**OBSERVACIONES DEL LA PRACTICA**

Lindsay Vanessa Pinto Morato Cod. 202023138

José Daniel Montero Cod. 202012732

**CONTENIDO**

[**EQUIPO DE PRUEBAS** 1](#_Toc73444864)

[**CONTROLES EN LAS PRUEBAS Y RESULTADOS** 2](#_Toc73444865)

[**COMPLEJIDAD** 2](#_Toc73444866)

[**ANALISIS DE CONSUMO DE MEMORIA Y TIEMPO** 3](#_Toc73444867)

[**MAQUINA 1** 3](#_Toc73444868)

[**MAQUINA 2** 4](#_Toc73444869)

[**CONCLUSIONES** 4](#_Toc73444870)

**LISTA DE TABLAS**

[Table 1. Especificaciones computadores 1](#_Toc73444871)

[Table 2. Complejidad temporal requerimientos 2](#_Toc73444872)

[Table 3. Comparación de consumo de datos y tiempo de ejecución para cada uno de los requerimientos en la Maquina 1. 4](#_Toc73444873)

[Table 4. Comparación de consumo de datos y tiempo de ejecución para cada uno de los requerimientos en la Maquina 2. 4](#_Toc73444874)

**LISTA DE ILUSTRACIONES**

**No se encuentran elementos de tabla de ilustraciones.**

# **EQUIPO DE PRUEBAS**

Table 1. Especificaciones computadores

|  |  |
| --- | --- |
|  | Máquina 1 |
| Marca | Computador Huawei Matebook D14 |
| Procesadores | AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10GHZ |
| Memoria RAM (GB) | 8,00GB (6.94 utilizable) |
| Sistema Operativo | Windows 10 Home Single languaje |
| Tipo de sistema | Sistema operativo de 64bits, procesador x64 |

|  |  |
| --- | --- |
|  | Máquina 2 |
| Marca | Personalizado |
| Procesadores | AMD Ryzen 5 2600X Hexa-core @ 3.6 GHz |
| Memoria RAM (GB) | 8.00 GB @ 2133 MHz |
| Sistema Operativo | Windows 10 Pro |
| Tipo de sistema | Sistema operativo de 64 bits, procesador x64 |

# **CONTROLES EN LAS PRUEBAS Y RESULTADOS**

Las pruebas se realizaron utilizando los mismos parámetros en cada uno de los equipos de esta forma:

* Se cerraron todas las aplicaciones a excepción de Visual Studio Code
* Se realizaron las pruebas con el cargador conectado
* **Para el requerimiento 1** se utilizaron las entradas de landing point 1: Redondo Beach, CA, United States y landing point 2: Vung Tau, Vietnam. Se aclara que se utilizó el nombre completo pues algunos landing points tienen el mismo nombre pero su ubicación es en diferente país.
* Para el **requerimiento 3** se utilizó como país 1 Colombia y país 2 Indonesia.

## **COMPLEJIDAD**

Table 2. Complejidad temporal requerimientos

|  |  |  |
| --- | --- | --- |
| **Requerimiento** | | **O(n)** |
| 1 | Encontrar cantidad de clusteres dentro de la red de cables submarinos | -> Lineal |
| 2 | Encontrar landing points que sirven como punto de interconexión a más cables | O(V + E) -> Lineal |
| 3 | Encontrar la ruta mínima para enviar información entre dos países | -> Linearítmica |
| 4 | Identificar la infraestructura crítica para garantizar el mantenimiento preventivo | O(n2) -> Cuadrática |
| 5 | Conocer el impacto que tendría el fallo de un determinado landing point | -> Cuadrática |

