

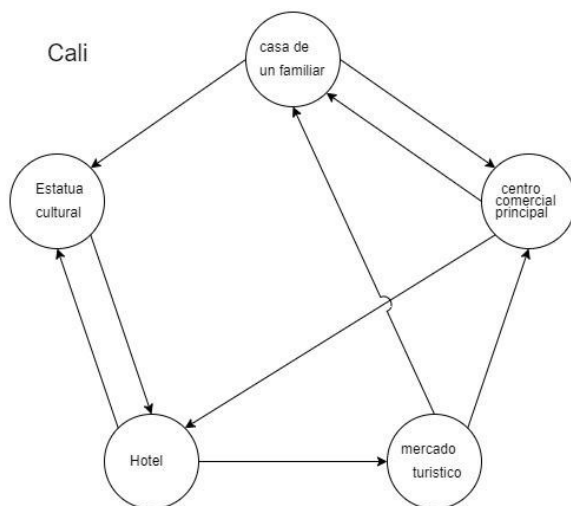
Método de la ingeniería

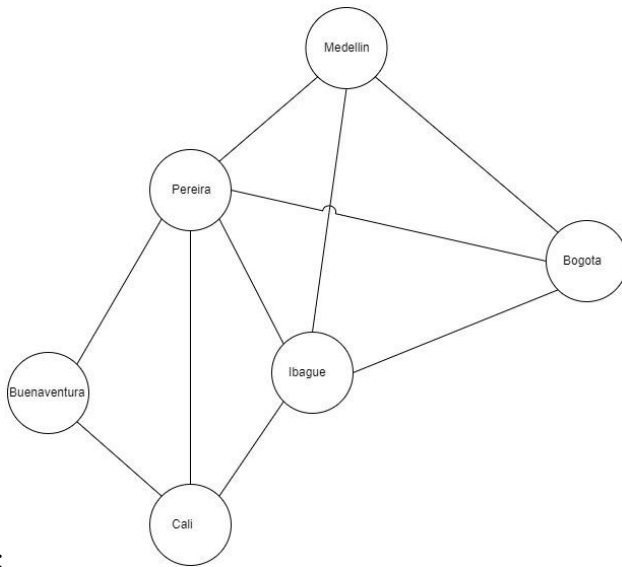
Contexto del problema:

Tú y tu familia van a salir de vacaciones a diferentes ciudades de Colombia que no hayan visitado hasta el momento. El viaje se realizará en carro y debido a esto tienen la necesidad de utilizar un mapa completo de todas las ciudades para conocer como llegar a cada una a través de la otra, además de los mapas de cada ciudad con puntos de interés marcados.

Durante una discusión acerca del paseo se encontró el problema de que todos en tu familia son terribles leyendo mapas y si intentan llegar a un supermercado terminan en Siloé, por lo que te pidieron a ti, un ingeniero de sistemas de Icesi, que crearas un programa que pudiese representar todos estos mapas y que de una forma simple indique cuales son los caminos para llegar de una ciudad a otra, o de un punto de interés al otro, preferiblemente los más cortos.

Para implementar el programa cuentas con los mapas digitales de Colombia (más específicamente, de la zona donde se encuentran las ciudades a visitar) y de cada ciudad. Los mapas tienen la información de las calles entre un punto al otro y sus longitudes en km.





Ej.:

*Los mapas utilizados no son representaciones reales de las calles/carreteras.

Fase 1 identificación del problema:

identificación de necesidades y síntomas

- El camino más corto de un punto de interés/ciudad al otro.
- Todos los puntos de interés o ciudades en el mapa.
- Cantidad de zonas inconexas (zonas a las que no se puede acceder por carro)
- El tiempo que se demora en llegar de un punto al otro teniendo en cuenta una velocidad introducida.

requerimientos funcionales :

Nombre	R0: Conseguir el camino más corto de un punto de interés/ciudad al otro
---------------	---

Resumen	Se da a conocer a través de qué camino se llega de un punto de interés/ciudad a la otra
Entrada	Ciudad de inicio, ciudad a llegar
Salida	Camino mínimo entre los dos puntos

Nombre	R1: Mostrar los puntos de interés/ciudades de un mapa
Resumen	Imprime los puntos de interés/ciudades que se encuentran en el mapa
Entrada	Mapa a revisar
Salida	puntos de interés/ciudades del mapa

Nombre	R2: Mostrar la cantidad de zonas inaccesibles por carro
Resumen	Da a conocer la cantidad de zonas que existen en el mapa a las que no son posibles de acceder por medio de un carro.
Entrada	Mapa
Salida	Cantidad de zonas inconexas.

