**Algoritmos y Estructuras de datos**

**Proyecto Final Método de la ingeniería**

**NICOLAS TABORDA HOYOS; LAURA HINCAPIE; JULIAN DAVID MABESOY; JAMES MONTEALEGRE**

# FASE 1: Identificación del problema

**Contexto Problemático:**

Eurail es el pase de tren numero 1 favorito entre las personas no residentes de Europa que viajan entre diferentes ciudades del continente. Es seguro, puedes hacer reservas y todos sus medios de pago no incluyen costos adicionales. Su servicio de trenes se encuentra en más de 60 ciudades de Europa, con estaciones principales en las ciudades más visitadas como lo son Madrid, Frankfurt, Estocolmo, Múnich, Praga, Budapest, entre otras.

Su oferta de más de 90 recorridos por las diferentes ciudades de Europa hace que exista más de un recorrido para llegar de una ciudad a otra, por esto se ha querido desarrollar una herramienta que permita a los turistas, calcular la manera más rápida para llegar a su destino desde la estación de tren en la que se encuentren.

Nuestra tarea como desarrolladores de esta herramienta es mostrarle al usuario todas las estaciones de tren disponibles y permitirle indicar un punto de partida entre ellas, y un punto de destino, y con esto la herramienta le mostrara cuales rutas de tren tomar para que el recorrido sea el más corto posible. También esta herramienta permitirá ver a los usuarios los diferentes lugares que pueden visitar con recorridos directos, desde una estación específica. El indicador para sugerir las mejores rutas será el tiempo, pues en este caso los precios de las diferentes rutas son muy parecidos.

## **Identificación del problema**

El problema que se desea resolver es permitirle al usuario de nuestro programa visualizar todas las estaciones de trenes de Europa y que pueda planear viajes teniendo un tiempo mínimo entre las estaciones que seleccione.

**Requerimientos Funcionales**

1. Requerimiento 1

|  |  |
| --- | --- |
| **Nombre** | Mostrar el recorrido más corto |
| **Resumen** | De todas las rutas de tren, se muestra cual es el recorrido más corto para llegar a una ciudad |
| **Entradas** | Ciudad de origen  Ciudad de destino |
| **Salidas** | No. De rutas a tomar  Nombre de las rutas a tomar  Tiempo total |

1. Requerimiento 2

|  |  |
| --- | --- |
| **Nombre** | Mostrar rutas de una ciudad |
| **Resumen** | Dado un punto de origen, se muestra todas las ciudades que tienen trenes directos desde esta ciudad |
| **Entradas** | Ciudad de origen |
| **Salidas** | No. De rutas con trenes directos  Rutas directas |

# FASE 2: Recopilación de la información

Estructuras de datos: Las estructuras de datos las podemos ver como formas de representar información, las cuales tienen un comportamiento interno específico, se rigen por determinadas reglas y tienen algunas restricciones condicionadas por la forma en la que están construidas internamente.

Estas estructuras son muy útiles pues nos permiten en algunos casos, solucionar problemas de manera muy sencilla. Entre estas podemos identificar los arreglos, que son estructuras de datos de acceso aleatorio y las listas enlazadas, las cuales con estructuras de datos de acceso secuencial.

Grafo: Un grafo es un tipo abstracto de datos que se conforma de nodos llamados vértices y arcos llamados aristas que establecen relaciones entre los nodos. Mate matemáticamente un grafo se define como G = (V, E), siendo V los vértices del grafo y E las aristas las cuales son pares de elementos en V.

Grafo dirigido: O también llamado dígrafo es un tipo de grafo en el cual las aristas tienen un sentido definido, a diferencia del grafo no dirigido, en el cual sus aristas son relaciones simétricas y no apuntan en ningún sentido.

Adyacencia: Dos vértices son adyacentes si forman un lado.

Incidencia: Un lado incide en los dos vértices que une.

Grado de un vértice: Es el número de lados que inciden en el vértice (número de ramificaciones que salen del vértice).

Trayectoria i-j: Son los vértices por los que hay que pasar para ir desde el vértice I hasta el vértice J.

Bosque: Un bosque es un conjunto de árboles; o de forma equivalente, un bosque es un grafo acíclico.

Bucle: Un bucle en un grafo es una arista que conecta al mismo vértice consigo mismo. Un grafo simple no puede tener bucles.

