

Fluxos máximos  
Método dos caminhos de aumento  
Caminhos de maior aumento

Juliana Barby Simão  
APOIO FINANCEIRO DA FAPESP  
PROCESSO 04/00580-8

Marcelo Hashimoto  
APOIO FINANCEIRO DA FAPESP  
PROCESSO 04/00581-4

ORIENTADOR: José Coelho de Pina

**Sumário**

1. Introdução . . . . .	2
2. Descrição . . . . .	2
3. Compilação e execução . . . . .	2
4. Referências . . . . .	2
5. Método dos caminhos de aumento . . . . .	3
8. Caminhos de maior aumento . . . . .	4
11. Aumento do fluxo através do caminho . . . . .	5
13. Fila de prioridade . . . . .	7
14. Função principal . . . . .	9
15. Consistência dos parâmetros . . . . .	9
19. Impressão do fluxo de intensidade máxima . . . . .	10
20. Impressão do separador de capacidade mínima . . . . .	11
22. Estrutura geral . . . . .	12
23. Bibliotecas . . . . .	12
24. Macros . . . . .	12

## 1. Introdução

Esta é uma implementação em CWEB-L<sup>A</sup>T<sub>E</sub>X do **algoritmo dos caminhos de maior aumento**, uma versão do **método dos caminhos de aumento** para resolver o **problema do fluxo máximo**. A plataforma SGB é necessária.

## 2. Descrição

Este programa recebe o nome de um arquivo que contém um grafo no formato SGB, o nome de um arquivo de saída, o nome de um vértice fonte e o nome de um vértice sorvedouro e imprime no arquivo de saída um fluxo de intensidade máxima e um separador de capacidade mínima da rede representada pelo grafo. Assume-se que as capacidades dos arcos estão representadas no campo *len*.

## 3. Compilação e execução

```
make maioraumento.tex para gerar o arquivo LATEX de documentação.  
make maioraumento.dvi para gerar o arquivo DVI de visualização.  
make maioraumento.pdf para gerar o arquivo PDF de visualização.  
make maioraumento.ps para gerar o arquivo PostScript de visualização.  
make maioraumento.c para gerar o código-fonte C do programa.  
make maioraumento para gerar o executável do programa.  
maioraumento para executar o programa.
```

## 4. Referências

Sobre a plataforma SGB:

<http://www-cs-faculty.stanford.edu/~knuth/sgb.html>

Sobre a linguagem de *literate programming* CWEB-L<sup>A</sup>T<sub>E</sub>X:

<http://www-cs-faculty.stanford.edu/~knuth/cweb.html>

## 5. Método dos caminhos de aumento

O método dos caminhos de aumento começa a partir de um fluxo inicial e em cada iteração encontra um caminho de aumento relativo ao fluxo atual e aumenta a intensidade do fluxo através desse caminho. O método pára quando não há mais caminhos de aumento. Nesta implementação, após uma busca por caminhos de aumento, todo vértice acessível a partir da fonte através de caminhos alternantes passa a ter um arco predecessor definido. Logo, a execução termina quando o sorvedouro não tem um arco predecessor definido. Devido ao interesse na complexidade experimental do algoritmo, imprime-se o número total de iterações após a execução. Como a rede originalmente não contém arcos irmãos, eles devem ser construídos antes para obtermos a rede residual.

```
<Algoritmo dos caminhos de maior aumento 5> ≡
void maioraumento(Graph *g, Vertex *fonte, Vertex *sorvedouro)
{
  <Variáveis da função maioraumento 12>
  <Obtém fluxo inicial 6>
  <Constrói arcos irmãos 7>
  iteracoes = 0;
  do {
    <Encontra caminho de aumento 8>
    <Aumenta fluxo através do caminho de aumento 11>
    iteracoes++;
  } while (sorvedouro->arcopred ≠ Λ);
  fprintf(stdout, "número de iterações: %d\n", iteracoes - 1);
  return;
}
```

Este código é usado no bloco 22.

6. O fluxo inicial  $x$  é tal que  $x_a = 0$  para todo arco  $a$ .

```
<Obtém fluxo inicial 6> ≡
for (i = g->vertices; i < g->vertices + g->n; i++) {
  for (a = i->arcs; a; a = a->next) {
    a->flx = 0;
  }
}
```

Este código é usado no bloco 5.

