

Problema do fluxo de custo mínimo

Método do cancelamento de circuitos

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 cancelamento de circuitos	3
7. Obtenção do fluxo viável inicial	4
14. Busca por circuitos negativos	8
17. Redução do custo do fluxo	10
21. Fila de vértices	12
22. Função principal	14
23. Consistência dos parâmetros	14
26. Impressão do fluxo viável de custo mínimo	15
27. Impressão dos custos reduzidos ótimos	16
29. Estrutura geral	17
30. Bibliotecas	17
31. Macros	17
32. Estruturas	18

1. Introdução

Esta é uma implementação em CWEB-L^AT_EX do **método do cancelamento de circuitos** para resolver o **problema do fluxo de custo mínimo**. A plataforma SGB é necessária.

2. Descrição

Este programa recebe o nome de um arquivo de entrada que contém um grafo no formato SGB e o nome de um arquivo de saída e imprime no arquivo de saída um fluxo viável de custo mínimo da rede representada pelo grafo. Assume-se a priori que as capacidades dos arcos estão representadas no campo *len*, os custos dos arcos estão no campo *a.I* e as demandas dos vértices no campo *u.I*.

3. Compilação e execução

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

5. Método do cancelamento de circuitos

O método do cancelamento de circuitos recebe um grafo, representando uma rede capacitada, com custos e demanda, e devolve um fluxo viável de custo mínimo nessa rede. Ele é um algoritmo iterativo e mantém um fluxo viável durante toda a sua execução. A idéia é, em cada iteração, transformar o fluxo corrente através de circuitos negativos na rede residual, de forma a diminuir seu custo, mas sem comprometer sua viabilidade. Quando a rede residual não contém circuitos negativos, o fluxo tem custo mínimo e o método pára.

Antes da primeira iteração, um fluxo viável deve ser estabelecido. Como a rede original não contém arcos irmãos, tais arcos são criados durante esse processo. Devido ao interesse na complexidade experimental do algoritmo, imprime-se o número total de iterações após a execução.

```
⟨Método do cancelamento de circuitos 5⟩ ≡
void cyclecanceling(Graph *g)
{
    ⟨Variáveis da função cyclecanceling 20⟩
    ⟨Associa estrutura de campos utilitários aos arcos 6⟩
    ⟨Busca fluxo viável inicial 7⟩
    iteracoes = 0;
    if (viavel) {
        do {
            ⟨Procura circuito negativo 14⟩
            if (i_ciclo ≠ Λ) {
                ⟨Reduz custo do fluxo através do circuito negativo 17⟩
                iteracoes++;
            }
        } while (i_ciclo ≠ Λ);
    }
    else {
        fprintf(stdout, "ERRO: Instância não é viável!\n");
        exit(-1);
    }
    fprintf(stdout, "Número de iterações: %d\n", iteracoes);
    return;
}
```

Este código é usado no bloco 29.

6. Como o SGB disponibiliza apenas 2 campos utilitários para cada arco, uma estrutura auxiliar se faz necessária para o armazenamento de todas as informações que precisamos sobre os arcos.

```
⟨Associa estrutura de campos utilitários aos arcos 6⟩ ≡
for (i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
        temp = malloc(sizeof(struct str_arc));
    }
}
```

```

    if (temp == Λ) {
        fprintf(stderr, "ERRO: Problemas com alocação de memória.\n");
        exit(-1);
    }
    a_est = (int) temp;
}
}

```

Este código é usado no bloco 5.

7. Obtenção do fluxo viável inicial

Para encontrar um fluxo viável inicial, resolve-se o seguinte problema de fluxo máximo auxiliar: dois vértices são adicionados ao grafo original, um fazendo papel de fonte e o outro, de sorvedouro; a fonte é ligada a cada vértice com demanda negativa e cada vértice com demanda positiva é ligado ao sorvedouro. As capacidades dos novos arcos correspondem às demandas de seus extremos que estão no grafo original. Os vértices e arcos adicionados à rede são chamados *artificiais*.

```

<Busca fluxo viável inicial 7> ≡
  <Cria vértices e arcos artificiais 8>
  <Resolve problema do fluxo máximo auxiliar 9>
  <Verifica se existe solução viável 10>
  <Remove vértices e arcos artificiais 11>

```

Este código é usado no bloco 5.

