Fluxos de custo mínimo
Método dos caminhos de viabilidade
Path 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

Esta é uma implementação em CWEB-LaTeX do algoritmo **path scaling**, uma implementação do **método dos caminhos de viabilidade** 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 pathscaling.tex` para gerar o arquivo LaTeX de documentação.
`make pathscaling.dvi` para gerar o arquivo DVI de visualização.
`make pathscaling.pdf` para gerar o arquivo PDF de visualização.
`make pathscaling.ps` para gerar o arquivo PostScript de visualização.
`make pathscaling.c` para gerar o código-fonte C do programa.
`make pathscaling` para gerar o executável do programa.
`pathscaling` 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`

5. Algoritmo path scaling

O método dos caminhos de viabilidade começa a partir de um fluxo não necessariamente viável e uma função potencial satisfazendo as condições de otimalidade e em cada iteração encontra um caminho de viabilidade e aumenta a viabilidade do fluxo através desse caminho. O algoritmo path scaling mantém um limitante inferior Δ, que é reduzido quando necessário, e em cada iteração só envia fluxo se a quantidade deste for superior a Δ. O método pára quando $\Delta = 1$ e não há mais caminhos de viabilidade, ou seja, quando uma varredura pelo grafo não consegue encontrar um vértice que está com excesso. 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 para obtermos a rede residual.

```
⟨ Algoritmo path scaling 5 ⟩ ≡
  void pathscaling(Graph * g)
  {
    ⟨ Variáveis da função pathscaling 24 ⟩
    ⟨ Tenta encontrar fluxo viável 6 ⟩
    ⟨ Obtém limitante inicial 10 ⟩
    ⟨ Constrói arcos artificiais 11 ⟩
    ⟨ Atribui fluxo inicial 14 ⟩
    ⟨ Constrói arcos irmãos 15 ⟩
    ⟨ Atribui potencial inicial 16 ⟩;
    iteracoes = 0;
    while (Delta ≥ 1) {
      ⟨ Encontra vértices excesso e deficit 18 ⟩
      while (excesso ≠ Λ ∧ deficit ≠ Λ) {
        ⟨ Obtém as distâncias dos vértices 19 ⟩
        ⟨ Atualiza os potenciais 22 ⟩
        ⟨ Aumenta viabilidade através de um caminho 23 ⟩
        ⟨ Encontra vértices excesso e deficit 18 ⟩
        iteracoes++;
      }
      ⟨ Elimina arcos não ótimos 17 ⟩
      Delta = Delta/2;
    }
    fprintf(stdout, "número de iteraç~oes: %d\n", iteracoes);
    return;
  }
```

Este código é usado no bloco 33.

6. Obtenção de fluxo viável

Para encontrar um fluxo viável, é preciso adicionar dois vértices ao grafo original, adicionar novos arcos e resolver um problema de fluxo máximo no novo grafo. Para facilitar o processamento, cria-se um novo grafo auxiliar.

⟨ Tenta encontrar fluxo viável 6 ⟩ ≡
```
⟨ Cria o novo grafo auxiliar 7 ⟩
⟨ Resolve problema de fluxo máximo 8 ⟩
⟨ Verifica se o fluxo obtido é viável 9 ⟩
```
Este código é usado no bloco 5.

7. O grafo auxiliar contém dois vértices a mais que o original: um faz o papel de fonte e outro o papel de sorvedouro do fluxo máximo. A fonte é ligada a cada vértice com demanda negativa e cada vértice com demanda positiva é ligado ao sorvedouro. A capacidade dos novos arcos corresponde à demanda.

⟨ Cria o novo grafo auxiliar 7 ⟩ ≡
```
h = gb_new_graph(g→n + 2);
fonte = h→vertices + g→n;
sorvedouro = h→vertices + g→n + 1;
for (i = h→vertices, j = g→vertices; j < g→vertices + g→n; i++, j++) {
  i→name = j→name;
  if (j→demanda < 0) gb_new_arc(fonte, i, −j→demanda);
  else gb_new_arc(i, sorvedouro, j→demanda);
  final = Λ;
  for (a = j→arcs; a; a = a→next) final = a;
  while (final) {
    k = h→vertices + (final→tip − g→vertices);
    gb_new_arc(i, k, final→cap);
    if (j→arcs ≡ final) final = Λ;
    else {
      for (a = j→arcs; a ≠ final; a = a→next) arctemp = a;
      final = arctemp;
    }
  }
}
```
Este código é usado no bloco 6.