BFS: Es un algoritmo que permite recorrer todos los vértices de un árbol de manera ordenada, recorriendo primero los vértices vecinos a la inicial, luego los vértices vecinos a los recorridos en el paso anterior y así sucesivamente hasta agotar la gráfica.

DFS: es un algoritmo que permite recorrer todos los vértices de un árbol de manera ordenada, avanzando sobre cada rama hasta que no haya posibilidad de continuar y luego se retrocede hasta la última bifurcación para seguir por otra rama. Puede usarse para recorrer un grafo cualquiera si se usa un árbol generador del grafo.

Camino: Un camino es una sucesión de vértices tal que de cada uno de sus vértices existe una arista hacia el vértice sucesor. Un camino simple es aquel que no repite vértices en su recorrido. Dos caminos son ajenos o independientes si no tienen ningún vértice en común excepto el primero y el último.

Camino euleriano: Un camino euleriano en un grafo es un camino que usa cada arista una y sólo una vez. Si existe tal camino decimos que el grafo es euleriano.

Camino hamiltoniano: Un camino hamiltoniano en un grafo es un camino que "visita" cada vértice una y sólo una vez.

Ciclo: Un Ciclo es un camino que empieza y acaba en el mismo vértice. Los ciclos de longitud 1 se denominan lazos o bucles.

Ciclo simple: es un ciclo que tiene como longitud al menos 3 y en el que el vértice inicial coincide con el vértice final.

Ciclo euleriano: Un ciclo euleriano en un grafo es un ciclo que usa cada arista una y sólo una vez.

Ciclo hamiltoniano: Un ciclo hamiltoniano en un grafo es un ciclo que visita cada vértice una y sólo una vez

Componente fuertemente conexo: Un componente fuertemente conexo es un grafo tal que, para cada par de vértices, existe un camino de uno hacia el otro, y viceversa.

Componentes fuertemente conexos de un grafo dirigido: son sus subgrafos máximos fuertemente conexos. Estos subgrafos forman una partición del grafo.

Grafo bipartito: Un grafo bipartito es cualquier grafo cuyos vértices pueden ser divididos en dos conjuntos, tal que no haya aristas entre los vértices del mismo conjunto. Se ve que un grafo es bipartito si no hay ciclos de longitud impar

Isomorfismo: Un Isomorfismo de grafos entre dos grafos G y H es una biyección f entre los conjuntos de sus vértices que preserva la relación de adyacencia. Es decir, cualquier par de vértices u y v de G son adyacentes si y solo si lo son sus imágenes, f(u) y f(v), en H.

Matriz de adyacencia: Una matriz de adyacencia es una matriz de n x n que permite representar un grafo o dígrafo finito, donde cada valor en la posición (i, j) representa el número de aristas desde el vértice i-ésimo al j-ésimo.

Subgrafo: Un subgrafo de un grafo G es un grafo cuyo conjunto de vértices es un subconjunto del de G, cuyo conjunto de aristas es un subconjunto del conjunto de las aristas de G, y tal que la aplicación w es la restricción de la aplicación de G.

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

Algoritmo de Prim: El algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

Algoritmo de Dijkstra: También llamado algoritmo de caminos mínimos es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista

Algoritmo de Floyd Warshall: El problema que intenta resolver este algoritmo es el de encontrar el camino más corto entre todos los pares de nodos o vértices de un grafo.

Fuentes:

https://sites.google.com/site/jlscestructuras/grafos-no-dirigidos  
https://es.wikipedia.org/wiki/Grafo\_(tipo\_de\_dato\_abstracto)  
https://es.wikipedia.org/wiki/Anexo:Glosario\_de\_teor%C3%ADa\_de\_grafos#Grafo\_ac%C3%ADclico  
https://www.ecured.cu/Algoritmo\_de\_Kruskal  
http://arodrigu.webs.upv.es/grafos/doku.php?id=algoritmo\_floyd\_warshall

# FASE 3: Búsqueda de soluciones creativas

1. Primer requerimiento: Mostrar el recorrido más corto

* Alternativa 1: Se podrían tomar a cada ciudad como un constante tipo int, después añadirla en un árbol binario de búsqueda, y con ayuda de los métodos de recorrido preOrden, posOrden y inorden buscar la ciudad de destino tomando la ciudad de origen como padre.
* Alternativa 2: Se podría utilizar un grafo simple, el cual cada vértice se podría ver como una ciudad y una arista como la vía que conecta a cada ciudad con las otras, con ayuda del peso sabremos el tiempo de que hay entre la ciudad de origen y la de destino.
* Alternativa 3: Se podría utilizar un multígrafo dirigido, el cual cada vértice se podría ver como una ciudad y una arista como la vía que conecta a cada ciudad con las otras, con ayuda del peso sabremos el tiempo de que hay entre la ciudad de origen y la de destino.