8. Conforme a documentação do SGB, todo grafo criado contém pelo menos 4 vértices extras. Dois desses vértices serão utilizados como fonte e sorvedouro no problema auxiliar.

```

<Cria vértices e arcos artificiais 8> ≡
  s = g_vertices + g_n;
  t = g_vertices + g_n + 1;
  for (i = g_vertices; i < g_vertices + g_n; i++) {
    if (i_demanda < 0) {
        gb_new_arc(s, i, 0);
        temp = malloc(sizeof(struct str_arc));
        if (temp == Λ) {
            fprintf(stderr, "ERRO: Problemas com alocação de memória.\n");
            exit(-1);
        }
        (s_arcs)_est = (int) temp;
        (s_arcs)_cap = -(i_demanda);
    }
    if (i_demanda > 0) {
        gb_new_arc(i, t, 0);
    }
}

```

```

temp = malloc(sizeof(struct str_arc));
if (temp == Λ) {
    fprintf(stderr, "ERRO: Problemas com alocação de memória.\n");
    exit(-1);
}
(i->arcs)->est = (int) temp;
(i->arcs)->cap = i->demanda;
}
}
g->n += 2;

```

Este código é usado no bloco 7.

9. O algoritmo para resolver o problema de fluxo máximo é o **algoritmo dos caminhos de aumento de comprimento mínimo**, implementação do **método dos caminhos de aumento**. Esse algoritmo é detalhado em seu próprio documento, não havendo necessidade de comentá-lo. Os arcos irmãos são gerados durante o processamento abaixo. Além disso, se existe um fluxo máximo para o problema auxiliar, então, após o processamento, tal fluxo é representado pelo campo $flx(a)$ de cada arco a da rede.

```

< Resolve problema do fluxo máximo auxiliar 9 > ≡
/* Define fluxo nulo inicial na rede. */
for (i = g->vertices; i < g->vertices + g->n; i++) {
    for (a = i->arcs; a; a = a->next) {
        flx(a) = 0;
    }
} /* Gera arcos irmaos para os arcos da rede g. */
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, 0);
            temp = malloc(sizeof(struct str_arc));
            if (temp == Λ) {
                fprintf(stderr,
                    "ERRO: Problemas com alocação de memória.\n");
                exit(-1);
            }
            (j->arcs)->est = (int) temp;
            irmao(a) = j->arcs;
            irmao(j->arcs) = a;
            flx(j->arcs) = -1;
            (j->arcs)->custo = -((irmao(j->arcs))->custo);
        }
    }
}
}

```

```

do { /* Obtém caminho de aumento mínimo. */
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    i→apred = Λ;
  }
  s→res = -1;
  init_queue(g, s);
  while (¬empty_queue()) {
    i = remove_first();
    for (a = i→arcs; a; a = a→next) {
      j = a→tip;
      if ((j→apred ≡ Λ) ∧ (j ≠ s)) {
        cap_residual = get_cap_residual(a);
        if (cap_residual > 0) {
          j→apred = a;
          if (cap_residual < i→res ∨ i→res ≡ -1) j→res = cap_residual;
          else j→res = i→res;
          enqueue(j);
        }
      }
    }
  }
}
if (t→apred ≠ Λ) {
  /* Aumenta fluxo através do caminho de aumento. */
  a = t→apred;
  while (a ≠ Λ) {
    if (¬arco_original(a)) flx(irmao(a)) = flx(irmao(a)) - t→res;
    else flx(a) = flx(a) + t→res;
    a = inicio(a)→apred;
  }
}
} while (t→apred ≠ Λ);

```

Este código é usado no bloco 7.

10. Se o fluxo máximo encontrado para o problema auxiliar satura todos os arcos artificiais, então o problema é viável.

⟨ Verifica se existe solução viável 10 ⟩ ≡

```

viavel = TRUE;
for (a = s→arcs; a; a = a→next) {
  if (flx(a) ≠ a→cap) {
    viavel = FALSE;
    break;
  }
}
if (viavel) {
  for (i = g→vertices; i < g→vertices + g→n - 2; i++) {

```

```

    if (i-demanda > 0) {
        for (a = i-arcs; a; a = a-next) {
            if (a-tip ≡ t ∧ flux(a) ≠ a-cap) {
                viavel = FALSE;
            }
        }
    }
}

```

Este código é usado no bloco 7.

