Mestrado Integrado em Engenharia Informática e Computação

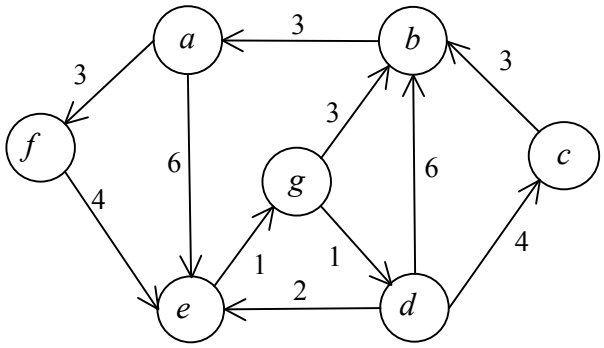
Complementos de Programação e Algoritmos

EXAME COM CONSULTA **11 Junho de 2007** DURAÇÃO: 2 horas

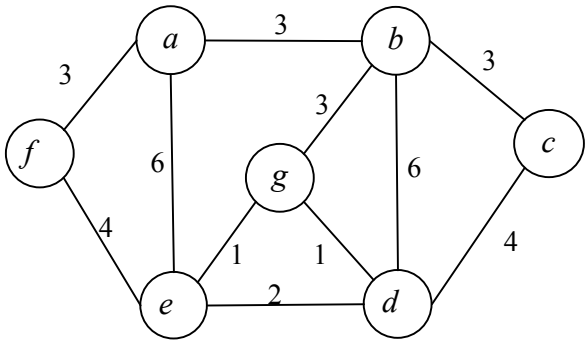
1. [6] Relativamente ao grafo dirigido ou ao grafo não dirigido subjacente da figura seguinte indique se existe e, em caso de existir, apresente:

a) [1] um circuito de Euler (no grafo dirigido);
b) [1] um circuito de Hamilton (no grafo dirigido);
c) [1] um fluxo de valor ≥ 6 de *f* (fonte) para *c* (poço), supondo que os pesos representam capacidades e que o fluxo pode passar nas arestas em qualquer sentido (no grafo não dirigido);
d) [1] uma árvore de expansão de peso ≤ 14 (no grafo não dirigido);
e) [1] um percurso do carteiro chinês de peso ≤ 40 (no grafo não dirigido);
f) [1] um emparelhamento de peso ≥ 13 (no grafo não dirigido).

Grafo dirigido



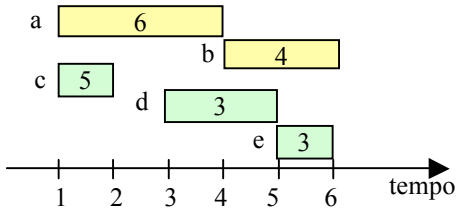
Grafo não dirigido subjacente



2. [6] Dado um conjunto de *n* actividades com instantes de início e fim $[s_i, f_i[$ e peso w_i, para $i = 1, ..., n$, pretende-se achar um subconjunto *S* de actividades não sobrepostas de peso total máximo.

Considere o seguinte algoritmo ganancioso: inserir em *S* a actividade de peso máximo w_i, remover todas as actividades que se sobrepõem à actividade *i* e repetir até não restarem actividades. Na prática, não é necessário remover actividades, basta processá-las por pesos decrescentes e inserir em *S* as que não se sobrepõem a outras já em *S*.

Por exemplo, no caso das actividades representadas por rectângulos na figura ao lado, o algoritmo ganancioso escolheria as actividades {a , b}, de peso total 10.
Note-se que o algoritmo ganancioso nem sempre garante a solução óptima, que neste caso seria {c, d, e}, de peso total 11.



a) [5] Implemente em Java este algoritmo ganancioso num método estático público `selectNonOverlapping` da classe `Activity`, partindo do seguinte esqueleto (pode criar métodos auxiliares e alterar a classe como melhor entender e deve usar colecções do Java):

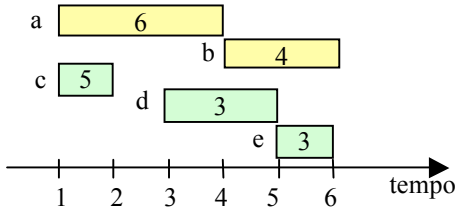
```
import java.util.Set;
public class Activity {
  private long start, finish, weight;
  // Construtor e selectores omitidos
  public static Set<Activity> selectNonOverlapping(Set<Activity> s) {...}
}
```

Nota: se não se recordar do nome e comportamento exacto de algum método das bibliotecas do Java, indique em comentário o comportamento esperado.

b) [1] Indique, justificando, a complexidade espacial e temporal do programa.

