Métodos de Programação II
2º Ano - LESI
Exame da 2ª Chamada - Resolução

Edgar Sousa

13 de Julho de 2007

Parte I

1.

```
int maximo(Tree t, int *x) {
    if(t!=NULL){
        if(maximo(t->dir,x)==1){ //se não encontrou máximo do lado direito

          *x = t->elem; //sou eu o máximo
        }
        return 0; //retornar sucesso
    }
    else //eu não sou o máximo
        return 1;
}
```

2.

Parcialmente correcto significa **NÃO** provar que o ciclo termina (variante).

- Descobrir o invariante
  A pós-condição (que é o cálculo-objectivo do programa) dá sempre pistas acerca do invariante!!
  Fazer uma tabela com as variáveis e para um exemplo, verificar a relação entre elas e a pós-condição.

i	a	s	n	$n*s$
0	**5**	0	**3**	*15*
1	"	5	"	"
2	"	10	"	"
3	"	15	"	"

  Como se pode observar, s vai acumulando um valor que quando $i = n$, s é o resultado. Logo podemos daqui (e porque não há NENHUMA instrução entre o fim do ciclo e o fim do programa) deduzir que o invariante $I = (s = i * a \wedge i \leq n)$

- Inicialização

$$\begin{array}{ccc} \{P\} & S_{init} & \{I\} \\ \{n \geq 0\} & i = 0;\, s = 0 & \{s = i * a \wedge i \leq n\} \\ \{n \geq 0\} & \Rightarrow & \{s = i * a \wedge i \leq n\}[i\backslash 0,\, s\backslash 0] \\ \{n \geq 0\} & \Rightarrow & \{0 = 0 * a \wedge 0 \leq n\} \end{array}$$

A primeira condição é imediata. A segunda é também trivial (se n maior ou igual a 0 então 0 é de certeza menor ou igual a n).

- Preservação

$$\begin{array}{rcl}
\{I \wedge Cond_{ciclo}\} & S_{ciclo} & \{I\} \\
\{s = i * a \wedge i \leq n \wedge i < n\} & s = s + a;\, i = i + 1 & \{s = i * a \wedge i \leq n\} \\
\{s = i * a \wedge i < n\} & \Rightarrow & \{s = i * a \wedge i \leq n\}[s \backslash s + a,\, i \backslash i + 1] \\
\{s = i * a \wedge i < n\} & \Rightarrow & \{(s + a) = (i + 1) * a \wedge (i + 1) \leq n\} \\
\{s = i * a \wedge i < n\} & \Rightarrow & \{s + a = i * a + a \wedge i + 1 \leq n\} \\
\{s = i * a \wedge i < n\} & \Rightarrow & \{s = i * a \wedge i + 1 \leq n\}
\end{array}$$

A primeira condição é provada de imediato. A condição $i + 1 \leq n$ é fácil, pois se i é estritamente menor que n então se somarmos 1 será menor ou igual.

Nota Importante: *Apenas podemos fazer as atribuições simultaneamente porque não partilham variáveis. Se isso acontecesse, teria que ser uma a uma, começando pela última!!!*

- Finalização

$$\begin{array}{rcl}
\{I \wedge \neg Cond_{ciclo}\} & S_{fim} & \{Q\} \\
\{s = i * a \wedge i \leq n \wedge i \geq n\} & \Rightarrow & \{s = n * a\} \\
\{s = i * a \wedge i = n\} & \Rightarrow & \{s = n * a\} \\
\{s = n * a\} & \Rightarrow & \{s = n * a\}
\end{array}$$

Prova-se assim a correcção parcial do algoritmo.

3.