11. Após a resolução do problema auxiliar, os arcos artificiais precisam ser removidos da rede. Para tanto, eles são retirados das listas de adjacências.

```

⟨Remove vértices e arcos artificiais 11⟩ ≡
    g-n -= 2;
    if (viavel) {
        for (i = g-vertices; i < g-vertices + g-n; i++) {
            if (i-demanda ≠ 0) {
                aux = Λ;
                for (a = i-arcs; a; a = a-next) {
                    if (a-tip ≡ s ∨ a-tip ≡ t) {
                        if (aux ≡ Λ) i-arcs = a-next;
                        else aux-next = a-next;
                    }
                }
                else {
                    aux = a;
                }
            }
        }
    }
}

```

Este código é usado no bloco 7.

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

```

⟨Função arco_original 12⟩ ≡
    int arco_original(Arc * a)
    {
        return (flux(a) ≥ 0);
    }

```

Este código é usado no bloco 29.

13. 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 13 > ≡
  long get_cap_residual(Arc *a)
  {
    if (¬arco_original(a)) {
      return flx(irmao(a));
    }
    else {
      return (cap - flx(a));
    }
  }

```

Este código é usado no bloco 29.

14. Busca por circuitos negativos

A busca por circuitos de custo negativo na rede residual é realizada da seguinte maneira: um vértice da rede é escolhido como *origem* e verifica-se se existe um circuito negativo acessível a partir desse vértice. Caso algum circuito negativo, digamos C , seja encontrado, então i_ciclo é definido como um vértice desse circuito e, para cada vértice i de C , $i_ciclo \rightarrow apred$ é seu arco predecessor em C . Se não existir circuito negativo acessível a partir de *origem* na rede residual, então escolhe-se um vértice j não acessível a partir de *origem*, caso exista, e o procedimento é repetido, tomando-se, agora, j como *origem*.

```

< Procura circuito negativo 14 > ≡
  for ( $i = g \rightarrow vertices$ ;  $i < g \rightarrow vertices + g \rightarrow n$ ;  $i++$ ) {
     $i \rightarrow dist = -1$ ;
     $i \rightarrow apred = \Lambda$ ;
  }
  origem =  $g \rightarrow vertices$ ;
  do {
    < Busca circuito negativo acessível a partir de origem 15 >
    if ( $i\_ciclo \equiv \Lambda$ ) {
      < Redefine origem 16 >
    }
  } while ( $i\_ciclo \equiv \Lambda \wedge origem \neq \Lambda$ );

```

Este código é usado no bloco 5.

15. Para verificar a existência de um circuito negativo acessível a partir de *origem* na rede residual, uma implementação do **algoritmo de Bellman, Ford e Moore** é utilizada. Tal algoritmo encontra caminhos de custo mínimo entre *origem* e todos os demais vértices acessíveis a partir do mesmo ou devolve em i_ciclo um vértice pertencente a um circuito negativo acessível a partir de

origem. Os caminhos encontrados vão sendo definidos pelo campo *i→apred* de cada vértice, que representa seu arco predecessor. Se ao final do processamento a condição (*i_ciclo* $\equiv \Lambda$) for verdadeira, então não existe circuito negativo acessível a partir de *origem*. Caso contrário, percorrendo-se os arcos predecessores a partir de *i_ciclo*, tal circuito pode ser determinado. Observe que, após o processamento, os vértices não acessíveis a partir de origem são caracterizados por não terem arco predecessor definido.

Uma descrição mais detalhada do algoritmo, bem como de suas possíveis implementações, pode ser encontrada nos livros *Introduction to Algorithms*, de T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Network Flows*, de R. K. Ahuja, T. L. Magnanti e J. B. Orlin e *Data Structures and Network Algorithms*, de R. E. Tarjan.