A continuación se muestra el análisis detallado de estas complejidades

**REQUERIMIENTO 1: Encontrar cantidad de clusteres dentro de la red de cables submarinos**

|  |  |
| --- | --- |
| def findClusters(analyzer, landingPoint1, landingPoint2):  scc = kos.KosarajuSCC(analyzer['connections'])  vertexAId = me.getValue(mp.get(analyzer['landingPointNames'], landingPoint1))  vertexBId = me.getValue(mp.get(analyzer['landingPointNames'], landingPoint2))  lpA = me.getValue(mp.get(analyzer['landingPoints'], vertexAId))  lpB = me.getValue(mp.get(analyzer['landingPoints'], vertexBId))  vertexA = lt.getElement(lpA['vertices'], 1)  vertexB = lt.getElement(lpB['vertices'], 1)  return kos.connectedComponents(scc), kos.stronglyConnected(scc, vertexA, vertexB) | O (V+E) -> lineal  O(1)  O(1)  O(1)  O(1)  O(1)  O(1) |
| Llamaremos a O(1) = C y se realizará suma de cada una de las complejidades  Resolviendo:  Por complejidad asintótica se tiene que la complejidad del algorítmo es de La cual es lineal | |

**REQUERIMIENTO 2: Encontrar landing points que sirven como interconexión a más cables**

|  |  |
| --- | --- |
| def findInterLandingPoints(analyzer):      total = 0      iterator = lli.newIterator(gr.vertices(analyzer["arches"]))      vList = lt.newList()      while lli.hasNext(iterator):          vertex = lli.next(iterator)          inDeg = gr.indegree(analyzer["arches"], vertex)          outDeg = gr.outdegree(analyzer["arches"], vertex)          if inDeg >= 1 and outDeg >1:              total += 1              lt.addLast(vList, vertex)      final = lt.newList()      nIterator = lli.newIterator(vList)      while lli.hasNext(nIterator):          elt = lli.next(nIterator)          cpl = mp.get(analyzer["countriesCodes"], elt)          value = me.getValue(cpl)          lt.addLast(final, value["id"])          lt.addLast(final, value["name"])      return total, final | O(1)  O(1)  O(1)  O(1)  O(V+E)  O(V+E)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1) |
| Llamaremos a O(1) = C y realizamos una suma de cada una de las complejidades:  C + C + C( C + O(V+E) + O(V+E) + C( C ) ) + C + C + C( C + C + C + C + C )  Resolviendo:  2C + C( C + 2(O(V+E)) + C2 ) + 2C + C( 5C )  4C + C2 + C3 + 2C(V+E) + 5C2  4C + 6C2 + C3 + 2C(V+E)  Se devuelve C = 1:  4(1) + 6(1)2 + (1)3 + 2(V+E)  2(V+E)  Por complejidad asintótica, se tiene que la complejidad del algorítmo es de O(V+E), la cual es lineal. | |

**REQUERIMIENTO 3: Encontrar la ruta mínima para enviar información entre dos países**

|  |  |
| --- | --- |
| def findShortestPath(analyzer, pais1, pais2):  country1 = me.getValue(mp.get(analyzer['countries'], pais1))  vertex1 = country1['vertex']  country2 = me.getValue(mp.get(analyzer['countries'], pais2))  vertex2 = country2['vertex']  search = djk.Dijkstra(analyzer['connections'], vertex1)  dist = djk.distTo(search, vertex2)  path = djk.pathTo(search, vertex2)  return path, dist | O(1)  O(1)  O(1)  O(1)  E log (v)  O(1) -> De acuerdo al algoritmo en la libreria  O(1) -> De acuerdo al algoritmo en la libreria |
| Llamaremos a O(1) = C y se realizará suma de cada una de las complejidades  Resolviendo:  Por complejidad asintótica se tiene que la complejidad del algorítmo es de la cual es linearítmica. | |

**REQUERIMIENTO 4: Identificar la infraestructura crítica para garantizar el mantenimiento preventivo**

|  |  |
| --- | --- |
| def criticalInfrastructure(analyzer):      vertex = gr.numVertices(analyzer["connections"])      tree = pr.PrimMST(analyzer["connections"])      weight = pr.weightMST(analyzer["connections"], tree)      branch = pr.edgesMST(analyzer["connections"], tree)      branch = branch["edgeTo"]["table"]["elements"]      max = 0      for i in range(len(branch)):          value = branch[i]["value"]          if (value != None) and (float(value["weight"]) > max):              max = value["weight"]  return vertex, weight, max | O(1)  O((V+E) log V) -> De acuerdo al algoritmo en la librería  O(1)  O(1)  O(1)  O(n)  O(n)  O(n)  O(1) |
| Llamaremos a O(1) = C y se realizará la suma de cada una de las complejidades:  C + O((V+E) log V) + C + C + C + n( n + n(C) )  Resolviendo:  4C + O((V+E) log V) + n( n + nC )  4C + O((V+E) log V) + n2 +n2C  Se devuelve C = 1:  4(1) + O((V+E) log V) + n2 + n2(1)  O((V+E) log V) + 2n2  Por complejidad asintótica, se tiene que la complejidad del algorítmo es de O(n2), la cual es cuadrática. | |

