Análisis de Complejidad

Nombre: Nicolas Merchan Cuestas

Código: 202112109

Correo: n.merchan@uniandes.edu.co

Notas:

- Los resultados de las pruebas de tiempos de ejecución y las correspondientes gráficas de los mismos se encuentran en '*Datos - Análisis de Complejidad.xslx*'.
- El programa 'Test_Function.py' realiza las pruebas de tiempos de ejecución de manera automática e ingresa los resultados en un archivo EXCEL llamado 'Test_Data.xlsx'.
- En el presente documento **'E'** representará el número de arcos en los grafos considerados. Ello se debe a que en todas las pruebas se utilizaron los archivos completos de las ciudades y los aeropuertos (**worldcities-utf8.csv** y **airports-utf8-large.csv**, respectivamente). Por lo tanto, el parámetro de comparación utilizado en las pruebas fue el número de rutas registradas.

Requerimiento 1

```
249 def Requirement1(catalog):
250     airports_list = catalog['airports_list']
251     digraph = catalog['digraph']
252     interconnections_RBT = rbt.newMap(Requirement1cmpFunction)
253 
254     num_connected_airports = 0
255     for airport in lt.iterator(airports_list):
256         IATA = airport['IATA']
257         airport_indegre = gp.indegree(digraph, IATA)
258         airport_outdegree = gp.outdegree(digraph, IATA)
259         num_interconnections = airport_indegre + airport_outdegree
260         element = airport, num_interconnections, airport_indegre, airport_outdegree
261         rbt.put(interconnections_RBT, (num_interconnections, airport_indegre, airport_outdegree), element)
262         if num_interconnections > 0:
263             num_connected_airports += 1
264 
265     requirement_list = inorder(interconnections_RBT)
266     requirement_list = lt.subList(requirement_list, 1, 5)
267 
268     return lt.iterator(requirement_list), num_connected_airports
```

La complejidad del requerimiento 1 está dada por la función **Requirement1()**. El proceso implementado para encontrar los aeropuertos más interconectados requiere obtener el grado de todos los aeropuertos (vértices) del grafo dirigido e ingresar dichos aeropuertos en un RBT (Red-Black Tree) en base a al grado de cada uno. Después, se realiza un recorrido inorder sobre el RBT en cuestión para obtener una lista de los aeropuertos ordenados en función de su grado de manera descendente. Finalmente, la complejidad del requerimiento 1 respecto al número de rutas (arcos) es **O(1)**, dado que tanto el proceso de búsqueda en la lista de adyacencias, como el proceso de adición de los aeropuertos en el RBT no dependen del número de rutas.

Requerimiento 2

```
272 def Requirement2(catalog, airport_1, airport_2):
273     digraph = catalog['digraph']
274     airports_map = catalog['airports_map']
275 
276     airports_info_list = lt.newList('ARRAY_LIST')
277     for airport in [airport_1, airport_2]:
278         airport_key_value = mp.get(airports_map, airport)
279         aiport_info = me.getValue(airport_key_value)
280         lt.addLast(airports_info_list, aiport_info)
281     airports_info_list = lt.iterator(airports_info_list)
282 
283     SCC = scc.KosarajuSCC(digraph)
284     num_SCC = scc.connectedComponents(SCC)
285     answer = scc.stronglyConnected(SCC, airport_1, airport_2)
286 
287     return airports_info_list, num_SCC, answer
```

