

Problema do fluxo máximo
Método dos caminhos de aumento
Fluxos bloqueadores de aumento

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 dos fluxos bloqueadores de aumento	3
11. Obtenção da rede acíclica	5
16. Obtenção do fluxo bloqueador	7
23. Aumento do fluxo através do fluxo bloqueador	10
25. Fila de vértices	11
26. Função principal	13
28. Consistência dos parâmetros	13
33. Impressão do fluxo de intensidade máxima	15
34. Impressão do separador de capacidade mínima	16
36. Estrutura geral	17
38. Bibliotecas	17
39. Macros	17

1. Introdução

Esta é uma implementação em $\text{CWEB-L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ do **algoritmo dos fluxos bloqueadores de aumento**, uma versão do **método dos caminhos de aumento** para resolver o **problema do fluxo máximo**. A plataforma SGB é necessária.

2. Descrição

Este programa recebe o nome de um arquivo que contém um grafo no formato SGB, o nome de um arquivo de saída, o nome de um vértice fonte e o nome de um vértice sorvedouro e imprime no arquivo de saída um fluxo de intensidade máxima e um separador de capacidade mínima da rede representada pelo grafo. Assume-se que as capacidades dos arcos estão representadas no campo *len*. Visando à realização de testes experimentais, o parâmetro adicional `-p` pode ser informado, caso se deseje que os fluxos bloqueadores sejam obtidos em redes acíclicas produzidas por busca em profundidade.

3. Compilação e execução

```
make flxbloq.tex para gerar o arquivo  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  de documentação.  
make flxbloq.dvi para gerar o arquivo DVI de visualização.  
make flxbloq.pdf para gerar o arquivo PDF de visualização.  
make flxbloq.ps para gerar o arquivo PostScript de visualização.  
make flxbloq.c para gerar o código-fonte C do programa.  
make flxbloq para gerar o executável do programa.  
flxbloq 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* $\text{CWEB-L}^{\text{A}}\text{T}_{\text{E}}\text{X}$:

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

Sítio do projeto:

<http://www.ime.usp.br/~coelho/oticomb/>

5. Algoritmo dos fluxos bloqueadores de aumento

O método dos caminhos de aumento recebe um grafo, representando uma rede capacitada, dois vértices s e t e devolve um st -fluxo de intensidade máxima e um st -separador de capacidade mínima nessa rede. Os vértices s e t são, respectivamente, a *fonte* e o *sorvedouro*. O st -separador mínimo é um certificado para a maximalidade do fluxo encontrado. O método começa com um st -fluxo x e busca, em cada iteração, aumentar sua intensidade através de um caminho de aumento. Quando a rede residual com relação a x não contém caminhos de aumento, o st -fluxo x é máximo e o método pára.

Dizemos que um st -fluxo x em uma rede é **bloqueador** se cada caminho de s a t na rede possui um arco saturado, isto é, cuja capacidade residual é nula com relação ao fluxo x . Podemos dizer também que um fluxo bloqueador é um **fluxo maximal** em uma rede.

O algoritmo dos fluxos bloqueadores de aumento mantém um st -fluxo x , representado pelo campo $flx(a)$ de cada arco a , e procura aumentar sua intensidade a cada iteração através de fluxos bloqueadores na **rede residual de caminhos mínimos**. A rede residual de caminhos mínimos é uma subrede acíclica da rede residual que contém apenas os arcos que estão em algum caminho mínimo entre a fonte e o sorvedouro nessa rede.

Note que, pelo teorema da decomposição de fluxos, um fluxo bloqueador em uma rede acíclica pode ser decomposto em um conjunto de fluxos-caminhos. Assim, aumentar a intensidade de um fluxo x através de um fluxo bloqueador em uma subrede acíclica da rede residual, equivale a aumentar a intensidade de x sucessivamente através de uma seqüência de caminhos de aumento. Portanto, o algoritmo dos fluxos bloqueadores de aumento é uma variação do método dos caminhos de aumento.

6. Antes da primeira iteração, um st -fluxo inicial é definido na rede e os arcos irmãos são gerados. Feito isso, o processo iterativo de busca por fluxos bloqueadores e aumento da intensidade do fluxo pode ser iniciado. Se o parâmetro *largura* estiver ativo, então a subrede acíclica da rede residual a ser considerada é a rede residual de caminhos mínimos, conforme descrição original do algoritmo, obtida a partir de uma busca em largura. Caso contrário, a rede acíclica é construída a partir de uma busca em profundidade. Apesar de não termos estabelecido limitantes para o consumo de tempo do algoritmo baseado nessa alteração, estamos interessados em analisá-lo experimentalmente.

A condição ($t\text{-}dist \equiv -1$) indica que não existem mais caminhos de aumento na rede residual e que, portanto, o algoritmo pode parar. A validade de tal condição de parada está justificada na descrição dos procedimentos para obtenção da rede acíclica. Devido ao interesse na complexidade experimental do algoritmo, imprime-se o número total de iterações após a execução.

(Algoritmo dos fluxos bloqueadores de aumento 6) \equiv

