ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Degree in

Informatics Engineering

Integrative Project – US19

2nd Semester | 1st Year

G123 – Tino e os Amigos

1DK-L

Students
Rui Filipe Cardoso Silva – 1231105
Rodrigo Soares Carneiro Barbosa – 1230573
Márcio Samuel Vieira Ferreira – 1230665
João Ricardo Almeida da Fonseca – 1231138

Project developed under the orientation of professors
Ana Isabel Pereira de Moura
Stella Maria Costa de Abreu
Alexandra Antunes Gavina

Porto, June of 2024

General Index

1. US14

This US consists of an implementation of the Kruskal Algorithm. As such, given an input of a list of edges (stored using an abstract data type that will be described later) and a list of vertex names, below is a pseudocode representation of our team's implementation of the Kruskal Algorithm:

```
Procedure kruskalAlgorithm(a[ 1 ], a[ 2 ]… a[ n ]: edges, b[ 1 ], b[ 2 ]… b[ m ]: strings)
        for j:= 1 to n - 1
                for z:=1 to n – j
                        if weightOf( a [ z ] ) > weightOf( a [ z + 1 ] ) then swap a[ z ] and a[ z + 1 ]
        listEdgeCounter := 0
        for i:= 1 to m
                nodeSets[ i ] := -1
        map := empty list of edges
        while sizeOf( map ) < m-1
                edgeOrigin = indexOfEdgeOrigin( a[ listEdgeCounter ] )
                edgeEnd = indexOfEdgeEnd( a[ listEdgeCounter ] )
                originParent = nodeSets[ edgeOrigin ]
                endParent = nodeSets[ edgeEnd ]
                while (originParent not equal to -1 or endParent not equal to -1) and originParent not
                equal to endParent
                        if originParent not equal to -1 then
                                edgeOrigin = originParent
                                originParent = nodeSets[ edgeOrigin ]
                        if endParent not equal to -1 then
                                edgeEnd = endParent
                                endParent = nodeSets[ edgeEnd ]
                if endParent equals -1 and edgeOrigin not equal to edgeEnd then
                        nodeSets[ edgeEnd ] := edgeOrigin
                        add to map edge a[ listEdgeCounter ]
                else if originParent equals -1 and edgeOrigin not equal to edgeEnd then
                        nodeSets[ edgeOrigin ] := edgeEnd
                        add to map edge a[ listEdgeCounter ]
                listEdgeCounter := listEdgeCounter + 1
        return map
```

The above procedure receives two lists as inputs: A list “a” of a graph’s edges, and a list “b” of the names of a graph’s vertices. Though list “b” is simply a list of strings, list “a” is a list of an abstract data type, “edges”.

One important aspect of this abstract data type is that it stores within itself three values: First, an “origin index” value, which stores the index of the tail vertex of that edge in the “b” list. Second, an “end index” value, which does the same but for that edge’s head vertex. Third, a “weight” value, describing that edge’s weight/length. Importantly, the “origin index” and “end index” values do not use “tail” and “head” vertices in the common sense of a directed graph’s edges: The above procedure only works for non-directed graphs, and which vertex is the tail and head is completely arbitrary for the sake of this algorithm.

The functions “weightOf()”, “indexOfEdgeOrigin()”, and “indexOfEdgeEnd()” all simply get these values, which are already stored inside the “edges” data structure. Due to these functions being mere reading of values, **they can be considered primitive operations, running on a constant time.**

The same can be said for the function “sizeOf()”: This function merely returns the amount of elements in a list of edges, which is a value already stored inside the data structure of a list. And, of course, the same applies to the procedures of swapping two elements in a list and adding an element to a list, being mere attributions. **All of these functions and procedures are primitive operations.**

With these facts in mind, we can proceed to the worst-case time complexity analysis.