8. O algoritmo para resolver o problema de fluxo máximo é o algoritmo dos vértices de maior rótulo, implementação do método do pré-fluxo. Esse algoritmo é detalhado em seu próprio documento, não havendo necessidade de comentá-lo.

⟨ Resolve problema de fluxo máximo 8 ⟩ ≡
```
for (i = h→vertices; i < h→vertices + h→n; i++) {
  i→dist = 0;
  i→exc = 0;
  i→atual = i→arcs;
}
for (i = h→vertices; i < h→vertices + h→n; i++) {
  for (a = i→arcs; a; a = a→next) {
```

```
      a→est = (int) malloc(sizeof(struct str_arc));
      if (i ≡ fonte ∧ a→tip ≠ fonte) flx(a) = a→cap;
      else flx(a) = 0;
      i→exc −= flx(a);
      a→tip→exc += flx(a);
    }
  }
  fonte→dist = h→n;
  for (i = h→vertices; i < h→vertices + h→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (flx(a) ≥ 0) {
        j = a→tip;
        gb_new_arc(j, i, a→cap);
        irmao(a) = j→arcs;
        irmao(a)→est = (int) malloc(sizeof(struct str_arc));
        flx(irmao(a)) = −1;
        irmao(irmao(a)) = a;
      }
    }
  }
  lista = ( Vertex * * ) malloc ( (2 * h→n) * sizeof ( Vertex * ) ) ;
  for (indice = 0; indice < 2 * h→n; indice++) lista[indice] = Λ;
  level = 0;
  size = 0;
  for (i = h→vertices; i < h→vertices + h→n; i++) {
    if (i ≠ sorvedouro ∧ i→exc > 0) {
      i→prox = lista[i→dist];
      lista[i→dist] = i;
      if (i→dist > level) level = i→dist;
      size++;
    }
  }
  while (size > 0) {
    while (lista[level] ≡ Λ) level−−;
    i = lista[level];
    lista[level] = i→prox;
    size−−;
    for (a = i→atual; a; a = a→next) {
      if (flx(a) ≥ 0) temp = a→cap − flx(a);
      else temp = flx(irmao(a));
      if (temp > 0 ∧ i→dist ≡ a→tip→dist + 1) break;
    }
    if (a ≡ Λ) {
      for (min = −1, a = i→arcs; a; a = a→next) {
        if (flx(a) ≥ 0) temp = a→cap − flx(a);
        else temp = flx(irmao(a));
```

```
        if (temp > 0 ∧ a→tip ≠ i) {
          if (min ≡ −1 ∨ min > a→tip→dist) min = a→tip→dist;
        }
      }
      i→dist = min + 1;
      i→prox = lista[i→dist];
      lista[i→dist] = i;
      if (i→dist > level) level = i→dist;
      size++;
      i→atual = i→arcs;
    }
    else {
      if (i→exc < temp) temp = i→exc;
      if (flx(a) ≥ 0) flx(a) += temp;
      else flx(irmao(a)) −= temp;
      i→exc −= temp;
      a→tip→exc += temp;
      if (i→exc > 0) {
        i→prox = lista[i→dist];
        lista[i→dist] = i;
        if (i→dist > level) level = i→dist;
        size++;
      }
      if (a→tip ≠ sorvedouro ∧ a→tip→exc > 0) {
        if (a→tip→exc − temp ≡ 0) {
          a→tip→prox = lista[a→tip→dist];
          lista[a→tip→dist] = a→tip;
          if (a→tip→dist > level) level = a→tip→dist;
          size++;
        }
      }
      i→atual = a;
    }
  }
  free(lista);
```

Este código é usado no bloco 6.

9. Para verificar se o fluxo máximo obtido corresponde a um fluxo viável para o grafo original, é preciso verificar se todos os arcos que saem da fonte ou entram no sorvedouro estão saturados. Caso exista um que não está saturado, o programa é imediatamente interrompido, pois o problema original é inviável.

```
⟨ Verifica se o fluxo obtido é viável 9 ⟩ ≡
  for (i = h→vertices; i < h→vertices + h→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (i ≡ fonte ∨ a→tip ≡ sorvedouro) {
```

```
        if (flx(a) ≥ 0 ∧ flx(a) < a→cap) {
          fprintf(stderr, "problema␣inviável\n");
          exit(0);
        }
      }
    }
  }
```

Este código é usado no bloco 6.

