Departamento de Ciência de Computadores — FCUP

Algoritmos (CC4010) — 2020/2021

Teste (13.01.2021) Cotação: 4.0, 1.5, 3.0 , 4.0, 6.0, 1.5 *duração: 2h (+30')*

Uma resolução de questões selecionadas (não inclui 4 e 6)

1. Recorde o problema *unit task scheduling*, de calendarização de n tarefas unitárias com penalização total mínima. Admita que d_i, com $1 \leq d_i \leq n$, é o prazo limite para execução da tarefa i e p_i o montante a pagar se a executar após esse prazo. As tarefas ficam concluídas no dia em que são executadas, só podendo executar uma tarefa em cada dia k, com $1 \leq k \leq n$.

a) [1.0] **Enuncie** e **justifique sucintamente** o critério dado nas aulas para decidir *se é possível* calendarizar, sem penalização, um conjunto S de tarefas (dado S).

Critério: É possível se e só se o número de tarefas $N_k(S)$ com prazo limite até k em S for menor ou igual a k, qualquer que seja $k \geq 1$. **Justificação:** Se $N_1(S) \leq 1$ então existe no máximo uma tarefa com prazo limite 1, a qual, se existir, pode ser excutada no *slot* 1. Se $N_{k-1}(S) \leq k-1$ e conseguirmos alocar sem penalização as tarefas com prazo até $k-1$, então restam $k-1-N_{k-1}(S)$ *slots* até $k-1$. Sendo $N_k(S) - N_{k-1}(S)$ o número de tarefas com prazo k e $N_k(S) \leq k$, tem-se $N_k(S) - N_{k-1}(S) \leq k - N_{k-1}(S)$ e existem $k - N_{k-1}(S)$ slots livres até k, que podem ser usados para alocar as tarefas com prazo k. Logo, se $N_k(S) \leq k$, para todo $k \geq 1$, podemos alocar sem penalização todas as tarefas de S. Tal não é possível se $N_k(S) > k$, para algum k, pois não teriamos *slots* suficientes para realizar as tarefas com prazo até k em S.

b) [2.0] Apresente os passos principais de um **algoritmo** com complexidade temporal $O(n^2)$ que determine uma solução ótima para *unit task scheduling*. Em caso de empate, optará pela tarefa com **identificador menor**.

- Ordenar as tarefas por ordem decrescente de penalização (em caso de empate, colocar primeiro a que tem menor identificador). Seja $T = \{t_1, \ldots, t_n\}$ o conjunto ordenado.
- Seja C o *array* de n slots (inicialmente livres, i.e., $C[i] = 0$, para todo i). Seja P a penalização total (inicialmente 0).
- Para j de 1 até n, tentar alocar t_j o mais tarde possível sem ultrapassar o seu prazo i.e., procurar $C[i] = 0$ para i desde $d[t_j]$ até 1. Se conseguir, definir $C[i] = t_j$. Se não, acrescentar $p[t_j]$ à penalização P e colocar t_j no último slot livre (procedendo de modo idêntico, a partir do slot n).

Considerando que a ordenação pode ser realizada em $O(n \log n)$, a calendarização domina a complexidade, conduzindo a $\Theta(n^2)$ se, por exemplo, todas as tarefas tiverem prazo n. Nesse caso, encontrar o slot livre para a tarefa j teria complexidade $\Theta(j)$ e $\sum_{j=1}^{n} \Theta(j) = \Theta(n^2)$. Nas aulas, foram dadas versões com complexidade amortizada menor.

c) [1.0] Ilustre a aplicação do algoritmo à instância seguinte.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
d_i	5	4	4	2	1	5	3	7	2	3	1	8	5	8
p_i	2	3	2	1	3	5	5	2	5	5	3	1	1	2

Apresente a **ordem** *pela qual as tarefas são calendarizadas* pelo algoritmo que apresentou, a **calendarização** que produziu (para as 14 tarefas) e a **penalização** total.