1. Segundo Requerimiento mostrar rutas directas de una ciudad

* Alternativa 1: Se debe utilizar el algoritmo Dijkstra para hallar la ruta directa y con menos costos desde una ciudad a otra. Utilizando un grafo simple no dirigido.
* Alternativa 2: Se utilizaría el algoritmo de Prim para hallar el árbol generador mínimo de todas las rutas del tren.
* Alternativa 3: Se podría utilizar el algoritmo de Floyd Warshall para hallar los recorridos mínimos entre todos los pares de ciudades en el grafo simple.

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

# IDEAS NO VIABLES

Del requerimiento 1 se decidió descartar la alternativa 1 ya que implicaría la utilización de un árbol para almacenar las ciudades, esta estructura de datos es muy útil a la hora de almacenar datos y realizar búsquedas, por esto es por lo que decidimos descartar la utilización de un árbol de búsqueda, ya que con otras estructuras de datos se nos facilitaría la implementación del requerimiento esperado a implementar.

Para el requerimiento 2, se decidió descartar la alternativa 2 ya que el requerimiento lo que solicita es hallar el camino mínimo entre dos ciudades ingresadas, y no pide generar un árbol mínimo, por esta razón es que la alternativa 2 no es muy viable mirando las otras alternativas planteadas en las ideas.

# DISEÑO PRELIMINAR DE IDEAS VIABLES POR IMPLEMENTAR

Después de realizar la respectiva evaluación para todas las ideas, se decidieron descartar las ideas que se mencionaron anteriormente, dejando dos alternativas para el desarrollo de cada requerimiento.

A partir de estas alternativas, se pretende llegar a la que más convenga a la hora de realizar las implementaciones y que cumple con los criterios que se establecerán más adelante.

De cada requerimiento nos quedan dos alternativas para la resolución de cada requerimiento, del primer requerimiento quedó la implementación de un grafo simple adicionando a las ciudades en el grafo como vértices o nodos, y las aristas de estas ciudades el costo que tomaría viajar entre estas dos aristas en cualquier sentido. La segunda alternativa que quedo para resolver el requerimiento 1 es la utilización de un multígrafo dirigido, implementando las ciudades del recorrido como vértices o nodos y las aristas de estos nodos siendo una para ir de una ciudad, por ejemplo, de “A” a “B” y otra arista para ir en el sentido “B” a “A”, con sus costos de viaje entre las ciudades seleccionadas.

Del requerimiento 2 nos quedaron también dos alternativas que deben ser evaluadas posteriormente en la fase 5, la primer alternativa que nos quedo fue la implementación de el algoritmo Dijkstra ya que este algoritmo con permite mirar la ruta con menor “costo” o “peso” entre dos ciudades indicadas, y la segunda alternativa que quedo para resolver el requerimiento 2 fue la implementación del algoritmo Floyd Warshall ya que este algoritmo nos permite calcular la ruta con menor “costo” o “peso” entre todos los pares de vértices o “ciudades” del grafo.

# FASE 5: Evaluación y selección de la mejor solución

A partir de las ideas que se realizaron anteriormente, se deben generar unos criterios de evaluación que permita clasificar las ideas de acuerdo con su calificación. Con base en el resultado obtenido después de la evaluación, se tomará la decisión de cuál alternativa implementar. A continuación, se encuentran los criterios que se escogieron. Cada uno tiene asociado varios valores numéricos para determinar el puntaje de cada alternativa.

*-Criterio A:* Precisión de la solución.Es solucionado el requerimiento de manera:

* [2]Completa
* [1]Incompleta

*-Criterio B*: Eficiencia. El requerimiento es resuelto de un modo:

* [4] Muy eficiente
* [3] Eficiente
* [2] Ineficiente
* [1] Muy deficiente

*-Criterio C*: Facilidad de codificar. ¿A partir de lo planeado, la solución permite desarrollar un algoritmo que cumpla con el requerimiento y que sea manejable a la hora de llevarlo al código?