7. Os arcos irmãos são construídos exatamente segundo sua definição. Nesta implementação, os arcos irmãos são reconhecidos por terem fluxo negativo.

```
<Constrói arcos irmãos 7> ≡
for (i = g->vertices; i < g->vertices + g->n; i++) {
  for (a = i->arcs; a; a = a->next) {
    if (a->flx ≥ 0) {
```

```

        j = a→tip;
        gb_new_arc(j, i, a→cap);
        a→irmao = j→arcs;
        a→irmao→flx = -1;
        a→irmao→irmao = a;
    }
}
}

```

Este código é usado no bloco 5.

## 8. Caminhos de maior aumento

O algoritmo dos caminhos de maior aumento sempre busca o caminho de aumento com a maior capacidade residual possível. Nesta implementação, isto é feito organizando-se os vértices em uma fila com prioridade. Executa-se uma busca a partir do vértice fonte, na qual os vértices com maior prioridade sempre são verificados primeiro. A prioridade de cada vértice é definida pela capacidade residual do caminho entre a fonte e ele. Portanto, inicialmente a fila só tem um elemento: o próprio vértice fonte cuja prioridade é trivialmente zero.

```

⟨Encontra caminho de aumento 8⟩ ≡
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    i→arcopred = Λ;
    i→estado = NAOVISTO;
  }
  inicializafila(g);
  inserenafile(fonte, 0);
  fonte→estado = VISITADO;
  while (¬filavazia()) {
    i = tiradafila();
    i→estado = EXAMINADO;
    ⟨Examina vértice retirado da fila 9⟩
  }
  finalizafile();

```

Este código é usado no bloco 5.

**9.** Ao examinar um vértice, visita-se seus vizinhos na rede residual que ainda não foram examinados. Ao invés de manter uma estrutura de dados separada para a rede residual, mantemos esta implícita. Para tanto, basta que a busca considere apenas os arcos com capacidade residual positiva. Utiliza-se uma variável temporária para armazenar a capacidade residual dos arcos.

```

⟨Examina vértice retirado da fila 9⟩ ≡
  for (a = i→arcs; a; a = a→next) {
    j = a→tip;
    if (j→estado ≠ EXAMINADO) {

```

```

    if ( $\alpha \rightarrow flx \geq 0$ )  $temp = \alpha \rightarrow cap - \alpha \rightarrow flx$ ;
    else  $temp = \alpha \rightarrow irmao \rightarrow flx$ ;
    if ( $temp > 0$ ) {
       $\langle$ Visita vértice vizinho 10 $\rangle$ 
    }
  }
}

```

Este código é usado no bloco 8.

10. Se o vértice visitado não está na fila, ele é inserido com a prioridade obtida a partir da capacidade residual do caminho da fonte até ele. Se ele está na fila, sua prioridade é atualizada caso o caminho da fonte até ele encontrado na iteração atual tenha capacidade residual superior ao anterior.

```

 $\langle$ Visita vértice vizinho 10 $\rangle \equiv$ 
  if ( $i \neq fonte \wedge temp > i \rightarrow res$ ) {
     $temp = i \rightarrow res$ ;
  }
  if ( $j \rightarrow estado \equiv$  NAOVISTO) {
     $inserenafile(j, temp)$ ;
     $j \rightarrow estado =$  VISITADO;
     $j \rightarrow arcopred = a$ ;
  }
  else if ( $temp > j \rightarrow res$ ) {
     $reinserenafile(j, temp)$ ;
     $j \rightarrow arcopred = a$ ;
  }
}

```

Este código é usado no bloco 9.

## 11. Aumento do fluxo através do caminho

Como cada vértice acessível a partir da fonte através de um caminho alternante tem um arco predecessor definido, o aumento do fluxo é simples: basta percorrer o caminho a partir do sorvedouro, valendo-se dos arcos predecessores, modificando o fluxo em cada arco de acordo com a rede na qual ele se encontra.

```

 $\langle$ Aumenta fluxo através do caminho de aumento 11 $\rangle \equiv$ 
   $a = sorvedouro \rightarrow arcopred$ ;
  while ( $a \neq \Lambda$ ) {
    if ( $\alpha \rightarrow flx \geq 0$ )  $\alpha \rightarrow flx = \alpha \rightarrow flx + sorvedouro \rightarrow res$ ;
    else  $\alpha \rightarrow irmao \rightarrow flx = \alpha \rightarrow irmao \rightarrow flx - sorvedouro \rightarrow res$ ;
    if ( $\alpha \rightarrow inicio \equiv fonte$ )  $a = \Lambda$ ;
    else  $a = \alpha \rightarrow inicio \rightarrow arcopred$ ;
  }
}

```

Este código é usado no bloco 5.

