1. **Identificación del problema**

**Contexto Problemático**

El bombardeo aéreo es un ataque de artillería contra instalaciones fortificadas, tropas, ciudades o edificios. Estrictamente, el término bombardeo se usa para referirse a objetivos indefensos con el propósito de desmoralizar a su oponente, especialmente a la población civil y a las autoridades, para conseguir de ese modo la rendición antes de que la totalidad de los edificios sea destruido. Esta práctica fue especialmente común durante el siglo XIX y el siglo XX, especialmente durante la Primera y Segunda Guerra Mundial es por esto que una milicia de la unión europea ha decidido que como medio de prevención, para estar capacitados ante otra guerra mundial, desarrollar un plan de bombardeo masivo efectivo con la meta de ahorrar tiempo y por ende combustible.

Por consiguiente, han listado a 24 ciudades que ven como potenciales objetivos, en caso de que estalle una hipotética guerra mundial, las ciudades son las siguientes: Washington, Kansas, Ciudad de México, Brasilia, Rio de Janeiro, Moscú, Krasnoyarsk, Pekín, Hong Kong, Camberra, New Delhi, Teherán, Riad, Buenos Aires, Astana, Ulán Bator, Argel, Ciudad Del Cabo, Bogotá, Kinsasa, Ottawa, Alberta, Pretoria, Bloemfontein.

**Identificación del problema**

Una milicia de la unión europea quiere un simulador (programa) con las características de que dada una ciudad, la ciudad inicial donde se empezará el bombardeo, y una segunda ciudad, la cual será la última ciudad a bombardear, muestre el camino más eficiente para bombardear todas las ciudades que estén en la trayectoria de esas dos ciudades seleccionadas, también la forma de bombardear a todas las ciudades de la manera más eficiente posible. Entonces, de acuerdo a lo anterior, las especificaciones que debe cumplir el programa a desarrollar son:

1. Registrar inicio, el programa debe permitir la entrada del punto de partida desde el cual se hará el recorrido.
2. Registrar final, el programa debe permitir la entrada del punto terminal al cual se llegará mediante el recorrido.
3. Encontrar recorrido el programa debe estar en la capacidad de encontrar el recorrido más corto para llegar de un punto a un punto b (nodo inicial hasta nodo final).
4. Mostrar la ruta más eficiente para bombardear todas las ciudades.

**2. Recopilación de la información necesaria**

**Camino más corto:** es el problema que consiste en encontrar un camino entre dos vértices (o nodos) de tal manera que la suma de los pesos de las aristas que lo constituyen es mínima. Un ejemplo de esto es encontrar el camino más rápido para ir de una ciudad a otra en un mapa. En este caso, los vértices representarían las ciudades y las aristas las carreteras que las unen, cuya ponderación viene dada por el tiempo que se emplea en atravesarlas.

**Grafo:** Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.

**Grafo simple:** Un grafo es simple si a lo sumo existe una arista uniendo dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos. Grafos conexos: Un grafo es conexo si cada par de vértices está conectado por un camino; es decir, si para cualquier par de vértices (a, b), existe al menos un camino posible desde a hacia b. Grafo no dirigido: Un grafo no dirigido es un grafo en el que todas las aristas son bidireccionales, es decir, las aristas no apuntan en una dirección específica.

**Grafo dirigido:** Un grafo dirigido es un grafo en el que todas las aristas son unidireccionales, es decir, las aristas apuntan en una sola dirección. Grafo ponderado: En un grafo ponderado, a cada arista se le asigna un peso o un costo.

**Lista 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 (lista ordenada de elementos) de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente. Matriz de adyacencia: Es una matriz cuadrada de orden NxN asociada a un grafo de orden N, donde sus filas y columnas se identifican con los vértices del grafo y en las celdas se indican la cantidad de aristas a los nodos asignado a la fila y columnas en cuestión.

**Multígrafo:** Es un grafo que está facultado para tener aristas múltiples; es decir, aristas que relacionan los mismos nodos. De esta forma, dos nodos pueden estar conectados por más de una arista. Los multígrafos podrían usarse, por ejemplo, para modelar las posibles conexiones de vuelo ofrecidas por una aerolínea. Para este caso tendríamos un grafo dirigido, donde cada nodo es una localidad y donde pares de aristas paralelas conectan estas localidades, según un vuelo es hacia o desde una localidad a la otra.

