Camilo Salinas |

Sebastián Lemus Cadena | 201716495

**Estructuras de Datos - Proyecto 2: Estimación de Complejidades**

**Introducción:**

En el presente documento se detallarán brevemente las complejidades de los algoritmos a utilizar en el Proyecto 2. Primero se explicarán las complejidades temporales de las estructuras de datos implementadas y, luego se estimará las complejidades que tendrá cada requerimiento del trabajo.

**Complejidades de Estructuras de Datos:**

A continuación, se muestra una descripción somera de la estructura del grafo y de las complejidades en las operaciones más importantes.

**Estructura del Grafo:**

Para la construcción del grafo se formaron dos clases “DiGraph” y “Vertex”, con tres parámetros genéricos: “K” (llave de cada vértice), “V” (valor de cada vértice), “W” (peso de cada arista entre vértices). Por cada clase se tiene una tabla de hash para organizar la información como se representa a continuación:

|  |  |
| --- | --- |
| **DiGraph** | **Vertex - i** |
|  |  |
|  |  |
| … | … |
|  |  |

Para la clase principal DiGraph se organizan todos los objetos Vertex existentes de acuerdo a su llave o identificador y, con respecto al punto de vista del “i-ésimo” objeto Vertex, se tiene otra tabla de hash donde se organizan todos sus vértices adyacentes de acuerdo a su llave; aquí el valor apuntado corresponde al peso que me “cuesta” en ir desde el vértice con la llave hasta el vértice con llave .

**Complejidades de las Operaciones del Grafo:**

Sea *v* el número de vértices totales y *n la* capacidad de la tabla de hash en DiGraph, y sea *a* y *m* el número de vértices adyacentes y la capacidad (respectivamente) de la tabla en Vertex, se deducen las siguientes complejidades bajo la suposición de uniformidad:

* **Añadir vértice:** Para añadir un vértice basta con llamar el método de inserción de la tabla de hash en DiGraph (“put”), por tanto, la complejidad para esta operación es O(v/n) si las llaves se distribuyen de manera uniforme en la tabla.
* **Añadir arista:** Se debe acceder a ambas tablas de hash con tal de añadir un arco entre vértices; primero se busca el vértice de salida (inicial) en DiGraph, y luego se realiza la inserción en la tabla de hash del vértice respectivo. Se suman las complejidades de ambas tablas en sus respectivas operaciones obteniendo:. En el peor de los casos el grado del vértice en cuestión es (está conectado con todos), por lo que , en cuyo caso se sobrescribiría el valor del peso de una arista.
* **Conseguir la información de un vértice:** Este método únicamente consiste en buscar en la tabla de hash de DiGraph y acceder a un atributo del vértice, por tanto, su complejidad es de O(v/m).
* **Conseguir la información de una arista:** La complejidad es la misma que la de añadir arista: .

**Complejidades de los Requerimientos Funcionales:**

**Cargar Información:**

**Requerimiento 1: Vértice más Congestionado en Chicago.**

Para cumplir el requerimiento se debe calcular los valores “outdegree” e “indegree” de cada vértice, mostrando el vértice con la mayor suma de estos dos resultados. Para obtener el “outdegree” de un vértice, simplemente se consulta el tamaño de su tabla de hash (operación de tiempo constante), pero para obtener el “indegree” se debe recorrer todos los vértices y contar cuántos de ellos apuntan al nodo en cuestión. Para iterar en una tabla de hash se tiene el método “keys()” que devuelve un objeto Iterable y a partir de éste se consulta la información de cada vértice. Siguiendo el esquema presentado en la sección de complejidades del grafo, la operación tendría una complejidad de . Una alternativa para mejorar la eficiencia en tiempo (sacrificando espacio en memoria), sería añadir una estructura dentro de la clase Vertex, que liste todos los elementos que apunten al nodo respectivo; de esta forma obtener el “outdegree” sería de complejidad constante, y la complejidad resultante sería aproximadamente lineal.

**Requerimiento 2: Componentes Fuertemente Conectadas.**

Para cumplir el requerimiento se tiene una clase llamada “StrongComponent” que tiene como atributo el color y una lista “linkeada” de los vértices que corresponden a la componente; el método que cumple este requerimiento retornará una lista de StrongComponent. Para calcular las componentes se recurre al algoritmo de Kosaraju, donde se realiza un recorrido DFS (búsqueda por profundidad) del grafo invertido en post-orden, y luego se vuelve a ejecutar el DFS sobre el grafo original teniendo en cuenta el post-orden hallado e invertido. Se sabe que para un número de vértices *v* y un número de aristas *e*, la cantidad de tiempo y espacio usado por el algoritmo es proporcional a ; por tanto la complejidad del requerimiento debe ser aproximadamente .

