DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Instituto Superior de Engenharia do Porto

MATEMÁTICA DISCRETA

US19

Team 3,14 G31 1DC
Luís Aquino - 1221286
Marta Domingues - 1231183
Nuno Teixeira - 1231375
Mariana Sousa - 1230792

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Este pdf contém a teoria estruturada e análise da complexidade do tempo do pior caso de procedimentos desenvolvidos nas respetivas User Stories. Os algoritmos de Dijkstra e Kruskal estão apresentados abaixo em pseudocódigo, com a sua complexidade analisada.
Antes de começar a analisar, temos abaixo a teoria usada para o desenvolvimento da análise feita ao detalhe.
As informações abaixo apresentam as operações principais dos algoritmos usados com as suas respetivas complexidades.

- atribuições (A);
- incrementos (I);
- comparações (C);
- retornos (R).

MÉTODO DO ALGORITMO DE KRUSKAL US13

```
Method calculateMinimumSpanningTree(edges, mstEdges)
edges = sort(edges by weight)
parent = new HashMap()
rank = new HashMap()
vertices = new HashSet()
for edge em edges
vertices.add(edge.from)
vertices.add(edge.to)
for edge em edges
rootX = find(parent, edge.from)
rootY = find(parent, edge.to)
if rootX != rootY
mstEdges.add(edge)
union(parent, rank, rootX, rootY)
if tamanho(mstEdges) = tamanho(vertices) – 1
break
Return mstEdges
```

PSEUDOCÓDIGO DA US13 E RESPETIVA ANÁLISE DETALHADA SOBRE SUA COMPLEXIDADE

Código	**Análise**
Method calculateMinimumSpanningTree(edges, mstEdges)	$O(n^2)$
edges = sort(edges by weight)	O(n log n)
parent = new HashMap() rank = new HashMap() vertices = new HashSet()	nA
for edge em edges	O(n)
vertices.add(edge.from) vertices.add(edge.to)	O(1)
for edge em edges	O(n)
rootX = find(parent, edge.from) rootY = find(parent, edge.to)	O(log n)
if rootX /= rootY	C
mstEdges.add(edge)	A
union(parent, rank, rootX, rootY)	O(log n)
if tamanho(mstEdges) = tamanho(vertices) - 1	C
break	------------------------
Return mstEdges	R

A complexidade de tempo O(n log n) para a operação de ordenação, como **edges = sort(edges by weight)**, decorre do uso de algorimos de ordenação eficientes, como QuickSort, MergeSort. Agora passando a uma explicação mais detalhada:

No QuickSort:

- Pior caso: $O(n^2)$, mas com boas implementações e na prática, é O(n log n)
- Média e melhor caso: O(n log n)
- Funcionamento: QuickSort é um algoritmo de divisão e conquista que escolhe um "pivô" e particiona a lista em dois sub-arranjos: um com elementos menores que o pivô e outro com elementos maiores. Em seguida, ele ordena os sub-arranjos recursivamente.

No MergeSort:

- Complexidade de tempo garantida: O(n log) para o melhor, pior e casos médios.
- Funcionamento: MergeSort também é um algoritmo de divisão que divide a lista em duas metades, ordenando cada metade recursivamente e depois as combina (merge).

Esses algoritmos eficientes têm uma complexidade de tempo O(n log n) por causa das seguintes razões:

1) Divisão:
    - Eles dividem a lista em partes menores (log n vezes)

- Em cada nível de divisão, o custo de combinar (merge) ou repartir a lista é O(n)

2) Profundidade de recursão:
   - A profundidade da árvore de recursão é **log n.**
   - O custo de cada nível de árvore de recursão é proporcional a **O(n).**

Concluindo, a complexidade total de tempo é **O(n log n)**, onde **n** é o número de elementos na lista (número de arestas no caso da US)

Para dar um exemplo mais compreensível: Se tivermos 8 elementos, a árvore de recursão pode ser visualizada assim:

```
                          [8 elementos]
                      /                      \
            [4 elementos]                [4 elementos]
          /         \                   /          \
      [2]              [2]            [2]             [2]
    /    \           /    \         /    \          /    \
 [1]  [1]          [1]  [1]      [1]      [1]  [1]        [1]
```

