ Split the edge wv 135 ⟩ ≡

```
{
  CHLPComponent *W;
  int cur_len, new_len, d_w, inc, p;
  W = (CHLPComponent *) COMPONENT(wv→from);
  inc = W→active ? W→edges→d + active_d : W→edges→d;
```

```
d_w = FIELDS(wv)→d + inc;
new_len = cur_len = FIELDS(wv)→curr_len;
if (cur_len ≤ (wv→len)/nk)      /* wv is a terminal edge */
  ⟨ Set uv and wv to represent the same terminal edge 136 ⟩
else {
  do {
    p = new_len % 2;
    new_len = p ? new_len/2 + 1 : new_len/2;
  } while (new_len > (wv→len)/nk ∧ cur_len − new_len ≤ d_w);
  if (cur_len − new_len > d_w) {
    ⟨ Set uv and wv to represent two external edges 137 ⟩
    FHEAPdecreaseKey(W→edges→heap, FIELDS(wv)→node, Λ);
    if (W→active) FHEAPdecreaseKey(active_heaps, node[ID(W)], Λ);
  }
  else ⟨ Set uv and wv to represent the same terminal edge 136 ⟩
}
FIELDS(uv)→node = FHEAPinsert(U→edges→heap, item_uv);
node[ID(U)] = FHEAPinsert(active_heaps, item_h_u);
}
```

Este código é usado no bloco 126.

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

136 ⟨ Set uv and wv to represent the same terminal edge 136 ⟩ ≡

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

Este código é usado no bloco 135.

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.

137 ⟨ Set uv and wv to represent two external edges 137 ⟩ ≡

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

Este código é usado no bloco 135.

138 ⟨ Discard unnecessary edges and return the resulting Steiner forest 138 ⟩ ≡

```
{
  SteinerForest *min_sf;
  min_sf = edgePruning(sf, g, term_sets);
  min_sf→dual_cost = dual_cost;
  ⟨ Free the auxiliary memory allocated 139 ⟩
  return min_sf;
}
```

Este código é usado no bloco 116.

139 ⟨ Free the auxiliary memory allocated 139 ⟩ ≡

```
{
  free(ef);
  gb_recycle(sf);
  free(node);
  free(h);
  free(components);
  free(min_edge_piece);
  free(min_lb_heap);
  free(intersect);
  BBSTfree( );
  FHEAPfree( );
  freeItens( );
}
```

Este código é usado no bloco 138.

Capítulo 6

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 da 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.

A definição da estrutura de dados é feita da seguinte forma. Dado um grafo orientado G, com custos nos arcos, e um número fixo C, iremos atribuir, a cada vértice de G, um número em $\{1, \ldots, C\}$. Para cada vértice $v \in V_G$, denotaremos por $cat(v)$ o número atribuído a v, ao qual iremos nos referir como a *categoria* de v. Desta forma, cada arco uv do grafo G fica associado a um par ordenado $(cat(u), cat(v))$. Daremos o nome de *bicategoria* ao conjunto dos arcos associados a um dado par ordenado de categorias. 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 à bicategoria b.
- $findMin(b)$, onde b é uma bicategoria. Esta operação encontra um arco que apresente o menor custo dentre todos aqueles pertencentes à bicategoria b.
- $changeCategory(v, c)$, onde $v \in V_G$ e c é uma das C categorias. Esta operação atribui a categoria c ao vértice v, mudando implicitamente de bicategoria todos os arcos incidentes em v.
- $contractEdge(a, c)$, onde $a \in E_G$ e c é uma das C categorias. Esta operação contrai o arco a, transformando-o em um novo vértice do grafo e atribuindo, a este vértice, a categoria c. Tal operação remove a de E_G.

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

6.1 Estruturas de dados correspondentes às bicategorias

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

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

- *decreaseCost*(b, δ): a fim de realizar de maneira rápida esta operação, associaremos, a cada um dos heaps e a cada um dos arcos correspondentes a uma bicategoria b, um número real, de maneira que o custo de cada arco a em b seja o resultado de uma soma de três parcelas: o número real associado a a, o número real associado ao heap em $H(b)$ que contém a e o número real associado a $H(b)$. Denotaremos por Δ_x o número real associado a x, onde x pode ser um arco ou um heap. Note que, de fato, o uso desta estratégia nos permite, de

maneira rápida, reduzir de um dado valor δ o custo de todos os arcos que se encontram em uma bicategoria b: para isto, basta subtrair δ de $\Delta_{H(b)}$. Assim, o tempo gasto em cada execução da operação *decreaseCost* é $O(1)$.

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

- *changeCategory*(v, c): como já vimos, esta operação atribui uma nova categoria c ao vértice v, alterando, desta forma, a bicategoria em que cada um dos arcos incidentes em v se encontra. Assim, se c_0 era a categoria anteriormente atribuída a v, para cada categoria c', tudo que precisamos fazer é mover os arcos que têm a ponta final em v e que se encontram na bicategoria (c', c_0) para a bicategoria (c', c) e, simetricamente, mover os arcos que têm a ponta inicial em v e que estão em (c_0, c') para (c, c'). O único cuidado que devemos tomar é o de, inicialmente, executar o processo acima para todo $c' \neq c_0$ e, somente após isso, executá-lo para $c' = c_0$ (acho que isto não é necessário).

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

tempo gasto no total com esse processo é $O(\log n)$ vezes o número de arcos contidos nas listas $extra^-(v)$ e $extra^+(v)$ e, desta forma, é fundamental que adotemos uma estratégia de atribuição de arcos aos vértices que garanta que o número de arcos nestas duas listas não seja muito grande.

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

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

portanto, cada descarte pode ser realizado em tempo $O(\log n)$. Logo, o tempo gasto no total para descartar arcos em qualquer seqüência de operações *changeCategory* é $O(m \log n)$. Deste modo, para executar uma seqüência de r operações *changeCategory*, o tempo gasto no total é $O(r\sqrt{m} \log n + m \log n)$.

- *contractEdge*(uv, c): inicialmente, o arco uv a ser contraído é removido do heap em que se encontra e da lista $extra^+$ ou $extra^-$ à qual pertence. Em seguida, mudamos a categoria dos vértices u e v para c, através de duas chamadas à operação *changeCategory*. Por fim, para cada bicategoria b, fazemos a união dos heaps correspondentes aos vértices u e v em $H(b)$ e, feito isso, realizamos a união das listas $extra^+(u)$ e $extra^+(v)$ e das listas $extra^-(u)$ e $extra^-(v)$.

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

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

  Definiremos o valor de Δ_h como sendo igual a Δ_{h_1} e, além disso, atualizaremos adequadamente o valor de Δ_a, para cada arco a em h_2, de modo que o custo dos arcos em h_2 não sofra alteração após a união. Observe que o uso desta estratégia garante que, em qualquer seqüência de operações *contractEdge*, o número de vezes que atualizaremos Δ_a, para cada arco a, é $O(\log n)$ e, desta forma, o tempo gasto com essas atualizações, em qualquer seqüência de operações *contractEdge*, é $O(m \log n)$.

  Também é importante ressaltar que pode haver a necessidade de alterarmos os vértices aos quais arcos que possuíam a ponta inicial em u ou em v se encontram atribuídos. Isso irá acontecer quando o vértice w resultante da contração da aresta uv tiver grau de saída alto e o mesmo não ocorrer para pelo menos um vértice dentre u e v (note que o grau de saída de w é dado pela soma dos graus de saída de u e de v, de acordo com o modo como definimos o grau de saída de um vértice a alguns parágrafos atrás). Neste caso, para cada vértice $x \in \{u, v\}$ que não possuir grau de saída alto, deveremos percorrer a lista $extra^-(x)$ e, para cada arco xy visitado neste percurso, deveremos remover xy do heap $H^+(y, b)$, onde b é a bicategoria a que xy pertence, inserir xy em $extra^+(y)$, inserir xy no heap $H^-(w, b)$ e, por fim, remover xy de $extra^-(w)$. Como o número de arcos para os quais executaremos o procedimento descrito acima é $O(\sqrt{m})$ (pois, como já vimos, para todo vértice x, o número de arcos em $extra^-(x)$ nunca ultrapassa $\sqrt{m}$), o tempo gasto no total por cada operação *contractEdge* para mudar a atribuição de arcos aos vértices é $O(\sqrt{m} \log n)$.

Por tudo que vimos acima, podemos concluir que o tempo gasto no total para se executar uma seqüência de r operações *contractEdge* é $O(r\sqrt{m}\log n + m\log n)$.

6.2 Descrição da implementação

Como o grafo G da entrada do 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.

O algoritmo é dividido em duas fases. Ao final da execução da primeira fase, obteremos uma lista F de arcos correspondentes às arestas de uma floresta de Steiner no grafo G da entrada. Tal lista será construída iterativamente, sendo que, em cada iteração, os arcos contidos nesta lista corresponderão às arestas de uma floresta geradora no grafo G. Inicialmente, F estará vazia. Em cada iteração da primeira fase, iremos selecionar um arco a do grafo corrente para ser inserido em F. Após a inserção de a em F, iremos contraí-lo, o que dará origem a um novo vértice que substituirá a e os dois vértices em seus extremos no grafo; em seguida, terá início uma nova iteração. Desta forma, no início de cada iteração, cada vértice do grafo corrente G' corresponderá a uma componente da floresta induzida por F no grafo G original. A categoria associada a cada vértice v de G' estará relacionada ao estado de atividade da componente correspondente a v: se v corresponde a uma componente ativa da floresta induzida por F 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).

Ao escolher o arco a ser inserido em F em cada iteração, não serão levados em conta laços, ou seja, arcos que possuem os dois extremos em um mesmo vértice. A justificativa para este critério é simples: tais arcos correspondem a arestas que possuem os dois extremos em uma mesma componente da floresta induzida por F no grafo G. Naturalmente, os demais arcos do grafo G' correspondem às arestas externas de G em relação à floresta induzida por F e, dentre eles, será escolhido, para ser inserido em F, um arco que corresponda a uma aresta externa de folga mínima. Para isto, associaremos, a cada arco $a \in E_{G'}$, um valor $\hat{c}_a$ que chamaremos de *custo reduzido* de a, definido como $c_{uv} - d(u) - d(v)$, onde uv é a aresta de E_G correspondente ao arco a. Desta forma, para cada arco $a \in E_{G'}$ que não é um laço, a folga da aresta externa correspondente a a é dada por $\frac{\hat{c}_a}{2}$, caso a seja um arco pertencente à bicategoria (ATIVO, ATIVO), e por $\hat{c}_a$, caso a pertença à bicategoria (INATIVO, ATIVO) ou à bicategoria (ATIVO, INATIVO). Assim, o arco a ser inserido em F pode ser selecionado, comparando-se a folga das arestas correspondentes aos arcos que apresentam custo reduzido mínimo nas bicategorias (ATIVO, ATIVO), (ATIVO, INATIVO) e (ATIVO, INATIVO). Os arcos

pertencentes à bicategoria (INATIVO, INATIVO) nunca serão examinados pois correspondem a arestas com os dois extremos em componentes inativas da floresta induzida por F em G. Após a escolha do arco a a ser incluído em F, deveremos decrementar de $2 \cdot folga(a')$, onde a' é a aresta em E_G que corresponde a a, o custo reduzido dos arcos em (ATIVO, ATIVO) e de $folga(a')$ o custo reduzido daqueles que se encontram em (ATIVO, INATIVO) e (INATIVO, ATIVO); isto equivale a incrementar de $folga(a')$ o valor de $d(v)$ para cada vértice v em uma componente ativa da floresta induzida por F em G. Por fim, note que, no início da primeira iteração, para todo arco $a \in E_{G'}$, temos que o custo reduzido de a é igual ao custo da aresta correspondente a a em E_G.

A partir do momento em que $cat(v) =$ INATIVO para todo vértice v do grafo corrente G', a floresta F_0 induzida em G por F será uma floresta de Steiner e, deste modo, chega ao fim a primeira fase do algoritmo. A segunda fase consiste em encontrar uma floresta de Steiner minimal F_1 contida em F_0 e, ao final da execução desta fase, F_1 é devolvida como resposta pelo algoritmo. Agora, estamos aptos para descrever o algoritmo de Goemans e Williamson em termos das operações sobre bicategorias definidas anteriormente. É importante ressaltar que, no algoritmo abaixo, as operações *findMin* e *decreaseCost* são executadas sobre as bicategorias levando-se em conta o custo reduzido dos arcos que delas fazem parte. Assim, por exemplo, após a execução da linha 7 do algoritmo, a_1 será um arco tal que $\hat{c}_{a_1}$ é mínimo, considerando o custo reduzido de todos os arcos em (ATIVO, ATIVO) e, após a execução da linha 11, o custo reduzido de todos os arcos em (ATIVO, ATIVO) sofrerá um decréscimo de $2 \cdot folga(\grave{a}^G)$. Denotaremos por a^G a aresta do grafo G correspondente a um dado arco $a \in E_{G'}$.

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

1 seja G' o grafo orientado obtido através de uma orientação arbitrária das arestas em E_G
2 $\hat{c}_a \leftarrow c_{a^G}$ para cada a em $E_{G'}$
3 $cat(v) \leftarrow$ ATIVO para cada v em algum $T \in \mathcal{R}$
4 $cat(v) \leftarrow$ INATIVO para cada vértice de Steiner v
5 $F \leftarrow \emptyset$
6 enquanto G' tem algum vértice v tal que $cat(v) =$ ATIVO faça
7 $\quad a_1 \leftarrow findMin$(ATIVO, ATIVO)
8 $\quad a_2 \leftarrow findMin$(ATIVO, INATIVO)
9 $\quad a_3 \leftarrow findMin$(INATIVO, ATIVO)
10 $\quad$ escolha $\dot{a} \in \{a_1, a_2, a_3\}$ tal que $folga(\dot{a}^G) = \min\{folga(a_j^G) \mid j \in \{1,2,3\}\}$
11 $\quad decreaseCost((\text{ATIVO, ATIVO}), 2 \cdot folga(\dot{a}^G))$
12 $\quad decreaseCost((\text{ATIVO, INATIVO}), folga(\dot{a}^G))$
13 $\quad decreaseCost((\text{INATIVO, ATIVO}), folga(\dot{a}^G))$
14 $\quad F \leftarrow F \cup \{\dot{a}\}$
15 $\quad$ seja H a componente à qual $\dot{a}^G$ pertence na floresta induzida por F em G
16 $\quad$ se H é ativa
17 $\quad\quad contractEdge(\dot{a},$ ATIVO)
18 $\quad$ senão
19 $\quad\quad contractEdge(\dot{a},$ INATIVO)
20 seja F_0 a floresta de Steiner induzida por F no grafo G
21 seja F_1 uma $\mathcal{R}$-floresta minimal contida em F_0
22 devolva F_1

Como, em cada iteração, o algoritmo executa um número constante de operações sobre bicategorias e, no máximo, são realizadas $n-1$ iterações, temos que, no total, são executadas $O(n)$ operações sobre bicategorias. Desta forma, o tempo gasto no total pelo algoritmo com estas operações é $O(n\sqrt{m}\log n + m\log n)$, ou seja, $O(n\sqrt{m}\log n)$. As demais operações podem ser realizadas dentro deste limite de tempo (como veremos na próxima seção) e, portanto, o tempo gasto no total pelo algoritmo é $O(n\sqrt{m}\log n)$.

