Métodos de Programação II
LESI – LMCC
Exercícios sobre Análise de Algoritmos

Manuel Bernardo Barbosa e Jorge Sousa Pinto

Abril 2004

Contents

1 Provas de Correcção: Invariantes de Ciclo **2**

2 Análise Assimptótica **3**
2.1 Notações 3
2.2 Análise de Algoritmos Não-recursivos 3
2.3 Análise de Algoritmos Recursivos: Recorrências e Árvores de Recursão . . 4
2.4 Recorrências: Método da Substituição 5

3 Exercícios Diversos **6**

4 Estruturas de Dados **12**
4.1 Tabelas 12
4.2 Árvores 13
4.3 Grafos 14

1 Provas de Correcção: Invariantes de Ciclo

1. Considere o algoritmo de ordenação Bubble Sort:

```
void bubble_sort(int A[], int N) {
  for (i=1 ; i<N ; i++)
    for(j=N ; j>i ; j--)
      if (A[j] < A[j-1])
         swap(A,j,j-1);
}

void swap(int A[], int i, int j){
  int aux=A[i];
  A[i]=A[j];
  A[j]=aux;
}
```

   (a) Escreva um invariante de ciclo para o ciclo interior do algoritmo. Demonstre a sua validade.

   (b) Escreva um invariante de ciclo para o ciclo exterior do algoritmo. Demonstre a correcção do algoritmo utilizando esse invariante. Baseie a sua demonstração na propriedade que demonstrou na alínea anterior.

2. Considere a seguinte versão alternativa do algoritmo de ordenação Bubble Sort:

```
void bubble_sort(int A[], int N) {
  for (i=1 ; i<N ; i++)
    for(j=N ; j>i ; j--)
      if (A[j] < A[i])
         swap(A,j,i);
}
```

   (a) Em que diferem as duas versões apresentadas?

   (b) Escreva um invariante de ciclo para o ciclo interior do algoritmo. Demonstre a sua validade.

   (c) Escreva um invariante de ciclo para o ciclo exterior do algoritmo. Demonstre a correcção do algoritmo utilizando esse invariante de ciclo. Baseie a sua demonstração na propriedade que demonstrou na alínea anterior.

2 Análise Assimptótica

2.1 Notações

1. Utilizando as definições das notações assimptóticas, demonstre que:

   (a) Se $f(n)$ e $g(n)$ forem funções assimptoticamente não negativas, então

   $$max(f(n), g(n)) = \Theta(f(n) + g(n))$$

   (b) O tempo de execução de um algoritmo é $\Theta(g(n))$ sse o seu tempo de execução no pior caso for $\mathcal{O}(g(n))$ e o seu tempo de execução no melhor caso for $\Omega(g(n))$.

2.2 Análise de Algoritmos Não-recursivos

1. Considere de novo o algoritmo Bubble Sort apresentado na secção anterior.

   (a) Caracterize o funcionamento deste algoritmo *no melhor e no pior caso* utilizando a notação Θ.

   (b) Caracterize o funcionamento *global* deste algoritmo utilizando as notações $\mathcal{O}$ e Ω. Que relação existe entre a resposta a esta alínea e a resposta à alínea anterior?

   (c) Altere o ciclo exterior do algoritmo por forma a terminar quando detectar que o vector já está ordenado.

   (d) Repita a sua análise das alíneas 1 e 2 para a nova versão do algoritmo.

2. Considere a seguinte descrição informal de um algoritmo de ordenação a que chamaremos *max sort*:

   > *A sequência a ordenar está em cada passo dividida em duas sub-sequências, uma não-ordenada seguida de uma ordenada (inicialmente a parte ordenada é vazia). Em cada passo o algoritmo selecciona o maior elemento na parte não-ordenada e* troca-o *com o último elemento dessa mesma parte. Neste momento esse elemento passa a fazer parte da sub-sequência ordenada.*

   Exemplo de execução (o caracter | indica a fronteira entre as sub-sequências):

   $$[3, 4, 1, 2\ |] \longrightarrow [3, 2, 1 \mid 4] \longrightarrow [1, 2 \mid 3, 4] \longrightarrow [1 \mid 2, 3, 4] \longrightarrow [|\ 1, 2, 3, 4]$$

   Efectue a análise assimptótica do comportamento no pior caso de uma implementação *baseada em ciclos* (i.e. sem recursividade) deste algoritmo.

