Universidad Icesi

Facultad de ingeniería

Algoritmos y Estructuras de Datos

Profesor: Juan Manuel Reyes

Tarea integradora III

Integrantes

Alejandra Diaz Parra

Juan Fernando Martinez

Santiago Rodas Rodriguez

Método de la ingeniería

Fase 1: Identificación del problema

El planeta Tierra ha llegado a tal punto crítico de inconvenientes y dificultades que una de las posibles soluciones para solucionar estos problemas es embarcarse en una misión para explorar los confines de la Vía Láctea. Es por esta razón que distintos países se han unido para construir una nave, en este caso llamada “USS DISCOVERY”. Ahora bien, al ser una nave interplanetaria, se necesita de un sistema completo de navegación en la nave, basado en una base de datos de sistemas planetarios existentes, claramente con rutas de navegación entre ellos. Con esto, no sólo se desea calcular el recorrido con menor tiempo de un planeta a otro, sino también de conocer la mayor cantidad posible de civilizaciones. Tomando en cuenta esa información introductoria, se tiene los siguientes requerimientos funcionales:

1. Indicar los sistemas más cercanos a la posición de la nave.

2. Calcular el orden en que se deben recorrer todos los sistemas desde la ubicación actual en el menor tiempo posible.

3. Calcular el recorrido que toma menos tiempo para viajar del sistema actual al sistema deseado.

4. Agregar un sistema planetario nuevo al programa y a su base de datos.

5. Eliminar un sistema planetario del software.

6. Implementar una pestaña donde se pueda buscar un sistema planetario para ver su información.

7. Calcular una ruta para recorrer la mayor cantidad de sistemas habitados en el menor tiempo posible partiendo de la ubicación actual de la nave.

8. (Opcional) Que cada campo de búsqueda despliegue una lista de autocompletar.

9. (Opcional) Que se calcule una ruta para recorrer los sistemas en el menor tiempo posible de acuerdo a los siguientes propósitos:

    a. Explorar los sistemas inhabitados

    b. Explorar las civilizaciones tipo 0 1 y 2 para la extracción y negociación de recursos.

c. Explorar las civilizaciones tipo III para entablar relaciones diplomáticas e intercambiar conocimiento y recursos.

Fase 2: Recopilación de la información necesaria

Para solucionar este problema tan complejo e importante, se necesita de una estructura eficiente que pueda almacenar grandes cantidades de datos, y al mismo tiempo tener una o varias relaciones entre ellos. Es por esa razón que, al analizar el problema y entenderlo de manera significativa, se logró encontrar una de las posibles soluciones al problema: los grafos. Ahora bien, al existir diferentes tipos de grafos se encontraron diferentes maneras de diseñarlos, por eso se detalla cada uno a continuación:

Grafo dirigido: Un grafo dirigido consta de un conjunto de vértices y aristas donde cada una se asocia de forma unidireccional a través de una flecha con otro. Las aristas, dependiendo de su salida o ingreso, reciben la calificación de entrante o saliente, la condición común es que siempre tienen un destino hacia un nodo.

Grafo no dirigido: Los grafos no dirigidos son aquellos que constan un conjunto de vértices que están conectados a un conjunto de aristas de forma no direccional. Esto significa que una arista puede indistintamente recorrer desde cualquiera de sus puntos y en cualquier dirección.

Además de los tipos, también se encuentran algunas propiedades interesantes: Una de las principales es que posee adyacencia. Ésta se trata de la relación entre dos aristas que comparten la conexión o relación con un vértice común. La incidencia se refiere simplemente al caso en el que un vértice está unido a otro. Por último, se deben entender las propiedades de ponderación de los grafos que corresponden a una función en la que cada arista es clasificada y cuantificada en diversos términos para aumentar la expresividad del modelo. Esta característica en especial es muy útil en estudios de optimización.