```
void fluxos_bloqueadores_aumento(Graph * g, Vertex * s, Vertex * t,
    boolean largura)
```

```

{
  < Variáveis da função fluxos_bloqueadores_aumento 24 >
  < Define associação dos arcos aos vetores de fluxos 7 >
  < Obtém fluxo inicial 8 >
  < Constrói arcos irmãos 9 >
  iteracoes = 0;
  do {
    if (largura) {
      < Obtém rede residual de caminhos mínimos 12 >
    }
    else {
      < Obtém rede acíclica com busca em profundidade 13 >
    }
    if (t-dist ≠ -1) {
      < Encontra fluxo bloqueador na rede acíclica 16 >
      < Aumenta fluxo através do fluxo bloqueador 23 >
      iteracoes++;
    }
  } while (t-dist ≠ -1);
  fprintf(stdout, "Número de iterações: %ld\n", iteracoes);
  return;
}

```

Este código é usado no bloco 36.

7. $flux$ e $flux_bloq$ são vetores cujas posições representam fluxo corrente e fluxo bloqueador em cada arco. A utilização de tais vetores é necessária pois o SGB disponibiliza apenas 2 campos utilitários para cada arco e estamos utilizando mais campos. A associação de cada arco a uma posição desses vetores será feita através do campo a_pos .

```

< Define associação dos arcos aos vetores de fluxos 7 > ≡
  indice = 0;
  for (i = g-vertices; i < g-vertices + g-n; i++) {
    for (a = i-arcs; a; a = a-next) {
      a-pos = indice++;
    }
  }

```

Este código é usado no bloco 6.

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

```

< Obtém fluxo inicial 8 > ≡
  for (i = g-vertices; i < g-vertices + g-n; i++) {
    for (a = i-arcs; a; a = a-next) {
      flux(a) = 0;
    }
  }

```

```

    }
  }

```

Este código é usado no bloco 6.

9. Os arcos irmãos são construídos exatamente segundo sua definição. Note que o arco irmão gerado é inicialmente apontado por $j \rightarrow arcs$.

```

⟨ Constrói arcos irmãos 9 ⟩ ≡
  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, a→cap);
        a→irmão = j→arcs;
        a→irmão→irmão = a;
        a→irmão→pos = indice++;
        flux(a→irmão) = -1;
      }
    }
  }

```

Este código é usado no bloco 6.

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

```

⟨ Função arco_original 10 ⟩ ≡
  int arco_original(Arc * a)
  {
    return (flux(a) ≥ 0);
  }

```

Este código é usado no bloco 36.

11. Obtenção da rede acíclica

Visitando cada vértice apenas uma vez, através de uma busca na rede residual, construímos uma árvore de busca. Tal árvore é a subrede acíclica a ser considerada pelo algoritmo na iteração.

Durante a busca, o campo $i \rightarrow dist$ de cada vértice i da rede é definido com a distância entre i e o vértice fonte s na árvore de busca. Assim, ao final do processamento, a subrede acíclica será caracterizada pela presença de arcos ij tais que $j \rightarrow dist = i \rightarrow dist + 1$.

Antes do início da busca, a distância de cada vértice a s é definida como -1. Sendo assim, se ao final do processamento $t \rightarrow dist \equiv -1$, então o vértice sorvedouro t não é acessível a partir do vértice fonte s na rede residual e, portanto,

não existem caminhos de aumento. Isso explica a condição de parada do **while** na função *fluxos_bloqueadores_aumento*.

Note que a busca deve ser feita na rede residual e, portanto, apenas arcos com capacidade residual positiva são considerados.

12. Se uma busca em largura for utilizada para o procedimento, a subrede acíclica será uma rede de caminhos mínimos.

