

CPS740 - Lista 2

Pedro Maciel Xavier
116023847

4 de agosto de 2020

Questão 1

a) O Algoritmo guloso abaixo consiste em colocar cada objeto k de peso $\mathbf{p}[k]$ e valor $\mathbf{v}[k]$ na mochila, caso esta ainda possua capacidade de armazená-lo, ou seja, não ultrapasse o peso máximo P . Ele retorna um par (V, \mathbf{x}) com o valor total da mochila e o vetor binário \mathbf{x} que representa itens escolhidos. Fazendo isso da maneira ingênua, encontramos um caso simples onde o algoritmo não funciona para encontrar a mochila de valor máximo: Basta a situação em que o algoritmo, por gula, escolhe uma combinação dentre os m primeiros objetos e temos que $\sum_{i=1}^m \mathbf{v}[i] \cdot \mathbf{x}[i] < \mathbf{v}[m+1]$ e $\mathbf{p}[m+1] \leq P$.

Diferentes abordagens podem surgir ordenando os objetos conforme o seu custo-benefício ou valor de modo a tentar inserir primeiro os candidatos segundo algum critério de prioridade. Isso, contudo, também não contorna o problema enunciado acima. Contra-exemplos: $P = 10; \mathbf{p} = \{4, 1, 10\}; \mathbf{v} = \{5, 4, 10\}$ e $P = 10; \mathbf{p} = \{4, 1, 10\}; \mathbf{v} = \{5, 6, 10\}$.

Uma descrição em pseudo-código encontra-se na próxima página.

Algoritmo 1: Algoritmo Guloso

```

1 // Insertion-sort em dois vetores
2 // segundo algum critério
3 def ordena(v[n], p[n], n):
4     para i de n até 2:
5         seja m  $\leftarrow$  critério(v[i], p[i])
6         seja k  $\leftarrow$  i
7         para j de i-1 até 1:
8             r  $\leftarrow$  critério(v[j], p[j])
9             se r > m:
10                 m  $\leftarrow$  r
11                 k  $\leftarrow$  j
12             v[i]  $\longleftrightarrow$  v[k]
13             p[i]  $\longleftrightarrow$  p[k]
14
15 def guloso(P, n, v[n], p[n]):
16     seja x[n]
17     seja V  $\leftarrow$  0
18
19     ordena(v, p, n)
20
21     enquanto n > 0:
22         se p[n]  $\leq$  P:
23             x[n]  $\leftarrow$  1
24             P  $\leftarrow$  P - p[n]
25             V  $\leftarrow$  V + v[n]
26         n  $\leftarrow$  n - 1
27
28     retorna (V, x)

```

b) O Algoritmo a seguir consiste em testar para subconjuntos cada vez menores do conjunto de objetos quais configurações de mochila são satisfatórias, escolhendo dentre estas a de maior valor. O processo se dá de maneira recursiva. Cada passo de recursão se divide ao decidir se colocamos um objeto na mochila ou não, assim como as possíveis consequências da escolha. Adicionar um item diminui a carga disponível para as próximas etapas e incrementa o valor obtido. Após cada escolha, passamos a olhar para um subconjunto contendo os objetos de índice menor do que aquele sob análise.

Cada chamada da função \mathbf{f} retorna dois números inteiros: O valor v arrecadado até o momento e um número x cuja representação binária representa as escolhas feitas até então. O i -ésimo bit deste inteiro indica se o i -ésimo objeto foi escolhido ou não. Por fim, o valor de x é decodificado em um vetor binário \mathbf{X} .

Como é de se esperar de um processo em programação dinâmica, o resultado da chamada de $\mathbf{f}(\mathbf{P}, \mathbf{n}, \dots)$ para um determinado par $[\mathbf{P}, \mathbf{n}]$ é armazenado em uma tabela de dispersão (*Hashmap*), criada especificamente para cada instância do problema (escolha de \mathbf{v} e \mathbf{p}). Isso permite que valores já calculados anteriormente sejam obtidos em tempo constante.

Casos na borda, ou seja, fora do espaço de busca, são tratados primeiro. Em seguida verifica-se a existência de um resultado existente armazenado para o par $[\mathbf{P}, \mathbf{n}]$. Por fim, se necessário, prossegue-se com o processo de recursão.

Uma descrição em pseudo-código encontra-se na próxima página.

