CPS740 - Lista 1

Pedro Maciel Xavier
116023847

31 de julho de 2020

Questão 1

i) Primeiro Algoritmo

a) Algoritmo

Algoritmo 1: Ordenação de Lista

```
1  seja L[n]
2
3  def ordena(L[], n):
4      seja m, k
5      para i = n até 2:
6          m ← L[i]
7          k ← i
8          // Busca o maior elemento da sub-lista
9          para j = i-1 até 1:
10             // Novo maior
11             se L[j] > m:
12                 m ← L[j]
13                 k ← j
14         // Troca maior pelo último da sub-lista
15         L[i]⟷L[k]
16     retorna L
```

b) Passo-a-passo

$$
\begin{aligned}
L = &\{2, 7, 5, 6, \mathbf{9}, 0, 1, 4, 8, 5, \mathbf{3}\}\ 1 \times 11 = 11 \\
&\{2, 7, 5, 6, 3, 0, 1, 4, \mathbf{8}, \mathbf{5}, 9\}\ 2 \times 10 = 20 \\
&\{2, \mathbf{7}, 5, 6, 3, 0, 1, 4, \mathbf{5}, 8, 9\}\ 3 \times 9 = 27 \\
&\{2, 5, 5, \mathbf{6}, 3, 0, 1, \mathbf{4}, 7, 8, 9\}\ 4 \times 8 = 32 \\
&\{2, 5, \mathbf{5}, 4, 3, 0, \mathbf{1}, 6, 7, 8, 9\}\ 5 \times 7 = 35 \\
&\{2, \mathbf{5}, 1, 4, 3, \mathbf{0}, 5, 6, 7, 8, 9\}\ 6 \times 6 = 36 \\
&\{2, 0, 1, \mathbf{4}, \mathbf{3}, 5, 5, 6, 7, 8, 9\}\ 7 \times 5 = 35 \\
&\{2, 0, 1, \mathbf{3}, 4, 5, 5, 6, 7, 8, 9\}\ 8 \times 4 = 32 \\
&\{\mathbf{2}, 0, \mathbf{1}, 3, 4, 5, 5, 6, 7, 8, 9\}\ 9 \times 3 = 27 \\
&\{\mathbf{1}, \mathbf{0}, 2, 3, 4, 5, 5, 6, 7, 8, 9\}\ 10 \times 2 = 20 \\
&\{0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9\}\ \text{Total de 275 passos}
\end{aligned}
$$

ii) Segundo Algoritmo

a) Algoritmo

Algoritmo 2: Ordenação de Lista

```
1  seja L[n]
2
3  def ordena(L[], n):
4      seja i ← 2
5      enquanto i <= n:
6          // Verifica inversão
7          se L[i-1] > L[i]:
8              // Resolve inversão
9              L[i-1]⟷L[i]
10             se i > 2:
11                 i ← i - 1
12             senão:
13                 i ← i + 1
14         senão:
15             i ← i + 1
16     retorna L
```