Nombre	R3: Dar a conocer el tiempo que se demora de una ciudad a otra al conocer la velocidad
Resumen	al usuario introducir la velocidad promedio, se calcula el tiempo de llegada de una ciudad a otra.
Entrada	Velocidad promedio, ciudad de inicio, ciudad a llegar
Salida	Tiempo estimado de llegada

Nombre	R4: Ilustrar el mapa elegido
Resumen	El usuario puede elegir un mapa para mostrar en pantalla.
Entrada	Mapa a mostrar
Salida	Ilustración del mapa

Fase 2 recopilación de información:

Municipios de Colombia

Los municipios de Colombia son el segundo nivel de división administrativa en Colombia, el país tiene 1122 municipios que se han registrados en el DANE. Dependiendo de la entidad gubernamental que maneja el terreno el número de municipios cambia. Debido a que ciertas comunidades creen que deberían ser considerados municipios bajo la definición, más algunas entidades no las registran como tales. Excepto por los municipios o áreas localizadas en San Andrés todos los municipios están conectadas mediante una serie de carreteras o vías que permiten el transporte de personas, y otros bienes. Dependiendo del área que se encuentren estos municipios el estado de sus carreteras y vías pueden variar. Algunas pueden estar en excelente estado lo que resulta con menor tiempo de viaje entre ellas, otras con vías más rurales dependiendo del vehículo que las transite pueden tomar más tiempo en llegar a su objetivo.

Representación grafica

En las más recientes actualizaciones de JavaFX aún no se ha implementado ninguna manera de poder mostrar gráficamente un grafo hecho en java. Una posible alternativa que se podría utilizar sería utilizar las figuras graficas que trae JavaFX y crear métodos que permitiría crear grafos y mostrarlos de manera gráfica. Mas esta alternativa toma mucho tiempo y los resultados pueden ser poco placenteros para los ojos. Mas gracias a la comunidad de programadores que existen en internet, y sus habilidades con el lenguaje de programación, se pueden utilizar sus códigos e implementarlos en nuestra solución.

Esta técnica sería utilizar una librería externa. Java contiene librerías propias que le permiten a los usuarios crear programas utilizándolas, más existen librerías que no son propias del lenguaje de programación más han sido implementadas en él. Y al implementar estas librerías tenemos acceso a muchas funciones más. En este caso podemos utilizar una librería externa que nos permita crear y mostrar grafos de manera gráfica para el usuario de la aplicación. O dependiendo de que librería se decida utilizar mostrar le mapa de Colombia y sus municipios conectados por carreteras.

Algoritmos de búsqueda del camino más corto

El problema de encontrar el camino más corto entre dos puntos ha sido uno que ha sido explorado infinitamente por científicos y matemáticos. Y entre sus muchas soluciones hay dos que sobresalen como las mejores alternativas para encontrar el camino más corto. El Algoritmo de Dijkstra y el algoritmo Floyd-Warshall. Estos dos algoritmos permiten solucionar el problema, pero cada uno tiene

diferencias clave que lo distinguen entre sí y permiten otros usos adicionales gracias a ellos. Y además tiene la ventaja de poder ser implementados de varias maneras dependiendo de la implementación del grafo que se decida utilizar para solucionar el problema.

Fase 3 búsqueda de soluciones creativas:

Alternativa 1 Crear un mapa con las rutas más cortas

Esta alternativa consiste en crear un mapa con las rutas más cortas entre municipios y sus tiempos aproximados de viajes entre ellos, así facilitando el viaje y resolviendo el problema planteado anteriormente. Además, podemos plantear un mapa por departamento para así facilitar el manejo y uso de estos mapas al público en general, y podría ser muy complejo para alguien que no esté muy familiarizado con este tipo de mapa.

Alternativa 2 Genera una lista de rutas cortas entre municipios

Esta alternativa contiene una lista ya generada de todos los 1122 municipios que existen en Colombia, esto funcionaría como una base de datos con todas las rutas cortas ya preparados para que el usuario las revise y siga en sus viajes. Estas rutas son además podrían representarse de manera gráfica si el usuario necesita alguna ayuda visualizando la ruta para que sus viajes sean de la manera más sencilla posible.