2.3 Análise de Algoritmos Recursivos: Recorrências e Árvores de Recursão

1. O algoritmo Insertion Sort pode ser definido recursivamente da seguinte forma: para ordenar $A[1..n]$, começa-se por ordenar $A[1..n-1]$, e insere-se depois $A[n]$ no array $A[1..n-1]$ já ordenado.

   (a) Descreva o tempo de execução desta implementação do Insertion Sort através de equações de recorrência.

   (b) Desenhe as árvores de recursividade correspondentes ao melhor caso e ao pior caso do tempo de execução. Utilize estas árvores para deduzir o comportamento assimptótico deste algoritmo.

2. Utilize uma árvore de recursividade para encontrar um bom limite superior para o tempo de execução dado pela seguinte recorrência:

$$T(n) = 3T(\lfloor \frac{n}{2} \rfloor) + n \tag{1}$$

3. Considere o seguinte algoritmo para o problema das *Torres de Hanoi*:

```
void Hanoi(int nDiscos, int esquerda, int direita, int meio)
{
  if (nDiscos > 0) {
    Hanoi(nDiscos-1, esquerda, meio, direita);
    printf("mover disco de %d para %d\n", esquerda, direita);
    Hanoi(nDiscos-1, meio, direita, esquerda);
  }
}
```

   Escreva uma relação de recorrência para este algoritmo. Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade.

4. Considere a seguinte codificação recursiva em C do algoritmo *maxsort* descrito acima:

```
void maxsort (int v[], int n)
{
  int i;
  if (n > 1) {
    i = max(v, n);
    swap(v, i, n);
    maxsort(v, n-1);
  }
}
```

em que a função `swap` troca os valores contidos em duas posições de um vector (em tempo constante) e a função `max` determina o elemento máximo de um vector (em tempo linear).

Escreva uma relação de recorrência para o tempo de execução no pior caso da função `maxsort`. Encontre a respectiva solução.

2.4 Recorrências: Método da Substituição

(n.b. matéria não leccionada em 2003-04)

1. Demonstre que a seguinte recorrência satisfaz $T(n) = \Theta(lg(n))$:

$$T(n) = T(\lceil \frac{n}{2} \rceil) + 1 \tag{2}$$

2. Demonstre que a seguinte recorrência satisfaz $T(n) = \Theta(n * lg(n))$:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor + 17) + n \tag{3}$$

3. Demonstre que a recorrência exacta do algoritmo Merge Sort é $T(n) = \Theta(n * lg(n))$:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) \tag{4}$$

4. Demonstre pelo método da substituição o limite superior que obteve para a última alínea do grupo anterior.

3 Exercícios Diversos

1. Considere o seguinte algoritmo para o problema da avaliação do valor de um polinómio

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

num ponto x dado, sendo o polinómio representado por um vector de coeficientes:

```
float Poly (float a[], int n, float x)
{
  float p, xpotencia;
  int i;
  p = a[0] + a[1] * x;
  xpotencia = x;
  for (i=2 ; i<=n ; i++) {
    xpotencia = xpotencia * x;
    p = p + a[i] * xpotencia;
  }
  return p;
}
```

   (a) Quantas operações de soma e multiplicação efectua este algoritmo? Caracterize o comportamento assimptótico no pior caso do seu tempo de execução.

   (b) Considere o seguinte invariante do ciclo `for`:

   > No início de cada iteração, a variável p contém o valor $a_{i-1}x^{i-1} + \ldots + a_1 x + a_0$.

   Estude as suas propriedades de Inicialização, Preservação, e Terminação.

2. Considere-se o algoritmo de Horner para o problema da avaliação do valor de um polinómio

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

num ponto x dado, sendo o polinómio representado por um vector de coeficientes. Este algoritmo é uma optimização do algoritmo *naif* de avaliação, que efectua uma *factorização* do polinómio [note-se que $ab + ac$ pode ser calculado com apenas uma multiplicação como $a(b + c)$]:

```
float HornerPoly (float a[], int n, float x)
{
  float p;
  int i;
  p = a[n];
  for (i=n-1 ; i>=0 ; i--)
```

```
    p = p*x + a[i];
  return p;
}
```

(a) Quantas operações de soma e multiplicação efectua este algoritmo? Caracterize o comportamento assimptótico no pior caso do seu tempo de execução.

(b) Identifique um *invariante* para o ciclo `for` do algoritmo, e utilize-o para provar que de facto o algoritmo efectua a avaliação do polinómio dado.