* [3] Manejable de llevar a la implementación en java
* [2] No es manejable, pero permite que sea llevado a la implementación en java
* [1] No es manejable y tampoco se puede llevar a la implementación en java

Requerimiento 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
| *Alternativa 2* | Completa [2] | *Muy eficiente [4]* | *Manejable [3]* | *9* |
| *Alternativa 3* | *Completa [2]* | Eficiente [3] | *Manejable [3]* | *8* |

Requerimiento 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
| *Alternativa 1* | *Completa [2]* | *Muy eficiente [4]* | *Manejable [3]* | *9* |
| *Alternativa 3* | *Completa [2]* | *Ineficiente [2]* | *Manejable [3]* | *7* |

Para el primer requerimiento la implementación de un multígrafo dirigido es una buena implementación para resolver el problema del requerimiento 1, pero realizar dobles aristas entre cada vértice es algo innecesario ya que con un grafo simple no dirigido, se puede entender que se puede viajar entre un par de nodos por medio de una arista no dirigida en cualquier dirección, por esta razón es que la alternativa 2 que es implementar un grafo simple no dirigido es la mejor opción para resolver el problema del requerimiento 1.

Por otra parte, para el requerimiento 2 la implementación del algoritmo Floyd Warshall es algo que no es necesario ya que solo se necesitaría saber el peso o “costo” mínimo entre dos vértices ingresados por el usuario, y así hallar la ruta, el algoritmo Floyd Warshall nos da las rutas con menor peso de todos los vértices y el requerimiento no pide lo que nos da la implementación de este algoritmo, por lo tanto se decidió implementar el algoritmo Dijkstra ya que ingresados dos nodos, la implementación de este algoritmo nos arroja la ruta de menor peso entre los dos nodos ingresados y se mostraría al usuario.

Después de realizarse la respectiva calificación y clasificación de las alternativas se llegaron a 2 alternativas que serían las que se implementaran en el código.

1. Requerimiento 1: Implementación de un grafo simple no dirigido para representar el mapa ferroviario de Europa y mostrar el recorrido mas corto
2. Requerimiento 2: Implementación de algoritmo Dijkstra para obtener la ruta con menor peso entre dos nos indicadas por el usuario.

# FASE 6: Preparación de informes y especificaciones

*Diseño de casos de pruebas unitarias*

## Prueba Algorithms

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método Kruskal(Graph<T> graph, String type) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: Algorithms* | | | *Método: +* *Kruskal(Graph<T> graph, String type): Graph<T>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método Kruskal(Graph<T> graph, String type) devuelve el árbol generador mínimo* | *Grafo con 7 vértices no direccionado y con peso.* | | *Grafo con 7 vértices no direccionado y con peso.* | *Árbol generador mínimo con peso igual a 52* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método BFS(Graph<T> graph, T startNode) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: Algorithms* | | | *Método: +* *BFS(Graph<T> graph, T startNode):* *ArrayList<T>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método BFS(Graph<T> graph, T startNode) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y sin peso. (Escenario 1)* | | *Grafo con 6 vértices no direccionado no ponderado y la ciudad donde empieza el recorrido* | *ArrayList<T> con las ciudades exploradas.* |
| *2* | *Verifica que el método BFS(Graph<T> graph, T startNode) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y sin peso. Diferentes valores en los vertices del grafo (Escenario 3)* | | *Grafo con 6 vértices no direccionado no ponderado y el nodo donde empieza el recorrido* | *ArrayList<T> con los nodos exploradas.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método* *DFS(Graph<T> graph, T startNode) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: Algorithms* | | | *Método: +* *DFS(Graph<T> graph, T startNode): ArrayList<T>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método DFS(Graph<T> graph, T startNode) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y sin peso. (Escenario 1)* | | *Grafo con 6 vértices no direccionado no ponderado y la ciudad donde empieza el recorrido* | *ArrayList<T> con las ciudades exploradas.* |
| *2* | *Verifica que el método DFS(Graph<T> graph, T startNode) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y sin peso. (Escenario 3)* | | *Grafo con 6 vértices no direccionado no ponderado y la ciudad donde empieza el recorrido* | *ArrayList<T> con las ciudades exploradas.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método* *DijkstraAlgorithm(Graph<T>graph, T start, T end) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: Algorithms* | | | *Método: +* *DijkstraAlgorithm(Graph<T>graph, T start, T end): AuxDijkstra<Integer,List<T>>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método DijkstraAlgorithm(Graph<T>graph, T start, T end) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y con peso. (Escenario 5)* | | *Grafo con 6 vértices no direccionado ponderado y la ciudad donde empieza el recorrido y donde termina.* | *AuxDijkstra*  *<Integer,List<T>>*  *Nuevo objeto creado con el camino que debe recorrer desde el inicio al final y la distancia total.* |
| *2* | *Verifica que el método DijkstraAlgorithm(Graph<T>graph, T start, T end) hace el recorrido BFS sobre el árbol.* | *Grafo con 6 vértices no direccionado y con peso. (Escenario 4)* | | *Grafo con 6 vértices no direccionado ponderado y la ciudad donde empieza el recorrido y donde termina.* | *AuxDijkstra*  *<Integer,List<T>>*  *Nuevo objeto creado con el camino que debe recorrer desde el inicio al final y la distancia total.* |