b) Passo-a-passo

$L = \{2, 7, 5, 6, 9, 0, 1, 4, 8, 5, 3\}$

$\{2, \mathbf{5}, \mathbf{7}, 6, 9, 0, 1, 4, 8, 5, 3\}$ 2

$\{2, 5, \mathbf{6}, \mathbf{7}, 9, 0, 1, 4, 8, 5, 3\}$ 3

$\{2, 5, 6, 7, \mathbf{0}, \mathbf{9}, 1, 4, 8, 5, 3\}$ 4

$\{2, 5, 6, \mathbf{0}, \mathbf{7}, 9, 1, 4, 8, 5, 3\}$ 1

$\{2, 5, \mathbf{0}, \mathbf{6}, 7, 9, 1, 4, 8, 5, 3\}$ 1

$\{2, \mathbf{0}, \mathbf{5}, 6, 7, 9, 1, 4, 8, 5, 3\}$ 1

$\{\mathbf{0}, \mathbf{2}, 5, 6, 7, 9, 1, 4, 8, 5, 3\}$ 1

$\{0, 2, 5, 6, 7, \mathbf{1}, \mathbf{9}, 4, 8, 5, 3\}$ 5

$\{0, 2, 5, 6, \mathbf{1}, \mathbf{7}, 9, 4, 8, 5, 3\}$ 1

$\{0, 2, 5, \mathbf{1}, \mathbf{6}, 7, 9, 4, 8, 5, 3\}$ 1

$\{0, 2, \mathbf{1}, \mathbf{5}, 6, 7, 9, 4, 8, 5, 3\}$ 1

$\{0, \mathbf{1}, \mathbf{2}, 5, 6, 7, 9, 4, 8, 5, 3\}$ 1

$\{0, 1, 2, 5, 6, 7, \mathbf{4}, \mathbf{9}, 8, 5, 3\}$ 7

$\{0, 1, 2, 5, 6, \mathbf{4}, \mathbf{7}, 9, 8, 5, 3\}$ 1

$\{0, 1, 2, 5, \mathbf{4}, \mathbf{6}, 7, 9, 8, 5, 3\}$ 1

$\{0, 1, 2, \mathbf{4}, \mathbf{5}, 6, 7, 9, 8, 5, 3\}$ 1

$\{0, 1, 2, 4, 5, 6, 7, \mathbf{8}, \mathbf{9}, 5, 3\}$ 6

$\{0, 1, 2, 4, 5, 6, 7, 8, \mathbf{5}, \mathbf{9}, 3\}$ 3

$\{0, 1, 2, 4, 5, 6, 7, \mathbf{5}, \mathbf{8}, 9, 3\}$ 1

$\{0, 1, 2, 4, 5, 6, \mathbf{5}, \mathbf{7}, 8, 9, 3\}$ 1

$\{0, 1, 2, 4, 5, \mathbf{5}, \mathbf{6}, 7, 8, 9, 3\}$ 1

$\{0, 1, 2, 4, 5, 5, 6, 7, 8, \mathbf{3}, \mathbf{9}\}$ 6

$\{0, 1, 2, 4, 5, 5, 6, 7, \mathbf{3}, \mathbf{8}, 9\}$ 1

$\{0, 1, 2, 4, 5, 5, 6, \mathbf{3}, \mathbf{7}, 8, 9\}$ 1

$\{0, 1, 2, 4, 5, 5, \mathbf{3}, \mathbf{6}, 7, 8, 9\}$ 1

$\{0, 1, 2, 4, 5, \mathbf{3}, \mathbf{5}, 6, 7, 8, 9\}$ 1

$\{0, 1, 2, 4, \mathbf{3}, \mathbf{5}, 5, 6, 7, 8, 9\}$ 1

$\{0, 1, 2, \mathbf{3}, \mathbf{4}, 5, 5, 6, 7, 8, 9\}$ 1

$\{0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9\}$ Total de 64 passos

Questão 2

Seja $G = (V, E)$ um grafo com m arestas e n vértices.

a) A **Matriz de Adjacências** $A^{n \times n}$ de G é dada por:

$$A_{i,j} = \begin{cases} 1, \text{ se } (i,j) \in E \\ 0, \text{ caso contrário} \end{cases}$$

Desta maneira a matriz terá suas n^2 entradas. Por isso, precisaremos de, no mínimo, n^2 *bits* para representá-lo assim.

b) A **Estrutura de Adjacências** L de G consiste em uma lista de tamanho n contendo em cada nó i um ponteiro para uma lista que contém os vértices adjacentes a i. Ou simplesmente:

$$j \in L_i \iff (i,j) \in E$$

Neste caso, a demanda por espaço não está diretamente relacionada ao número de vértices, mas sim, à quantidade de arestas. Grafos com poucas conexões podem ser representados de maneira eficiente com essa estrutura. Por outro lado, redes muito conexas tornam esta abordagem indesejável.