Tomando en cuenta la información introductoria y/o básica sobre la estructura de datos grafo, se pueden enumerar las posibles representaciones que se tienen de éste:

1. **Matrices de adyacencia:** Una de las maneras más fáciles de implementar un grafo es usar una matriz bidimensional. Aquí, cada una de las filas y columnas representa un vértice en el grafo. El valor que se almacena en la celda en la intersección de la fila V y la columna W indica si hay una arista desde el vértice v y el vértice w. Cuando dos vértices están conectados por una arista, decimos que son adyacentes.

2. **Matrices de peso:** Son similares a las matrices de adyacencia, donde en vez de un 1 para representar una arista entre dos nodos se agrega un número que representa el peso de cada arista. Este número es 0 u infinito si no hay ninguna relación directa entre dos nodos del grafo, y cualquier otro número cuando sí existe una relación directa.

3. **Matrices de incidencia:** Son matrices binarias que se utilizan como una forma de representar relaciones binarias. Aquí se toman en cuenta varios aspectos: Las columnas de la matriz representan las *aristas* del grafo. Las filas representan a los distintos nodos. Por cada nodo unido por una arista, se indica un uno (1) en el lugar correspondiente, y se llena el resto de las ubicaciones con ceros (0).

4. **Listas de adyacencia:** Una lista de adyacencia es una representación de todas las aristas o arcos de un grafo mediante una lista. Si el grafo es no dirigido, cada entrada es un conjunto o multiconjunto de dos vértices conteniendo los dos extremos de la arista correspondiente. Si el grafo es dirigido, cada entrada es una tupla de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente.

5. **Representación por medio de relaciones:** La posibilidad de representar de forma gráfica un conjunto de vértices y aristas es una solución importante en términos de análisis. La visualización de grafos no debe ser confundida con la representación del grafo en sí, sino del conjunto complejo que resulta de la posición de los vértices y la orientación de las aristas. Con esto se busca darle una imagen mental al conjunto de datos y su incorrecta representación puede afectar la comprensión, usabilidad y costo de la información.

6. **Listas de aristas o listas de incidencia:** Es una manera relativamente sencilla de almacenar un grafo en donde se tiene un arreglo con un tamaño equivalente al número de aristas. Éste almacena pares ordenados o grupos de dos nodos representando una arista. En un grafo dirigido la primera posición de cada par representa el nodo de salida y en la segunda el nodo de llegada. En un grafo no dirigido no hay distinción de orden.

Además de la información básica y las representaciones que se tuvieron en cuenta, es importante destacar las opciones de autocompletado que en transcurso del tiempo se han investigado e implementado:

1. **ControlsFx:** Librería que da ciertas herramientas útiles para complementar JavaFX. Entre sus herramientas se encuentra la clase TextFields que permite enlazar un campo de texto TextField con una lista de autocompletar. La librería de ControlsFx se encarga de mostrar el listado de predicciones cada vez que se escriba que se actualice el campo de texto.

2. **Trie:** Para obtener las palabras que se muestran en el autocompletar esta estructura es de bastante utilidad. Se encarga de almacenar cada palabra que se agregue en nodos que representan cada carácter de la palabra. Es muy eficiente también para la búsqueda de palabras debido a que la complejidad temporal depende únicamente del largo de la palabra a buscar.

Por último, se tiene como informacion adicional algunos algoritmos que se desean o se deben implementar en la estructura del grafo:

1. **BFS:** Es un algoritmo de búsqueda utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

2. **DFS:** Es un algoritmo de búsqueda utilizado para recorrer todos los nodos de un grafo de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

3. **Dijkstra:** La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajaría el costo general del camino al pasar por una arista con costo negativo).

4. **Floyd-Warshall:** El algoritmo de Floyd-Warshall compara todos los posibles caminos a través del grafo entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo V^3 comparaciones (esto es notable considerando que puede haber hasta V^2 aristas en el grafo, y que cada combinación de aristas se prueba). Lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima.

5. **Prim:** es un algoritmo perteneciente a la teoría de grafos para encontrar un un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas. En otras palabras, 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.

6. **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 artistas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de

todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo.

