Kruskal's and Dijkstra's Algorithm Pseudocode Analysis

Introduction

When comparing two or more algorithms that solve the same problem, it is important to choose the one that achieves the solution in the least possible time and uses the least amount of space. This analysis should be independent of the hardware's speed and the software's programming language. The primary measure for this analysis is the number of primitive operations required to perform the procedure and the associated cost of executing each operation. These primitive operations can be:

- **Arithmetic Operations:** Basic operations such as addition, subtraction, multiplication, and division.
- **Comparison Operations:** Operations that involve comparing two values, such as greater than, less than, or equal to.
- **Reading of an element in a list:** Iterations over arrays or other data structures, often the primary contributors to time complexity adding "n" complexity where n is the length of the list.
- **Atribution of values:** Operations that attribute a value to a variable or increment it.
- **Function Returns:** The return of values calculated in other methods.
- **Function Calls:** The action associated with calling methods.

Understanding these operations and their impact helps in analyzing and optimizing algorithms for better performance.

Algorithm Analysis US013

In this section, we will analyze the different methods of our Kruskal's algorithm developed in US013. Each method is presented with its code lines in a table, along with a the time complexity analysis.

Method: bubbleSort

The `bubbleSort` method sorts the edges based on their weights. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Code	**Analysis**
method bubbleSort(edges)	$O(n^2)$
Para i de 1 até tamanho(edges)-1	$(n-1)C$
Para j de 0 até tamanho(edges)-i-1	$\leq (n-1)(n-i)C$
Se edges[j].weight > edges[j+1].weight	$(n-1)(n-i)(1C+2L)$
Trocar edges[j] e edges[j+1]	$\leq (n-1)(n-i)(3A)$
Retornar edges	$1R$

Table 1: Method `bubbleSort`

Proceeding with the analysis of the worst-case complexity, the first line of the algorithm is executed $n-1$ times. For each iteration i of the first loop, $n-i$ iterations of the second loop are performed. In each of these iterations, $n-i$ comparisons are made and, at most, $n-i$ swaps. Thus, the total number of comparisons performed (which is also the total number of iterations of the second loop and the maximum number of swaps) is given by:

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

The time complexity of the `bubbleSort` method is dominated by the nested loops, which both iterate over the list of edges. In the worst case, this results in a quadratic time complexity, $O(n^2)$. Each comparison and swap operation within the inner loop also runs in constant time, but the total number of these operations scales quadratically with the number of edges. Therefore, the overall time complexity of the `bubbleSort` method is $O(n^2)$, making it less efficient for large datasets compared to more advanced sorting algorithms. x

Method: countUniqueNodes

The `countUniqueNodes` method counts the number of unique nodes present in the list of edges. This is essential for determining the number of iterations required to form the MST.

Code	**Analysis**
Function countUniqueNodes(edges)	$O(n)$
uniqueNodes ← novo Conjunto()	$1A$
Para cada edge em edges	nL
Adicionar edge.from a uniqueNodes	$n * O(1)$
Adicionar edge.to a uniqueNodes	$n * O(1)$
Retornar tamanho(uniqueNodes)	$1R$

Table 2: Function `countUniqueNodes`

The time complexity of the `countUniqueNodes` method is determined by the need to iterate through all edges and add their nodes to a set. The loop iterates n times (where n is the number of edges), and each insertion into the set occurs in constant time, $O(1)$. Therefore, the overall time complexity of the `countUniqueNodes` function is $O(n)$.

Method: find

The `find` method locates the representative of the set to which a given element belongs. This function is used to check for already formed cycles, making the Kruskal algorithm faster.

Code	**Analysis**
Method find(parent, node)	$O(\log(n))$
Se parent[node] não existe	$1L$
parent[node] ← node	$1A$
Se parent[node] $\neq$ node	$1L + 1C$
parent[node] ← find(parent, parent[node])	$O(\log(n))$
Retornar parent[node]	$1R$

Table 3: Function `find`

The time complexity of the `find` method is, in the worst case, a logarithmic time complexity, $O(\log n)$. Each individual operation within the method, such as checks and assignments, operates in constant time, contributing minimally to the overall complexity and becoming insignificant in the long term. Each

recursive call reduces the problem size approximately by half, leading to a logarithmic number of calls as it is presented in the example 3.8 from the notebook page 78.