LINE NUMBER	Nº OF ITERATIONS	O(x) APPROXIMATION
Line 1	n – 1 + 1	O(n)
Line 2	(n * (n – i) / 2) + n	O(n^2)
Line 3	n * (n – i) / 2	O(n^2)
Line 4	1	O(1)
Line 5	m + 1	O(m)
Line 6	m	O(m)
Line 7	1	O(1)
Line 8	n + 1	O(n)
Line 9	n	O(n)
Line 10	n	O(n)
Line 11	n	O(n)
Line 12	n	O(n)
Line 13	n * m + n	O(n * m)
Line 14	n * m	O(n * m)
Line 15	n * m	O(n * m)
Line 16	n * m	O(n * m)
Line 17	n * m	O(n * m)

Line 18	n * m	O(n * m)
Line 19	n * m	O(n * m)
Line 20	n	O(n)
Line 21	n	O(n)
Line 22	n	O(n)
Line 23	n	O(n)
Line 24	n	O(n)
Line 25	n	O(n)
Line 26	n	O(n)
Line 27	1	O(1)

We believe that the reason for these iteration numbers is trivial on all lines except 1, 2, 5, 8, and 13. Thus, the reason for the iteration numbers of those lines will be explained while all others will be disregarded.

It must be noted that there are two unknown variables in this procedure: “n” and “m”. These variables are, respectively, the amount of edges on the input graph (which we will use as a measure of the input size), and the amount of vertices on the input graph. The connection between these variables will be explained at the end: For now, both variables will be used in the calculation of time complexity.

The first three lines of the procedure consist of a sorting method typically referred to as a Bubble Sort. This method will sort the edges in the procedure’s input from lowest to highest weight. The first line of this sorting method will be performed n-1 times, and for every iteration “i” of the first line, the second line is performed n-i times, with its contents being a comparison that will be performed n-i times, and a swap that will be performed, at most, n-i times as well. With this in mind, the total amount of comparisons and swaps made, which is also the total number of iterations of the second line, can be given as the sum of n-1 over n-i times: A mathematical expression which can be simplified to (n (n – i) / 2). Accounting additionally for the additional single iteration of the first line and the additional total of n iterations for the second line as the exits of these loops, and applying the theorems taught to us in the MDISC UC, we find that this algorithm’s worst time complexity can be defined as O(n) + O(n^2) + O(n^2) + O(n^2), which results in a final time complexity of O(n^2).

The fifth and sixth lines of the procedure happen m times (with an additional exit iteration for the fifth line) as they iterate through the amount of vertices to initialize the nodeSets list, giving this loop a complexity of O(m).

The eight line a while loop that will be responsible for adding every necessary edge onto the final minimum cost spanning tree. This while loop will finish once the amount of edges in the tree equals the amount of vertices -1: The worst case scenario for this process is that the last edge to be added is the edge with the highest weight, which, due to the sorting mentioned above, is the very last edge in the list. This would make this while loop run n times, plus an exit iteration on the eighth line itself, giving it a time complexity of O(n).

Inside this while loop, the 13th line is another while loop. This while loop is more complex: Due to Java not natively implementing a system of Disjoint Sets (also known as Union-Find sets), which the Kruskal Algorithm relies on, we are forced to implement some method of keeping track of which vertices belong to which sets. The way in which we achieve this is by assigning “ancestors” to each vertex: When an edge is added to the spanning tree, one vertex from that edge is registered in the nodeSets list as the ancestor of the other. Thus, to check if two vertices belong to the same set, we simply backtrack through each vertice’s ancestors until we reach a vertex that has no ancestors. Should this vertex be the same for both vertices we want to test, we will know that the two vertices are both in the same set: Otherwise, they are in different sets. This can be said to be our implementation of the Find operation in a Union-Find set.

The 13th line is the while loop that handles this detection. Due to the method described above relying on back-tracking through a vertice’s ancestors, which are other, separate vertices, we can deduce that the worst case for this loop would be having to iterate through every single vertex: Meaning, m iterations, m being the number of vertices in the input graph. Therefore, in a worst case analysis, this loop will run m times, for all n iterations of the 8th line loop, plus an additional n exit iterations: Thus, a number of iterations of n * m + n, which translates into a time complexity of O(n * m).