Alternativa 3 Crear una página web que permita mostrara las rutas más cortas

Esta alternativa consiste en como su nombre lo indica en crear una página web bajo un dominio. Este tendría todas los municipios de Colombia y un mapa que muestre de manera visual como se relacionan estos municipios y obviamente la parte más importante dejaría mostrar la ruta más corta entre todos los municipios de Colombia.

Alternativa 4 Crear una campaña de movilización:

Esta opción consiste en crear una campaña que le informe al público que vías tomar para minimizar su tiempo de viaje entre municipios de Colombia. Esta campaña tendría de público objetivo a gente que se transporte mucho entre municipios como camioneros, o turistas extranjeros que puede que no estén muy familiarizados con las rutas que conectan a estos municipios.

Alternativa 5: Crear un programa que permite mostrar la ruta más corta

En esta opción se propone crear un programa que permita crear un grafo con los vértices siendo ya los municipios y siendo conectado por aristas que representan las carreteras colombianas. Luego esta permitiría buscar la ruta más corta entre estos municipios.

Alternativa 6: Crear un programa que sirva de guía para viajar por Colombia

Esta alternativa es parecida a la pasada , pero la diferencia seria la presentación y algunas funciones adicionales que se podrían implementar para hacer la experiencia mucho más agradable y sencilla para el usuario, como agregar carreteras y municipio que no tengan registro o nuevas localidades que les gustaría visitar.

Fase 4 Transición de la formulación de ideas a los diseños preliminares:

Antes de empezar, de una vez podemos descartar algunas alternativas previamente planteadas en la fase 3. Primero la alternativa numero 1 debido a varios aspectos. Primero no tenemos el tiempo y nos los datos necesarios para poder generar un mapa adecuado o que le hiciera justicia a la complejidad a nivel espacial que tiene Colombia. Además, que nuestras habilidades como ingenieros no son suficientes para poder cumplir de manera efectiva esta alternativa.

La segunda alternativa también se puede descartar debido a que tomaría demasiado tiempo y un manejo de demasiados datos para poder ejecutar esta alternativa de la mejor manera posible. Además, tener una lista de las mejores rutas no sería muy intuitivo para el uso de los usuarios y hay mejores alternativas que se pueden utilizar.

La cuarta y tercera alternativa se pueden descartar por prácticamente las mismas razones. La cual es la falta de tiempo y experiencias que tenemos en el manejo y control de estos proyectos. Y nuestras áreas de experiencias y nuestras conexiones a nivel de campañas son pocas a no decir mínimas entonces ejecutar la idea 3 está fuera de nuestros alcances. Y la alternativa numero 4 no podemos generar páginas web y con el tiempo disponible no sería una alternativa plausible.

Empezamos descomponiendo las alternativas:

Alternativa 5: Crear un programa que permite mostrar la ruta más corta

- Se encuentra dentro de nuestra área de practica
- El diseño de la solución puede ser más sencilla de implementar
- Podemos utilizar librerías externas para facilitar la visualización del grafo
- La implementación de los algoritmos es más accesible
- El diseño de la solución es más sencillo

- Ya tenemos los algoritmos de la ruta más corta y su implementación debería ser sencilla

Alternativa 6: Crear un programa que sirva de guía para viajar por Colombia

- La implementación del programa es sencilla
- Tenemos una gran variedad de alternativas para desarrollar la aplicación
- Tenemos la opción de usar librerías externas para el desarrollo de la visualización del grafo
- Podemos implementar las demás funciones como agregar municipio y carretera de manera fácil y sencilla
- La implementación puede ser de la manera más eficiente posible.
- Tenemos ya los algoritmos de encontrar la distancia más corta.

Fase 5. Evaluación y selección de la mejor solución:

Criterios

Con el fin de seleccionar la mejor solución se presentarán ciertos criterios ponderados que al sumar los valores resultantes para cada solución se elegirá cual tenga el mayor puntaje. Los criterios son los siguientes:

- **Criterio A.** Complejidad de implementación:
 - [3] Fácil
 - [2] Medio
 - [1] Difícil
- **Criterio B.** Interfaz intuitiva:
 - [3] intuitiva
 - [2] normal
 - [1] confusa
- **Criterio C.** utilidades:
 - [3] muchas
 - [2] suficientes
 - [1] pocas

Evaluación

	Criterio A	Criterio B	Criterio C	Total
Alternativa 5: Crear un programa que	2	3	2	7

permite mostrar la ruta más corta				
Alternativa 6: Crear un programa que sirva de guía para viajar por Colombia	3	3	2	8

Selección

Según la evaluación se encuentra que la alternativa 6, aunque un poco mas complicada de implementar, presenta una mejor relación usuario-programa, además una mayor lista de utilidades en comparación a la alternativa 5, por lo tanto, la alternativa 6 será elegida.