Ordem: $6, 7, 9, 10, 2, 5, 11, 1, 3, 8, 14, 4, 12, 13$

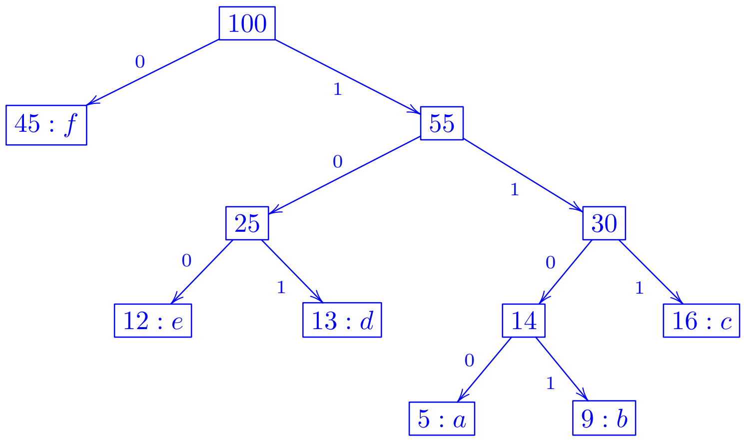
Calendarização:

10	9	7	2	6	12	8	14	13	4	3	1	11	5

Penalização: $3 + 3 + 2 + 2 + 1 + 1 = 12$

Em "unit talk scheduling", as tarefas realizadas após o prazo limite podem ser realizadas por qualquer ordem, mas o algoritmo apresentado define um slot no momento em que analisa a tarefa, sendo esta a solução produzida.

2. [1.5] Suponha que temos um ficheiro com 100000 caracteres, em que ocorrem apenas os caracteres `a`, `b`, `c`, `d`, `e`, `f`, com as frequências seguintes (em milhares): `a` - 5, `b` - 9, `c` - 16, `d` - 13, `e` - 12, `f` - 45. Indique os códigos que resultam da aplicação do algoritmo de Huffmann para compressão. Apresente a árvore (de prefixos) que constrói (indicando os valores nos nós e nos ramos e o seu significado).



Códigos
a : 1100
b : 1101
c : 111
d : 101
e : 100
f : 0

Os valores nos nós internos representam a soma das frequências dos nós filhos, podendo ser interpretados como a frequência de um carater que substituisse todos os carateres nas folhas da sub-árvore que tem raíz nesse nó. Os valores nos ramos são usados para construir os códigos. O código de um caracter é obtido por concatenação dos valores nas arestas do caminho da raíz da árvore até à folha que contém tal caracter.

3. [1.5+1.5] O Clay Mathematics Institute atribuirá um prémio de 1 milhão de dólares a quem resolver o problema **"P = NP?"**. Admita que alguém descobria um algoritmo polinomial que, dada uma qualquer instância de TSP, com n nós e peso $d(u,v) \in \mathbb{Z}^+$ para o ramo (u,v), determinava um ciclo de Hamilton C com $d(C) \leq n^2 d(C^\star)$, sendo $C^\star$ um ciclo de Hamilton com peso mínimo. Explique porque é que ganharia o prémio. Recordando que o algoritmo de Christofides garante um fator de aproximação $3/2$, não serviria?

- O problema TSP consiste na determinação de um ciclo de Hamilton $C^\star$ com peso total mínimo num grafo $G = (V,E,d)$ não dirigido completo, com pesos nos ramos. Foi provado nas aulas a inaproximabilidade de TSP por qualquer algoritmo polinomial que garanta um fator de aproximação $\alpha(n)$, qualquer que seja a função $\alpha(n)$ computável em tempo polinomial, a menos que P=NP. Isso significa que se existir um algoritmo polinomial que, dada uma instancia qualquer de TSP, produz um ciclo de Hamilton C com $d(C) \leq \alpha(n)d(C^\star)$, então P=NP. Como a função $\alpha(n) = n^2$ é computável em tempo polinomial, se descobrir o algoritmo referido, teria a prova de que P=NP. Pelo que, ganharia o prémio.
- O algoritmo de Christofides não se aplica a instâncias genéricas, mas apenas a instâncias que satisfazem a desigualdade triangular, $d(x,y) \leq d(x,z) + d(z,y)$, paa todos $x,y,z \in V$. Assim, não serviria, apesar de satisfazer a condição $d(C) \leq n^2 d(C^\star)$, se $n \geq 2$.

