Problema do fluxo de custo mínimo
Algoritmo cost 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 cost scaling 3
   7. Obtenção do fluxo viável inicial 4
   14. Definição de potencial e limitante iniciais 8
   15. Melhorando a aproximação 8
22. Fila de vértices ativos 12
23. Função principal 14
   24. Consistência dos parâmetros 14
   27. Impressão do fluxo viável de custo mínimo 15
   28. Impressão dos custos reduzidos ótimos 16
30. Estrutura geral 17
   31. Bibliotecas 17
   32. Macros 17
   33. Estruturas 18

1. Introdução

Esta é uma implementação em `CWEB`-LaTeX do **algoritmo cost scaling** 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 costscaling.tex` para gerar o arquivo LaTeX de documentação.
`make costscaling.dvi` para gerar o arquivo DVI de visualização.
`make costscaling.pdf` para gerar o arquivo PDF de visualização.
`make costscaling.ps` para gerar o arquivo PostScript de visualização.
`make costscaling.c` para gerar o código-fonte C do programa.
`make costscaling` para gerar o executável do programa.
`costscaling` 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. Algoritmo cost scaling

O cost scaling é um algoritmo polinomial para a resolução do problema do fluxo de custo mínimo que utiliza idéias semelhantes às utilizadas no método do pré-fluxo para a resolução do problema do fluxo máximo.

O algoritmo é iterativo e mantém durante toda a sua execução um fluxo ϵ-ótimo que é uma *solução aproximada* para o problema. Em cada iteração, ele busca melhorar a aproximação transformando o fluxo ϵ-ótimo em um fluxo $\frac{\epsilon}{2}$-ótimo através de pushes e relabels. Além disso, o valor de ϵ é reduzido pela metade. O algoritmo pára quando $\epsilon < 1/n$, uma vez que, para ϵ nessas condições, todo fluxo ϵ-ótimo em uma rede com custos inteiros é ótimo.

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.

```
⟨ Algoritmo cost scaling 5 ⟩ ≡
  void costscaling(Graph *g)
  {
    ⟨ Variáveis da função costscaling 21 ⟩
    ⟨ Associa estrutura de campos utilitários aos arcos e vértices 6 ⟩
    ⟨ Busca fluxo viável inicial 7 ⟩
    ⟨ Define potencial e limitante epsilon iniciais 14 ⟩;
    iteracoes = 0;
    if (viavel) {
      while (epsilon ≥ 1) {
        ⟨ Melhora aproximação 15 ⟩
        epsilon = epsilon/2;
      }
    }
    else {
      fprintf(stdout, "ERRO:␣Inst^ancia␣n~ao␣é␣viável!\n");
      exit(−1);
    }
    fprintf(stdout, "Número␣de␣iteraç~oes:␣%d\n", iteracoes);
    return;
  }
```

Este código é usado no bloco 30.

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 e vértices 6 ⟩ ≡
  potencial = (double *) malloc(g→n * sizeof(double));
  indice = 0;
  for (i = g→vertices; i < g→vertices + g→n; i++) {
```

```
    i->pos = indice++;
    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;
    inicializa_fila(g);
    insere_fila(s);
    while (¬fila_vazia()) {
      i = remove_fila();
      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;
            insere_fila(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 30.

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 30.

14. Definição de potencial e limitante iniciais

O potencial inicial π é tal que $\pi(i) = 0$ para todo vértice i.

Sabemos que se C é o maior custo em módulo dentre os módulos dos custos de todos os arcos, então qualquer fluxo viável é ϵ-ótimo, se $\epsilon \geq C$.

Entretanto, ao invés de inicializarmos a variável *epsilon* com C, vamos multiplicar os custos de todos os arcos por $g \rightarrow n$ e definir $epsilon = 2^{\lceil \log(nC) \rceil}$, para que *epsilon* permaneça inteiro durante toda a execução do algoritmo.

⟨ Define potencial e limitante *epsilon* iniciais 14 ⟩ ≡

```
max_custo = −1;
for (i = g→vertices; i < g→vertices + g→n; i++) {
  potencial(i) = 0;
  for (a = i→arcs; a; a = a→next) {
    c = a→custo;
    if (c < 0) c *= −1;
    if (c > max_custo) max_custo = c;
    a→custo *= g→n;
  }
}
epsilon = pow(2, ceil(log(g→n * max_custo)/log(2)));
```
Este código é usado no bloco 5.