Para cada aresta introduzida no grafo temos um custo de $2e$ *bits*, onde e representa o armazenamento de um nó j em uma lista e depende da implementação. O custo dobrado vem da necessidade de armazenar cada aresta em duas listas distintas. Por fim, o custo total para armazenar o grafo G desta maneira seria de $2e \times m + v \times n$, considerando v o custo para armazenar L.

c) Comparando as duas escolhas de representação, é vantajoso optar pela **Matriz de Adjacências** sempre que

$$n^2 < 2e \times m + v \times n$$

Mesmo assim, outros fatores devem ser levados em consideração, dependendo do tipo de operação que será realizada com mais frequência. Por exemplo, se queremos calcular o grau de cada vértice de um grafo, teremos maior eficiência se estivermos trabalhando com uma **Estrutura de Adjacências**. Se queremos simplesmente verificar se uma aresta arbitrária (v, w) pertence ao grafo, seria melhor estar lidando com uma **Matriz de Adjacências**.

Questão 3

O Teorema 2.1[1] diz que um grafo G possui ciclo euleriano se e somente se o grau de cada um de seus vértices for par.

Algoritmo 3: Verificando a existência de um ciclo euleriano

```
1  seja G(V, E)
2
3  def tem_ciclo_euleriano(G):
4      seja V ← v(G) // Vértices de G
5      para cada v em V:
6          se grau(v) % 2 == 1: // grau ímpar
7              retorna 0 // Falso
8      retorna 1 // Verdadeiro
```

A princípio, podemos perceber que o algoritmo demanda n verificações de grau, uma para cada vértice. No entanto, a operação `grau(v)` tem custo variado, a depender da implementação. Em uma matriz de adjacências simples, para cada vértice temos que verificar n entradas da matriz para contabilizar o grau. Em uma lista de adjacências, o custo pode ser um pouco menor a depender da densidade do grafo, mas ainda será $O(n^2)$. Uma estratégia para contornar este problema consiste em adotar uma estrutura de dados adicional com o papel de contabilizar o grau de cada vértice. Os valores são atualizados a cada inserção ou remoção de aresta. Isso aumenta o custo destas operações. No entanto, o cálculo do grau de um vértice se torna uma simples consulta, realizada em tempo constante $O(1)$. Isso garante a verificação da presença de um ciclo euleriano em tempo linear (amortizado) $O(n)$. Isso é feito para cada componente conexa do grafo.

Questão 4

O Teorema 2.2[1] afirma que um grafo bipartido só possui ciclos de tamanho par.

Podemos pensar no problema de bipartir o grafo como uma tarefa de coloração. Isto é, para um grafo $G(V_1 \cup V_2, E)$, podemos dizer que os vértices de V_1 possuem uma determinada "cor"**1** e os de V_2 são "coloridos"com a "cor"**2**.

Algoritmo 4: Verificando se um grafo é bipartido

```
1  seja G(V, E)
2
3  def bipartido(G):
4      seja V ← v(G) // Vértices de G
5      para cada v em V: // Descolore os vértices
6          v.cor ← nulo
7
8      seja v em V          // Vértice qualquer de V
9      seja G' ← G          // Cópia do grafo
10     seja cor ← 0         // Cor inicial
11     v.cor ← cor         // Colore o vértice
12     seja Q ← fila({v}) // Uma fila contendo v
13     enquanto !vazio(Q):
14         v ← S.remover()
15         // Inverte a cor
16         cor ← !v.cor
17         para cada w em viz(G', v):
18             // Já foi colorido
19             se w.cor == cor:
20                 continua
21             // Falha ao colorir
22             se w.cor == !cor:
23                 retorna 0
24             // Colore o vértice
25             se w.cor == nulo:
26                 w.cor ← cor
27                 // Coloca `w` no fim da fila
28                 Q.inserir(w)
29         // Remove `v` de G'
30         G' ← G' - v
31     retorna 1
```

Aqui, realizamos uma operação para cada vértice, assim como uma outra para cada aresta. A complexidade total desta busca em largura é $O(|V| + |E|)$ para cada componente conexa do grafo.