Os exercícios seguintes analisam 3 estratégias alternativas para encontrar uma solução óptima para o problema proposto no exercício 2.

Repete-se ao lado o exemplo para facilitar a leitura.



3. [4] Uma solução óptima para o problema 2 pode ser encontrada eficientemente com base na técnica de programação dinâmica.

Inicialmente, ordenam-se as actividades por forma a definir os *arrays*:
- `activity[i]` - i-ésima actividade por ordem de instantes de fim crescentes;
- `prev[i]` - maior índice *j* tal que $j < i$ e as actividades *i* e *j* não se sobrepõem, isto é, tal que $f_j \leq s_i$ (é 0 se não existir nenhum *j* nessas condições).

De seguida aplica-se a técnica de programação dinâmica preenchendo dois *arrays*:
- `best[i]` - custo da melhor solução que se consegue usando apenas as actividades de 1 a i;
- `incl[i]` - indica se a actividade i está incluída na melhor solução com actividades de 1 a i.

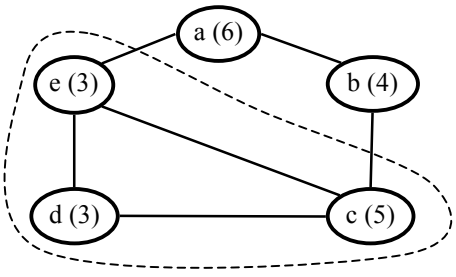
A tabela seguinte exemplifica o preenchimento destes *arrays* para o exemplo da figura acima.

i	0	1	2	3	4	n = 5
activity[i]	-	c	a	d	e	b
prev[i]	-	0	0	1	3	2
best[i]	0	5	6	8	11	11
incl[i]	-	true	true	true	true	false

a) [1] Indique por código Java as fórmulas e ordem de cálculo dos *arrays* `best[i]` e `incl[i]`.
b) [1] Indique por código Java o processo de obtenção da solução óptima `S` (do tipo `Set<Activity>`) a partir dos *arrays* já calculados.
c) [1] Indique (por palavras) como se pode calcular o *array* `prev[i]` em tempo *O(n log n)*.
d) [1] Indique, justificando, a complexidade espacial e temporal do algoritmo.

4. [2] Uma solução óptima para o problema 2 pode ser encontrada eficientemente através da sua redução ao problema de encontrar um caminho simples de peso máximo (caminho "mais comprido") num grafo dirigido acíclico, o qual pode ser resolvido em tempo linear no tamanho do grafo. Explique como se pode efectuar essa redução, exemplificando para o caso da figura (mostrar o grafo correspondente e o caminho óptimo).

5. [2] Uma solução óptima para o problema 2 pode ser encontrada de forma potencialmente ineficiente através da sua redução ao problema de encontrar um clique (subgrafo completo) de peso máximo num grafo não dirigido com pesos nos vértices. Para esse efeito, cria-se um grafo em que os vértices representam as actividades (com os respectivos pesos) e as arestas ligam actividades não sobrepostas. A figura ao lado mostra o grafo e a solução óptima para o exemplo dado.



Sabendo-se que o problema de encontrar um clique de tamanho máximo num grafo não dirigido é NP-completo, mostre que o problema de encontrar um clique de peso máximo num grafo não dirigido com pesos nos vértices é também NP-completo. Sendo assim, justifique porque é que esta estratégia de resolução do problema 2 é potencialmente ineficiente.

FIM

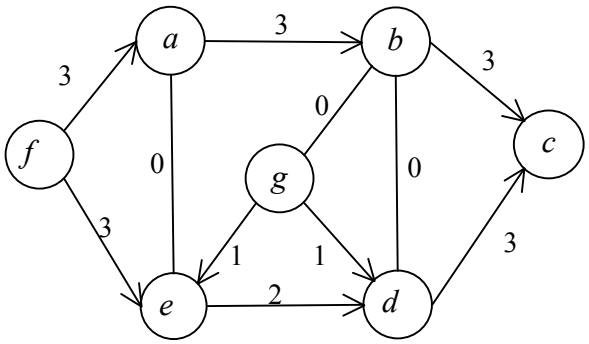
Resolução

1.