```

< Obtém rede residual de caminhos mínimos 12 > ≡
for (i = g-vertices; i < g-vertices + g-n; i++) {
    i-dist = -1;
}
s-dist = 0;
inicializa_filas(g);
insere_fila(s);
while (¬filas_vazia()) {
    i = remove_fila();
    for (a = i-arcs; a; a = a-next) {
        if (get_cap_residual(a) > 0) {
            j = a-tip;
            if (j-dist ≡ -1) {
                j-dist = i-dist + 1;
                insere_fila(j);
            }
        }
    }
}

```

Este código é usado no bloco 6.

13. Uma busca em profundidade também determina uma subrede acíclica.

```

< Obtém rede acíclica com busca em profundidade 13 > ≡
for (i = g-vertices; i < g-vertices + g-n; i++) {
    i-dist = -1;
}
s-dist = 0;
busca_em_profundidade(s);

```

Este código é usado no bloco 6.

14. A busca em profundidade é feita através de uma função recursiva.

```

< Função busca_em_profundidade 14 > ≡
void busca_em_profundidade(Vertex * i)
{
    Vertex * j;
}

```

```

    Arc * a;
    for (a = i→arcs; a; a = a→next) {
        if (get_cap_residual(a) > 0) {
            j = a→tip;
            if (j→dist ≡ -1) {
                j→dist = i→dist + 1;
                busca_em_profundidade(j);
            }
        }
    }
    return;
}

```

Este código é usado no bloco 36.

15. 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 15 ⟩ ≡
    long get_cap_residual(Arc * a)
    {
        if (¬ arco_original(a)) {
            return flux(a→irmao);
        }
        else {
            return (a→cap - flux(a));
        }
    }

```

Este código é usado no bloco 36.

16. Obtenção do fluxo bloqueador

Para a obtenção do fluxo bloqueador, tentamos enviar uma quantidade máxima de fluxo a partir da fonte s ao sorvedouro t . A idéia é enviar o fluxo recursivamente através dos vizinhos da fonte e de forma que todos os caminhos de s a t possuam um arco saturado ao final do processamento.

O fluxo bloqueador será representado pelo campo $flux_bloq(a)$ de cada arco a da rede acíclica.

```

⟨ Encontra fluxo bloqueador na rede acíclica 16 ⟩ ≡
    ⟨ Gera limitante fluxo bloqueador 17 ⟩
    ⟨ Inicializa fluxo bloqueador 18 ⟩
    envia_fluxo_bloqueador(s, t, max_fluxo);

```

Este código é usado no bloco 6.

17. Um limitante para o fluxo bloqueador é dado pela soma das capacidades residuais dos arcos que saem da fonte na rede acíclica.

```

< Gera limitante fluxo bloqueador 17 > ≡
    max_fluxo = 0;
    for (a = s→arcs; a; a = a→next) {
        if (a→tip→dist ≡ s→dist + 1) max_fluxo += get_cap_residual(a);
    }

```

Este código é usado no bloco 16.

18. Antes de encontrar o fluxo bloqueador da iteração, “limpamos” o fluxo bloqueador da iteração passada.

```

< Inicializa fluxo bloqueador 18 > ≡
    for (i = g→vertices; i < g→vertices + g→n; i++) {
        for (a = i→arcs; a; a = a→next) {
            flx_bloc(a) = 0;
        }
    }

```

Este código é usado no bloco 16.