**REQUERIMIENTO 5: Conocer el impacto que tendría el fallo de un determinado landing point**

|  |  |
| --- | --- |
| def failImpact(analyzer, landingPoint):  vertexId = me.getValue(mp.get(analyzer['landingPointNames'], landingPoint))  lp = me.getValue(mp.get(analyzer['landingPoints'], vertexId))  countryName = lp['country']  country = me.getValue(mp.get(analyzer['countries'], countryName))  countries = mp.newMap(numelements=300, maptype='PROBING')  for vertex in lt.iterator(lp['vertices']):  adjVertices = gr.adjacents(analyzer['connections'], vertex)  for adjVertex in lt.iterator(adjVertices):  if adjVertex == country['vertex']:  if not mp.contains(countries, countryName):  ctryEdge = gr.getEdge(analyzer['connections'], vertex, country['vertex'])  mp.put (countries, countryName, ctryEdge['weight'])  else:  adjVertexId = adjVertex.split('-')[0]  adjLP = me.getValue(mp.get(analyzer['landingPoints'], adjVertexId))  if not mp.contains(countries, adjLP['country']):  adjEdge = gr.getEdge(analyzer['connections'], vertex, adjVertex)  mp.put (countries, adjLP['country'], adjEdge['weight'])    countriesList = mp.keySet(countries)  result = om.newMap('BST')  for ctry in lt.iterator(countriesList):  distance = me.getValue(mp.get(countries, ctry))  if om.contains(result, distance):  ctryList = me.getValue(om.get(result, distance))  lt.addLast(ctryList, ctry)  else:  newList = lt.newList('SINGLE\_LINKED', compareCountries)  lt.addLast(newList, ctry)  om.put(result, distance, newList)  return mp.size(countries), result | O(1)  O(1)  O(1)  O(1)  O(1)  O(n)  O(1)  O(n)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(log(n))  O(n)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1) |
| Llamaremos a O(1) = C y se realizará suma de cada una de las complejidades  Resolviendo:  Devolviendo C = 1  Por complejidad asintótica se tiene que la complejidad del algorítmo es de La cual es cuadratica. | |

## **ANALISIS DE CONSUMO DE MEMORIA Y TIEMPO**

### **MAQUINA 1**

**Resultados**

Table 3. Comparación de consumo de datos y tiempo de ejecución para cada uno de los requerimientos en la Maquina 1.

|  |  |  |
| --- | --- | --- |
|  | | |
| REQUERIMIENTO | **CONSUMO DE DATOS [KB]** | **TIEMPO DE EJECUCIÓN [MS]** |
| Carga | 1046,019 | 510,608 |
| 1 | 117,632 | 4042,227 |
| 2 |  |  |
| 3 | 21,044 | 1221,357 |
| 4 | 18,773 | 1528,92 |
| 5 | 22,134 | 6,348 |

### **MAQUINA 2**

**Resultados**

Table 4. Comparación de consumo de datos y tiempo de ejecución para cada uno de los requerimientos en la Maquina 2.

|  |  |  |
| --- | --- | --- |
|  | | |
| REQUERIMIENTO | **CONSUMO DE DATOS [KB]** | **TIEMPO DE EJECUCIÓN [MS]** |
| Carga | 4626,8756 | 865,4718 |
| 1 | 121,2606 | 3682,378 |
| 2 | 587,539 | 69,505 |
| 3 | 27,0874 | 1114,013 |
| 4 | 19,6922 | 1390,254 |
| 5 | 22,9188 | 2,536 |

# **CONCLUSIONES**