6.3 Implementação em CWEB

Bicategorias

141 ⟨ Header files of bicat.c 142 ⟩
⟨ Data structures of bicat.c 143 ⟩
⟨ Global variables of bicat.c 155 ⟩
⟨ Internal functions of bicat.c 146 ⟩
⟨ External functions of bicat.c 150 ⟩

142 ⟨ Header files of bicat.c 142 ⟩ ≡

```
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "gb_graph.h"
#include "item.h"
#include "fibheap.h"
#include "uf.h"
#include "bicat.h"
```

Este código é usado no bloco 141.

143 ⟨ Data structures of bicat.c 143 ⟩ ≡

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

Veja também blocos 144, 145 e 149.

Este código é usado no bloco 141.

144 ⟨ Data structures of bicat.c 143 ⟩ +≡

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

145 #**define** ARC_FIELDS(*A*) ((**ArcFields** ∗)(*A*)→*b*.*S*)

⟨ Data structures of bicat.c 143 ⟩ +≡

```
typedef struct arc_fields {
  double Delta;
  FHnode node;
  Arc *prev;
  Arc *next;
} ArcFields;
```

146 ⟨ Internal functions of bicat.c 146 ⟩ ≡

```
double arc_key(void *p)
{
  Arc *a = p;
  return ARC_FIELDS(a)→Delta;
}
```

Veja também blocos 147, 148 e 177.

Este código é usado no bloco 141.

147 ⟨ Internal functions of bicat.c 146 ⟩ +≡

```
double heap_key(void *p)
{
  LbHeap h = p;
  if (h→heap) {
    if (¬FHEAPempty(h→heap)) {
      Arc *min = getValue(FHEAPfindMin(h→heap));
      return h→Delta + ARC_FIELDS(min)→Delta;
    }
    else return DBL_MAX;
  }
  return −DBL_MAX;
}
```

148 ⟨ Internal functions of bicat.c 146 ⟩ +≡

```
static void LISTinsert(Arc *a, Arc *b)
{
  Arc *c = ARC_FIELDS(a)→next;
```

```
    ARC_FIELDS(b)→next = c;
    ARC_FIELDS(c)→prev = b;
    ARC_FIELDS(a)→next = b;
    ARC_FIELDS(b)→prev = a;
  }
  static void LISTdelete(Arc *b)
  {
    Arc *a = ARC_FIELDS(b)→prev;
    Arc *c = ARC_FIELDS(b)→next;
    ARC_FIELDS(a)→next = c;
    ARC_FIELDS(c)→prev = a;
  }
  static void LISTconcat(Arc *h1, Arc *h2)
  {
    if (ARC_FIELDS(h2)→next ≠ h2) {
      Arc *last1 = ARC_FIELDS(h1)→prev;
      Arc *first2 = ARC_FIELDS(h2)→next;
      Arc *last2 = ARC_FIELDS(h2)→prev;
      ARC_FIELDS(last1)→next = first2;
      ARC_FIELDS(first2)→prev = last1;
      ARC_FIELDS(last2)→next = h1;
      ARC_FIELDS(h1)→prev = last2;
    }
  }
```

149 `#define VERTEX_FIELDS(V) ((VertexFields *)(V)→w.S)`

⟨ Data structures of bicat.c 143 ⟩ +≡

```
  typedef struct vertex_fields {
    int id;
    int out_degree;
    Arc *extra_in;
    Arc *extra_out;
    int weight;
  } VertexFields;
```

150 `#define from a.V`

⟨ External functions of bicat.c 150 ⟩ ≡

```
Vertex *From(Arc *a)
{
  int i = VERTEX_FIELDS(a→from)→id;
  int j = UFfind(i);
  return a→from + j − i;
}
```

Veja também blocos 151, 152, 153, 154, 161, 162, 163, 168 e 181.

Este código é usado no bloco 141.

151 ⟨ External functions of bicat.c 150 ⟩ +≡

```
Vertex *Tip(Arc *a)
{
  int i = VERTEX_FIELDS(a→tip)→id;
  int j = UFfind(i);
  return a→tip + j − i;
}
```

152 ⟨ External functions of bicat.c 150 ⟩ +≡

```
int weight(Vertex *v)
{
  return VERTEX_FIELDS(v)→weight;
}
```

153 #define *category* *z.I*

⟨ External functions of bicat.c 150 ⟩ +≡

```
double cost(Arc *a)
{
  Vertex *u, *v;
  Bicategory *B;
  double Delta_H, Delta_h;
  u = From(a);
  v = Tip(a);
  B = b0 + u→category * C + v→category;
  Delta_H = B→H→Delta;
  if (VERTEX_FIELDS(u)→out_degree ≥ sqrt_m) {
```

```
    Item item_h = FHEAPdelete(B→H→heap, B→out[VERTEX_FIELDS(u)→id]);
    LbHeap h = getValue(item_h);
    Delta_h = h→Delta;
    B→out[VERTEX_FIELDS(u)→id] = FHEAPinsert(B→H→heap, item_h);
  }
  else {
    Item item_h = FHEAPdelete(B→H→heap, B→in[VERTEX_FIELDS(v)→id]);
    LbHeap h = getValue(item_h);
    Delta_h = h→Delta;
    B→in[VERTEX_FIELDS(v)→id] = FHEAPinsert(B→H→heap, item_h);
  }
  return Delta_H + Delta_h + ARC_FIELDS(a)→Delta;
}
```

154 ⟨ External functions of bicat.c 150 ⟩ +≡

```
void initBicategories(Graph *G, int c)
{
  int n = G→n;
  ⟨ Allocate space for c² bicategories 156 ⟩
  ⟨ Assign a direction to each edge of G and initialize the arcs an vertices fields 157 ⟩
  ⟨ Assign each arc to one of its two endpoints 159 ⟩
}
```

155 ⟨ Global variables of bicat.c 155 ⟩ ≡

```
Bicategory *b0;
int C;
```

Veja também blocos 158 e 160.

Este código é usado no bloco 141.

156 ⟨ Allocate space for c^2 bicategories 156 ⟩ ≡

```
{
  int i;
  LbHeap lh;
  FHnode *nodes;
```

```
  C = c;
  b0 = malloc(C * C * sizeof(Bicategory));
  lh = malloc(C * C * sizeof(struct labeled_heap));
  nodes = malloc(2 * n * C * C * sizeof(FHnode));
  FHEAPalloc(2 * n * C * C + C * C, 2 * n * C * C + G→m/2);
  for (i = 0; i < C * C; i++) {
    Bicategory *B = b0 + i;
    B→H = lh + i;
    B→H→heap = FHEAPinit( );
    B→H→Delta = 0;
    B→in = nodes + 2 * i * n;
    B→out = B→in + n;
  }
}
```

Este código é usado no bloco 154.

157 ⟨ Assign a direction to each edge of G and initialize the arcs an vertices fields 157 ⟩ ≡

```
{
  int i, j;
  Arc *head;
  ArcFields *af;
  VertexFields *vf;
  vf = malloc(n * sizeof(VertexFields));
  af = malloc((G→m/2 + 2 * n) * sizeof(ArcFields));
  head = malloc(2 * n * sizeof(Arc));
  min_cost_arc = malloc(n * sizeof(Arc *));
  for (i = 0, j = 0; i < n; i++) {
    Arc *a;
    Vertex *v;
    v = G→vertices + i;
    v→w.S = (char *)(vf + i);
    VERTEX_FIELDS(v)→id = i;
    VERTEX_FIELDS(v)→out_degree = 0;
    for (a = v→arcs; a; a = a→next) {
      Vertex *w = a→tip;
      if (v < w) {
        a→b.S = (char *)(af + j);
```

```
            ARC_FIELDS(a)→Delta = a→len;
            j++;
            VERTEX_FIELDS(v)→out_degree++;
          }
        }
        a = head + i;
        a→b.S = (char *)(af + j++);
        ARC_FIELDS(a)→prev = ARC_FIELDS(a)→next = a;
        VERTEX_FIELDS(v)→extra_in = a;
        a = head + n + i;
        a→b.S = (char *)(af + j++);
        ARC_FIELDS(a)→prev = ARC_FIELDS(a)→next = a;
        VERTEX_FIELDS(v)→extra_out = a;
        VERTEX_FIELDS(v)→weight = 1;
        min_cost_arc[i] = Λ;
      }
      UFinit(n);
    }
```

Este código é usado no bloco 154.

158 ⟨ Global variables of bicat.c 155 ⟩ +≡

```
Arc **min_cost_arc;
```

159 ⟨ Assign each arc to one of its two endpoints 159 ⟩ ≡

```
{
  Vertex *v;
  LbHeap h_in, h_out;
  sqrt_m = sqrt(G→m/2);
  LH = malloc(2 * C * C * n * sizeof(struct labeled_heap));
  min_heap = malloc(sizeof(struct labeled_heap));
  min_heap→heap = Λ;
  allocItens(2 * n * C * C + G→m/2);
  for (v = G→vertices + n − 1; v ≥ G→vertices; v−−) {
    Arc *a;
    Bicategory *B;
    int i;
```

```
h_in = LH + (G→vertices + n − 1 − v) * 2 * C * C;
h_out = h_in + C * C;
for (i = 0; i < C * C; i++) {
  B = b0 + i;
  h_in[i].heap = FHEAPinit( );
  h_in[i].Delta = 0;
  B→in[VERTEX_FIELDS(v)→id] = FHEAPinsert(B→H→heap, newItem(h_in + i,
      heap_key, min_heap));
  h_out[i].heap = FHEAPinit( );
  h_out[i].Delta = 0;
  B→out[VERTEX_FIELDS(v)→id] = FHEAPinsert(B→H→heap, newItem(h_out + i,
      heap_key, min_heap));
}
min_arc = malloc(sizeof(Arc));
min_arc→b.S = (char *) malloc(sizeof(ArcFields));
ARC_FIELDS(min_arc)→Delta = −DBL_MAX;
for (a = v→arcs; a; a = a→next) {
  int bicat_id;
  Item item;
  FHnode node;
  Vertex *w = a→tip;
  if (v > w) continue;
  bicat_id = v→category * C + w→category;
  B = b0 + bicat_id;
  item = newItem(a, arc_key, min_arc);
  if (VERTEX_FIELDS(v)→out_degree ≥ sqrt_m) {
    ARC_FIELDS(a)→node = FHEAPinsert(h_out[bicat_id].heap, item);
    node = B→out[VERTEX_FIELDS(v)→id];
    FHEAPdecreaseKey(B→H→heap, node, Λ);
    LISTinsert(VERTEX_FIELDS(w)→extra_in, a);
  }
  else {
    LbHeap h;
    Item item_h;
    node = B→in[VERTEX_FIELDS(w)→id];
    item_h = FHEAPdelete(B→H→heap, node);
    h = getValue(item_h);
    ARC_FIELDS(a)→node = FHEAPinsert(h→heap, item);
```

```
        B→in[VERTEX_FIELDS(w)→id] = FHEAPinsert(B→H→heap, item_h);
        LISTinsert(VERTEX_FIELDS(v)→extra_out, a);
      }
    }
  }
}
```

Este código é usado no bloco 154.

160 ⟨ Global variables of bicat.c 155 ⟩ +≡

```
double sqrt_m;
Arc *min_arc;
LbHeap LH, min_heap;
```

161 ⟨ External functions of bicat.c 150 ⟩ +≡

```
void decreaseCost(int cat1, int cat2, double delta)
{
  Bicategory *B = b0 + cat1 * C + cat2;
  B→H→Delta -= delta;
}
```

162 ⟨ External functions of bicat.c 150 ⟩ +≡

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

163 ⟨ External functions of bicat.c 150 ⟩ +≡

```
void changeCategory(Vertex *v, int new_cat)
{
  int old_cat = v→category;
  Arc *a, *head;
```

```
    v→category = new_cat;
    ⟨ Change the bicategory of the arcs assigned to v 164 ⟩
    ⟨ Discard parallel arcs in the extra_in list of v 165 ⟩
    ⟨ Change the bicategory of each arc in the extra_in list of v 166 ⟩
    ⟨ Change the bicategory of each arc in the extra_out list of v 167 ⟩
  }
```

164 ⟨ Change the bicategory of the arcs assigned to v 164 ⟩ ≡

```
  {
    int id_v = VERTEX_FIELDS(v)→id, c;
    for (c = 0; c < C; c++) {
      int old_id, new_id;
      Bicategory *old, *new;
      Item item_h1, item_h2;
      LbHeap h1, h2;
      old_id = c * C + old_cat;
      new_id = c * C + new_cat;
      old = b0 + old_id;
      new = b0 + new_id;
      item_h1 = FHEAPdelete(old→H→heap, old→in[id_v]);
      item_h2 = FHEAPdelete(new→H→heap, new→in[id_v]);      /* este heap aqui é vazio */
      h1 = getValue(item_h1);
      h2 = getValue(item_h2);
      h1→Delta += old→H→Delta − new→H→Delta;
      new→in[id_v] = FHEAPinsert(new→H→heap, item_h1);
      h2→Delta = 0;
      old→in[id_v] = FHEAPinsert(old→H→heap, item_h2);
      old_id = old_cat * C + c;
      new_id = new_cat * C + c;
      old = b0 + old_id;
      new = b0 + new_id;
      item_h1 = FHEAPdelete(old→H→heap, old→out[id_v]);
      item_h2 = FHEAPdelete(new→H→heap, new→out[id_v]);
        /* este heap aqui é vazio */
      h1 = getValue(item_h1);
      h2 = getValue(item_h2);
      h1→Delta += old→H→Delta − new→H→Delta;
      new→out[id_v] = FHEAPinsert(new→H→heap, item_h1);
```

```
    h2→Delta = 0;
    old→out[id_v] = FHEAPinsert(old→H→heap, item_h2);
  }
}
```

Este código é usado no bloco 163.

