Universidade Estadual do Centro Oeste
UNICENTRO-PR

O Problema do fluxo de custo mínimo

Algoritmo Big-M
Implementação e comentários

Trabalho apresentado ao professor Fábio Hernandes, da disciplina de Tópicos especiais em Ciência da Computação, pelo aluno **Diego Gadens dos Santos**.

Guarapuava, junho de 2008

Problema do fluxo de custo mínimo

Suponha que uma empresa administra um conjunto de estabelecimentos formado por fábricas e depósitos de um determinado produto. Cada fábrica produz uma certa quantidade do produto e cada depósito deve armazenar uma certa quantidade. Os estabelecimentos são interconectados por uma rede de vias de transporte, como por exemplo estradas, sobre a qual são definidos custos de transporte e limites para a quantidade transportada. O desejo da empresa é estabelecer uma distribuição do produto sobre essa rede de forma a satisfazer as ofertas e demandas dos estabelecimentos, respeitar os limites das estradas e minimizar o custo total de transporte. Este é um exemplo de instância do problema conhecido como problema do fluxo de custo mínimo. [IME08].

Associando a cada aresta da rede um valor c(a) que corresponde à capacidade da aresta, podemos definir uma função x(a) que correspondente ao fluxo da aresta, ou seja quantidade de unidades correntes por esta aresta. Esta quantidade deve ser menor ou igual à capacidade do arco. A solução ideal para este problema é passar a maior quantidade de fluxo possível pela rede, respeitando as capacidades dos arcos, e ainda, com o menor custo possível.

O objetivo deste trabalho é apresentar o algoritmo Big-M, que é um algoritmo que resolve este problema. O Big-M é divido em duas fases, a primeira fase é responsável por encontrar uma solução factível para a rede fornecida pelo usuário, atendendo as suas restrições de capacidade. Após esta primeira parte, o algoritmo inicia sua segunda fase, conhecida como simplex especializado [WIKI08], a qual tem a função de encontrar uma solução ótima para o problema, com base na solução inicial encontrada pela primeira fase do Big-M.

```java
import java.util.LinkedList;

public class Fifo
{
    /*Esta classe dispensa maiores comentários, serve apenas para a
implementação de uma fila do tipo FIFO, utilizando a classe LinkedList do
Java.*/

    public LinkedList<Integer> fila;

    public Fifo()
    {
        fila = new LinkedList<Integer>();
    }

    public void enfileira(int x)
    {
        fila.addLast(x);
    }

    public Object desenfileira()
    {
        if(fila.isEmpty() != true)
          return fila.removeFirst();
        else
          return "A fila está vazia";
    }

    public int pegaPrimeiro()
    {
          return ((int) fila.getFirst());
    }

    public boolean isEmpty()
    {
        return fila.isEmpty();
    }

    public boolean contém(int x)
    {
        return (fila.contains(x));
    }

    public void clear()
    {
        fila.clear();
    }
}
```

```java
public class Nó
{
    /*Esta classe dispensa maiores comentários, serve apenas para representar os nós presentes no grafo. Contém também os métodos “get” e “set” para estas variáveis*/

    private int id_no;
    private int demanda;
    private int potencial;
    private boolean potencial_calculado = false;

    public Nó(int id_no, int demanda)
    {
        this.id_no = id_no;
        this.demanda = demanda;
        potencial = 0;
    }

    public int getId()
    {
        return id_no;
    }

    public int getDemanda()
    {
        return demanda;
    }

    public int getPotencial()
    {
        return potencial;
    }

    public void setPotencial(int x)
    {
        potencial = x;
    }

    public boolean potencialCalculado()
    {
        return potencial_calculado;
    }

    public void setPotencialCalculado(boolean x)
    {
        potencial_calculado = x;
    }
}
```

```
public class Arco
{
    /*Esta classe dispensa maiores comentários, serve apenas para
representar os arcos do grafo. Recebe os parâmetros do programa principal,
e com isso cria os arcos. Conta também com os métodos “get” e “set” para
acessar e alterar as variáveis dos arcos.*/

    private int nó_origem, nó_destino, lim_sup, custo, custo_relativo,
fluxo;
    private static final int lim_inf = 0;
    private boolean básico, artificial, estáNoCiclo;
    private int sentido;
    private boolean visitado;

    public Arco(int nó_origem, int nó_destino, int lim_sup, int custo,
boolean básico, boolean artificial )
    {
        this.nó_origem = nó_origem;
        this.nó_destino = nó_destino;
        this.lim_sup = lim_sup;
        this.custo = custo;
        custo_relativo = 0;
        fluxo = 0;
        this.básico = básico;
        this.artificial = artificial;
    }

    public int getOrigem()
    {
        return nó_origem;
    }

    public int getDestino()
    {
        return nó_destino;
    }

    public int getLimInf()
    {
        return lim_inf;
    }

    public int getLimSup()
    {
        return lim_sup;
    }

    public int getCusto()
    {
        return custo;
    }

    public int getCustoRelativo()
    {
        return custo_relativo;
    }
```

```
public void setCustoRelativo(int x)
{
    custo_relativo = x;
}

public int getFluxo()
{
    return fluxo;
}

public void setFluxo(int x)
{
    fluxo = x;
}

public boolean ehBásico()
{
    if(básico == true)
        return true;
    else
        return false;
}

public void setBásico(boolean x)
{
    básico = x;
}

public boolean ehArtificial()
{
    if(artificial == true)
        return true;
    else
        return false;
}

public int getSentido()
{
    return sentido;
}

public boolean estáNoCiclo()
{
    return estáNoCiclo;
}

public void setEstáNoCiclo(boolean x)
{
    estáNoCiclo = x;
}

public void setSentido(int x)
{
    sentido = x;
}
```

```
    public boolean visitado()
    {
        return visitado;
    }

    public void setVisitado(boolean x)
    {
        visitado = x;
    }


    public String toString()
    {
        return ("(" + nó_origem + "," + nó_destino + ") ");
    }

    public void imprimeArco()
    {
        System.out.println("\n("+nó_origem+","+nó_destino+") ->
"+"["+lim_inf+","+lim_sup+","+custo+"]\nFluxo atual: "+fluxo);
    }

}
```

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.StringTokenizer;

public class BigM
{
    /*Esta é a classe principal deste trabalho, ela é a responsável pelo controle de instância das demais classes (Arcos e Nós), bem como do tratamento destes objetos, a fim de encontrar a solução ótima para os problemas*/

