Universidade do Porto L:EIC

Desenho de Algoritmos

2021/22

Exame (16.07.2022) - Parte Prática

90 minutos

Uma resolução

1. a)

```
int distrGreedy(int n, int m, int c, int d[], int sol[])
{
  unsigned int j, k, jmax;
  int maxLoad=0, seats[m];

  // initialize number of free places in each bus
  for(j = 0; j < m; j++)  seats[j] = c;

  // assign groups to buses
  for(k=0; k<n; k++){

    for(j=0; j< m && seats[j] < d[k]; j++);

    if (j == m) return -1;   // cannot find a bus for group k

    // bus with the largest number of free places
    jmax = j;
    for(++j; j < m; j++)
      if (seats[j] > seats[jmax])
        jmax = j;

    // assign group k to bus jmax
    sol[k] = jmax;
    seats[jmax] -= d[k];   // update available seats
    if (c-seats[jmax] > maxLoad)
      maxLoad = c-seats[jmax];     // update max load
  }

  return maxLoad;
}
```

b) A complexidade temporal é $O(nm)$, sendo $\Theta(nm)$ no pior caso, o qual acontece quando consegue atribuir lugar a todos os grupos, porque, para cada `k`, a procura de lugar tem complexidade $\Theta(m)$ e corresponde à complexidade de cada iteração do bloco do ciclo "`for(k...`".

Observação: É possível ter uma implementação com complexidade $O(n \log m)$ se `seats` for suportada por, por exemplo, uma fila de prioridade (*heap* de máximo, com operação `decreaseKey`).

c) Para algumas instâncias tal estratégia *greedy* falha pois:

- pode não encontrar solução embora exista solução: $m = 2$, $n = 5$, $c = 9$ e $d = [6, 5, 3, 2, 2]$.

  Fica com $(6+2; 5+3)$ e não consegue atribuir o último grupo. Mas existia solução: $(6+3; 5+2+2)$.

- pode encontrar uma solução que não é ótima: $m = 2$, $n = 5$, $c = 14$ e $d = [6, 5, 4, 4, 3]$

  Obtém $(6+4; 5+4+3)$ com `maxLoad` $= 12$ mas a solução ótima é $(6+5; 4+4+3)$ com `maxLoad` $= 11$.

d) Seja L o valor de `maxLoad` da solução *greedy*. Podemos usar L para reduzir a árvore de pesquisa. Em *branch-and-bound*, podemos acrescentar no início a restrição $z \leq L - 1$, sendo z a variável que majora a carga máxima da solução (ver **1f)**). Na pesquisa com retrocesso (*backtracking*), o valor $L - 1$ pode ser usado para podar a árvore de pesquisa, evitando uma descida que não levaria a uma solução melhor se a carga da *solução parcial* (formada pelas escolhas já efetuadas) for maior ou igual a L.

e) No problema de decisão PARTITION é dado um conjunto $S = \{a_0, a_1, \ldots, a_{n-1}\} \subseteq \mathbb{Z}^+$, e é necessário decidir se S contém um conjunto A tal que $\sum_{x \in A} x = \sum_{y \in S \setminus A} x$.

Seja EXCURSION o problema de otimização enunciado. Dada uma instância de PARTITION, podemos construir em tempo polinomial uma instância de EXCURSION com:

$$m = 2 \qquad c = \sum_{i=0}^{n-1} a_i \qquad d_k = a_k, \ \text{para } 0 \leq k < n$$

Esta instância tem solução (pois podemos colocar todos os grupos no mesmo veículo) e o ótimo (i.e., o valor mínimo de `maxLoad`) é $c/2$ se e só se resposta para PARTITION é "sim". Esta redução mostra que EXCURSION é NP-difícil (*NP-hard*), pois PARTITION é NP-Completo.

Um algoritmo polinomial que resolvesse EXCURSION podia ser usado para decidir PARTITION porque, depois de calcular o ótimo, bastaria verificar se é igual a $c/2$ ou se é superior.

Assim, se P $\neq$ NP, tal algoritmo não pode existir pois se se encontrar um algoritmo polinomial para resolver algum problema NP-completo então P $=$ NP.

f)

Variáveis de Decisão

- $x_{kj} \in \{0, 1\}$, com $0 \leq j < m$ e $0 \leq k < n$, indica se o grupo k fica ou não no veículo j
- $z \in \mathbb{Z}_0^+$ indica um limite superior para a carga máxima.

Modelo matemático:

minimizar z
sujeito a

$$\begin{cases} z \leq c & \text{a carga máxima não excede a capacidade de nenhum veículo} \\[2ex] \displaystyle\sum_{k=0}^{n-1} d_k x_{kj} \leq z, \quad \text{para } 0 \leq j < m & \text{a carga máxima é maior ou igual que a carga de cada veículo} \\[2ex] \displaystyle\sum_{j=0}^{m-1} x_{kj} = 1, \quad \text{para } 0 \leq k < n & \text{cada grupo fica exatamente num veículo} \\[2ex] x_{kj} \in \{0, 1\}, \quad \text{para } 0 \leq j < m \text{ e } 0 \leq k < n & \\[2ex] z \in \mathbb{Z}_0^+ & \end{cases}$$

As restrições não definem z como o valor da carga máxima. Apenas o definem como um majorante da carga máxima. Mas, como se requer que a solução minimize z, o valor de z no ótimo é igual à carga máxima para a solução (alguma das restrições "$\leq$" é satisfeita como igualdade).

