Melqui Jair Aguirre García A00346739
Alejandra González Vélez A00346154
Angie Valentina Córdoba Pinzón A00347829

IDENTIFICACIÓN DEL PROBLEMA

En los últimos años son cada vez más las personas que viajan constantemente en un avión, esto por el precio cada vez más bajo de los viajes, la comodidad y rapidez que estos tienen. (AERONÁUTICA CIVIL DE COLOMBIA, n.d.) En septiembre de 2018, 1.953.315 pasajeros se movilizaron a nivel nacional, haciendo uso de distintas aerolíneas. Avianca es la Compañía que programa más vuelos, seguido de LAN Colombia, Copa Airlines, Satena, Easyfly y Viva Colombia. De todos estos pasajeros, muchos de ellos no programan su viaje de la manera más eficiente en cuanto a tiempo o dinero por desconocimiento de las rutas que cada aerolínea ofrece. Para lo anterior, se requiere el modelamiento de una herramienta que permita al usuario conocer las opciones para la ruta deseada, teniendo a su disposición la mejor ruta en cuanto a tiempo y la mejor ruta en cuanto a costo. Pudiendo así planificar su viaje en las mejores condiciones, conociendo las opciones que tiene disponibles

RECOPILACIÓN DE INFORMACIÓN

Definiciones

-Grafo: Los grafos aparecen como una extensión del concepto de árbol, ya que en este nuevo tipo de estructuras cada elemento puede tener, además de más de un sucesor, varios elementos predecesores. Esta propiedad hace a los grafos las estructuras más adecuadas para representar situaciones donde la relación entre los elementos es completamente arbitraria, como pueden ser mapas de carreteras, sistemas de telecomunicaciones, circuitos impresos o redes de ordenadores.

(Fuente: http://www6.uniovi.es/usr/cesar/Uned/EDA/Apuntes/TAD_apUM_07.pdf)

-Algoritmo Dijkstra: El Algoritmo de Dijkstra, también denominado Algoritmo de caminos mínimos, es un modelo que se clasifica dentro de los algoritmos de búsqueda. Su objetivo, es determinar la ruta más corta, desde el nodo origen, hasta cualquier nodo de la red. Su metodología se basa en iteraciones, de manera tal que, en la práctica, su desarrollo se dificulta a medida que el tamaño de la red aumenta, dejándolo en clara desventaja, frente a métodos de optimización basados en programación matemática.

(Fuente: https://www.ingenieriaindustrialonline.com/herramientas-para-el-ingeniero-industrial/investigacion-de-operaciones/algoritmo-de-dijkstra/)

-Algoritmo BFS:

El algoritmo BFS (Breadth-First Search) es una forma de encontrar todos los vértices alcanzables de un grafo partiendo de un vértice origen dado. Como en el algoritmo de búsqueda en profundidad, BFS recorre una componente conexa de un grafo y define un árbol de expansión.

(Fuente:
http://www.dma.fi.upm.es/personal/gregorio/grafos/web/iagraph/busqueda.html)

SOLUCIONES CREATIVAS

- **Alternativa1:** Grafo dirigido.
  Un grafo dirigido es un grafo en el cual las aristas están dirigidas (y se acostumbra a llamarles arcos). Es decir, la relación entre dos vértices adyacentes no necesariamente es simétrica (por ejemplo, "hijo de"). La metáfora básica para este tipo de relaciones no simétricas es la de dominancia: si hay un arco que va de A a B (relación que se denota a veces como A→B) entonces se dice que a domina a B.
  (Fuente: http://www.matetam.com/glosario/definicion/grafos-dirigidos-digrafos)
- **Alternativa2:** Grafo no dirigido.
  Cuando no importa la dirección de las aristas el grafo se denomina no dirigido. En un grafo no dirigido hay aristas no dirigidas. Se dibuja igual sin poner la flecha en la arista. En vez de denotar la arista con (a, b) utilizamos conjuntos y denotamos la arista con {a, b}. En general cuando no se especifica si un grafo G es o no dirigido se supone que es no dirigido.
  (Fuente:
  https://eva.fing.edu.uy/pluginfile.php/180058/mod_resource/content/1/mdl2-teorico2.pdf)
- **Alternativa3:** Árbol binario de búsqueda.
  Son árboles binarios en los que se cumple que, para cada nodo, el valor de la clave de la raíz del subárbol izquierdo es menor que el valor de la clave del nodo y que el valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo. Las operaciones para un ABB son similares que las realizadas en un árbol binario general.
  (Fuente: https://www.ecured.cu/%C3%81rbol_binario_de_b%C3%BAsqueda)
- **Alternativa4:** Lista enlazada
  La lista enlazada es un TDA que nos permite almacenar datos de una forma organizada, al igual que los vectores, pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener.

