UNIVERSIDADE FEDERAL DE OURO PRETO
PCC104 - Projeto e Análise de Algoritmos
Prof. Rodrigo Silva

Força Bruta e Busca Exaustiva

Sara Câmara

12 de abril de 2023

1 Questão

Problema: Implementar o algoritmo *Selection Sort*. E apresentar a análise de complexidade de tempo do algoritmo.

Solução:
a) Implementação:

```
def selection_sort(array):
    size = len(array)
    for position in range(size-1):
        index_min = position
        for step in range(position+1, size):
            if array[step] < array[index_min]:
                index_min = step
        if index_min != position:
            array[position], array[index_min] = array[index_min], array[position]
    return array
```

b) Operação básica: Comparação para encontrar o menor elemento.

```
array[step] < array[index_min]
```

c) Função de custo:

$$C_w(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

d) Cálculo da função de custo:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$C\left(n\right)=\sum_{i=0}^{n-2}(n-1)-\sum_{i=0}^{n-2}i=(n-1)\sum_{i=0}^{n-2}1-\frac{(n-2)(n-1)}{2}$$

$$C\left(n\right)=(n-1)^2-\frac{(n-2)(n-1)}{2}=\frac{(n-1)n}{2}\approx\frac{1}{2}n^2$$

e) Eficiência (O ou Θ):

$$\frac{1}{2}n^2\in\Theta(n^2)$$

Para provar esta afirmação, devemos achar constantes $c_1 > 0$, $c_2 > 0$,
tais que: $c_1n^2 \leq \frac{1}{2}n^2 \leq c_2n^2\ \forall n \geq n_o$

Essa inequação se mantém verdadeira para $c_1 = \frac{1}{4}$, $c_2 = 2$ e $n_0 = 1$.

2 Questão

Problema: Implementar o algoritmo *SequentialSearch2*. E apresentar a análise de complexidade de tempo do algoritmo.

Solução:
a) Implementação:

```
def sequential_search2(array, searched_item):
    array.append(searched_item)
    index = 0
    while array[index] != searched_item:
        index += 1
    if index < len(array)-1:
        return print("A posicao do valor procurado:", index)
    else:
        return print("Valor procurado nao encontrado")
```

b) Operação básica: Comparar se o item atual é o item procurado.

```
    array[index] == searched_item:
```

c) Função de custo:

$$C_w\left(n\right)=\sum_{i=0}^{n}1$$

d) Cálculo da função de custo:

$$C_w\left(n\right)=\sum_{i=0}^{n}1=n+1$$

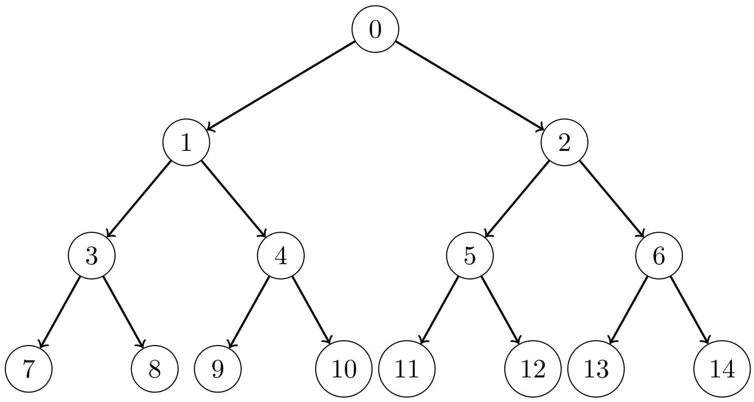
e) Eficiência (O ou Θ):

$$n+1\in\Theta(n)$$

Para provar esta afirmação, devemos achar constantes $c_1 > 0$, $c_2 > 0$,
tais que: $c_1n \leq n+1 \leq c_2n\ \forall\ n \geq n_o$

Essa inequação se mantém verdadeira para $c_1 = 1$, $c_2 = 3$ e $n_0 = 1$.
Melhor caso: $O(1)$

Para o desenvolvimento e análise do algoritmo de busca em largura para grafos (bfs) e busca em profundidade (dfs) foi considerado o seguinte grafo:



start: 0

goal: 10

Visited(bsf): $v = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
Path(bsf): $p = \{0, 1, 4, 10\}$

Visited(dsf): $v = \{0, 1, 3, 7, 8, 4, 9, 10\}$
Path(dsf): $p = \{0, 1, 4, 10\}$

---

3 Questão

Problema: Implementar o algoritmo de busca em largura para grafos. E apresentar a análise de complexidade de tempo do algoritmo.

Solução:
a) Implementação:

```
def search_bfs(graph, start, goal):
    visited, frontier = set(), Queue()
    frontier.put((start, [start]))
    while frontier:
        (node, path) = frontier.get()
        if node == goal:
            return path
        for neighbour in graph[node] - visited:
            visited.add(node)
            frontier.put((neighbour, path + [neighbour]))
    return None
```

4 Questão

Problema: Implementar o algoritmo de busca em profundidade para grafos. E apresentar a análise de complexidade de tempo do algoritmo.

Solução:
a) Implementação:

```
def search_dfs(graph, start, goal):
    visited, frontier = set(), LifoQueue()
    frontier.put((start, [start]))
    while frontier:
        (node, path) = frontier.get()  # pop
        if node == goal:
            return path
        for neighbour in graph[node] - visited:
            visited.add(node)
            frontier.put((neighbour, path + [neighbour]))
    return None
```

Como a diferença da implementação dos algoritmos BFS e DFS se trata da estrutura de dados usada, a análise de custo é igual para os dois algoritmos $O(|V|)$.

b) Operação básica: adicionar nós para serem visitados na fronteira *frontier.put*.

c) Função de custo:

$$C(h, m) = \sum_{i=0}^{h-1} m^i$$

d) Cálculo da função de custo:

$$C(h, m) = \sum_{i=0}^{h-1} m^i = \frac{X(m^h - 1)}{(m-1)} = \frac{(m^h - 1)}{(m-1)}$$

$$C(h, m) = \frac{(m^h - 1)}{(m-1)}$$

$$\Theta(\frac{m^h - 1}{m - 1}) \in O(m^h)$$

e) Eficiência (O ou Θ): $O(m^h)$ sendo m o número de vizinhos e h a altura da árvore.