- Número de níveis: $\log_2(8) = 3$
- Operações por nível: O(n) (combinando ou dividindo os elementos)

Multiplicando o número de níveis (log n) pelas operações por nível (n), obtemos O(n log n), sendo assim o motivo para **edges = sort(edges by weight)** ter complexidade O(n log n)

Explicando agora a inicialização de **parent , rank, vertices** ser considerado O(n) ou nA:

1) **parent = new HashMap():**

- **Explicação:** A inicialização de um **HashMap** em si é uma operação de tempo constante, O(1). No entanto, o uso do **HashMap** será O(n) ao longo do tempo, pois vamos armazenar cada nó individualmente no **HashMap** durante o processo das arestas.
- **Operação associada**: Durante o processamento, a inserção de **n** nós em um **HashMap** envolve **n** operações, cada uma sendo O(1) Portanto, o uso geral do **HashMap** para armazenar os "pais" dos nós será O(n).

2) **rank = new HashMap**

- **Explicação**: Similar ao **parent**, a inicialização do **HashMap** é uma operação de tempo constante, O(1). No entanto, durante a execução do algoritmo, vamos armazenar e acessar informações de **rank** para cada nó, o que, acumulativamente, é O(n).
- **Operação associada**: Durante o processamento, a inserção e acesso de **n ranks** de nós em um **HashMap** envolve **n** operações, cada uma sendo O(1). Portanto, o uso geral do **HashMap** para armazenar os **ranks** dos nós será O(n).

3) vertices = new HashSet()

- **Explicação**: A inicialização de um **HashSet** também é uma operação de tempo constante, O(1). No entanto, o uso do **HashSet** será O(n) ao longo do tempo, pois vamos armazenar cada nó individualmente no **HashSet** durante o processamento das arestas.
- **Operação associada**: Durante o processamento, a inserção de **n** nós em um **HashSet** envolve **n** operações, cada uma sendo O(1). Portanto, o uso geral do HashSet para armazenar os nós será O(n) (ou nA).

Concluindo, embora a criação inicial de cada estrutura (HashMap ou HashSet) seja uma operação de tempo constante, O(1), o termo nA ou O(n) reflete assim, o uso cumulativo dessas estruturas durante o processamento de todas as arestas no gráfico. Como vamos armazenar informações para cada nó (até **n** nós, onde **n** é o número de nós), a complexidade acumulativa para essas operações será sempre O(n).

Antes de continuar gostaria de clarificar a diferença entre as notações de complexidade e a razão pela qual nA e O(n) são utilizados de forma diferente no contexto de pseudocódigo.

O(n): É uma notação assintótica que descreve o comportamento da complexidade de tempo ou espaço de um algoritmo à medida que no tamanho da entrada cresce. Indica que a complexidade é linear em relação ao tamanho da entrada.

nA: No contexto do pseudocódigo fornecido, nA é uma notação específica que está sendo usada para descrever operações acumulativas em termos de número de operações (A - Atribuições). Neste contexto, nA representa n operações de atribuição.

For edge em edges - O(n):

- Aqui, estamos repetir sobre cada aresta na **lista edges**. Se há **n** arestas, essa operação de iteração ocorre **n** vezes.
- **Complexidade**: O(n) onde x é o número de arestas.

vertices.add(edge.from) - O(1):

- A operação **add** em um **HashSet** tem complexidade de tempo constante O(1) no caso médio, devido ao uso de tabelas de **hash**.
- **Complexidade por operação:** O(1).

vertices.add(edge.to) - O(1):

- Semelhante à operação **add** para adicionar o nó to ao **HashSet** também tem complexidade O(1).
- **Complexidade por operação:** O(1).

---

for edge em edges → O(n)
rootX ← find(parent, edge.from) → O(log n)
rootY ← find(parent, edge.to) → O(log n)

O(n): Esta complexidade ocorre quando repetimos linearmente sobre uma coleção de tamanho **n**. No caso do **for edge em edge**, repetir sobre todas as arestas (edges) é uma operação O(n).

O(log n): Esta complexidade ocorre para a operação find no contexto da estrutura de dados Union-Find com compressão de caminho. Cada find pode ter que percorrer até log n nós na pior das hipóteses devido à compressão de caminho, tornando a operação eficientemente O(log n).