Complementando toda la información, también se tiene los requerimientos no funcionales:

1. Modelar e implementar el sistema de navegación en dos grafos diferentes.

2. Implementar la interfaz gráfica con JavaFx.

3. Contar con un menú superior en la ventana, el cual permite cambiar todos los elementos cuando se esté trabajando en opciones diferentes.

4. Implementar los algoritmos de recorridos sobre grafos BFS y DFS.

5. Codificar los algoritmos de camino de peso mínimo Dijkstra y Floyd-Warshall.

6. Desarrollar los algoritmos de árbol de recubrimiento mínimo Prim y Kruskal.

7. Cargar los nombres de los sistemas planetarios desde un archivo plano.

8. Generar sistemas a partir de un listado de nombres posibles.

Fase 3: Búsqueda de soluciones creativas

Para llevar a cabo esta etapa, se formularon tres procesos:

- Analizar los requerimientos indicados para el proyecto para poder diseñar un problema cuya solución pueda plantearse en términos de lo solicitado en el enunciado.

- Realizar una lluvia de ideas entre los integrantes del equipo para determinar el tema del proyecto cuya solución será modelada como un software.

- Definir los atributos y exigencias propias del problema elegido para ser desarrollado.

A diferencia de los proyectos trabajados anteriormente, en este proyecto no se realizó ninguna lluvia de ideas para elegir las estructuras con las que se podría resolver mejor el problema, puesto que uno de los requerimientos del proyecto es que sea modelado en grafos. Sin embargo, sí se realizó un análisis para la elección del tipo de grafo preferible para la cuestión a resolver, ya que entre las opciones estaban grafos dirigidos y no dirigidos, y cada uno de estos podían o no presentar aristas que tuvieran peso. Ahora bien, para las dos representaciones de grafos en las que se implementará el problema se tuvieron las ya mencionadas:

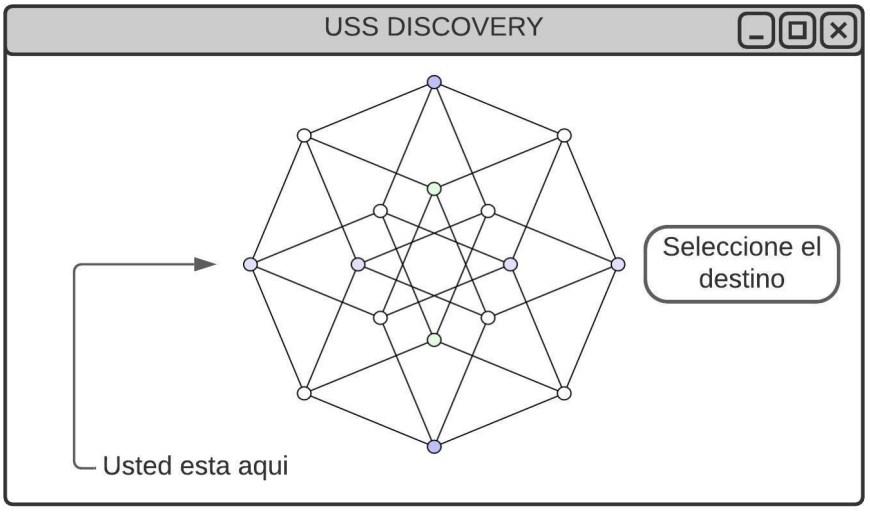
- Matrices de peso
- Matrices de incidencia
- Matrices de adyacencia
- Listas de adyacencia
- Representación por medio de relaciones (similar a un árbol)
- Listas de aristas o listas de incidencia