Fase 6. Preparación de informes y especificaciones

Especificaciones del problema:

Problema: A partir de un mapa de Colombia, con municipios(vértices) siendo interconectados por carreteras(aristas) cada una con distancias variantes, encontrar la ruta mas corta entre dos municipios en el mapa

Entrada: Un vértice(municipio) de partida y un vértice(municipio) de llegada

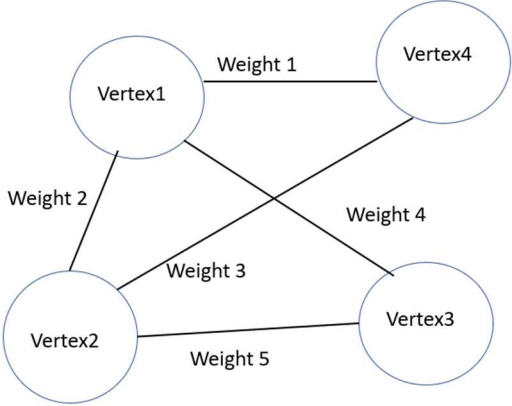
Salida: Una lista de las carreteras y municipios por las cual el viajero tendría que recorrer para llegar a su destino de la manera mas rápida posible.

Consideraciones:

- 1.) Los vértices que el usuario puede colocar solamente pueden estar dentro del grafo
- 2.) Solamente si el usuario desea agregar un nuevo vértice que no se encuentra en el mapa se podrá poner de llegada o partida
- 3.) No se pueden agregar vértices si una carretera que lo conecte con el resto del mapa
- 4.) La carretera siempre debe tener un peso asociado sino no se puede usar para las calculaciones

Tipos de datos abstractos (TAD):

TAD WeightedMatrixGraph:

TAD: WeightedMatrixGraph		
		
Inv{any two vertex can only have one edge connecting them}		
·	AddVertex: Vertex	→ Boolean
·	AddEdge: Vertex X Vertex X weight	→ Boolean
·	WeightedMatrixGraph:	→ Graph
·	SearchVertex: Texto	→ Vertex
·	DepthFirstSearch: Graph X Vertex	→ List
·	BreadthFirstSearch: Graph X Vertex	→ List
·	Dijkstra: Graph X Vertex	→ List
·	FloydWarshall: Graph	→ List

- **RemoveEdge: Vertex X Vertex → Boolean**
- **RemoveVertex Vertex → Boolean**

AddVertex(Vertex)

“Adds a vertex object to the graph”

{pre : Graph != null, Vertex \notin Graph }

{post Vertex added, Vertex \in Graph}

AddEdge(Vertex 1, Vertex 2, weight)

“Adds weighted Edge to the graph connecting two Vertexes belonging to the graph”

{pre : Graph != null, Vertex1 \wedge Vertex2 \in Graph \wedge weight \in Double }

{post Edge added in between two vertexes}

AddEdge(Vertex 1, Vertex 2)

“Adds non-weighted Edge to the graph connecting two Vertexes belonging to the graph”

{pre : Graph != null, Vertex1 \wedge Vertex2 \in Graph}

{post Edge added in between two vertexes}

WeightedMatrixGraph()

“Creates an empty graph, no vertices, no edges”

{pre : }

{post:empty Graph}

SearchVertex(name)

“from a text searches a vertex in the graph”

{pre : Vertex \in Graph[^] name \in Text }

{post returns Vertex}

DepthFirstSearch()

“Traverses the Graph in DepthFirst manner”

{pre : Graph != null}

{post returns list in order DFS}

BreadthFirstSearch()

“Traverses the Graph in BreadthFirst manner”

{pre :,Graph != null}

{post returns list in BFS}

Dijkstra(Vertex 1, Vertex 2)

“Searchs for the shortest distance between two Vertexs in the graph Graph”

{pre :Vertex1 ^ Vertex2 ∈ Graph , Graph != null}

{post: returns a vertex list with the vertexes forming the shortest path between the two vertexes}

FloydWarshall()

“Searches for the shortest distance between all the vertexes in the graphs”

{pre : Graph != null}

{post: returns the list of all the vertex forming the shortest path}

RemoveEdge(Vertex 1, Vertex 2)

“Eliminates the edge between two vertexes”