---

if rootX ≠ rootY
mstEdges.add(edge)

C: Condição

- Representa uma verificação condicional.
- Indica que uma operação está a ser feita para verificar uma condição lógica.

A: Atribuição

- Representa uma operação de atribuição ou adição a uma estrutura de dados.
- Indica que um valor está a ser atribuído ou adicionado.

If rootX /= rootY → C

Descrição: Esta linha verifica se rootX é diferente de rootY.
Motivo para C:

- **C** representa uma operação de condição ou verificação lógica.
- Aqui, a condição **rootX≠ rootY** é avaliada para decidir se o código dentro do bloco **if** está a ser executado.
- A complexidade dessa verificação é O(1) (constante), mas em termos de pseudocódigo detalhado, é anotada como C para indicar que uma condição está a ser avaliada.

mstEdges.add(edge) → A

Descrição: Esta linha adiciona a aresta edge à lista mstEdges, que armazena as arestas da Árvore Geradora de Custo Minimo.
Motivo para A:

- **A** representa uma operação de atribuição.
- Aqui, **edge** está sendo adicionada à lista **mstEdges**.
- A operação de adição a uma lista é uma operação de **atribuição** e é marcada como **A** para indicar que um valor está sendo atribuído ou adicionado a uma estrutura de dados.
- A complexidade dessa operação é O(1) (constante), mas em termos de pseudocódigo detalhado, é anotada como **A** para indicar que uma atribuição está ocorrendo.

---

union(parent, rank, rootX, rootY) → O(log n)

A operação **union(parent, rank, rootX, rootY**) tem complexidade **O(log n)** devido à necessidade de encontrar as raízes dos conjuntos dos nós envolvidos (o que é O(log n) por causa da compressão de caminho) e subquentemente, união dos conjuntos baseados em seus status/ranks (o que é O(1)).

Portanto, a complexidade total da operação **union** é dominada pela complexidade de encontrar as raízes, resultando em **O(log n).**

---

if tamanho(mstEdges) = tamanho(vertices) - 1 → C

Descrição: Verifica se o número de arestas na da Árvore Geradora de Custo Minimo (mstEdges) é igual a n - 1, onde n é o número total de vértices no grafo. Isso é uma verificação para determinar se a árvore mínima já foi completamente construída.

Motivo para C:

- A complexidade dessa verificação é O(1) porque envolve apenas comparação de dois valores.

- No entanto, em termos de notação detalhada, é marcada como C para indicar que uma condição está sendo avaliada.

Return mstEdges → R

Descrição: Esta linha vai retornar a Árvore Geradora de Custo Minimo construída até o momento.
Motivo para R:
- A complexidade do retorno de uma estrutura de dados é **O(1)** porque não envolve nenhum processamento adicional, apenas a operação de retorno.
- No entanto, em termos de notação detalhada, é marcada como **R** para indicar que é uma operação de retorno.

MÉTODO DO ALGORITMO DE DIJKSTRA

```
Class DijkstraAlgorithm {

List<Route> edges


Function findShortestPathToAnyEndPoint(origin: Point, endPoints: List<Point>, edges: List<Route>) -> List<Route> {

// Assign the provided edges to the class variable

this.edges = edges


// Create a list of unique points from the edges

List<Point> points = new List

For each route in edges:

addUniquePoint(points, route.getOrigin())

addUniquePoint(points, route.getDestination())

End For


// Initialize arrays for costs, previous nodes, and visited nodes

Array<double> costs = new Array of size points.size()

Array<String> previousNodes = new Array of size points.size()

Array<boolean> visitedNodes = new Array of size points.size()


// Set default values for the arrays

For i from 0 to points.size() - 1:

costs[i] = Double.MAX_VALUE

previousNodes[i] = null

visitedNodes[i] = false

End For


// Set the cost of the origin point to 0

Integer originIndex = getPointIndex(points, origin)

costs[originIndex] = 0.0


// Initialize a list to keep track of node costs

List<NodeCost> nodeCosts = new List

nodeCosts.add(new NodeCost(origin, 0.0))
```

```
// Process nodes until all reachable nodes are visited
While nodeCosts is not empty:
NodeCost currentNodeCost = getMinimumDistancePoint(nodeCosts)
Point currentPoint = currentNodeCost.point
Integer currentIndex = getPointIndex(points, currentPoint)

// Skip this point if it has already been visited
If visitedNodes[currentIndex] is true:
Continue
End If

// Mark the current point as visited
visitedNodes[currentIndex] = true

// Update the cost for each neighbor of the current point
For each route in edges:
If route.getOrigin() equals currentPoint:
Point neighbor = route.getDestination()
Integer neighborIndex = getPointIndex(points, neighbor)
Double newCost = costs[currentIndex] + route.getCost()

// Update the cost and path if a cheaper path is found
If newCost < costs[neighborIndex]:
costs[neighborIndex] = newCost
previousNodes[neighborIndex] = currentPoint.getId()
nodeCosts.add(new NodeCost(neighbor, newCost))
End If
End If
End For
End While

// Find the closest endpoint with the minimum cost
Point closestEndPoint = null
Double minCost = Double.MAX_VALUE
For each endPoint in endPoints:
Integer endIndex = getPointIndex(points, endPoint)
```

```
If costs[endIndex] < minCost:
minCost = costs[endIndex]
closestEndPoint = endPoint
End If
End For

// Build and return the path to the closest endpoint
Return buildPath(origin, closestEndPoint, points, previousNodes)
End Function
```