## Prueba ListGraph

## 

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método addEdge(T n1, T n2, int weight) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: ListGraph* | | | *Método: +* *addEdge(T n1, T n2, int weight):void* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método addEdge(T n1, T n2, int weight) agrega una arista entre dos vértices con un peso indicado* | *Grafo con 4 vértices no direccionado y con peso. (Escenario 3)* | | Dos vértices en los cuales se les quiere agregar la arista con su respectivo peso. (3,4,12) | *Arista agregada entre el vértice 3 y 4 con un peso de 12.* |
| *2* | *Verifica que el método addEdge(T n1, T n2, int weight) agrega una arista entre dos vértices con un peso indicado* | *Grafo con 4 vértices direccionados y con peso. (Escenario 4)* | | Dos vértices en los cuales se les quiere agregar la arista con su respectivo peso. (3,4,8) | *Arista agregada entre el vértice 3 y 4 con un peso de 8.* |
| *3* | *Verifica que el método addEdge(T n1, T n2, int weight) agrega una arista entre dos vértices con un peso indicado* | *Grafo con 4 vértices no direccionados y sin peso. (Escenario 1)* | |  | *Verifica si los nodos 1 y 2 tiene vértice, y el nodo 4 y 1 tiene vértice.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método deleteNode(T node) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: ListGraph* | | | *Método: +*  *deleteNode(T node): boolean* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método deleteNode(T node) elimina el nodo ingresado.* | *Grafo con 4 vértices no direccionado y con peso. (Escenario 3)* | | Vértice que debe ser eliminado. (1) | *Eliminación del nodo 1 y se verifica su eliminación* |
| *2* | *Verifica que el método deleteNode(T node) elimina el nodo ingresado.* | *Grafo con 4 vértices direccionados y con peso. (Escenario 4)* | | Vértice que debe ser eliminado. (1) | *Eliminación del nodo 1 y se verifica su eliminación* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método deleteEdge(T node1, T node2) funciona correctamente en diferentes ocasiones* | | | | | |
| *Clase: ListGraph* | | | *Método: +*  *deleteEdge(T node1, T node2): boolean* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método deleteEdge(T node1, T node2) elimina la arista entre los nodos ingresados.* | *Grafo con 4 vértices no direccionado y con peso. (Escenario 3)* | | Par de nodos entre los cuales debe ser eliminada la arista. (1,2) | *Eliminación de la arista entre los nodos 1 y 2* |
| *2* | *Verifica que el método deleteEdge(T node1, T node2) elimina la arista entre los nodos ingresados.* | *Grafo con 4 vértices direccionados y con peso. (Escenario 4)* | | Par de nodos entre los cuales debe ser eliminada la arista. (1,2) | *Eliminación de la arista entre los nodos 1 y 2* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método addNode(T node) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: ListGraph* | | | *Método: +* *addNode(T node):void* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método addNode(T node) agregue un nodo.* | *Grafo con 4 vértices no direccionado y sin peso. (Escenario 1)* | | *Nodo para agregar (10)* | *Nodo 10 agregado con su respectiva verificación en el grafo.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método isEdge(T node1, T node2) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: ListGraph* | | | *Método: +* *isEdge(T node1, T node2): boolean* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método isEdge(T node1, T node2) existe una arista entre los vértices ingresados, no direccionados.* | *Grafo con 4 vértices no direccionado y sin peso. (Escenario 1)* | | *Ingresa los valores de los nodos 1 y 2.* | *True, ya que si hay una arista entre estos dos vertices.* |
| *2* | *Verifica que el método isEdge(T node1, T node2) existe una arista entre los vértices ingresados, la arista es direccionada.* | *Grafo con 4 vértices direccionados y sin peso. (Escenario 2)* | | *Ingresa los valores de los nodos 2 y 1. 1 y 2* | *Si hay dirección en la entrada 1 y 2, pero en la 2 y 1 no hay dirección.* |