{pre : Vertex1 \wedge Vertex2 \in Graph, Graph \neq null}

{post: returns a boolean value, true if it could be removed, false if the edge could not be removed}

RemoveVertex(name)

“Eliminates a vertex of the graph”

{Vertex \in Graph \wedge name \in Text, Graph \neq null}

{post: returns the boolean value, true if the vertex was removed, false if the vertex was not}

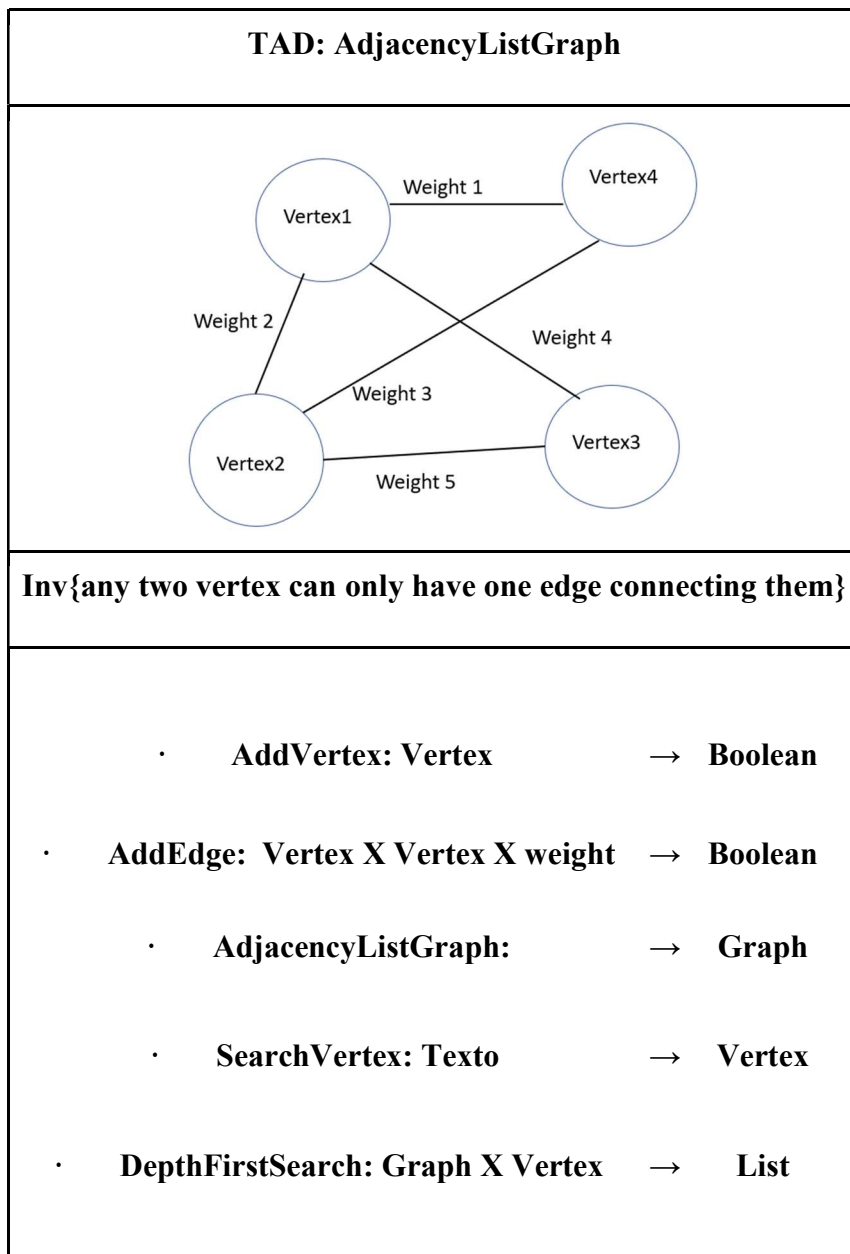
areConnected(Vertex 1, Vertex 2)

“Checks if two vertices are connected”

{pre : Vertex1 \wedge Vertex2 \in Graph, Graph \neq null}

{post: returns the boolean value, true if the verticves}

TAD AdjacencyListGraph:



- **BreadthFirstSearch: Graph X Vertex → List**
- **Dijkstra: Graph X Vertex → List**
- **FloydWarshall: Graph → List**
- **RemoveEdge: Vertex X Vertex → Boolean**
- **RemoveVertex Vertex → Boolean**

AddVertex(Vertex)