PSEUDOCÓDIGO DA US19 E RESPETIVA ANÁLISE DETALHADA SOBRE SUA COMPLEXIDADE

Linha de Código	Número de Iterações	Complexidades Acumuladas
this.edges = edges	1	$O(1)$
List points = new List	1	$O(1)$
For each route in edges:	n	$O(nC + nI)$
addUniquePoint(points, route.getOrigin())	n	$O(nA)$
addUniquePoint(points, route.getDestination())	n	$O(nA)$
Array costs = new Array of size points.size()	1	$O(1)$
Array previousNodes = new Array of size points.size()	1	$O(1)$
Array visitedNodes = new Array of size points.size()	1	$O(1)$
For i from 0 to points.size() - 1:	n	$O(nC + nI)$
costs[i] = Double.MAX_VALUE	n	$O(nA)$
previousNodes[i] = null	n	$O(nA)$
visitedNodes[i] = false	n	$O(nA)$
Integer originIndex = getPointIndex(points, origin)	1	$O(1)$

costs[originIndex] = 0.0	1	$O(1)$
List nodeCosts = new List	1	$O(1)$
nodeCosts.add(new NodeCost(origin, 0.0))	1	$O(1)$
While nodeCosts is not empty:	n	$O(nC)$
NodeCost currentNodeCost = getMinimumDistancePoint(nodeCosts)	n	$O(nA)$
Point currentPoint = currentNodeCost.point	n	$O(nA)$
Integer currentIndex = getPointIndex(points, currentPoint)	n	$O(nA)$
If visitedNodes[currentIndex] is true:	n	$O(nC)$
Continue	-	-
visitedNodes[currentIndex] = true	n	$O(nA)$
For each route in edges:	n^2	$O(n^2C + n^2I)$
If route.getOrigin() equals currentPoint:	n^2	$O(n^2C)$
Point neighbor = route.getDestination()	n^2	$O(n^2A)$
Integer neighborIndex = getPointIndex(points, neighbor)	n^2	$O(n^2A)$
Double newCost = costs[currentIndex] + route.getCost()	n^2	$O(n^2Op)$
If newCost < costs[neighborIndex]:	n^2	$O(n^2C)$
costs[neighborIndex] = newCost	n^2	$O(n^2A)$
previousNodes[neighborIndex] = currentPoint.getId()	n^2	$O(n^2A)$
nodeCosts.add(new NodeCost(neighbor, newCost))	n^2	$O(n^2A)$
For each endPoint in endPoints:	n	$O(nC + nI)$
Integer endIndex = getPointIndex(points, endPoint)	n	$O(nA)$
If costs[endIndex] < minCost:	n	C
minCost = costs[endIndex]	n	$O(nA)$
closestEndPoint = endPoint	n	$O(nA)$
Return buildPath(origin, closestEndPoint, points, previousNodes)	1	R

this.edges = edges: Isto é uma atribuição direta de uma variável sendo feito em tempo constante, O(1), pois não depende do tamanho da entrada.

List points = new List: Neste caso, estamos apenas a incializar uma lista o que também é feito em tempo constante, O(1).

