Mestrado Integrado em Engenharia Informática e Computação

Complementos de Programação e Algoritmos

EXAME COM CONSULTA **9 Julho de 2007** DURAÇÃO: 2 horas

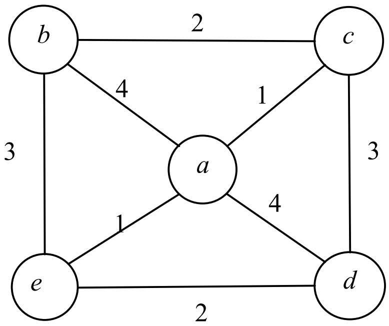
1. [5] Relativamente ao grafo da figura ao lado indique se existe e, em caso de existir, apresente:

a) [1] um conjunto de no máximo 3 vértices cobrindo todas as arestas;

b) [1] um clique de tamanho ≥ 4;

c) [1] um percurso do caixeiro viajante de peso ≤ 12 (percurso fechado que passa pelo menos uma vez em cada vértice);

d) [2] um percurso do carteiro chinês de peso ≤ 24 (percurso fechado que passa pelo menos uma vez em cada aresta).



Nota: na alínea d), começar por encontrar um percurso óptimo do carteiro chinês, seguindo os passos estudados nas aulas.

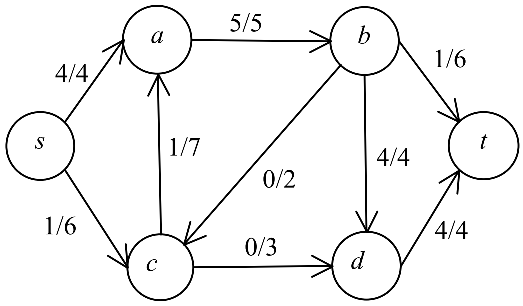
2. [2.5] Considerar o grafo de fluxos e capacidades numa rede de transporte indicado na figura ao lado.

a) [0.5] Qual é o valor do fluxo que passa na rede?

b) [1] Desenhe o grafo de resíduos correspondente.

c) [0.5] Identifique um caminho de aumento no grafo de resíduos e o valor de fluxo correspondente.

d) [0.5] Actualize o grafo de fluxos aplicando o caminho de aumento anterior.

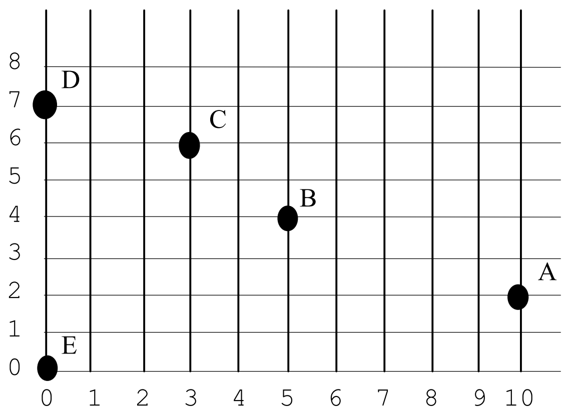


3. [2.5] Considere cinco cidades localizadas nas coordenadas A(10,2), B(5,4), C(3,6), D(0,7) e E(0,0). Pretende-se ligar estas cidades com uma quantidade mínima de cabo. As distâncias Euclidianas aproximadas entre estas cidades estão representadas no quadro abaixo.

a) [0.5] Que algoritmo usaria para resolver o problema?

b) [0.5] Desenhe o grafo a fornecer como entrada para o algoritmo.

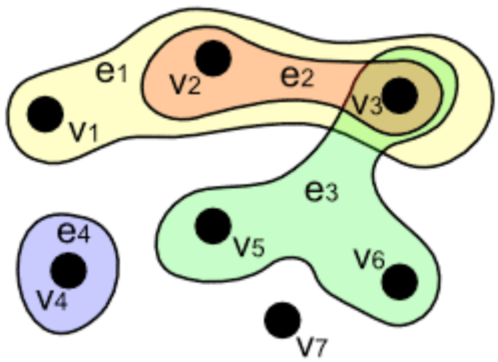
c) [1.5] Obtenha uma solução para o problema usando o algoritmo. Indicar todos os passos.



	B	**C**	**D**	**E**
A	5.4	8.1	11.2	10.2
B		2.8	5.8	6.4
C			3.2	6.7
D				7.0

4. [2] Considere as strings S="ABCD" e T="ACED".
a) [0.5] Qual é a distância de edição entre estas strings?
b) [1.5] Construa a matriz D[i,j] estudada nas aulas para determinar a distância de edição pelo método de programação dinâmica.