```

⟨Busca circuito negativo acessível a partir de origem 15⟩  $\equiv$ 
    passo = 0;
    ultimo = origem;
    origem→dist = 0; origem→apred = ( Arc * ) origem;
    init_queue(g, origem);
    while ((passo  $\neq$  g→n)  $\wedge$  ( $\neg$ empty_queue())) {
        i = remove_first();
        for (a = i→arcs; a; a = a→next) {
            if (get_cap_residual(a) > 0) {
                d = i→dist + a→custo;
                j = a→tip;
                if (d < j→dist  $\vee$  (j→apred  $\equiv \Lambda$   $\wedge$  j  $\neq$  origem)) {
                    j→dist = d;
                    j→apred = a;
                    if ( $\neg$ is_enqueued(j)) {
                        enqueue(j);
                    }
                }
            }
        }
        if (i  $\equiv$  ultimo) {
            ultimo = last_enqueued();
            passo ++;
        }
    }
    i_ciclo =  $\Lambda$ ;
    if ( $\neg$ empty_queue()) {
        for (i = g→vertices; i < g→vertices + g→n; i++) i→dist = 0;
        ultimo→dist = 1;
        i = inicio(ultimo→apred);
        while (i→dist  $\neq$  1) {
            i→dist = 1;
            i = inicio(i→apred);
        }
    }

```

```

    }
    i_ciclo = i;
  }

```

Este código é usado no bloco 14.

16. Se um circuito negativo não foi encontrado a partir de *origem* na rede residual, verificamos se existe algum vértice não acessível a partir de *origem* nessa rede e que ainda não tenha sido processado. Caso exista um vértice *j* nessas condições, *j* torna-se a nova origem e procuramos um circuito negativo na rede residual acessível a partir dele.

```

⟨ Redefine origem 16 ⟩ ≡
  origem = Λ;
  for (j = g-vertices; j < g-vertices + g-n; j++) {
    if (j-aped ≡ Λ) {
      origem = j;
      break;
    }
  }

```

Este código é usado no bloco 14.

17. Redução do custo do fluxo

Para reduzir o custo do fluxo, percorremos o circuito de custo negativo e atualizamos o fluxo em seus arcos ou arcos irmãos de seus arcos em função de sua capacidade residual.

```

⟨ Reduz custo do fluxo através do circuito negativo 17 ⟩ ≡
  ⟨ Encontra capacidade residual do circuito 18 ⟩
  ⟨ Atualiza fluxo através do circuito 19 ⟩

```

Este código é usado no bloco 5.

18. A capacidade residual do circuito negativo é a menor capacidade residual dentre as capacidades residuais de seus arcos.

```

⟨ Encontra capacidade residual do circuito 18 ⟩ ≡
  init_queue(g, i_ciclo);
  capres_ciclo = get_cap_residual(i_ciclo-aped);
  for (i = inicio(i_ciclo-aped); i ≠ i_ciclo; i = inicio(i-aped)) {
    a = i-aped;
    if (get_cap_residual(a) < capres_ciclo) {
      capres_ciclo = get_cap_residual(a);
    }
    enqueue(i);
  }

```

Este código é usado no bloco 17.

19. O fluxo só é alterado nos arcos da rede original pertencentes ao circuito ou nos arcos da rede original cujos irmãos pertencem ao circuito. Se o arco original está no circuito, adicionamos ao seu fluxo a capacidade residual do circuito. Caso contrário, se o seu irmão está no circuito, subtraímos de seu fluxo a capacidade residual do circuito.

```

<Atualiza fluxo através do circuito 19> ≡
  while (¬empty_queue()) {
    i = remove_first();
    a = i→apred;
    if (arco_original(a)) {
      flux(a) += capres_ciclo;
    }
    else {
      flux(irmao(a)) -= capres_ciclo;
    }
  }

```

Este código é usado no bloco 17.

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

```

<Variáveis da função cyclecanceling 20> ≡
  int iteracoes, d, passo;
  long cap_residual, capres_ciclo;
  boolean viavel;
  Vertex *i, *j, *s, *t, *origem, *ultimo, *i_ciclo;
  Arc *a, *aux;
  void *temp;

```

Este código é usado no bloco 5.

21. Fila de vértices

A fila utilizada na resolução do problema de fluxo máximo auxiliar e na busca e no processamento de circuitos negativos 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.