With all the important lines explained, we will advance to explaining the connection between “n” and “m”.

Given that “n” represents the number of edges of the input graph, and “m” represents the number of vertices of said input graph, to create a connection between the two we will require a formula which ties the number of edges with the number of vertices. For this purpose, we will use the Handshaking Lemma, presented below:

Let G = (V, E) be a non-directed graph. Then

$$2|E| = \sum_{v \in V} degree(v).$$

In simpler terms, the Handshaking Lemma dictates that the sum of the degrees of all the vertices in a graph is twice the number of edges in the same graph.

With this in mind, the connection between “n” and “m” becomes clear: The worst case, in which “m” is as high as possible, is the one in which every vertex in the graph has a degree of 1. With this in mind, we can state that:

$$2|E| = \sum_{v \in V} 1 \leftrightarrow 2|E| = V(G).$$

Meaning that in the worst case, m = 2n.

With this we can state that n * m = n * 2n = 2n^2, meaning that the time complexity of lines 13-19 becomes O(n^2) and the time complexity of lines 5-6 becomes O(n). Applying this correction and applying the known theorems taught to us in the MDISC UC, we can deduce that the final time complexity for this algorithm is O(n^2).

2. US17

This US consists essentially of an implementation of the Dijkstra Algorithm. Thus, given an input of a square matrix of positive integers that represent edge weights (or 0 if no edge exists between two vertices) and an input of an array of strings that represents each vertice's name, both ordered in the same way so that the string array can serve as an index to the integer matrix, as well as an integer representing the index of the desired end vertex (the Assembly Point), below is the pseudocode representation of our team's Dijkstra Algorithm:

```
Procedure dijkstraAlgorithm(a[1][1], a[1][2],… a[n][n]: integers, b[1], b[2],... b[n]: strings, c: integer)
        nodeAncestors := empty list of integers
        routeLengths := empty list of integers
        markedTemporary := empty list of integers
        for i := 1 to n
                if i not equal to c then
                        add to nodeAncestors value -1
                        add to routeLengths value infinity
                        add to markdTemporary value i
                else
                        add to nodeAncestors value c
                        add to routeLengths value 0
        minimumMarkNode := c
        while sizeOf( markedTemporary ) > 0
                for every integer nodeIndex in markedTemporary
                        if a[ nodeIndex ][ minimumMarkNode ] equals 0 then
                                continue
                        valueToCompare := a[ nodeIndex ][ minimumMarkNode ] + routeLengths[mini-
                        mumMarkNode]
                        if valueToCompare < a[ nodeIndex ][ minimumMarkNode ] then
                                valueToCompare = a[ nodeIndex ][ minimumMarkNode ]
                        routeLengths[ nodeIndex ] = valueToCompare
                        nodeAncestors[ nodeIndex ] = minimumMarkNode
                minimumMarkNode := markedTemporary[ 1 ]
                minimumMarkValue := routeLengths[ minimumMarkNode ]
                minimumMarkIndex := 1
                for i := 1 to sizeOf( markedTemporary )
                        if routeLengths[ markedTemporary[ i ] ] < minimumMarkValue
                                minimumMarkNode = markedTemporary[ i ]
                                minimumMarkIndex = i
                                minimumMarkValue = routeLengths[ minimumMarkNode ]
                remove value at index minimumMarkIndex from markedTemporary
        return nodeAncestors and routeLengths
```

As stated previously, the above procedure receives three inputs: First, a square matrix “a” of inputs that represents every edge’s weight, with a side length equal to the number of vertices in the graph. Second, an array “b” of strings that represents every vertice’s name, ordered in the same way as the ordering of the vertices in the “a” matrix. Lastly, an integer “c” that represents the index of the point we wish to calculate all shortest paths to: In the context of the US, this would be the Assembly Point.