3. Considere-se o problema de *pesquisa* numa sequência de números:

> Dada uma sequência de números inteiros v de dimensão n e um inteiro x, pretende-se obter como resultado o índice da primeira ocorrência de x em v, ou o valor -1 caso x não ocorra em v.

A função seguinte (escrita em C) resolve o problema em tempo linear:

```
int search(int v[], int n, int x)
{
  int i = 0, found = 0;
  while (i < n && !found) {
    if (v[i] == x) found = 1;
    i++;
  }
  if (!found) return -1;
  else return i-1;
}
```

(a) Identifique um *invariante* para o ciclo `while` do algoritmo e analise as suas propriedades.

(b) Se o vector v estiver ordenado é possível melhorar o comportamento do algoritmo. Basta considerar a posição no meio do vector e comparar o seu conteúdo com o elemento x, o que permite eliminar a inspecção de metade do vector. O algoritmo de *pesquisa binária* repete este procedimento até encontrar x ou o sub-vector considerado ser vazio. Argumente que este algoritmo executa em tempo logarítmico no pior caso.

4. Considere o seguinte algoritmo para o problema da exponenciação módulo um número inteiro: $y = x^b \ \% \ p$.

```
int modExp (int x, int p, int b[], int l)
{
  int i,y;
```

```
  y=1;
  for (i=l-1; i>=0 ; i--) {
    y=y*y % p;
    if (b[i]==1) y=y*x % p;
  }
  return y;
}
```

Considere ainda que:

- todos os números têm `l` bits.
- $b = b_{l-1}2^{l-1} + b_{l-2}2^{l-2} + \ldots + b_1 2^1 + b_0$, sendo `b[0 ... l-1]` a representação binária do expoente,
- `%` representa o resto da divisão inteira,
- as operações `*` e `%` têm um custo $\mathcal{O}(l^2)$
- as operações `+` e `-` têm um custo $\mathcal{O}(l)$

(a) Quantas operações de soma, multiplicação e divisão efectua este algoritmo no melhor e no pior caso? E quantas vezes executa a operação `if`?

(b) Numa avaliação do tempo de execução de um qualquer algoritmo de exponenciação, como definiria o tamanho do input? Justifique e estabeleça um paralelo com este algoritmo em particular.

(c) Caracterize o comportamento assimptótico deste algoritmo utilizando as notações $\mathcal{O}$, Ω e Θ. Justifique a sua resposta e retire conclusões quanto à informação fornecida por estas notações, nomeadamente no que respeita a factores de escala e a detalhes para pequenos inputs.

(d) Demonstre a correcção do algoritmo utilizando o seguinte invariante de ciclo:
No início de cada iteração, temos $y = x^e \,\%\, p$, *com*

$$e = b_{l-1}2^{l-2-i} + b_{l-2}2^{l-3-i} + \ldots + b_{i+1}$$

5. Recorde a análise do tempo de execução do algoritmo Mergesort que tem como resultado um tempo de execução $\Theta(n * lg\ n)$.

   Essa análise considerava que o algoritmo ordenava uma lista de elementos armazenados num array e que, como tal, a operação de partição do array em dois tinha um custo $\Theta(1)$.

   Considere a seguinte versão do algoritmo Mergesort, operando sobre uma lista ligada.

```
typedef struct list List;
```

```
struct list {
  int num;
  struct list *next;
};

void divide(List *l, List **esq, List **dir);
List *merge(List *esq, List *dir);

List *mergeSort(List *l) {
  List *dir,*esq;
  divide(l,&esq,&dir);
  esq=mergeSort(esq);
  dir=mergeSort(dir);
  return merge(esq,dir);
}
```

(a) Forneça uma implementação da função `divide` sem recorrer à alocação de espaço de memória extra.

(b) Forneça uma implementação da funcão `merge` sem recorrer à alocação de espaço de memória extra.

(c) Escreva uma equação de recorrência que descreva o tempo de execução desta nova versão do algoritmo.

(d) Efectue uma análise assimptótica do tempo de execução para esta versão do algoritmo utilizando uma árvore de recursividade. (Sugestão: analize as diferenças que aparecem ao nível do custo de cada uma das operações de divisão, conquista e combinação em relação à versão do algoritmo estudada nas aulas).

