CPS740 - Lista 2

Pedro Maciel Xavier
116023847

13 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 ← critério(v[i], p[i])
6          seja k ← i
7          para j de i-1 até 1:
8              r ← critério(v[j], p[j])
9              se r > m:
10                 m ← r
11                 k ← j
12         v[i]←→v[k]
13         p[i]←→p[k]
14
15 def guloso(P, n, v[n], p[n]):
16     seja X[n]
17     seja V ← 0
18
19     ordena(v, p, n)
20
21     enquanto n > 0:
22         se p[n] <= P:
23             X[n] ← 1
24             P ← P - p[n]
25             V ← V + v[n]
26         n ← 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 `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 `f(P, n, ...)` para um determinado par `[P, 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 `[P, 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 progdina(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 [V, 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

Uma abordagem por Grafos

Neste problema, dado um conjunto T de tarefas, onde cada tarefa $t \in T$ é representada por um par $(\mathbf{i}(t), \mathbf{f}(t))$ com $\mathbf{i}(t) \leq \mathbf{f}(t)$ 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:

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

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 através de um Grafo

```
def tarefas(T[n][2], n):
    // Conjuntos de vértices e arestas
    seja V{} ← {v para cada v de 1 até n}
    seja E{} ← {}

    // Construção do complemento de G
    para cada i de 1 até n:
        para cada j de i + 1 até n:
            // tarefas não concomitantes
            se T[i][2] <= T[j][1] ou T[j][2] <= T[i][1]:
                E ← E ∪ {(i, j)}
    retorna clique_maximal(V, E)

def clique_maximal(V{}, E{}):
    // Retorna o tamanho do maior clique no grafo G(V, E)
    se vazio(V):
        retorna 0
    // Cliques de tamanho 2 (conjunto de conjuntos)
    seja K{} ← {{v, w} para cada (v, w) em E}
    se vazio(K):
        retorna 1
    senão:
        seja r ← 2
        seja X{} // Conjunto de r-cliques
        seja x{} // Conjunto dos vértices do clique
        enquanto r <= |V|:
            para cada k em K:
                para cada v em V:
                    x ← k ∪ {v}
                    se x em X:
                        continua
                    senão se forma_clique(E, k, v):
                        X ← X ∪ {x}
            se vazio(X):
                para
            senão:
                K ← X
                X ← {}
                r ← r + 1
                continua
        retorna r

def forma_clique(E{}, k{}, v):
    // Verifica se k ∪ {v} forma uma clique
    // dado que k é uma clique
    para cada w em k:
        se (v, w) em E:
            continua
        senão:
            retorna 0
    retorna 1
```

Questão 3

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

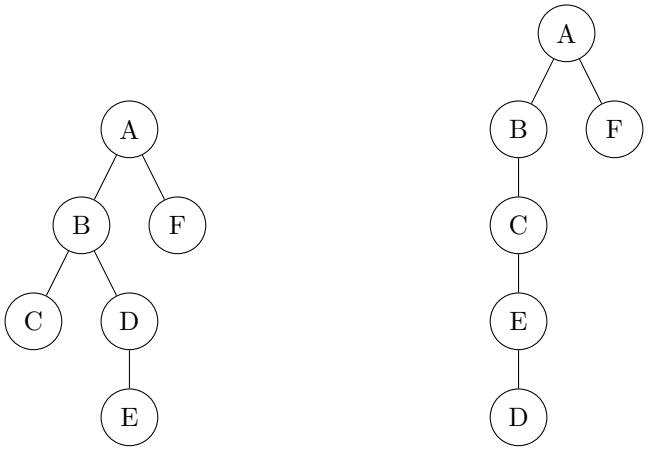

Figura 2: Árvores geradoras mínimas

b) A aresta (F, D) não pode compor uma árvore gerador mínima porquê 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 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.

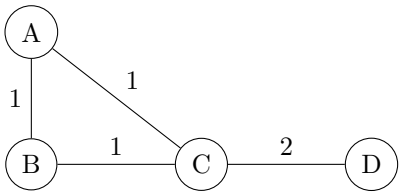

Figura 3: Um grafo com um ciclo

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) Comecemos o desenvolvimento pela observação de que as árvores geradoras de K_n rotulado são simplesmente as diferentes possíveis árvores rotuladas com n vértices. Escolhendo uma árvore T qualquer em $\mathbb{T}_n$ (conjunto das árvores rotuladas com n vértices), podemos dizer que $E(T) \subset E(K_n)$, já que sempre é possível ligar dois vértices em K_n. Um exemplo sobre K_5:

Figura 4: K_5 e uma árvore geradora.

Seja $T_n = |\mathbb{T}_n|$. Vamos contar as árvores que existem para $n = 1, 2, 3, 4$. Para $n = 1$ temos apenas a árvore de um único vértice. Com $n = 2$, a única configuração possível é a ligação entre os vértices 1 e 2.

Figura 5: $\mathbb{T}_n$ para $n = 1, 2$

Para $n = 3$, construir uma árvore geradora consiste em escolher 2 dentre as 3 arestas da clique K_3. Com isso produzimos 3 configurações distintas.

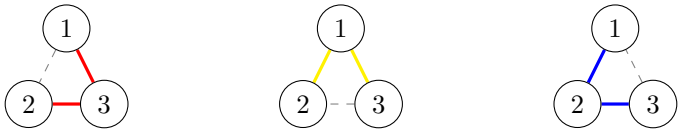

Figura 6: Árvores em $\mathbb{T}_3$

Por fim, para $n = 4$, temos de escolher 3 dentre as 6 arestas da clique, mas sem formar ciclos.