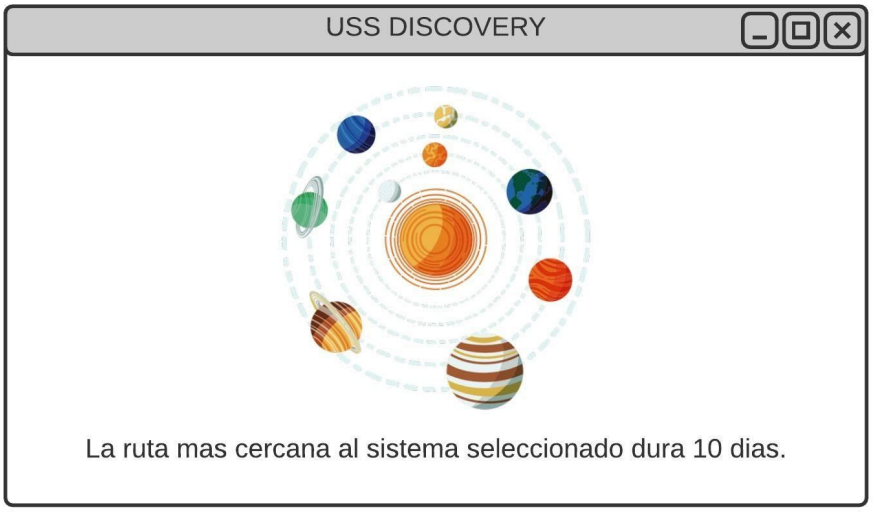
También se plantean como ideas el uso de todos los algoritmos planteados en la fase dos:

1. BFS
2. DFS
3. Dijkstra
4. Floyd-Warshall
5. Prim
6. Kruskal

Teniendo en cuenta todo lo mencionado anteriormente, se planeó la interfaz de usuario usando la herramienta online Lucidchart. Como resultado, se consiguieron los siguientes diseños básicos:


USS DISCOVERY
Bienvenido al sistema
de navegacion


USS DISCOVERY
Seleccione el
destino
Usted esta aqui



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

La primera decisión que se toma para la implementación de la solución de software es descartar el uso de grafos dirigidos, pues para la nave es posible navegar de un sistema A a un sistema B de la misma manera en que navegaría del sistema B al sistema A.

La segunda opción que se descarta es la representación a través de grafos no ponderados, ya que para el problema es clave que se tengan en cuenta los pesos de las aristas entre los nodos del grafo, pues estos representan el tiempo de viaje entre sistemas.

Con lo anterior se tiene entonces que los grafos con los que es posible modelar el problema son aquellos que permiten que sus aristas no sean dirigidas, y que además tengan pesos asociados. Por esto último, también se descarta el uso de matrices de adyacencia, resultando esto en las siguientes opciones viables:

- Usar una matriz de peso, ya que sus filas y columnas podrían representar los sistemas de la Vía Láctea, y las celdas de la matriz simularán el tiempo de viaje entre cada uno de estos sistemas.

- La lista de aristas también puede ser de utilidad ya que a cada par de nodos se le puede agregar un tercer campo que represente el tiempo de viaje entre dos sistemas. De igual forma puede crearse un objeto nodo que guarde dicha información.

- La matriz de incidencia puede llegar a resultar igualmente útil, ya que dentro de ésta sucede una situación importante: cada nodo que incide sobre una arista se puede tener un número que indique el tiempo de viaje entre los sistemas que inciden sobre una determinada arista, en vez de un 1 como indicador. También puede tenerse una lista de aristas para guardar la información del tiempo, donde la arista en la posición 1 representa la columna 1 de la matriz.

- La representación por medio de relaciones puede mantener en cada relación de los nodos a otros nodos el tiempo de viaje como un atributo adicional o la relación a un objeto arista que se encarga de almacenar el tiempo y conectar los nodos.

- La lista de adyacencia con pesos se adapta la representación original para incluir un par vértice y peso de la arista.

Ahora bien, también es importante analizar cuáles algoritmos son factibles a la hora de solucionar el problema. Esto teniendo en cuenta la restricción de que tiene que haber al menos ⅔ de grupos utilizados (búsqueda, caminos mínimos o árboles de recubrimiento mínimo) aunque todos los algoritmos deban estar implementados.