$T_{example}(n) = t_{atrib} + t_{compar} + \sum_{i=1}^{n} (t_{compar} + T_{insert}(n) + t_{atrib}) + T_{convert}(n)$

$T_{example}(n) = C_1 + n\,(C_2 + \mathcal{O}(\log n)) + \mathcal{O}(n^2)$

$T_{example}(n) = \mathcal{O}(1) + \mathcal{O}(n \log n) + \mathcal{O}(n^2)$

No pior caso, *example* tem tempo de execução $\mathcal{O}(n^2)$, pois n^2 é assimptoticamente superior a $n \log n$

4.

```
int hashCode(int key);

typedef struct cell {
   int key;
   char *val;
   struct cell *next;
} *ListaPosTabela;
typedef ListaPosTabela Tabela[MAX_N];

int insere(Tabela t, int k, char *val){
   int code = hashCode(k);
   ListaPosTabela lst = t[code];
   (struct cell *) tmp;
   if(lst==NULL){ //não contém nenhum valor
      tmp = malloc(sizeof(struct cell));
      tmp->key = k;
      tmp->val = val; //guardar só o apontador
      t[code]=tmp; //como a lista estava vazia, colocar a nova!!
      return 0; //a chave não existia na tabela
```

```
} else {
    while(lst->key!=k && lst->next!=NULL){ //percorrer a lista
        lst = lst->next;
    }
    if(lst->key==k){ //encontramos a chave, trocar o valor
        lst->val = val;
        return 1;
    }
    else { //não encontramos a chave e estamos na cauda da lista
        tmp = malloc(sizeof(struct cell));
        tmp->key = k;
        tmp->val = val;
        return 0;
    }
}
```

5.

$$T(n) = \begin{cases} c_1 & se\ n \leq 1 \\ c_2 + 5T(\frac{n}{3}) & se\ n > 1 \end{cases}$$

N.º de níveis da árvore: $\frac{n}{3^k} = 1 \Leftrightarrow n = 3^k \Leftrightarrow k = \log_3 n$ ou seja, níveis desde 0 até $(\log_3 n) - 1$.

N.º de nodos em cada nível: começa com um e aumenta 5 vezes em cada nível: 5^i, com i a começar em 0.

Tempo de EXTRA de execução de cada nodo: c_2. Tempo de execução das FOLHAS: c_1

$T(n) = tempo\_nodos + tempo\_folhas = \left(\sum_{i=0}^{(\log_3 n)-2} 5^i \times c_2 \right) + 5^{(\log_3 n)-1} \times c_1$

$T(n) = \frac{c_2}{4}(5^{(\log_3 n)-1} - 1) + c_1 5^{(log_3 n)-1} = \frac{c_2}{4} 5^{(\log_3 n)-1} + c_1 5^{(\log_3 n)-1} - \frac{c_2}{4}$

$T(n) = (c_1 + \frac{c_2}{4}) 5^{(\log_3 n)-1} - \frac{c_2}{4}$

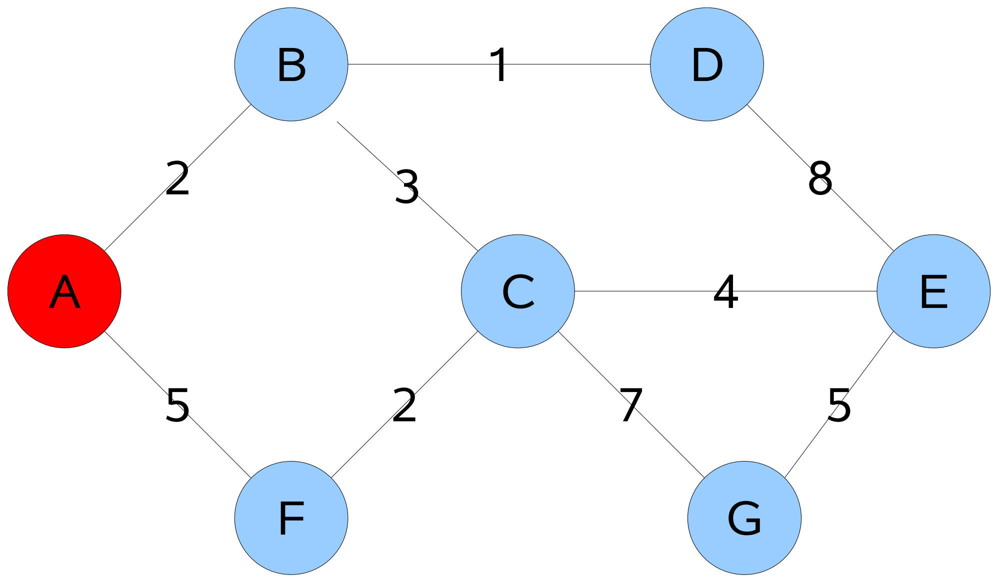
NOTA: Regra do somatório usada: $\sum_{i=0}^{n-1} a^i = \frac{a^n - 1}{a - 1}$