**12.** Como toda a função foi definida, podemos declarar as variáveis.

⟨Variáveis da função *maioraument* 12⟩ ≡

```
int iteracoes, temp;
```

```
Vertex *i, *j;
```

```
Arc *a;
```

Este código é usado no bloco 5.

### 13. Fila de prioridade

A estrutura de dados aqui utilizada para implementar a fila de prioridade é um *heap*. A implementação abaixo é totalmente baseada no livro *Introduction to Algorithms* de T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein.

⟨Fila de prioridade 13⟩ ≡

```
Vertex **heap;
int size;
int parent(int k)
{
    return (k/2);
}
int left(int k)
{
    return (2 * k);
}
int right(int k)
{
    return ((2 * k) + 1);
}
void exchange(int k1, int k2)
{
    Vertex *i;
    i = heap[k1];
    heap[k1] = heap[k2];
    heap[k2] = i;
    heap[k1]→indice = k1;
    heap[k2]→indice = k2;
    return;
}
void heapify(int k)
{
    int l, r, maior;
    l = left(k);
    r = right(k);
    if (l ≤ size ∧ heap[l]→res > heap[k]→res) maior = l;
    else maior = k;
    if (r ≤ size ∧ heap[r]→res > heap[maior]→res) maior = r;
    if (maior ≠ k) {
        exchange(k, maior);
        heapify(maior);
    }
    return;
}
```

```

void inicializafila(Graph * g) { heap = (Vertex **) malloc ( (g→n + 1) *
    sizeof (Vertex *) );
    size = 0;
return; } void finalizafila()
{
    free(heap);
return;
}
boolean filavazia()
{
    if (size ≡ 0) return (TRUE);
return (FALSE);
}
Vertex * retiradafila()
{
    Vertex * i;
    i = heap[1];
    heap[1] = heap[size];
    heap[1]→indice = 1;
    size --;
    heapify(1);
return (i);
}
void reinserenafila(Vertex * i, int key)
{
    int k;
    k = i→indice;
    i→res = key;
    while (k > 1 ∧ heap[parent(k)]→res < heap[k]→res) {
        exchange(k, parent(k));
        k = parent(k);
    }
return;
}
void inserenafila(Vertex * i, int key)
{
    size ++;
    heap[size] = i;
    i→indice = size;
    reinserenafila(i, key);
return;
}

```

Este código é usado no bloco 22.



#### 14. Função principal

O programa consiste de três fases: inicialização, execução do algoritmo e finalização. A inicialização consiste em verificar a consistência dos parâmetros de entrada. A aplicação do algoritmo é simplesmente a chamada da função que já definimos anteriormente. A finalização consiste em imprimir no arquivo de saída o fluxo máximo obtido e o separador de capacidade mínima.

```
< Função principal 14 > ≡
int main(int argc, char *argv[])
{
    Graph *g;
    Vertex *fonte, *sorvedouro;
    < Variáveis secundárias da função principal 21 >
    < Verifica consistência dos parâmetros 15 >
    maioramento(g, fonte, sorvedouro);
    < Imprime fluxo máximo 19 >
    < Imprime separador de capacidade mínima 20 >
    return (0);
}
```

Este código é usado no bloco 22.

#### 15. Consistência dos parâmetros

Para que o programa seja executado corretamente, exige-se que o nome de arquivo fornecido referencie um grafo válido no formato SGB, onde o campo *len* dos arcos corresponde à capacidade. Também é necessário que os nomes de vértices referenciem vértices que de fato existem no grafo e que a rede contenha somente capacidades não-negativas. Caso um número de parâmetros incorreto seja fornecido, as instruções do programa são impressas, exibindo a sintaxe.

```
< Verifica consistência dos parâmetros 15 > ≡
if (argc ≠ 5) {
    fprintf(stderr, "%s<in>_<out>_\"source\"_\"sink\"\\n", argv[0]);
    exit(-1);
}
< Verifica validade dos arquivos 16 >
< Verifica existência dos vértices 17 >
< Verifica sinal das capacidades 18 >
```

Este código é usado no bloco 14.