- Dijkstra (sistemas más cercanos y cálculo de la mejor ruta)
- BFS (búsqueda de un sistema específico)
- DFS (Búsqueda normal)

Fase 5: Selección y evaluación de la mejor solución

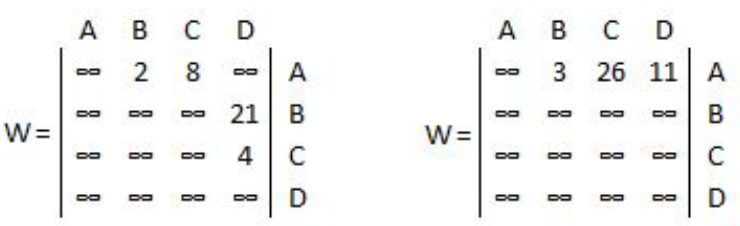
Al tener esta problemática tan importante, se deben de utilizar estructuras que sean totalmente eficaces, y que al mismo tiempo no sean tan complejas de implementar. Es por esa razón, que la resolución más emblemática que se encontró fueron los grafos.
Para seleccionar, evaluar y desarrollar las mejores soluciones, en una tabla que se muestra a continuación se muestra cada una de las representaciones posibles de grafos, junto a las complejidades, desventajas y ventajas respectivas:

Representación	Complejidad de implementación (1-5) Muy fácil a muy difícil	Desventajas	Ventajas
Matriz de pesos	3	Ocupa el máximo espacio posible sin importar la cantidad de aristas, se requiere una lista de vértices aparte.	Uso amplio; no requiere lista de aristas; fácil adaptación a algoritmos.
Listas de adyacencia	3	Cada vértice puede aparecer más de una vez; cuando se tienen muchas aristas ocupa más espacio que la matriz de pesos; la complejidad temporal aumenta	Se tiene una sola lista con vértices y pesos de las aristas; adaptación inmediata a la mayoría de algoritmos.

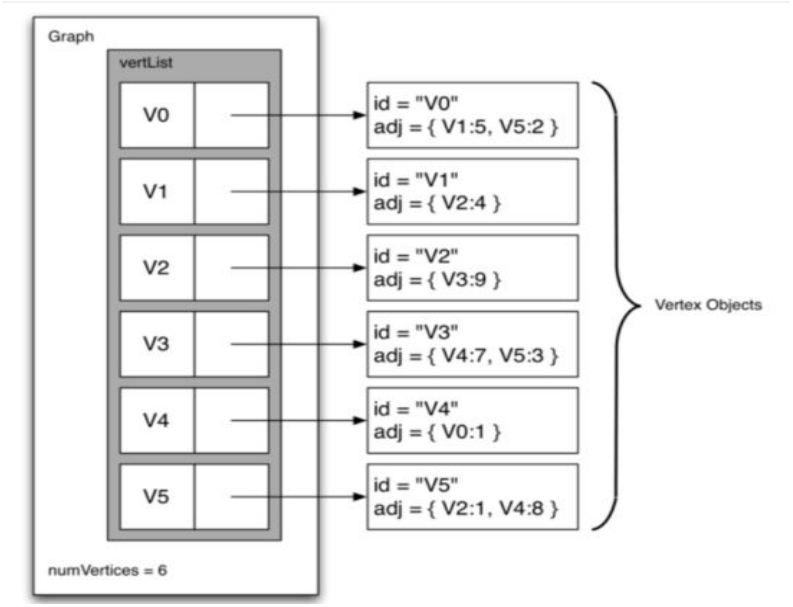
		drásticamente en grafos densos.	
Listas de incidencia	4	Se tarda en encontrar el vértice correspondiente a una arista. La lista crece mucho en tamaño (grafos densos).	Una sola lista con aristas; implementación más sencilla de un grafo.
Matriz de incidencia	4	El peso de la arista es duplicado; se requiere una lista de vértices y posiblemente una de aristas.	-
Por relaciones	5	Puede ser complejo de adaptar a los algoritmos; no es tan sencillo y eficiente el añadir vértices y aristas; se debe tener un arreglo de vértices ya que no hay una raíz.	Representación más intuitiva.

