Análise da complexidade do pior caso da US13

Algoritmo de Ordenação - Bubble Sort

Pseudocódigo:

```
procedure bubble_sort(List<Edge> e)
     for i:= 1 to E-1
          for j:= 1 to E-i
               if e[j] > e[j+1] then swap e[j] and e[j+1]
```

O ***E*** corresponde ao número de arestas e a primeira linha do bubble sort é efetuada ***E−1*** vezes. Para cada iteração ***i*** do primeiro ciclo ***for***, são efetuadas ***E−i*** iterações do segundo ciclo ***for*** e, em cada uma destas iterações, são feitas ***E−i*** comparações e, no máximo, ***E−i*** trocas. Então, o número total de comparações efetuadas (que é também o número total de iterações do segundo ciclo e o número máximo de trocas) é dado por

$$\sum_{i=1}^{E-1}(E-i) = \sum_{i=1}^{E-1}E - \sum_{i=1}^{E-1}i = E(E-1) - \frac{1+(E-1)}{2}(E-1) = \frac{E(E-1)}{2}$$

Por fim, a primeira e a segunda linha do algoritmo foram ainda efetuadas, respetivamente, **1** e ***E*** vezes, de cada vez que se concluiu que o ciclo terminou. Logo, a complexidade do pior caso é dada por **O(E) + O(E^2) + O(E^2) + O(E^2) = O(E^2).**

Método find

Pseudocódigo:

```
function find(parent, vertex)
     root = vertex
     while parent[root] is not root
          root = parent[root]
     while vertex is not root
          next = parent[vertex]
          parent[vertex] = root
          vertex = next
     return root
```

A primeira linha da função tem complexidade **O(1).** A segunda, que contêm o primeiro ciclo *while*, percorre o grafo a partir de um determinado vértice até encontrar a raiz, logo, no pior caso, irá correr ***V*** vezes, tendo, assim, complexidade **O(V)**, onde o ***V*** corresponde ao número de vértices. O número de iterações da quarta linha, onde se encontra o segundo ciclo *while*, irá ser semelhante ao anterior, ou seja, **V** iterações e, como tal, tem, também, complexidade **O(V).** A última linha apresenta complexidade

O(1). Por fim, concluímos que a complexidade do pior caso é dada por **O(1) + O(V) + O(V) + O(1) = O(V).**

Algoritmo de Kruskal

<u>Pseudocódigo:</u>

```
function kruskalAlgorithm(graph)
     minimumGraph = empty map
     parent = empty map
     for each vertex in graph
          parent[vertex] = vertex
          minimumGraph[vertex] = empty list

     edges = empty list
     for each edgeList in graph
          add all edges from edgeList to edges

     bubble_sort (edges)

     for each edge in edges
          sourceParent = find(parent, edge.vertex1)
          destParent = find(parent, edge.vertex2)
          if sourceParent is not equal to destParent
               add edge to minimumGraph[edge.vertex1]
               parent[sourceParent] = destParent

return minimumGraph
```

As duas primeiras linhas do algoritmo têm complexidade de **O(1)**. A terceira linha, que contém o primeiro ciclo *for*, onde se inicializam os *maps*, tem complexidade de **O(V)**, uma vez que corre para todos os vértices. ***V*** corresponde ao número de vértices. A sexta linha tem complexidade de **O(1)** e a quinta, onde se encontra outro ciclo *for*, cria uma lista com todas as arestas do grafo. Assim, a complexidade será **O(E)**, sendo que **E** corresponde ao número de arestas. Como já havíamos visto, o algoritmo *bubble sort* tem complexidade de **O(E^2).** Por fim, o último ciclo *for*, tem complexidade de **O(E x V)**, uma vez que, o *for* corre para todas as arestas e o *find*, que se encontra dentro do ciclo, corre para todos os vértices, como já tínhamos visto acima. A última linha tem complexidade de **O(1).** Posto isto, a complexidade do pior caso é dada por **O(1)+ O(1) + O(V) + O(1) + O(E) + O(E^2) + O(E x V) + O(1) = O(E^2).**