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

```
⟨ Obtém limitante inicial 10 ⟩ ≡
  for (Delta = −1, i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (a→custo ≥ 0) {
        if (Delta < a→custo) Delta = a→custo;
      }
      if (a→custo < 0) {
        if (Delta < −a→custo) Delta = −a→custo;
      }
    }
  }
  fprintf(stdout, "custo␣máximo:␣%d\n", Delta);
  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.

11. O método dos caminhos de viabilidade assume que o grafo residual é sempre fortemente conexo. Para garantir essa propriedade, adicionam-se arcos novos de um vértice a todos os outros e de todos os outros a esse vértice, onde cada arco novo tem capacidade muito alta. Esses arcos artificiais não podem fazer parte da solução ótima, então a eles é atribuído um custo muito alto.

```
⟨ Constrói arcos artificiais 11 ⟩ ≡
  ⟨ Obtém capacidade alta capalta 12 ⟩
  ⟨ Obtém custo alto custalto 13 ⟩
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      a→est = (int) malloc(sizeof(struct str_arc));
      novo(a) = FALSE;
```

```
    }
  }
  for (i = g→vertices + 1; i < g→vertices + g→n; i++) {
    gb_new_arc(g→vertices, i, capalta);
    g→vertices→arcs→custo = custalto;
    g→vertices→arcs→est = (int) malloc(sizeof(struct str_arc));
    novo(g→vertices→arcs) = TRUE;
    gb_new_arc(i, g→vertices, capalta);
    i→arcs→custo = custalto;
    i→arcs→est = (int) malloc(sizeof(struct str_arc));
    novo(i→arcs) = TRUE;
  }
```

Este código é usado no bloco 5.

12. A capacidade alta é a maior demanda vezes o número de vértices. E para garantir que a rede residual restrita é fortemente conexa, adicionamos Δ.

```
⟨ Obtém capacidade alta capalta 12 ⟩ ≡
  for (max = −1, i = g→vertices; i < g→vertices + g→n; i++) {
    if (i→demanda ≥ 0) {
      if (max < i→demanda) max = i→demanda;
    }
    else {
      if (max < −i→demanda) max = −i→demanda;
    }
  }
  capalta = (max ∗ g→n) + Delta;
```

Este código é usado no bloco 11.

13. O custo alto é o maior custo vezes o número de arcos.

```
⟨ Obtém custo alto custalto 13 ⟩ ≡
  for (max = −1, i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (max < a→custo) max = a→custo;
    }
  }
  custalto = max ∗ g→m;
```

Este código é usado no bloco 11.

14. O fluxo x definido na primeira iteração do algoritmo é tal que $x_a = 0$ para todo arco a tal que $c_a \geq 0$ e $x_a = u_a$ para todo arco a tal que $c_a < 0$.

⟨ Atribui fluxo inicial 14 ⟩ ≡

```
for (i = g→vertices; i < g→vertices + g→n; i++) {
  i→exc = 0;
}
for (i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (a→custo ≥ 0) flx(a) = 0;
    else flx(a) = a→cap;
    i→exc −= flx(a);
    a→tip→exc += flx(a);
  }
}
```

Este código é usado no bloco 5.

15. Os arcos irmãos são construídos exatamente segundo sua definição. Nesta implementação, os arcos irmãos são reconhecidos por terem fluxo negativo.

⟨ Constrói arcos irmãos 15 ⟩ ≡

```
for (i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (flx(a) ≥ 0) {
      j = a→tip;
      gb_new_arc(j, i, a→cap);
      irmao(a) = j→arcs;
      irmao(a)→est = (int) malloc(sizeof(struct str_arc));
      flx(irmao(a)) = −1;
      irmao(irmao(a)) = a;
      irmao(a)→custo = −a→custo;
    }
  }
}
```

Este código é usado no bloco 5.

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

⟨ Atribui potencial inicial 16 ⟩ ≡

```
potencial = (int *) malloc(g→n * sizeof(int));
for (indice = 0; indice < g→n; indice++) {
  potencial[indice] = 0;
}
```

Este código é usado no bloco 5.

17. O uso do limitante Δ pode fazer com que alguns arcos que não satisfazem as condições de otimalidade estejam indevidamente na rede residual. Se esse caso ocorre, estes arcos devem ser eliminados através da sua saturação.