Pero, como se desea desarrollar un software de calidad, y sin ningún tipo de error o inconveniente secundario, con la implementación de grafos se piensa representarlos de dos maneras:

1. Matriz de pesos: En la siguiente imagen se muestra una imagen asociada al dato anteriormente mencionado.



2. Lista de adyacencia: En la siguiente imagen se muestra una imagen asociada al dato anteriormente mencionado.



Fase 6: Preparación de informes y especificaciones

- Diseño de casos de pruebas

Collections

MatrixGraph

Name	Class	Scenario
setUp1()	TestMatrixGraph	Creates two empty weighted undirected graphs.
setUp2()	TestMatrixGraph	Creates two weighted undirected graph with the following vertices: 1,2,3,4,5
setUp3()	TestMatrixGraph	Creates a weighted undirected graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)
setUp4()	TestMatrixGraph	Creates an empty weighted directed graph
setUp5()	TestMatrixGraph	Creates two weighted directed

		graph with the following vertices: 1,2,3,4,5
setUp6()	TestMatrixGraph	Creates a weighted directed graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)

Class	Method	Setup	Input	Output
TestGraph	testAddVertex1()	setup1()	the following vertices: 1,2,3,4,5	True, all the elements have been added correctly.
TestGraph	testAddVertex2()	setup2()	x = 64	True, the element has been added correctly.
TestGraph	testAddVertex3()	setup1()	One thousand vertices with values from 0 to 1000.	True, all the elements have been added successfully.
TestGraph	testAddEdge1()	setup2()	v1 = 2, v2 = 15, w = 67	True, the element has been added to the graph.
TestGraph	testAddEdge2()	setup2()	v1 = 3, v2 = 16, w = 9	False because an edge between those 2 nodes already exists.
TestGraph	testAddEdge3()	setup3()	v1 = 13, v2 = 35	True, the element has been added to the graph.
TestGraph	testIsEmpty1()	setup1()	-	True, because there aren't elements in the graph.
TestGraph	testIsEmpty2()	setup2()	-	False, because there are elements.
TestGraph	testIsEmpty3()	setup3()	-	False,because there are elements in the graph.
TestGraph	testRemoveVertex1()	setup1()	v = 70	False, because the

				graph is empty.
TestGraph	testRemoveVertex2()	setup2()	v = 20	True. The element has been removed.
TestGraph	testRemoveVertex3()	setup3()	v = 1459	False, because the element does not exist.
TestGraph	testRemoveEdge1()	setup2()	v1 = 18, v2 = 11	True. The edge has been removed.
TestGraph	testRemoveEdge2()	setup2()	v1 = 7, v2 = 1	False, because there is not an edge between those two nodes.
TestGraph	testRemoveEdge3()	setup3()	v1 = 50, v3 = 700	False, because there is not an edge between those two nodes.
TestGraph	testGetAdjacencyListI1()	Setup1() Setup4()	None	Empty adjacency List
TestGraph	testGetAdjacencyListI2()	Setup2() Setup5()	None	ND and D 1 → empty 2 → empty 3 → empty 4 → empty 5 → empty
TestGraph	testGetAdjacencyListI3()	Setup3() Setup6()	None	ND 1 → 2(5) → 3(6) 2 → 1(5) → 3(8) → 5(9) 3 → 2(8) → 5(1) → 1(6) 4 → 5(4) 5 → 4(4) → 3(1) → 2(9) D 1 → 2(5) 2 → 3(8) 3 → 1(6) 4 → 5 → 4(4) → 3(1) → 2(9)
TestAdjacencyListGraph	testGetEdgeList1()	Setup1() Setup4()	(ND1) None (ND2) 1 (D1) None (D2) 1	Edge list is empty for all cases
TestAdjacencyListGraph	testGetEdgeList2()	Setup2() Setup5()	(ND1) None (ND2) 1	Edge list is empty for all cases