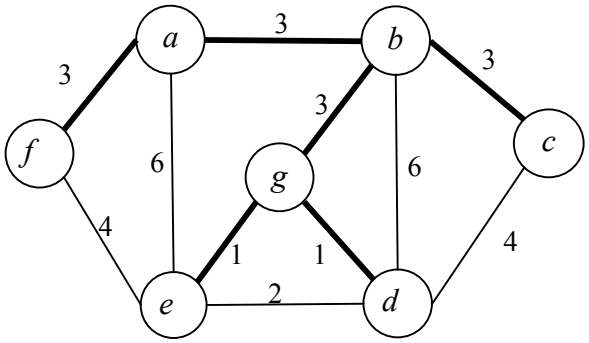
a) Não tem nenhum circuito de Euler porque existem vértices com diferentes graus de entrada (nº de arestas a entrar) e saída (nº de arestas a sair), como por exemplo o vértice *a*.

b) Sim - o circuito: $f \rightarrow e \rightarrow g \rightarrow d \rightarrow c \rightarrow b \rightarrow a \rightarrow f$.

c) Sim. É possível passar um fluxo de valor 6, como indica o seguinte grafo de fluxos:

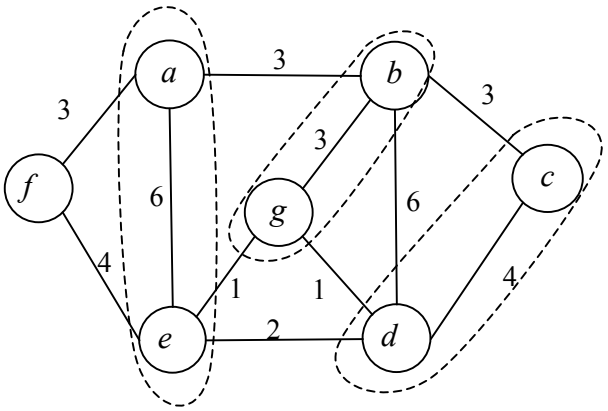


d) Sim, pois a árvore de expansão mínima (indicada na figura seguinte) tem precisamente peso 14.



e) Começa-se por achar o percurso óptimo (de peso mínimo) do carteiro chinês. Os únicos vértices de grau ímpar são os vértices *a* e *g* e o caminho mais curto entre eles é o caminho *a-b-g*, que é necessário duplicar para se obter um grafo Euleriano. Somando os pesos das arestas do grafo original (36) com o peso das arestas duplicadas (6), obtemos 42>40. Logo a resposta é não.

f) Sim. O emparelhamento {(a,e), (b,g), (d,c)} (ver figura seguinte) tem peso total 13.



2.

a) Uma resolução possível:

```
import java.util.Set;
import java.util.HashSet;
import java.util.PriorityQueue;

public class Activity implements Comparable<Activity> {
    private long start, finish, weight;

    // Não pedido, criado só para efectuar testes
    public Activity(long start, long finish, long weight) {
        assert start <= finish && weight >= 0;
        this.start = start; this.finish = finish; this.weight = weight;
    }

    // Obtém um subconjunto de actividades não sobrepostas de peso máximo
    public static Set<Activity> selectNonOverlapping(Set<Activity> s) {
        PriorityQueue<Activity> q = new PriorityQueue<Activity>(s);
        Set<Activity> result = new HashSet<Activity>();
        while ( ! q.isEmpty() ) {
            //  nota: extrai mínimo da fila que, dada a definição de compareTo,
            // vai dar a actividade de peso máximo
            Activity a = q.poll();
            if ( ! a.overlapAny(result) )
                result.add(a);
        }
        return result;
    }

    // Verifica se esta actividade se sobrepõe a alguma dum conjunto dado
    private boolean overlapAny(Set<Activity> s) {
        for (Activity a : s)
            if ( this.overlap(a) )
                return true;
        return false;
    }

    // Verifica se esta actividade (this) se sobrepõe a outra (a)
    public boolean overlap(Activity a) {
        return this.start < a.finish && a.start < this.finish;
    }

    // Compara esta actividade (this) com outra (a) por peso de forma inversa
    public int compareTo(Activity a) {
        return Long.signum(a.weight - this.weight);
               // signum dá +1, 0 ou -1 conforme o argumento é >0, 0 ou <0
    }
}
```