165 ⟨ Discard parallel arcs in the *extra_in* list of *v* 165 ⟩ ≡

```
{
  head = VERTEX_FIELDS(v)→extra_in;
  a = ARC_FIELDS(head)→next;
  while (a ≠ head) {
    Arc *next = ARC_FIELDS(a)→next;
    Vertex *u = From(a);
    int id_u = VERTEX_FIELDS(u)→id;
    if (¬min_cost_arc[id_u]) min_cost_arc[id_u] = a;
    else {
      Bicategory *B = b0 + (u→category * C + old_cat);
      Item out_u = FHEAPdelete(B→H→heap, B→out[id_u]);
      LbHeap h_out_u = getValue(out_u);
      Arc *b = min_cost_arc[id_u];
      if (ARC_FIELDS(a)→Delta < ARC_FIELDS(b)→Delta) {
        min_cost_arc[id_u] = a;
        LISTdelete(b);      /* remove b da lista extra_in de v */
        FHEAPdelete(h_out_u→heap, ARC_FIELDS(b)→node);
            /* liberar a memória do item correspondente ao arco removido */
      }
      else {
        LISTdelete(a);      /* remove a da lista extra_in de v */
        FHEAPdelete(h_out_u→heap, ARC_FIELDS(a)→node);
            /* liberar a memória do item correspondente ao arco removido */
      }
      B→out[id_u] = FHEAPinsert(B→H→heap, out_u);
    }
    a = next;
  }
}
```

Este código é usado no bloco 163.

166 ⟨ Change the bicategory of each arc in the *extra_ in* list of v 166 ⟩ ≡

```
{
  for (a = ARC_FIELDS(head)→next; a ≠ head; a = ARC_FIELDS(a)→next) {
    Vertex *u;
    int id_u;
    Bicategory *old, *new;
    Item item_h, item_a;
    LbHeap h_old, h_new;
    u = From(a);
    id_u = VERTEX_FIELDS(u)→id;
    min_cost_arc[id_u] = Λ;
    old = b0 + u→category * C + old_cat;
    item_h = FHEAPdelete(old→H→heap, old→out[id_u]);
    h_old = getValue(item_h);
    item_a = FHEAPdelete(h_old→heap, ARC_FIELDS(a)→node);
    old→out[id_u] = FHEAPinsert(old→H→heap, item_h);
    new = b0 + u→category * C + new_cat;
    item_h = FHEAPdelete(new→H→heap, new→out[id_u]);
    h_new = getValue(item_h);
    ARC_FIELDS(a)→Delta += h_old→Delta + old→H→Delta − h_new→Delta − new→H→Delta;
    ARC_FIELDS(a)→node = FHEAPinsert(h_new→heap, item_a);
    new→out[id_u] = FHEAPinsert(new→H→heap, item_h);
  }
}
```

Este código é usado no bloco 163.

167 ⟨ Change the bicategory of each arc in the *extra_ out* list of v 167 ⟩ ≡

```
{
  head = VERTEX_FIELDS(v)→extra_out;
  for (a = ARC_FIELDS(head)→next; a ≠ head; a = ARC_FIELDS(a)→next) {
    Vertex *w;
    int id_w;
    Bicategory *old, *new;
    Item item_h, item_a;
    LbHeap h_old, h_new;
    w = Tip(a);
    id_w = VERTEX_FIELDS(w)→id;
```

```
    old = b0 + old_cat * C + w->category;
    item_h = FHEAPdelete(old->H->heap, old->in[id_w]);
    h_old = getValue(item_h);
    item_a = FHEAPdelete(h_old->heap, ARC_FIELDS(a)->node);
    old->in[id_w] = FHEAPinsert(old->H->heap, item_h);
    new = b0 + new_cat * C + w->category;
    item_h = FHEAPdelete(new->H->heap, new->in[id_w]);
    h_new = getValue(item_h);
    ARC_FIELDS(a)->Delta += h_old->Delta + old->H->Delta - h_new->Delta - new->H->Delta;
    ARC_FIELDS(a)->node = FHEAPinsert(h_new->heap, item_a);
    new->in[id_w] = FHEAPinsert(new->H->heap, item_h);
  }
}
```

Este código é usado no bloco 163.

168 ⟨ External functions of bicat.c 150 ⟩ +≡

```
void contractEdge(Arc *a, int cat)
{
  ⟨ Local variables of contractEdge 169 ⟩
  u = From(a);
  v = Tip(a);
  for (x = u, i = 0; i ≤ 1; i++, x = v) {
    if (x->category ≠ cat) changeCategory(x, cat);
    else ⟨ Discard parallel arcs in the extra_in list of x 170 ⟩
  }
  ⟨ Discard all arcs with both endpoints in u and v 172 ⟩
  if (VERTEX_FIELDS(u)->weight < VERTEX_FIELDS(v)->weight) {
    Vertex *tmp = u;
    u = v;
    v = tmp;
  }
  ⟨ For each bicategory b, join the heaps of u and v in b 176 ⟩
  ⟨ Change the assignment of the arcs incident to u or v if necessary 178 ⟩
  LISTconcat(VERTEX_FIELDS(u)->extra_in, VERTEX_FIELDS(v)->extra_in);
  LISTconcat(VERTEX_FIELDS(u)->extra_out, VERTEX_FIELDS(v)->extra_out);
  ⟨ Set u to represent the vertex resulting from the contraction of a 180 ⟩
}
```

169 ⟨ Local variables of *contractEdge* 169 ⟩ ≡

```
int i;
Vertex *u, *v, *x;
```

Veja também blocos 171, 174 e 179.

Este código é usado no bloco 168.

170 ⟨ Discard parallel arcs in the *extra_in* list of *x* 170 ⟩ ≡

```
{
  Arc *head = VERTEX_FIELDS(x)→extra_in;
  Arc *b = ARC_FIELDS(head)→next;
  while (b ≠ head) {
    Arc *next = ARC_FIELDS(b)→next;
    Vertex *w = From(b);
    int id_w = VERTEX_FIELDS(w)→id;
    if (¬min_cost_arc[id_w]) min_cost_arc[id_w] = b;
    else {
      Bicategory *B = b0 + (w→category * C + x→category);
      Item out_w = FHEAPdelete(B→H→heap, B→out[id_w]);
      LbHeap h_out_w = getValue(out_w);
      Arc *b_ = min_cost_arc[id_w];
      if (ARC_FIELDS(b)→Delta < ARC_FIELDS(b_)→Delta) {
        min_cost_arc[id_w] = b;
        LISTdelete(b_);      /* remove b da lista extra_in de v */
        FHEAPdelete(h_out_w→heap, ARC_FIELDS(b_)→node);
            /* liberar a memória do item correspondente ao arco removido */
      }
      else {
        LISTdelete(b);       /* remove a da lista extra_in de v */
        FHEAPdelete(h_out_w→heap, ARC_FIELDS(b)→node);
            /* liberar a memória do item correspondente ao arco removido */
      }
      B→out[id_w] = FHEAPinsert(B→H→heap, out_w);
    }
    b = next;
  }
  if (x ≡ u) vu = min_cost_arc[VERTEX_FIELDS(v)→id];
  if (x ≡ v) uv = min_cost_arc[VERTEX_FIELDS(u)→id];
```

```
    for (b = ARC_FIELDS(head)→next; b ≠ head; b = ARC_FIELDS(b)→next) {
      Vertex *w = From(b);
      min_cost_arc[VERTEX_FIELDS(w)→id] = Λ;
    }
  }
```

Este código é usado no bloco 168.

171 ⟨ Local variables of *contractEdge* 169 ⟩ +≡

```
  Arc *uv = Λ, *vu = Λ;
```

172 ⟨ Discard all arcs with both endpoints in u and v 172 ⟩ ≡

```
  {
    ⟨ Discard all arcs uv from the graph 173 ⟩
    ⟨ Discard all arcs vu from the graph 175 ⟩
  }
```

Este código é usado no bloco 168.

173 ⟨ Discard all arcs uv from the graph 173 ⟩ ≡

```
  {
    Bicategory *B = b0 + (u→category * C + v→category);
    id_u = VERTEX_FIELDS(u)→id;
    id_v = VERTEX_FIELDS(v)→id;
    if (VERTEX_FIELDS(u)→out_degree < sqrt_m) {
      Item in_v = FHEAPdelete(B→H→heap, B→in[id_v]);
      LbHeap h_in_v = getValue(in_v);
      Arc *head = VERTEX_FIELDS(u)→extra_out;
      Arc *b = ARC_FIELDS(head)→next;
      while (b ≠ head) {
        Arc *next = ARC_FIELDS(b)→next;
        if (Tip(b) ≡ v) {
          LISTdelete(b);
          FHEAPdelete(h_in_v→heap, ARC_FIELDS(b)→node);
        }
        b = next;
      }
      B→in[id_v] = FHEAPinsert(B→H→heap, in_v);
```

```
    }
    else {
      Item out_u = FHEAPdelete(B→H→heap, B→out[id_u]);
      LbHeap h_out_u = getValue(out_u);
      if (¬uv) {
        Arc *head = VERTEX_FIELDS(v)→extra_in, *b;
        for (b = ARC_FIELDS(head)→next; From(b) ≠ u; b = ARC_FIELDS(b)→next) ;
        LISTdelete(b);
        FHEAPdelete(h_out_u→heap, ARC_FIELDS(b)→node);
      }
      else {
        LISTdelete(uv);
        FHEAPdelete(h_out_u→heap, ARC_FIELDS(uv)→node);
      }
      B→out[id_u] = FHEAPinsert(B→H→heap, out_u);
    }
  }
```

Este código é usado no bloco 172.

174 ⟨ Local variables of *contractEdge* 169 ⟩ +≡

```
  int id_u, id_v;
```

175 ⟨ Discard all arcs *vu* from the graph 175 ⟩ ≡

```
  {
    Bicategory *B = b0 + (v→category * C + u→category);
    if (VERTEX_FIELDS(v)→out_degree < sqrt_m) {
      Item in_u = FHEAPdelete(B→H→heap, B→in[id_u]);
      LbHeap h_in_u = getValue(in_u);
      Arc *head = VERTEX_FIELDS(v)→extra_out;
      Arc *b = ARC_FIELDS(head)→next;
      while (b ≠ head) {
        Arc *next = ARC_FIELDS(b)→next;
        if (Tip(b) ≡ u) {
          LISTdelete(b);
          FHEAPdelete(h_in_u→heap, ARC_FIELDS(b)→node);
        }
```

```
        b = next;
      }
      B→in[id_u] = FHEAPinsert(B→H→heap, in_u);
    }
    else {
      Item out_v = FHEAPdelete(B→H→heap, B→out[id_v]);
      LbHeap h_out_v = getValue(out_v);
      if (¬vu) {
        Arc *head = VERTEX_FIELDS(u)→extra_in, *b;
        for (b = ARC_FIELDS(head)→next; b ≠ head; b = ARC_FIELDS(b)→next)
          if (From(b) ≡ v) {
            LISTdelete(b);
            FHEAPdelete(h_out_v→heap, ARC_FIELDS(b)→node);
            break;
          }
      }
      else {
        LISTdelete(vu);
        FHEAPdelete(h_out_v→heap, ARC_FIELDS(vu)→node);
      }
      B→out[id_v] = FHEAPinsert(B→H→heap, out_v);
    }
  }
```

Este código é usado no bloco 172.

176 ⟨ For each bicategory b, join the heaps of u and v in b 176 ⟩ ≡

```
  {
    Bicategory *B;
    id_u = VERTEX_FIELDS(u)→id;
    id_v = VERTEX_FIELDS(v)→id;
    for (B = b0; B < b0 + C * C; B++) {
      double inc;
      void *args[1];
      Item in_u, in_v, out_u, out_v;
      LbHeap h_in_u, h_in_v, h_out_u, h_out_v;
      in_u = FHEAPdelete(B→H→heap, B→in[id_u]);
      h_in_u = getValue(in_u);
```

```
    in_v = FHEAPdelete(B→H→heap, B→in[id_v]);
    h_in_v = getValue(in_v);
    inc = h_in_v→Delta − h_in_u→Delta;
    args[0] = &inc;
    FHEAPtraverse(h_in_v→heap, inc_field_delta, args);
    h_in_u→heap = FHEAPjoin(h_in_u→heap, h_in_v→heap);
    B→in[id_u] = FHEAPinsert(B→H→heap, in_u);
    out_u = FHEAPdelete(B→H→heap, B→out[id_u]);
    h_out_u = getValue(out_u);
    out_v = FHEAPdelete(B→H→heap, B→out[id_v]);
    h_out_v = getValue(out_v);
    inc = h_out_v→Delta − h_out_u→Delta;
    FHEAPtraverse(h_out_v→heap, inc_field_delta, args);
    h_out_u→heap = FHEAPjoin(h_out_u→heap, h_out_v→heap);
    B→out[id_u] = FHEAPinsert(B→H→heap, out_u);
  }
}
```

Este código é usado no bloco 168.