La complejidad del requerimiento 2 está dada por la función **Requirement2()**. El proceso implementado para encontrar los componentes fuertemente conectados (SCC) en el grafo dirigido conformado por aeropuertos (vértices) y rutas (arcos) implica utilizar el algoritmo de Kosaraju. Así mismo, con el fin de conocer si dos aeropuertos pertenecen al mismo SCC solamente es necesario comparar los valores referentes al SCC al que pertenece cada aeropuerto y comparar si son iguales. La complejidad asociada al algoritmo de Kosaraju con respecto al número de arcos es **O(E)**, dado que es necesario recorrer todos los arcos al menos una vez en su implementación. Además, la verificación de pertenencia al mismo SCC de dos aeropuertos tiene una complejidad de **O(1)**, dado se consulta una tabla de Hash generada en el algoritmo. Finalmente, la complejidad del requerimiento 2 respecto al número de rutas es **O(E)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 3

```
291  def Requirement3(catalog, choosen_cities):
292      digraph = catalog['digraph']
293      routes_map = catalog['routes_map']
294      airports_map = catalog['airports_map']
295
296      origin_city = choosen_cities[0]
297      destiny_city = choosen_cities[1]
298
299      oringin_RBT = origin_city['RBT']
300      destiny_RBT = destiny_city['RBT']
301
302      origin_airport_list = inorder(oringin_RBT)
303      destiny_airport_list = inorder(destiny_RBT)
304
305      origin_airport_list_size = lt.size(origin_airport_list)
306      destiny_airport_list_size = lt.size(destiny_airport_list)
307      path = False
308      index_1 = 1
```

```
309      while path == False and index_1 <= origin_airport_list_size:
310          origin_airport = lt.getElement(origin_airport_list, index_1)
311          origin_airport_info = origin_airport[0]
312          origin_airport_IATA = origin_airport_info['IATA']
313          Dijkstra_path = Dijkstra(digraph, origin_airport_IATA)
314          index_2 = 1
315          while path == False and index_2 <= destiny_airport_list_size:
316              destiny_airport = lt.getElement(destiny_airport_list, index_2)
317              destiny_airport_info = destiny_airport[0]
318              destiny_airport_IATA = destiny_airport_info['IATA']
319              path_key_value = mp.get(Dijkstra_path['visited'], destiny_airport_IATA)
320              path = me.getValue(path_key_value)
321              distance = path['distTo']
322
323              if distance != inf:
324                  path == True
325              else:
326                  index_2 += 1
327          index_1 += 1
```

La complejidad del requerimiento 3 está dada por la función **Requirement3()**. El proceso implementado para encontrar la ruta más corta entre dos aeropuertos (vértices) por medio de rutas (arcos) en el grafo dirigido solamente implica utilizar el algoritmo de Dijkstra iniciando del aeropuerto de salida y consultando el camino generado al aeropuerto de llegada. Así mismo, antes de ejecutar el algoritmo de Dijkstra es necesario consultar las ciudades asociadas a un nombre homónimo. Ello se logra agrupando todas las ciudades homónimas en una lista que es el valor asociado a la llave del nombre de las ciudades en una tabla de Hash. La complejidad asociada al algoritmo de Dijkstra respecto al número de rutas es **O(log(E))**, dado que, en el caso particular de

los datos analizados, el número de arcos entre aeropuertos incrementa de manera logarítmica con el incremento de rutas. Ello se debe a que existen muchas rutas en **routes-utf8-large.csv** que representan los mismos arcos en los grafos cargados. Además, la complejidad de la consulta de ciudades homónimas es **O(1)**¸ dado que solo se realiza una consulta en una tabla de Hash. Finalmente, la complejidad del requerimiento 3 respecto al número de rutas es **O(log(E))**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 4

```
366  def Requirement4(catalog, choosen_city, miles):
367      routes_map = catalog['routes_map']
368      city_RBT = choosen_city['RBT']
369      graph= catalog['graph']
370      airports_list_city = inorder(city_RBT)
371      airport_info = lt.getElement(airports_list_city, 1)[0]
372      IATA = airport_info['IATA']
373
374      airport_list = lt.newList('ARRAY_LIST')
375      lt.addLast(airport_list, airport_info)
376
377      search = prim.initSearch(graph)
378      prim_structure = prim.prim(graph, search, IATA)
379      max_traveling_distance = prim.weightMST(graph, search)
380      airports_list = prim_structure['pq']['elements']
381      edge_To_map = prim_structure['edgeTo']
382      mp.put(edge_To_map, IATA, {'vertexA': None, 'vertexB': IATA, 'weight': 0})
383
384      major_leaf = IATA
385      num_possible_airports = lt.size(airports_list)
386      airports_path_map = mp.newMap(num_possible_airports)
387      mp.put(airports_path_map, major_leaf, (0,0))
```

```
389      longest_path = 0
390      for airport in lt.iterator(airports_list):
391          airport_IATA = airport['key']
392
393          if airport_IATA != IATA:
394              queue = qu.newQueue()
395              current_path_key_value = mp.get(edge_To_map, airport_IATA)
396              current_path_info = me.getValue(current_path_key_value)
397              current_node = airport_IATA
398              father_node = current_path_info['vertexA']
399              route_distance = current_path_info['weight']
400
401              counter = 0
402              total_distance = 0
403              current_node_key_value = None
404              while current_node_key_value == None:
405                  total_distance += route_distance
406
407                  qu.enqueue(queue, current_node)
408                  current_path_key_value = mp.get(edge_To_map, father_node)
409                  current_path_info = me.getValue(current_path_key_value)
410                  current_node = father_node
411                  father_node = current_path_info['vertexA']
412                  route_distance = current_path_info['weight']
413                  current_node_key_value = mp.get(airports_path_map, current_node)
414                  counter += 1
```

```
416             additional_values = me.getValue(current_node_key_value)
417             additional_count = additional_values[0]
418             additional_distance = additional_values[1]
419             counter += additional_count
420             total_distance += additional_distance
421 
422             if counter > longest_path:
423                 longest_path = counter
424                 major_leaf = airport_IATA
425                 major_distance_path = total_distance
426             elif counter == longest_path:
427                 if total_distance < major_distance_path:
428                     major_leaf = airport_IATA
429                     major_distance_path = total_distance
430 
431             num_elements_queue = lt.size(queue)
432             for i in range(num_elements_queue):
433                 airport_path_IATA = qu.dequeue(queue)
434                 mp.put(airports_path_map, airport_path_IATA, (counter, total_distance))
435                 counter -= 1
```

