

Problema do fluxo máximo
Método dos caminhos de aumento
Caminhos de aumento de comprimento mínimo

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 dos caminhos de aumento de comprimento mínimo . . .	3
9. Identificando caminhos de aumento de comprimento mínimo . .	4
12. Aumento do fluxo através do caminho	6
14. Fila de vértices	7
15. Função principal	9
16. Consistência dos parâmetros	9
20. Impressão do fluxo de intensidade máxima	10
21. Impressão do separador de capacidade mínima	11
23. Estrutura geral	12
24. Bibliotecas	12
25. Macros	12

1. Introdução

Esta é uma implementação em CWEB-L^AT_EX do **algoritmo dos caminhos de aumento de comprimento mínimo**, 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 aumentominimos.tex para gerar o arquivo LATEX de documentação.  
make aumentominimos.dvi para gerar o arquivo DVI de visualização.  
make aumentominimos.pdf para gerar o arquivo PDF de visualização.  
make aumentominimos.ps para gerar o arquivo PostScript de visualização.  
make aumentominimos.c para gerar o código-fonte C do programa.  
make aumentominimos para gerar o executável do programa.  
aumentominimos 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^AT_EX:

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

Sítio do projeto:

<http://www.ime.usp.br/~coelho/oticomb/>

5. Algoritmo dos caminhos de aumento de comprimento mínimo

O método dos caminhos de aumento recebe um grafo, representando uma rede capacitada, dois vértices s e t e devolve um st -fluxo de intensidade máxima e um st -separador de capacidade mínima nessa rede. Os vértices s e t são, respectivamente, a *fonte* e o *sorvedouro*. O st -separador mínimo é um certificado para a maximalidade do fluxo encontrado. O método começa com um st -fluxo x e busca, em cada iteração, aumentar sua intensidade através de um caminho de aumento. Quando a rede residual com relação a x não contém caminhos de aumento, o st -fluxo x é máximo e o método pára.

O algoritmo dos caminhos de aumento de comprimento mínimo é uma implementação do método dos caminhos de aumento que, em cada iteração, escolhe um caminho de aumento com o menor número de arcos possível na rede residual para efetuar o aumento do fluxo.

Antes da primeira iteração, o st -fluxo nulo é definido na rede e os arcos irmãos são gerados. Feito isso, o processo iterativo de busca por caminhos de aumento e aumento da intensidade do fluxo pode ser iniciado. A condição de parada do while, ($t \rightarrow apred \equiv \Lambda$), está explicada na descrição da busca por caminhos de aumento. Note que o fluxo corrente na rede é representado pelo campo $a \rightarrow flux$ de cada arco. Devido ao interesse na complexidade experimental do algoritmo, imprime-se o número total de iterações após a execução.

```
<Algoritmo dos caminhos de aumento de comprimento mínimo 5> ≡
void caminhos_aumento_minimos(Graph *g, Vertex *s, Vertex *t)
{
  <Variáveis da função caminhos_aumento_minimos 13>
  <Obtém fluxo inicial 6>
  <Constrói arcos irmãos 7>
  iteracoes = 0;
  <Encontra caminho de aumento minimo 9>
  while (t → apred ≠ Λ) {
    <Aumenta fluxo através do caminho de aumento 12>
    <Encontra caminho de aumento minimo 9>
    iteracoes++;
  }
  fprintf(stdout, "Número de iterações: %d\n", iteracoes);
  return;
}
```

Este código é usado no bloco 23.

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 → flux = 0;
  }
}
```

```

    }

```

Este código é usado no bloco 5.

7. Os arcos irmãos são construídos exatamente segundo sua definição. Note que o arco irmão gerado é inicialmente apontado por $j \rightarrow arcs$.

```

⟨ Constrói arcos irmãos 7 ⟩ ≡
  for (i = g → vertices; i < g → vertices + g → n; i++) {
    for (a = i → arcs; a; a = a → next) {
      if (arco_original(a)) {
        j = a → tip;
        gb_new_arc(j, i, a → cap);
        a → irmao = j → arcs;
        a → irmao → flux = -1;
        a → irmao → irmao = a;
      }
    }
  }

```

Este código é usado no bloco 5.

8. Nesta implementação, os arcos irmãos dos arcos da rede original são reconhecidos por terem fluxo negativo.