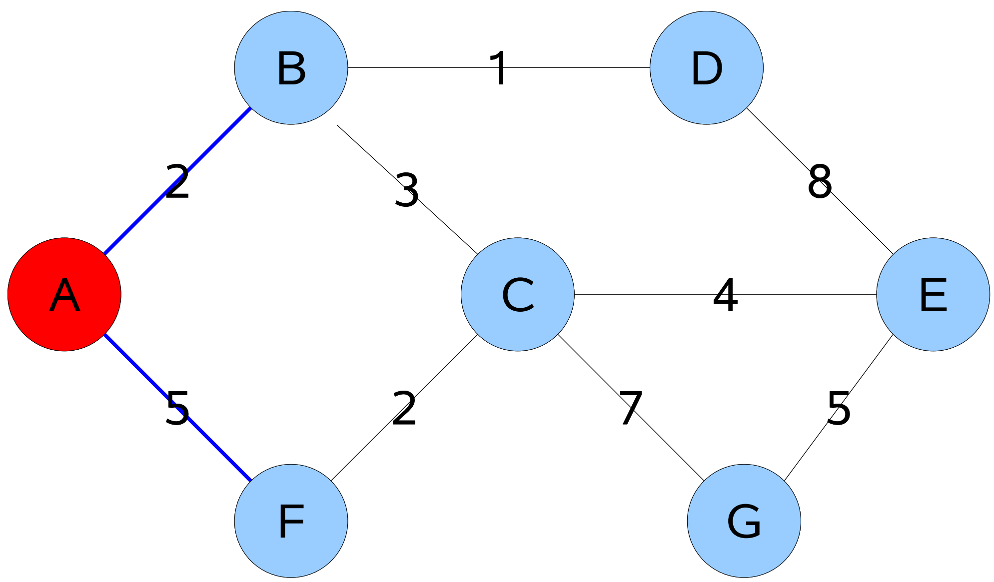
6.

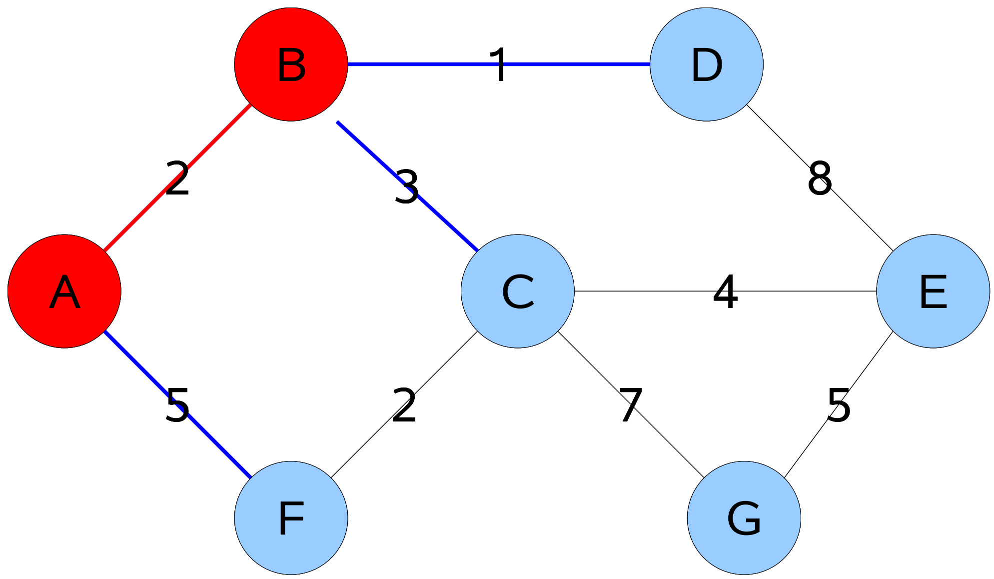
Apresenta-se a seguir os passos de construção da *Árvore Geradora de Custo Mínimo.*