Questão 5

Sendo $G(V, E)$ um grafo bipartido, podemos escrever $G(V_1 \cup V_2, E)$.

a) O problema de encontrar um conjunto independente de tamanho k em um gravo $G(V, E)$ pode ser entendido como achar uma clique de mesmo tamanho no grafo complementar $G(V, \overline{E})$, onde $\overline{E} = \{e \in V \times V : e \notin E\}$. Dessa maneira, quando olhamos para o complemento de $G(V, E)$, vemos duas cliques se formarem entre os vértices de V_1 e V_2 respectivamente, já que não havia uma aresta sequer no interior destes conjuntos de vértices. Assim, sabendo que um gravo é bipartido, encontramos em tempo linear $O(m+n)$ uma 2-coloração para cada componente conexa. Em seguida, comparamos o valor desejado de k com o tamanho dos conjuntos V_1 e V_2 em tempo constante $O(1)$.

Considerando os diversos custos envolvidos, podemos afirmar que, para grafos bipartidos, somos capazes de resolver este problema em **Tempo Polinomial**.

Algoritmo 5: Subconjunto independente em grafo bipartido

```
1  seja G(V, E) // Grafo bipartido
2
3  def tem_subconjunto_independente(G, k):
4      // colore o grafo (Algoritmo 5)
5      se bipartido(G):
6          seja V ← v(G)
7      senão:
8          retorna "Erro: Jurastes ser bipartido!"
9
10     seja V0 ← {v em V: v.cor == 0}
11     seja V1 ← {v em V: v.cor == 1}
12
13     se k <= |V0| ou k <= |V1|:
14         retorna 1
15     senão:
16         retorna 0
```

b) Um ciclo hamiltoniano deve passar por todos os vértices exatamente uma vez, retornando ao ponto de partida. Como um grafo bipartido só possui arestas ligando vértices de V_1 a vértices de V_2, podemos afirmar que para passar uma única vez por cada vértice é preciso que os dois conjuntos tenham o mesmo tamanho. Caso contrário, após $2 \cdot min(|V_1|, |V_2|)$ passos não teríamos mais vértices não-visitados em um dos conjuntos, ficando assim presos no maior deles. Esta condição é necessária, porém não suficiente. Isso pode nos ajudar a fazer uma verificação rápida nos casos em que essa condição não é atendida. Uma outra ideia é verificar a existência de vértices cujo grau é 1. Estes não podem fazer parte um ciclo. Ambas as verificações podem ser realizadas em tempo linear $O(m+n)$ e não alteram a complexidade final do algoritmo, que é de uma ordem superior.

O algoritmo apresentado consiste inserir os vértices que fazem parte do caminho em uma pilha `P` conforme é feita a busca em profundidade. Os vértices a serem visitados são inseridos em uma outra pilha `S`, precedidos por um valor ***nulo***, que sinaliza o retorno (desempilhamento do vértice no topo de `P`). Quando o caminho conta com todos os vértices do grafo, é feita uma verificação quanto a presença do primeiro vértice na vizinhança do último, para assim completar um ciclo.

Para tratar da complexidade, lembramos que um Ciclo Hamiltoniano é uma sequência de vértices $\mathbb{H} = (v_1, v_2, \ldots, v_n, v_{n+1}), v_1 = v_{n+1}, v_i \in V, |V| = n$. Assim, podemos dizer que ao escolhermos um vértice arbitrário v_1 como ponto de partida nos deparamos com, no máximo, $n-1$ opções de trajetórias em nosso primeiro passo. Um segundo passo dado em direção a v_2 nos trás $n-2$ novas possibilidades, visto que não planejamos em nenhum momento da caminhada retornar a um local transpassado anteriormente. Seguindo este raciocínio, podemos ter até mesmo $(n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 = (n-1)!$ caminhos diferentes. Em cada caminho que trilhamos, realizamos operações de tempo constante (inserções, remoções e comparações) sobre os n vértices do grafo. Assim, podemos afirmar que este algoritmo possui complexidade assintótica $O(n!)$.

Conclui-se que a complexidade de tempo é **Fatorial** e, portanto, de **Tempo Exponencial**, visto que $\exists n_0$ tal que se $a > 1, a^n < n!\ \forall n > n_0$. Portanto, este algoritmo **não é Polinomial**.