Much like in US14, some technical matters must be addressed: First of all, it should be noted that the method “sizeOf()” functions the same way here as it did in US14: It merely reads a value that represents the array’s length. Much like in US14, then, this method is a primitive operation. The same can be said for the “add” operations performed here, which function exactly as in US14. The same can also be said for the “continue” operation, which just advances the “for” loop it is in to the next iteration. **All of these operations are therefore primitive.**

However, by contrast, **the “remove” operation performed at the end of the algorithm is not primitive.** In mathematics, removing an element from a list or a set is considered a primitive operation: However, in Java, this is not possible. Due to technical constraints, the method that we use, in Java, to remove an element from a list, is to create a whole backup list that contains all elements of the list other than the one we wish to remove. Then, we empty the first list, and add every element of the second list into it. This results in an operation that has to iterate through every element of the list, giving it a time complexity of O(n). For simplicity’s sake we represent the remove operation with a single line rather than a full algorithm, but please remember that **this operation is not primitive.**

Finally, it must be addressed how, at the very end of the algorithm, this procedure returns 2 objects: The “nodeAncestors” list and the “routeLengths” list. The reader may note that, in Java, much like most programming languages, this is not possible, and indeed, that’s not what the actual program is doing: In the actual program, the “nodeAncestors” and “routeLengths” objects are being stored in an abstract data type, much like the edges in US14, and what is actually returned is an object of said abstract data type that contains the “nodeAncestors” and “routeLengths” lists. Due to this abstract data type’s irrelevance in the pseudocode interpretation of this algorithm, we have elected to omit its use. **This fact will, however, become relevant in the analysis of US18.**

With these facts in mind, we can proceed to the worst-case time complexity analysis.

LINE NUMBER	Nº OF ITERATIONS	O(x) APPROXIMATION
Line 1	1	O(1)
Line 2	1	O(1)
Line 3	1	O(1)
Line 4	n + 1	O(n)
Line 5	n	O(n)
Line 6	n	O(n)
Line 7	n	O(n)
Line 8	n	O(n)
Line 9	n	O(n)
Line 10	n	O(n)
Line 11	n	O(n)
Line 12	1	O(1)
Line 13	n – 1 + 1	O(n)
Line 14	(n – 1) (n – 1) + n	O(n^2)
Line 15	(n – 1) (n – 1)	O(n^2)
Line 16	(n – 1) (n – 1)	O(n^2)
Line 17	(n – 1) (n – 1)	O(n^2)
Line 18	(n – 1) (n – 1)	O(n^2)
Line 19	(n – 1) (n – 1)	O(n^2)
Line 20	(n – 1) (n – 1)	O(n^2)
Line 21	(n – 1) (n – 1)	O(n^2)
Line 22	n - 1	O(n)
Line 23	n - 1	O(n)
Line 24	n - 1	O(n)
Line 25	(n – 1) (n – 1) + n	O(n^2)
Line 26	(n – 1) (n – 1)	O(n^2)
Line 27	(n – 1) (n – 1)	O(n^2)
Line 28	(n – 1) (n – 1)	O(n^2)
Line 29	(n – 1) (n – 1)	O(n^2)
Line 30	(n – 1) (n – 1) + n	O(n^2)
Line 31	1	O(1)

Much like in US14, we believe many of these lines have trivial explanations. Thus, only a select few lines will have their iteration count and time complexity explained: Lines 4, 13, 14, 25, and 30. There is also only one variable of input size in this algorithm: “n”, the number of vertices in the graph.

Line 4 consists of a for loop that initializes the nodeAncestors, routeLengths, and markedTemporary lists. All these lists store information about each of the graph’s vertices, so they all have a size equal to the number of vertices (that is to say, “n”), so its initialization is a loop that runs n times, with an additional exit iteration. The purpose of these lists is simple: nodeAncestors registers the index of each vertice’s ancestor in their path to the start point. routeLength registers the total lengths (the sum of all the weights) of each vertice’s path to the start point. Finally, markedTemporary registers the index of every vertice which is currently defined as having only a temporary path: That is to say, every vertice for which the true shortest path has not been found yet. All of these lists will be of great importance later in the algorithm.