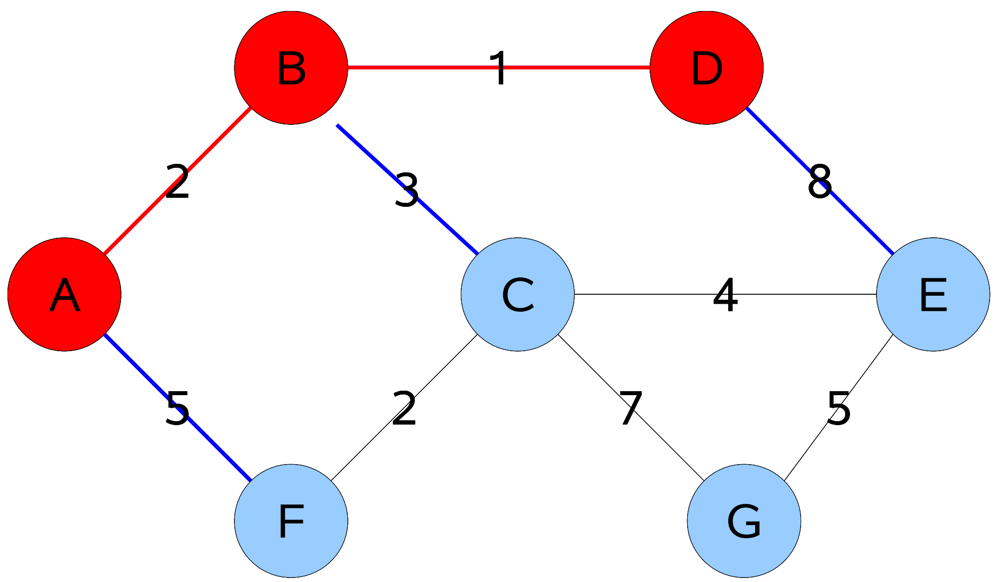
Os nodos e arcos da árvore de resultado são indicados a vermelho; os arcos candidatos são os indicados a azul QUE ligam um nodo da árvore de resultado a um outro que NÃO faz parte da árvore de resultado; estes últimos são os nós da orla.

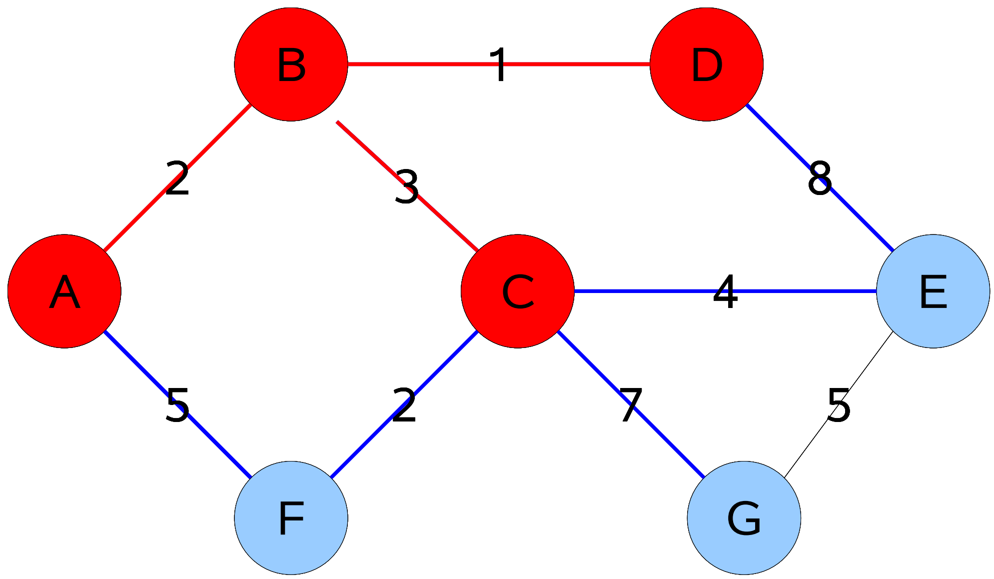
O algoritmo de Prim dentre os arcos candidatos (segundo a def. acima), acrescenta o arco de menor peso e o nodo ao qual dá ligação à árvore de resultado.