```

⟨ Função arco_original 8 ⟩ ≡
  int arco_original(Arc * a)
  {
    return (a → flux ≥ 0);
  }

```

Este código é usado no bloco 23.

9. Identificando caminhos de aumento de comprimento mínimo

Para que um caminho de aumento de comprimento mínimo seja identificado, uma busca em largura é efetuada na rede residual. Para cada vértice i , o campo $i \rightarrow apred$ é definido como seu arco predecessor em um caminho alternante de comprimento mínimo entre s e i na rede residual. Além disso, o campo $i \rightarrow res$ é definido como a capacidade residual do caminho alternante entre s e i determinado pelos arcos predecessores.

Ao final do processamento, o vértice sorvedouro t possui um arco predecessor definido unicamente se existe um caminho de aumento entre s e t na rede. Isso explica a condição de parada do **while** que aumenta o fluxo através de caminhos de aumento na função *caminhos_aumento_minimos*.

Assim, se o arco $t \rightarrow apred$ é definido durante a busca, então ele é o arco predecessor de t em um caminho de aumento de comprimento mínimo cuja capacidade residual é $t \rightarrow res$.

```

⟨ Encontra caminho de aumento mínimo 9 ⟩ ≡
for ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-n}$ ;  $i++$ ) {
     $i\text{-apred} = \Lambda$ ;
}
 $s\text{-res} = 0$ ;
inicializa_filha( $g$ );
insere_filha( $s$ );
while ( $\neg$ filha_vazia()) {
     $i = \text{remove\_filha}()$ ;
    ⟨ Examina vértice retirado da fila 10 ⟩
}

```

Este código é usado no bloco 5.

10. Ao examinar um vértice, visita-se seus vértices vizinhos na rede residual que ainda não foram atingidos pela busca em largura. Os vértices ainda não atingidos são identificados por não terem arco predecessor definido em *apred*.

Ao invés de manter uma estrutura de dados separada para a rede residual, mantemos esta implícita, através da capacidade residual de cada arco. Como os caminhos de aumento devem ser procurados na rede residual, então a busca considera apenas arcos com capacidade residual positiva.

Quando um vértice j é visitado, um arco predecessor l é atribuído, de forma a definir um caminho alternante de comprimento mínimo entre s e j . A capacidade residual desse caminho, armazenada em $j\text{-res}$, é o mínimo entre a capacidade residual do arco $j\text{-apred}$ e a capacidade residual do caminho alternante mínimo já definido entre s e i , vértice predecessor de j no caminho, armazenada em $i\text{-res}$.

```

⟨ Examina vértice retirado da fila 10 ⟩ ≡
for ( $a = i\text{-arcs}$ ;  $a$ ;  $a = a\text{-next}$ ) {
     $j = a\text{-tip}$ ;
    if ( $j\text{-apred} \equiv \Lambda \wedge j \neq s$ ) {
         $cap\_residual = \text{get\_cap\_residual}(a)$ ;
        if ( $cap\_residual > 0$ ) {
            if ( $cap\_residual < i\text{-res} \vee i \equiv s$ ) {
                 $j\text{-res} = cap\_residual$ ;
            }
        }
        else {
             $j\text{-res} = i\text{-res}$ ;
        }
         $j\text{-apred} = a$ ;
        insere_filha( $j$ );
    }
}
}

```

Este código é usado no bloco 9.

11. A capacidade residual de um arco da rede original corresponde à diferença entre sua capacidade e seu fluxo corrente. A capacidade residual de um arco irmão de um arco original corresponde ao fluxo corrente em seu arco irmão.

```

< Função get_capacidade_residual 11 > ≡
  long get_cap_residual(Arc *a)
  {
    if (¬arco_original(a)) {
      return a_irmao_flux;
    }
    else {
      return (a_cap - a_flux);
    }
  }

```

Este código é usado no bloco 23.

12. 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.