“Adds a vertex object to the graph”

{pre : Graph != null, Vertex \notin Graph }

{post Vertex added, Vertex \in Graph}

AddEdge(Vertex 1, Vertex 2, weight)

“Adds weighted Edge to the graph connecting two Vertexes belonging to the graph”

{pre : Graph != null, Vertex1 \wedge Vertex2 \in Graph \wedge weight \in Double }

{post Edge added in between two vertexes}

AddEdge(Vertex 1, Vertex 2)

“Adds non-weighted Edge to the graph connecting two Vertexes belonging to the graph”

{pre : Graph != null, Vertex1 \wedge Vertex2 \in Graph}

{post Edge added in between two vertexes}

AdjacencyListGraph()

“Creates an empty graph, no vertices, no edges”

{pre : }

{post:empty Graph}

SearchVertex(name)

“from a text searches a vertex in the graph”

{pre : Vertex \in Graph[^] name \in Text }

{post returns Vertex}

DepthFirstSearch()

“Traverses the Graph in DepthFirst manner”

{pre : Graph != null}

{post returns list in order DFS}

BreadthFirstSearch()

“Traverses the Graph in BreadthFirst manner”

{pre :,Graph != null}

{post returns list in BFS}

Dijkstra(Vertex 1, Vertex 2)

“Searchs for the shortest distance between two Vertexs in the graph Graph”

{pre :Vertex1 ^ Vertex2 ∈ Graph , Graph != null}

{post: returns a vertex list with the vertexes forming the shortest path between the two vertexes}

FloydWarshall()

“Searches for the shortest distance between all the vertexes in the graphs”

{pre : Graph != null}

{post: returns the list of all the vertex forming the shortest path}

RemoveEdge(Vertex 1, Vertex 2)

“Eliminates the edge between two vertexes”

{pre : Vertex1 \wedge Vertex2 \in Graph, Graph \neq null}

{post: returns a boolean value, true if it could be removed, false if the edge could not be removed}

RemoveVertex(name)

“Eliminates a vertex of the graph”

{Vertex \in Graph \wedge name \in Text, Graph \neq null}

{post: returns the boolean value, true if the vertex was removed, false if the vertex was not}

areConnected(Vertex 1, Vertex 2)

“Checks if two vertices are connected”

{pre : Vertex1 \wedge Vertex2 \in Graph, Graph \neq null}

{post: returns the boolean value, true if the verticves}

Diseño de pruebas unitarias:

Escenarios:

nombre	clase	Escenario
SetupScenary()	WeightMatrixGraph	WeightMatrixGraph WG = new WeightMatrixGraph(false, 4)
SetupScenary1()	AdjencyListGraph	AdjencyListGraph AG = new AdjencyListGraph(False)
SetupScenary2()	GraphAlgorithims	WeightMatrixGraph WG = new WeightMatrixGraph(false, 4) AdjencyListGraph AG = new AdjencyListGraph(False) GraphAlgorithims = GA = new GraphAlgorithims()

Objetivo de la prueba : Comprobar que se creo el grafo Correctamente				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	WeightMatrixGraph()	SetupScenary()		Se creo el grafo de manera adecuada

Objetivo de la prueba : Comprobar que se agrego un vertice de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	AddVertex	SetupScenary()	V vertex V vertex1	Se Agrego El vértice de manera correcta

Objetivo de la prueba : Comprobar que se agrego una arista al grafo de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	AddEdge	SetupScenary()	V vertex V vertex1 double w = 5	Se agrego la arista de manera correcta al grafo

Objetivo de la prueba : Comprobar que se quito la arista de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	RemoveEdge	SetupScenary()	V vertex V vertex1	Se elimino la arista para el par de vertices

Objetivo de la prueba : Comprobar que se quito el vertice de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	RemoveVertex	SetupScenary()	V vertex	Se elimino correctamente el vertice

Objetivo de la prueba : Comprobar que el grafo se encuentre dirigido				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	isDirected	SetupScenary()		El grafo se encuentra de manera dirigida

Objetivo de la prueba : Comprobar que el tamaño del grafo sea el indicado				
Clase	Método	Escenario	Valores de Entrada	Resultado
WeightMatrixGraph	VertexSize()	SetupScenary()		El tamaño del grafo es indicado

AdjencyList Graph :

Objetivo de la prueba : Comprobar que se creo el grafo Correctamente				
Clase	Método	Escenario	Valores de Entrada	Resultado