    //Conjunto dos Nós
    private ArrayList<Nó> nós = new ArrayList<Nó>();
    //Conjunto dos Arcos do grafo
    private ArrayList<Arco> grafo_original = new ArrayList<Arco>();
    //Conjunto de arcos para identificação do ciclo
    private ArrayList<Arco> ciclo = new ArrayList<Arco>();
    private ArrayList<Arco> saiDoCiclo = new ArrayList<Arco>();
    private Arco candidatoSair = new Arco(0,0,0,0,false,false);
    private Arco candidatoEntrar = new Arco(0,0,0,0,false,false);
    private int num_nós;
    private int num_arestas;
    private int nó_artificial;
    private Scanner entrada = new Scanner(System.in);
    public static final int MGRANDE = 10000;
    public static final int INFINITO = 10000;
    private Fifo fila = new Fifo();
    int delta = 10000;


    public BigM(int num_nós, int num_arestas)
    {
        this.num_nós = num_nós;
        this.num_arestas = num_arestas;
        nó_artificial = (num_nós + 1);


/*Este trecho de código é a parte principal do algoritmo, é onde são feitas todas as chamadas de funções para os diversos métodos que compõem o Big-M*/

        //Coleta os dados do grafo, demandas, custos e capacidades dos arcos e nós.
        lê_entrada();
        //Cria os arcos artificiais para dar início à execução do algoritmo.
        cria_arcos_articiais();

        //Variável para controlar o número de iterações
        int i = 1;

        //Estrutura de repetição, que faz com que sejam executados os passos básicos do algoritmo, enquanto existir mais do que um arco artificial no grafo.
        while (contaArcosArtificiais()>1)
        {
```

```
            System.out.println("\n"+i+"ª ITERAÇÃO:");

             //Delay simples para visualização da iteração corrente. Para
o programa por 1000 milisegundos, ou seja 1 segundo.
             try
             {
                 Thread.sleep(1000);
             }
             catch (Exception e)
             {
             }

             //Calculo dos potenciais
             calculaPotenciais(nó_artificial);
             //Calculo dos custos-relativos
             calculaCustosRelativos();
             //Escolhe o candidato a entrar na base
             candidatoAEntrar();
             //Encontra o ciclo formado pelos arcos da base + o arco que
foi escolhido para entrar na base.
             encontraCiclo(candidatoEntrar);
             //Verifica qual arco deste ciclo está bloqueando o fluxo, e
marca-o para remoção
             calculaBloqueio();
             //Atualiza os fluxos, e remove o arco candidato a sair SE ESTE
FOR ARTIFICIAL, ou marca o como não-básico SE ESTE FOR NÃO-ARTIFICIAL.
             redefineFluxos();
             //Reseta as variáveis de controle do algoritmo
             resetaVariáveis();
             //Incrementa a iteração
             i++;
        }

        //Quando existir somente um arco artificial na base, verifica se
esta é uma solução factível, se sim, passa-se para a fase 2, senão o
problema não tem solução.
        soluçãoFactível();

        //fase2();
    }

    public void lê_entrada()
    {
        //Método utilizado para realizar a leitura dos dados do grafo
(Arcos e nós), como origem, destino, custo, capacidades e demandas.
        int id, demanda;

        System.out.println("Insira os dados dos nós separados por um
espaço");

        for(int i=0; i<num_nós; i++)
        {
            System.out.print("Insira o id do nó, seguido de sua demanda
");
            String linha_digitada = entrada.nextLine();
```

```
            StringTokenizer tokenizer = new
StringTokenizer(linha_digitada, " ");
              id = Integer.parseInt(tokenizer.nextToken());
              demanda = Integer.parseInt(tokenizer.nextToken());

            nós.add(new Nó(id, demanda));
        }
        //Adiciona cada novo nó inserido pelo usuário no conjunto dos nós
        nós.add(new Nó(nó_artificial, 0));
        nós.get(nó_artificial-1).setPotencialCalculado(true);

        //Agora, coletam-se as arestas, da mesma maneira que os nós. Para
cada entrada, instancia-se uma nova aresta e a adiciona no conjunto das
arestas.
        for(int i=0; i<num_arestas; i++)
        {
            int nó_origem, nó_destino, lim_sup, custo, básico,
artificial;
            System.out.print("Insira os dados: origem, destino, lim_sup,
custo");
            String linha_digitada = entrada.nextLine();

            StringTokenizer tokenizer = new
StringTokenizer(linha_digitada, " ");
              nó_origem = Integer.parseInt(tokenizer.nextToken());
              nó_destino = Integer.parseInt(tokenizer.nextToken());
              lim_sup = Integer.parseInt(tokenizer.nextToken());
              custo = Integer.parseInt(tokenizer.nextToken());
              básico = 0;
              artificial = 0;

            grafo_original.add(new Arco(nó_origem, nó_destino, lim_sup,
custo, ((básico==1)? true : false), ((artificial==1)? true : false)));
        }
    }


    //Este método tem a função de criar os arcos artificiais. Este é o
primeiro passo executado neste algoritmo. Os sentidos dos arcos são
definidos de acordo com a demanda dos nós, sendo criado um arco
artificial para cada nó, ligando este nó ao nó artificial. Todos estes
arcos artificiais formam a base inicial.
    private void cria_arcos_articiais()
    {
        Arco aux;

            for(Nó x : nós)
            {
                if((x.getId() < nó_artificial) && (x.getDemanda()<0))
                {
                    aux = (new Arco(nó_artificial, x.getId(), INFINITO,
MGRANDE , true, true));
                    aux.setFluxo(x.getDemanda() * -1);
                    grafo_original.add(aux);
                }
                else
                if((x.getId()<nó_artificial) && (x.getDemanda()>=0))
```

```
                {
                    aux = (new Arco(x.getId(), nó_artificial, INFINITO,
MGRANDE , true, true));
                    aux.setFluxo(x.getDemanda());
                    grafo_original.add(aux);
                }
            }

            for(Arco x : grafo_original)
              x.imprimeArco();
    }


    //Este método tem a função de calcular os potenciais de todos os nós
do grafo. Ele percorre o conjunto dos nós por amplitude, à partir do nó
artificial. Para alcançar todos os nós é criada uma fila, partindo do nó
artificial, visitando seus filhos, e os filhos dos filhos, até que todos
os nós passem pela fila, e, consequentemente tenham seus potenciais
calculados.
    private void calculaPotenciais(int nó_raiz)
    {
        int raiz = nó_raiz;
        Nó i, j;
        int wi, wj;
        fila.enfileira(raiz);
        do
        {
              for(Arco x : grafo_original)
              {
                 i = nós.get(x.getOrigem()-1);
                 j = nós.get(x.getDestino()-1);
                 wi = (0 + i.getPotencial());
                 wj = (0 + j.getPotencial());

                 if ((x.ehBásico()) && (x.getDestino() ==
fila.pegaPrimeiro()) && (i.potencialCalculado()==false))
                 {
                    i.setPotencial( wj + x.getCusto());
                    i.setPotencialCalculado(true);
                    fila.enfileira(i.getId());
                 }
                 else
                 if ((x.ehBásico()) && (x.getOrigem() ==
fila.pegaPrimeiro()) && (j.potencialCalculado()==false))
                 {
                    j.setPotencialCalculado(true);
                    j.setPotencial(wi + (-1 * x.getCusto()));
                    fila.enfileira(j.getId());
                 }
              }

              fila.desenfileira();
              if(fila.isEmpty()==false)
                raiz = fila.pegaPrimeiro();
        }
        while(fila.isEmpty()==false);
```