```
<Fila de vértices 21>  $\equiv$ 
    Vertex * ini, *fim, *zero;
    Vertex * max;
void enqueue(Vertex * i)
{
    fim-vertice = i;
    i-nafila = TRUE;
    fim++;
    if (fim > max) {
        fim = zero;
    }
    if (fim  $\equiv$  ini) {
        fprintf(stderr, "ERRO: Fila excedeu sua capacidade.\n");
        exit(-1);
    }
}
void init_queue(Graph * g, Vertex * i)
{
    Vertex * j;
    ini = g-vertices;
    fim = g-vertices;
    zero = g-vertices;
    max = g-vertices + g-n;
    for (j = g-vertices; j < g-vertices + g-n; j++) {
        j-nafila = FALSE;
    }
    enqueue(i);
    return;
}
boolean empty_queue()
{
    if (ini  $\equiv$  fim) return TRUE;
    return FALSE;
}
Vertex * remove_first()
```

```

{
  Vertex * i;
  if (¬empty_queue()) {
    i = ini→vertice;
    ini ++;
    if (ini > max) {
      ini = zero;
    }
    i→nafila = FALSE;
    return i;
  }
  return Λ;
}
Vertex * last_enqueued()
{
  if (¬empty_queue()) {
    return fim→vertice;
  }
  return Λ;
}
boolean is_enqueued(Vertex * i)
{
  return i→nafila;
}

```

Este código é usado no bloco 29.

22. 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 de custo mínimo obtido e os custos reduzidos ótimos, que certificam a minimalidade do custo do fluxo obtido.

```
< Função principal 22 > ≡
int main(int argc, char *argv[])
{
    Graph *g;
    < Variáveis secundárias da função principal 28 >
    < Verifica consistência dos parâmetros 23 >
    cyclecanceling(g);
    < Imprime fluxo de custo mínimo 26 >
    < Imprime custos reduzidos ótimos 27 >
    return (0);
}
```

Este código é usado no bloco 29.

23. 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, o campo *a.I*, ao custo e o campo *u.I* dos vértices corresponde à demanda. Também é necessário 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 23 > ≡
if (argc ≠ 3) {
    fprintf(stderr, "Uso: %s <in> <out>\n", argv[0]);
    exit(-1);
}
< Verifica validade dos arquivos 24 >
< Verifica sinal das capacidades 25 >
```

Este código é usado no bloco 22.

24. 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 24 > ≡
if ((g = restore_graph(argv[1])) ≡ Λ) {
    fprintf(stderr, "ERRO: Problemas com arquivo de entrada.\n");
    exit(-2);
}
```

```

}
if ((saida = fopen(argv[2], "w")) ≡  $\Lambda$ ) {
    fprintf(stderr, "ERRO: Arquivo de saída inválido.\n");
    exit(-3);
}

```

Este código é usado no bloco 23.

25. 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 25 > ≡
for (i = gvertices; i < gvertices + gn; i++) {
    for (a = i→arcs; a; a = a→next) {
        if (a→cap < 0) {
            fprintf(stderr, "ERRO: Capacidade negativa encontrada.\n");
            exit(-4);
        }
    }
}

```

Este código é usado no bloco 23.

26. Impressão do fluxo viável de custo mínimo

Após a execução do algoritmo, se a instância for viável, imprime-se o fluxo e o custo. Para confirmar a viabilidade do fluxo obtido, imprime-se todos os excessos e demandas.

```

< Imprime fluxo de custo mínimo 26 > ≡
for (i = gvertices; i < gvertices + gn; i++) {
    i→exc = 0;
}
for (min = 0, i = gvertices; i < gvertices + gn; 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, flx(a));
            min += flx(a) * a→custo;
            i→exc -= flx(a);
            (a→tip)→exc += flx(a);
        }
    }
}
for (i = gvertices; i < gvertices + gn; i++) {
    fprintf(saida, "\s\ :\demanda\ld_e\excesso:\ld\n", i→name,
        i→demanda, i→exc);
    if (i→demanda ≠ i→exc) {

```

```

        fprintf(stderr, "ERRO: Fluxo nao-viavel.\n");
    }
}
fprintf(saida, "Custo: %ld\n", min);
fprintf(stdout, "Custo do fluxo encontrado: %ld\n", min);

```

Este código é usado no bloco 22.