```

< Aumenta fluxo através do caminho de aumento 12 > ≡
  a = t_apred;
  while (a ≠ Λ) {
    if (¬arco_original(a)) a_irmao_flux = a_irmao_flux - t_res;
    else a_flux = a_flux + t_res;
    a = a_inicio_apred;
  }

```

Este código é usado no bloco 5.

13. Resta, agora, declarar as variáveis da função.

```

< Variáveis da função caminhos_aumento_minimos 13 > ≡
  int iteracoes, cap_residual;
  Vertex *i, *j;
  Arc *a;

```

Este código é usado no bloco 5.

14. Fila de vértices

A fila utilizada na busca em largura é implementada como uma fila circular através da utilização do próprio vetor de vértices de um grafo do SGB.

O campo *i*-*vertice* de determinado vértice *i* do grafo *g*, utilizado para a construção da fila, representa o vértice que ocupa a mesma posição de *i* no vetor *g*-*vertices*. O primeiro vértice na fila é dado por *ini*-*vertices* e o último, por *fim*-*vertices*. A condição (*ini* \equiv *fim*) indica que a fila está vazia. A implementação supõe que no máximo *g*-*n* vértices ocuparão a fila ao mesmo tempo, o que sempre ocorre durante uma busca na rede residual.

```
<Fila circular de vértices 14>  $\equiv$ 
  Vertex * ini, *fim, *zero;
  Vertex * max;
void inicializa_fila(Graph * g)
{
  ini = g-vertices;
  fim = g-vertices;
  zero = g-vertices;
  max = g-vertices + g-n;
  return;
}
void insere_fila(Vertex * i)
{
  fim-vertice = i;
  fim++;
  if (fim > max) {
    fim = zero;
  }
  if (fim  $\equiv$  ini) {
    fprintf(stderr, "ERR0: Fila excedeu sua capacidade.\n");
    exit(-1);
  }
}
boolean fila_vazia()
{
  if (ini  $\equiv$  fim) return TRUE;
  return FALSE;
}
Vertex * remove_fila()
{
  Vertex * i;
  if (!fila_vazia()) {
    i = ini-vertice;
    ini++;
    if (ini > max) {
```

```
        ini = zero;
    }
    return i;
}
return  $\Lambda$ ;
}
Vertex * primeiro_fila()
{
    if ( $\neg$ fila_vazia()) {
        return ini-vertice;
    }
    return  $\Lambda$ ;
}
```

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 >
    caminhos_aumento_minimos(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* de cada arco corresponda à sua capacidade. Também é necessário que os nomes de vértices referenciem vértices que de fato existam 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, "Uso: %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, "ERRO: Problemas com arquivo de entrada.\n");
    exit(-2);
  }
  if ((saida = fopen(argv[2], "w")) ≡ Λ) {
    fprintf(stderr, "ERRO: Arquivo de saída inválido.\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, "ERRO: 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, "ERRO: Capacidade negativa encontrada.\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 máximo encontrado e sua 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 (arco_original(a)) {
        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);
  fprintf(stdout, "Intensidade do fluxo máximo: \d\n", max);

```

Este código é usado no bloco 15.

21. Impressão do separador de capacidade mínima

O separador de capacidade mínima contém os vértices da rede original acessíveis a partir da fonte através de caminhos alternantes, ou seja, os vértices examinados durante a última busca por um caminho de aumento. Tais vértices são identificados por possuírem um arco predecessor em *apred* definido.

```

<Imprime separador de capacidade mínima 21> ≡
  fprintf(saida, "\nSeparador: \n");
  for (min = 0, i = g→vertices; i < g→vertices + g→n; i++) {
    if (i→apred ≠ Λ ∨ i ≡ fonte) {
      fprintf(saida, "\s\ \n", i→name);
      for (a = i→arcs; a; a = a→next) {
        if (arco_original(a) ∧ (a→tip→apred ≡ Λ ∧ a→tip ≠ fonte))
          min += a→cap;
      }
    }
  }
  fprintf(saida, "Capacidade: \d\n", min);
  fprintf(stdout, "Capacidade do separador mínimo: \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;
  FILE *saida;

```

Este código é usado no bloco 15.