16. O programa utiliza as funções padrão para abrir o arquivo desejado. Caso o arquivo não possa ser aberto, o programa é imediatamente interrompido.

```

< Verifica validade dos arquivos 16 > ≡
  if ((g = restore_graph(argv[1])) ≡ Λ) {
    fprintf(stderr, "entrada_inválida\n");
    exit(-2);
  }
  if ((saida = fopen(argv[2], "w")) ≡ Λ) {
    fprintf(stderr, "saída_inválida\n");
    exit(-3);
  }

```

Este código é usado no bloco 15.

**17.** Os vértices do grafo são examinados um por um até que os nomes fornecidos sejam encontrados. No caso de nomes iguais, considera-se o primeiro.

```

< Verifica existência dos vértices 17 > ≡
  fonte = Λ;
  sorvedouro = Λ;
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    if (¬strcmp(i→name, argv[3])) fonte = i;
    if (¬strcmp(i→name, argv[4])) sorvedouro = i;
  }
  if (fonte ≡ Λ ∨ sorvedouro ≡ Λ) {
    fprintf(stderr, "vértices_inválidos\n");
    exit(-4);
  }

```

Este código é usado no bloco 15.

**18.** Os arcos do grafo são examinados um por um. O programa é interrompido imediatamente se um arco com capacidade negativa for encontrado.

```

< Verifica sinal das capacidades 18 > ≡
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (a→cap < 0) {
        fprintf(stderr, "capacidades_negativas\n");
        exit(-5);
      }
    }
  }

```

Este código é usado no bloco 15.

## 19. Impressão do fluxo de intensidade máxima

Após a execução do algoritmo, imprime-se o fluxo e a intensidade.

```

<Imprime fluxo máximo 19> ≡
for (max = 0, i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
        if (a→flx ≥ 0) {
            fprintf(saida, "fluxo de \s\ "a\s\ ": \ld\n", i→name,
                a→tip→name, a→flx);
            if (i ≡ sorvedouro) max -= a→flx;
            if (a→tip ≡ sorvedouro) max += a→flx;
        }
    }
}
fprintf(saida, "intensidade: \d\n", max);

```

Este código é usado no bloco 14.

## 20. Impressão do separador de capacidade mínima

O separador de capacidade mínima contém os vértices acessíveis a partir da fonte através de caminhos alternantes, ou seja, os vértices examinados.

```

<Imprime separador de capacidade mínima 20> ≡
fprintf(saida, "separador: \n");
for (min = 0, i = g→vertices; i < g→vertices + g→n; i++) {
    if (i→estado ≠ NAOVISTO) {
        fprintf(saida, "\s\ \n", i→name);
        for (a = i→arcs; a; a = a→next) {
            if (a→flx ≥ 0 ∧ a→tip→estado ≡ NAOVISTO) min += a→cap;
        }
    }
}
fprintf(saida, "capacidade: \d\n", min);
fclose(saida);

```

Este código é usado no bloco 14.

## 21. Podemos agora definir as variáveis secundárias da função principal.

```

<Variáveis secundárias da função principal 21> ≡
    Vertex * i;
    Arc * a;
    int min, max;
    FILE *saida;

```

Este código é usado no bloco 14.

## 22. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

```
< Bibliotecas necessárias 23 >  
< Fila de prioridade 13 >  
< Algoritmo dos caminhos de maior aumento 5 >  
< Função principal 14 >
```

## 23. Bibliotecas

Além das bibliotecas básicas, é preciso usar a plataforma SGB.

```
< Bibliotecas necessárias 23 > ≡  
#include <stdio.h>  
#include <stdlib.h>  
#include <gb_graph.h>  
#include <gb_save.h>
```

Este código é usado no bloco 22.