27. Impressão dos custos reduzidos ótimos

Podemos fixar um vértice i_k de cada componente C_k da rede residual e definir o potencial $\pi(i)$ de um vértice i pertencente ao componente C_k como o oposto da distância entre i e i_k na rede residual. Feito isso, definimos o custo reduzido de um arco a como: $c_a^\pi := c_a - \pi(i) + \pi(j)$. Uma vez que ao final do processamento a rede residual não contém circuitos negativos, então o potencial π deve ser um certificado para a otimalidade do fluxo encontrado.

Os valores dos custos reduzidos dos arcos da rede residual certificam a otimalidade do fluxo viável obtido se todos eles forem não-negativos.

```

⟨ Imprime custos reduzidos ótimos 27 ⟩ ≡
    fprintf(saida, "Custos reduzidos:\n");
    for (i = g-vertices; i < g-vertices + g-n; i++) {
        for (a = i-arcs; a; a = a-next) {
            if (get_cap_residual(a) > 0) {
                custo_reduzido = a-custo - ((a-tip)-dist - (inicio(a))-dist);
                fprintf(saida, "Custo reduzido de %s\ %s\ %s\": %ld\n",
                    i-name, a-tip-name, custo_reduzido);
                if (custo_reduzido < 0) {
                    fprintf(stderr, "ERRO: Fluxo nao-é ótimo.\n");
                }
            }
        }
    }
}
fclose(saida);

```

Este código é usado no bloco 22.

28. Podemos, agora, definir as variáveis secundárias da função principal.

```

⟨ Variáveis secundárias da função principal 28 ⟩ ≡
    Arc * a;
    Vertex * i;
    FILE *saida;
    long min, custo_reduzido;

```

Este código é usado no bloco 22.

29. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

```
< Bibliotecas necessárias 30 >  
< Estruturas de informação 32 >  
< Fila de vértices 21 >  
< Função arco_original 12 >  
< Função get_capacidade_residual 13 >  
< Método do cancelamento de circuitos 5 >  
< Função principal 22 >
```

30. Bibliotecas

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

```
< Bibliotecas necessárias 30 > ≡  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <gb_graph.h>  
#include <gb_save.h>
```

Este código é usado no bloco 29.

31. Macros

Definimos aqui todas as macros utilizadas no programa.

```
#define boolean int  
#define FALSE 0  
#define TRUE 1  
#define demanda u.I  
#define exc v.I  
#define res v.I  
#define apred w.A  
#define dist x.I  
#define nafila y.I  
#define vertice z.V  
#define cap len  
#define custo a.I  
#define est b.I  
#define flx(a) ((struct str_arc *) a→b.I)→flx  
#define irmao(a) ((struct str_arc *) a→b.I)→irmao  
#define inicio(a) ((struct str_arc *) a→b.I)→irmao→tip
```

32. Estruturas

Conforme já mencionado, devido ao limite de campos utilitários do SGB, uma estrutura especial precisa ser definida.

⟨Estruturas de informação 32⟩ ≡

```
struct str_arc {  
    long flx;  
    Arc * irmao;  
};
```

Este código é usado no bloco 29.