15. Melhorando a aproximação

Através da execução de pushes e relabels, o fluxo ϵ-ótimo é transformado em um fluxo $\frac{\epsilon}{2}$-ótimo. As operações push e relabel são executadas enquanto existem vértice ativos na rede. Nessa implementação, os vértices ativos são ordenados através de uma fila FIFO.

$\langle$ Melhora aproximação 15 $\rangle \equiv$
  $\langle$ Define pré-fluxo inicial e excessos 16 $\rangle$
  $\langle$ Insere vértices ativos na fila 17 $\rangle$
  while $(\neg \mathit{fila\_vazia}())$ {
    $i = \mathit{primeiro\_fila}()$;
    $\langle$ Executa push ou relabel 18 $\rangle$
    $\mathit{iteracoes}{+}{+}$;
  }

Este código é usado no bloco 5.

16. Antes da primeira iteração da melhoria de aproximação, um pré-fluxo x' é definido a partir do fluxo corrente x da seguinte maneira: $x'_a = 0$, para todo arco a tal que $c^\pi_a > 0$, $x'_a = u_a$, para todo arco a tal que $c^\pi_a < 0$ e $x'_a = x_a$, para todo arco a tal que $c^\pi_a = 0$.

O excesso de cada vértice com relação ao pré-fluxo x' é a diferença entre o fluxo x' que sai e o fluxo x' que entra. Aproveitamos também para inicializar o apontador para o arco corrente na lista de adjacências de cada vértice, o qual será utilizado na busca por arcos admissíveis.

$\langle$ Define pré-fluxo inicial e excessos 16 $\rangle \equiv$
  for $(i = g{\to}\mathit{vertices};\ i < g{\to}\mathit{vertices} + g{\to}n;\ i{+}{+})$ {
    $i{\to}\mathit{exc} = 0$;
    $i{\to}\mathit{atual} = i{\to}\mathit{arcs}$;
  }
  for $(i = g{\to}\mathit{vertices};\ i < g{\to}\mathit{vertices} + g{\to}n;\ i{+}{+})$ {
    for $(a = i{\to}\mathit{arcs};\ a;\ a = a{\to}\mathit{next})$ {
      if $(\mathit{arco\_original}(a))$ {
        $\mathit{custo\_reduzido} = a{\to}\mathit{custo} - \mathit{potencial}(i) + \mathit{potencial}(a{\to}\mathit{tip})$;
        if $(\mathit{custo\_reduzido} > 0)\ \mathit{flx}(a) = 0$;
        else if $(\mathit{custo\_reduzido} < 0)\ \mathit{flx}(a) = a{\to}\mathit{cap}$;
        $i{\to}\mathit{exc} \mathrel{-}= \mathit{flx}(a)$;
        $(a{\to}\mathit{tip}){\to}\mathit{exc} \mathrel{+}= \mathit{flx}(a)$;
      }
    }
  }

Este código é usado no bloco 15.

17. Um vértice é ativo com relação ao pré-fluxo corrente se o seu excesso é estritamente maior do que sua demanda. Todos os vértices que se tornaram ativos com a definição do pré-fluxo inicial devem ser inseridos na fila.

⟨ Insere vértices ativos na fila 17 ⟩ ≡

```
inicializa_fila(g);
for (i = g→vertices; i < g→vertices + g→n; i++) {
   if (i→exc > i→demanda) insere_fila(i);
}
```

Este código é usado no bloco 15.

18. Neste contexto do cost scaling, um arco é admissível se $-\frac{1}{2}\epsilon \leq c_a^\pi < 0$.

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. Deve-se examinar apenas arcos da rede residual, isto é, com capacidade residual positiva.

⟨ Executa push ou relabel 18 ⟩ ≡

```
for (a = i→atual; a; a = a→next) {
   if (get_cap_residual(a) > 0) {
      custo_reduzido = a→custo − potencial(i) + potencial(a→tip);
      if ((custo_reduzido < 0) ∧ (custo_reduzido ≥ −(epsilon/2))) break;
   }
}
if (a ≡ Λ) {
   ⟨ Executa um relabel 20 ⟩
   i→atual = i→arcs;
}
else {
   ⟨ Executa um push 19 ⟩
   i→atual = a;
}
```

Este código é usado no bloco 15.

19. Executar um push resume-se a atualizar valores: o fluxo no arco ou em seu irmão deve ser atualizado, bem como os excessos em seus extremos. Ademais, se a ponta final do arco que sofre a operação torna-se ativo, então tal vértice deve entrar na fila.