5. [3] Um hipergrafo é uma generalização de um grafo em que as arestas podem ligar qualquer número de vértices. Formalmente, um hipergrafo é um par (V,E) em que V é um conjunto de elementos, chamados *nós* ou *vértices*, e E é um conjunto de subconjuntos não vazios de X chamados *hiperarestas*. A figura ao lado mostra um exemplo de um hipergrafo.



$V = \{v_1,v_2,v_3,v_4,v_5,v_6,v_7\}$
$E = \{e_1,e_2,e_3,e_4\}$
$\quad = \{\{v_1,v_2,v_3\},\{v_2,v_3\}, \{v_3,v_5,v_6\},\{v_4\}\}.$

a) [1] Mostre que o problema de verificar se um hipergrafo tem um conjunto com K ou menos vértices cobrindo todas as arestas é NP-completo.

b) [2] Indique (por palavras) um algoritmo de aproximação ganancioso para obter um conjunto de vértices de tamanho mínimo cobrindo todas as arestas. Que resultado daria no caso da figura (em que a solução óptima é $\{v_3,v_4\}$). Qual é a eficiência espacial e temporal desse algoritmo?

6. [5] Escreva uma classe `Hypergraph` em Java que permita executar o código de teste indicado abaixo, referente ao exemplo do problema 5. Implemente o algoritmo ganancioso do problema 5.b) no método `getMinVertexCover`.

```
class HypergraphTest extends TestCase {
  public void testExample() {
    Hypergraph g = new Hypergraph();

    g.addVertex(1); g.addVertex(2); g.addVertex(3); g.addVertex(4);
    g.addVertex(5); g.addVertex(6); g.addVertex(7);

    g.addEdge(1); g.addVertexToEdge(1, 1); g.addVertexToEdge(2,1);
                  g.addVertexToEdge(3,1);
    g.addEdge(2); g.addVertexToEdge(2, 2); g.addVertexToEdge(3,2);
    g.addEdge(3); g.addVertexToEdge(3, 3); g.addVertexToEdge(5,3);
         g.addVertexToEdge(6,3);
    g.addEdge(4); g.addVertexToEdge(4, 4);

    Set<Integer> vertexCover = new HashSet<Integer>();
    vertexCover.add(3); vertexCover.add(4);

    assertEquals( vertexCover, g.getMinVertexCover());
  }
}
```

Nota: A parte da classe `Hypergraph` que permite representar e construir o hipergrafo vale 2.5 valores; a parte referente ao método `getMinVertexCover` vale os restantes 2.5 valores.

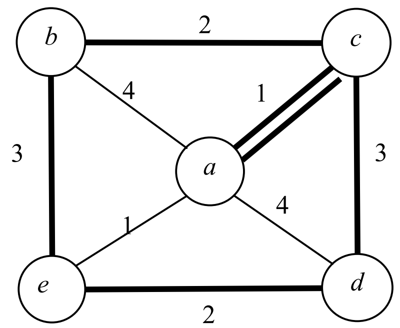
Resolução

1.

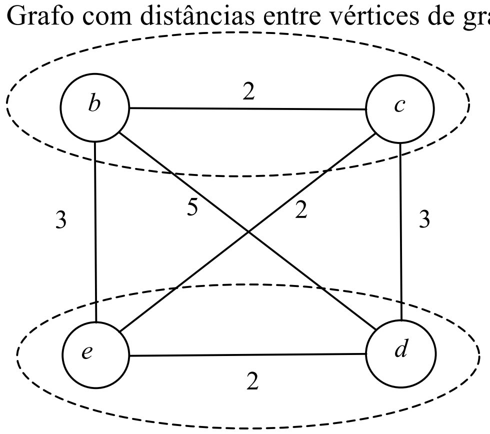
a) Existe: {a, b, d} ou {a, c, e}

b) Não existe. Só existem cliques de tamanho 3 (qualquer dos triângulos do grafo).

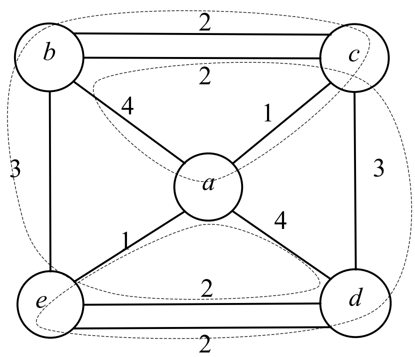
c) Existe. Por exemplo o percurso que passa nas arestas indicadas a traço forte na figura passa em todos os vértices e tem peso total 12.



d)
Grafo com distâncias entre vértices de grau ímpar, e emparelhamento óptimo de peso mínimo:



Grafo inicial duplicando caminhos mais curtos entre vértices emparelhados, e um percurso possível (peso total 24).