19. Na função *envia_fluxo_bloqueador*, o parâmetro *fluxo* indica a quantidade de fluxo que o vértice *i* recebeu da fonte. Busca-se, então, repassar esse fluxo aos vizinhos de *i* na rede acíclica. Primeiramente, tenta-se enviar ao primeiro vizinho de *i*, digamos *j*, o mínimo entre o fluxo recebido e a capacidade residual do arco que os conecta. O vizinho *j* tenta enviar o fluxo recebido recursivamente e devolve a quantidade de fluxo que efetivamente chegou até o sorvedouro. A variável *flx_sobra* é atualizada, então, com a quantidade de fluxo que o vértice *i* ainda pode tentar distribuir. O processo é repetido para os demais vizinhos de *i*, sempre tentando repassar o máximo da quantidade de fluxo que ainda resta, o que equivale ao mínimo entre *flx_sobra* e a capacidade residual do arco que liga *i* ao vizinho. Note que a função devolve a quantidade de fluxo que efetivamente chegou à fonte a partir do vértice *i*.

```

< Função envia_fluxo_bloqueador 19 > ≡
    long envia_fluxo_bloqueador(Vertex *i, Vertex *t, long fluxo)
    {
        { Variáveis da função envia_fluxo_bloqueador 22 }
        flx_sobra = fluxo;
        if (i ≡ t) {
            return fluxo;
        }
        for (a = i→arcs; a ∧ flx_sobra > 0; a = a→next) {
            if (get_cap_residual(a) > 0) {
                j = a→tip;
                if (j→dist ≡ i→dist + 1) {

```

```

        }
    }
}
return (fluxo - flux_sobra);
}

```

Este código é usado no bloco 36.

20. Um vértice só pode repassar o fluxo a um vizinho se a capacidade residual do arco que os conecta ainda não foi atingida pelo fluxo bloqueador em construção.

```

< Busca repassar fluxo ao vizinho 20 > ≡
if (flux_bloq(a) < get_cap_residual(a)) {
    flux_caminho = envia_fluxo_bloqueador(j, t, min(flux_sobra,
        (get_cap_residual(a) - flux_bloq(a)));
    flux_bloq(a) += flux_caminho;
    flux_sobra -= flux_caminho;
}

```

Este código é usado no bloco 19.

21. É preciso definir a função auxiliar *min*, utilizada acima, que devolve o mínimo entre os inteiros *a* e *b*.

```

< Função min 21 > ≡
long min(long a, long b)
{
    if (a > b) {
        return b;
    }
    return a;
}

```

Este código é usado no bloco 36.

22. Resta também definir as variáveis da função *envia_fluxo_bloqueador*.

```

< Variáveis da função envia_fluxo_bloqueador 22 > ≡
    Vertex * j;
    Arc * a;
    long flux_sobra;
    long flux_caminho;

```

Este código é usado no bloco 19.

23. Aumento do fluxo através do fluxo bloqueador

Uma vez que o fluxo bloqueador foi definido, podemos aumentar a intensidade do fluxo corrente na rede da seguinte forma: somamos ao fluxo de um arco original o fluxo bloqueador que o mesmo transporta e subtraímos o fluxo bloqueador transportado por seu irmão.

```
< Aumenta fluxo através do fluxo bloqueador 23 > ≡  
  for (i = gvertices; i < gvertices + gn; i++) {  
    for (a = i→arcs; a; a = a→next) {  
      if (¬arco_original(a)) {  
        flux(a→irmao) -= flux_bloc(a);  
      }  
      else {  
        flux(a) += flux_bloc(a);  
      }  
    }  
  }
```

Este código é usado no bloco 6.

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

```
< Variáveis da função fluxos_bloqueadores_aumento 24 > ≡  
  long iteracoes, indice, max_fluxo;  
  Vertex * i, * j;  
  Arc * a;
```

Este código é usado no bloco 6.

25. Fila de vértices

A fila utilizada na busca em largura é implementada como uma fila circular através da utilização do próprio vetor de vértices de um grafo do SGB.

O campo $i \rightarrow \text{vertice}$ 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 $g \rightarrow \text{vertices}$. O primeiro vértice na fila é dado por $ini \rightarrow \text{vertices}$ e o último, por $fim \rightarrow \text{vertices}$. 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, o que sempre ocorre durante uma busca na rede residual.

```
< Fila circular de vértices 25 > ≡
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  $\Lambda$ ;
}
Vertex * primeiro_fila()
{
    if ( $\neg$ fila_vazia()) {
        return ini-vertice;
    }
    return  $\Lambda$ ;
}
```

Este código é usado no bloco 36.