```
        for(Nó x : nós)
        {
            System.out.print("Potencial de " + x.getId() + " -> ");
            System.out.println(x.getPotencial());
            x.setPotencialCalculado(false);
        }
    }

    //Este é o método para calcular os custos relativos, para todos arcos não-básicos do grafo esse cálculo é nescessário, portanto, percorre-se o conjuntos de arestas, efetuando o cálculo para todos os arcos que não estiverem na base. Para o cálculo dos custos relativos é utilizado os valores dos potenciais, que neste ponto do código também já foram calculados.
    private void calculaCustosRelativos()
    {
        int wi, wj, cij;
        for(Arco x : grafo_original)
        {
            if (!x.ehBásico())
            {
                wi = nós.get(x.getOrigem()-1).getPotencial();
                wj = nós.get(x.getDestino()-1).getPotencial();
                cij = x.getCusto();
                x.setCustoRelativo(wi - wj - cij);
            }
        }

        System.out.println("\n");
        for(Arco x : grafo_original)
        {
            if(!x.ehBásico())
              System.out.println("Custo Relativo de ("+x.getOrigem() +", "+ x.getDestino() + "): " + x.getCustoRelativo() );
        }
    }


    //Método responsável por escolher um candidato a entrar na base. A cada iteração, entra na base aquele arco que possuir o maior custo relativo positivo e além disso que não esteja com seu fluxo no limitante superior.
    private void candidatoAEntrar()
    {
        Arco candidato = new Arco(0,0,0,0,false,false);;
        int maiorCusto = 0;

        for(Arco x : grafo_original)
        {
            if ((!x.ehBásico()) && (x.getCustoRelativo() > maiorCusto) && (x.getFluxo() < x.getLimSup()))
            {
                candidato = x;
                maiorCusto = candidato.getCustoRelativo();
            }
        }
```

```
        System.out.println("\nCandidato a entrar na base:");
        candidato.imprimeArco();
        System.out.println("\n");
        candidatoEntrar = candidato;

    }

    //Método responsável por identificar o ciclo formado entre o arco candidato a entrar na base, e os arcos que já estão na base. A cada iteração, um ciclo é formado, com o arco que vai entrar na base. Logo após este ciclo é desfeito, retirando-se o arco que bloquear o fluxo. Para encontrar o ciclo este método adiciona todos os arcos básicos no conjuto (ArrayList) chamado ciclo, e entõ inicia um processo de remoção de folhas, baseando-se na adjacência dos nós. Enquanto existirem folhas para serem retiradas, elas são retiradas do conjunto do ciclo. Após todas as folhas terem sido removidas, sobrarão no ArrayList apenas os arcos que fazem parte do ciclo.
    private void encontraCiclo(Arco arco)
    {
        ciclo.add(arco);
        candidatoEntrar.setBásico(true);

        for(Arco x : grafo_original)
        {
            if((x.ehBásico())&&(x != arco))
            {
                ciclo.add(x);
            }
        }

        while(temFolha())
        {
            removeFolhas();
        }

        System.out.println("Ciclo:");

        defineSentido(arco);
     }

     //Este método utiliza o conjunto já formado do ciclo, e tem a função de definir o sentido destes arcos em relação ao arco candidato a entrar na base. Esta verificação é muito importante, pois é com base na direção do arco que será atualizado o seu fluxo. Este método percorre o conjunto dos ciclos e faz comparações entre nós origens e destinos para definir os sentidos dos arcos.
     private void defineSentido(Arco sentido)
     {
        sentido.setSentido(1);
        int origem = sentido.getOrigem();
        int destino = sentido.getDestino();

        do
        {
            for(Arco x : ciclo)
            {
              if( x != sentido)
```

```
                {
                  if(x.ehBásico() && (origem != nó_artificial))
                  {
                      if((x.getOrigem() == origem))
                      {
                          x.setSentido(-1);
                          origem = x.getDestino();
                      }
                      else
                      if((x.getDestino() == origem))
                      {
                          x.setSentido(1);
                          origem = x.getOrigem();
                      }
                  }
                }
            }
        }while (origem != nó_artificial);

        do
        {
            for(Arco x : ciclo)
            {
              if(x != sentido)
              {
                if((x.ehBásico()) && (destino != nó_artificial))
                {
                    if((x.getOrigem() == destino) && (!x.estáNoCiclo()))
                    {
                        x.setSentido(1);
                        destino = x.getDestino();
                    }
                    else
                    if((x.getDestino() == destino) && (!x.estáNoCiclo()))
                    {
                        x.setSentido(-1);
                        destino = x.getOrigem();
                    }
                }
              }
            }
        }while (destino != nó_artificial);
    }


    //Função booleana para retornar se o conjunto do ciclo ainda contém
folhas, enquanto esta função tiver retorno verdadeiro, é porque existe
alguma folha no ciclo que precisa ser removida. Portanto a cada chamada,
esta função marca um arco que tenha nó folha, para ser removido do ciclo.
    private boolean temFolha()
    {
        for(Arco x : ciclo)
        {
            if(((getAdjacência(x.getOrigem())<2) ||
(getAdjacência(x.getDestino())<2)) && (x != candidatoEntrar))
            {
                saiDoCiclo.add(x);
```

```
                return true;
            }
        }
        return false;
    }

    //Se existir um nó folha, então o arco que contém este nó é removido do ciclo.
    private void removeFolhas()
    {
        Arco aRemover = new Arco(0,0,0,0,false, false);
        for(Arco x : saiDoCiclo)
           for(Arco y : ciclo)
           if(x == y)
               aRemover = y;

        ciclo.remove(aRemover);
    }


    //Método que retorna a adjacência de um determinado nó, se um nó tiver adjacencia 1, isto quer dizer que ele precisa ser removido do ciclo, pois todos os arcos do ciclo, terão pelo menos um grau 2.
    private int getAdjacência(int adj)
    {
      int total = 0;
      for(Arco x : ciclo)
      {
          if((x.getOrigem() == adj) || (x.getDestino() == adj))
          total++;
      }
      return total;
    }