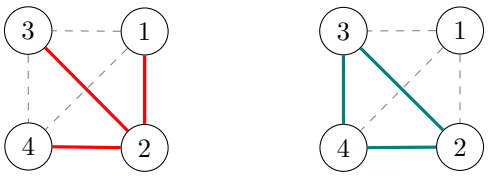

Figura 7: Uma árvore em $\mathbb{T}_4$ e um 3-clique em K_4

O único ciclo possível seria um subgrafo isomorfo a K_3. Formar um ciclo seria escolher 3 dentre os 4 vértices do grafo. Enunciadas estas relações, temos que:

$$T_4 = \binom{6}{3} - \binom{4}{3} = 20 - 4 = 16$$

Organizando os valores obtidos em uma tabela temos:

n	1	2	3	4
T_n	1	1	3	16

Figura 8: Número de árvores rotuladas com n vértices.

O leitor atento logo percebe que $16 = 4^2$. Isso é convite a escrever os demais resultados como potências de n. De fato, $3 = 3^1$, $1 = 2^0$ e $1 = 1^{-1}$. Portanto, seria um palpite bastante agradável dizer que $T_n = n^{n-2}$.

b) Uma vez conhecida a definição de uma *sequência de Prüfer* e analisando um conjunto V de n vértices indexados de 1 até n podemos construir uma sequência de $n-2$ elementos (não necessariamente distintos) escolhendo uma dentre as n opções dadas por V para cada uma das $n-2$ posições da sequência. A partir disso concluímos que o número de sequências geradas com n elementos é:

$$S_n = \underbrace{n \cdot n \cdot n \cdots n}_{n-2 \text{ vezes}} = n^{n-2}$$

Algoritmos de codificação de Prüfer

Antes de prosseguir para as duas demonstrações (itens **c)** e **d)**), vamos definir um par de funções, que chamaremos ψ e ψ^{-1}. Idealmente temos que:

$$\begin{aligned} \psi &: \mathbb{T} \to \mathbb{S} \\ \psi^{-1} &: \mathbb{S} \to \mathbb{T} \end{aligned}$$

Apresento, na página seguinte, algoritmos que cumprem os papéis de ψ e ψ^{-1}, respectivamente.

Algoritmo 4: Uma implementação de ψ

```
def folhas(T(V, E)):
    retorna {v para cada v em v(T) se grau(T, v) == 1}

def prufer_seq(T(V, E)):
    seja n ← |v(T)|        // Número de vértices em T
    seja S[max(n - 2, 0)] // Vetor da sequência de Prüfer

    seja k ← 0
    enquanto n > 2:
        // Folha com menor rótulo em T
        seja v ← min(folhas(T))

        // Único vizinho de v em T
        seja w em viz(T, v)

        S[k] ← w

        // Remove o vértice v de T
        T ← T - v

        k ← k + 1
        n ← n - 1
    retorna S
```

Algoritmo 5: Uma implementação de ψ^{-1}

```
def àrvore_rotulada(S[n]):
    seja n ← |S| + 2 // Comprimento da sequência S + 2

    seja V{} ← {v para cada v de 1 até n}
    seja E{} // Conjunto vazio

    seja T(V, E) // Árvore em construção

    seja L[] ← [j para cada j de 1 até n] // Lista

    enquanto |L| > 2:
        // Menor elemento em L que não está em S
        seja v ← min(L - S)
        // Primeiro elemento de S
        seja w ← S[0]

        // Conecta v e w
        T ← T + (v, w)

        L.remover(v)
        S.remover(w)

    // Conecta os vértices restantes
    T ← T + (L[0], L[1])
    retorna T
```

Analisando com atenção, percebe-se que cada operação do algoritmo de codificação possui um correspondente no de decodificação capaz de desfazê-la imediatamente. Com este argumento, mostramos que com o algoritmo proposto para ψ^{-1} somos capazes de decodificar em uma árvore T o código gerado pela mesma através do algoritmo que implementa ψ, ou seja, $[\ \psi^{-1} \circ \psi\ ](T) = T$ para qualquer árvore T em $\mathbb{T}$, assim como $[\ \psi \circ \psi^{-1}\ ](s) = s$ para qualquer sequência em $\mathbb{S}$

c) Queremos encontrar uma injeção de $\mathbb{T}$ em $\mathbb{S}$, isto é, uma função $f : \mathbb{T} \to \mathbb{S}$ tal que $x \neq y \implies f(x) \neq f(y)$. Sejam $P, Q \in \mathbb{T}$. Vamos supor por absurdo que ψ não é injetiva, ou seja, $P \neq Q$ e $\psi(P) = \psi(Q)$. Aplicamos a função ψ^{-1} dos dois lados da equação temos $\psi^{-1}(\psi(P)) = \psi^{-1}(\psi(Q)) \implies P = Q$. Um absurdo. Portanto, ψ é injetiva.

d) Seguindo raciocínio análogo, queremos uma injeção de $\mathbb{S}$ em $\mathbb{T}$. Sejam $p, q \in \mathbb{S}$. Vamos supor que ψ não é injetiva, ou seja, $p \neq q$ e $\psi^{-1}(p) = \psi^{-1}(q)$. Então aplicamos ψ dos dois lados da equação temos $\psi(\psi^{-1}(p)) = \psi(\psi^{-1}(q)) \implies p = q$. Uma contradição. Logo, ψ^{-1} é injetiva.

Já sabíamos que $S_n = |\mathbb{S}|$. Com estes resultados, temos uma relação de bijeção entre $\mathbb{T}$ e $\mathbb{S}$, portanto $|\mathbb{T}| = |\mathbb{S}|$ e, por fim, $T_n = S_n$.

Referências

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