En una lista enlazada, cada elemento apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es null. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista suelen recibir también el nombre de nodos de la lista.
(Fuente: http://www.calcifer.org/documentos/librognome/glib-lists-queues.html)

TRANSICION DE LAS IDEAS A LOS DISEÑOS PRELIMINARES

Se descartó el uso de la alternativa 4, ya que según el planteamiento del problema, no se requiere una estructura organizada tal como una lista enlazada, pues las necesidades son distintas a los recursos que esta estructura puede aportar. Así mismo sucede con la alternativa 3, pues de acuerdo con el problema no se requiere una estructura organizada tal como un árbol binario de búsqueda.

La revisión cuidadosa de las alternativas conduce a:

Alternativa1: Grafo dirigido.

- Se puede determinar el sentido de la relación entre vértices
- Dependiendo de la representación, se requiere de un almacenamiento de |v|*|v|, es decir $O(n^2)$
- Dependiendo de la implementación, se puede determinar en un tiempo fijo y constante si un enlace(arco) pertenece o no al grafo.

Alternativa2: Grafo no dirigido.

- No se puede determinar el sentido de la relación entre vértices, pues las aristas son relaciones simétricas y no apuntan en ningún sentido.
- Dependiendo de la representación, se requiere de un almacenamiento de |v|*|v|, es decir $O(n^2)$
- Dependiendo de la implementación, se puede determinar en un tiempo fijo y constante si un enlace(arco) pertenece o no al grafo.

EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN

A continuación, se evalúan cada una de las alternativas, teniendo en cuenta que la de mayor puntaje es la más eficiente.

Criterios:

Criterio A: La alternativa es la adecuada para solucionar alguno de los problemas que plantea la situación.

- [2] Ayuda en la solución

- [1] No ayuda en la solución

Criterio B: La alternativa es eficiente a un nivel:

- [4] Constante.

- [3] Mayor a constante.

- [2] Logarítmica

- [1] Lineal o mayor

Criterio C: La alternativa permite combinarse con otras para generar una mejor solución

- [2] Mejora con otra alternativa

- [1] Es suficiente por ella misma

	Criterio A	Criterio B	Criterio C	Total
Alternativa 1	2	1	2	5
Alternativa2	1	1	2	4

Selección:

De acuerdo con la evaluación anterior, la alternativa que más se ajusta al problema es la alternativa 1, ya que obtuvo la mejor puntuación y es la más adecuada para la solución del problema.

PREPARACIÓN DE LOS INFORMES Y ESPECIFICACIONES

Especificación del problema:

Problema 1: Entregar al usuario la ruta más corta en cuanto a distancia para el destino al cual desea llegar.

Entrada: Destino de salida, destino de llegada.

A ^ B, donde A ^ B ε W (Ciudades)

Salida: Ruta más corta que se debe recorrer para llegar al destino

f { A_1, A_2 , A_3 , …, A_n} donde A_n Son los nodos que marcan la ruta.

Problema 2: Entregar al usuario la mejor ruta en cuanto a costo para el destino al cual desea llegar

Entrada: Destino de salida, destino de llegada.

A ^ B, donde A ^ B ε W (Ciudades)

Salida: Ruta más corta que se debe recorrer para llegar al destino

f { A_1, A_2 , A_3 , …, A_n} donde f es el conjunto de los nodos que marcan la ruta

Problema 3: Entregar al usuario las rutas disponibles por aerolínea con detalles de costo y distancia

Entrada: Aerolínea de la cual se desea conocer la información

A, donde A ε W (Aerolíneas)

Salida: Conjunto de nodos adyacentes con su nombre y distancia

f { A_1, A_2 , A_3 , …, A_n} donde f es el conjunto de nodos que contienen la información necesitada.

Pseudocódigo del método Dijkstra

DIJKSTRA (Grafo G, nodo_fuente s)

```
1  para u ∈ V[G] hacer
2  distancia[u] = INFINITO
3  padre[u] = NULL
4  visto[u] = false
5  distancia[s] = 0
6  adicionar (cola, (s, distancia[s]))
   //mientras que cola no es vacía hacer
7  u = extraer mínimo(cola)
8  visto[u] = true
   //para todos v ∈ adyacencia[u] hacer
9  si distancia[v] > distancia[u] + peso (u, v) hacer
10 distancia[v] = distancia[u] + peso (u, v)
11 padre[v] = u
12 adicionar(cola,(v, distancia[v]))
```

