# Fluxos máximos
# Método do pré-fluxo
# Excess scaling

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. Algoritmo excess scaling . . . . . 3
11. Execução de um relabel . . . . . 5
12. Execução de um push . . . . . 5
14. Lista de vértices . . . . . 7
15. Função principal . . . . . 8
16. Consistência dos parâmetros . . . . . 8
20. Impressão do fluxo de intensidade máxima . . . . . 9
21. Impressão do separador de capacidade mínima . . . . . 10
23. Estrutura geral . . . . . 11
24. Bibliotecas . . . . . 11
25. Macros . . . . . 11

## 1. Introdução

Esta é uma implementação em CWEB-LaTeX do **algoritmo excess scaling**, uma versão do **método do pré-fluxo** para resolver o **problema do fluxo máximo**. A plataforma SGB é necessária para a execução do programa.

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

## 4. Referências

Sobre a plataforma SGB:
`http://www-cs-faculty.stanford.edu/~knuth/sgb.html`
Sobre a linguagem de *litterate programming* CWEB-LaTeX:
`http://www-cs-faculty.stanford.edu/~knuth/cweb.html`

## 5. Algoritmo excess scaling

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 algoritmo excess scaling mantém um limitante inferior $\Delta$ e seleciona vértices ativos cujo excesso é maior que o valor $\Delta/2$. O método pára quando a lista de vértices ativos está vazia e $\Delta < 1$. 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 de maneira implícita.

```
⟨ Algoritmo excess scaling 5 ⟩ ≡
  void excessscaling(Graph * g, Vertex * fonte, Vertex * sorvedouro)
  {
    ⟨ Variáveis da função excessscaling 13 ⟩
    ⟨ Obtém limitante inicial 6 ⟩
    ⟨ Executa pré-processamento 7 ⟩
    ⟨ Constrói arcos irmãos 8 ⟩
    iteracoes = 0;
    inicializalista(g);
    while (Delta ≥ 1) {
      ⟨ Insere vértices ativos 9 ⟩
      while (¬listavazia()) {
        i = retiradalista();
        ⟨ Executa push ou relabel 10 ⟩
        iteracoes++;
      }
      Delta = Delta/2;
    }
    finalizalista();
    fprintf(stdout, "número de iteraç~oes: %d\n", iteracoes);
    return;
  }
```

Este código é usado no bloco 23.

**6.** O valor inicial de $\Delta$ é a maior capacidade vezes o número de arcos.

```
⟨ Obtém limitante inicial 6 ⟩ ≡
  for (Delta = −1, i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (Delta < a→cap) Delta = a→cap;
    }
  }
  fprintf(stdout, "capacidade máxima: %d\n", Delta);
  Delta = Delta * g→m;
  Delta = (int) pow(2.0, floor(log((double) Delta)/log(2.0)));
```

Este código é usado no bloco 5.

**7.** 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 $d(fonte) = n$ e $d(i) = 0$ para todo vértice $i$ restante.

⟨ Executa pré-processamento 7 ⟩ ≡

```
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.

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

⟨ Constrói arcos irmãos 8 ⟩ ≡

```
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.

**9.** Os vértices ativos com excesso suficientemente grande são adicionados à lista. Deve-se adicionar uma condição extra para não adicionar o sorvedouro.

⟨ Insere vértices ativos 9 ⟩ ≡

```
for (i = g→vertices; i < g→vertices + g→n; i++) {
  if (i ≠ sorvedouro ∧ i→exc > Delta/2) inserenalista(i);
}
```

Este código é usado no bloco 5.

**10.** 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 10 ⟩ ≡
  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 11 ⟩
    i→atual = i→arcs;
  }
  else {
    ⟨ Executa um push 12 ⟩
    i→atual = a;
  }
```

Este código é usado no bloco 5.

### 11. 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 11 ⟩ ≡
  for (min = −1, a = i→arcs; a; a = a→next) {
    if (a→flx ≥ 0) temp = a→cap − a→flx;
    else temp = a→irmao→flx;
    if (temp > 0 ∧ a→tip ≠ i) {
      if (min ≡ −1 ∨ min > a→tip→dist) min = a→tip→dist;
    }
  }
  i→dist = min + 1;
  inserenalista(i);
```

Este código é usado no bloco 10.

### 12. Execução de um push

Executar um push resume-se a atualizar valores.

```
⟨ Executa um push 12 ⟩ ≡
  if (i→exc < temp) temp = i→exc;
  if (a→tip ≠ sorvedouro ∧ Delta − a→tip→exc < temp)
    temp = Delta − a→tip→exc;
  if (a→flx ≥ 0) a→flx += temp;
```

```
  else a→irmao→flx −= temp;
  i→exc −= temp;
  a→tip→exc += temp;
  if (i→exc > Delta/2) inserenalista(i);
  if (a→tip ≠ sorvedouro ∧ a→tip→exc > Delta/2) {
    if (a→tip→exc − temp ≤ Delta/2) inserenalista(a→tip);
  }
```

Este código é usado no bloco 10.

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

```
⟨ Variáveis da função excessscaling 13 ⟩ ≡
  int iteracoes, temp, min, Delta;
  Vertex * i, *j;
  Arc * a;
```

Este código é usado no bloco 5.

## 14. 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 14 ⟩ ≡

```
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;
    }
    booleanlistavazia( )
    {
      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 23.

### 15. 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 15 ⟩ ≡
  int main(int argc, char *argv[])
  {
    Graph * g;
    Vertex * fonte, *sorvedouro;
    ⟨ Variáveis secundárias da função principal 22 ⟩
    ⟨ Verifica consistência dos parâmetros 16 ⟩
    excessscaling(g, fonte, sorvedouro);
    ⟨ Imprime fluxo máximo 20 ⟩
    ⟨ Imprime separador de capacidade mínima 21 ⟩
    return (0);
  }
```

Este código é usado no bloco 23.