US17

Pseudocódigo:

```
Function Dijkstra(graph, start):
    // Initialize structures
    distances = Map()
    previousNodes = Map()
    unvisited = Set()
    shortestPath = Map()

    // Initialize distances and previous nodes
    For each node in graph.keys():
        distances[node] = INFINITY
        previousNodes[node] = NULL
        unvisited.add(node)

    distances[start] = 0

    // Main loop
    While unvisited is not empty:
        // Find the unvisited node with the smallest distance
        currentNode = NULL
        smallestDistance = INFINITY

        For each node in unvisited:
            currentDistance = distances[node]
            If currentDistance < smallestDistance:
                smallestDistance = currentDistance
                currentNode = node

        If currentNode == NULL:
            break

        unvisited.remove(currentNode)

        // Update distances for neighboring nodes
        For each edge in graph[currentNode]:
            neighbor = edge.target
            If neighbor not in unvisited:
                continue

            newDistance = distances[currentNode] + edge.weight
            If newDistance < distances[neighbor]:
                distances[neighbor] = newDistance
                previousNodes[neighbor] = currentNode

    // Construct the shortest path graph
    For each node in graph.keys():
        shortestPath[node] = List()

    For each node in previousNodes.keys():
        previousNode = previousNodes[node]
        If previousNode != NULL:
            weight = distances[node] - distances[previousNode]
            shortestPath[previousNode].add(Edge(node, weight))

    Return shortestPath
```

Análise de Complexidade Temporal O

1. Inicialização:
    - Inicialização do mapa de distâncias e do mapa de vértices anteriores: Isso leva O(V) tempo, onde V é o número de vértices no grafo.
    - Inicialização do conjunto de não visitados com todos os nós: Isso leva O(V) tempo.
2. Loop Principal
    - O loop principal corre até que todos os vértices sejam visitados, o que significa que corre V vezes.
    - Encontrar o Nó com a menor distância:
        - No pior caso, para cada iteração, percorremos todos os nós não visitados para encontrar o nó com a menor distância, ou seja O(V) por iteração.
        - Como o loop principal corre V vezes, o tempo total gasto nessa operação é de O(V^2).
    - Atualizar distâncias para os vértices vizinhos:
        - Para cada vértice, examinamos todos os seus vizinhos. No total, para todos os nós, examinamos todas as arestas, que são E (número de arestas).
        - Atualizar a distância e o mapa de nós anteriores para cada aresta leva O(1) tempo.
        - Portanto, o tempo total gasto com esta operação é O(E).
3. Construção do Grafo do Caminho de Menor Custo
    - Iterar sobre os vértices do grafo para inicializar shortestPath: O(V).
    - Iterar sobre todos os vértices em previousNodes para construir as arestas do caminho mais curto: O(V).
    - A construção do grafo do caminho mais curto leva $O(V)$ tempo.

Complexidade Temporal Total

- Inicialização: O(V)
- Loop Principal:
    - Encontar o vértice com menor distância: O(V^2)
    - Atualizar distâncias: O(E)
- Construção do grafo do caminho de menor custo: O(V)

Portanto, a complexidade temporal total é de O(V^2 + E) ou O(n^2).

Análise da Complexidade temporal do algoritmo da us18

Linhas	procedure findShortestPathToAP(graph, start, AP)	nº vezes executada	estimativa O
1	minCost = infinity	1	O(1)
2	minCostAssemblyPoint = null	1	O(1)
3	minCostPath = null	1	O(1)
	path = null		
4		k	
5	for each assemblyPoint in AP	k	O(k)
6	path = disjkstra(graph, assemblyPoint)	k	O(k * (n^2 + E))
	cost = path[assemblyPoint][0] get weigth	k	O(1)
7			
8	if cost < minCost	k	O(1)
9	minCost = cost	k	O(k)
10	minCostAssemblyPoint = assemblyPoint	k	O(1)
	minCostPath = path		O(1)

As quatro primeiras linhas são inicializações de variáveis e são executadas apenas uma vez, tendo uma complexidade temporal de O(1).

Estabelecemos que existem n vértices no grafo e k *assembly points*. Vamos percorrer todos os k assembly points, resultando em k execuções de todas as restantes linhas.

O dijkstra é executado para cada assembly point. A estimativa O é a complexidade temporal do algoritmo, que é O(n^2 + E), onde "n" continua a ser o número de vértices do grafo e "E" o número de arestas. Juntando a questão dos assembly points ao algoritmo Dijkstra, a estimativa O fica O(k * (n^2 + E)). Obter e atribuir o valor do custo, comparar e atualizar valores tem uma complexidade temporal constante O(1).

Logo, a pior complexidade temporal para este procedimento é logarítmica, O(k * (n^2 + E)).