Método de Ingeniería

1. **Identificación del problema:**

   En este punto identificamos el problema que se debe resolver y las necesidades que se deben satisfacer a lo largo del proyecto:

   El programa debe permitir conocer cuál es el mínimo tiempo de recorrido aéreo entre algunas de las 50 ciudades más turísticas de Europa. Para ello se deben tener en cuenta algunas cosas:

   - Se debe obtener el tiempo de recorrido entre el aeropuerto de una ciudad y los aeropuertos de las ciudades a los cuales se puede dirigir.
   - La solución del problema debe cargar las ciudades desde un archivo .CSV establecido previamente.
   - La solución debe generar las conexiones entre los aeropuertos de cada ciudad.
   - Se debe calcular el menor tiempo de recorrido entre las ciudades seleccionadas por el usuario.
   - La solución debe implementar 2 versiones de grafos y ser completamente funcional en ambas.

2. **Recopilación de Información:**

   Con el objetivo de tener claras las opciones posibles para la implementación de grafos, se hizo una búsqueda a fondo de los tipos de algoritmos de grafos se podrían usar.

   BFS: Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Given a graph G= (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s.
   To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u,v) \in E$ and vertex u is black, then vertex is either gray or black; that is, all vertices adjacent to

black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

DFS: The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Dijkstra: La idea principal del algoritmo es muy sencilla: Se irá construyendo un grafo F, inicialmente formado únicamente por el nodo origen. En cada paso se miran todos los nodos a los que podamos llegar directamente desde F, es decir aquellos que compartan arista con un nodo de F, y añadiremos el nodo cuya distancia al origen sea mínima (con la arista que minimice dicha distancia). Pararemos el algoritmo en el momento en que el vértice a incluir a F sea el nodo destino. En cada iteración, al incluir un nodo (v) a F podemos estar seguros de que lo estamos tomando utilizando el menor camino del origen al nodo v. De existir otro camino, este debería pasar por alguno de los otros nodos a los que podemos llegar directamente desde F, los cuales están a mayor o igual distancia del origen que el nodo v, por construcción. Dado que los pesos son no negativos, cualquier camino que pase por alguno de estos nodos para llegar a v no puede tener una distancia menor a la que estábamos considerando inicialmente. De modo que al final del algoritmo habremos hallado la distancia mínima entre el origen y el destino.

Floyd-Warshall: The Floyd-Warshall Algorithm is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the

paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.

Kruskal: El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

Pasos del algoritmo:

- Se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado
- Se crea un conjunto C que contenga a todas las aristas del grafo
- Mientras C es no vacío:
  - Eliminar una arista de peso mínimo de C
  - Si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol
  - En caso contrario, se desecha la arista
- Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

Prim: Prim's Algorithm is an algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Fuentes:

- T.H Corner. Breadth-first search. En Introduction to Algorithms. Capítulo 22.2, páginas 594-601. Tercera edición, 2009.
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://en.wikipedia.org/wiki/Breadth-first_search
- T.H Corner. Depth-first search. En Introduction to Algorithms. Capítulo 22.3, páginas 603-610. Tercera edición, 2009.
- https://www.encora.com/es/blog/dfs-vs-bfs
- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- T.H Corner. The Floyd-Warshall algorithm. En Introduction to Algorithms. Capítulo 25.2, páginas 693-699. Tercera edición, 2009.
- https://aprende.olimpiada-informatica.org/algoritmia-dijkstra-bellman-ford-floyd-warshall
- https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra
- T.H Corner. Dijkstra's algorithm. En Introduction to Algorithms. Capítulo 24.3, páginas 658-662. Tercera edición, 2009.
- https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal
- T.H Corner. Kruskal's algorithm. En Introduction to Algorithms. Capítulo 23.2, páginas 631-633. Tercera edición, 2009.
- https://nodo.ugto.mx/wp-content/uploads/2018/08/Kruskal.pdf
- T.H Corner. Prim's algorithm. En Introduction to Algorithms. Capítulo 23.2, páginas 634-636. Tercera edición, 2009.
- https://en.wikipedia.org/wiki/Prim%27s_algorithm
- https://www.scaler.com/topics/data-structures/prims-algorithm/
- https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

3. **Búsqueda de Soluciones Creativas:**

   Para este punto se propuso una lluvia de ideas en la que nos planteáramos las maneras en la que se podía resolver el problema. Estos fueron los resultados de la lluvia de ideas:

   - Implementar un algoritmo de búsqueda BFS en grafos.
   - Implementar un algoritmo de búsqueda DFS.
   - Implementar en la solución un algoritmo de Dijkstra.
   - Implementar un algoritmo de Floyd-Warshall.
   - Implementar un algoritmo de Prim
   - Implementar un algoritmo de Kruskal.

4. **Transición de la formulación de ideas a los diseños preliminares:**

   En este paso evaluaremos las ideas que podrían dar solución al problema las cuales fueron planteadas en el anterior punto. Pero primero se descartarán las ideas que no son factibles. Descartaremos usar DFS debido a que esta alternativa no tiene en cuenta el peso de las aristas, por lo tanto, no se podría calcular el tiempo de cada recorrido.

   La revisión de las otras alternativas nos conduce a los siguiente:

   - Alternativa 1. BFS:
     - Esta alternativa supone una fácil implementación y manejo en la estructura.
   - Alternativa 2. Dijkstra:
     - Esta alternativa supone un difícil manejo para la estructura de la implementación
   - Alternativa 3. Floyd-Warshall:
     - Esta alternativa supone una mayor facilidad a la hora del manejo de la estructura.
   - Alternativa 4. Prim:
     - Esta alternativa supone una complejidad alta a la hora de la implementación.
   - Alternativa 5. Kruskal:
     - Esta alternativa supone una complejidad alta a la hora de la implementación en código.