Índice Remissivo

apred: 9, 14, 15, 16, 18, 19, 31.
Arc: 12, 13, 15, 20, 28, 32.
arco_original: 9, 12, 13, 19, 26.
arcs: 6, 8, 9, 10, 11, 15, 25, 26, 27.
argc: 22, 23.
argv: 22, 23, 24.
aux: 11, 20.
boolean: 20, 21, 31.
cap: 8, 10, 13, 25, 31.
cap_residual: 9, 20.
capres_ciclo: 18, 19, 20.
custo: 9, 15, 26, 27, 31.
custo_reduzido: 27, 28.
cyclecanceling: 5, 22.
d: 20.
demanda: 8, 10, 11, 26, 31.
dist: 14, 15, 27, 31.
empty_queue: 9, 15, 19, 21.
enqueue: 9, 15, 18, 21.
est: 6, 8, 9, 31.
exc: 26, 31.
exit: 5, 6, 8, 9, 21, 23, 24, 25.
FALSE: 10, 21, 31.
fclose: 27.
fm: 21.
flx: 9, 10, 12, 13, 19, 26, 31, 32.
fopen: 24.
fprintf: 5, 6, 8, 9, 21, 23, 24, 25, 26, 27.
gb_new_arc: 8, 9.
get_cap_residual: 9, 13, 15, 18, 27.
Graph: 5, 21, 22.
i_ciclo: 5, 14, 15, 18, 20.
ini: 21.
inicio: 9, 15, 18, 27, 31.
init_queue: 9, 15, 18, 21.
irmao: 9, 13, 19, 31, 32.
is_enqueued: 15, 21.
iteracoes: 5, 20.
last_enqueued: 15, 21.
len: 2, 31.
main: 22.
malloc: 6, 8, 9.
max: 21.
min: 26, 28.
nafla: 21, 31.
name: 26, 27.
next: 6, 9, 10, 11, 15, 25, 26, 27.
origem: 14, 15, 16, 20.
passo: 15, 20.
remove_first: 9, 15, 19, 21.
res: 9, 31.
restore_graph: 24.
saida: 24, 26, 27, 28.
stderr: 6, 8, 9, 21, 23, 24, 25, 26, 27.
stdout: 5, 26.
str_arc: 6, 8, 9, 31, 32.
temp: 6, 8, 9, 20.
tip: 9, 10, 11, 15, 26, 27, 31.
TRUE: 10, 21, 31.
ultimo: 15, 20.
Vertex: 20, 21, 28.
vertice: 21, 31.
vertices: 6, 8, 9, 10, 11, 14, 15, 16, 21, 25, 26, 27.
viavel: 5, 10, 11, 20.
zero: 21.

Lista de Refinamentos

- ⟨ Associa estrutura de campos utilitários aos arcos 6 ⟩ Usado no bloco 5.
- ⟨ Atualiza fluxo através do circuito 19 ⟩ Usado no bloco 17.
- ⟨ Bibliotecas necessárias 30 ⟩ Usado no bloco 29.
- ⟨ Busca circuito negativo acessível a partir de *origem* 15 ⟩ Usado no bloco 14.
- ⟨ Busca fluxo viável inicial 7 ⟩ Usado no bloco 5.
- ⟨ Cria vértices e arcos artificiais 8 ⟩ Usado no bloco 7.
- ⟨ Encontra capacidade residual do circuito 18 ⟩ Usado no bloco 17.
- ⟨ Estruturas de informação 32 ⟩ Usado no bloco 29.
- ⟨ Fila de vértices 21 ⟩ Usado no bloco 29.
- ⟨ Função principal 22 ⟩ Usado no bloco 29.
- ⟨ Função *arco_original* 12 ⟩ Usado no bloco 29.
- ⟨ Função *get_capacidade_residual* 13 ⟩ Usado no bloco 29.
- ⟨ Imprime custos reduzidos ótimos 27 ⟩ Usado no bloco 22.
- ⟨ Imprime fluxo de custo mínimo 26 ⟩ Usado no bloco 22.
- ⟨ Método do cancelamento de circuitos 5 ⟩ Usado no bloco 29.
- ⟨ Procura circuito negativo 14 ⟩ Usado no bloco 5.
- ⟨ Redefine *origem* 16 ⟩ Usado no bloco 14.
- ⟨ Reduz custo do fluxo através do circuito negativo 17 ⟩ Usado no bloco 5.
- ⟨ Remove vértices e arcos artificiais 11 ⟩ Usado no bloco 7.
- ⟨ Resolve problema do fluxo máximo auxiliar 9 ⟩ Usado no bloco 7.
- ⟨ Variáveis da função *cyclecanceling* 20 ⟩ Usado no bloco 5.
- ⟨ Variáveis secundárias da função principal 28 ⟩ Usado no bloco 22.
- ⟨ Verifica consistência dos parâmetros 23 ⟩ Usado no bloco 22.
- ⟨ Verifica se existe solução viável 10 ⟩ Usado no bloco 7.
- ⟨ Verifica sinal das capacidades 25 ⟩ Usado no bloco 23.
- ⟨ Verifica validade dos arquivos 24 ⟩ Usado no bloco 23.