AdjencyListGraph	AdjencyListGraph()	SetupScenary1()		Se creo el grafo de manera adecuada
------------------	--------------------	-----------------	--	-------------------------------------

Objetivo de la prueba : Comprobar que se agrego un vertice de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	AddVertex	SetupScenary1()	V vertex V vertex1	Se Agrego El vértice de manera correcta

Objetivo de la prueba : Comprobar que se agrego una arista al grafo de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	AddEdge	SetupScenary()	V vertex V vertex1 double w = 5	Se agrego la arista de manera correcta al grafo

Objetivo de la prueba : Comprobar que se quito la arista de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	RemoveEdge	SetupScenary1()	V vertex V vertex1	Se elimino la arista para el par de vertices

Objetivo de la prueba : Comprobar que se quito el vertice de manera correcta				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	RemoveVertex	SetupScenary1()	V vertex	Se elimino correctamente el vertice

Objetivo de la prueba : Comprobar que el grafo se encuentre dirigido				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	isDirected	SetupScenary1()		El grafo se encuentra de manera dirigida

Objetivo de la prueba : Comprobar que dos vértices estén conectados				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	isConnected	SetupScenary1()	V vertex V vertex1	Los dos vértices se encuentran conectados

Objetivo de la prueba : Comprobar que el tamaño del grafo sea el indicado				
Clase	Método	Escenario	Valores de Entrada	Resultado
AdjencyListGraph	VertexSize()	SetupScenary1()		El tamaño del grafo es indicado

GraphAlgorithims:

Objetivo de la prueba : Comprobar que se recorrió de manera DFS el grafo				
Clase	Método	Escenario	Valores de Entrada	Resultado
GraphAlgorithims	DepthFirstSearch	SetupScenary2()		Se recorrió de manera correcta el grafo

Objetivo de la prueba : Comprobar que se recorrió de manera BFS el grafo				
Clase	Método	Escenario	Valores de Entrada	Resultado
GraphAlgorithims	BreadthFirstSearch	SetupScenary2()		Se recorrió de manera correcta el grafo

Objetivo de la prueba : Comprobar que se encontró de manera correcta el recorrido mas corto				
Clase	Método	Escenario	Valores de Entrada	Resultado
GraphAlgorithims	Dijkstra()	SetupScenary2()	V vertex V vertex1 V vertex2 V vertex3	Se encontró de manera correcta el recorrido mas corto para los vertices

Objetivo de la prueba : Comprobar que se encontró de manera correcta el recorrido mas corto				
Clase	Método	Escenario	Valores de Entrada	Resultado
GraphAlgorithims	FloydWarshall	SetupScenary2()	V vertex V vertex1 V vertex2 V vertex3	Se encontró de manera correcta el recorrido mas corto para los vertices

Objetivo de la prueba : Comprobar que se encontró el vertice indicado				
Clase	Método	Escenario	Valores de Entrada	Resultado

GraphAlgorithms	Search()	SetupScenary2()	String name = "name1"	Se encontró el vertice indicado
-----------------	----------	-----------------	-----------------------	---------------------------------

Diagrama de clases:

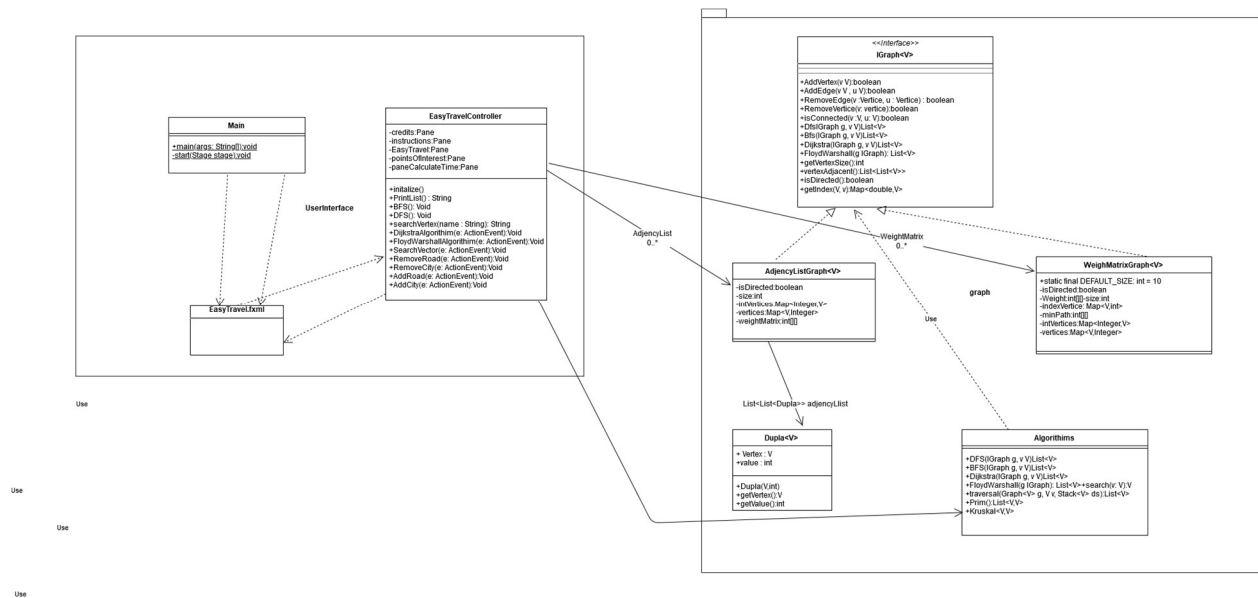
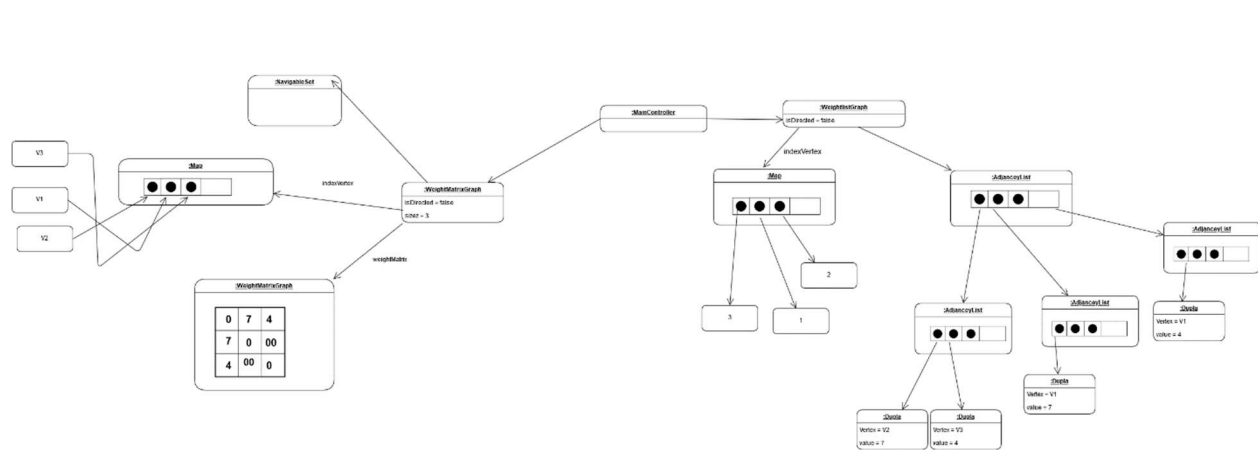


Diagrama d objetos:



Fase 7 Implementación del Diseño:

Lista de Tareas a implementar:

- a.) Generar el grafo con los municipios predeterminados
- b.) Mostrar el grafo al usuario de manera grafica
- c.) A partir del punto de partida y llegada planteado por el usuario generar el camino más corto hacia el
- d.) Si el usuario lo desea agregar municipios al grafo con todas las especificaciones planteadas anteriormente
- e.) Agregar mas carreteras con su duración de tiempo si el usuario lo desea

Especificaciones de subrutinas:

Descripción: calcula el camino mas corto entre el vértice y los demás vértices

Entrada: int [][] graph una matriz de pesos que representa el grafo, in v un valor numérico que representa el vertice

Salida: void no tiene salida

```
public static <V> void dijkstra(int[][] g, int v){
    int[] dis = new int[g.length];
    boolean[] vis = new boolean[g.length];

    for (int i=0; i < dis.length; i++) {
        dis[i] = Integer.MAX_VALUE;
    }
    dis[v] = 0;

    for (int i=0; i<v-1; i++) {
        int u = minIndex(dis, vis);
```

```
vis[u] = true;

for (int j=0; j<v; j++) {
    if (!vis[v] && g[u][v] != 0 && dis[u] !=
Integer.MAX_VALUE && dis[u] + g[u][v] < dis[v])
        dis[v] = dis[u] + g[u][v];
}
}
```