

Fluxos máximos
Método do pré-fluxo
Vértices ativos de maior rótulo

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 do pré-fluxo	3
10. Execução de um relabel	5
11. Execução de um push	5
13. Lista de vértices	6
14. Função principal	7
15. Consistência dos parâmetros	7
19. Impressão do fluxo de intensidade máxima	8
20. Impressão do separador de capacidade mínima	9
22. Estrutura geral	10
23. Bibliotecas	10
24. Macros	10

1. Introdução

Esta é uma implementação em `CWEB-LATEX` do **algoritmo dos vértices ativos de maior rótulo**, uma versão do **método do pré-fluxo** para resolver o **problema do fluxo máximo**. A plataforma SGB é necessária para a execução.

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

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

5. Método do pré-fluxo

O método do pré-fluxo começa a partir de um pré-fluxo inicial e em cada iteração escolhe um vértice ativo para sofrer uma operação push ou uma operação relabel. O método pára quando a lista de vértices ativos está vazia. 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 vértices ativos de maior rótulo 5 > ≡
void maiorrotulo(Graph * g, Vertex * fonte, Vertex * sorvedouro)
{
  < Variáveis da função maiorrotulo 12 >
  < Executa pré-processamento 6 >
  < Constrói arcos irmãos 7 >
  iteracoes = 0;
  inicializalista(g);
  < Insere vértices iniciais 8 >
  while (!listavazia()) {
    i = retiradalista();
    < Executa push ou relabel 9 >
    iteracoes++;
  }
  finalizalista();
  fprintf(stdout, "número de iterações: %d\n", iteracoes);
  return;
}

```

Este código é usado no bloco 22.

6. O pré-processamento consiste em atribuir o pré-fluxo inicial e os rótulos iniciais. O pré-fluxo inicial x é tal que $x_a = u_a$ para todo arco a que sai do vértice fonte e $x_a = 0$ para todo arco a restante do grafo. A função distância inicial d é tal que $d(\text{fonte}) = n$ e $d(i) = 0$ para todo vértice i restante.

```

< Executa pré-processamento 6 > ≡
for (i = g→vertices; i < g→vertices + g→n; i++) {
  i→dist = 0;
  i→exc = 0;
  i→atual = i→arcs;
}
for (i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (i ≡ fonte ∧ a→tip ≠ fonte) a→flx = a→cap;
    else a→flx = 0;
    i→exc -= a→flx;
    a→tip→exc += a→flx;
  }
}

```

```

}
fonte→dist = g→n;

```

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. Os vértices ativos obtidos no pré-processamento são adicionados à lista. Deve-se adicionar uma condição extra para não adicionar o sorvedouro.

```

⟨ Inere vértices iniciais 8 ⟩ ≡
for (i = g→vertices; i < g→vertices + g→n; i++) {
  if (i ≠ sorvedouro ∧ i→exc > 0) inserenalista(i);
}

```

Este código é usado no bloco 5.

9. Se o vértice ativo i examinado é origem de algum arco admissível a , executa-se um push no arco a . Senão, executa-se um relabel no vértice i .

```

⟨ Executa push ou relabel 9 ⟩ ≡
for (a = i→atual; a; a = a→next) {
  if (a→flx ≥ 0) temp = a→cap - a→flx;
  else temp = a→irmao→flx;
  if (temp > 0 ∧ i→dist ≡ a→tip→dist + 1) break;
}
if (a ≡ Λ) {
  ⟨ Executa um relabel 10 ⟩
  i→atual = i→arcs;
}
else {
  ⟨ Executa um push 11 ⟩
}

```

```

     $i \rightarrow atual = a;$ 
}

```

Este código é usado no bloco 5.

10. Execução de um relabel

Para executar um relabel é necessário visitar todos os vizinhos do vértice examinado i na rede residual. 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.

```

⟨ Executa um relabel 10 ⟩ ≡
for ( $min = -1, a = i \rightarrow arcs; a; a = a \rightarrow next$ ) {
    if ( $a \rightarrow flx \geq 0$ )  $temp = a \rightarrow cap - a \rightarrow flx;$ 
    else  $temp = a \rightarrow irmao \rightarrow flx;$ 
    if ( $temp > 0 \wedge a \rightarrow tip \neq i$ ) {
        if ( $min \equiv -1 \vee min > a \rightarrow tip \rightarrow dist$ )  $min = a \rightarrow tip \rightarrow dist;$ 
    }
}
 $i \rightarrow dist = min + 1;$ 
 $inserenalista(i);$ 

```

Este código é usado no bloco 9.

11. Execução de um push

Executar um push resume-se a atualizar valores.

```

⟨ Executa um push 11 ⟩ ≡
if ( $i \rightarrow exc < temp$ )  $temp = i \rightarrow exc;$ 
if ( $a \rightarrow flx \geq 0$ )  $a \rightarrow flx += temp;$ 
else  $a \rightarrow irmao \rightarrow flx -= temp;$ 
 $i \rightarrow exc -= temp;$ 
 $a \rightarrow tip \rightarrow exc += temp;$ 
if ( $i \rightarrow exc > 0$ )  $inserenalista(i);$ 
if ( $a \rightarrow tip \neq sorvedouro \wedge a \rightarrow tip \rightarrow exc > 0$ ) {
    if ( $a \rightarrow tip \rightarrow exc - temp \equiv 0$ )  $inserenalista(a \rightarrow tip);$ 
}

```

Este código é usado no bloco 9.

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

