

Fluxos máximos

Método dos caminhos de aumento

Capacity 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 capacity scaling	3
9. Caminhos de aumento	4
11. Aumento do fluxo através do caminho	5
13. Fila para busca em largura	7
14. Função principal	8
15. Consistência dos parâmetros	8
19. Impressão do fluxo de intensidade máxima	9
20. Impressão do separador de capacidade mínima	10
22. Estrutura geral	11
23. Bibliotecas	11
24. Macros	11

1. Introdução

Esta é uma implementação em CWEB- $\text{\LaTeX}$ do **algoritmo capacity scaling**, uma versão do **método dos caminhos de aumento** para resolver o **problema do fluxo máximo**. A plataforma SGB é necessária para 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 capacityscaling.tex para gerar o arquivo $\text{\LaTeX}$ de documentação.
make capacityscaling.dvi para gerar o arquivo DVI de visualização.
make capacityscaling.pdf para gerar o arquivo PDF de visualização.
make capacityscaling.ps para gerar o arquivo PostScript de visualização.
make capacityscaling.c para gerar o código-fonte C do programa.
make capacityscaling para gerar o executável do programa.
capacityscaling 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- $\text{\LaTeX}$:

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

5. Algoritmo capacity scaling

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. O algoritmo capacity scaling mantém um limitante inferior Δ para a capacidade residual dos caminhos de aumento desejados. 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 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.

⟨ Algoritmo capacity scaling 5 ⟩ ≡

```
void capacityscaling(Graph * g, Vertex * fonte, Vertex * sorvedouro)
{
    ⟨ Variáveis da função capacityscaling 12 ⟩
    ⟨ Obtém limitante inicial 6 ⟩
    ⟨ Obtém fluxo inicial 7 ⟩
    ⟨ Constrói arcos irmãos 8 ⟩
    iteracoes = 0;
    while (Delta ≥ 1) {
        do {
            ⟨ Encontra caminho de aumento 9 ⟩
            ⟨ Aumenta fluxo através do caminho de aumento 11 ⟩
            iteracoes++;
        } while (sorvedouro→arcopred ≠  $\Lambda$ );
        Delta = Delta / 2;
    }
    fprintf(stdout, "número de iterações: %d\n", iteracoes - 1);
    return;
}
```

Este código é usado no bloco 22.

6. O valor inicial de Δ corresponde à maior capacidade.

⟨ 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 = (int) pow(2.0, floor(log((double) Delta) / log(2.0)));
```

Este código é usado no bloco 5.

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

```

< Obtém fluxo inicial 7 > ≡
  for ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-}n$ ;  $i++$ ) {
    for ( $a = i\text{-arcs}$ ;  $a; a = a\text{-next}$ ) {
       $a\text{-flx} = 0$ ;
    }
  }

```

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 ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-}n$ ;  $i++$ ) {
    for ( $a = i\text{-arcs}$ ;  $a; a = a\text{-next}$ ) {
      if ( $a\text{-flx} \geq 0$ ) {
         $j = a\text{-tip}$ ;
         $gb\_new\_arc(j, i, a\text{-cap})$ ;
         $a\text{-irmao} = j\text{-arcs}$ ;
         $a\text{-irmao}\text{-flx} = -1$ ;
         $a\text{-irmao}\text{-irmao} = a$ ;
      }
    }
  }

```

Este código é usado no bloco 5.

9. Caminhos de aumento

Os caminhos de aumento são obtidos através de uma busca em largura simples a partir do vértice fonte, implementada através de uma fila tradicional. Inicialmente a fila só tem um único elemento: o próprio vértice fonte.

```

< Encontra caminho de aumento 9 > ≡
  for ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-}n$ ;  $i++$ ) {
     $i\text{-arccopred} = \Lambda$ ;
     $i\text{-estado} = \text{NAOVISTO}$ ;
  }
   $inicializa\text{fila}()$ ;
   $inserena\text{fila}(\text{fonte})$ ;
   $\text{fonte}\text{-estado} = \text{VISITADO}$ ;
  while ( $\neg \text{filavazia}()$ ) {
     $i = \text{retirada\text{fila}}()$ ;
    < Examina vértice retirado da fila 10 >
  }

```

Este código é usado no bloco 5.

10. Ao examinar um vértice, visita-se seus vizinhos na rede residual restrita e insere-os na fila. A capacidade residual do caminho alternante até o vértice é mantida no atributo *res*. 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 maior ou igual a Δ . Utiliza-se uma variável temporária para armazenar a capacidade residual dos arcos.

```

< Examina vértice retirado da fila 10 >  $\equiv$ 
for ( $a = i \rightarrow arcs$ ;  $a$ ;  $a = a \rightarrow next$ ) {
     $j = a \rightarrow tip$ ;
    if ( $j \rightarrow estado \equiv \text{NAOVISTO}$ ) {
        if ( $a \rightarrow flx \geq 0$ )  $temp = a \rightarrow cap - a \rightarrow flx$ ;
        else  $temp = a \rightarrow irmao \rightarrow flx$ ;
        if ( $temp \geq \Delta$ ) {
            if ( $i \neq fonte \wedge temp > i \rightarrow res$ )  $j \rightarrow res = i \rightarrow res$ ;
            else  $j \rightarrow res = temp$ ;
             $insereNaFila(j)$ ;
             $j \rightarrow estado = \text{VISITADO}$ ;
             $j \rightarrow arcopred = a$ ;
        }
    }
}

```

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.

```

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

```

Este código é usado no bloco 5.