177 ⟨ Internal functions of bicat.c 146 ⟩ +≡

```
void inc_field_delta(Item item, void *args[])
{
  Arc *a = getValue(item);
  double inc = *((double *) args[0]);
  ARC_FIELDS(a)→Delta += inc;
}
```

178 ⟨ Change the assignment of the arcs incident to u or v if necessary 178 ⟩ ≡

```
{
  Bicategory *B;
  out_degree_u = VERTEX_FIELDS(u)→out_degree;
  out_degree_v = VERTEX_FIELDS(v)→out_degree;
  if (out_degree_u + out_degree_v ≥ sqrt_m) {
    int i;
    Vertex *x;
```

```
    if (out_degree_u < sqrt_m ∨ out_degree_v < sqrt_m) {
      h_u = malloc(C * C * sizeof(LbHeap));
      item_u = malloc(C * C * sizeof(Item));
      for (i = 0; i < C * C; i++) {
        B = b0 + i;
        item_u[i] = FHEAPdelete(B→H→heap, B→out[id_u]);
        h_u[i] = getValue(item_u[i]);
      }
    }
    for (i = 1, x = u; i ≤ 2; i++, x = v) {
      if (VERTEX_FIELDS(x)→out_degree < sqrt_m) {
        Arc *head = VERTEX_FIELDS(x)→extra_out;
        Arc *a = ARC_FIELDS(head)→next;
        Item item_a;

        while (a ≠ head) {
          Arc *next = ARC_FIELDS(a)→next;
          Vertex *y;
          int id_y, bicat_id;
          Item item_y;
          LbHeap h_y;

          y = Tip(a);
          id_y = VERTEX_FIELDS(y)→id;
          bicat_id = x→category * C + y→category;
          B = b0 + bicat_id;
          item_y = FHEAPdelete(B→H→heap, B→in[id_y]);
          h_y = getValue(item_y);
          item_a = FHEAPdelete(h_y→heap, ARC_FIELDS(a)→node);
          B→in[id_y] = FHEAPinsert(B→H→heap, item_y);
          ARC_FIELDS(a)→Delta += h_y→Delta − h_u[bicat_id]→Delta;
          ARC_FIELDS(a)→node = FHEAPinsert(h_u[bicat_id]→heap, item_a);
          LISTdelete(a);      /* remove a da lista extra_out de x */
          LISTinsert(VERTEX_FIELDS(y)→extra_in, a);
          a = next;
        }
      }
    }
  }
}
```

Este código é usado no bloco 168.

179 ⟨ Local variables of *contractEdge* 169 ⟩ +≡

```
int out_degree_u, out_degree_v;
LbHeap *h_u;
Item *item_u;
```

180 ⟨ Set u to represent the vertex resulting from the contraction of a 180 ⟩ ≡

```
{
  VERTEX_FIELDS(u)→out_degree += out_degree_v;
  VERTEX_FIELDS(u)→weight += VERTEX_FIELDS(v)→weight;
  UFunion(id_u, id_v);
  if (out_degree_u + out_degree_v ≥ sqrt_m)
    if (out_degree_u < sqrt_m ∨ out_degree_v < sqrt_m) {
      int i;
      for (i = 0; i < C * C; i++) {
        Bicategory *B = b0 + i;
        B→out[id_u] = FHEAPinsert(B→H→heap, item_u[i]);
      }
      free(h_u);
      free(item_u);
    }
}
```

Este código é usado no bloco 168.

181 ⟨ External functions of bicat.c 150 ⟩ +≡

```
void freeBicatMem(Graph *g)
{
  Arc *a;
  Vertex *v;
  free(b0→H);
  free(b0→in);
  free(b0);
  free(LH);
  v = g→vertices;
  free(VERTEX_FIELDS(v)→extra_in);
```

```
    free(v→w.S);
    do {
      do {
        a = v→arcs;
        v++;
      } while (¬a);
      while (a ∧ a→tip < a→from) a = a→next;
    } while (¬a);
    free(a→b.S);
    free(min_cost_arc);
    free(min_heap);
    free(min_arc→b.S);
    free(min_arc);
    UFdestroy( );
    FHEAPfree( );
    freeItens( );
  }
```

182 ⟨ bicat.h 182 ⟩ ≡

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

Algoritmo

183 ⟨ Header files of sf.c 71 ⟩ +≡

```
#include "bicat.h"
#include "float.h"
```

184 #**define** INACTIVE 0
#**define** ACTIVE 1

⟨ Steiner forest construction functions 77 ⟩ +≡
SteinerForest ∗*sf_klein*(**Graph** ∗*g*, **TermSet** ∗*term_sets*)
{
Graph ∗*sf*;
SteinerForest ∗*min_sf*;
Intersection *intersect*;
int *num_actives*;
double *dual_cost* = 0;
⟨ Assign one category to each vertex of *g* 185 ⟩
initBicategories(*g*, 2);
⟨ Set *sf* to the initial spanning forest 186 ⟩
while (*num_actives* > 0) {
Arc ∗*uv*, ∗*a1*, ∗*a2*, ∗*a3*;
double *delta*, *d1*, *d2*, *d3*;
int *cat*;
uv = Λ;
a1 = *findMin*(INACTIVE, ACTIVE);
a2 = *findMin*(ACTIVE, INACTIVE);
a3 = *findMin*(ACTIVE, ACTIVE);
d1 = (¬*a1*) ? DBL_MAX : *cost*(*a1*);
d2 = (¬*a2*) ? DBL_MAX : *cost*(*a2*);
d3 = (¬*a3*) ? DBL_MAX : *cost*(*a3*)/2;
if (*d1* ≤ *d2*) {
delta = *d1*;
uv = *a1*;
}
else {
delta = *d2*;
uv = *a2*;
}
if (*d3* ≤ *delta*) {
delta = *d3*;
uv = *a3*;
}
if (¬*uv*) **return** Λ; /∗ the problem is unfeasible ∗/
dual_cost += *delta* ∗ *num_actives*;

```
    decreaseCost(INACTIVE, ACTIVE, delta);
    decreaseCost(ACTIVE, INACTIVE, delta);
    decreaseCost(ACTIVE, ACTIVE, 2 * delta);
    ⟨ Include uv in sf 187 ⟩
    if (delta ≡ d3) num_actives--;
    ⟨ Set cat to the state of the component in sf that contains uv 188 ⟩
    contractEdge(uv, cat);
  }
  freeBicatMem(g);
  min_sf = edgePruning(sf, g, term_sets);
  min_sf→dual_cost = dual_cost;
  ⟨ Free the extra memory allocated 190 ⟩
  return min_sf;
}
```

185 `#define ts_intersect x.S`
`#define ts_disconnect y.I`
`#define category z.I`

⟨ Assign one category to each vertex of g 185 ⟩ ≡

```
{
  int i;
  Vertex *v;
  num_actives = 0;
  for (i = 0; i < g→num_ts; i++) num_actives += term_sets[i].num_vertices;
  intersect = malloc(num_actives * sizeof(struct i_struct));
  BBSTalloc(num_actives + g→num_ts);
  allocItens(num_actives);
  for (v = g→vertices, i = 0; v < g→vertices + g→n; v++) {
    v→ts_intersect = (char *) BBSTinit( );
    if (v→termset_id < 0) {
      v→category = INACTIVE;
      v→ts_disconnect = 0;
    }
    else {
      Item y;
      Intersection I = intersect + i++;
      I→ts_id = v→termset_id, I→n = 1;
```

```
        v→ts_intersect = (char *) BBSTinsert((BalBST) v→ts_intersect, newItem(I, I_key,
            Λ), &y);
        v→ts_disconnect = 1;
        v→category = ACTIVE;
      }
    }
  }
```

Este código é usado no bloco 184.

186 ⟨ Set *sf* to the initial spanning forest 186 ⟩ ≡

```
{
  Vertex *v;
  sf = gb_new_graph(g→n);
  for (v = sf→vertices; v < sf→vertices + sf→n; v++) {
    Vertex *w = g→vertices + (v − sf→vertices);
    v→name = gb_save_string(w→name);
  }
}
```

Este código é usado no bloco 184.

187 ⟨ Include *uv* in *sf* 187 ⟩ ≡

```
{
  Vertex *u = sf→vertices + (uv→from − g→vertices);
  Vertex *v = sf→vertices + (uv→tip − g→vertices);
  Arc *vu = (u < v) ? uv + 1 : uv − 1;
  gb_new_edge(u, v, uv→len);
  u→arcs→original_arc = uv;
  v→arcs→original_arc = vu;
}
```

Este código é usado no bloco 184.

188 ⟨ Set *cat* to the state of the component in *sf* that contains *uv* 188 ⟩ ≡

```
{
  void *args[3];
  Vertex *u = From(uv);
  Vertex *v = Tip(uv);
```

```
  cat = 1;
  if (weight(u) < weight(v)) {
    Vertex *tmp = u;
    u = v;
    v = tmp;
  }
  args[0] = term_sets;
  args[1] = u;
  args[2] = &cat;
  BBSTtraverse((BalBST)(v→ts_intersect), insertIfNotConnected, args);
  if (cat ≡ INACTIVE) num_actives−−;
  BBSTdestroy((BalBST) v→ts_intersect);
}
```

Este código é usado no bloco 184.

189 ⟨ Auxiliary functions 76 ⟩ +≡

```
void insertIfNotConnected(Item item, void *args[])
{
  Item item_;
  Intersection I = getValue(item);
  TermSet *termsets = args[0];
  Vertex *u = args[1];
  int *cat = args[2];
  TermSet *ts;
  ts = termsets + I→ts_id;
  if (ts→connected) return;
  u→ts_intersect = (char *) BBSTinsert((BalBST)(u→ts_intersect), item, &item_);
  if (item_ ≠ Λ) {
    Intersection J = getValue(item_);
    J→n += I→n;
    if (J→n ≡ ts→num_vertices) {
      ts→connected++;
      u→ts_disconnect−−;
      if (¬(u→ts_disconnect)) *cat = INACTIVE;
    }
  }
  else {
```

```
      u->ts_disconnect++;
      if (u->ts_disconnect == 1) *cat = ACTIVE;
    }
  }
```

190 ⟨ Free the extra memory allocated 190 ⟩ ≡

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

Este código é usado no bloco 184.

Apêndice A

Cálculo do ancestral comum mais próximo

Seja T uma árvore enraizada e seja x um vértice qualquer de T. Dizemos que todo vértice que ocorre no caminho que vai de x até a raiz de T é um *ancestral* de x em T. Agora, sejam u e v dois vértices de T e P_u e P_v os caminhos que vão, respectivamente, de u até a raiz e de v até a raiz. O *ancestral comum mais próximo* (em inglês, *nearest common ancestor* ou *least common ancestor*) de u e v em T é o primeiro vértice que ocorre tanto em P_u quanto em P_v.

Considere então o seguinte

> **Problema**: Dados uma árvore enraizada T e dois vértices u e v de T, encontrar o ancestral comum mais próximo de u e v em T.

Apresentaremos aqui uma implementação de um algoritmo que preprocessa T em tempo linear no seu número de vértices e, após isso, responde a consultas pelo ancestral comum mais próximo de quaisquer dois vértices de T em tempo constante. Este algoritmo, concebido por Schieber e Vishkin, foi originalmente apresentado em [17]. Uma descrição mais detalhada pode ser encontrada em [16].

A.1 Dois casos especiais: cadeias e árvores binárias completas

O algoritmo para o cálculo do ancestral comum mais próximo em árvores enraizadas arbitrárias se baseia em algoritmos para dois casos especiais: cadeias e árvores binárias completas.

Quando a árvore T do problema é uma cadeia (ou seja, cada vértice interno de T contém apenas um único filho) é facil determinar o ancestral comum mais próximo de u e v: basta

determinar qual dos dois vértices se encontra mais próximo da raiz e este será o vértice ancestral procurado. Em outras palavras, o ancestral comum mais próximo de u e v será aquele, dentre u e v, que se encontra no nível de menor valor (isto é, no nível mais alto) da árvore (lembrando que o nível de um vértice na árvore é dado por sua distância até a raiz).

Quando T é uma árvore binária completa (ou seja, uma árvore onde cada vértice interno possui exatamente dois filhos e todas as folhas se encontram em um mesmo nível) precisamos inicialmente calcular a ordem em que cada vértice de T é visitado em um percurso de T em inordem. Isto é necessário, pois a representação binária do número $in(v) \geq 1$ associado a ordem em que cada vértice v é visitado neste percurso apresenta propriedades bastante úteis. Assumindo que os bits da representação binária de $in(v)$ estão indexados de 0 (bit menos significativo) a l (bit mais significativo), onde l é o nível em que se encontram as folhas de T, e que i é o índice do bit 1 mais à direita (ou seja, menos significativo), essas propriedades são as seguintes:

- $l - i$ indica o nível em que o vértice v se encontra em T;
- se v_e é o filho esquerdo do vértice v, então a representação binária de $in(v_e)$ consiste dos $l - i$ primeiros bits de $in(v)$ (ou seja, os $l - i$ bits mais significativos), seguidos de um bit 0, um bit 1 e $i - 1$ bits 0;
- se v_d é o filho direito do vértice v, então a representação binária de $in(v_d)$ consiste dos $l - i$ primeiros bits de $in(v)$, seguidos de dois bits 1 e $i - 1$ bits 0.

A partir dessas propriedades, pode-se deduzir facilmente que um vértice v é ancestral de um vértice w se, e somente se, os números $in(v)$ e $in(w)$ satisfazem as seguintes condições, onde i é o índice do bit 1 mais à direita na representação binária de $in(v)$:

1. os $l - i$ primeiros bits de $in(w)$ correspondem exatamente aos $l - i$ primeiros bits de $in(v)$;
2. o índice do bit 1 mais à direita de $in(w)$ é menor ou igual a i.

Desta forma, dados dois vértices x e y, um vértice z é ancestral comum de x e y se os $l - i$ primeiros bits de $in(z)$ (onde i é, como antes, o índice do bit 1 mais à direita na representação binária de $in(z)$) são iguais aos $l - i$ primeiros bits de $in(x)$ e de $in(y)$ e, além disso, o índice do bit 1 mais à direita em ambos, $in(x)$ e $in(y)$, é menor ou igual a i. Note que, assim, o ancestral comum mais próximo de x e y é aquele para o qual i é mínimo (pois este é o ancestral que se encontra num nível mais baixo — ou seja, de maior valor — que os demais).

O algoritmo abaixo se baseia nestas idéias para encontrar o ancestral comum mais próximo de dois vértices x e y em uma árvore binária completa T.