Pseudocódigo del método BFS

BFS(grafo G, nodo_fuente s) {

```
// recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
// distancia INFINITA y padre de cada nodo NULL
1    for u ∈ V[G] do
2    {
3      estado[u] = NO_VISITADO;
4      distancia[u] = INFINITO; /* distancia infinita si el nodo no es alcanzable */
5      padre[u] = NULL;
6    }
7    estado[s] = VISITADO;
8    distancia[s] = 0;
9    padre[s] = NULL;
10   CrearCola(Q); /* nos aseguramos que la cola está vacía */
11   Encolar(Q, s);
12   while !vacia(Q) do
13   {
// extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
14     u = extraer(Q);
15     for  v ∈ adyacencia[u]  do
16     {
17       if estado[v] == NO_VISITADO then
18       {
19          estado[v] = VISITADO;
20          distancia[v] = distancia[u] + 1;
21          padre[v] = u;
22          Encolar(Q, v);
23       }
24     }
25   }
26 }
```

IMPLEMENTACIÓN DEL DISEÑO

Implementación en el lenguaje de programación

Lista de tareas a implementar:

A. Agregar nodos al grafo simulando las ciudades a las cuales llega cada aerolínea
B. Agregar aristas al grafo para conectar cada ciudad a los destinos que esta tiene disponibles
C. Encontrar todas las rutas de una aerolínea especificada
D. Encontrar las mejores rutas en cuanto a distancia y dinero para una ciudad de salida y una ciudad de llegada

Cada uno de los anteriores algoritmos estarán implementados en las dos representaciones del grafo (matriz de adyacencias y lista de adyacencias). A continuación, se presentarán los métodos para cada una de las implementaciones

Matriz de adyacencias:

A.

Nombre	addNodeM
Descripción	Agrega un nuevo nodo al grafo
Entrada	Identificador del nodo
Retorno	Sin retorno

Código:

```
public void addNodeM(T key) {
        NodeM<T> node = searchNodeM(key);
        if(node == null) {
            node = new NodeM<T>(key);
            nodes.add(node);
            node.setPos(nodes.indexOf(node));
            limit++;
        }
}
```

B.

Nombre	addEdge
Descripción	Agrega una nueva arista al grafo
Entrada	Identificador de los nodos que conecta y el peso de la arista
Retorno	Sin retorno

Código:

```
public void addEdge(T key, T key2, int dis) {

        NodeM<T> v1 = searchNodeM(key);
        NodeM<T> v2 = searchNodeM(key2);
        if(v1 == null) {
            addNodeM(key);
            v1 = searchNodeM(key);
        }
        if(v2 == null) {
            addNodeM(key2);
            v2 = searchNodeM(key2);
        }
        int pos1 = nodes.indexOf(v1);
        int pos2 = nodes.indexOf(v2);

        matrix[pos1][pos2] = dis;
        v1.getAdjacents().add(v2);
    }
```

C.

Nombre	BFS
Descripción	Devuelve los elementos del grafo recorridos por amplitud
Entrada	Nodo desde el cual se empezará a hacer el recorrido
Retorno	Conjunto de nodos pertenecientes al grafo

Código:

```
public Tree<T> BFS(T origin) throws Exception{
		Tree<T> bfs = new Tree<T>();
		NodeM<T> n = searchNodeM(origin);
		boolean[] visit = new boolean[nodes.size()] ;
		visit[n.getPos()] = true;
		Queque<Integer> queue = new StructureStackQueue<Integer>();
		bfs.add(nodes.get(n.getPos()).getElem(),null);
		queue.enqueque(n.getPos());
		while(!queue.isEmptyQ()) {
			int u = queue.dequeque();
			for(int i = 0; i< nodes.size(); i++) {
				if( !visit[i]) {
					queue.enqueque(i);
		bfs.add(nodes.get(i).getElem(),nodes.get(u).getElem());
					visit[i] = true;
		}	}
		}
		return bfs;
	}
```

D.

Nombre	Dijkstra
Descripción	Devuelve la mejor ruta desde un nodo a otro
Entrada	Nodo de salida y nodo de llegada
Retorno	Mejor ruta encontrada con la distancia o costo total dependiendo del criterio pedido por el usuario

Código:

```
public ArrayList<T> DFS(T origin) throws Exception {
		boolean[] visit = new boolean[nodes.size()];
		ArrayList<T> path = new ArrayList<T>();
		for(int i=0; i<nodes.size(); i++) {
			visit[i]=false; }
		NodeM<T> n = searchNodeM(origin);
		Queque<Integer> queque = new StructureStackQueue<>();
		visit[n.getPos()]=true;
		queque.enqueque(n.getPos());
		path.add(n.getElem());
		while(!queque.isEmptyQ()) {
			int u = queque.dequeque();
			for(int i=0; i<nodes.size(); i++) {
				if(!visit[i]) {
					queque.enqueque(i);
					visit[i]=true;
					path.addAll(DFS(nodes.get(i).getElem()));
			}
		}
	}
		return path;
}
```

Lista de adyacencias:

A.