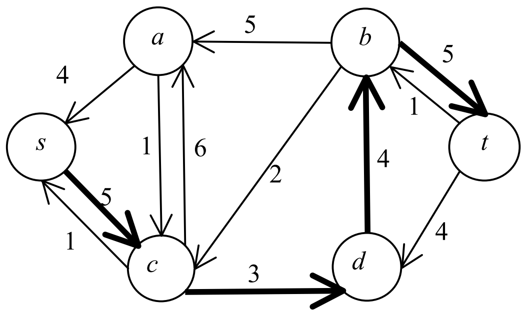


Como o peso total deste percurso é 24, a resposta é afirmativa.

2.

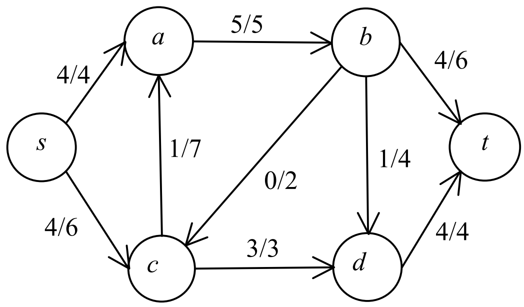
a) 5

b)



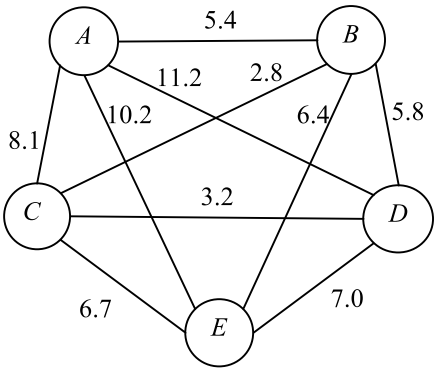
c) Ver caminho a traço forte na figura anterior.
O valor do fluxo de aumento é 3.

d)



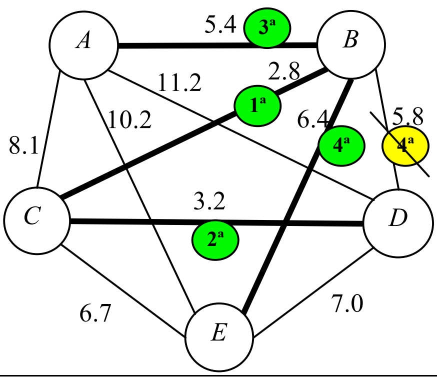
3. a) Determinação da árvore de expansão mínima, por exemplo pelo algoritmo de Kruskal.

b)

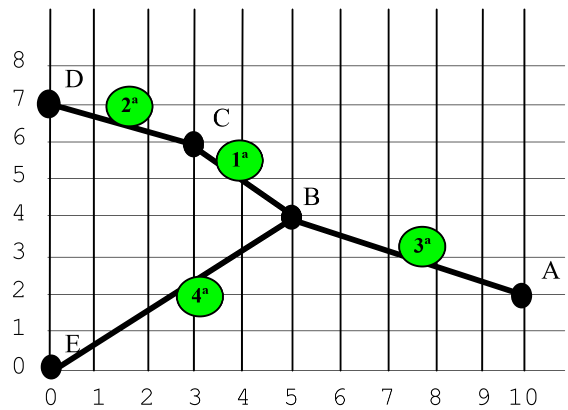


c) Escolhe-se sucessivamente uma aresta de peso mínimo que não causa um ciclo, até ligar todos os vértices.

Resultado e ordem de construção
(arestas a traço forte e nºs de ordem):



Visto agora no "mapa" (não pedido):



4. a) A distância de edição é 2 (por exemplo, em T pode-se substituir C por B e E por C).

b)

D[i,j]		**A**	**C**	**E**	**D**
	0	1	2	3	4
A	1	0	1	2	3
B	2	1	1	2	3
C	3	2	1	2	3
D	4	3	2	2	**2**

5. a) É necessário provar que:
(i) está em NP (i.e., que uma solução potencial pode ser verificada em tempo polinomial);
(ii) existe um problema NP-completo que é redutível a este.
(i) Dada uma potencial solução W (subconjunto de vértices), é trivial verificar em tempo polinomial que $|W| \leq K$ e que todas as hiperarestas são cobertas pelos vértices em W.
(ii) Sabe-se que o problema da cobertura de vértices em grafos normais (em que cada aresta liga dois vértices) é NP-completo. Um grafo normal pode ser visto como um hipergrafo em que por acaso todas as hiperarestas incidem em 2 vértices (ou pode ser trivialmente convertido para um hipergrafo desse tipo).