## Prueba MatrixGraph

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método addNode(T node) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *addNode(T node):void* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método addNode(T node) agregue un nodo.* | Matriz de nodos con capacidad de 10 nodos, pero se encuentra vacía (Escenario 1) | | Ingresan 4 nodos a la matriz | *Se agregan 4 nodos a la matriz de nodos y se verifica su tamaño después de ser agregados* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método addEdge(T node1, T node2, int weight) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *addEdge(T node1, T node2, int weight):void* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método addEdge(T node1, T node2, int weight) agregue una arista entre dos nodos con un peso ingresado* | Matriz de nodos con capacidad de 10 nodos, pero se encuentra vacía (Escenario 1) | | Ingresan 6 nodos a la matriz y luego se agregan 6 vértices entre los nodos *(Cali, Bogotá)*  *(Bogotá. Medellín)*  *(Cartagena, Santa Marta)*  *(Cartagena, Cali)*  *(Pereira, Medellín)*  *(Bogotá Cali)* | *Se agregan 6 nodos a la matriz de nodos y se agregan vertices entre (Cali, Bogotá)*  *(Bogotá. Medellín)*  *(Cartagena, Santa Marta)*  *(Cartagena, Cali)*  *(Pereira, Medellín)*  *(Bogotá Cali)* |

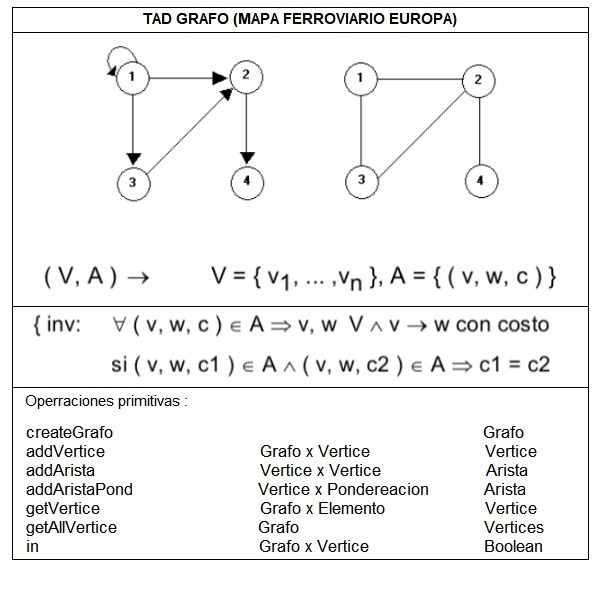
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método isEdge(T node1, T node2) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *isEdge(T node1, T node2): boolean* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método isEdge(T node1, T node2) existe una arista entre los vértices ingresados, la arista es direccionada* | Matriz de nodos con capacidad de 5 nodos, con cuatro nodos ya en la matriz y con aristas ya creadas. (Escenario 3) | | Verifica que haya aristas entre los nodos (A, D) (C, B)  (A, B) | *Si hay aristas entre los vértices a verificar.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método isNode(T node) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *isNode(T node):boolean* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método isNode(T node) verifique que el nodo ingresado es un nodo en la matriz.* | Matriz de nodos con capacidad de 5 nodos, con cuatro nodos ya en la matriz y con aristas ya creadas. (Escenario 3) | | Nodo para verificar (D) | *El nodo si es un nodo de la matriz.* |