Line 13 consists of a while loop which will handle the main functions of the Dijkstra Algorithm. In this loop, every vertice marked as temporary will be iterated over, and a shortest path towards the start point for it will be found: First by finding every vertice connected by an edge to the vertice currently registered as the “Marked Minimum” vertice (the vertice which was marked as permanent in the previous iteration of this while loop, or the start vertice itself on the first iteration), checking if it’s advantageous to change that vertice’s path to the start point to pass through the “Marked Minimum” vertice, applying that change if necessary, and after repeating that process for every vertice marked as temporary, finding a new vertice to mark as the “Marked Minimum” vertice and remove from the list of temporary vertices. This whole process will iterate through every vertice marked as temporary: And given that at the start of the algorithm this applies to every vertice except for the start vertice itself, we can say that the worst case scenario for this while loop is having to iterate through every vertice except one. Thus we apply that notion and approximate its time complexity to O(n).

Line 14’s functions were well described in the previous paragraph: Line 14 is a for loop that handles the process of finding temporary vertices connected to the “Marked Minimum” vertice, checking if their paths to the start point should be changed to run through the “Marked Minimum” vertice, and applying that change if necessary. Of course, as this description makes clear, this means that this loop will iterate over every vertice marked as temporary: As previously stated, this process’ worst case time complexity is O(n), except being that is being run inside another O(n) for loop, the time complexity becomes O(n^2).

This exact same logic can be applied to Lines 25 and 30: These loops are responsible for iterating over every vertice marked as temporary and, in the case of Line 25, find a new “Marked Minimum” vertice (a vertice with the shortest path to the start point), and in the case of Line 30, mark it as permanent by removing it from the temporary list. Both of these iterate over each vertice marked as temporary: As discussed, this process being run inside another O(n) loop gives it a time complexity of O(n^2).

Applying the MDISC UC’s theorems to these results, the final time complexity of the algorithm becomes O(n^2).

3. US18

This US consists of an algorithm that makes use of US17's Dijkstra Algorithm implementation, with the caveat that this time, there are several Assembly Points rather than one. The inputs for this algorithm are therefore the same as the Dijkstra Algorithm's implementation, except instead of an integer representing the desired end point, it instead receives a list of integers: The indexes of all the Assembly Points. Below is the pseudocode:

```
Procedure multipleAP(a[1][1], a[1][2]… a[n][n]: integers, b[1], b[2]… b[n]: strings, c[1], c[2]… c[n]: integers)
    returnShortestRoutes := empty ShortestRoutes object
    allAPShortestRoutes := empty list of ShortestRoutes objects
    for every integer nodeIndex in c
        add return of dijkstraAlgorithm( a, b, nodeIndex) to allAPShortestRoutes
    returnShortestRoutes.nodeAncestors := empty list of integers
    returnShortestRoutes.routeLengths := empty list of integers
    for i := 1 to n
        isAP := false
        for every integer nodeIndex in c
            if nodeIndex equals i then
                isAP := true
        if isAP equals true
            add i to returnShortestRoutes.nodeAncestors
            add 0 to returnShortestRoutes.routeLengths
            continue
        add -1 to returnShortestRoutes.nodeAncestors
        add infinity to returnShortestRoutes.routeLengths
        for every ShortestRoute route in allAPShortestRoutes
            if route.routeLenghts[ i ] < returnShortestRoutes.routeLengths[ i ] then
                returnShortestRoutes.routeLengths[ i ] = route.routeLengths[ i ]
                returnShortestRoutes.nodeAncestors[ i ] = route.nodeAncestors[ i ]
    return returnShortestRoutes
```

Once again, some clarifications must be made regarding this algorithm's function: In particular, the "ShortestRoutes" data type used in this procedure. As mentioned in the US17 chapter, the real, java implementation of the dijkstra algorithm returns an object of an abstract data type that contains two lists: routeLengths and nodeAncestors. "ShortestRoutes" is this data type, and in this algorithm, we elected to represent the data type's existance to simplify the notation used.