Algoritmo *ACMP-BC(T, x, y)*

```
1  i_x ← índice do bit 1 mais à direita de in(x)
2  i_y ← índice do bit 1 mais à direita de in(y)
3  i_max ← max{i_x, i_y}
4  se os l − i_max primeiros bits de in(x) são iguais aos l − i_max primeiros bits de in(y)
5  então
6        se i_x = i_max
7        então devolva x
8        senão devolva y
9  senão
10       i ← índice do bit mais à esquerda em que in(x) e in(y) diferem
11       seja b o número formado pelos l − i primeiros bits de in(x) seguidos por um bit 1
          e i bits 0
12       seja z o vértice de T tal que in(z) = b
13       devolva z
```

A.2 O algoritmo

Como já foi mencionado, o algoritmo para árvores enraizadas arbitrárias se baseia nos algoritmos descritos no bloco anterior para dois casos especiais do problema. A idéia do algoritmo é particionar os vértices da árvore T do problema em cadeias disjuntas e mapear, através de uma função injetiva Ψ, este conjunto de cadeias no conjunto de vértices de uma árvore binária completa B. Tal mapeamento tem a seguinte propriedade (*preservação da ascendência*): se v é um ancestral de u em T então $\Psi(C_v)$ é um ancestral de $\Psi(C_u)$ em B, onde C_v e C_u são as cadeias em T às quais pertencem, respectivamente, v e u.

Assim, dados dois vértices x e y de T, se x e y pertencem a uma mesma cadeia, então é fácil determinar o ancestral comum mais próximo de x e y, como vimos no bloco anterior (caso especial em que a árvore do problema é uma cadeia). Caso contrário, sejam C_x e C_y as cadeias às quais x e y pertencem. Para descobrir o ancestral comum mais próximo de x e y em T, o algoritmo inicialmente encontra o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B (através do algoritmo *ACMP-BC* visto no bloco anterior). Então, usando algumas informações adicionais, como veremos adiante, o algoritmo identifica a cadeia em T que contém o ancestral comum mais próximo de x e y e, por fim, determina e devolve tal vértice.

Pré-processamento

O objetivo da fase de pré-processamento do algoritmo é fazer com que seja possível realizar em tempo constante consultas pelo ancestral comum mais próximo de quaisquer dois vértices da árvore T. Para isto, a árvore T é pré-processada de modo que, ao final, três rótulos estejam associados a cada vértice v de T: $level(v)$, $inlabel(v)$ e $ascendant(v)$. Além disso, é construída uma tabela auxiliar, a qual chamaremos de *head*.

Para cada v, o rótulo $level(v)$ indica o nível em que o vértice v se encontra em T. O nível de um vértice na árvore indica sua distância até a raiz e, desta forma, podemos defini-lo recursivamente da seguinte forma, onde $p(v)$ indica o pai do vértice v em T:

$$level(v) = \begin{cases} 0 & \text{se } v \text{ é a raiz de } T \\ level(p(v)) + 1 & \text{caso contrário.} \end{cases}$$

Note que, assim, $level(v)$ pode ser facilmente calculado para todo v, percorrendo-se a árvore T (uma única vez) em pré-ordem.

O rótulo *inlabel* é usado para particionar o conjunto de vértices de T em cadeias disjuntas. Assim, dois vértices u e v pertencem a uma mesma cadeia em T se e somente se $inlabel(u) = inlabel(v)$. Para calcular o rótulo *inlabel*, é necessário primeiramente calcular, para cada vértice v em T, os rótulos $pre(v)$ e $size(v)$ que indicam, respectivamente, a ordem em que v é visitado em um percurso em pré-ordem de T e o número de vértices na sub-árvore de T com raiz em v. O percurso de T em pré-ordem apresenta a seguinte propriedade: para cada vértice w na sub-árvore de T com raiz em v, $pre(v) \leq pre(w) \leq pre(v) + size(v) - 1$. Chamaremos de *intervalo de* v o intervalo fechado $[pre(v), pre(v) + size(v) - 1]$. Desta forma, vamos definir o rótulo $inlabel(v)$, para cada v, como sendo igual ao inteiro no intervalo de v cujo o índice do bit 1 mais à direita em sua representação binária é máximo.

Por que o rótulo *inlabel* definido desta maneira particiona os vértices de T em cadeias disjuntas? Primeiramente, note que, dados dois vértices u e v de T, se $inlabel(u) = inlabel(v) = p$, então o vértice w tal que $pre(w) = p$ se encontra tanto na sub-árvore de T com raiz em u quanto na sub-árvore de T com raiz em v. Desta forma, u e v são ancestrais de w e, portanto, u e v se encontram em uma mesma cadeia em T. Além disso, se dois vértices u e v se encontram em uma cadeia cujos extremos superior e inferior, s e t, respectivamente, são tais que $inlabel(s) = inlabel(t)$, então $inlabel(u) = inlabel(v)$. Vamos mostrar que isso é verdade. Sejam i_u, i_s e i_t os índices do bit 1 mais à direita em $inlabel(u)$, em $inlabel(s)$ e em $inlabel(t)$, respectivamente. Como t está na sub-árvore de T com raiz em u, temos que o intervalo de t está contido no intervalo de u e, desta forma, $i_u \geq i_t$. Seguindo o mesmo raciocínio, temos que $i_s \geq i_u$. Assim, se $inlabel(u) \neq inlabel(t)$ então $i_s \geq i_u > i_t$ e, portanto, $i_s > i_t$ o que é absurdo já que $inlabel(s) = inlabel(t)$. Logo, $inlabel(u) = inlabel(t)$. Do mesmo modo, concluímos que

$inlabel(v) = inlabel(t)$ e, portanto, $inlabel(u) = inlabel(v)$.

O rótulo *inlabel* também é usado para definir uma função injetiva Ψ que mapeia as cadeias em que T foi particionada nos vértices de uma árvore binária completa B com $2^{\lfloor \log |V_T| \rfloor + 1} - 1$ vértices (note que esta é a menor árvore que contém pelo menos $|V_T|$ vértices). Seja C_i a cadeia cujo valor do rótulo *inlabel* de cada um dos vértices é igual a i. Vamos definir $\Psi(C_i)$ da seguinte forma:

$$\Psi(C_i) = v \in V_B \text{ tal que } in(v) = i.$$

A função Ψ definida desta maneira satisfaz a *propriedade de preservação da ascendência*, ou seja, se u e v são dois vértices de T tais que v é ancestral de u, então $\Psi(C_v)$ é ancestral de $\Psi(C_u)$, onde C_v e C_u são as cadeias às quais pertencem v e u, respectivamente. Para enxergar isto, note primeiramente que, como v é ancestral de u, sabemos que u se encontra na sub-árvore de T com raiz em v e, deste modo, o intervalo de u está contido no intervalo de v. Assim, $inlabel(u)$ pertence ao intervalo de v e, portanto, $i_v \geq i_u$, onde i_u e i_v são os índices do bit 1 mais à direita em $inlabel(u)$ e em $inlabel(v)$, respectivamente. Além disso, pelo mesmo motivo, os $l - i_v$ bits mais à esquerda de $inlabel(u)$ são iguais aos $l - i_v$ bits mais à esquerda de $inlabel(v)$. Logo, $inlabel(v) = in(\Psi(C_v))$ e $inlabel(u) = in(\Psi(C_u))$ satisfazem as condições 1 e 2 descritas no bloco 2 e, portanto, $\Psi(C_v)$ é um ancestral de $\Psi(C_u)$ em B.

A recíproca da propriedade acima nem sempre é verdadeira, ou seja, é possível que, embora v não seja ancestral de u em T, $\Psi(C_v)$ seja um ancestral de $\Psi(C_u)$ em B. Esse fato motiva a utilização do rótulo *ascendant*. Para cada vértice u, $ascendant(u)$ guarda todos os ancestrais de $\Psi(C_u)$ em B que estão associados a ancestrais de u em T. Mais precisamente, $ascendant(u)$ guarda todos os números $inlabel(v)$ para os quais v é um ancestral de u em T (note que $inlabel(v) = in(\Psi(C_v))$ e que, pela propriedade da preservação da ascendência, $\Psi(C_v)$ é ancestral de $\Psi(C_u)$ em B). Podemos guardar todos estes números em apenas um único inteiro no intervalo $[0, 2^{\lfloor \log |V_T| \rfloor + 1} - 1]$. Isso segue do fato de que, para cada w ancestral de $\Psi(C_u)$ em B, podemos determinar $in(w)$ a partir do índice i do bit 1 mais à direita em sua representação binária, já que os $l - i$ primeiros bits de $in(w)$ são iguais aos $l - i$ primeiros bits de $in(\Psi(C_u)) = inlabel(u)$.

Desta forma, vamos definir $ascendant(u)$ como sendo o inteiro cuja representação binária é formada pelos bits A_j, $0 \leq j \leq \lfloor \log |V_T| \rfloor$, que satisfazem a seguinte propriedade:

$$A_j = 1 \Leftrightarrow \begin{array}{l} j \text{ é o índice do bit 1 mais à direita em } inlabel(v) \\ \text{para algum vértice } v \text{ ancestral de } u \text{ em } T. \end{array}$$

Note que o único ancestral da raiz de T é ela própria e que o valor do rótulo *inlabel* da raiz é 2^l. Além disso, se u é um vértice de T diferente da raiz, o único ancestral de u que não é ancestral

de $p(u)$ é o próprio u. Assim, podemos definir recursivamente o rótulo $ascendant(u)$, para cada vértice u em T, da seguinte forma, onde i é o índice do bit 1 mais à direita na representação binária de $inlabel(u)$:

$$ascendant(u) = \begin{cases} 2^l & \text{se } u \text{ é a raiz de } T \\ ascendant(p(u)) & \text{se } inlabel(u) = inlabel(p(u)) \\ ascendant(p(u)) + 2^i & \text{caso contrário.} \end{cases}$$

Note que a definição acima sugere uma estratégia simples para o cálculo do rótulo *ascendant* de cada vértice de T: percorrer T (uma única vez) em pré-ordem, testando para cada vértice u visitado as condições da definição acima e atribuindo o valor correspondente a $ascendant(u)$.

Para concluir a nossa descrição sobre a fase de pré-processamento do algoritmo, vamos falar agora sobre a tabela auxiliar *head*. Esta tabela nada mais é do que um vetor indexado pelo valor dos rótulos *inlabel* dos vértices da árvore T (note que o tamanho deste vetor é linear em $|V_T|$). Para cada i, $head[i]$ guarda o vértice da árvore T que se encontra mais próximo da raiz e cujo valor do rótulo *inlabel* é igual a i. Note que, desta forma,

$$head[i] = \begin{cases} \text{raiz de } T & \text{se } i = 2^l \\ v \text{ tal que } i = inlabel(v) \neq inlabel(p(v)) & \text{caso contrário.} \end{cases}$$

Assim, podemos facilmente preencher a tabela *head* em um percurso em pré-ordem da árvore T.

Consultas

Após realizar o pré-processamento da árvore T, como foi descrito no bloco anterior, é possível responder a consultas pelo ancestral comum mais próximo de quaisquer dois vértices de T em tempo constante. Dados dois vértices x e y de T, vamos descrever a partir de agora o algoritmo para encontrar o ancestral comum mais próximo de x e y.

Se $inlabel(x) = inlabel(y)$, então x e y se encontram em uma mesma cadeia em T e, desta forma, o ancestral comum mais próximo de x e y será x, se $level(x) \leq level(y)$, e será y, caso contrário. Agora, se $inlabel(x) \neq inlabel(y)$, então x e y se encontram em cadeias distintas em T e, neste caso, iremos encontrar o ancestral mais próximo de x e y em quatro passos:

1. Sejam C_x e C_y as cadeias em T em que se encontram, respectivamente, x e y. Este primeiro passo consiste em encontrar o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B. Mais precisamente, estamos interessados em encontrar a ordem em que tal vértice é visitado em um percurso em inordem de B. Note que, para isto, basta utilizar uma

versão levemente modificada do algoritmo *ACMP-BC* (visto no bloco 2) com B, $\Psi(C_x)$ e $\Psi(C_y)$ como argumentos, já que B é uma árvore binária completa (na implementação deste passo, a árvore B será dada implicitamente e o algoritmo receberá como argumentos $inlabel(x) = in(\Psi(C_x))$ e $inlabel(y) = in(\Psi(C_y))$).

2. Seja v o ancestral comum mais próximo de $\Psi(C_x)$ e $\Psi(C_y)$ em B. A partir de $in(v)$, vamos descobrir a cadeia em T em que se encontra z, o ancestral comum mais próximo de x e y. Mais especificamente, vamos descobrir o valor de $inlabel(z)$ o qual sabemos que é igual a $in(\Psi(C_z))$, onde C_z é a cadeia em T em que se encontra z. Note que, como nem todo ancestral comum de $\Psi(C_x)$ e $\Psi(C_y)$ em B está associado à cadeia de um ancestral comum de x e y em T, v não é necessariamente igual a $\Psi(C_z)$. Entretanto, não é difícil de concluir que $\Psi(C_z)$ é o ancestral mais próximo de v que está associado à cadeia de um ancestral comum de x e y em T (tal fato segue da propriedade da preservação da ascendência).

   Então, sejam i o índice do bit 1 mais à direita em $in(v)$, j o índice do bit 1 mais à direta em $inlabel(z)$ e $ascendant_{[l-i]}(x)$ e $ascendant_{[l-i]}(y)$ os números formados pelos $l-i$ primeiros bits de $ascendant(x)$ e $ascendant(y)$, respectivamente. Como $\Psi(C_z)$ é ancestral de v e $in(\Psi(C_z)) = inlabel(z)$, temos que $j \geq i$ e, desta forma, j é o índice de algum dos bits 1 de $ascendant_{[l-i]}(x) \wedge ascendant_{[l-i]}(y)$. Como o nível de $\Psi(C_z)$ é o mais baixo possível, sabemos que j é o índice do bit 1 mais à direita em $ascendant_{[l-i]}(x) \wedge ascendant_{[l-i]}(y)$. Por fim, note que uma vez que conseguimos determinar j, determinamos também o valor de $inlabel(z)$. Isso segue do fato que $inlabel(z)$ é o número formado pelos $l-j$ primeiros bits de $inlabel(x)$ seguidos por um bit 1 e j bits 0, já que $\Psi(C_z)$ é ancestral de $\Psi(C_x)$ em B, $in(\Psi(C_z)) = inlabel(z)$ e $in(\Psi(C_x)) = inlabel(x)$.