Algoritmo 2: Programação Dinâmica

```

1  def progдина(P, n, v[n], p[n]):
2      seja H[?] // Tabela de dispersão (Hashmap)
3      seja X[n] // Vetor das escolhas
4
5      seja [v, x] ← f(P, n - 1, v, p, H)
6
7      // Transforma x em um vetor de bits X
8      para cada i de 1 até n:
9          X[i] ← x % 2
10         x ← x >> 1
11
12     retorna X
13
14 def f(P, n, v[n], p[n], H[?]):
15     se n < 0:
16         retorna [0, 0]
17
18     senão se P == 0:
19         se p[n] <= 0:
20             retorna [v[n], (1 << n)]
21         senão:
22             retorna [0, 0]
23
24     senão se P < 0:
25         retorna [-∞, 0]
26
27     senão se [P, n] em H:
28         retorna H[P, n]
29
30     senão:
31         // Incluindo o n-ésimo objeto
32         [v_i, x_i] ← f(P - p[n], n - 1, v, p, H)
33         v_i ← v_i + v[n] // Marca o n-ésimo
34         x_i ← x_i + (1 << n) // bit de x (2^n)
35
36         // Excluindo o n-ésimo objeto
37         [v_e, x_e] ← f(P, n - 1, v, p, H)
38
39         se v_i > v_e:
40             H[P, n] ← [v_i, x_i]
41         senão:
42             H[P, n] ← [v_e, x_e]
43
44     retorna H[P, n]

```

Questão 2

Neste problema, dado um conjunto T de tarefas, onde cada tarefa $t \in T$ é representada por um par $(i(t), f(t))$ com $i(t) \leq f(t)$ contendo o tempo de início e fim da mesma, queremos encontrar um planejamento de tarefas sem sobreposição de horário, para realizar o máximo número possível de tarefas. Isto é, buscamos o subconjunto $\arg \max_{S \subseteq T} |S|$ tal que:

$$]i(s_i), f(s_i)[\cap]i(s_j), f(s_j)[= \emptyset \quad \forall s_i \neq s_j \in S$$

Podemos pensar neste problema como sendo equivalente a encontrar um conjunto independente em um grafo G cujas arestas são definidas pela relação de sobreposição dos horários entre um par de tarefas. Vejamos o seguinte exemplo para $T = \{(2, 5), (11, 15), (4, 9), (7, 10)\}$. As tarefas $t_1 = (2, 5)$ e $t_3 = (4, 9)$, assim como as tarefas $t_3 = (4, 9)$ e $t_4 = (7, 10)$ apresentam sobreposição. Isso significa que as arestas do grafo G serão as do conjunto $E(G) = \{(1, 3), (3, 4)\}$.



Figura 1: Grafo G e seu complemento \overline{G}

Tarefas que não podem ser realizadas em um mesmo planejamento possuem uma aresta que as liga em G . Queremos agendar o máximo possível de tarefas que não possuem esta característica, isto é, encontrar o maior conjunto independente de vértices do grafo. A este processo equivale buscar a maior clique no complemento \overline{G} de G . No exemplo, a maior clique de \overline{G} é composta pelos vértices 1, 2 e 4.

A descrição em pseudo-código para encontrar o tamanho da clique maximal do Grafo complementar encontra-se na próxima página.

Algoritmo 3: Agendamento de tarefas

```
1 def tarefas(T[n][2], n):  
2     // Conjuntos de vértices e arestas  
3     seja V  $\leftarrow$  {v para cada v de 1 até n}  
4     seja E  $\leftarrow$  {}  
5  
6     para cada i de 1 até n:  
7         para cada j de i + 1 até n:  
8             se T[i][2]  $\leq$  T[j][1] ou T[j][2]  $\leq$  T[i][1]:  
9                 E  $\leftarrow$  E  $\cup$  {(i, j)}
```

Questão 3

- a) Existem duas árvores geradoras mínimas com $\sum_{e \in E_T} d(e) = 14$.