23. Estrutura geral

Para concluir o programa, basta definir a estrutura geral.

```
< Bibliotecas necessárias 24 >  
< Fila circular de vértices 14 >  
< Função arco_original 8 >  
< Função get_capacidade_residual 11 >  
< Algoritmo dos caminhos de aumento de comprimento mínimo 5 >  
< Função principal 15 >
```

24. Bibliotecas

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

```
< Bibliotecas necessárias 24 > ≡  
#include <stdio.h>  
#include <stdlib.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 apred u.A  
#define res v.I  
#define vertice w.V  
#define cap len  
#define flx a.I  
#define irmao b.A  
#define inicio irmao-tip
```

Índice Remissivo

apred: 5, 9, 10, 12, 21, 25.
Arc: 8, 11, 13, 22.
arco_original: 7, 8, 11, 12, 20, 21.
arcs: 6, 7, 10, 19, 20, 21.
argc: 15, 16.
argv: 15, 16, 17, 18.
boolean: 14, 25.
caminhos_aumento_minimos: 5,
9, 15.
cap: 7, 11, 19, 21, 25.
cap_residual: 10, 13.
exit: 14, 16, 17, 18, 19.
FALSE: 14, 25.
fclose: 21.
fila_vazia: 9, 14.
fim: 14.
flx: 5, 6, 7, 8, 11, 12, 20, 25.
fonte: 15, 18, 21.
fopen: 17.
fprintf: 5, 14, 16, 17, 18, 19,
20, 21.
gb_new_arc: 7.
get_cap_residual: 10, 11.
Graph: 5, 14, 15.
ini: 14.
inicializa_fila: 9, 14.
inicio: 12, 25.
insere_fila: 9, 10, 14.
irmao: 7, 11, 12, 25.
iteracoes: 5, 13.
len: 16, 25.
main: 15.
max: 14, 20, 22.
min: 21, 22.
name: 18, 20, 21.
next: 6, 7, 10, 19, 20, 21.
primeiro_fila: 14.
remove_fila: 9, 14.
res: 9, 10, 12, 25.
restore_graph: 17.
saida: 17, 20, 21, 22.
sorvedouro: 15, 18, 20.
st: 5.
stderr: 14, 16, 17, 18, 19.
stdout: 5, 20, 21.
strcmp: 18.
tip: 7, 10, 20, 21, 25.
TRUE: 14, 25.
Vertex: 5, 13, 14, 15, 22.
vertice: 14, 25.
vertices: 6, 7, 9, 14, 18, 19, 20, 21.
zero: 14.

Lista de Refinamentos

- ⟨ Algoritmo dos caminhos de aumento de comprimento mínimo 5 ⟩ Usado no bloco 23.
- ⟨ Aumenta fluxo através do caminho de aumento 12 ⟩ Usado no bloco 5.
- ⟨ Bibliotecas necessárias 24 ⟩ Usado no bloco 23.
- ⟨ Constrói arcos irmãos 7 ⟩ Usado no bloco 5.
- ⟨ Encontra caminho de aumento mínimo 9 ⟩ Usado no bloco 5.
- ⟨ Examina vértice retirado da fila 10 ⟩ Usado no bloco 9.
- ⟨ Fila circular de vértices 14 ⟩ Usado no bloco 23.
- ⟨ Função principal 15 ⟩ Usado no bloco 23.
- ⟨ Função *arco_original* 8 ⟩ Usado no bloco 23.
- ⟨ Função *get_capacidade_residual* 11 ⟩ Usado no bloco 23.
- ⟨ Imprime fluxo máximo 20 ⟩ Usado no bloco 15.
- ⟨ Imprime separador de capacidade mínima 21 ⟩ Usado no bloco 15.
- ⟨ Obtém fluxo inicial 6 ⟩ Usado no bloco 5.
- ⟨ Variáveis da função *caminhos_aumento_minimos* 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.