			(D1) None (D2) 1	
TestAdjacencyListGraph	testGetEdgeList3()	Setup3() Setup6()	(ND1) None (ND2) 1 (D1) None (D2) 1	(NDI) A list with the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6) (NDII) A list with the following edges 1-2(5), 3-1(6) (DI) A list with the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6) (II) A list with the following edges 1-2(5)
TestAdjacencyListGraph	testGetAdjacentVertices1()	Setup1() Setup4()	(ND and D)The following vertex: 1	Empty vertices list
TestAdjacencyListGraph	testGetAdjacentVertices2()	Setup2() Setup5()	(ND and D) The following vertex: 1	Empty vertices list
TestAdjacencyListGraph	testGetAdjacentVertices3()	Setup3() Setup6()	(ND1) 1 (ND2) 99 (D1) 1 (D2) 99	(NDI) A list with the following vertices: 2, 3 (NDII) Null vertex list (DI) A list with the following vertices: 2 (II) Null vertex list

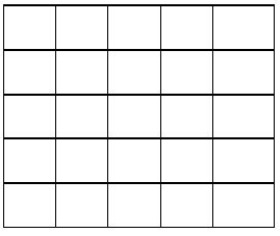
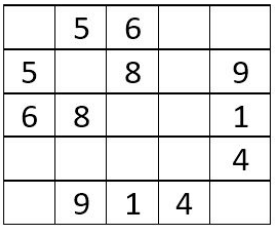
Collections

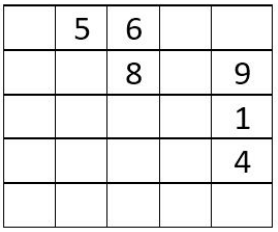
AdjacencyListGraph

Name	Class	Scenario
setUp1()	TestMatrixGraph	Creates two empty weighted undirected graphs.
setUp2()	TestMatrixGraph	Creates two weighted undirected graph with the following vertices: 1,2,3,4,5
setUp3()	TestMatrixGraph	Creates a weighted undirected graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)
setUp4()	TestMatrixGraph	Creates an empty weighted directed graph
setUp5()	TestMatrixGraph	Creates two weighted directed graph with the following vertices: 1,2,3,4,5

setUp6()	TestMatrixGraph	Creates a weighted directed graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)

Class	Method	Setup	Input	Output
TestAdjacencyListGraph	testAddVertex1()	Setup1()	x1 = 10 v2 = 0 x2 = 20 v2 = 1	True, both elements were added to the system.
TestAdjacencyListGraph	testAddVertex2()	Setup2()	Ten elements with different information.	Equals, the weight of the graph is correct.
TestAdjacencyListGraph	testAddVertex3()	Setup3()	One thousand elements with the same information, but with different vertex.	False, the correct weight is one thousand, no two thousand.
TestAdjacencyListGraph	testAddEgde1()	Setup1()	v1 = 5 v2 = 10 w = 500	True, the element was added to the system.
TestAdjacencyListGraph	testAddEgde2()	Setup2()	Ten elements with different information.	Equals, the weight of the graph is correct.
TestAdjacencyListGraph	testAddEgde3()	Setup3()	One thousand elements with the same information, but with different vertex.	False, the correct weight is one thousand, no two thousand.
TestAdjacencyListGraph	testRemoveVertex1()	Setup1()	v1 = 5 v2 = 10	True, all the elements were removed to the system.

TestAdjacencyListGraph	testRemoveVertex2()	Setup2()	v1 = 2	False, the element doesn't exist in the system.
TestAdjacencyListGraph	testRemoveVertex3()	Setup3()	v1 = 25 v2 = 75	True and false: the first element was removed, but the other not.
TestAdjacencyListGraph	testRemoveEdge1()	Setup1()	v1 = 1 v2 = 2 v1 = 5 v2 = 10	True, both elements were removed from the system.
TestAdjacencyListGraph	testRemoveEdge2()	Setup2()	v1 = 152 v2 = 241	False, the element doesn't exist in the system.
TestAdjacencyListGraph	testRemoveEdge3()	Setup3()	v1 = 10 v2 = 20 v1 = 50 v2 = 70	True and false: the first element was removed, but the other not.
TestAdjacencyListGraph	testGetWeightMatrix1()	Setup1()	None	Empty weight matrix
TestAdjacencyListGraph	testGetWeightMatrix2()	Setup2()	None	ND and D 
TestAdjacencyListGraph	testGetWeightMatrix3()	Setup3()	None	ND  D