In the same vein, it must be noted that the usage of notation like "returnShortestRoutes.nodeAncestors" and the such merely represents reading the nodeAncestors or routeLengths lists inside a ShortestRoutes object. These are therefore mere reading operations, and are therefore primitive. And, for the same reason as explained in US14 and US17, the add and continue operations are also primitive. **All of these operations are considered primitive.**

With these facts in mind, below is the worst case time complexity analysis of this algorithm.

LINE NUMBER	Nº OF ITERATIONS	O(x) APPROXIMATION
Line 1	1	O(1)
Line 2	1	O(1)
Line 3	n + 1	O(n)
Line 4	n * n^2 + n	O(n^3)
Line 5	1	O(1)
Line 6	1	O(1)
Line 7	n + 1	O(n)
Line 8	n	O(n)
Line 9	n * n + n	O(n^2)
Line 10	n * n	O(n^2)
Line 11	n * n	O(n^2)
Line 12	n	O(n)
Line 13	n	O(n)
Line 14	n	O(n)
Line 15	n	O(n)
Line 16	n	O(n)
Line 17	n	O(n)
Line 18	n * n + n	O(n^2)
Line 19	n * n	O(n^2)
Line 20	n * n	O(n^2)
Line 21	n * n	O(n^2)
Line 22	1	O(1)

In this algorithm, we feel like the only lines worth explaining the complexity of are 3, 4, 7, 9, and 18. Additionally, much like in US17, there is only one unknown variable of input size: “n”, the number of vertices.

Line 3 consists of an iteration through every Assembly Point in the graph in order to register all shortest routes from every other vertex to that Assembly Point. Logically, the worst case scenario for this loop is that every vertex is an assembly point, making this loop require n iterations, plus an additional exit iteration, giving us a result of an O(n) time complexity.

Line 4 consists of the usage of the Dijkstra Algorithm as implemented in US17 to get all shortest routes from each vertex to the specified assembly point. The Dijkstra Algorithm, as previously proven, runs on a worst-case time complexity of O(n^2): However, being that this algorithm is being run inside a loop with a worst-case scenario of running “n” times, this line ends up having a time complexity of O(n^3), as each O(n^2) iteration of the Dijkstra Algorithm runs inside a loop of O(n) complexity: Applying the rules of multiplication of time complexity, this line’s final time complexity becomes O(n^3).

Line 7 is the beginning of a loop which will extract the true shortest route to any Assembly Point from the results of the previous iteration: This loop is based on the idea of iterating through every vertex in the graph and checking if it is an Assembly Point or not: If it’s an Assembly Point, we register the shortest path of that vertex to any Assembly Point as an immediate path to itself of weight 0. If it’s not an Assembly Point, we search through the results of the previous loop to find which path to an Assembly Point from that vertex is shortest and choose that one. This process requires, as is obvious, iterating through every vertex in the graph: “n” iterations plus an exit iteration. Thus, this line ends with a time complexity of O(n).

Finally, Lines 9 and 18 simply perform the functions described earlier: Finding out if the vertex currently being iterated over is an Assembly Point in the case of Line 9, and finding which Assembly Point provides the shortest route to it in the case of Line 18. Both of these require iterating over every Assembly Point: Thus, in the worst possible case of every vertex being an Assembly Point, both of these loops would contain “n” iterations plus an additional exit iteration. Being that these loops are being run inside another loop that contains “n” iterations, we apply the rules of complexity multiplication to arrive at a final time complexity of O(n^2) for both of these lines.

With these operations explained, we apply all the theorems taught to us in the MDISC UC to calculate the final time complexity, and we arrive at the conclusion that this algorithm, due entirely to the time complexity of Line 4 and its use of the O(n^2) Dijkstra Algorithm procedure inside an O(n) loop, has a final worst case time complexity of O(n^3): The highest time complexity of all the algorithms developed in this project.