26. Função principal

O programa consiste basicamente de três fases: inicialização, execução do algoritmo e finalização. A inicialização consiste em verificar a consistência dos parâmetros de entrada. A aplicação do algoritmo é simplesmente a chamada da função que já definimos anteriormente. A finalização consiste em imprimir no arquivo de saída o fluxo máximo obtido e o separador de capacidade mínima.

```
< Função principal 26 > ≡
int main(int argc, char *argv[])
{
    Graph *g;
    Vertex *fonte, *sorvedouro;
    boolean largura;
    < Variáveis secundárias da função principal 35 >
    < Verifica consistência dos parâmetros 28 >
    < Aloca vetores flux e flux_bloq 27 >
    fluxos_bloqueadores_aumento(g, fonte, sorvedouro, largura);
    < Imprime fluxo máximo 33 >
    < Imprime separador de capacidade mínima 34 >
    return (0);
}
```

Este código é usado no bloco 36.

27. Entretanto, antes da aplicação do algoritmo, é preciso alocar os vetores *flux* e *flux_bloq* que armazenam respectivamente o fluxo corrente em cada arco e o fluxo bloqueador da iteração. Os vetores são alocados com tamanho igual a duas vezes o número de arcos do grafo, pois armazenarão dados sobre arcos originais e sobre seus arcos irmãos.

```
< Aloca vetores flux e flux_bloq 27 > ≡
flux = (long *) malloc(2 * g→m * sizeof(long));
flux_bloq = (long *) malloc(2 * g→m * sizeof(long));
if (¬flux ∨ ¬flux_bloq) {
    fprintf(stderr, "ERRO: Problemas com alocação de memória.\n");
    exit(-1);
}
```

Este código é usado no bloco 26.

28. 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* de cada arco corresponda à sua capacidade. Também é necessário que os nomes de vértices referenciem vértices que de fato existam no grafo e que a rede contenha somente capacidades não-negativas. Por fim, é preciso verificar se a opção

referente ao tipo de busca foi informada corretamente. Caso um número de parâmetros incorreto seja fornecido, as instruções do programa são impressas, exibindo a sintaxe.

```

< Verifica consistência dos parâmetros 28 > ≡
  if (argc ≠ 5 ∧ argc ≠ 6) {
    fprintf(stderr, "Uso: %s<in><out>\\"source\\" \"sink\\" [-p] \n",
              argv[0]);
    exit(-1);
  }
< Verifica validade dos arquivos 29 >
< Verifica existência dos vértices 30 >
< Verifica sinal das capacidades 31 >
< Verifica tipo de busca 32 >

```

Este código é usado no bloco 26.

29. 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 29 > ≡
  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 28.

30. Os vértices do grafo são examinados um por um até que os nomes fornecidos sejam encontrados. No caso de nomes iguais, considera-se o primeiro.

```

< Verifica existência dos vértices 30 > ≡
  fonte = Λ;
  sorvedouro = Λ;
  for (i = g-vertices; i < g-vertices + g-n; i++) {
    if (strcmp(i-name, argv[3]) ≡ 0) fonte = i;
    if (strcmp(i-name, argv[4]) ≡ 0) sorvedouro = i;
  }
  if (fonte ≡ Λ ∨ sorvedouro ≡ Λ) {
    fprintf(stderr, "ERRO: Vértices inválidos.\n");
    exit(-4);
  }

```

Este código é usado no bloco 28.

31. 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 31 > ≡
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(-5);
        }
    }
}

```

Este código é usado no bloco 28.

32. A opção `-p` indica que o fluxo bloqueador deve ser obtido em uma rede acíclica produzida por uma busca em profundidade. Por padrão, a rede acíclica é construída através de uma busca em largura.

```

< Verifica tipo de busca 32 > ≡
largura = TRUE;
if (argc ≡ 6) {
    if (strcmp("-p", argv[5]) ≡ 0) largura = FALSE;
    else {
        fprintf(stderr, "ERRO: □Opç~ao□inválida:□%s.\n", argv[5]);
        exit(-6);
    }
}

```

Este código é usado no bloco 28.

33. Impressão do fluxo de intensidade máxima

Após a execução do algoritmo, imprime-se o fluxo máximo encontrado e sua intensidade.