```
⟨ Elimina arcos não ótimos 17 ⟩ ≡
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    for (a = i→arcs; a; a = a→next) {
      if (flx(a) ≥ 0) temp = a→cap − flx(a);
      else temp = flx(irmao(a));
      ddp = potencial[i − g→vertices] − potencial[a→tip − g→vertices];
      if (temp ≥ Delta/2 ∧ a→custo − ddp < 0) {
        if (flx(a) ≥ 0) flx(a) += temp;
        else flx(irmao(a)) −= temp;
        i→exc −= temp;
        a→tip→exc += temp;
      }
    }
  }
```

Este código é usado no bloco 5.

18. Os vértices em excesso e déficit são encontrados varrendo-se o grafo.

```
⟨ Encontra vértices excesso e deficit 18 ⟩ ≡
  excesso = Λ;
  deficit = Λ;
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    if (i→exc − i→demanda ≥ Delta) excesso = i;
    if (i→exc − i→demanda ≤ −Delta) deficit = i;
    if (excesso ≠ Λ ∧ deficit ≠ Λ) break;
  }
```

Este código é usado no bloco 5.

19. Atribuição de distâncias aos vértices

O algoritmo atribui a cada vértice um valor que corresponde à distância do vértice em excesso até ele. Distância nesse caso significa o custo de um caminho de custo mínimo com respeito aos custos reduzidos. O algoritmo também atribui a cada vértice um arco que é seu predecessor no caminho de custo mínimo. O algoritmo utilizado aqui é o de Dijkstra, que necessita de uma fila de prioridade.

```
⟨ Obtém as distâncias dos vértices 19 ⟩ ≡
  for (i = g→vertices; i < g→vertices + g→n; i++) {
    i→estado = NAOVISTO;
    i→arcopred = Λ;
  }
  inicializafila(g);
  excesso→estado = VISITADO;
  inserenafila(excesso, 0);
  while (¬filavazia( )) {
    i = retiradafila( );
```

```
    i→estado = EXAMINADO;
    ⟨ Examina vértice retirado da fila 20 ⟩
  }
  finalizafila( );
```
Este código é usado no bloco 5.

20. Ao examinar um vértice, visita-se seus vizinhos na rede residual que ainda não foram examinados. 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 pelo menos Δ. Utiliza-se uma variável temporária para armazenar a capacidade residual dos arcos.

```
⟨ Examina vértice retirado da fila 20 ⟩ ≡
  for (a = i→arcs; a; a = a→next) {
    j = a→tip;
    if (j→estado ≠ EXAMINADO) {
      if (flx(a) ≥ 0) temp = a→cap − flx(a);
      else temp = flx(irmao(a));
      if (temp ≥ Delta) {
        ⟨ Visita vértice vizinho 21 ⟩
      }
    }
  }
```
Este código é usado no bloco 19.

21. Se o vértice visitado não está na fila, ele é inserido com a prioridade obtida a partir do custo total do caminho do vértice em excesso até ele, com respeito aos custos reduzidos. Se ele está na fila, sua prioridade é atualizada caso o custo do caminho encontrado na iteração atual seja inferior ao anterior.

```
⟨ Visita vértice vizinho 21 ⟩ ≡
  ddp = potencial[i − g→vertices] − potencial[j − g→vertices];
  temp = i→dist + a→custo − ddp;
  if (j→estado ≡ NAOVISTO) {
    inserenafila(j, temp);
    j→estado = VISITADO;
    j→arcopred = a;
  }
  else if (temp < j→dist) {
    reinserenafila(j, temp);
    j→arcopred = a;
  }
```
Este código é usado no bloco 20.

22. Atualização dos potenciais

Os novos potenciais são obtidos subtraindo-se a distância obtida.