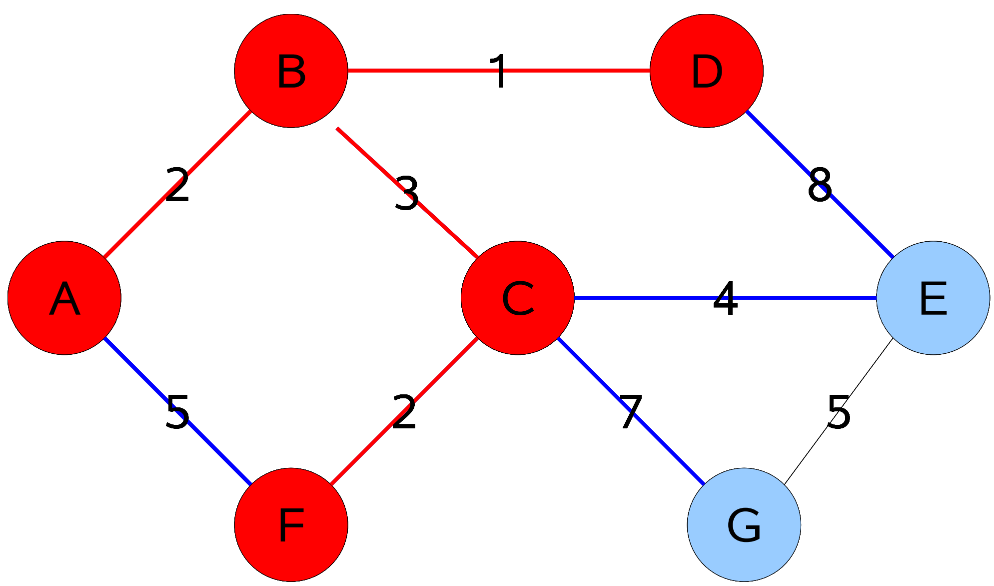

B
1
D
2
3
8
A
C
4
E
5
2
7
5
F
G

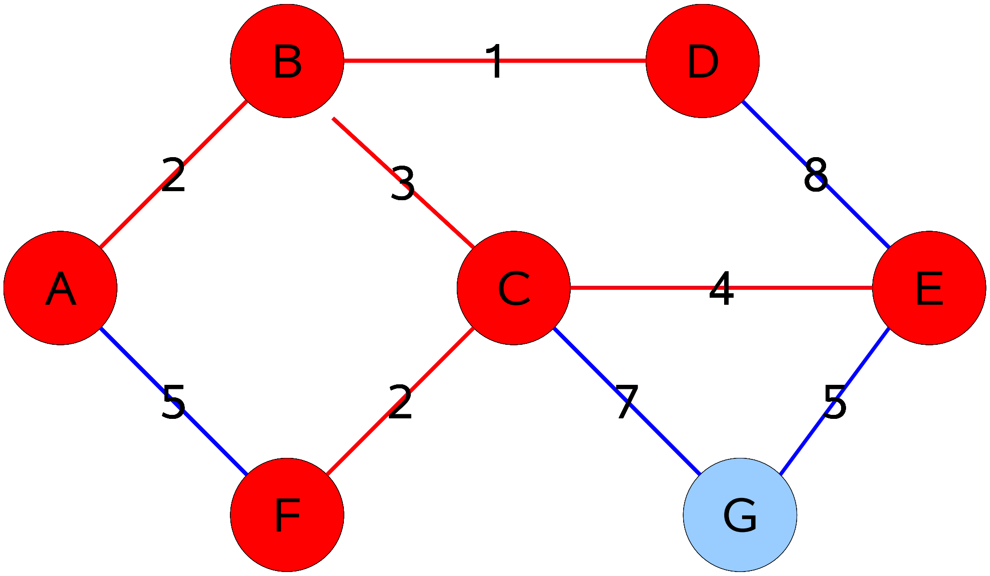

B
1
D
2
3
8
A
C
4
E
5
2
7
5
F
G


B
D
A
C
E
F
G
1
2
3
8
4
5
2
7
5

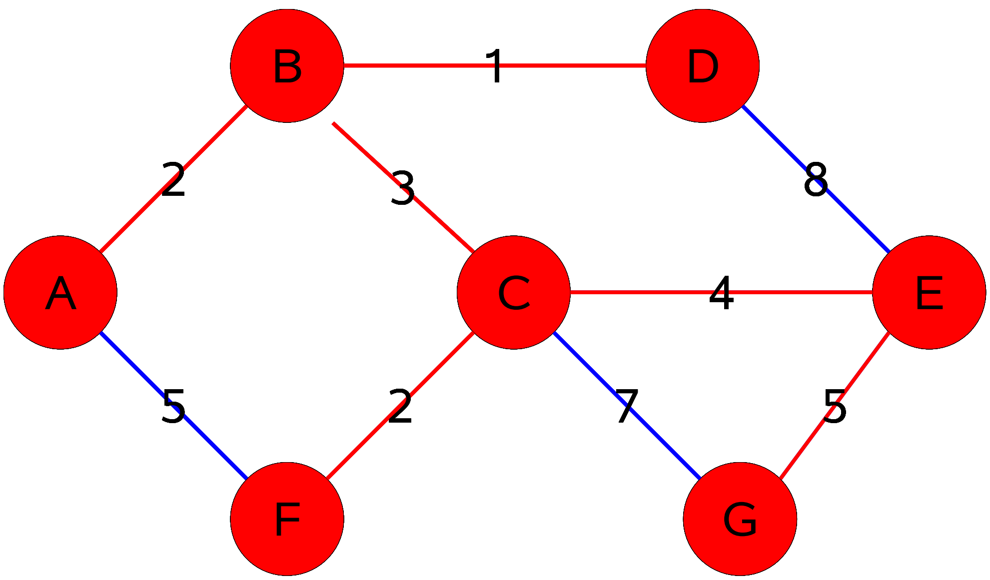

B
D
A
C
E
F
G
1
2
3
8
4
5
2
7
5


B
1
D
2
3
8
A
C
4
E
5
2
7
5
F
G

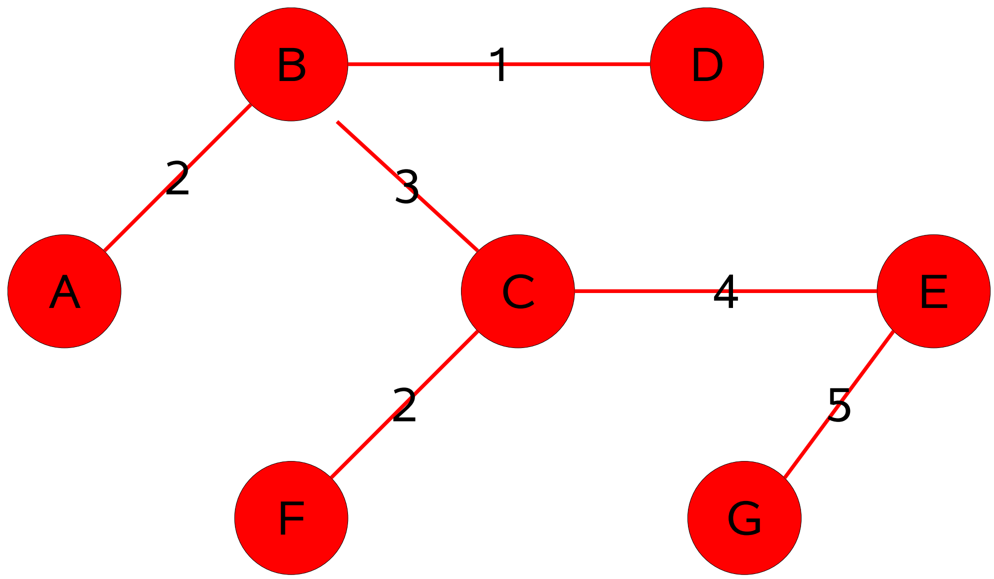

B
1
D
2
3
8
A
C
4
E
5
2
7
5
F
G



O arco escolhido vai ser o arco $E \rightarrow G$ pois é o arco candidato com menor peso. O arco $A \rightarrow F$ deixou de ser uma possibilidade a partir do momento em que tanto A como F já fazem parte da árvore de resultado.



Esta é a árvore de menor custo:


B
1
D
2
3
A
C
4
E
2
5
F
G