⟨ Executa um push 19 ⟩ ≡

```
j = a→tip;
alpha = get_cap_residual(a);
if ((i→exc − i→demanda) < alpha) {
   alpha = (i→exc − i→demanda);
}
if (¬arco_original(a)) {
   flx(irmao(a)) −= alpha;
}
else {
   flx(a) += alpha;
```

```
  }
  i→exc −= alpha;
  j→exc += alpha;
  if (i→exc ≡ i→demanda) {
    remove_fila();
  }
  if ((j→exc > j→demanda) ∧ ((j→exc − alpha) ≤ j→demanda)) {
    insere_fila(j);
  }
```

Este código é usado no bloco 18.

20. Um relabel de um vértice i consiste simplesmente no aumento de seu potencial em $\epsilon/2$.

⟨ Executa um relabel 20 ⟩ ≡

```
  potencial(i) += (epsilon/2);
  remove_fila();
  insere_fila(i);
```

Este código é usado no bloco 18.

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

⟨ Variáveis da função *costscaling* 21 ⟩ ≡

```
  int iteracoes, indice;
  long max_custo, c, cap_residual, alpha;
  double custo_reduzido, epsilon;
  boolean viavel;
  Vertex *i, *j, *s, *t;
  Arc *a, *aux;
  void *temp;
```

Este código é usado no bloco 5.

22. Fila de vértices ativos

A fila utilizada para ordenar os vértices ativos a serem processados 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 22 ⟩ ≡

```
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 ≡ ini) {
    fprintf(stderr, "ERRO:␣Fila␣excedeu␣sua␣capacidade.\n");
    exit(−1);
  }
}
boolean fila_vazia()
{
  if (ini ≡ 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 Λ;
  }
  Vertex * primeiro_fila()
  {
    if (¬fila_vazia()) {
      return ini→vertice;
    }
    return Λ;
  }
```
Este código é usado no bloco 30.

23. 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.

```
⟨ Função principal 23 ⟩ ≡
  int main(int argc, char *argv[])
  {
    Graph *g;
    ⟨ Variáveis secundárias da função principal 29 ⟩
    ⟨ Verifica consistência dos parâmetros 24 ⟩
    costscaling(g);
    ⟨ Imprime fluxo de custo mínimo 27 ⟩
    ⟨ Imprime custos reduzidos ótimos 28 ⟩
    return (0);
  }
```

Este código é usado no bloco 30.

24. 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 24 ⟩ ≡
  if (argc ≠ 3) {
    fprintf(stderr, "Uso:␣%s␣<in>␣<out>\n", argv[0]);
    exit(−1);
  }
  ⟨ Verifica validade dos arquivos 25 ⟩
  ⟨ Verifica sinal das capacidades 26 ⟩
```

Este código é usado no bloco 23.

25. 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 25 ⟩ ≡
  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 24.

26. 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 26 ⟩ ≡

```
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 24.

27. 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 27 ⟩ ≡

```
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/g→n);
      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 23.

28. Impressão dos custos reduzidos ótimos

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

⟨ Imprime custos reduzidos ótimos 28 ⟩ ≡

```
  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 − potencial(inicio(a)) +
            potencial(a→tip))/g→n;
        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 23.

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

⟨ Variáveis secundárias da função principal 29 ⟩ ≡

```
  Arc *a;
  Vertex *i;
  FILE *saida;
  long min, custo_reduzido;
```

Este código é usado no bloco 23.

30. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

⟨ Bibliotecas necessárias 31 ⟩
⟨ Estruturas de informação 33 ⟩
⟨ Fila de vértices 22 ⟩
⟨ Função *arco_original* 12 ⟩
⟨ Função *get_capacidade_residual* 13 ⟩
⟨ Algoritmo cost scaling 5 ⟩
⟨ Função principal 23 ⟩

31. Bibliotecas

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

⟨ Bibliotecas necessárias 31 ⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gb_graph.h>
#include <gb_save.h>
```

Este código é usado no bloco 30.

32. 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 pos w.I
#define apred x.A
#define atual y.A
#define vertice z.V
#define potencial(i) potencial[i->pos]
#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
```

33. Estruturas

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

⟨ Estruturas de informação 33 ⟩ ≡

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

Este código é usado no bloco 30.