```
⟨ Atualiza os potenciais 22 ⟩ ≡
  for (indice = 0; indice < g→n; indice++)
    potencial[indice] −= (g→vertices + indice)→dist;
```

Este código é usado no bloco 5.

23. Aumento da viabilidade através de um caminho

Como cada vértice acessível a partir do vértice em excesso tem um arco predecessor definido, o aumento da viabilidade é simples: basta percorrer o caminho a partir do vértice em déficit, valendo-se dos arcos predecessores, modificando o fluxo em cada arco de acordo com a rede na qual ele se encontra.

```
⟨ Aumenta viabilidade através de um caminho 23 ⟩ ≡
  a = deficit→arcopred;
  while (a ≠ Λ) {
    if (flx(a) ≥ 0) flx(a) += Delta;
    else flx(irmao(a)) −= Delta;
    inicio(a)→exc −= Delta;
    a→tip→exc += Delta;
    if (inicio(a) ≡ fonte) a = Λ;
    else a = inicio(a)→arcopred;
  }
```

Este código é usado no bloco 5.

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

```
⟨ Variáveis da função pathscaling 24 ⟩ ≡
  Graph *h;
  Vertex **lista;
  Arc *a, *final, *arctemp;
  Vertex *i, *j, *k, *fonte, *sorvedouro, *excesso, *deficit;
  int iteracoes, temp, min, level, size, indice, capalta, custalto, max, ddp,
      Delta;
```

Este código é usado no bloco 5.

25. Fila de prioridade

A estrutura de dados aqui utilizada para implementar a fila de prioridade é um *heap*. A implementação abaixo é totalmente baseada no livro *Introduction to Algorithms* de T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein.

⟨ Fila de prioridade 25 ⟩ ≡

```
Vertex **heap;
int heapsize;
int parent(int k)
{
  return (k/2);
}
int left(int k)
{
  return (2 * k);
}
int right(int k)
{
  return ((2 * k) + 1);
}
void exchange(int k1, int k2)
{
  Vertex *i;
  i = heap[k1];
  heap[k1] = heap[k2];
  heap[k2] = i;
  heap[k1]->heapindex = k1;
  heap[k2]->heapindex = k2;
  return;
}
void heapify(int k)
{
  int l, r, menor;
  l = left(k);
  r = right(k);
  if (l <= heapsize ∧ heap[l]->dist < heap[k]->dist) menor = l;
  else menor = k;
  if (r <= heapsize ∧ heap[r]->dist < heap[menor]->dist) menor = r;
  if (menor ≠ k) {
    exchange(k, menor);
    heapify(menor);
  }
  return;
}
```

```
void inicializafila(Graph * g) { heap = ( Vertex * * ) malloc ( (g→n + 1) *
        sizeof ( Vertex * ) ) ;
    heapsize = 0;
    return; } void finalizafila( )
    {
      free(heap);
      return;
    }
    boolean filavazia( )
    {
      if (heapsize ≡ 0) return (TRUE);
      return (FALSE);
    }
    Vertex * retiradafila( )
    {
      Vertex * i;
      i = heap[1];
      heap[1] = heap[heapsize];
      heap[1]→heapindex = 1;
      heapsize−−;
      heapify(1);
      return (i);
    }
    void reinserenafila(Vertex * i, int key)
    {
      int k;
      k = i→heapindex;
      i→dist = key;
      while (k > 1 ∧ heap[parent(k)]→dist > heap[k]→dist) {
        exchange(k, parent(k));
        k = parent(k);
      }
      return;
    }
    void inserenafila(Vertex * i, int key)
    {
      heapsize++;
      heap[heapsize] = i;
      i→heapindex = heapsize;
      reinserenafila(i, key);
      return;
    }
```

Este código é usado no bloco 33.

26. 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 26 ⟩ ≡
  int main(int argc, char *argv[])
  {
    Graph * g;
    ⟨ Variáveis secundárias da função principal 32 ⟩
    ⟨ Verifica consistência dos parâmetros 27 ⟩
    pathscaling(g);
    ⟨ Imprime fluxo de custo mínimo 30 ⟩
    ⟨ Imprime custos reduzidos ótimos 31 ⟩
    return (0);
  }
```

Este código é usado no bloco 33.