   //Método utilizado para escolher um nó que será removido do grafo, o arco que estiver no ciclo e bloquear o fluxo (delta mínimo, extraído com base nas fórmulas de calculo de bloqueio) será removido (Se for artificial) ou será setado como não básico (Se for não-artificial). Esta função também dá prioridade de remoção a um nó artificial, caso haja empate de bloqueio entre um artificial e um não-artificial.
    private void calculaBloqueio()
    {
        int bloqueio;

        for(Arco x : ciclo)
        {
                if(x.getSentido() == 1)
                {

                    bloqueio = (x.getLimSup() - x.getFluxo());
                    if(bloqueio < delta)
                    {
                        delta = bloqueio;
                        candidatoSair = x;
                        x.imprimeArco();
                    }
                    else
```

```
                    if(bloqueio == delta)
                    {
                        if(x.ehArtificial())
                        {
                            candidatoSair = x;
                            x.imprimeArco();
                        }
                    }
                }
                else
                if(x.getSentido() == -1)
                {
                    bloqueio = (x.getFluxo() - x.getLimInf());
                    if(bloqueio < delta)
                    {
                        delta = bloqueio;
                        candidatoSair = x;
                        x.imprimeArco();
                    }
                    else
                    if(bloqueio == delta)
                    {
                        if(x.ehArtificial())
                        {
                            candidatoSair = x;
                            x.imprimeArco();
                        }
                    }
                 }
       }

        System.out.println("\nDelta: "+delta);
        System.out.println("Candidato a sair da base:");
        candidatoSair.imprimeArco();
    }

    //Após ser escolhido qual arco deixara a base, é necessário atualizar
os fluxos dos outros nós que permanecerão no grafo. Estes arcos devem
aumentar ou diminuir seus fluxos, com base no valor de DELTA, e com base
em seus sentidos dentro do ciclo.
    private void redefineFluxos()
    {
        for (Arco x : ciclo)
        {
             if(x.getSentido() == 1)
             {
                 x.setFluxo(x.getFluxo() + delta);
             }
             else
             if(x.getSentido() == -1)
             {
                 x.setFluxo(x.getFluxo() - delta);
             }
        }

        System.out.println("\n\nFluxos atualizados:");
        for (Arco x : ciclo)
```

```
        {
            x.imprimeArco();
        }

        if(candidatoSair.ehArtificial())
          grafo_original.remove(candidatoSair);
        else
          candidatoSair.setBásico(false);

        System.out.println("\nSai da base o arco:");
        candidatoSair.imprimeArco();
          ciclo.remove(candidatoSair);
          System.out.println("----------------------------------------
-----------------");


    } //Todos os passos descritos até aqui serão realizados enquanto a
condição de parada não for cumprida, ou seja, enquanto existir mais do
que um arco artificial na base.


    //Reseta as variáveis utilizada ao longo da execução dos passos acima.
    private void resetaVariáveis()
    {
        for (Arco x : grafo_original)
        {
            x.setEstáNoCiclo(false);
        }

        for (Nó x : nós)
        {
            if(x.getId() != nó_artificial)
            {
                x.setPotencial(0);
                x.setPotencialCalculado(false);
            }
        }

        saiDoCiclo.clear();
        ciclo.clear();
        fila.clear();
        delta = 25000;
        candidatoSair = new Arco(0,0,0,0,false,false);

    }

    //Método para definir se o algoritmo ira parar ou não em uma dada
iteração. Este método conta quantos arcos artificiais ainda estão
presentes.
    private int contaArcosArtificiais()
    {
        int contador = 0;
            for(Arco x : grafo_original)
            {
                if(x.ehArtificial())
                contador++;
            }
```

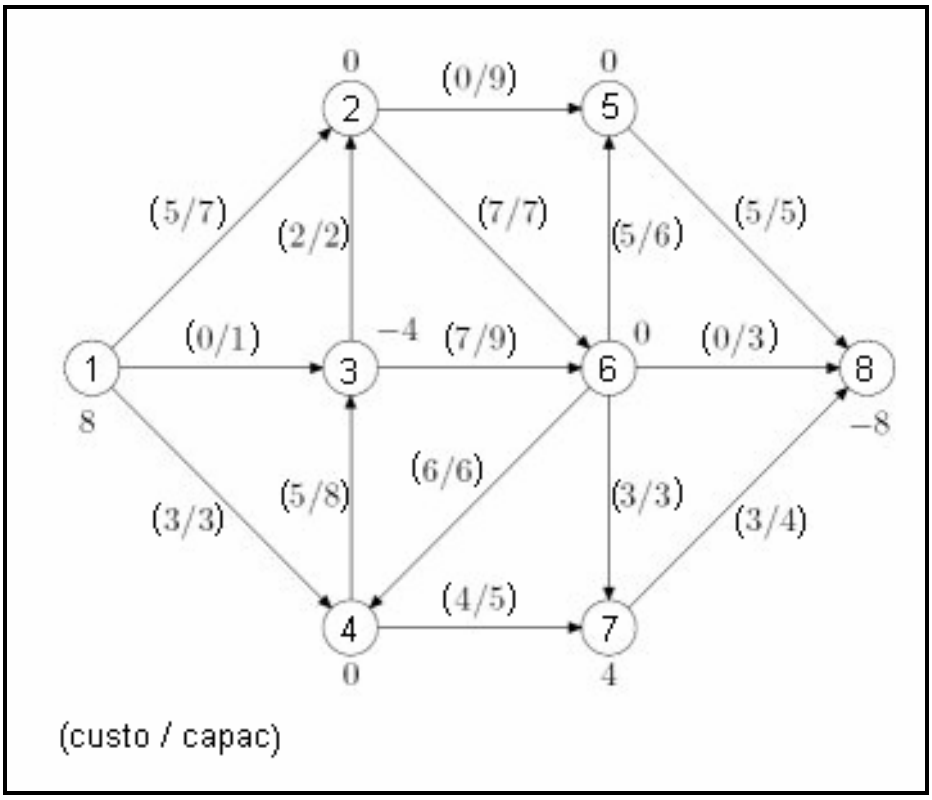
```
            return contador;
    }