Nota: O pseudo-código se encontra na próxima página.

Algoritmo 6: Ciclo Hamiltoniano

```
seja G(V, E)

def tem_ciclo_hamiltoniano(G):
    // colore o grafo (Algoritmo 5)
    se bipartido(G):
        seja V ← v(G)
    senão:
        retorna "Erro: Jurastes ser bipartido!"
    // Verifica o tamanho das partições
    seja V0 ← {v em V: v.cor == 0}
    seja V1 ← {v em V: v.cor == 1}
    se |V0| != |V1|:
        retorna 0
    // Verifica o grau e descolore
    para cada v em V:
        se grau(G, v) <= 1:
            retorna 0
        senão:
            v.cor ← nulo

    seja r em V // Vértice qualquer de G
    seja P ← pilha() // Pilha vazia
    seja S ← pilha({nulo, r}) // Pilha

    enquanto |S| > 0:
        v ← S.remover()
        se v != nulo:
            P.inserir(v)
            se |P| == |V|:
                // Encontrou
                se r em viz(G, v):
                    retorna 1
            senão:
                // Avante
                v.cor ← 1
                para cada w em viz(G, v):
                    se w.cor == nulo:
                        S.inserir(nulo)
                        S.inserir(w)
                continua
        // Retroceder
        v ← P.remover()
        v.cor ← nulo
    retorna 0
```

Questão 6

a) Existe o caminho simples (M, S).

b) Não existe nenhum ciclo simples no digrafo.

c) O grafo subjacente é conexo. No entanto, partindo de P não alcançamos nem N, nem O nem Q. Portanto, o digrafo é dito fracamente conexo.

d) O grau de saída de N é 4 e o de entrada é 1. Já o grau de saída de R é 0 e o de entrada é 2.

e)

Estrutura de Adjacências:

$$\{M : \{S\}, N : \{O, Q, R, S\}, O : \{M, N\}, P : \{M, P, R\}, Q : \{\}, R : \{\}, S : \{M\}\}$$

Matriz de Adjacências:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Questão 7

a) A soma dos quadrados dos números de 1 até n.

b) Temos n multiplicações e n somas, totalizando $2n$ operações, ou seja, tempo linear $O(n)$.

c) Existem algoritmos melhores. Demonstração:
Temos que

$$\begin{aligned} x^3 - (x-1)^3 &= x^3 - (x^3 - 3x^2 + 3x - 1) \\ &= x^3 - x^3 + 3x^2 - 3x + 1 \\ &= 3x^2 - 3x + 1 \end{aligned}$$

Assim,

$$\begin{aligned} &\implies && (x-1)^3 - (x-2)^3 = 3(x-1)^2 - 3(x-1) + 1 \\ &\implies && [x^3 - (x-1)^3] + [(x-1)^3 - (x-2)^3] = [3x^2 - 3x + 1] + [3(x-1)^2 - 3(x-1) + 1] \\ &\implies && x^3 - (x-2)^3 = 3[x^2 + (x-1)^2] - 3[x + (x-1)] + 2 \\ & && \vdots \\ &\implies && x^3 - (x-(n+1))^3 = 3\sum_{i=1}^{n}(x-i)^2 - 3\sum_{i=1}^{n}(x-i) + (n+1) \end{aligned}$$

Escolhendo $x = 0$:

$$\begin{aligned} &\implies && -(-(n+1))^3 = 3\sum_{i=1}^{n}(-i)^2 - 3\sum_{i=1}^{n}(-i) + (n+1) \\ &\implies && (n+1)^3 = 3\sum_{i=1}^{n} i^2 + 3\sum_{i=1}^{n} i + (n+1) \\ &\implies && \sum_{i=1}^{n} i^2 = \frac{(n+1)^3 - (n+1)}{3} - \sum_{i=1}^{n} i \\ & && = \frac{(n+1)^3 - (n+1)}{3} - \frac{n(n+1)}{2} \end{aligned}$$

Desta maneira, temos 3 somas, 3 multiplicações e 2 divisões, totalizando 8 operações, ou seja, tempo constante $O(1)$.

Referências

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