27. 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 27 ⟩ ≡
  if (argc ≠ 3) {
    fprintf(stderr, "%s␣<in>␣<out>\n", argv[0]);
    exit(−1);
  }
  ⟨ Verifica validade dos arquivos 28 ⟩
  ⟨ Verifica sinal das capacidades 29 ⟩
```

Este código é usado no bloco 26.

28. 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 28 ⟩ ≡
  if ((g = restore_graph(argv[1])) ≡ Λ) {
    fprintf(stderr, "entrada␣inválida\n");
    exit(−2);
  }
```

```
if ((saida = fopen(argv[2], "w")) ≡ Λ) {
  fprintf(stderr, "saída␣inválida\n");
  exit(−3);
}
```
Este código é usado no bloco 27.

29. 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 29 ⟩ ≡
for (i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (a→cap < 0) {
      fprintf(stderr, "capacidades␣negativas\n");
      exit(−5);
    }
  }
}
```
Este código é usado no bloco 27.

30. Impressão do fluxo viável de custo mínimo

Após a execução do algoritmo, 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 30 ⟩ ≡
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);
}
for (min = 0, i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (flx(a) ≥ 0 ∧ novo(a) ≡ FALSE) {
      fprintf(saida, "fluxo␣de␣\"%s\"␣a␣\"%s\":␣%d\n", i→name,
          a→tip→name, flx(a));
      min += flx(a) ∗ a→custo;
    }
  }
}
fprintf(saida, "custo:␣%d\n", min);
```
Este código é usado no bloco 26.

31. 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 31 ⟩ ≡

```
fprintf(saida, "custos␣reduzidos:\n");
for (min = 0, i = g→vertices; i < g→vertices + g→n; i++) {
  for (a = i→arcs; a; a = a→next) {
    if (flx(a) ≥ 0) temp = a→cap − flx(a);
    else temp = flx(irmao(a));
    if (flx(a) ≥ 0 ∧ temp > 0 ∧ novo(a) ≡ FALSE) {
      ddp = potencial[i − g→vertices] − potencial[a→tip − g→vertices];
      fprintf(saida, "custo␣reduzido␣de␣\"%s\"␣a␣\"%s\":␣%ld\n",
          i→name, a→tip→name, a→custo − ddp);
    }
    if (flx(a) < 0 ∧ temp > 0 ∧ novo(irmao(a)) ≡ FALSE) {
      ddp = potencial[i − g→vertices] − potencial[a→tip − g→vertices];
      fprintf(saida, "custo␣reduzido␣de␣\"%s\"␣a␣\"%s\":␣%ld\n",
          i→name, a→tip→name, a→custo − ddp);
    }
  }
}
```

Este código é usado no bloco 26.

32. Podemos agora definir as variáveis secundárias da função principal.

⟨ Variáveis secundárias da função principal 32 ⟩ ≡

```
Arc *a;
Vertex *i;
FILE *saida;
int min, ddp, temp;
```

Este código é usado no bloco 26.

33. Estrutura geral

Para concluir o programa basta definir a estrutura geral.

```
⟨ Bibliotecas necessárias 34 ⟩
⟨ Estruturas de informação 36 ⟩
int *potencial;
⟨ Fila de prioridade 25 ⟩
⟨ Algoritmo path scaling 5 ⟩
⟨ Função principal 26 ⟩
```

34. Bibliotecas

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

```
⟨ Bibliotecas necessárias 34 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gb_graph.h>
#include <gb_save.h>
```

Este código é usado no bloco 33.

35. 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 EXAMINADO 2
#define demanda u.I
#define dist v.I
#define exc w.I
#define atual x.A
#define arcopred x.A
#define prox y.V
#define heapindex y.I
#define estado z.I
#define cap len
#define custo a.I
#define est b.I
#define flx(a) ((struct str_arc *) a→b.I)→f
#define irmao(a) ((struct str_arc *) a→b.I)→i
#define novo(a) ((struct str_arc *) a→b.I)→n
#define inicio(a) ((struct str_arc *) a→b.I)→i→tip
```

36. Estruturas

Devido ao limite de campos do SGB, estruturas especiais são necessárias.

⟨ Estruturas de informação 36 ⟩ ≡

```
struct str_arc {
  int f;
  Arc * i;
  booleann;
};
```

Este código é usado no bloco 33.

Índice Remissivo

Lista de Refinamentos