For each route in edges:: Este caso implica em percorrer em todos os elementos em "edges", que contém "n" elementos. Então, a complexidade é O(nI + nC), onde nI é o custo a percorrer sobre cada elemento e nC é o custo de realizar operações em cada elemento.

addUniquePoint(points, route.getOrigin()) e **addUniquePoint(points, route.getDestination())**: Neste caso, as operações adicionam pontos à lista e se a lista ainda não contém o ponto, ele é adicionado. Se a lista for grande, pode haver uma busca para verificar a existência do ponto, resultando em complexidade O(nA), onde nA é o custo de adicionar um ponto único.

Array costs = new Array of size points.size(): Iniicar uma matriz com o tamanho da lista de pontos é feito em tempo constante, O(1).

Array previousNodes = new Array of size points.size() e **Array visitedNodes = new Array of size points.size()**: Ambas as operações envolvem a inicialização de arrays com base no tamanho da lista de pontos, então também são O(1).

For i from 0 to points.size() - 1: Temos um Loop sobre todos os elementos da lista de pontos sendo a complexidade é O(nI+nC), onde nI é o custo a percorrer sobre cada elemento e nC é o custo de realizar operações em cada elemento.

costs[i] = Double.MAX_VALUE, **previousNodes[i] = null** e **visitedNodes[i] = false**: Estas operações são executadas para cada elemento na lista de pontos, portanto, a complexidade é O(nA), onde nA é o custo de atribuir um valor a cada elemento.

Integer originIndex = getPointIndex(points, origin) e **costs[originIndex] = 0.0**: São operações de busca em uma lista e atribuição de um valor, respectivamente. Portanto, cada uma é O(1).

List nodeCosts = new List e **nodeCosts.add(new NodeCost(origin, 0.0))**: Novamente, essas operações são feitas em tempo constante, O(1).

While nodeCosts is not empty: Um loop que continua até que "nodeCosts" esteja vazio. A complexidade é O(nC), onde nC é o custo das operações realizadas em cada iteração.

NodeCost currentNodeCost = getMinimumDistancePoint(nodeCosts), **Point currentPoint = currentNodeCost.point** e **Integer currentIndex = getPointIndex(points, currentPoint)**: Cada uma dessas operações é O(nA), onde nA é o custo da operação.

If visitedNodes[currentIndex] is true:: Uma operação de verificação. Isso é feito em tempo constante, O(1).

visitedNodes[currentIndex] = true: Atribuição de um valor a um elemento na lista "visitedNodes". Portanto, é O(1).

For each route in edges: Um loop que percorre todos os elementos em "edges". A complexidade é $O(n^2C+n^2I)$, onde n^2I é o custo de iterar sobre cada elemento e n^2C é o custo de realizar operações em cada elemento.

If route.getOrigin() equals currentPoint:: Uma verificação. Isso é feito em tempo constante, O(1).

Point neighbor = route.getDestination() e **Integer neighborIndex = getPointIndex(points, neighbor)**: São operações que envolvem buscar informações em uma lista. Assim, cada uma é O(1).

Double newCost = costs[currentIndex] + route.getCost(): Uma operação de adição. É O(1).

If newCost < costs[neighborIndex]: Uma operação de comparação, C.

costs[neighborIndex] = newCost e **previousNodes[neighborIndex] = currentPoint.getId()**: Operações de atribuição A, cada uma é O(1).

nodeCosts.add(new NodeCost(neighbor, newCost)): Adição de um elemento a uma lista. Isso é O(1).

For each endPoint in endPoints: Temos um loop que percorre todos os elementos em "endPoints". A complexidade é O(nC+nI), onde nI é o custo de percorrer sobre cada elemento e nC é o custo de realizar operações em cada elemento.

Integer endIndex = getPointIndex(points, endPoint): Uma operação de busca de uma informação em uma lista. É O(1).

If costs[endIndex] < minCost: Uma operação de comparação, C. Isso é feito em tempo constante, O(1).

minCost = costs[endIndex] e **closestEndPoint = endPoint**: Atribuição de valores. Cada uma é O(1).

Return buildPath(origin, closestEndPoint, points, previousNodes): Uma operação de retorno, R. Isso é feito em tempo constante, O(1).