5. **Evaluación y Selección:**

   Para la selección de la solución final tendremos en cuenta los siguientes criterios:

   - Criterio A: Suplencia de requerimientos.

     - [2] Cumple con todos los requerimientos.

     - [1] Incumple algún requerimiento.

   - Criterio B: Complejidad general de la implementación.

     - [3] Complejidad Estándar.

     - [2] Complejidad Media-Alta.

     - [1] Complejidad Alta.

   - Criterio C: Complejidad en el manejo del grafo.

     - [3] Complejidad Estándar.

     - [2] Complejidad Media-Alta.

     - [1] Complejidad Alta.

	Criterio A	Criterio B	Criterio C
Alternativa 1. BFS	Cumple con todos los requerimientos. 2	Complejidad Estándar 3	Complejidad Estándar 3
Alternativa 2. Dijkstra	Cumple con todos los requerimientos. 2	Complejidad Estándar 3	Complejidad Media-Alta. 2
Alternativa 3. Floyd-Warshall	Cumple con todos los requerimientos. 2	Complejidad Estándar 3	Complejidad Estándar 3
Alternativa 4. Prim	Cumple con todos los requerimientos. 2	Complejidad Media-Alta. 2	Complejidad Media-Alta. 2
Alternativa 5. Kruskal	Cumple con todos los requerimientos. 2	Complejidad Alta 1	Complejidad Media-Alta. 2

Selección: De acuerdo con la evaluación anterior y teniendo en cuenta que se deben implementar 2 algoritmos, debemos tomar las alternativas 1 y 3 ya que obtuvieron la mayor puntuación.

6. **Preparación de Informes y Especificaciones:**

   Especificación del Problema:

   Problema: Conocer cuál es el mínimo tiempo de recorrido aéreo entre algunas de las 50 ciudades más turísticas de Europa.

   Entrada:

   - Número de ciudades que se desean visitar
   - Nombre de las ciudades que se desean visitar

   Salidas: Tiempo mínimo de recorrido entre los aeropuertos de las ciudades

   que se desean visitar.

7. **Implementación del Diseño:**

   Implementación en Java.

   Lista de Tareas para implementar:

   a) Pedir las entradas.
   b) Generar el grafo.
   c) Calcular el tiempo mínimo de recorrido con los algoritmos seleccionados.
   d) Imprimir la salida.

   Especificación de subrutinas:

   a)

Nombre:	requestData
Descripción:	Pide los datos necesarios para el funcionamiento del problema.
Entrada:	Cantidad de ciudades a visitar y nombre de las ciudades a visitar.
Salida	

b)

Nombre:	Init
Descripción:	Genera el grafo con los vértices y aristas necesarios.
Entrada:	
Salida	

c)

Nombre:	floydWarshall
Descripción:	Utiliza el algoritmo de Floy-Warshall para hallar el camino más corto.
Entrada:	Grafo
Salida	

d)

Nombre:	searchRoad
Descripción:	Calcula el tiempo mínimo de recorrido entre las ciudades elegidas por el usuario
Entrada:	Ciudades que se desean recorrer.
Salida:	Tiempo mínimo de recorrido.

Construcción:

a)

```
public void requestData() {

    System.out.println("Enter the number of Cities to visit");
    int t = sc.nextInt();
    ArrayList<String>cities=new ArrayList<>();
    for(int i=0;i<t;i++) {
        String city= sc.next();
        cities.add(city);

    }

    mapEurope.floydWarshall(mapEurope.costMatrix);
    System.out.println(mapEurope.searchRoad(cities));
}
```

b)

```
public void init() {
    mapEurope = new GraphList<>(6);

    mapEurope.addVertex("Barcelona");
    mapEurope.addVertex("Paris");
    mapEurope.addVertex("Londres");
    mapEurope.addVertex("Zurich");
    mapEurope.addVertex("Roma");
    mapEurope.addVertex("Moscu");

    mapEurope.addEdge("Barcelona", "Londres", 5);
    mapEurope.addEdge("Londres", "Paris",3);
    mapEurope.addEdge("Barcelona", "Paris",2);
    mapEurope.addEdge("Paris", "Zurich",1);
    mapEurope.addEdge("Paris", "Roma",2);
    mapEurope.addEdge("Roma", "Zurich",2);
    mapEurope.addEdge("Moscu", "Roma",7);

}
```

c)

```
public void floydWarshall(int[][]graph) {
    int nV=graph.length;

        int i=0;
        int j=0;
        int k=0;

       for( i=0;i<matrixFloyd.length;i++) {
           for(j=0;j<matrixFloyd[0].length;j++) {
               matrixFloyd[i][j]=graph[i][j];

           }
       }

       for (k = 0; k < nV; k++) {
            for (i = 0; i < nV; i++) {
                for (j = 0; j < nV; j++) {

                    if (matrixFloyd[i][j] > (matrixFloyd[i][k] + matrixFloyd[k][j])
                        && (matrixFloyd[k][j] != Integer.MAX_VALUE
                            && matrixFloyd[i][k] != Integer.MAX_VALUE))
                        matrixFloyd[i][j] = matrixFloyd[i][k] + matrixFloyd[k][j];
                }
            }
        }
}
```

d)

```
public int searchRoad(ArrayList<T>cities) {
    int time=9999999;
    int b=0;

    for(int i=0;i<cities.size();i++) {
        int a = searchPos(cities,cities.get(i));
        int parcialtime=0;
       for(int j=0;j<cities.size();j++) {

           b = searchPos(cities,cities.get(j));
           parcialtime+=matrixFloyd[a][b];

       }

       if(parcialtime<time) {
           time=parcialtime;

       }


    }
    return time;
}
```