4. Sejam $A_1, A_2, \ldots, A_n$ matrizes de inteiros, tendo A_k dimensão $d_k \times d_{k+1}$, i.e., d_k linhas e d_{k+1} colunas), para $1 \leq k \leq n$. Pretendemos calcular o produto $A_1 A_2 \cdots A_n$. Será usada a **definição usual de produto de matrizes**, mas queremos minimizar o número total de multiplicações (de inteiros) efetuadas.

Note que, por exemplo, para $A_1 : 2 \times 3$, $A_2 : 3 \times 5$ e $A_3 : 5 \times 2$, se usarmos $A_1(A_2A_3)$ efetuamos $30 + 12 = 42$ multiplicações e se usarmos $(A_1A_2)A_3$ efetuamos $30 + 20 = 50$. Recorde que se $C = A_1A_2$ então C tem dimensão $d_1 \times d_3$ e no cálculo de $C[i,j] = \sum_{p=1}^{d_2} A_1[i,p] \times A_2[p,j]$ efetuamos d_2 multiplicações (de inteiros), para $1 \leq i \leq d_1$, $1 \leq j \leq d_3$.

Para isso, exploramos o facto de o produto de matrizes ser associativo. Por exemplo, para $A_1A_2A_3A_4$, teriamos de considerar quatro possibilidades $A_1(A_2(A_3A_4))$, $A_1((A_2A_3)A_4)$, $(A_1A_2)(A_3A_4)$, $(A_1(A_2A_3))A_4$, $((A_1A_2)A_3)A_4$. Em geral, o número de casos é exponencial. Mas, podemos calcular uma solução ótima usando **programação dinâmica**.

N.º | Nome

Apresente **um algoritmo**, em pseudocódigo, que use **programação dinâmica** para resolver o problema. Deve calcular o **mínimo** $M[k,m]$ para $A_k A_{k+1} \ldots A_m$, com $1 \leq k \leq m \leq n$, e o **índice** $P[k,m]$ que indica como partir. $P[k,m] = s$ corresponderia a $(A_k A_{k+1} \ldots A_s)(A_{s+1} \ldots A_m)$. Qual é a sua complexidade temporal e espacial (admitindo que os valores podem ser calculados em $O(1)$)? Justifique sucintamente.

Resposta omitida

O programa pode seguir a recorrência:

- $P[k,k] = M[k,k] = 0$, para $1 \leq k \leq n$;
- Para $1 \leq k < m \leq n$, definir

$$M[k,m] = \min_{k \leq s < m} (M[k,s] + M[s+1,m] + d_k d_{s+1} d_{m+1})$$
$$P[k,m] = \text{escolher } s \text{ com } M[k,s] + M[s+1,m] + d_k d_{s+1} d_{m+1} \text{ mínimo}$$

Para uma abordagem iterativa (bottom-up), preenchemos as matrizes por diagonais: na iteração i, calculamos os elementos nas posições $(k, k+i)$, com $1 \leq k \leq k+i \leq n$.

Em alternativa, podemos implementar uma versão top-down, com recursão e memoização, começando por definir todas as entradas de M com valor -1 (o que indicaria que ainda não estava determinada).

Complexidade temporal $\Theta(n^3)$ e complexidade espacial $\Theta(n^2)$.