6. O algoritmo de ordenação Tree Sort utiliza o seguinte procedimento para ordenar uma lista de n elementos:

   - constrói uma árvore binária de pesquisa, inserindo os elementos da lista um a um.
   - percorre a árvore binária de pesquisa por forma a obter uma versão ordenada da lista.

   As seguintes declarações pressupõem um funcionamento deste tipo:

```
typedef struct treenode TreeNode, *Tree;

struct treenode {
  int num;
  struct treenode *esq;
```

```
  struct treenode *dir;
}

Tree insertTree(Tree t, int num);
void treeSort(int A[], int n);
```

(a) Forneça uma implementação da função `insertTree` e escreva uma equação de recorrência que exprima o seu tempo de execução.

(b) Implemente a função `treeSort` , justificando a forma como irá percorrer a árvore binária de pesquisa. Utilize a função `insertTree` para construir a árvore de binária de pesquisa.

(c) Analize o tempo de execução de `treeSort` no melhor e no pior caso, e exprima esses comportamentos utilizando a notação Θ.

(d) Exprima o comportamento global de `treeSort` utilizando as notações $\mathcal{O}$ e Ω.

7. Considere o seguinte algoritmo para o cálculo de números de Fibonacci:

```
int fib (int n)
{
  if (n==0 || n==1) return 1;
  else return fib(n-1) + fib(n-2);
}
```

Apesar de traduzir exactamente a definição da sequência de números de Fibonacci, este algoritmo é muito ineficiente (de tempo exponencial). Assuma que as operações aritméticas elementares se efectuam em tempo $\mathcal{O}(1)$.

(a) Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.

(b) Efectue uma análise assimptótica do tempo de execução deste algoritmo. Sugestão: Utilize a árvore que desenhou na alínea anterior para fundamentar o seu raciocínio.

(c) Escreva em **C** um algoritmo alternativo mais eficiente. Analise o seu tempo de execução.

(d) Demonstre a correcção do algoritmo que escreveu na alínea anterior utilizando um invariante de ciclo.

8. Considere a seguinte definição:

> *Seja $\mathcal{A}$ um conjunto de pares ordenados (x, y) em que $x < y$, com x, y números inteiros positivos.* (x, y) *diz-se um par* mínimo *se não existe qualquer outro par $(x', y') \in \mathcal{A}$ tal que $x \leq x'$ e $y' \leq y$.*

Considere agora o tipo de dados com que se representa os ditos pares ordenados:

```
typedef struct { int p; int s; } pair;
```

A função seguinte pretende resolver o problema. O seu resultado é o vector `v`, que depois da execução do algoritmo conterá, na posição `i`, o valor 1 sse `A[i]` for um par mínimo (no caso contrário deverá conter 0). Considera-se que o vector `A` se encontra à partida ordenado por ordem crescente da primeira componente dos pares.

```
void minpairs (pair A[], char v[])
{
  int i,j;
  for (i=1; i<=N; i++)  v[i] = 1;
  for (i=N; i>=1; i--)
    for (j=i-1; j>=1; j--)
      if ((A[i].s <= A[j].s)) v[j] = 0;
}
```

(a) Efectue a análise assimptótica do tempo de execução no pior e no melhor casos do algoritmo. Justifique convenientemente a sua resposta.

(b) O ciclo interior obedece ao seguinte invariante:

> *No fim de cada iteração,* $\mathtt{v}[k] = 0$ *sse* $\mathtt{A}[k].\mathtt{p} \leq \mathtt{A}[\mathtt{i}].\mathtt{p}$ *e* $\mathtt{A}[\mathtt{i}].\mathtt{s} \leq \mathtt{A}[k].\mathtt{s}$*, para qualquer* k *tal que* $j \leq k \leq i-1$*.*

Estude as suas propriedades de inicialização, preservação, e terminação.