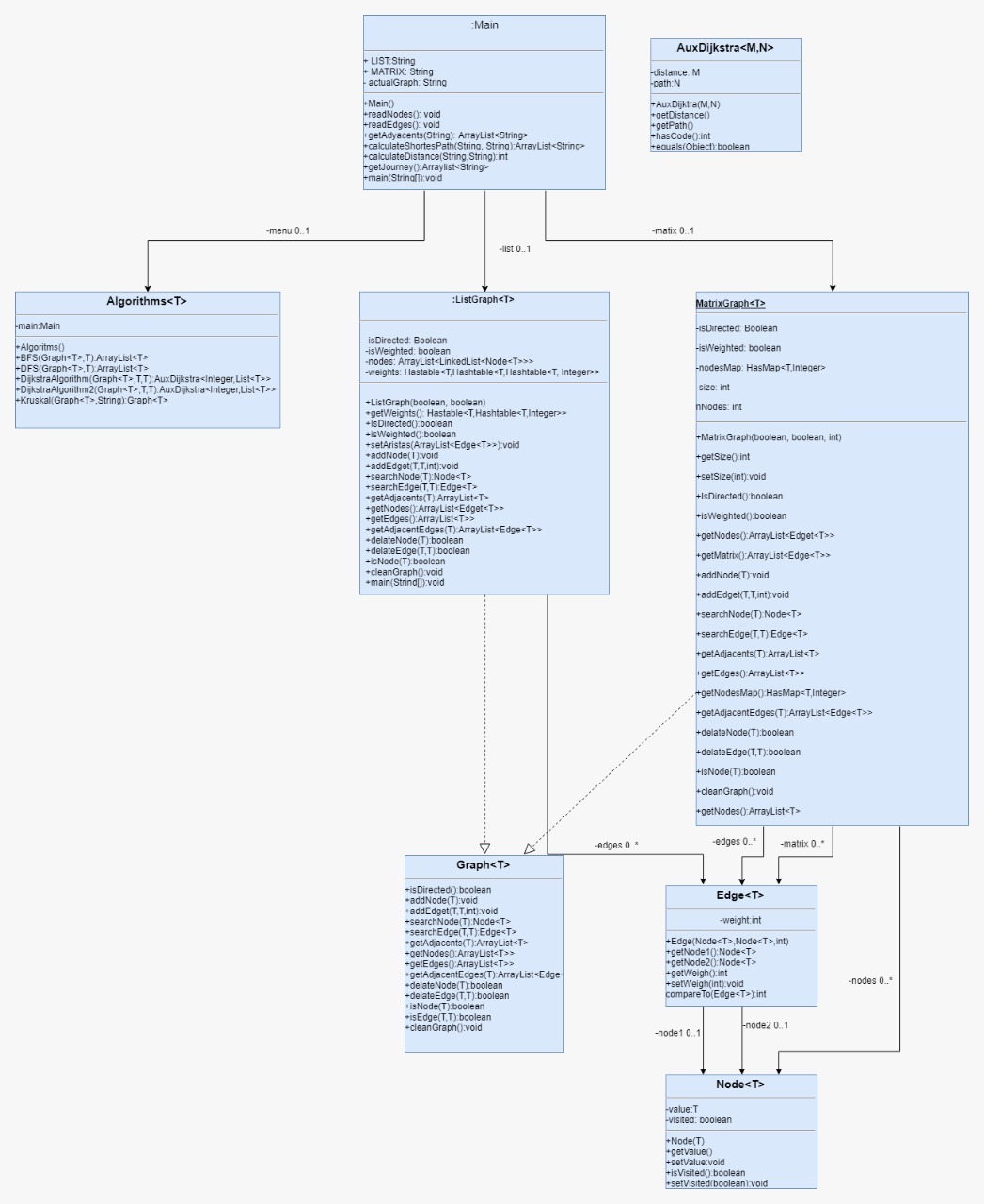
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método searchEdge(T node1, T node2) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *searchEdge(T node1, T node2): Edge* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método searchEdge(T node1, T node2) verifique que entre los nodos ingresado hay una arista y la devuelva.* | Matriz de nodos con capacidad de 5 nodos, con cuatro nodos ya en la matriz y con aristas ya creadas. (Escenario 3) | | Nodos para verificar si entre ellos hay aristas (A, C) (A, D) | *Entre los nodos (A, C) no hay arista y entre los nodos (A, D) si hay arista.* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Objetivo: Probar que el método getAdjacents(T node) funciona correctamente en diferente ocasiones* | | | | | |
| *Clase: MatrixGraph* | | | *Método: +* *getAdjacents(T node):* *ArrayList<T>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que el método getAdjacents(T node) verifique que devuelva los nodos adyacentes del nodo ingresado* | Matriz de nodos con capacidad de 5 nodos, con cuatro nodos ya en la matriz y con aristas ya creadas. (Escenario 3) | | Nodo para verificar sus adyacentes (A) | *ArrayList<T> != null* |

**TAD GRAFO**



# Diagrama de clases Modelo



# Diagrama de clases Interfaz