5. No problema **Load balancing** existem n tarefas que terão de ser realizadas por máquinas do mesmo tipo. Cada tarefa é processada sem interrupções por uma máquina. Cada máquina só pode estar a realizar uma tarefa em cada instante. A tarefa j tem duração d_j, para $j = 1, \ldots, n$. Existem m máquinas idênticas. As tarefas podem ser realizadas por qualquer ordem. Pretendemos atribuir tarefas às máquinas de modo a *minimizar a carga máxima* L das máquinas, definida por $L = \max_i C[i]$, sendo $C[i]$ a soma das durações das tarefas atribuídas à máquina i, para $i = 1, \ldots, m$. Admita que os valores d_j são inteiros positivos.

a) [0.8] Justifique que qualquer que seja a instância se tem $L^\star \geq \max_j d_j$ e $L^\star \geq \frac{1}{m}\sum_j d_j$, sendo $L^\star$ o ótimo.

- $L^\star \geq \max_j d_j$ pois alguma das máquinas terá de realizar a tarefa que tem duração máxima.
- $L^\star \geq \frac{1}{m}\sum_j d_j$ pois, no melhor caso, teriamos a carga total distribuída de forma idêntica pelas m máquinas. Cada máquina ficaria com $\frac{1}{m}\sum_j d_j$ mas tal pode obrigar a fracionar tarefas, o que não é possível. Logo, $L^\star \geq \frac{1}{m}\sum_j d_j$.

b) [1.5] No problema **Partition**, que é NP-completo, são dados n inteiros $a_1, \ldots, a_n$ e pretendemos saber se existe um subconjunto J de $\{1, \ldots, n\}$ tal que $\sum_{j \in J} a_j = \sum_{j \notin J} a_j$. Usando **Partition**, prove que o problema de decisão correspondente a *Load Balancing* é **NP-completo**, para $m = 2$.

No problema de decisão correspondente a *Load Balancing*, dado (I, q), onde I uma instância de *Load Balancing* e $q \in \mathbb{Q}^+$, há que decidir se é possível atribuir as tarefas às máquinas de modo que a carga máxima $L \leq q$. Se existir um algoritmo de decisão polinomial para este problema, podemos usá-lo para decidir PARTITION, porque podemos definir uma instância I de *Load Balancing-Decision*, com $d_j = a_j$, para todo j, e $q = \frac{1}{2}\sum_j a_j$. Se aplicarmos o algoritmo a esta instância e a resposta for "yes", responderiamos "yes" para Partition; caso contrário, respondiamos "no". Esta redução é polinomial.

Assim, *Load Balancing-Decision* é NP-hard. Por outro lado, *Load Balancing-Decision* pertence a NP pois, se for dada uma qualquer instância "Yes" (I, q) e a prova de que a resposta é "Yes" (ou seja, uma distribuição das tarefas pelas máquinas que satisfaça $L \leq q$), podemos verificar polinomialmente essa prova. Logo, *Load Balancing-Decision* é NP-completo.

Se definirmos $q \in \mathbb{Z}^+$ em vez de $q \in \mathbb{Q}^+$, então, para efetuarmos a redução, bastaria definir $d_j = 2a_j$ e $q = \sum_j a_j$.

c) Considere o algoritmo apresentado à direita, suportado por uma *heap de mínimo*, em que, na operação INCREASEKEY, a localização do nó correspondente à máquina k pode ser efetuada em $O(1)$.

1. [1.5] Indique a complexidade temporal do algoritmo. Justifique sucintamente.

2. Sabendo que **Load-Balancing** é NP-hard e assumindo **P≠NP**, comente a veracidade das afirmações, onde L é o **valor obtido** pelo algoritmo para a instância e $L^\star$ o **ótimo** correspondente.

   (i) [1.0] Para alguma instância (d, n, m), tem-se $L \neq L^\star$.