## 24. Macros

Definimos aqui todas as macros utilizadas no programa.

```
#define boolean int  
#define FALSE 0  
#define TRUE 1  
#define NAOVISTO 0  
#define VISITADO 1  
#define EXAMINADO 2  
#define arcopred u.A  
#define res v.I  
#define estado w.I  
#define indice x.I  
#define cap len  
#define flx a.I  
#define irmao b.A  
#define inicio irmao-tip
```

## Índice Remissivo

*Arc*: 12, 21.  
*arcopred*: 5, 8, 10, 11, 24.  
*arcs*: 6, 7, 9, 18, 19, 20.  
*argc*: 14, 15.  
*argv*: 14, 15, 16, 17.  
*boolean*: 13, 24.  
*cap*: 7, 9, 18, 20, 24.  
*estado*: 8, 9, 10, 20, 24.  
EXAMINADO: 8, 9, 24.  
*exchange*: 13.  
*exit*: 15, 16, 17, 18.  
FALSE: 13, 24.  
*fclose*: 20.  
*filavazia*: 8, 13.  
*finalizafila*: 8, 13.  
*flx*: 6, 7, 9, 11, 19, 20, 24.  
*fonte*: 5, 8, 10, 11, 14, 17.  
*fopen*: 16.  
*fprintf*: 5, 15, 16, 17, 18, 19, 20.  
*free*: 13.  
*gb\_new\_arc*: 7.  
*Graph*: 5, 13, 14.  
*heap*: 13.  
*heapify*: 13.  
*indice*: 13, 24.  
*inicializafila*: 8, 13.  
*inicio*: 11, 24.  
*inserenafila*: 8, 10, 13.  
*irmao*: 7, 9, 11, 24.  
*iteracoes*: 5, 12.  
*k*: 13.  
*key*: 13.  
*k1*: 13.  
*k2*: 13.  
*l*: 13.  
*left*: 13.  
*len*: 24.  
*main*: 14.  
*maior*: 13.  
*maioraument*: 5, 14.  
*malloc*: 13.  
*max*: 19, 21.  
*min*: 20, 21.  
*name*: 17, 19, 20.  
NAOVISTO: 8, 10, 20, 24.  
*next*: 6, 7, 9, 18, 19, 20.  
*parent*: 13.  
*r*: 13.  
*reinserenafila*: 10, 13.  
*res*: 10, 11, 13, 24.  
*restore\_graph*: 16.  
*retiradafila*: 8, 13.  
*right*: 13.  
*saida*: 16, 19, 20, 21.  
*size*: 13.  
*sorvedouro*: 5, 11, 14, 17, 19.  
*stderr*: 15, 16, 17, 18.  
*stdout*: 5.  
*strcmp*: 17.  
*temp*: 9, 10, 12.  
*tip*: 7, 9, 19, 20, 24.  
TRUE: 13, 24.  
*Vertex*: 5, 12, 13, 14, 21.  
*vertices*: 6, 7, 8, 17, 18, 19, 20.  
VISITADO: 8, 10, 24.

## Lista de Refinamentos

- ⟨ Algoritmo dos caminhos de maior aumento 5 ⟩ Usado no bloco 22.
- ⟨ Aumenta fluxo através do caminho de aumento 11 ⟩ Usado no bloco 5.
- ⟨ Bibliotecas necessárias 23 ⟩ Usado no bloco 22.
- ⟨ Constrói arcos irmãos 7 ⟩ Usado no bloco 5.
- ⟨ Encontra caminho de aumento 8 ⟩ Usado no bloco 5.
- ⟨ Examina vértice retirado da fila 9 ⟩ Usado no bloco 8.
- ⟨ Fila de prioridade 13 ⟩ Usado no bloco 22.
- ⟨ Função principal 14 ⟩ Usado no bloco 22.
- ⟨ Imprime fluxo máximo 19 ⟩ Usado no bloco 14.
- ⟨ Imprime separador de capacidade mínima 20 ⟩ Usado no bloco 14.
- ⟨ Obtém fluxo inicial 6 ⟩ Usado no bloco 5.
- ⟨ Variáveis da função *maioraument* 12 ⟩ Usado no bloco 5.
- ⟨ Variáveis secundárias da função principal 21 ⟩ Usado no bloco 14.
- ⟨ Verifica consistência dos parâmetros 15 ⟩ Usado no bloco 14.
- ⟨ Verifica existência dos vértices 17 ⟩ Usado no bloco 15.
- ⟨ Verifica sinal das capacidades 18 ⟩ Usado no bloco 15.
- ⟨ Verifica validade dos arquivos 16 ⟩ Usado no bloco 15.
- ⟨ Visita vértice vizinho 10 ⟩ Usado no bloco 9.