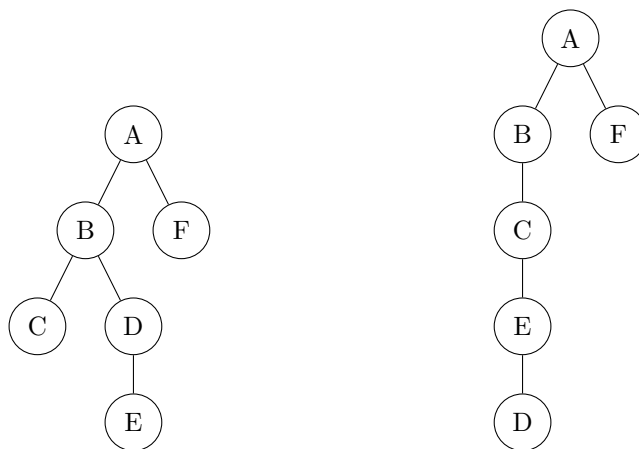


Figura 2: Árvores geradoras mínimas

- b) A aresta (F, D) não pode compor uma árvore geradora mínima porque existe um caminho entre F e D cuja soma das arestas é inferior a $d((F, D)) = 10$. Por exemplo, o caminho (F, A, B, D) , tem soma das arestas $1 + 1 + 4 = 6$.
- c) $\chi(G) = 2$. Não é possível colorir o grafo com uma só cor, visto que ele é conexo e é composto por mais de um vértice. No entanto, o grafo é bipartido em $V_1 = \{A, C, D\}$ e $V_2 = \{B, E, F\}$ e, portanto, pode ser pintado com duas cores somente.

Questão 4

a) Em uma matriz de adjacências, verificar a existência de uma aresta (v, w) significa simplesmente ler a entrada $A[v][w]$ da matriz, o que nos custa tempo constante. A etapa de remover a aresta segue da mesma maneira e consiste em atribuir um valor a esta mesma posição. Com estas duas operações em sequência temos um algoritmo de complexidade $O(1)$.

Uma estrutura de adjacências, por sua vez, apresenta um custo maior. Um algoritmo que faça essa verificação precisa percorrer todos os nós da lista encadeada $A[v]$, o que levará $\text{grau}(v)$ passos. Em um grafo completo (o pior caso), o grau de um vértice pode ser, no máximo, $n - 1$. Com isso, a busca tem complexidade $O(n)$. Caso encontre o vértice w na lista da vizinhança de v e tratando-se de uma lista encadeada, basta fazer com que o nó que apontava para w aponte para aquele que w apontava anteriormente. Ao todo, verificar a existência e remover a aresta demanda tempo linear $O(n)$.

b) Ao utilizar vetores ordenados no lugar das listas encadeadas na estrutura de adjacências temos um ganho na verificação da existência de uma aresta, já que podemos encontrar um elemento em um vetor ordenado em tempo logarítmico $O(\log n)$ através de uma busca binária. A remoção da aresta, no entanto, acaba por se mostrar um processo mais custoso. No paradigma anterior, era possível fazer isso em tempo constante $O(1)$. Agora que temos um vetor, isso demanda tempo linear $O(n)$ pois ao remover um elemento é preciso copiar todos aqueles que o sucedem para a posição anterior, ou seja, se encontramos o elemento na posição i do vetor, precisamos copiar o valor da entrada j para a posição $j - 1$ para todo $j > i$. De qualquer forma, nesse caso temos também complexidade linear $O(n + \log n) = O(n)$.

Nota: Importante lembrar que no caso de grafos não-direcionados temos custo de remoção dobrado para cada uma das implementações, o que não altera a classe de complexidade das análises.

Questão 5

a) Para que o algoritmo falhe, basta existir um ciclo cujas arestas possuam peso inferior aos pesos das demais ligações dos vértices que pertencem ao ciclo. Caso isto aconteça, o algoritmo dará prioridade a formar o ciclo devido ao baixo valor de suas arestas e a condição de que M se torne uma árvore não será atendida, pela própria definição.

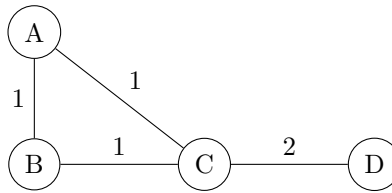


Figura 3: Um grafo com um ciclo

No grafo acima, encontramos este caso, em que o algoritmo prefere completar o ciclo (A, B, C, A) e acaba por não conseguir criar uma árvore a partir do grafo.

b) Respondendo ao questionamento proposto na "Dica": Como foi dito no item acima, ao completar um ciclo, não somos mais capazes de obter uma árvore. Assim, acrescentamos à 5ª linha do algoritmo, a condição para que a aresta só seja inserida à árvore M caso esta operação não provoque a formação de um ciclo.

c) Toda árvore com $n = |V(G)|$ vértices, além de ser um grafo acíclico e conexo, tem exatamente $m = n - 1 = |E(G)|$ arestas. Dessa maneira, se o algoritmo garante que não existem ciclos na construção de M por conta da condição imposta no item b), este garante que M não será uma árvore enquanto não possuir $|E(G)| = |V(G)| - 1$ arestas.

Após $|V(G)| - 1$ inserções, o algoritmo nos garante que M será uma árvore geradora mínima, conforme o Lema 5.2[1].

Questão 6

- a)
- b)
- c)
- d)

Referências

- [1] SZWARCFITER, Jayme Luiz, **Teoria Computacional de Grafos**, 1^a edição, Rio de Janeiro, 2018.