3. Agora que já sabemos em que cadeia de T o vértice z se encontra, vamos descobrir, neste passo, os vértices $\bar{x}$ e $\bar{y}$ que são, respectivamente, o ancestral mais próximo de x em C_z e o ancestral mais próximo de y em C_z. Aqui, descreveremos apenas como determinar $\bar{x}$. O procedimento para determinar $\bar{y}$ é análogo.

   Note que, se $inlabel(x) = inlabel(z)$, então x se encontra em C_z e, desta forma, $\bar{x} = x$. Caso contrário, seja w o ancestral de x cujo pai é $\bar{x}$. Como $\bar{x}$ é o ancestral mais próximo de x que se encontra em C_z, temos que $inlabel(w) \neq inlabel(\bar{x})$ e, desta forma, w é o vértice que se encontra no nível mais alto em sua cadeia. Logo, $w = head[inlabel(w)]$. Assim, tudo que precisamos fazer para determinar w e, desta forma, determinar $\bar{x}$, é descobrir o valor de $inlabel(w)$. Observe que, para isto, basta que calculemos i_w, o índice do bit 1 mais à direita em $inlabel(w)$, já que $inlabel(w)$ é o número formado pelos $l - i_w$ primeiros bits de $inlabel(x)$ seguidos por um bit 1 e i_w bits 0. É exatamente isso que vamos fazer agora. Seja $i_{\bar{x}}$ o índice do bit 1 mais à direita em $inlabel(\bar{x})$. Como w está na sub-árvore de T com raiz em $\bar{x}$ sabemos que o intervalo de w está contido no intervalo de $\bar{x}$ e, portanto, $i_{\bar{x}} > i_w$. Do mesmo modo, podemos concluir que para todo w' tal que w é ancestral de w', $i_w \geq i_{w'}$,

onde $i_{w'}$ é o índice do bit 1 mais à direita em $inlabel(w')$. Logo, i_w é o índice do bit 1 mais à esquerda em $ascendant_{i_{\bar{x}}}(x)$, número formado pelos $i_{\bar{x}}$ últimos bits de $ascendant(x)$.

4. Por fim, note que podemos determinar z facilmente a partir de $\bar{x}$ e $\bar{y}$. Para isto, basta observar que z é aquele, dentre estes dois vértices, que se encontra em um nível mais alto em T. Esta observação segue do fato que, como $\bar{x}$ e $\bar{y}$ se encontram em uma mesma cadeia em T, um dos dois (o que está no nível mais alto) é ancestral do outro e, portanto, é ancestral comum de x e y. Assim, pela maneira como definimos $\bar{x}$ e $\bar{y}$, temos que se $level(\bar{x}) > level(\bar{y})$ então $z = \bar{x}$ e, caso contrário, $z = \bar{y}$.

A.3 Implementação em CWEB

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

192 ⟨ Header files of lca.c 193 ⟩
⟨ Global variables of lca.c 197 ⟩
⟨ Internal functions of lca.c 198 ⟩
⟨ External functions of lca.c 196 ⟩

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.

193 ⟨ Header files of lca.c 193 ⟩ ≡
```
#include "gb_graph.h"
```

Veja também blocos 195, 202 e 208.

Este código é usado no bloco 192.

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.

194 ⟨ `lca.h` 194 ⟩ ≡
```
void lca_preprocessing(Vertex *r);
Vertex *lca(Vertex *u, Vertex *v);
```

195 ⟨ Header files of lca.c 193 ⟩ +≡
```
#include "lca.h"
```

A função *lca_preprocessing* recebe um ponteiro r para a raiz de uma árvore e preprocessa esta árvore, associando a cada vértice u três rótulos inteiros: $u\rightarrow level$, $u\rightarrow inlabel$ e $u\rightarrow ascendant$, 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, $r\rightarrow parent\_edge\rightarrow tip \equiv r$ e, se $v \neq r$, $v\rightarrow parent\_edge\rightarrow tip$ será um ponteiro para o pai de v na árvore. Por fim, para todo vértice v, o arco $v\rightarrow parent\_edge$ não deve fazer parte da lista de adjacência de v na árvore.

196
```
#define parent_edge  u.A
#define level  v.I
#define inlabel  x.I
#define ascendant  y.I
```
⟨ External functions of lca.c 196 ⟩ ≡
```
void lca_preprocessing(Vertex *r)
{
  ⟨ Compute the level and the inlabel number of each vertex 199 ⟩
  ⟨ Compute the ascendant number of each vertex and construct the head table 201 ⟩
}
```

Veja também bloco 204.

Este código é usado no bloco 192.

197 ⟨ Global variables of lca.c 197 ⟩ ≡
```
Vertex **head;
int max_inlabel = 0;
```

Este código é usado no bloco 192.

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.

198 ⟨ Internal functions of lca.c 198 ⟩ ≡

```
int msb(unsigned long x)
{
  register unsigned long b;
  register float f;
  if (x ≡ 0) return 0;
  f = (float) x;
  __asm__ ("mov␣␣␣%1,␣%0\n\t"
       "sar␣␣␣$0x17,␣%0\n\t"
       "sub␣␣␣$0x7F,␣%0\n\t"
       : "=r"(b)
       :"r"(f)
       ) ;
  return ((int) b);
}
```

Veja também blocos 200, 203 e 206.

Este código é usado no bloco 192.

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.

199 ⟨ Compute the level and the inlabel number of each vertex 199 ⟩ ≡

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

Este código é usado no bloco 196.

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*.

200 #**define** *preorder* *y.I*
#**define** *size* *z.I*

⟨ Internal functions of lca.c 198 ⟩ +≡

```
void label1 (Vertex *v)
{
```

```
  Arc *a;
  Vertex *w;
  int i, x;
  static long p;
  w = v→parent_edge→tip;
  v→level = w→level + 1;
  if (w ≡ v) p = 1;
  v→preorder = p++;
  v→size = 0;
  for (a = v→arcs; a; a = a→next) {
    label1(a→tip);
    v→size += a→tip→size;
  }
  (v→size)++;
  x = (v→preorder − 1) ⊕ (v→preorder + v→size − 1);
  i = msb(x);
  v→inlabel = ((v→preorder + v→size − 1) ≫ i) ≪ i;
  if (v→inlabel > max_inlabel) max_inlabel = v→inlabel;
}
```

Aqui é feito o cálculo do valor do rótulo *ascendant* de cada um dos vértices na árvore e, além disso, a tabela $head[0..max\_inlabel + 1]$ é criada e preenchida. A maior parte desse serviço é feita pela função *label2*, a qual descreveremos logo adiante.

201 ⟨ Compute the ascendant number of each vertex and construct the *head* table 201 ⟩ ≡

```
{
  Arc *a;
  head = (Vertex **) malloc((max_inlabel + 1) * sizeof(Vertex *));
  r→ascendant = r→inlabel;
  head[r→inlabel] = r;
  for (a = r→arcs; a; a = a→next) label2(a→tip);
}
```

Este código é usado no bloco 196.

202 ⟨ Header files of lca.c 193 ⟩ +≡

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

A função *label2* recebe um vértice v de uma árvore, da qual v não é raiz, e calcula o valor do rótulo *ancendant* de todo vértice que se encontra na sub-árvore com raiz em v. Além disso, esta função também preenche a tabela *head* como especificado no algoritmo descrito no bloco 4.

203 ⟨ Internal functions of lca.c 198 ⟩ +≡

```
void label2(Vertex *v)
{
  Arc *a;
  Vertex *w = v->parent_edge->tip;
  if (v->inlabel ≡ w->inlabel) v->ascendant = w->ascendant;
  else {
    int x = v->inlabel;
    v->ascendant = w->ascendant + (x - (x & (x - 1)));
    head[x] = v;
  }
  for (a = v->arcs; a; a = a->next) label2(a->tip);
}
```

A função *lca* recebe como argumentos dois vértices x e y de uma árvore e devolve o ancestral como mais próximo de x e y. Para tanto, tal árvore deve ter sido pré-processada anteriormente através de uma chamada à função *lca_preprocessing*.

204 ⟨ External functions of lca.c 196 ⟩ +≡

```
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 205 ⟩
  ⟨ Find and return the lca of x and y 207 ⟩
}
```

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.

205 ⟨ Find *lca_inlabel*, the inlabel number of the lca of x and y 205 ⟩ ≡

```
{
  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 204.

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.

206 ⟨ Internal functions of lca.c 198 ⟩ +≡

```
int cbt_lca(int x, int y)
{
  int i, j, k;
  i = msb(x - (x & (x - 1)));
  j = msb(y - (y & (y - 1)));
  k = ((i > j) ? i : j) + 1;
  if ((x >> k) ≠ (y >> k)) {
    k = msb(x ⊕ y);
    return ((x >> (k + 1)) << (k + 1)) + (1 << k);
  }
  else return ((i ≥ j) ? x : y);
}
```

Por fim, aqui é calculado o ancestral comum mais próximo de x e y na árvore. O trecho de código abaixo é uma implementação dos passos 3 e 4 do algoritmo descrito no bloco 5.

207 ⟨ Find and return the lca of x and y 207 ⟩ ≡

```
{
  Vertex *x_, *y_, *w;
  int l = msb(INT_MAX);      /* index of the leftmost bit of an int variable */
  int i_w, w_inlabel;
  if (x->inlabel ≡ lca_inlabel) x_ = x;
  else {
```

```
    i_w = msb(((unsigned)(x→ascendant ≪ (l + 1 − j))) ≫ (l + 1 − j));
    w_inlabel = ((x→inlabel ≫ (i_w + 1)) ≪ (i_w + 1)) + (1 ≪ i_w);
    w = head[w_inlabel];
    x_ = w→parent_edge→tip;
  }
  if (y→inlabel ≡ lca_inlabel) y_ = y;
  else {
    i_w = msb(((unsigned)(y→ascendant ≪ (l − j + 1))) ≫ (l − j + 1));
    w_inlabel = ((y→inlabel ≫ (i_w + 1)) ≪ (i_w + 1)) + (1 ≪ i_w);
    w = head[w_inlabel];
    y_ = w→parent_edge→tip;
  }
  return ((x_→level ≤ y_→level) ? x_ : y_);
}
```

Este código é usado no bloco 204.

208 ⟨ Header files of lca.c 193 ⟩ +≡

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

Apêndice B

Função *main* do programa

Vamos apresentar aqui a função *main* do programa que será responsável por executar, sobre a entrada, as funções correspondentes a cada implementação. Ao final de sua execução, o programa exibirá, para cada implementação, as arestas da floresta de Steiner construída, juntamente com um valor que limita superiormente a razão entre o custo desta floresta e o custo de uma solução ótima do problema.

209 ⟨ The main program 209 ⟩ ≡

```
int main(int argc, char *argv[])
{
  ⟨ Local variables of function main 212 ⟩
  ⟨ Process the comand line 210 ⟩
  ⟨ Initialize the field from of each edge 218 ⟩
  ⟨ Find a Steiner forest 221 ⟩
  ⟨ Show the results 224 ⟩
  return 0;
}
```

Este código é usado no bloco 1.

Basicamente, há duas formas de executar o programa. Uma delas é digitando o nome do programa ("`sf`") seguido pelo nome do arquivo que contém a especificação do grafo de entrada e dos conjuntos de terminais neste grafo (a qual deve seguir um formato que descreveremos mais a diante). Eis um exemplo de como executar o programa desta forma:

```
sf nome_do_arquivo_de_entrada
```

A segunda forma é utilizando a opção "`-sgb`". Esta opção permite que o usuário especifique o grafo de entrada através de um arquivo contendo a descrição de um grafo do SGB [?], isto é, através de um arquivo gerado pela função *save_graph* do módulo GB_SAVE do SGB. Neste caso, para executar o programa o usuário deve digitar algo semelhante ao seguinte, na linha de comando:

```
sf -sgb nome_do_arquivo_sgb nome_do_arquivo_de_entrada
```

onde `nome_do_arquivo_de_entrada` é o nome de um arquivo que contém a especificação dos conjuntos de terminais do grafo descrito no arquivo `nome_do_arquivo_sgb`.

É possível ainda especificar a implementação que o usuário deseja aplicar sobre a entrada. Para aplicar apenas a implementação sugerida por Goemans e Wiliamson, basta acrescentar "`-gw`" após o nome do arquivo ou dos arquivos que compõem a entrada. Da mesma forma, se o usuário deseja aplicar apenas a implementação proposta por Cole et al., ele deve acrescentar "`-chlp`" a linha de comando (exatamente como no caso anterior) e, em seguida, fornecer o valor do parâmetro k tomado por esta implementação (um inteiro maior ou igual a 1). Caso nenhuma dessas opções tenham sido especificadas, o programa irá aplicar cada uma das implementações sobre a entrada.

Como já mencionamos, ao final de sua execução, o programa irá exibir, para cada implementação aplicada sobre a entrada, uma lista das arestas que fazem parte da floresta de Steiner construída, juntamente com o custo desta floresta e um limitante superior para a razão entre este custo e o custo de uma floresta de Steiner ótima. Se o usuário já tiver conhecimento prévio do custo de uma floresta de Steiner ótima, é possível informa-lo ao programa através da opção "`-opt`". Para isto, deve-se digitar "`-opt cost`", onde "cost" é o custo de uma floresta de Steiner ótima, após o nome do arquivo ou dos arquivos que fazem parte da entrada do programa.

210 ⟨ Process the comand line 210 ⟩ ≡