```

< Imprime fluxo máximo 33 > ≡
for (max = 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));
            if (i ≡ sorvedouro) max -= flx(a);
            if (a→tip ≡ sorvedouro) max += flx(a);
        }
    }
}
fprintf(saida, "Intensidade:□%d\n", max);
fprintf(stdout, "Intensidade□do□fluxo□máximo:□%d\n", max);

```

Este código é usado no bloco 26.

34. Impressão do separador de capacidade mínima

O separador de capacidade mínima contém os vértices da rede original acessíveis a partir da fonte através de caminhos alternantes, ou seja, os vértices examinados durante a última busca por um fluxo bloqueador na rede residual. Tais vértices são identificados por possuírem o campo *dist* definido com um valor não-negativo.

```
< Imprime separador de capacidade mínima 34 > ≡
fprintf(saida, "\nSeparador:\n");
for (min = 0, i = g-vertices; i < g-vertices + g-n; i++) {
    if (i-dist ≥ 0) {
        fprintf(saida, "\t%s\n", i-name);
        for (a = i-arcs; a; a = a-next) {
            if (flx(a) ≥ 0 ∧ a-tip-dist ≡ -1) min += a-cap;
        }
    }
}
fprintf(saida, "Capacidade: %d\n", min);
fprintf(stdout, "Capacidade do separador mínimo: %d\n", min);
fclose(saida);
```

Este código é usado no bloco 26.

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

```
< Variáveis secundárias da função principal 35 > ≡
Vertex * i;
Arc * a;
int min, max;
FILE *saida;
```

Este código é usado no bloco 26.

36. Estrutura geral

Para concluir o programa, basta definir a estrutura geral.

```
< Bibliotecas necessárias 38 >  
< Variáveis globais 37 >  
< Fila circular de vértices 25 >  
< Função min 21 >  
< Função arco_original 10 >  
< Função get_capacidade_residual 15 >  
< Função envia_fluxo_bloqueador 19 >  
< Função busca_em_profundidade 14 >  
< Algoritmo dos fluxos bloqueadores de aumento 6 >  
< Função principal 26 >
```

37. Os vetores *flx* e *flx_bloq* são variáveis globais e precisam ser declarados.

```
< Variáveis globais 37 > ≡  
    long *flx, *flx_bloq;
```

Este código é usado no bloco 36.

38. Bibliotecas

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

```
< Bibliotecas necessárias 38 > ≡  
#include <stdio.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <gb_graph.h>  
#include <gb_save.h>
```

Este código é usado no bloco 36.

39. Macros

Definimos aqui todas as macros utilizadas no programa.