    //Quando só existir um arco artificial na base, a solução factível
pode ser montada. Se este único arco tiver seu fluxo no limitante
inferior (zero, neste caso), então remove-se este arco e a solução
factível está montada. Porém se o único arco tiver seu fluxo não-nulo, o
problema não tem solução.
    private void soluçãoFactível()
    {
        Arco removeÚltimo = new Arco(0,0,0,0,false,false);
        for(Arco x : grafo_original)
        {
            if((x.getOrigem()== nó_artificial) || (x.getDestino()==
nó_artificial))
            {
                if(x.getFluxo() > 0)
                    System.out.println("PROBLEMA SEM SOLUÇÃO - FASE I");
                else
                    removeÚltimo = x;
            }
        }

        if(removeÚltimo.getOrigem() != 0)
        {
            grafo_original.remove(removeÚltimo);

            System.out.println("\nSOLUÇÃO FACTÍVEL: ");
            for(Arco x : grafo_original)
            {
                x.imprimeArco();
            }
            System.out.println("\nFIM DA FASE 1");
            System.out.println("----------------------------------------
-------------------");
        }
   }
   //Fim da primeira fase do Algoritmo


   //Se foi possível construir uma solução factível, o algoritmo passa
então para sua segunda fase. Um nó raiz é escolhido, tem seu potencial
definido como zero e inicia-se novamente os métodos da fase 1, enquanto
existir um arco com custo relativo maior do que zero, e fluxo diferente
do limitante superior. A sequência de passos é a mesma, mudando apenas a
condição de parada. Se a condição de parada for atingida, então a solução
ótima para o problema foi encontrada. Imprime-se então os resultados
finais, que são os fluxos que estão passando nos arcos, bem como a função
objetivo que é o somatório do fluxo vezes o custo destes arcos.
   public void fase2()
   {
   int raiz;

       System.out.println("INÍCIO DA FASE 2");
       System.out.println("Insira um nó raiz para iniciar a fase 2: ");
```

```
        raiz = entrada.nextInt();

        for(Nó x : nós )
        {
             if(x.getId() == raiz)
             x.setPotencial(0);
        }

       calculaPotenciais(raiz);
       calculaCustosRelativos();

       while(condiçãoDeParada()==false)
       {
         candidatoAEntrar();
         encontraCiclo(candidatoEntrar);
         calculaBloqueio();
         redefineFluxos();
         resetaVariáveis();
         calculaPotenciais(raiz);
         calculaCustosRelativos();
       }

        System.out.println("SOLUÇÃO OTIMIZADA PELA FASE II:");
        //soluçãoFactível();
        soluçãoÓtima();
        System.out.println("\nFIM DA FASE 2");
        System.out.println("-----------------------------------------------
--------------");

    }


    //Condição de parada da fase 2, verifica se existem arcos com custo
relativo maior do que zero.
    private boolean condiçãoDeParada()
    {
        for(Arco x : grafo_original)
        {
            if((!x.ehBásico()) && (x.getFluxo() < x.getLimSup()))
            {
                if(x.getCustoRelativo() > 0)
                return false;
            }
        }
        System.out.println("\nNão existem mais arcos com custo relativo
maior do que zero,\nou os arcos com custo relativo maior que zero já estão
saturados\n");
        return true;
    }

    //Imprime a solução final do algoritmo, ou seja a solução ótima
encontrada na Fase 2.
    private void soluçãoÓtima()
    {
        int somatório = 0;
        for(Arco x : grafo_original)
        {
```

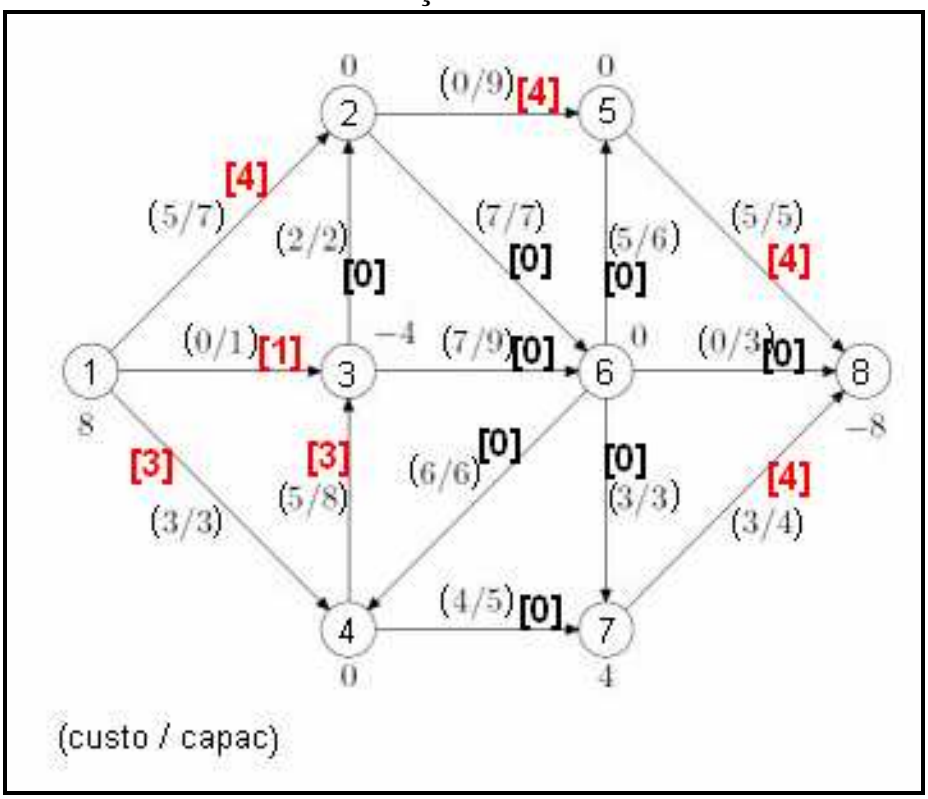
```
            if(x.getFluxo()>0)
            {

System.out.println("("+x.getOrigem()+","+x.getDestino()+") --> "+"Fluxo:
"+x.getFluxo()+" -- "+"Custo: "+x.getCusto());
                somatório += (x.getFluxo()*x.getCusto());
            }
        }