b) (i) Escolhe-se o vértice que cobre mais hiperarestas, simula-se a eliminação desse vértice e das hiperarestas por ele cobertas, e repete-se o processo até não existem mais hiperarestas.
(ii) No exemplo da figura seria escolhido primeiro o vértice v_3 e depois o vértice v_4, dando portanto a solução óptima $\{v_3,v_4\}$.
(iii) *Eficiência temporal* (detalhando o algoritmo): Seja n o nº de vértices, m o nº de hiperarestas e r o nº de pares (vértice, aresta). Para evitar a eliminação de vértices e hiperarestas, mantém-se em cada vértice um contador do nº de hiperarestas remanescentes cobertas por esse vértice, e mantém-se em cada aresta uma flag a indicar se já foi coberta. No início o contador é inicializado com o nº de hiperarestas incidentes no vértice e a flag é inicializada com false, o que pode ser feito em tempo $O(n+m)$. Em cada iteração, o vértice com o maior valor do contador pode ser escolhido em tempo $O(n)$. A simulação da eliminação consiste em percorrer as arestas incidentes no vértice seleccionado e, para cada aresta que não estava ainda coberta (flag=false), marcar a flag a true e decrementar os contadores dos vértices abrangidos por essa aresta. As várias simulações de eliminação podem ser feitas em tempo $O(r)$ se existirem listas ligadas que permitem aceder dos vértices às arestas e vice-versa. O tempo total do algoritmo fica então $\underline{O(n^2+r)}$. Usando uma fila com prioridades para guardar os vértices com o de valor mais alto do contador à cabeça, consegue-se $\underline{O(r \log n + r)}$, que pode ser melhor que o anterior, dependendo do valor de r.
Eficiência espacial: se não for usada uma fila de prioridades, é necessário apenas o espaço para guardar os dados auxiliares (contador e flag), ou seja, $\underline{O(n+m)}$.

6.
```
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;
import java.util.Set;

public class Hypergraph {
   static class Vertex {
      int num;
      LinkedList<Edge> edges = new LinkedList<Edge>();
      int numRemEdges; // nº de arestas cobertas (para getMinVertexCover)
      public Vertex(int num) {
         this.num = num;
      }
   }
```

```java
    static class Edge {
       int num;
       LinkedList<Vertex> vertices = new LinkedList<Vertex>();
       boolean covered; // usado por getMinVertexCover
       public Edge(int num) {
          this.num = num;
       }
    }

    // Conjuntos de vértices e arestas acessíveis pelo seu número
    private Map<Integer,Vertex> vertexSet = new HashMap<Integer,Vertex>();
    private Map<Integer,Edge> edgeSet = new HashMap<Integer, Edge>();

    public void addVertex(int num) {
       vertexSet.put(num, new Vertex(num));
    }
    public void addEdge(int num) {
       edgeSet.put(num, new Edge(num));
    }
    public void addVertexToEdge(int vertexNum, int edgeNum) {
       Edge e = edgeSet.get(edgeNum);
       Vertex v = vertexSet.get(vertexNum);
       v.edges.add(e);
       e.vertices.add(v);
    }

    public Set<Integer> getMinVertexCover() {
       // Inicializa resultado e dados auxiliares
       Set<Integer> result = new HashSet<Integer>();
       int numCovered = 0;
       for (Vertex v : vertexSet.values())
          v.numRemEdges = v.edges.size();
       for (Edge e : edgeSet.values())
          e.covered = false;
       // Ciclo principal
       while (numCovered != edgeSet.size()) {
           // Escolhe o vértice que cobre mais arestas remanescentes
          // Nota: podia ser optimizado com fila de prioridades
          Vertex bestV = null;
          int numRemEdges = 0;
          for (Vertex v : vertexSet.values() ) {
             if (v.numRemEdges > numRemEdges) {
                numRemEdges = v.numRemEdges;
                bestV = v;
             }
          }
          // Acrescenta esse vértice ao resultado e actualiza dados auxiliares
          for (Edge e: bestV.edges)
             if ( ! e.covered ) {
                result.add(bestV.num);
                numCovered++;
                e.covered = true;
                for (Vertex w : e.vertices)
                   w.numRemEdges --;
             }
       }
       return result;
    }
}
```