Thus, while each individual operation within the `find` method takes constant time, the recursive nature of the method results in a worst-case time complexity of $O(\log n)$. This logarithmic complexity arises because the method may need to traverse up a balanced tree structure, where the height of the tree (and thus the number of recursive calls) is proportional to the logarithm of the number of nodes.

Method: calculateMinimumSpanningTreeCost

The `calculateMinimumSpanningTreeCost` method computes the minimum cost to connect all nodes in the graph using Kruskal's algorithm. It iterates through all edges, adds them to the MST if they do not form a cycle, and uses the `find` function to check for cycles.

Code	**Analysis**
Method calculateMinimumSpanningTreeCost	$O(n^2)$
edges ← bubble sort	$O(n^2)$
nNodes ← countUniqueNodes	$O(n)$
minCost ← 0	A
parent ← new HashTable()	nA
n ← 0	A
While n < nNodes - 1	nC
For each edge in edges	nA
rootX ← find(parent, edge.from)	$n * O(\log n)$
rootY ← find(parent, edge.to)	$n * O(\log n)$
If rootX ≠ rootY	C
parent[rootX] ← rootY	A
minCost ← minCost + edge.weight	A
Add edge to mstEdges	A
n ← n + 1	A
Return minCost	R

Table 4: Method `calculateMinimumSpanningTreeCost`

The dominant factor is sorting the edges, the method used is bubbleSort, which has a time complexity of $O(n^2)$. Each individual operation, such as finding the roots and performing unions, operates in logarithmic or constant time. Thus, the total complexity in the worst-case scenario is approximately $O(n^2)$, where n is the number of edges to be sorted in accordance with the theorem 3.3 from page 70 of the notebook where is stated that the worst case time complexity of an algorithm is the time complexity of the higher degree.

Algorithm Analysis US017 and US018

In this section, we will analyze the method of our Dijkstra's algorithm developed in US017. The method is presented with its code lines in a table, along with the time complexity analysis.

Method: minDistance

The `minDistance` method finds the vertex with the minimum distance value from the set of vertices not yet included in the shortest path tree.

Code	**Analysis**
Method minDistance(dist, sptSet, V)	$O(n)$
min ← Integer.MAX_VALUE	$1A$
minIndex ← -1	$1A$
Para v de 0 até V-1	$(n+1)C$
Se não sptSet[v] e dist[v] ≤ min	$n(1C+2L)$
min ← dist[v]	$\leq nA$
minIndex ← v	$\leq nA$
Retornar minIndex	$1R$

Table 5: Method `minDistance`

The time complexity of the `minDistance` method is $O(n)$, where n is the number of vertices. This is because the method involves iterating through all vertices to find the one with the minimum distance value.

Method: dijkstra

The `dijkstra` method implements Dijkstra's single-source shortest path algorithm for a graph represented using an adjacency matrix.

Code	**Analysis**
Method dijkstra(graph, src, target, names, finalPath)	$O(n^2)$
V ← tamanho(graph)	$1A+1F$
dist ← novo vetor de tamanho V	$1A$
sptSet ← novo vetor de tamanho V	$1A$
pred ← novo vetor de tamanho V	$1A$
Preencher dist com Integer.MAX_VALUE	nA
Preencher sptSet com falso	nA
Preencher pred com -1	nA
dist[src] ← 0	$1A$
Para count de 0 até V-1	$(n+1)C$
u ← minDistance(dist, sptSet, V)	nF
Se u == target então quebrar	$1C$
sptSet[u] ← verdadeiro	$1A$
Para v de 0 até V-1	$n \times (nC)$
Se não sptSet[v] e graph[u][v] ≠ 0 e dist[u] ≠ Integer.MAX_VALUE e dist[u] + graph[u][v] < dist[v]	$n \times (n(1C+1L+1L+1C))$
dist[v] ← dist[u] + graph[u][v]	$\leq n^2A$
pred[v] ← u	$\leq n^2A$
Construir o caminho como uma lista de arestas	F
Retornar reversedEdges	$1R$

Table 6: Method `dijkstra`

The time complexity of the `dijkstra` method is $O(n^2)$. This is because, we have two for loops iterations that iterate through every vertice, therefor $n \times n$ times `minDistance` which iterates over n vertices, resulting in $O(n \times n)$ complexity is also a factor that solidates the time complexity of dijkstra's method.