```
{
  FILE *file;
  int next_arg;
  if (argc < 2) ⟨ Print a help message and exit 211 ⟩
  if (¬strcmp(argv[1], "-sgb")) {
    if (argc < 4) ⟨ Print a help message and exit 211 ⟩
    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 219 ⟩
    next_arg = 4;
  }
  else {
    if (¬(file = fopen(argv[1], "r"))) {
      fprintf(stderr, "Cannot␣open␣file␣%s.\n", argv[1]);
      exit(1);
    }
    ⟨ Try to create a graph from the file; exit if unsuccessful 215 ⟩
    next_arg = 2;
  }
  fclose(file);
  opt = −1;
  show_edges = 0;
  imp = ALL;
  while (argc − 1 ≥ next_arg) {
    if (¬strcmp(argv[next_arg], "-gw")) imp = GW;
    else {
      if (¬strcmp(argv[next_arg], "-chlp") ∧ argc − 1 ≥ next_arg + 1) {
        imp = CHLP;
        k = atoi(argv[++next_arg]);
        if (k < 1) {
          fprintf(stderr, "\n␣Invalid␣value␣for␣parameter\n");
          exit(1);
        }
      }
      else {
        if (¬strcmp(argv[next_arg], "-klein")) imp = KLEIN;
        else {
          if (¬strcmp(argv[next_arg], "-opt") ∧ argc − 1 ≥ next_arg + 1)
            opt = atoi(argv[++next_arg]);
          else {
            if (¬strcmp(argv[next_arg], "-e")) show_edges = 1;
            else {
              if (¬strcmp(argv[next_arg], "-t")) test = 1;
```

```
            else ⟨ Print a help message and exit 211 ⟩
          }
        }
      }
    }
  }
  next_arg++;
 }
}
```

Este código é usado no bloco 209.

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

211 ⟨ Print a help message and exit 211 ⟩ ≡

```
{
  fprintf(stderr,
      "\n␣sf␣[-sgb␣filename_gb]␣filename␣[-gw␣|␣-chlp␣k␣|␣-opt␣cost]\n\n");
  exit(1);
}
```

Este código é usado no bloco 210.

Se não ocorrerem erros, após o processamento da linha de comando do programa, o valor guardado na variável *imp* irá indicar qual ou quais das implementações deverão ser aplicadas sobre a entrada. Caso este valor seja igual a `CHLP`, então a variável k estará guardando o valor correspondente ao parâmetro tomado pela implementação de Cole et al. Vale a pena mencionar também que a variável *opt* será usada para guardar o custo de uma floresta de Steiner ótima, sempre que este custo for informado pelo usuário.

212 ⟨ Local variables of function *main* 212 ⟩ ≡

```
enum {
  ALL, GW, CHLP, KLEIN
} imp;
int k = 1, show_edges;
int test = 0;
long opt;
```

Veja também blocos 214, 220 e 223.

Este código é usado no bloco 209.

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.

213 ⟨ Header files of sf.c 71 ⟩ +≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_save.h"
```

Vamos declarar aqui mais uma das variáveis locais da função *main* que foi usada acima: *graph*. Se não ocorrer nenhum erro durante o processamento da linha de comando ou leitura dos arquivos de entrada, esta variável guardará um apontador para a estrutura que representa o grafo da entrada.

214 ⟨ Local variables of function *main* 212 ⟩ +≡

```
Graph *graph;
```

Para fornecer a especificação do grafo de entrada através de um arquivo, o usuário deve seguir as seguintes convenções. Na primeira linha, deve-se informar, o número n de vértices do grafo e, nas n linhas seguintes, digitar o rótulo ou nome (com no máximo 30 caracteres) de cada um dos vértices, um vértice por linha. Na linha seguinte, vem o número m de arestas do grafo; nas próximas m linhas do arquivo, o usuário deve digitar, para cada aresta uv, as posições (entre 0 e $n-1$) em que os vértices de rótulo u e v aparecem na lista dos vértices no arquivo e, em seguida, o custo da aresta. Cada uma dessas m linhas deve corresponder a definição de uma, e somente uma, aresta.

Os conjuntos de terminais do grafo devem ser especificados no arquivo de entrada do seguinte modo. Inicialmente, é necessário fornecer o número t de conjuntos de terminais no grafo. As próximas t linhas do arquivo devem informar os vértices que fazem parte de cada um dos conjuntos de terminais, sendo que cada linha deve corresponder a um, e somente um, conjunto. Se a opção `-sgb` estiver presente na linha de comando, cada vértice deverá ser especificado através de seu índice no vetor de vértices do grafo descrito no arquivo de entrada; caso contrário, os vértices deverão ser informados através de números inteiros (a partir de 0) correspondendo a ordem em que foram listados no arquivo de entrada.

Eis um exemplo de um arquivo de entrada satisfazendo as convenções acima:

```
5
A
B
C
D
E
7
0 1 15
2 3 12
3 4 18
2 0 24
0 4 30
1 3 28
4 1 32
2
0 3
1 4
```

O grafo especificado neste exemplo contém cinco vértices, sete arestas e dois conjuntos de terminais. Os vértices são *A, B, C, D* e *E*; as arestas são *(A, B)* (custo 15), *(C, D)* (custo 12), *(D, E)* (custo 18), *(C, A)* (custo 24), *(A, E)* (custo 30), *(D, B)* (custo 28) e *(E, B)* (custo 32); por fim, os conjuntos de terminais são *{A, D}* e *{B, E}*.

O trecho de código abaixo é responsável pela criação do grafo especificado no arquivo de entrada.

215 ⟨ Try to create a graph from the file; exit if unsuccessful 215 ⟩ ≡

```
{
  long num_vertices;
  if (¬fscanf(file, "␣%ld", &num_vertices)) {
    fprintf(stderr, "Cannot␣read␣the␣number␣of␣vertices.\n");
    exit(1);
  }
  if (¬(graph = gb_new_graph(num_vertices))) {
    fprintf(stderr, "Cannot␣create␣a␣graph␣with␣%ld␣vertices.\n", num_vertices);
    exit(1);
  }
  ⟨ Read the vertices from the file; exit if unsuccessful 216 ⟩
  ⟨ Read the edges from the file; exit if unsuccessful 217 ⟩
  ⟨ Read the terminal sets from the file; exit if unsuccessful 219 ⟩
```

```
}
```

Este código é usado no bloco 210.

216 ⟨ Read the vertices from the file; exit if unsuccessful 216 ⟩ ≡

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

Este código é usado no bloco 215.

217 ⟨ Read the edges from the file; exit if unsuccessful 217 ⟩ ≡

```
{
  long num_edges, i;
  if (¬fscanf(file, "␣%ld", &num_edges)) {
    fprintf(stderr, "Cannot␣read␣the␣number␣of␣edges.\n");
    exit(1);
  }
  for (i = 0; i < num_edges; i++) {
    long p, q, l;
    if (¬fscanf(file, "␣%ld␣%ld␣%ld", &p, &q, &l)) {
      fprintf(stderr, "Error␣in␣definition␣of␣edge␣%ld.\n", i + 1);
      exit(1);
    }
    gb_new_edge(graph→vertices + p, graph→vertices + q, l);
  }
}
```

Este código é usado no bloco 215.

Nas implementações que iremos descrever, será necessário ter acesso, de maneira rápida, aos extremos de uma dada aresta. É importante ressaltar que, no SGB, cada aresta uv é representada através dos arcos uv e vu. Como a estrutura que representa um arco contém apenas um apontador para o vértice de destino (ou seja, o vértice para o qual o arco aponta), será preciso pré-processar o grafo, inicializando, para cada arco, o campo *from* que contém um apontador para o vértice de origem do arco.

218 #**define** *from* *a.V*

⟨ Initialize the field *from* of each edge 218 ⟩ ≡

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

Este código é usado no bloco 209.

A leitura da especificação dos conjuntos de terminais e a criação das estruturas que os representam são feitas no trecho de código abaixo. Tais estruturas serão guardadas no vetor *term_sets* e o número de elementos deste vetor será armazenado no campo *num_ts* da estrutura que representa o grafo da entrada. Faremos com que o campo *id* de cada conjunto de terminais corresponda ao seu índice no vetor *term_sets*. Além disso, para cada vértice v do grafo, v→*termset_id* guardará o identificador do conjunto de terminais do qual o vértice faz parte (o valor deste campo será igual a -1 para os vértices que não pertencem aos conjuntos de terminais do grafo).

219 ⟨ Read the terminal sets from the file; exit if unsuccessful 219 ⟩ ≡

```
{
  int i;
  Vertex *v;
  if (¬fscanf(file, "␣%ld", &(graph→num_ts))) {
    fprintf(stderr, "Cannot␣read␣the␣number␣of␣terminal␣sets.\n");
    exit(1);
  }
  getc(file);
  for (v = graph→vertices; v < graph→vertices + graph→n; v++) v→termset_id = −1;
  term_sets = (TermSet *) malloc((graph→num_ts) * sizeof(TermSet));
  for (i = 0; i < graph→num_ts; i++) {
```

```
    TermSet *ts = term_sets + i;
    Vertex x, *t;
    char c;
    int n = 0;
    ts→id = i;
    t = ts→vertices = &x;
    do c = getc(file); while (c ≡ '␣');
    while (c ≠ EOF ∧ c ≠ '\n') {
      int v;
      ungetc(c, file);
      if (fscanf(file, "%d", &v) ∧ v ≥ 0 ∧ v < graph→n) {
        (graph→vertices + v)→termset_id = ts→id;
        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);
      }
      do c = getc(file); while (c ≡ '␣');
    }
    if (n ≡ 0) {
      fprintf(stderr, "There␣are␣less␣than␣%ld␣terminal␣sets␣in␣file␣%s.\n",
          graph→num_ts, (argc ≡ 4) ? argv[3] : argv[1]);
      exit(1);
    }
    t→next_terminal = Λ;
    ts→vertices = ts→vertices→next_terminal;
    ts→num_vertices = n;
    ts→connected = ((n ≡ 1) ? 1 : 0);
  }
}
```

Este código é usado nos blocos 210 e 215.

220 ⟨ Local variables of function *main* 212 ⟩ +≡

```
TermSet *term_sets;
```

221 ⟨ Find a Steiner forest 221 ⟩ ≡

```
{
  int i;
  long nk = graph→n;
  Vertex *v;
  TermSet *ts;
  if (graph→id) fprintf(stdout, "\nName: %s", graph→id);
  fprintf(stdout, "\nnumber of vertices: %ld", graph→n);
  fprintf(stdout, "\nnumber of edges: %ld", (graph→m)/2);
  fprintf(stdout, "\nnumber of terminal sets: %ld\n\n", graph→num_ts);
  switch (imp) {
  case GW: fprintf(stdout, "***** Goemans and Williamson *****\n");
    start = clock( );
    sf = sf_gw(graph, term_sets);
    end = clock( );
    break;
  case CHLP:
    fprintf(stdout, "***** Cole, Hariharan, Lewenstein and Porat *****\n");
    for (i = 1; i < k; i++) {
      if (LONG_MAX/nk < graph→n) {
        fprintf(stderr, "\nToo high value for parameter. ");
        break;
      }
      nk = nk * graph→n;
    }
    for (v = graph→vertices; v < graph→vertices + graph→n; v++) {
      Arc *a;
      for (a = v→arcs; a; a = a→next)
        if (LONG_MAX/nk < a→len) {
          if (LONG_MAX/nk ≥ graph→n)
            fprintf(stderr, "\nToo high value for parameter.\n\n");
          else fprintf(stderr, "\n\n");
          exit(1);
        }
    }
    if (i < k) fprintf(stderr, "Using k = %d.\n", i);
```

```
    k = i;
    start = clock();
    sf = sf_chlp(graph, term_sets, k);
    end = clock();
    break;
  case KLEIN: fprintf(stdout, "***** Klein *****\n");
    start = clock();
    sf = sf_klein(graph, term_sets);
    end = clock();
    break;
  case ALL: fprintf(stdout, "***** Goemans and Williamson *****\n");
    start = clock();
    sf = sf_gw(graph, term_sets);
    end = clock();
    ⟨ Show the results 224 ⟩
    for (ts = term_sets; ts < term_sets + graph→num_ts; ts++) ts→connected = 0;
    fprintf(stdout,
        "***** ""Cole, Hariharan, Lewenstein and Porat [k = %d]"" *****\n", k);
    start = clock();
    sf = sf_chlp(graph, term_sets, k);
    end = clock();
    ⟨ Show the results 224 ⟩
    for (ts = term_sets; ts < term_sets + graph→num_ts; ts++) ts→connected = 0;
    fprintf(stdout, "***** Klein *****\n");
    start = clock();
    sf = sf_klein(graph, term_sets);
    end = clock();
    break;
  }
}
```

Este código é usado no bloco 209.

222 ⟨ Header files of sf.c 71 ⟩ +≡

```
#include <time.h>
```

223 ⟨ Local variables of function *main* 212 ⟩ +≡

```
  SteinerForest *sf;
  clock_t start, end;
```

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.

224 ⟨ Show the results 224 ⟩ ≡

```
{
  Arc *a;
  double r;
  if (show_edges) {
    fprintf(stdout, "\nEdges:\n");
    for (a = sf→edges; a; a = a→next_edge)
      fprintf(stdout, "(%s, %s)\n", a→from→name, a→tip→name);
  }
  if (opt ≡ −1) r = ((double) sf→cost)/(sf→dual_cost);
  else r = ((double) sf→cost)/opt;
  fprintf(stdout, "\nCost: %ld [<= %.4f opt]\n", sf→cost, r);
  fprintf(stdout, "\nTime: %.3fs\n\n", ((double)(end − start))/CLOCKS_PER_SEC);
  if (test) {
    if (¬isSteinerForest(sf, graph, term_sets))
      printf("# Something bad has ocurred...\n\n");
    else {
      if (¬isMinimal(sf, graph, term_sets))
        printf("# Something bad has ocurred...\n\n");
      else printf("# It's everithing all right!\n\n");
    }
  }
}
```

Este código é usado nos blocos 209 e 221.