### 16. 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 16 ⟩ ≡
  if (argc ≠ 5) {
    fprintf(stderr, "%s␣<in>␣<out>␣\"source\"␣\"sink\"\n", argv[0]);
    exit(−1);
  }
  ⟨ Verifica validade dos arquivos 17 ⟩
  ⟨ Verifica existência dos vértices 18 ⟩
  ⟨ Verifica sinal das capacidades 19 ⟩
```

Este código é usado no bloco 15.

**17.** 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 17 ⟩ ≡
  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 16.

**18.** 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 18 ⟩ ≡
  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 16.

**19.** 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 19 ⟩ ≡
  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 16.

## 20. Impressão do fluxo de intensidade máxima

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

⟨ Imprime fluxo máximo 20 ⟩ ≡

```
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",
          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 15.

### 21. 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 21 ⟩ ≡

```
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→arcs; 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 15.

**22.** Podemos agora definir as variáveis secundárias da função principal.

⟨ Variáveis secundárias da função principal 22 ⟩ ≡

```
Vertex * i;
Arc * a;
int min, max, k;
FILE *saida;
```

Este código é usado no bloco 15.

### 23. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

⟨ Bibliotecas necessárias 24 ⟩
⟨ Lista de vértices 14 ⟩
⟨ Algoritmo excess scaling 5 ⟩
⟨ Função principal 15 ⟩

### 24. Bibliotecas

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

⟨ Bibliotecas necessárias 24 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gb_graph.h>
#include <gb_save.h>
```
Este código é usado no bloco 23.

### 25. 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*: 13, 22.
*arcs*: 6, 7, 8, 10, 11, 19, 20, 21.
*argc*: 15, 16.
*argv*: 15, 16, 17, 18.
*atual*: 7, 10, 25.
*boolean*: 14, 25.
*cap*: 6, 7, 8, 10, 11, 19, 21, 25.
*Delta*: 5, 6, 9, 12, 13.
*dist*: 7, 10, 11, 14, 21, 25.
*exc*: 7, 9, 12, 25.
*excessscaling*: 5, 15.
*exit*: 16, 17, 18, 19.
FALSE: 14, 25.
*fclose*: 21.
*finalizalista*: 5, 14.
*floor*: 6.
*flx*: 7, 8, 10, 11, 12, 20, 21, 25.
*fonte*: 5, 7, 15, 18.
*fopen*: 17.
*fprintf*: 5, 6, 16, 17, 18, 19, 20, 21.
*free*: 14.
*gb_new_arc*: 8.
*Graph*: 5, 14, 15.
*indice*: 14.
*inicializalista*: 5, 14.
*inicio*: 20, 25.
*inserenalista*: 9, 11, 12, 14.
*irmao*: 8, 10, 11, 12, 25.
*iteracoes*: 5, 13.
*k*: 22.
*len*: 25.
*level*: 14.
*lista*: 14.
*listavazia*: 5, 14.
*log*: 6.
*main*: 15.
*malloc*: 14.
*max*: 20, 22.
*min*: 11, 13, 21, 22.
*name*: 18, 20, 21.
*next*: 6, 7, 8, 10, 11, 19, 20, 21.
*pow*: 6.
*prox*: 14, 25.
*restore_graph*: 17.
*retiradalista*: 5, 14.
*saida*: 17, 20, 21, 22.
*size*: 14.
*sorvedouro*: 5, 9, 12, 15, 18, 20.
*stderr*: 16, 17, 18, 19.
*stdout*: 5, 6.
*strcmp*: 18.
*temp*: 10, 11, 12, 13.
*tip*: 7, 8, 10, 11, 12, 20, 21, 25.
TRUE: 14, 25.
*Vertex*: 5, 13, 14, 15, 22.
*vertices*: 6, 7, 8, 9, 18, 19, 20, 21.

**Lista de Refinamentos**

⟨ Algoritmo excess scaling 5 ⟩ Usado no bloco 23.
⟨ Bibliotecas necessárias 24 ⟩ Usado no bloco 23.
⟨ Constrói arcos irmãos 8 ⟩ Usado no bloco 5.
⟨ Executa pré-processamento 7 ⟩ Usado no bloco 5.
⟨ Executa push ou relabel 10 ⟩ Usado no bloco 5.
⟨ Executa um push 12 ⟩ Usado no bloco 10.
⟨ Executa um relabel 11 ⟩ Usado no bloco 10.
⟨ Função principal 15 ⟩ Usado no bloco 23.
⟨ Imprime fluxo máximo 20 ⟩ Usado no bloco 15.
⟨ Imprime separador de capacidade mínima 21 ⟩ Usado no bloco 15.
⟨ Insere vértices ativos 9 ⟩ Usado no bloco 5.
⟨ Lista de vértices 14 ⟩ Usado no bloco 23.
⟨ Obtém limitante inicial 6 ⟩ Usado no bloco 5.
⟨ Variáveis da função *excessscaling* 13 ⟩ Usado no bloco 5.
⟨ Variáveis secundárias da função principal 22 ⟩ Usado no bloco 15.
⟨ Verifica consistência dos parâmetros 16 ⟩ Usado no bloco 15.
⟨ Verifica existência dos vértices 18 ⟩ Usado no bloco 16.
⟨ Verifica sinal das capacidades 19 ⟩ Usado no bloco 16.
⟨ Verifica validade dos arquivos 17 ⟩ Usado no bloco 16.