        System.out.println("\nFunção Objetivo: "+somatório);
    }

}
```

Exemplo 1

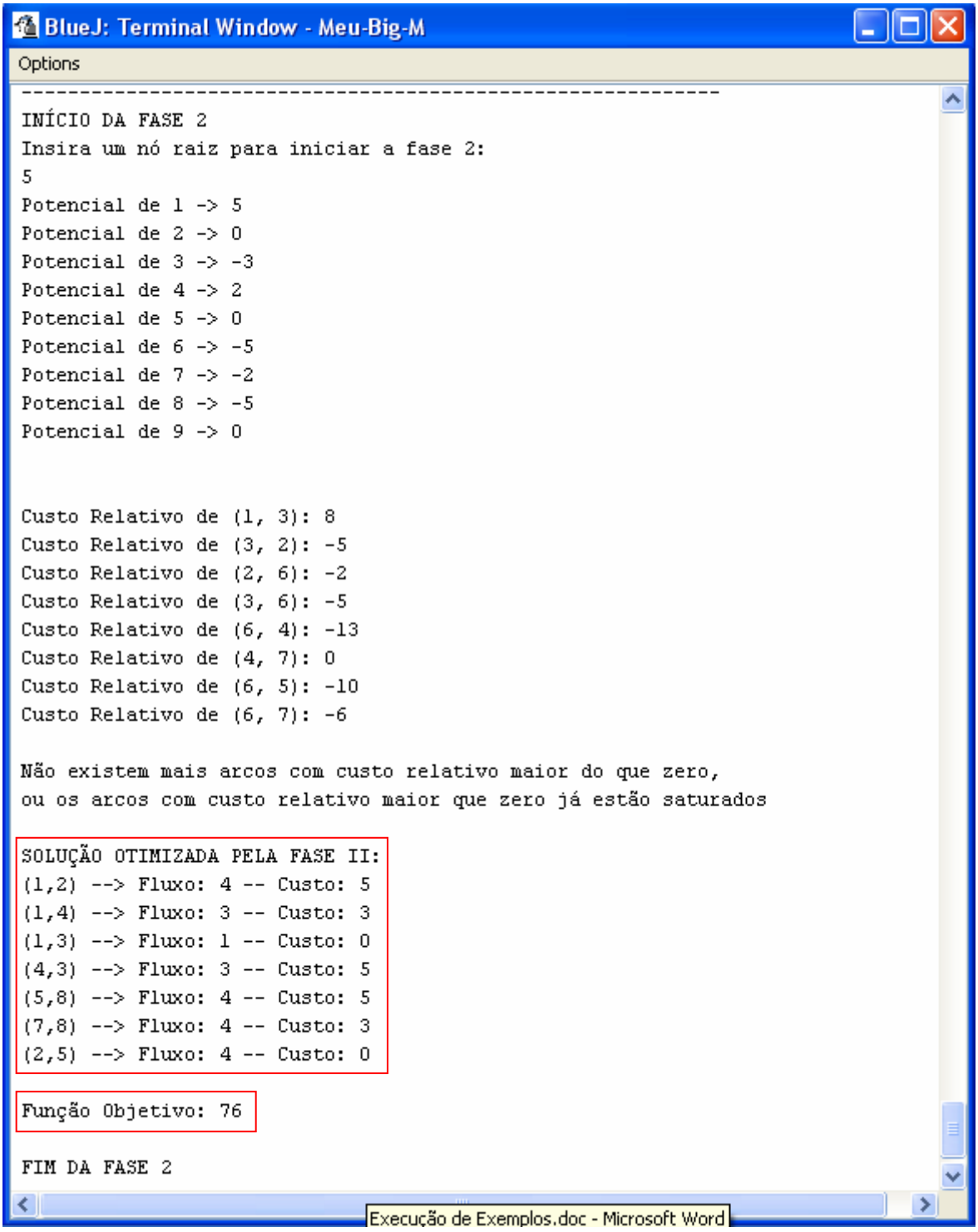
Entrada:



Solução Ótima

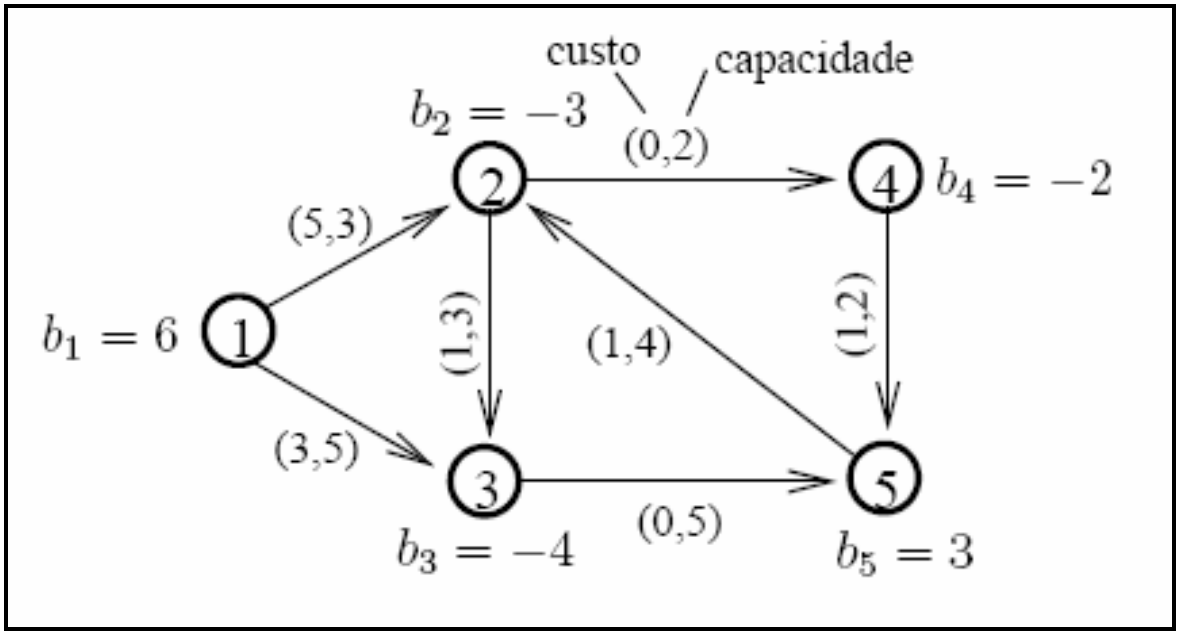


Execução do ex. 1: Solução Ótima – Fase II

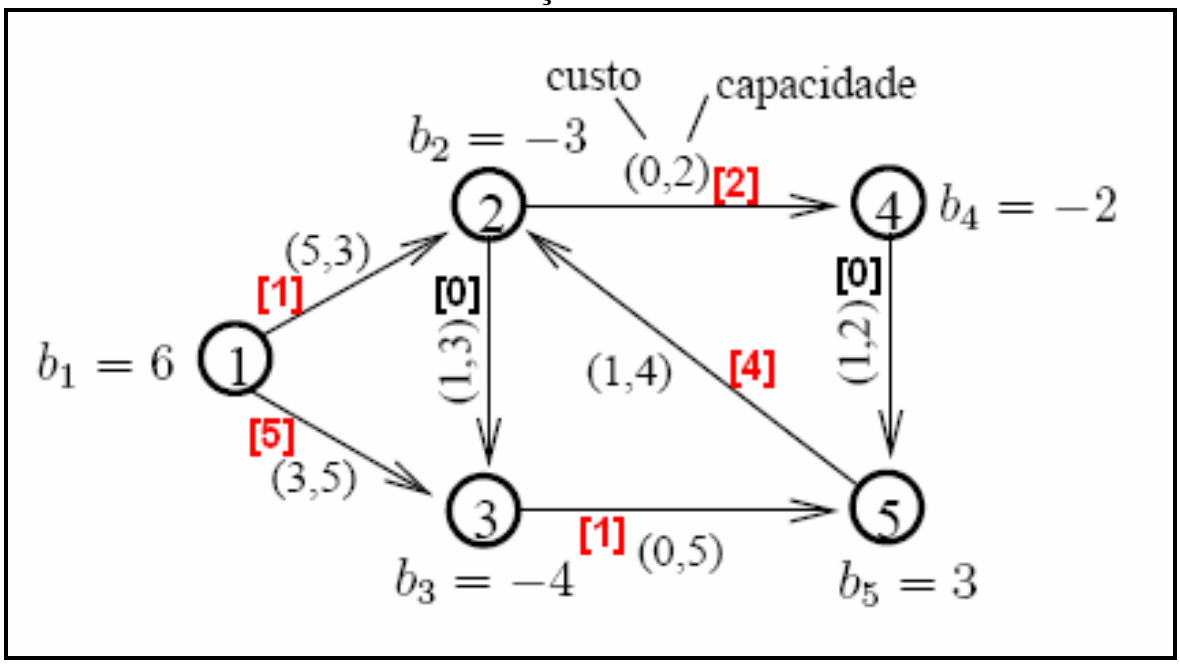


Exemplo 2

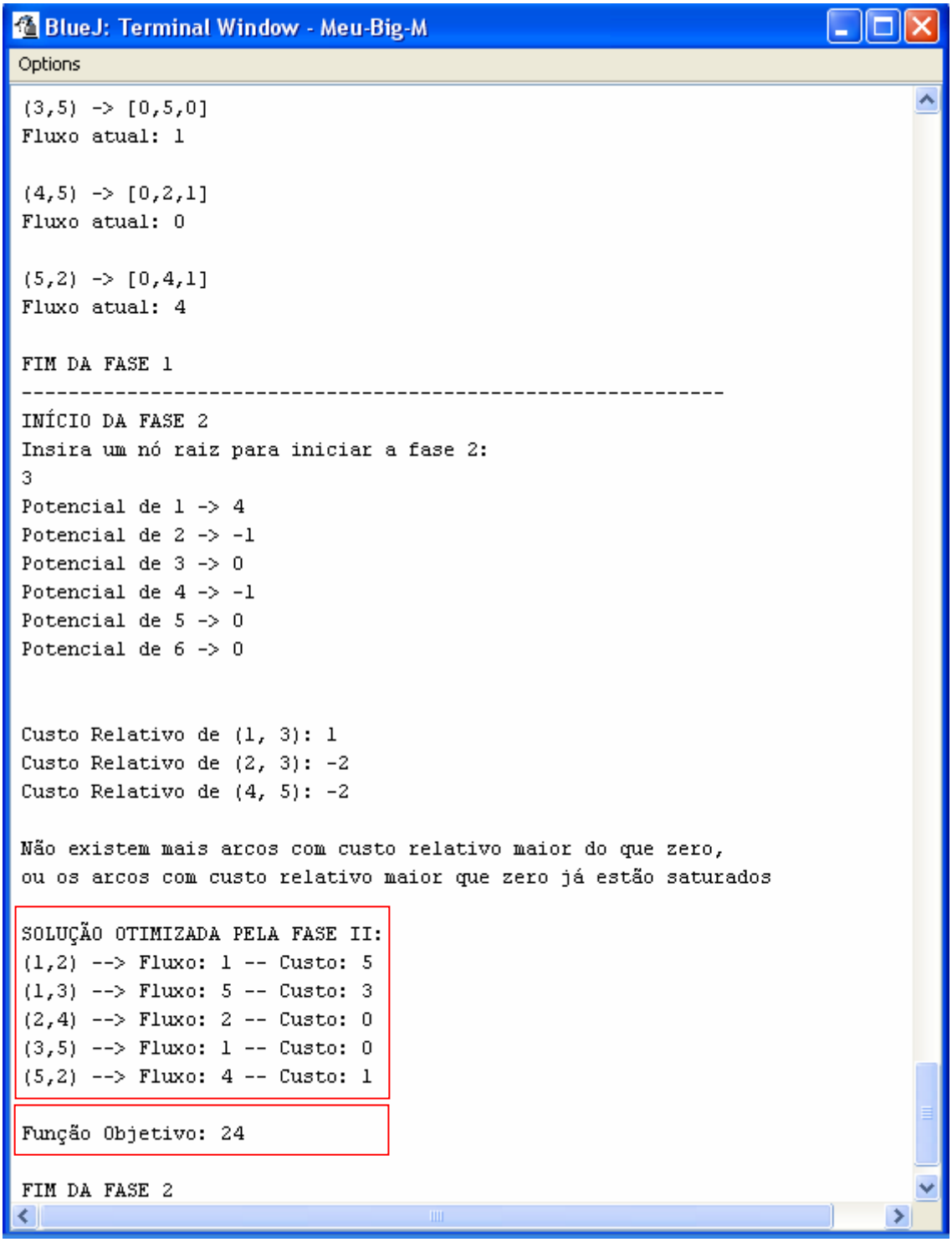