				
TestAdjacencyListGraph	testGetEdgeList1()	Setup1() Setup4()	(ND1) None (ND2) 1 (D1) None (D2) 1	Edge list is empty for all cases
TestAdjacencyListGraph	testGetEdgeList2()	Setup2() Setup5()	(ND1) None (ND2) 1 (D1) None (D2) 1	Edge list is empty for all cases
TestAdjacencyListGraph	testGetEdgeList3()	Setup3() Setup6()	(ND1) None (ND2) 1 (D1) None (D2) 1	(NDI) A list with the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6) (NDII) A list with the following edges 1-2(5), 3-1(6) (DI) A list with the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6) (II) A list with the following edges 1-2(5)
TestAdjacencyListGraph	testGetAdjacentVertices1()	Setup1() Setup4()	(ND and D)The following vertex: 1	Empty vertices list
TestAdjacencyListGraph	testGetAdjacentVertices2()	Setup2() Setup5()	(ND and D) The following vertex: 1	Empty vertices list

TestAdjacencyListGraph	testGetAdjacentVertices3()	Setup3() Setup6()	(ND1) 1 (ND2) 99 (D1) 1 (D2) 99	(NDI) A list with the following vertices: 2, 3 (NDII) Null vertex list (DI) A list with the following vertices: 2 (II) Null vertex list

Collections

Graph Algorithms

Name	Class	Scenario
setUp1()	TestMatrixGraph	Creates two empty weighted undirected graphs.
setUp2()	TestMatrixGraph	Creates two weighted undirected graph with the following vertices: 1,2,3,4,5
setUp3()	TestMatrixGraph	Creates a weighted undirected graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)
setUp4()	TestMatrixGraph	Creates an empty weighted directed graph
setUp5()	TestMatrixGraph	Creates two weighted directed graph with the following vertices: 1,2,3,4,5

setUp6()	TestMatrixGraph	Creates a weighted directed graph with the following vertices: 1,2,3,4,5 And the following edges 1-2(5) 5-4(4) 2-3(8) 5-3(1) 5-2(9) 3-1(6)

Class	Method	Setup	Input	Output
GraphAlgorithms	BFS()	setup1() setup4()	graph	Empty traversal list
GraphAlgorithms	BFS()	setup2() setup5()	graph	Empty traversal list
GraphAlgorithms	BFS()	setup3() setup6()	graph	null
GraphAlgorithms	DFS()	setup1() setup4()	graph	null
GraphAlgorithms	DFS()	setup2() Setup5()	graph	null
GraphAlgorithms	DFS()	setup3() setup6()	graph	null
GraphAlgorithms	dijkstra()	setup1() setup4()	graph, 1	Empty distances list

GraphAlgorithms	dijkstra()	setup2() Setup5()	graph, 1	Distances list filled with infinity
GraphAlgorithms	dijkstra()	setup3() setup6()	graph, 1	
GraphAlgorithms	floydWarshall()	setup1() setup4()	graph	
GraphAlgorithms	floydWarshall()	setup2() Setup5()	graph	
GraphAlgorithms	floydWarshall()	setup3() setup6()	graph	
GraphAlgorithms	prim()	setup1() setup4()	graph	
GraphAlgorithms	prim()	setup2() Setup5()	graph	
GraphAlgorithms	prim()	setup3() setup6()	graph	
GraphAlgorithms	kruskal()	setup1() setup4()	graph	

GraphAlgorithms	kruskal()	setup2() Setup5()	graph	
GraphAlgorithms	kruskal()	setup3() setup6()	graph	

- Diagrama de clases
- Diagrama de pruebas

Fase 7: Implementación del diseño

Ver el repositorio en GitHub.