```

⟨ Variáveis da função maiorrotulo 12 ⟩ ≡
int  $iteracoes, temp, min;$ 
Vertex *  $i, *j;$ 
Arc *  $a;$ 

```

Este código é usado no bloco 5.

13. Lista de vértices

A estrutura de dados aqui utilizada para armazenar os vértices ativos foi baseada no livro *Network Flows* de R. K. Ahuja, T. L. Magnanti e J. B. Orlin.

⟨Lista de vértices 13⟩ ≡

```
Vertex **lista;

int level, size; void inicializalista(Graph *g) { int indice; lista = ( Vertex
    ** ) malloc ( (2 * g→n) * sizeof ( Vertex * ) );
    for (indice = 0; indice < 2 * g→n; indice++) lista[indice] = Λ;
    level = 0;
    size = 0;
    return; } void finalizalista()
{
    free(lista);
    return;
}
boolean listavazia ()
{
    if (size ≡ 0) return (TRUE);
    return (FALSE);
}
Vertex * retiradalista()
{
    Vertex * i;
    while (lista[level] ≡ Λ) level--;
    i = lista[level];
    lista[level] = i→prox;
    size--;
    return (i);
}
void inserenalista(Vertex * i)
{
    i→prox = lista[i→dist];
    lista[i→dist] = i;
    if (i→dist > level) level = i→dist;
    size++;
    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 >
    maiorrotulo(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-arc; a; a = a-next) {
        if (a-flx >= 0) {
            fprintf(saida, "fluxo de \s\ "a\s\ "i\ : \ld\n",
                a-inicio-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 é definido por um valor k tal que $d(i) \neq k$ para todo vértice i . Os vértices do separador possuem distância maior que k .

```

< Imprime separador de capacidade mínima 20 > ≡
for (k = 1; k < g-n; k++) {
    for (i = g-vertices; i < g-vertices + g-n; i++) {
        if (i-dist ≡ k) break;
    }
    if (i ≡ g-vertices + g-n) break;
}
fprintf(saida, "separador: \n");
for (min = 0, i = g-vertices; i < g-vertices + g-n; i++) {
    if (i-dist > k) {
        fprintf(saida, "\s\ \n", i-name);
        for (a = i-arc; a; a = a-next) {
            if (a-flx >= 0 & a-tip-dist < k) 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, k;
FILE *saida;

```

Este código é usado no bloco 14.

22. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

```
< Bibliotecas necessárias 23 >  
< Lista de vértices 13 >  
< Algoritmo dos vértices ativos de maior rótulo 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 dist u.I  
#define exc v.I  
#define prox w.V  
#define atual x.A  
#define cap len  
#define flx a.I  
#define irmao b.A  
#define inicio irmao→tip
```

Índice Remissivo

Arc: 12, 21.
arcs: 6, 7, 9, 10, 18, 19, 20.
argc: 14, 15.
argv: 14, 15, 16, 17.
atual: 6, 9, 24.
boolean: 13, 24.
cap: 6, 7, 9, 10, 18, 20, 24.
dist: 6, 9, 10, 13, 20, 24.
exc: 6, 8, 11, 24.
exit: 15, 16, 17, 18.
FALSE: 13, 24.
fclose: 20.
finalizalista: 5, 13.
flx: 6, 7, 9, 10, 11, 19, 20, 24.
fonte: 5, 6, 14, 17.
fopen: 16.
fprintf: 5, 15, 16, 17, 18, 19, 20.
free: 13.
gb_new_arc: 7.
Graph: 5, 13, 14.
indice: 13.
inicializalista: 5, 13.
inicio: 19, 24.
inserenalista: 8, 10, 11, 13.
irmao: 7, 9, 10, 11, 24.
iteracoes: 5, 12.
k: 21.
len: 24.
level: 13.
lista: 13.
listavazia: 5, 13.
main: 14.
maiorrotulo: 5, 14.
malloc: 13.
max: 19, 21.
min: 10, 12, 20, 21.
name: 17, 19, 20.
next: 6, 7, 9, 10, 18, 19, 20.
prox: 13, 24.
restore_graph: 16.
retiradalista: 5, 13.
saida: 16, 19, 20, 21.
size: 13.
sorvedouro: 5, 8, 11, 14, 17, 19.
stderr: 15, 16, 17, 18.
stdout: 5.
strcmp: 17.
temp: 9, 10, 11, 12.
tip: 6, 7, 9, 10, 11, 19, 20, 24.
TRUE: 13, 24.
Vertex: 5, 12, 13, 14, 21.
vertices: 6, 7, 8, 17, 18, 19, 20.

Lista de Refinamentos

- ⟨ Algoritmo dos vértices ativos de maior rótulo 5 ⟩ Usado no bloco 22.
- ⟨ Bibliotecas necessárias 23 ⟩ Usado no bloco 22.
- ⟨ Constrói arcos irmãos 7 ⟩ Usado no bloco 5.
- ⟨ Executa pré-processamento 6 ⟩ Usado no bloco 5.
- ⟨ Executa push ou relabel 9 ⟩ Usado no bloco 5.
- ⟨ Executa um push 11 ⟩ Usado no bloco 9.
- ⟨ Executa um relabel 10 ⟩ Usado no bloco 9.
- ⟨ 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.
- ⟨ Insere vértices iniciais 8 ⟩ Usado no bloco 5.
- ⟨ Lista de vértices 13 ⟩ Usado no bloco 22.
- ⟨ Variáveis da função *maiorrotulo* 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.