   (ii) [0.5] Qualquer que seja a instância, $L \geq (1+\varepsilon)L^\star$, para algum $\varepsilon > 0$.

```
LOADBALANCING(d, n, m, S, C)
1.   L = 0
2.   for i = 1 to m
3.       C[i] = 0
4.   Q = MAKEHEAPMIN(C, m)
5.   for j ← 1 to n
6.       k = EXTRACTMIN(Q)
7.       S[j] = k
8.       C[k] = C[k] + d[j]
9.       INCREASEKEY(Q, k, C[k])
10.      if C[k] > L then L = C[k]
11   return L
```

- A complexidade é $O(m + n\log_2 m)$ e, se admitirmos $n > m$, seria $O(n\log_2 m)$. Nesta análise, assumimos a implementação descrita nas aulas para *heaps*. A construção da *heap* (na linha 4), pode ser efetuada em $\Theta(m)$ e o ciclo 1-2 também. Portanto, o bloco 1-4 tem complexidade $\Theta(m)$. As operações de extração e de incremento da chave têm complexidade $O(\log_2 size)$, em que $size$ é o número de elementos que ainda estão na *heap*. Assim, o custo de cada operação desse tipo é $O(\log_2 m)$, já que, no máximo, temos m elementos na heap. Logo, o ciclo 5-10 tem complexidade $O(n\log_2 m)$.
- (i) Verdade. Tem que existir alguma instância com $L \neq L^\star$ pois, caso contrário, i.e., se $L = L^\star$ para todas as instâncias (i.e., se o algoritmo determinasse sempre a solução ótima), então podia ser usado para resolver *Load-Balancing Decison*. Como o problema é NP-hard, concluiamos que P=NP, em contradição com a hipótese.
- (ii) Falso. Se $L \geq (1+\varepsilon)L^\star$, para algum $\varepsilon > 0$, então $L > L^\star$. Mas, por exemplo, para as instâncias em que todas as tarefas têm duração unitária, o algoritmo produz a solução ótima. Portanto não é verdade, que produza sempre uma solução pior do que a ótima.

Por gralha, no enunciado distribuído estava "Qualquer que seja a instância, $L \geq (1+\varepsilon)L^\star$, para algum $\varepsilon > 1$". A mesma justificação serviria mas, de facto, a questão era trivialmente falsa, considerando que se pedia, na questão 5d), para mostrar que o algoritmo garante uma aproximação de fator 2. Logo, teriamos $L \leq 2L^\star$, para **todas** as instâncias. Assim, como $L \geq (1+\varepsilon)L^\star$, com $\varepsilon > 1$, implica $L > 2L^\star$, então **nenhuma instância** satisfaz tal condição.

d) [0.7] Seja L o valor retornado pelo algoritmo. Seja k' uma máquina que fica com carga L na solução (ou seja, $L = C[k']$). Seja j' a última tarefa atribuída à máquina k'. Tendo em conta a estratégia *greedy* que o algoritmo implementa, justifique que $L - d_{j'} \leq C[i]$, para todas as máquinas i, considerando as cargas no momento em que processou j' no algoritmo e as cargas finais. Usando esse facto e **5a)**, conclua que, na linha 11, se tem $L - d_{j'} \leq \frac{1}{m}\sum_i C[i] = \frac{1}{m}\sum_i d_i \leq L^\star$, e que o algoritmo apresentado produz uma aproximação de fator 2, pelo que **Load Balancing** pertence à classe APX.

Sejam $C_0[i]$ e $C_f[i]$ as cargas da máquina i no instante em que escolhe k' para executar a tarefa j' e no fim.