Method: addPathToFinal

The `addPathToFinal` method adds the path from the source to the target to the final path list by backtracking using the predecessor array.

Code	**Analysis**
Method addPathToFinal(pred, target, finalPath)	$O(n)$
tempPath ← nova lista	$1A$
Enquanto target ≠ -1	nC
Adicionar target a tempPath na posição 0	nA
target ← pred[target]	nA
Se finalPath não está vazia	$1C$
Remover o primeiro elemento de tempPath	$1A$
Adicionar todos os elementos de tempPath a finalPath	nA

Table 7: Method `addPathToFinal`

The time complexity of the `addPathToFinal` method is $O(n)$, where n is the number of vertices. This is because it potentially iterates through all vertices to construct the path from the target to the source.

Main differences between US017 and US018

The `main` method in each US is slightly different, where in US017 the dijkstra's algorithm is repeated for each point that we need to go throught, finding the minimum cost path from a point to another ending in the meeting point. The `main` method in the US018 the dijkstra method is called one time for each existing assembly point and then comparing the weight of each path deciding which one is the best.

US017 Analysis

Code	**Analysis**
US017 main()	$O(n^3)$
para point em mustpass[]	$n(L + A)$
edges ← dijkstra(currentSrc, point)	$n(O(n^2))$
para edge em edges[]	$n(nA)$
custo ← custo + edge.peso	nA
finalEdges.add(edge)	n^2A
curentSrc ← point	nA
edges ← dijkstra(currentSrc, point)	$O(n^2)$
para edge em edges[]	$n(L + A)$
custo ← custo + edge.peso	nL
finalEdges.add(edge)	nA

Table 8: Method `addPathToFinal`

The time complexity of the `main` method in the US017 is $O(n^3)$, where n is the number of edges. This is because the dijkstra method is iterated n times therefore $O(n \times n^2) = O(n^3)$

US018 Analysis

Code	Analysis
US018 main()	$O(n^3)$
para ap em assemblyPoints[]	$n(L + A)$
lista temPath ← nova lista	$n(A)$
array tempEdges ← dijkstra(ap, tempPath)	$n(eA + O(r^2))$
dist ← 0	$n(nA)$
para edge em edges[]	$n(nA)$
dist ← dist + edge.peso	nA
Se (dist ≤ minDist)	nC
minDist ← dist	$\leq nA$
closestAp ← ap	$\leq nA$
shortestPath ← tempEdges	$\leq n(eA)$
finalPath ← tempPath	$\leq n(vA)$

Table 9: Method `addPathToFinal`

The time complexity of the `main` method in the US018 is $O(n^3)$, where n is the number of assemblyPoints that are available, r is the nember of total edges to be analysed in the dijkstra's method, e is the number of temporary Edges and v is the number of edges in the temporary path.We also know that the number of edges represented by e or v, will always be less than r so they are not accounted in the final complexity once they are of a lesser degree than the dijkstra's method. Furthermore, both r, e and n have no rational limit, they all tend to an infinite number, so, the Expression$n(eA + O(r^2))$ can be simplified:

$$n(eA + O(r^2)) = n(nA + nO(n^2))$$

Once we make this simplification we can also conclude through the theorem 3.3 on the page 70 of the notebook that the approximate complexity of the algorithm is $O(n^3)$

Conclusion of Analysis of US17 and 18

Although with different outputs and expected results both US's have approximately the same time complexity, $O(n^3)$. In US017 the Dijkstra's algorithm is repeated with the intention of calculating the best route to each point that we need to go trough, in the US018, the Dijkstra's algorithm is repeated with the source point immutable and changing the assembly point, then comparing results and choosing the best assembly point for that one source point.