Nota: Não seria boa ideia tentar usar a capacidade de eliminação de duplicados de HashSet, pois seria necessário definir não só o método equals mas também o método hashCode (por forma a retornarem true e o mesmo valor de hash para acontecimentos duplicados).

b) Nesta implementação (pouco optimizada) a complexidade temporal é $T(n) = O(n^2 + n \log n) = O(n^2)$. A parcela $O(n \log n)$ tem a ver com $2n$ operações na fila de prioridades. A parcela $O(n^2)$ tem a ver com n execuções de `overlapAny`. A complexidade espacial é $S(n) = O(n)$ devido à utilização da fila de prioridades.

3. a)
```
best[0] = 0;
for (int i = 1; i <= n; i++) {
  best[i] = Math.max(best[i-1], activity[i].weight + best[prev[i]]);
  incl[i] = best[i] > best[i-1];
}
```

b)
```
for (int i = n; i > 0; )
  if (incl[i]) {
        result.add(activity[i]);
        i = prev[i];
  }
  else
        i--;
```

c) Ordena-se o *array* `startorder[] = {1, 2, ..., n}`, em que os nºs identificam actividades no *array* `activity[]`, por ordem de instantes de início crescentes. Percorrendo em paralelo os *arrays* `startorder[]` e `activity[]` consegue-se calcular o array `prev[]`:

```
for (int i = 0, j = 0; j < n; j++) {
     while (i < n && activity[i+1].finish <= activity[startorder[j]].start)
           i++;
     prev[startorder[j]] = i;
}
```

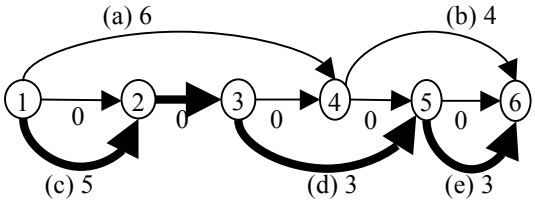
Dado que a ordenação pode ser feita em tempo *O(n log n)* e o ciclo acima corre em tempo *O(n)* (a instrução `i++` é executada *n* vezes no máximo), `prev[]` é calculado em tempo *O(n log n)*.

d) A construção e ordenação do array `activity[]` pode ser feita em tempo *O(n log n)*. De acordo com a alínea anterior, o array `prev[]` pode ser calculado no mesmo tempo. O resto do trabalho (alíneas a) e b)) é feito em tempo linear. Logo a complexidade temporal é $T(n) = O(n \log n)$. A complexidade espacial é obviamente *O(n)*.

---

4. Cria-se um grafo em que os vértices representam os instantes de tempo referenciados nas várias actividades (instantes de início ou fim, sem repetições). Criam-se arestas de peso 0 a ligar esses vértices por ordem cronológica. Representa-se cada actividade por uma aresta dirigida do respectivo instante de início para o respectivo instante de fim, com o respectivo peso.

Exemplo:



Nota 1: Usando um HashSet, pode-se construir o conjunto de instantes de tempo / vértices, sem repetições, em tempo *O(n)*. A ordenação necessária para adicionar as arestas de peso 0 pode ser feita em tempo *O(n log n)*. As arestas correspondentes às actividades podem ser adicionadas em tempo *O(n)*. Uma vez que o tamanho do grafo é de ordem *O(n)*, o caminho óptimo é encontrado em tempo *O(n)*. Tem-se assim a mesma complexidade espacial e temporal que no exercício 3! Em alternativa, podia-se construir um grafo de precedências entre actividades, mas a complexidade espacial e temporal piorava para $O(n^2)$.

---

5. Reduz-se o problema do clique de tamanho máximo num grafo G ao problema do clique de peso máximo num grafo G', considerando pesos unitários. Por outro lado, é óbvio que o problema do clique de peso máximo está em NP (uma solução pode ser verificada em tempo polinomial). Assim, o problema do clique de peso máximo é NP-completo. Isto implica que, segundo o estado actual do conhecimento, não existe nenhum algoritmo de tempo polinomial para resolver o problema do clique de peso máximo. Portanto é má ideia usar esta abordagem para resolver o problema 2.