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

```

< Variáveis da função capacityscaling 12 >  $\equiv$ 
int  $iteracoes, temp, \Delta$ ;

```

Vertex * *i*, **j*;

Arc * *a*;

Este código é usado no bloco 5.

13. Fila para busca em largura

A implementação da fila utilizada na busca em largura é feita através do uso de um apontador em cada vértice e dois apontadores auxiliares extras.

⟨Fila para busca em largura 13⟩ ≡

```
Vertex * head, *tail;

void inicializafila()
{
    head =  $\Lambda$ ;
    tail =  $\Lambda$ ;
}

boolean filavazia()
{
    if (head  $\equiv$   $\Lambda$ ) return (TRUE);
    return (FALSE);
}

Vertex * retiradafila()
{
    Vertex * i = head;
    head = head→prox;
    if (head  $\equiv$   $\Lambda$ ) tail =  $\Lambda$ ;
    return (i);
}

void inserenafila(Vertex * i)
{
    if (head  $\equiv$   $\Lambda$ ) {
        head = i;
        tail = i;
    }
    else {
        tail→prox = i;
        tail = i;
    }
    i→prox =  $\Lambda$ ;
    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 >  
    capacityscaling(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 >  $\equiv$ 
  if (( $g = \text{restore\_graph}(argv[1]) \equiv \Lambda$ ) {
    fprintf(stderr, "entrada_inválida\n");
    exit(-2);
  }
  if (( $saida = \text{fopen}(argv[2], "w") \equiv \Lambda$ ) {
    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 >  $\equiv$ 
  fonte =  $\Lambda$ ;
  sorvedouro =  $\Lambda$ ;
  for ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-}n$ ;  $i++$ ) {
    if ( $\neg \text{strcmp}(i\text{-name}, argv[3])$ ) fonte =  $i$ ;
    if ( $\neg \text{strcmp}(i\text{-name}, argv[4])$ ) sorvedouro =  $i$ ;
  }
  if ( $\text{fonte} \equiv \Lambda \vee \text{sorvedouro} \equiv \Lambda$ ) {
    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 >  $\equiv$ 
  for ( $i = g\text{-vertices}$ ;  $i < g\text{-vertices} + g\text{-}n$ ;  $i++$ ) {
    for ( $a = i\text{-arcs}$ ;  $a$ ;  $a = a\text{-next}$ ) {
      if ( $a\text{-cap} < 0$ ) {
        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.

22. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

⟨ Bibliotecas necessárias 23 ⟩
⟨ Fila para busca em largura 13 ⟩
⟨ Algoritmo capacity scaling 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 <math.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 arcopred u.A
#define estado w.I
#define res x.I
#define prox y.V
#define cap len
#define flx a.I
#define irmao b.A
#define inicio irmao→tip
```

Índice Remissivo

Arc: 12, 21.
arcopred: 5, 9, 10, 11, 24.
arcs: 6, 7, 8, 10, 18, 19, 20.
argc: 14, 15.
argv: 14, 15, 16, 17.
boolean: 13, 24.
cap: 6, 8, 10, 18, 20, 24.
capacityscaling: 5, 14.
Delta: 5, 6, 10, 12.
estado: 9, 10, 20, 24.
exit: 15, 16, 17, 18.
FALSE: 13, 24.
fclose: 20.
filavazia: 9, 13.
floor: 6.
flx: 7, 8, 10, 11, 19, 20, 24.
fonte: 5, 9, 10, 11, 14, 17.
fopen: 16.
fprintf: 5, 6, 15, 16, 17, 18, 19, 20.
gb_new_arc: 8.
Graph: 5, 14.
head: 13.
inicializafila: 9, 13.
inicio: 11, 24.
inserenafila: 9, 10, 13.
irmao: 8, 10, 11, 24.
iteracoes: 5, 12.
len: 24.
log: 6.
main: 14.
max: 19, 21.
min: 20, 21.
name: 17, 19, 20.
NAOVISTO: 9, 10, 20, 24.
next: 6, 7, 8, 10, 18, 19, 20.
pow: 6.
prox: 13, 24.
res: 10, 11, 24.
restore_graph: 16.
retiradafila: 9, 13.
saida: 16, 19, 20, 21.
sorvedouro: 5, 11, 14, 17, 19.
stderr: 15, 16, 17, 18.
stdout: 5, 6.
strcmp: 17.
tail: 13.
temp: 10, 12.
tip: 8, 10, 19, 20, 24.
TRUE: 13, 24.
Vertex: 5, 12, 13, 14, 21.
vertices: 6, 7, 8, 9, 17, 18, 19, 20.
VISITADO: 9, 10, 24.

Lista de Refinamentos

- ⟨ Algoritmo capacity scaling 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 8 ⟩ Usado no bloco 5.
- ⟨ Encontra caminho de aumento 9 ⟩ Usado no bloco 5.
- ⟨ Examina vértice retirado da fila 10 ⟩ Usado no bloco 9.
- ⟨ Fila para busca em largura 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 7 ⟩ Usado no bloco 5.
- ⟨ Obtém limitante inicial 6 ⟩ Usado no bloco 5.
- ⟨ Variáveis da função *capacityscaling* 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.