(c) O algoritmo pode ser melhorado (no que respeita ao seu comportamento temporal) mantendo a mesma estrutura de dois ciclos. Para isso complete as expressões P e Q no algoritmo abaixo. Diga o que se alterou no comportamento no pior e no melhor casos.

```
void minpairs (pair A[], char v[])
{
  int i,j;
  for (i=1; i<=N; i++)  v[i] = 1;
  for (i=N; i>=1; i--)
    if (___P___)
      for (j=i-1; j>=1; j--)
        if (___Q___ && (A[i].s <= A[j].s)) v[j] = 0;
}
```

4 Estruturas de Dados

4.1 Tabelas

1. Considere a seguinte implementação de uma tabela de hash:

```
typedef struct entry {
   char *key;
   void *info;
} Entry;

typedef struct hashtable {
  Entry *table;
  int size;
  int count;
} HashTable;

int hash(char *key, int size) {
   int h=0;
   h=(int) key[0];
   return (h);
}

HashTable doubleSize(HashTable H);

HashTable insert(HashTable H, Entry item) {
   int probe_count=0;
   int probe, h;

   h=hash(item.key,H.size);
   probe=h;
   while((H.table[probe].key != NULL)&&(probe_count < H.size)) {
      probe_count++;
      probe=(h+probe_count) % H.size;
   }
   if (H.table[probe].key == NULL) {
      H.table[probe]=item;
      H.count++;
      H=doubleSize(H);
   }
   return H;
}
```

   (a) Explique o funcionamento de uma tabela deste tipo, referindo-se nomeadamente

à estratégia de resolução de colisões e a possíveis alternativas de implementação.

(b) Indique o que entende por *factor de carga* (λ), e qual o significado deste parâmetro no contexto das tabelas de hash. No caso particular desta tabela, qual a gama de valores que este parâmetro pode tomar?

(c) A função `hash`apresenta vários problemas. Modifique esta função de uma forma que julgue aceitável, explicando as alterações que introduzir.

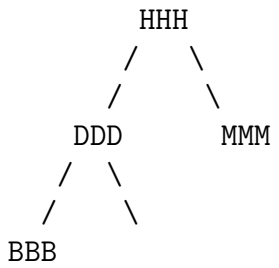
(d) Uma das formas de gerir uma tabela de hash com estas características consiste em duplicar o tamanho da tabela assim que λ ultrapassa o valor 0.5. Descreva o processo que teria de ser levado a cabo para este efeito.

(e) Implemente a função `doubleSize`utilizando o procedimento que descreveu na alínea anterior.

4.2 Árvores

1. (a) Descreva por palavras suas os princípios subjacentes à utilização de árvores AVL.

(b) Considere a seguinte árvore AVL. Mostre o resultado de se inserir sucessivamente nesta árvore os nós com chave AAA e CCC, preservando sempre o invariante destas árvores.



(c) Escreva em **C** uma função que receba uma árvore (AVL) e um inteiro d, e efectue uma rotação simples da árvore á direita (se $d = 1$) ou à esquerda (se $d = 0$). Utilize o tipo de dados:

```
struct _node_ {
  Info i;
  struct _node_ * e;
  struct _node_ *d;
};

typedef struct _node_ *bintree;
```

(d) Diga por palavras suas como implementaria um algoritmo iterativo (i.e. sem utilização de recursividade) para efectuar uma travessia PREORDER de uma árvore binária.

2. Considere as seguintes declarações com que se pretende representar uma árvore binária de pesquisa:

```
typedef struct node {
   int data;
   struct node* lptr;
   struct node* rptr;
} Node;

typedef Node *Tree;

int ordem(Tree t, int k);
int gama(Tree t, int a, int b);
Tree merge(Tree a, Tree b);
```

   (a) Implemente a função `ordem` que retorna o elemento de ordem k existente na árvore i.e. retorna o menor elemento tal que existem $k-1$ elementos de valor inferior.

   (b) Explique porque é que o problema da alínea anterior não pode ser resolvido em menos do que $\mathcal{O}(2N) = \mathcal{O}(N)$. Forneça uma definição alternativa do tipo `Tree` que permita implementar esta função em $\mathcal{O}(lg\ \mathrm{n}\ )$.

   (c) Implemente a função `gama` que imprime os elementos da árvore com valores $a \leq x \leq b$. O tempo de execução da sua função deverá ser $\mathcal{O}(n+N)$ em que n é o número de elementos imprimidos, e N o número total de elementos na árvore.

   (d) Implemente a função `merge`que funde duas árvores binárias de pesquisa numa só.

4.3 Grafos

1. Assumindo as seguintes definições de tipos de dados (representação mista de grafos):

```
typedef struct _edge_ {
  int dest;
  int weight;
  struct _edge_ *next;
} edge;

typedef edge* Graph[MAX];
```

   (a) Escreva uma função que calcule o peso do arco de menor peso de um grafo.

   (b) Efectue a análise assimptótica do tempo de execução da função que escreveu. O que se alteraria caso se utilizasse as implementações puramente ligada e por tabela de adjacências?