Índice Remissivo

alpha: 19, 21.
apred: 9, 32.
Arc: 12, 13, 21, 29, 33.
arco_original: 9, 12, 13, 16, 19, 27.
arcs: 6, 8, 9, 10, 11, 14, 16, 18, 26, 27, 28.
argc: 23, 24.
argv: 23, 24, 25.
atual: 16, 18, 32.
aux: 11, 21.
boolean: 21, 22, 32.
c: 21.
cap: 8, 10, 13, 16, 26, 32.
cap_residual: 9, 21.
ceil: 14.
costscaling: 5, 23.
custo: 9, 14, 16, 18, 27, 28, 32.
custo_reduzido: 16, 18, 21, 28, 29.
demanda: 8, 10, 11, 17, 19, 27, 32.
epsilon: 5, 14, 18, 20, 21.
est: 6, 8, 9, 32.
exc: 16, 17, 19, 27, 32.
exit: 5, 6, 8, 9, 22, 24, 25, 26.
FALSE: 10, 22, 32.
fclose: 28.
fila_vazia: 9, 15, 22.
fim: 22.
flx: 9, 10, 12, 13, 16, 19, 27, 32, 33.
fopen: 25.
fprintf: 5, 6, 8, 9, 22, 24, 25, 26, 27, 28.
gb_new_arc: 8, 9.
get_cap_residual: 9, 13, 18, 19, 28.
Graph: 5, 22, 23.
indice: 6, 21.
ini: 22.
inicializa_fila: 9, 17, 22.
inicio: 9, 28, 32.
insere_fila: 9, 17, 19, 20, 22.
irmao: 9, 13, 19, 32, 33.
iteracoes: 5, 15, 21.
len: 2, 32.
log: 14.
main: 23.
malloc: 6, 8, 9.
max: 22.
max_custo: 14, 21.
min: 27, 29.
name: 27, 28.
next: 6, 9, 10, 11, 14, 16, 18, 26, 27, 28.
pos: 6, 32.
potencial: 6, 14, 16, 18, 20, 28, 32, 33.
pow: 14.
primeiro_fila: 15, 22.
remove_fila: 9, 19, 20, 22.
res: 9, 32.
restore_graph: 25.
saida: 25, 27, 28, 29.
stderr: 6, 8, 9, 22, 24, 25, 26, 27, 28.
stdout: 5, 27.
str_arc: 6, 8, 9, 32, 33.
temp: 6, 8, 9, 21.
tip: 9, 10, 11, 16, 18, 19, 27, 28, 32.
TRUE: 10, 22, 32.
Vertex: 21, 22, 29.
vertice: 22, 32.
vertices: 6, 8, 9, 10, 11, 14, 16, 17, 22, 26, 27, 28.
viavel: 5, 10, 11, 21.
zero: 22.

Lista de Refinamentos

⟨ Algoritmo cost scaling 5 ⟩ Usado no bloco 30.
⟨ Associa estrutura de campos utilitários aos arcos e vértices 6 ⟩ Usado no bloco 5.
⟨ Bibliotecas necessárias 31 ⟩ Usado no bloco 30.
⟨ Busca fluxo viável inicial 7 ⟩ Usado no bloco 5.
⟨ Cria vértices e arcos artificiais 8 ⟩ Usado no bloco 7.
⟨ Define potencial e limitante *epsilon* iniciais 14 ⟩ Usado no bloco 5.
⟨ Define pré-fluxo inicial e excessos 16 ⟩ Usado no bloco 15.
⟨ Estruturas de informação 33 ⟩ Usado no bloco 30.
⟨ Executa push ou relabel 18 ⟩ Usado no bloco 15.
⟨ Executa um push 19 ⟩ Usado no bloco 18.
⟨ Executa um relabel 20 ⟩ Usado no bloco 18.
⟨ Fila de vértices 22 ⟩ Usado no bloco 30.
⟨ Função principal 23 ⟩ Usado no bloco 30.
⟨ Função *arco_original* 12 ⟩ Usado no bloco 30.
⟨ Função *get_capacidade_residual* 13 ⟩ Usado no bloco 30.
⟨ Imprime custos reduzidos ótimos 28 ⟩ Usado no bloco 23.
⟨ Imprime fluxo de custo mínimo 27 ⟩ Usado no bloco 23.
⟨ Insere vértices ativos na fila 17 ⟩ Usado no bloco 15.
⟨ Melhora aproximação 15 ⟩ 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 *costscaling* 21 ⟩ Usado no bloco 5.
⟨ Variáveis secundárias da função principal 29 ⟩ Usado no bloco 23.
⟨ Verifica consistência dos parâmetros 24 ⟩ Usado no bloco 23.
⟨ Verifica se existe solução viável 10 ⟩ Usado no bloco 7.
⟨ Verifica sinal das capacidades 26 ⟩ Usado no bloco 24.
⟨ Verifica validade dos arquivos 25 ⟩ Usado no bloco 24.