Entrada:



Solução Ótima:

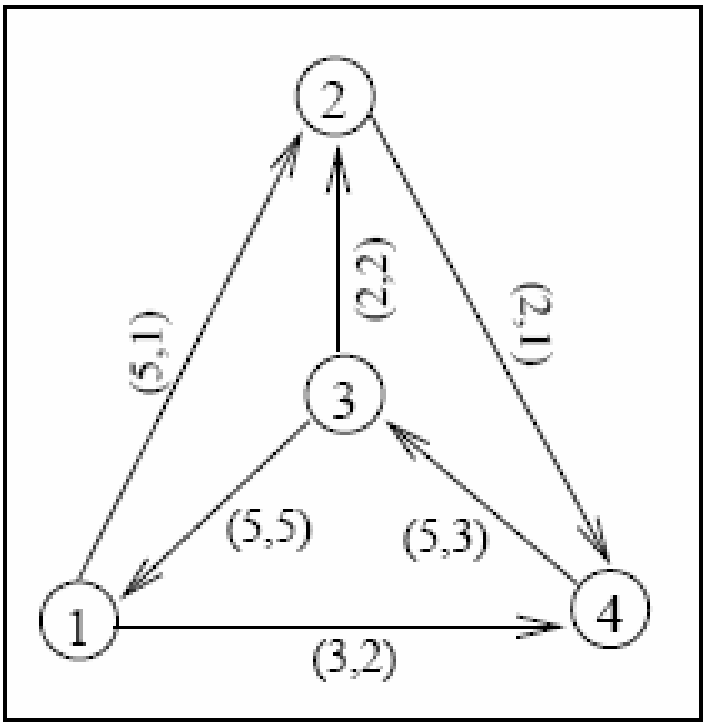


Execução do ex. 2: Solução Ótima – Fase II

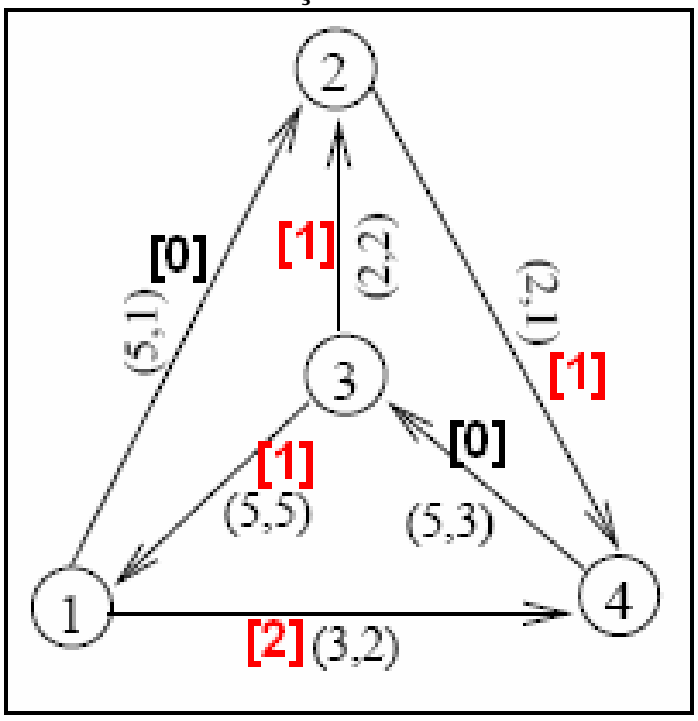


Exemplo 3

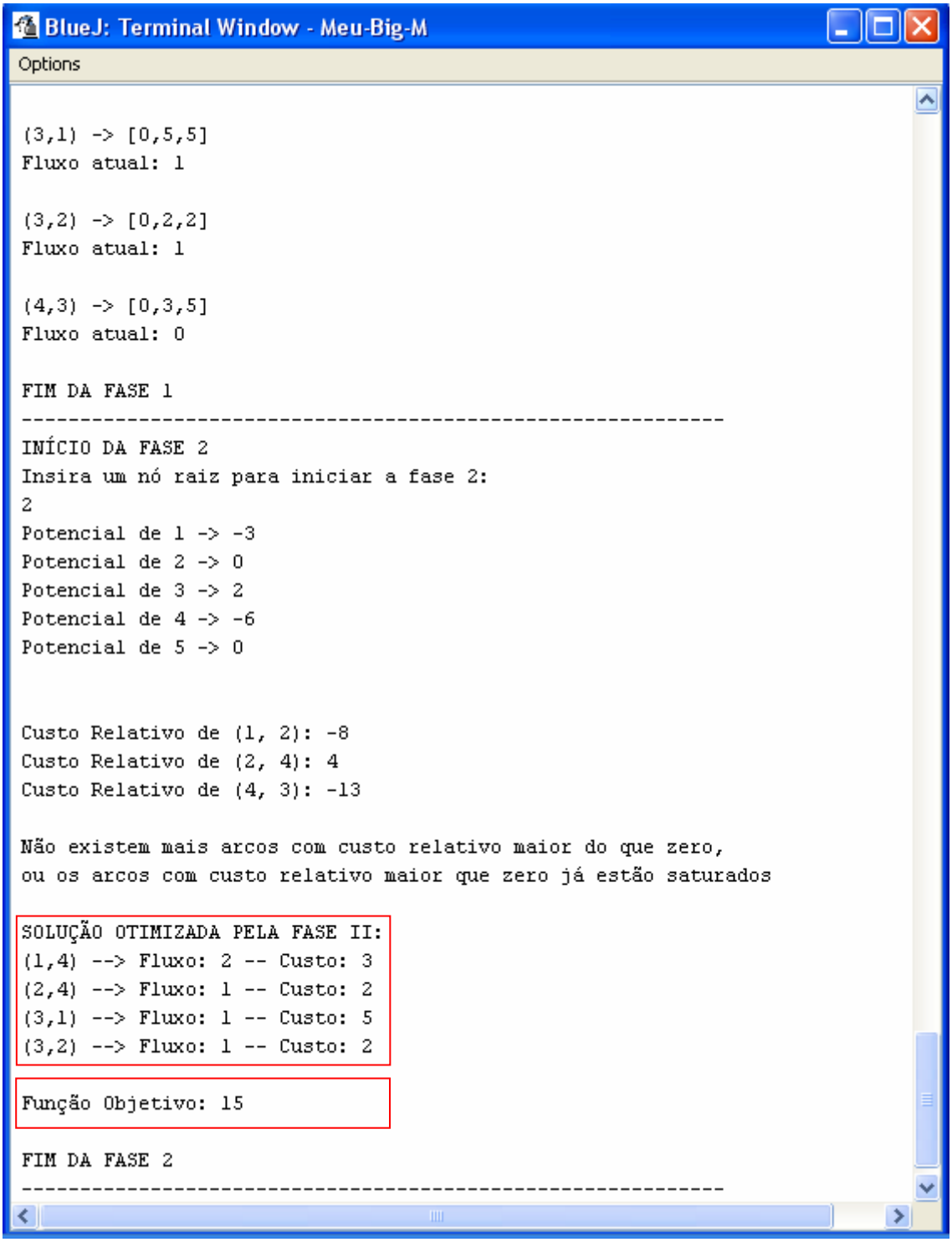
Entrada:



Solução Ótima:



Execução do ex. 3: Solução Ótima – Fase II



Referências

[IME08] Algoritmos, Experimentação e Teoria em Otimização Combinatória, disponível em http://www.linux.ime.usp.br/~cef/mac499-04/monografias/rec/juliana/minflow.html, acessado em 5 de julho de 2008.

[WIKI08] Algoritmo Simplex, disponível em http://pt.wikipedia.org/wiki/Algoritmo_simplex, acessado em 4 de julho de 2008.