(c) No contexto de um sistema de reservas disponível em agências de viagens, considere o problema da determinação das ligações mais rápidas entre duas cidades, dado um horário de voos (directos). Este problema *não pode ser visto como um problema de caminhos mais curtos num (multi-)grafo cujos nós correspondam a cidades e arcos a voos.*

Justifique esta afirmação. Qual a consequência deste facto em termos da eficiência de um algoritmo para resolução do problema?

(d) O mesmo problema pode no entanto ser visto como um problema de caminhos mais curtos desde que se utilize um grafo em que os tempos de espera entre dois voos apareçam explicitamente representados como arcos no grafo.

Descreva detalhadamente como se deverá representar um conjunto de voos directos por um tal grafo, e discuta eventuais vantagens / desvantagens desta solução.

2. Considere o grafo representado pela seguinte lista de adjacências:

$$\begin{array}{ll} A: & B(4), F(2) \\ B: & A(1), C(3), D(4) \\ C: & A(6), B(3), D(7) \\ D: & A(6), E(2) \\ E: & D(5) \\ F: & D(2), E(3) \end{array}$$

(por exemplo, existem arcos de A para B com peso 4 e de A para F com peso 2)

(a) Desenhe o grafo assim representado. Classifique-o de acordo com os vários critérios que conhece.

(b) Quantos caminhos mais curtos existem do nó C para E? Qual destes seria determinado pelo algoritmo de Dijkstra (sobre a representação fornecida)? Justifique.

(c) Assumindo as seguintes definições de tipos de dados (representação mista de grafos):

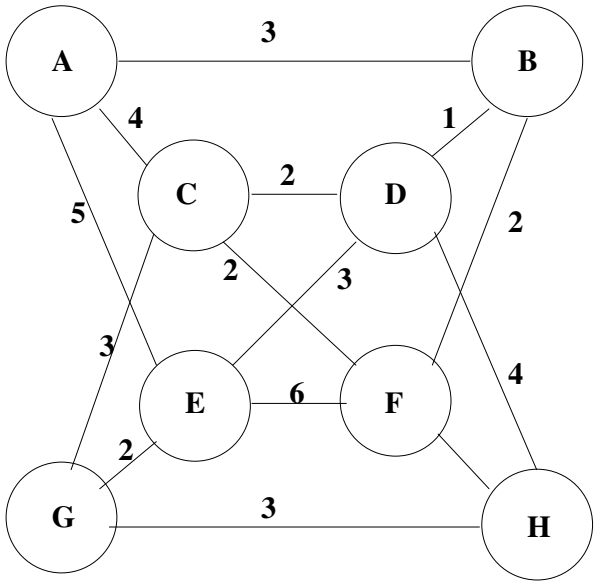
```
typedef struct _edge_ {
  int dest;
  int weight;
  struct _edge_ *next;
} edge;

typedef edge* Graph[MAX];
```

Escreva uma função que determine se existem dois arcos com o mesmo peso num grafo (deverá devolver 1 ou 0).

(d) Efectue a análise assimptótica do tempo de execução da função que escreveu. Como se poderia tornar esta função mais eficiente (em relação ao tempo de execução), sendo conhecido à partida o valor máximo dos pesos?

3. Considere o seguinte grafo:



(a) Classifique o grafo da figura.

(b) Desenhe uma árvore de antecessores passível de ser produzida pelo algoritmo de travessia Depth First que estudou nas aulas da disciplina. Considere que o nó A é utilizado como fonte.

(c) Escreva a declaração dos tipos de dados adequados para representar este grafo por listas de adjacência, numa implementação mista. Desenhe uma possível representação do grafo na figura utilizando os tipos de dados que definiu.

(d) Considere o algoritmo de Prim para o cálculo da Árvore Geradore Mínima de um grafo. Represente o grafo anterior, identificando os conjuntos dos nós da arvore, dos nós da orla e os arcos candidatos ao longo de uma execução do referido algoritmo. Considere que o nó A é o primeiro nó a ser escolhido para ser incluído na árvore.

(e) Escreva uma função que receba como parâmetros um grafo e um array de inteiros, e devolva o array preenchido da seguinte forma:

- Cada componente ligado do grafo o algoritmo fará corresponder um código inteiro sequêncial.
- O array indicará qual o componente ligado a que pertence o vértice `i`, devendo conter o código inteiro correspondente.