Nombre	addNode
Descripción	Agrega un nuevo nodo al grafo
Entrada	Identificador del nodo
Retorno	Sin retorno

Código:

```
public void addNode(T node) throws Exception {
        NodeL<T> n = new NodeL<T>(node);
        if(nodes.get(node) != null) {
            throw new Exception("Nodo ya existente");
        }
        if(totalNodes == maxNodes) {
            throw new Exception("Número máximo de nodos alcanzados");
        }
        nodes.put(node, n);
        totalNodes++;
    }
```

B.

Nombre	addEdge
Descripción	Agrega una nueva arista al grafo
Entrada	Identificador de los nodos que conecta y el peso de la arista
Retorno	Sin retorno

Código:

```
public void addEdge(T node1, T node2, double distance) throws Exception {
		NodeL<T> n1 = nodes.get(node1);
		NodeL<T> n2 = nodes.get(node2);
		if(n1 == null || n2 == null) {
			throw new Exception("Uno de los nodos no existe");
		}
		n1.addAdjacents(n2);
		n1.addDistance(n2, distance);
	}
```

C.

Nombre	BFS
Descripción	Realiza un recorrido por amplitud sobre el grafo
Entrada	Sin entrada
Retorno	Suma de las distancias

Código:

```
public int BFS() {
		int bfs;
		Set<T> keys = nodes.keySet();
		try {
			bfs = BFS(keys.iterator().next());
		}catch(Exception e) {
			bfs = 0;
		}
		return bfs;
	}
```

Método auxiliar BFS:

```
public int BFS(T Norigin) throws Exception {
		for(T na : nodes.keySet()) {
			nodes.get(na).setVisit(false);
		}
		int find = 0;
		NodeL<T> act = nodes.get(Norigin);
		if(act == null) {
			throw new Exception("Nodo existe el nodo");
		}
		Queque<NodeL<T>> queque = new StructureStackQueue<NodeL<T>>();
		queque.enqueque(act);
		find++;
		act.setVisit(true);
		while(!queque.isEmptyQ()) {
			NodeL<T> n = queque.dequeque();
			ArrayList<INodeL<T>> adjacents = n.getAdjacents();
			for(int i = 0; i < adjacents.size(); i++) {
				if(!adjacents.get(i).isVisit()) {
					adjacents.get(i).setVisit(true);
					find++;
					adjacents.get(i).setParent(n);
					queque.enqueque((NodeL<T>)adjacents.get(i));
				}
			}
			n.setVisit(true);
		}
```

D.

Nombre	Dijkstra
Descripción	Devuelve la mejor ruta desde un nodo a otro
Entrada	Nodo de salida y nodo de llegada
Retorno	Mejor ruta encontrada con la distancia o costo total dependiendo del criterio pedido por el usuario

Código:

```
public WeightList<T> Dijkstra(T node1, T node2) throws Exception {
        HashMap<NodeL<T>, Double> l = new HashMap<>();
        HashMap<NodeL<T>, Double> s = new HashMap<>();
        for(T na : nodes.keySet()) {
            l.put(nodes.get(na), Double.MAX_VALUE);
        }
        NodeL<T>  n1 = nodes.get(node1);
        NodeL<T>  n2 = nodes.get(node2);
        if(n1 == null || n2 == null) {
            throw new Exception("Uno de los nodos no existe");
        }
        l.put(n1, 0.0);
        PriorityQueue<NodeL<T>> heap = new PriorityQueue<>();
        heap.add(n1);
        boolean find = false;
        while(s.get(n2) == null && !heap.isEmpty()) {
            NodeL<T> actual = heap.poll();
        HashMap<INodeL<T>, Double> actualDistances = actual.getDistances();
            double lActual = l.get(actual);
            s.put(actual, 0.0);
```

```
            if(actual == n2){
                find = true;
            }
            ArrayList<INodeL<T>> adjacents = actual.getAdjacents();
            for(int i = 0; i < adjacents.size(); i++) {
                NodeL<T> actualAdjacent = (NodeL<T>) adjacents.get(i);
            Double newActualDistances = actualDistances.get(actualAdjacent);
                if(newActualDistances + lActual < l.get(actualAdjacent)) {
                    actualAdjacent.setParent(actual);
            actualAdjacent.setDistancePrevPath(newActualDistances + lActual);
                    l.put(actualAdjacent, newActualDistances + lActual);
                    heap.add(actualAdjacent);
                }
            }
        }
        if(!find)
            throw new Exception("Imposible llegar del nodo 1 al nodo 2");
        LinkedList<T> path = new LinkedList<T>();
        NodeL<T> actual = n2;
        while(actual != n1) {
            T elem = actual.getElem();
            path.addFirst(elem);
            actual = (NodeL<T>) actual.getParent();
        }
        path.addFirst(n1.getElem());

        return new WeightList<>(path, l.get(n2));    }
```