2. a)

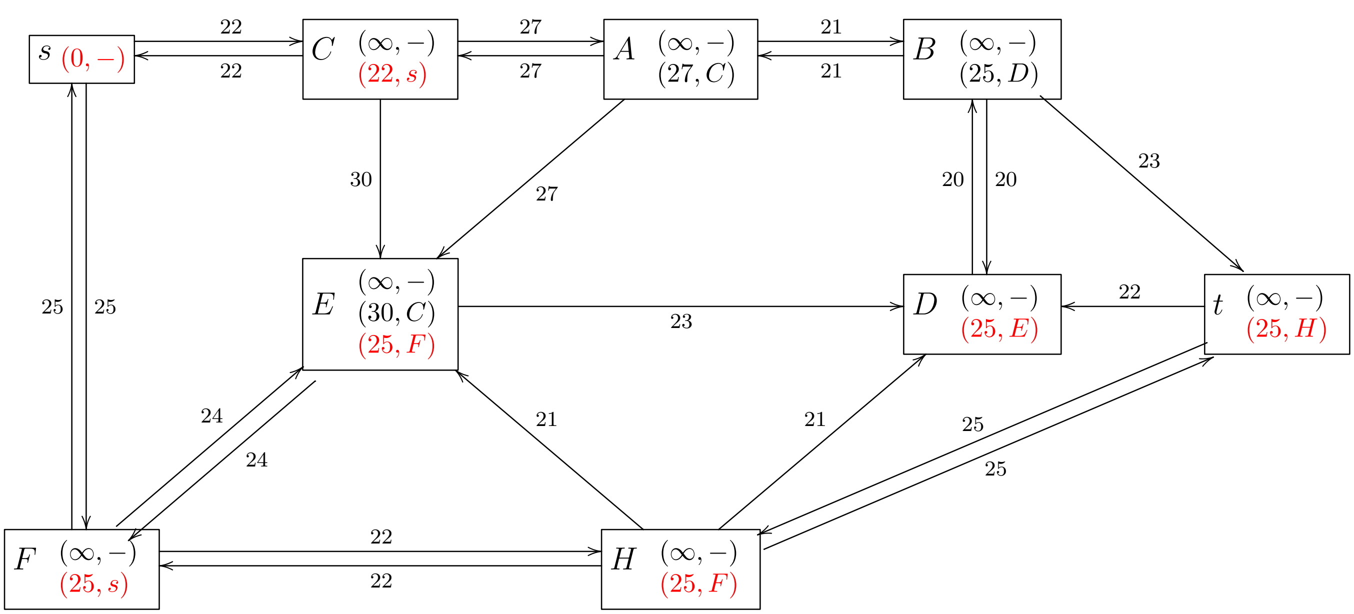
```
CaminhoMinTempMax(G, s, t, temp, pai)
    Para cada v ∈ G.V fazer { pai[v] ← NULL; temp[v] ← ∞;}
    temp[s] ← 0;
    Q ← Mk_PQ_HeapMin(temp, V);
    Enquanto (PQ_Not_Empty(Q)) fazer
        v ← ExtractMin(Q);
        Se (v = t) então retorna;
        Para cada w ∈ G.Adjs[v] fazer
            Se max(temp[v], G.T(v, w)) < temp[w] então
                temp[w] ← max(temp[v], G.T(v, w));
                pai[w] ← v;
                DecreaseKey(Q, w, temp[w]);
```

Assume-se que os parâmetros $temp$ e pai são *arrays* e que os nós em V foram previamente numerados.

b) A complexidade temporal é $O((n+m)\log n)$, sendo $n = |V|$ e $m = |E|$, se a fila de prioridade for suportada por uma *heap de mínimo*, em que as operações ExtractMin e DecreaseKey são realizadas em $O(\log size)$, sendo $size$ o número de elementos na *heap*. Para $m > n$, podemos escrever que é $O(m \log n)$.

À semelhança do algoritmo de Dijkstra, este algoritmo combina uma abordagem *greedy*, por em cada iteração explorar o nó v que tem o valor mínimo de $temp$ entre os que estavam na fila, e de *programação dinâmica*, pela forma como vai mantendo em $temp[v]$ o melhor valor encontrado para v, que é o ótimo quando considerados caminhos que só podem passar por nós já explorados (i.e., que já saíram da fila).

c)



Ordem pela qual os nós saíram da heap: s, C, F, E, H, D, t. Os valores nos nós mostram as atualizações (a vermelho, o final). Supusemos que, quando têm a mesma chave (i.e., mesmo valor de $temp$), o primeiro a sair da *heap* é o que tinha o valor há mais tempo (na implementação, poderiamos ter outro critério, por exemplo, desempate por ordem alfabética). Os nós A e B ficaram na fila, por t ter saído antes.

Observação: Para este algoritmo, como para o algoritmo de Dijkstra, pode-se provar que o valor de $temp[v]$ é o mínimo (i.e., é ótimo) para os nós v que estão na *heap* e têm $temp$ mínimo. Portanto, é correto retirar qualquer um deles.