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

Esta é uma implementação em CWEB-LaTeX 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 *litterate programming* CWEB-LaTeX:
`http://www-cs-faculty.stanford.edu/~knuth/cweb.html`
Sítio do projeto:
`http://www.ime.usp.br/~coelho/oticomb/`

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 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: Instˆancia n˜ao é viável!\n");
      exit(−1);
    }
    fprintf(stdout, "Número de iteraç˜oes: %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 alocaç~ao 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 *artificias*.

```
⟨ 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 alocaç~ao 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 alocaç~ao 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 alocaç~ao 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 minimo. */
  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 ∧ flx(a) ≠ a→cap) {
            viavel = FALSE;
          }
        }
      }
    }
  }
```

Este código é usado no bloco 7.

11. Após a resolução do problema auxiliar, os arcos artificias 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 (flx(a) ≥ 0);
  }
```

Este código é usado no bloco 29.

13. A capacidade residual de um arco da rede original correponde 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 (a→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→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→vertices; i < g→vertices + g→n; i++) {
    i→dist = −1;
    i→apred = Λ;
  }
  origem = g→vertices;
  do {
    ⟨ Busca circuito negativo acessível a partir de origem 15 ⟩
    if (i_ciclo ≡ Λ) {
      ⟨ Redefine origem 16 ⟩
    }
  } while (i_ciclo ≡ Λ ∧ origem ≠ Λ);
```

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 $i\rightarrow apred$ 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 ⟩ ≡

```
passo = 0;
ultimo = origem;
origem→dist = 0; origem→apred = ( Arc * ) origem;
init_queue(g, origem);
while ((passo ≠ g→n) ∧ (¬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 ∨ (j→apred ≡ Λ ∧ j ≠ origem)) {
        j→dist = d;
        j→apred = a;
        if (¬is_enqueued(j)) {
          enqueue(j);
        }
      }
    }
  }
  if (i ≡ ultimo) {
    ultimo = last_enqueued( );
    passo++;
  }
}
i_ciclo = Λ;
if (¬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 ≠ 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 encontrardo 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→apred ≡ 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→apred);
  for (i = inicio(i_ciclo→apred); i ≠ i_ciclo; i = inicio(i→apred)) {
    a = i→apred;
    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)) {
    flx(a) += capres_ciclo;
  }
  else {
    flx(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 $i\rightarrow vertice$ 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 $g\rightarrow vertices$. O primeiro vértice na fila é dado por $ini\rightarrow vertices$ e o último, por $fim\rightarrow vertices$. A condição $(ini \equiv fim)$ indica que a fila está vazia. A implementação supõe que no máximo $g\rightarrow n$ vértices ocuparão a fila ao mesmo tempo.

```
⟨ Fila de vértices 21 ⟩ ≡
  Vertex * ini, *fim, *zero;
  Vertex * max;
  void enqueue(Vertex * i)
  {
    fim→vertice = i;
    i→nafila = TRUE;
    fim++;
    if (fim > max) {
      fim = zero;
    }
    if (fim ≡ 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 ≡ 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")) ≡ Λ) {
  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 = 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(−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 = g→vertices; i < g→vertices + g→n; i++) {
    i→exc = 0;
  }
  for (min = 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, flx(a));
        min += flx(a) * a→custo;
        i→exc −= flx(a);
        (a→tip)→exc += flx(a);
      }
    }
  }
  for (i = g→vertices; i < g→vertices + g→n; 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\" a \"%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

Lista de Refinamentos