Por definição de j' e k', tem-se $C_f[k'] - d_{j'} = C_0[k']$, sendo $L = C_f[k']$. Como o algoritmo escolhe a máquina com carga mínima para executar j', então $C_0[k'] \leq C_0[i]$, para qualquer máquina $i \neq k'$ (e também para k'). E, como $C_0[i] \leq C_f[i]$, para todo i, então

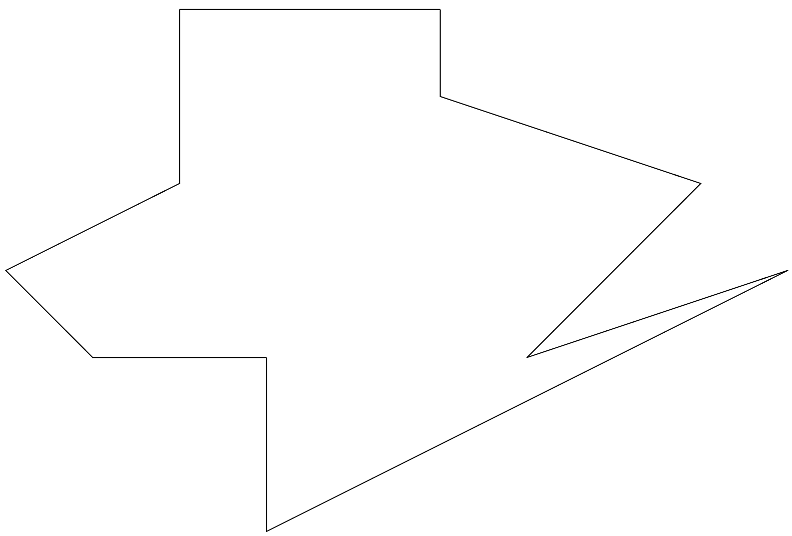
$$L - d_{j'} = C_f[k'] - d_{j'} \leq C_0[i] \leq C_f[i],$$

para todo i. Adicionando todos os termos, obtém-se $m(L - d_{j'}) = \sum_{i=1}^{m}(C_f[k'] - d_{j'}) \leq \sum_{i=1}^{m} C_0[i] \leq \sum_{i=1}^{m} C_f[i] = \sum_{j=1}^{n} d_j$. Dividindo por m e usando 5a), conclui-se que $L - d_{j'} \leq \frac{1}{m}\sum_{i=1}^{m} C_0[i] \leq \frac{1}{m}\sum_{i=1}^{m} C_f[i] = \frac{1}{m}\sum_{j=1}^{n} d_j \leq L^\star$. Logo, $L - d_{j'} \leq L^\star$ e, como $d_{j'} \leq L^\star$, conclui-se que $L \leq 2L^\star$. Portanto, o algoritmo garante uma aproximação de fator 2 e, como é polinomial, tal signfica que o problema pertence à classe APX.

N.º | Nome

6. [1.3+0.2] Usando a figura, explique a prova de Fisk para o teorema de Chvátal, de que bastam $\lfloor n/3 \rfloor$ para vigiar qualquer polígono simples com n vértices.

Sabendo que os vértices são $(3,0), (9,3), (6,2), (8,4), (5,5), (5,6), (2,6), (2,4), (0,3), (1,2), (3,2)$, indique qual seria o mínimo para esta instância.



Resposta omitida

A triangulação é uma partição do polígono, definida por diagonais internas, as quais ligam vértices e não se intersectam, a não ser em vértices.

Não esquecer de indicar o mínimo.

Master theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Stirling's approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n)) = \sqrt{2\pi n}\left(\frac{n}{e}\right)^{\alpha_n}, \quad \text{with } 1/(12n+1) < \alpha_n < 1/(12n)$$

Some useful results:

$$\log(\prod_{k=1}^{n} a_k) = \sum_{k=1}^{n} \log a_k \qquad \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}, \quad \text{for } |x| < 1$$

If $(u_k)_k$ is an arithmetic progression (i.e., $u_{k+1} = r + u_k$, for some constant $r \neq 0$), then $\sum_{k=1}^{n} u_k = \frac{(u_1+u_n)n}{2}$.

If $(u_k)_k$ is a geometric progression (i.e., $u_{k+1} = ru_k$, for some constant $r \neq 1$), then $\sum_{k=1}^{n} u_k = \frac{u_{n+1}-u_1}{r-1}$.

If $f \geq 0$ is continuous and a monotonically increasing function, then

$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx$$