La complejidad del requerimiento 4 está dada por la función **Requirement4()**. El proceso implementado para encontrar el árbol de expansión mínima (MST) y la rama más larga de dicho MST implica utilizar el algoritmo Prim (Eager) desde el aeropuerto de salida en un grafo no dirigido conformado por aeropuertos (vértices) y rutas (arcos). Primero, se obtiene el MST respectivo, luego se obtiene una lista de los aeropuertos que hacen parte del MST y finalmente se obtiene la tabla de Hash que contiene a los aeropuertos como llave y como valor la información referente a sus padres y el peso de los arcos que los unen (distancia), todo ello se implementa entre las líneas 377 a 381. Posteriormente, se procede a buscar la longitud de cada rama del MST siguiendo la línea de sucesión dada por la tabla de Hash y utilizando la lista de aeropuertos que pertenecen al MST, todo ello se implementa entre las líneas 390 a 435. El proceso de búsqueda de la rama más larga se realiza paralelamente a la búsqueda de la longitud de cada rama. Así mismo, antes de ejecutar el algoritmo de Dijkstra es necesario consultar las ciudades asociadas a un nombre homónimo. Ello se logra agrupando todas las ciudades homónimas en una lista que es el valor asociado a la llave del nombre de las ciudades en una tabla de Hash. La complejidad asociada al algoritmo Prim (Eager) con respecto al número de rutas es Prim **O(E)**, dado que es necesario consultar la mayoría de los arcos del grafo no dirigido. También, el proceso de búsqueda de la rama más larga es **O(1)**, dado esta solo depende del número de aeropuertos. Además, la complejidad de la consulta de ciudades homónimas es **O(1)**¸ dado que solo se realiza una consulta en una tabla de Hash. Finalmente, la complejidad del requerimiento 4 respecto al número de rutas es **O(E)**¸ dado que el proceso más complejo tiene dicha complejidad asociada.

Requerimiento 5

```
463  def Requirement5(catalog, IATA):
464      airports_map = catalog['airports_map']
465      compl_graph = catalog['compl_graph']
466      graph = catalog['graph']
467      digraph = catalog['digraph']
468
469      digraph_airport_indegre = gp.indegree(digraph, IATA)
470      digraph_airport_outdegree = gp.outdegree(digraph, IATA)
471      resulting_num_routes_digraph = digraph_airport_indegre + digraph_airport_outdegree
472
473      resulting_num_routes_graph = gp.degree(graph, IATA)
474
475      effected_airports_IATA_list = gp.adjacents(compl_graph, IATA)
476      possible_affected_airports = lt.size(effected_airports_IATA_list)
477      airports_list = lt.newList('ARRAY_LIST')
478      effected_airports_map = mp.newMap(possible_affected_airports)
479      for airport_IATA in lt.iterator(effected_airports_IATA_list):
480          airport_key_value = mp.get(airports_map, airport_IATA)
481          airport_info = me.getValue(airport_key_value)
482          if mp.get(effected_airports_map, airport_IATA) == None:
483              lt.addLast(airports_list, airport_info)
484              mp.put(effected_airports_map, airport_IATA, 0)
485
486      num_affected_airports = lt.size(airports_list)
```

La complejidad del requerimiento 5 está dada por la función **Requirement5()**. El proceso implementado para encontrar el número de rutas (arcos) afectadas por el cierre de un aeropuerto (vértice) en un grafo no dirigido implica encontrar el grado del aeropuerto cerrado y el número de aeropuertos adyacentes a dicho aeropuerto. Ello se logró creando un grafo no dirigido con todas las rutas registradas en **routes-utf8-large.csv** con el fin de facilitar la búsqueda de los aeropuertos adyacentes al aeropuerto cerrado. La complejidad del requerimiento 5 respecto al número de rutas es **O(E)**¸ dado que el número de rutas a analizar incrementa de manera lineal el tiempo de procesamiento del conteo de rutas afectadas.