225

```
#define TESTE
#define mark a.I
#define next_elem w.V
#define _from_(i) ((g→vertices + i)→x.V)
```

⟨ Auxiliary functions 76 ⟩ +≡

```
int isSteinerForest(SteinerForest *sf, Graph *g, TermSet *term_sets)
{
  Arc *a;
  Vertex *v;
```

```
  TermSet *ts;
  int i;
  for (a = sf→edges, i = 0; a; a = a→next_edge, i++) _from_(i) = a→from;
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    v→visit = 0;
    for (a = v→arcs; a; a = a→next) a→mark = 0;
  }
  for (a = sf→edges, i = 0; a; a = a→next_edge, i++) {
    Arc *a_ = (_from_(i) < a→tip) ? a + 1 : a − 1;
    a→mark = a_→mark = 1;
  }     /* os terminais estão todos conectados? */
  for (ts = term_sets; ts < term_sets + g→num_ts; ts++) {
    Vertex *head, *tail;
    v = ts→vertices;
    v→visit = 1;
    v→next_elem = Λ;
    head = tail = v;
    while (head) {
      Vertex *u = head;
      for (a = u→arcs; a; a = a→next) {
        Vertex *w = a→tip;
        if (¬a→mark ∨ w→visit) continue;
        w→visit = 1;
        w→next_elem = Λ;
        tail→next_elem = w;
        tail = w;
      }
      head = head→next_elem;
    }
    for (v = ts→vertices; v; v = v→next_terminal)
      if (¬v→visit) {
#ifdef VERBOSE
        printf("-----> There are disconnected terminal sets\n");
#endif      /* reestabelecendo o valor do campo from dos arcos */
        for (v = g→vertices; v < g→vertices + g→n; v++) {
          Arc *a;
          for (a = v→arcs; a; a = a→next) a→from = v;
```

```
            }
            return 0;
          }
        for (v = g→vertices; v < g→vertices + g→n; v++) v→visit = 0;
      }     /* as arestas em sf são arestas de uma floresta? */
#ifdef DEFEITO2
      markAllEdges(g);
#endif
      for (v = g→vertices; v < g→vertices + g→n; v++) v→visit = 0;
      for (v = g→vertices; v < g→vertices + g→n; v++) {
        Vertex *head, *tail;
        if (v→visit) continue;
        v→visit = 1;
        v→next_elem = Λ;
        head = tail = v;
        while (head) {
          Vertex *u = head;
          int n = 0;
          for (a = u→arcs; a; a = a→next) {
            Vertex *w = a→tip;
            if (¬a→mark) continue;
            if (w→visit) {
              n++;
              continue;
            }
            w→visit = 1;
            w→next_elem = Λ;
            tail→next_elem = w;
            tail = w;
          }
          if (n ≥ 2) {
#ifdef VERBOSE
            printf("-----> There is a cycle in sf!\n");
#endif      /* reestabelecendo o valor do campo from dos arcos */
            for (v = g→vertices; v < g→vertices + g→n; v++) {
              Arc *a;
              for (a = v→arcs; a; a = a→next) a→from = v;
            }
```

```
            return 0;
          }
          head = head→next_elem;
        }
      }     /* reestabelecendo o valor do campo from dos arcos */
      for (v = g→vertices; v < g→vertices + g→n; v++) {
        Arc *a;
        for (a = v→arcs; a; a = a→next) a→from = v;
      }
      return 1;
    }
```

226 ⟨ Auxiliary functions 76 ⟩ +≡

```
    int isMinimal(SteinerForest *sf, Graph *graph, TermSet *term_sets)
    {
      int i, j, num_edges;
      Arc *a, **edges;
      printf("␣(Teste␣de␣Minimalidade)\n\n");
      num_edges = 0;
      for (a = sf→edges; a; a = a→next_edge) num_edges++;
#ifdef VERBOSE
      printf("\nnum_edges␣=␣%d\n\n", num_edges);
#endif
      edges = malloc(num_edges * sizeof(Arc *));
      for (i = 0, a = sf→edges; a; i++, a = a→next_edge) edges[i] = a;
      j = -1;
      do {
        j++;
        sf→edges = Λ;
        for (i = 0; i < j; i++) {
          edges[i]→next_edge = sf→edges;
          sf→edges = edges[i];
        }
        for (i = j + 1; i < num_edges; i++) {
          edges[i]→next_edge = sf→edges;
          sf→edges = edges[i];
        }
#ifdef VERBOSE
```

```
        printf("%d: ", j);
#endif
        if (isSteinerForest(sf, graph, term_sets)) {
          free(edges);
          return 0;
        }
      } while (j < num_edges - 1);
      free(edges);
      return 1;
    }
```

Referências Bibliográficas

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: an approximation algorithm for the generalizad Steiner problem on networks. *SIAM Journal on Computing*, 24(2):440–456, 1995.

[2] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.

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

[4] F. Chudak, T. Roughgarden, and D.P. Williamson. Approximate k-msts and k-Steiner trees via the primal-dual method and Lagrangean relaxation. In *Proc. 8th Conference on Integer Programming and Combinatorial Optimization Conference (IPCO)*, volume 2081 of *Lecture Notes in Computer Science*, page 60 ff. Springer, 2001.

[5] R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. A faster implementation of the Goemans-Williamson clustering algorithm. In *Symposium on Discrete Algorithms*, pages 17–25, 2001.

[6] H.N. Gabow, M.X. Goemans, and D.P. Williamson. An efficient approximation algorithm for the survivable network design problem. *Mathematical Programming*, 82(1-2, Ser. B):13–40, 1998.

[7] N. Garg, V.V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18:3–20, 1997.

[8] M.X. Goemans and D.P. Williamson. A general approximation technique for constrained forest problems. In G. Frederickson, R.L. Graham, D.S. Hochbaum, E. Johnson, S.R. Kosaraju, M. Luby, N. Megiddo, B. Schieber, P. Vaidya, and F. Yao, editors, *Proceedings of the Third ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 307–316. SIAM, 1992.

[9] M.X. Goemans and D.P. Williamson. Primal-dual approximation algorithms for feedback problems in planar graphs. Manuscript, 1995.

[10] D.S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, 1997.

[11] K. Jain, I. Măndoiu, V.V. Vazirani, and D.P. Williamson. A primal-dual schema based approximation algorithm for the element connectivity problem. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 484–489, 1999.

[12] K. Jain and V.V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. *17th International Symposium on Mathematical Programming (ISMP)*, 2000. URL: `http://www.cc.gatech.edu/people/home/kjain/`.

[13] P. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Information Processing Letters*, 52(6):303–307, 1994.

[14] D.E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, 1993.

[15] S. Rajagopalan and V.V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM Journal on Computing*, 28(2):526–541, 1999.

[16] B. Schieber. *Synthesis of Parallel Algorithms*, chapter 6, pages 259–273. Morgan Kaufmann Publishers, Inc., 1993.

[17] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[18] V.V. Vazirani. *Approximation Algorithms.* Springer, 2001.

Lista de Refinamentos

⟨ Discard all arcs vu from the graph 175 ⟩ Usado no bloco 172.
⟨ Discard parallel arcs in the *extra_ in* list of v 165 ⟩ Usado no bloco 163.
⟨ Discard parallel arcs in the *extra_ in* list of x 170 ⟩ Usado no bloco 168.
⟨ Discard unnecessary edges and return the resulting Steiner forest 138 ⟩ Usado no bloco 116.
⟨ Disjoint sets manipulation functions 63, 64, 65, 66 ⟩ Usado no bloco 59.
⟨ Drop out unnecessary edges and return the resulting Steiner forest 96 ⟩ Usado no bloco 77.
⟨ External functions of bicat.c 150, 151, 152, 153, 154, 161, 162, 163, 168, 181 ⟩ Usado no bloco 141.
⟨ External functions of lca.c 196, 204 ⟩ Usado no bloco 192.
⟨ Find a Steiner forest 221 ⟩ Usado no bloco 209.
⟨ Find and return the lca of x and y 207 ⟩ Usado no bloco 204.
⟨ Find the lca of each terminal set 100 ⟩ Usado no bloco 98.
⟨ Find *lca_ inlabel*, the inlabel number of the lca of x and y 205 ⟩ Usado no bloco 204.
⟨ For each bicategory b, join the heaps of u and v in b 176 ⟩ Usado no bloco 168.
⟨ For each edge a in g split a in two pieces 120 ⟩ Usado no bloco 116.
⟨ Free all the auxiliary memory allocated 97 ⟩ Usado no bloco 96.
⟨ Free the auxiliary memory allocated 139 ⟩ Usado no bloco 138.
⟨ Free the extra memory allocated 190 ⟩ Usado no bloco 184.
⟨ Global variables of bbst.c 23 ⟩ Usado no bloco 19.
⟨ Global variables of bicat.c 155, 158, 160 ⟩ Usado no bloco 141.
⟨ Global variables of fibheap.c 39 ⟩ Usado no bloco 35.
⟨ Global variables of item.c 7, 16 ⟩ Usado no bloco 3.
⟨ Global variables of lca.c 197 ⟩ Usado no bloco 192.
⟨ Global variables of uf.c 62 ⟩ Usado no bloco 59.
⟨ Header files of bbst.c 20 ⟩ Usado no bloco 19.
⟨ Header files of bicat.c 142 ⟩ Usado no bloco 141.
⟨ Header files of fibheap.c 36 ⟩ Usado no bloco 35.
⟨ Header files of item.c 4 ⟩ Usado no bloco 3.
⟨ Header files of lca.c 193, 195, 202, 208 ⟩ Usado no bloco 192.
⟨ Header files of sf.c 71, 74, 75, 102, 109, 113, 183, 213, 222 ⟩ Usado no bloco 1.
⟨ Header files of uf.c 60 ⟩ Usado no bloco 59.
⟨ Heaps manipulation functions 38, 40, 46, 47, 48, 49, 50, 51, 53, 54, 56, 57 ⟩ Usado no bloco 35.
⟨ Include an edge piece in the forest while there is some active component 126 ⟩ Usado no bloco 116.
⟨ Include each vertex of V in U and set *active_* to the new state of U 131 ⟩ Usado no bloco 130.
⟨ Include a in the forest sf 87 ⟩ Usado no bloco 84.
⟨ Include uv in the spanning forest sf 128 ⟩ Usado no bloco 126.
⟨ Include uv in sf 187 ⟩ Usado no bloco 184.
⟨ Increment $d(v)$ for each vertex v in some active component 88 ⟩ Usado no bloco 84.

⟨ Initialize the field *from* of each edge 218 ⟩ Usado no bloco 209.
⟨ Initialize $d(v)$ for each vertex v of the graph 82 ⟩ Usado no bloco 77.
⟨ Insert each vertex of `C2` in `C1` and set *active_* to the new state of `C1` 90 ⟩ Usado no bloco 89.
⟨ Internal functions of bbst.c 25, 26, 27 ⟩ Usado no bloco 19.
⟨ Internal functions of bicat.c 146, 147, 148, 177 ⟩ Usado no bloco 141.
⟨ Internal functions of fibheap.c 41, 42, 43, 44, 45, 52, 55, 58 ⟩ Usado no bloco 35.
⟨ Internal functions of item.c 9 ⟩ Usado no bloco 3.
⟨ Internal functions of lca.c 198, 200, 203, 206 ⟩ Usado no bloco 192.
⟨ Itens manipulation functions 6, 8, 10, 11, 12, 13, 14, 15 ⟩ Usado no bloco 3.
⟨ Let a be an edge with the smallest slackness *inc* 86 ⟩ Usado no bloco 84.
⟨ Local variables of function *main* 212, 214, 220, 223 ⟩ Usado no bloco 209.
⟨ Local variables of *contractEdge* 169, 171, 174, 179 ⟩ Usado no bloco 168.
⟨ Local variables of *sf_ chlp* 119, 121, 123, 125, 127 ⟩ Usado no bloco 116.
⟨ Local variables of *sf_ gw* 79, 81, 83, 85 ⟩ Usado no bloco 77.
⟨ Merge the components `C1` and `C2` 89 ⟩ Usado no bloco 84.
⟨ Merge the edge heaps of `C1` and `C2` 94 ⟩ Usado no bloco 89.
⟨ More auxiliary functions 99, 104 ⟩ Usado no bloco 98.
⟨ Multiply by n^k the cost of each edge in g 117 ⟩ Usado no bloco 116.
⟨ Print a help message and exit 211 ⟩ Usado no bloco 210.
⟨ Process the comand line 210 ⟩ Usado no bloco 209.
⟨ Read the edges from the file; exit if unsuccessful 217 ⟩ Usado no bloco 215.
⟨ Read the terminal sets from the file; exit if unsuccessful 219 ⟩ Usado nos blocos 210 e 215.
⟨ Read the vertices from the file; exit if unsuccessful 216 ⟩ Usado no bloco 215.
⟨ Restore the original costs of the edges of g 118 ⟩ Usado no bloco 116.
⟨ Set *cat* to the state of the component in *sf* that contains uv 188 ⟩ Usado no bloco 184.
⟨ Set *sf* to the initial spanning forest 186 ⟩ Usado no bloco 184.
⟨ Set uv and wv to represent the same terminal edge 136 ⟩ Usado no bloco 135.
⟨ Set uv and wv to represent two external edges 137 ⟩ Usado no bloco 135.
⟨ Set uv to point to an edge with the smallest slackness 129 ⟩ Citado no bloco 135. Usado no bloco 126.
⟨ Set u to represent the vertex resulting from the contraction of a 180 ⟩ Usado no bloco 168.
⟨ Show the results 224 ⟩ Usado nos blocos 209 e 221.
⟨ Split the edge wv 135 ⟩ Usado no bloco 126.
⟨ Steiner forest construction functions 77, 116, 184 ⟩ Usado no bloco 1.
⟨ The main program 209 ⟩ Usado no bloco 1.
⟨ Trees manipulation functions 22, 24, 28, 29, 32, 33 ⟩ Usado no bloco 19.
⟨ Try to create a graph from the file; exit if unsuccessful 215 ⟩ Usado no bloco 210.
⟨ Unite the components U and V 130 ⟩ Usado no bloco 126.

⟨ Unite the edge heaps of U and V 133 ⟩ Usado no bloco 130.
⟨ While there is some active component include an edge in the forest 84 ⟩ Usado no bloco 77.
⟨ `bbst.h` 18 ⟩
⟨ `bicat.h` 182 ⟩
⟨ `fibheap.h` 34 ⟩
⟨ find the lca of each terminal set in the list *ts_list* 101 ⟩ Usado no bloco 100.
⟨ `item.h` 17 ⟩
⟨ `lca.h` 194 ⟩
⟨ select the edges in the path from z to w in *sf_graph* 105 ⟩ Usado no bloco 103.
⟨ `uf.h` 67 ⟩