**Requerimiento 3: Mapa de Vértices y Componentes Conectadas.**

La principal operación de este requerimiento es la de calcular el porcentaje de servicios que salen y llegan a cada vértice. En el modelo del mundo se tiene una clase llamada AdjacentVertices, el cual corresponde al valor (parámetro V) del grafo, y se encarga de agrupar los servicios por ubicaciones cercanas a un punto de referencia. Adicionalmente se tiene una clase ArcServices que representa los pesos de cada arco entre vértices, y contiene una lista de servicios que corresponden a aquellos que tengan un punto inicial y final entre los vértices evaluados. Para calcular los porcentajes en un vértice, se consulta el tamaño de su lista (servicios totales en el servicio) y se realiza el porcentaje con el tamaño de la lista de los arcos entrantes y salientes. Para lograr esto, se debe consultar la información de los arcos por cada vértice. Siguiendo el esquema descrito en la sección de complejidad para el grafo, si se tienen “*v”* vértices totales, y un número medio de arcos por vértices “*a”,* la complejidad sería . En términos del número total de arcos la complejidad sería de .

**Requerimiento 4: Camino de Costo Mínimo para Servicios Aleatorios.**

Primero se debe buscar los vértices más cercanos a la latitud y longitud de los puntos entregados para evaluar. Para esto se debe encontrar el mínimo de distancia para todos los vértices con los dos puntos en cuestión, recorriendo todos los vértices. La complejidad de esta operación para *v* vértices es lineal .

Posterior a buscar los vértices, se utilizará el Algoritmo de Dijkstra para buscar el camino con menor distancia (atributo de ArcServices que se accede en tiempo constante) desde el vértice fuente entregado; la ejecución del algoritmo terminará cuando se encuentre el vértice final mediante una verificación. La complejidad de este algoritmo para *v* vértices y *e* aristas es implementado en una cola de prioridad orientada a mayor.

**Requerimiento 5: Caminos de Mínima y Máxima Duración para Servicios Aleatorios.**

Para encontrar los vértices iniciales y finales a partir de puntos aleatorios en el archivo de “StreetLines.csv”, se realiza el mismo método que el Requerimiento 4 con una complejidad . Para escoger los puntos aleatorios se utiliza la función *“random()”* de la clase Math de Java, tomando en cuenta la cantidad de puntos que existen en el archivo.

Luego de encontrar los vértices inicial y final, se utiliza el algoritmo de Dijkstra para encontrar el camino con menor costo temporal (atributo que se accede en tiempo constante); la complejidad en este caso es de . Por otro lado, para encontrar el camino de costo mayor hay dos situaciones que hay que tener presente:

* Si el grafo tiene ciclos, el problema es “NP-Hard” (voraz) y la única solución que existe en este momento es utilizar fuerza bruta, evaluando todos los posibles caminos desde el nodo fuente, con una complejidad de .
* Si el grafo es acíclico y dirigido (DAG), el problema se reduce a encontrar el orden topológico para todos los vértices e identificar el camino más largo desde el vértice en cuestión. Ya que el orden topológico es esencialmente el uso de DFS (Búsqueda a Profundidad), la complejidad del algoritmo es .

Si el grafo tiene ciclos la complejidad total del requerimiento es , de lo contrario es . Para este problema, por tanto, es recomendable obviar o remover todos los ciclos que tiene el grafo de servicios.

**Requerimiento 6: Caminos con Peaje Ordenados Temporalmente y por Costos para Servicios Aleatorios.**

Primero para encontrar los vértices con puntos aleatorios se sigue la misma metodología que en los requerimientos 4 y 5, obteniendo una complejidad para *v* vértices. Luego de esto, para encontrar todos los caminos entre los dos puntos se aplica DFS desde el nodo inicial y se verifica si el camino tiene peaje o no; si tiene peaje, se crea una nueva instancia de ArcServices (un arco) que represente el camino total con la sumatoria de los costos de cada intersección; si no tiene peaje el camino se ignora y se continúa por otro camino. Las diferentes instancias de arcos se irán añadiendo a una cola de prioridad orientada a mayor, la cual se organizará con el algoritmo de heap sort. Para *v* vértices y *e* arcos en el grafo, la complejidad del DFS es y la complejidad de heapsort es de ; esta última es la complejidad total para el requerimiento

**Nota:** Para más información sobre las clases y estructuras de datos del proyecto refiérase a los diagramas UML dentro del directorio “UML” justo a dentro de la carpeta principal.

**Bibliografía:**

[1] Sedgewick, R., & Wayne, K. (2011). *Algorithms Fourth Edition* (Addison-Wesley). Boston: Addison-Wesley.