```
#define boolean int  
#define FALSE 0  
#define TRUE 1  
#define dist u.I  
#define vertice v.V  
#define cap len  
#define pos a.I  
#define irmao b.A  
#define inicio irmao->tip  
#define flx_bloq(a) flx_bloq[a-pos]  
#define flx(a) flx[a-pos]
```

Índice Remissivo

a: 21.
Arc: 10, 14, 15, 22, 24, 35.
arco_original: 9, 10, 15, 23, 33.
arcs: 7, 8, 9, 12, 14, 17, 18, 19, 23, 31, 33, 34.
argc: 26, 28, 32.
argv: 26, 28, 29, 30, 32.
b: 21.
boolean: 6, 25, 26, 39.
busca_em_profundidade: 13, 14.
cap: 9, 15, 31, 34, 39.
dist: 6, 11, 12, 13, 14, 17, 19, 34, 39.
envia_fluxo_bloqueador: 16, 19, 20, 22.
exit: 25, 27, 28, 29, 30, 31, 32.
FALSE: 25, 32, 39.
fclose: 34.
fila_vazia: 12, 25.
fim: 25.
fluxo: 19.
fluxos_bloqueadores_aumento: 6, 11, 26.
flx: 5, 7, 8, 9, 10, 15, 23, 27, 33, 34, 37, 39.
flx_blog: 7, 16, 18, 20, 23, 27, 37, 39.
flx_caminho: 20, 22.
flx_sobra: 19, 20, 22.
fonte: 26, 30.
fopen: 29.
fprintf: 6, 25, 27, 28, 29, 30, 31, 32, 33, 34.
gb_new_arc: 9.
get_cap_residual: 12, 14, 15, 17, 19, 20.
Graph: 6, 25, 26.
ij: 11.
indice: 7, 9, 24.
ini: 25.
inicializa_fila: 12, 25.
inicio: 39.
insere_fila: 12, 25.
irmao: 9, 15, 23, 39.
iteracoes: 6, 24.
largura: 6, 26, 32.
len: 28, 39.
main: 26.
malloc: 27.
max: 25, 33, 35.
max_fluxo: 16, 17, 24.
min: 20, 21, 34, 35.
name: 30, 33, 34.
next: 7, 8, 9, 12, 14, 17, 18, 19, 23, 31, 33, 34.
pos: 7, 9, 39.
primeiro_fila: 25.
remove_fila: 12, 25.
restore_graph: 29.
saida: 29, 33, 34, 35.
sorvedouro: 26, 30, 33.
st: 5, 6.
stderr: 25, 27, 28, 29, 30, 31, 32.
stdout: 6, 33, 34.
strcmp: 30, 32.
tip: 9, 12, 14, 17, 19, 33, 34, 39.
TRUE: 25, 32, 39.
Vertex: 6, 14, 19, 22, 24, 25, 26, 35.
vertice: 25, 39.
vertices: 7, 8, 9, 12, 13, 18, 23, 25, 30, 31, 33, 34.
zero: 25.

Lista de Refinamentos

- ⟨ Algoritmo dos fluxos bloqueadores de aumento 6 ⟩ Usado no bloco 36.
- ⟨ Aloca vetores *flx* e *flx_bloq* 27 ⟩ Usado no bloco 26.
- ⟨ Aumenta fluxo através do fluxo bloqueador 23 ⟩ Usado no bloco 6.
- ⟨ Bibliotecas necessárias 38 ⟩ Usado no bloco 36.
- ⟨ Busca repassar fluxo ao vizinho 20 ⟩ Usado no bloco 19.
- ⟨ Constrói arcos irmãos 9 ⟩ Usado no bloco 6.
- ⟨ Define associação dos arcos aos vetores de fluxos 7 ⟩ Usado no bloco 6.
- ⟨ Encontra fluxo bloqueador na rede acíclica 16 ⟩ Usado no bloco 6.
- ⟨ Fila circular de vértices 25 ⟩ Usado no bloco 36.
- ⟨ Função principal 26 ⟩ Usado no bloco 36.
- ⟨ Função *arco_original* 10 ⟩ Usado no bloco 36.
- ⟨ Função *busca_em_profundidade* 14 ⟩ Usado no bloco 36.
- ⟨ Função *envia_fluxo_bloqueador* 19 ⟩ Usado no bloco 36.
- ⟨ Função *get_capacidade_residual* 15 ⟩ Usado no bloco 36.
- ⟨ Função *min* 21 ⟩ Usado no bloco 36.
- ⟨ Gera limitante fluxo bloqueador 17 ⟩ Usado no bloco 16.
- ⟨ Imprime fluxo máximo 33 ⟩ Usado no bloco 26.
- ⟨ Imprime separador de capacidade mínima 34 ⟩ Usado no bloco 26.
- ⟨ Inicializa fluxo bloqueador 18 ⟩ Usado no bloco 16.
- ⟨ Obtém fluxo inicial 8 ⟩ Usado no bloco 6.
- ⟨ Obtém rede acíclica com busca em profundidade 13 ⟩ Usado no bloco 6.
- ⟨ Obtém rede residual de caminhos mínimos 12 ⟩ Usado no bloco 6.
- ⟨ Variáveis da função *envia_fluxo_bloqueador* 22 ⟩ Usado no bloco 19.
- ⟨ Variáveis da função *fluxos_bloqueadores_aumento* 24 ⟩ Usado no bloco 6.
- ⟨ Variáveis secundárias da função principal 35 ⟩ Usado no bloco 26.
- ⟨ Variáveis globais 37 ⟩ Usado no bloco 36.
- ⟨ Verifica consistência dos parâmetros 28 ⟩ Usado no bloco 26.
- ⟨ Verifica existência dos vértices 30 ⟩ Usado no bloco 28.
- ⟨ Verifica sinal das capacidades 31 ⟩ Usado no bloco 28.
- ⟨ Verifica tipo de busca 32 ⟩ Usado no bloco 28.
- ⟨ Verifica validade dos arquivos 29 ⟩ Usado no bloco 28.