**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. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959. En múltiples aplicaciones donde se aplican los grafos, es necesario conocer el camino de menor costo entre dos vértices dados: • Distribución de productos a una red de establecimientos comerciales. • Distribución de correos postales.

• Sea G = (V, A) un grafo dirigido ponderado. El problema del camino más corto de un vértice a otro consiste en determinar el camino de menor costo, desde un vértice u a otro vértice v. El costo de un camino es la suma de los costos (pesos) de los arcos que lo conforman.

**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).

• Funcionamiento del algoritmo de Kruskal:

1. Se selecciona, de entre todas las aristas restantes, la de menor peso siempre que no cree ningún ciclo.

2. Se repite el paso 1 hasta que se hayan seleccionado |V| - 1 aristas. Siendo V el número de vértices.

**Algoritmo de prim:** El algoritmo de Prim, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de expansión mínima. Es decir, es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

• Funcionamiento del algoritmo de Prim:

1. Se marca un vértice cualquiera. Será el vértice de partida.

2. Se selecciona la arista de menor peso incidente en el vértice seleccionado anteriormente y se selecciona el otro vértice en el que incide dicha arista.

3. Repetir el paso 2 siempre que la arista elegida enlace un vértice seleccionado y otro que no lo esté. Es decir, siempre que la arista elegida no cree ningún ciclo.

4. El árbol de expansión mínima será encontrado cuando hayan sido seleccionados todos los vértices del grafo.

**Depth first search (dfs):** Es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) 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 (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

**Breadth first search (bfs):** Es un algoritmo de búsqueda no informada 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 árbol.

**Floyd-warshall:** Es un algoritmo de análisis sobre grafos que permite encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución, constituyendo un ejemplo de programación dinámica.

• Características:

1. Obtiene la mejor ruta entre todo par de nodos.

2. Trabaja con la matriz D inicializada con las distancias directas entre todo par de nodos.

3. La iteración se produce sobre nodos intermedios, o sea para todo elemento de la matriz se prueba si lo mejor para ir de i a j es a través de un nodo intermedio elegido o como estaba anteriormente, y esto se prueba con todos los nodos de la red. Una vez probados todos los nodos de la red como nodos intermedios, la matriz resultante da la mejor distancia entre todo par de nodos.

4. El algoritmo da sólo la menor distancia; se debe manejar información adicional para encontrar tablas de encaminamiento. 5. Hasta no hallar la última matriz no se encuentran las distancias mínimas. 6. Su complejidad es del orden de n3.

# **Referencias**

[https://es.wikipedia.org/wiki/Problema\_del\_camino\_m%C3%A1s\_corto](https://es.wikipedia.org/wiki/Problema_del_camino_m%C3%A1s_corto%20) [https://brilliant.org/wiki/dijkstras-short-path-finder](https://brilliant.org/wiki/dijkstras-short-path-finder%20)

[https://es.wikipedia.org/wiki/Grafo](https://es.wikipedia.org/wiki/Grafo%20)

[https://es.wikipedia.org/wiki/Teor%C3%ADa\_de\_grafos#Grafo\_simple](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos%23Grafo_simple%20)

[https://es.wikipedia.org/wiki/Teor%C3%ADa\_de\_grafos#Grafos\_conexos](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos%23Grafos_conexos)

<https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/tutorial/>[https://es.wikipedia.org/wiki/Lista\_de\_adyacencia](https://es.wikipedia.org/wiki/Lista_de_adyacencia%20)

[https://www.ecured.cu/Matriz\_de\_adyacencia](https://www.ecured.cu/Matriz_de_adyacencia%20)

[https://es.wikipedia.org/wiki/Multigrafo](https://es.wikipedia.org/wiki/Multigrafo%20)

<https://www.ecured.cu/Algoritmo_de_Dijkstra>

[https://www.ecured.cu/Algoritmo\_de\_Kruskal](https://www.ecured.cu/Algoritmo_de_Kruskal%20)

[https://sites.google.com/site/complejidadalgoritmicaes/prim](https://sites.google.com/site/complejidadalgoritmicaes/prim%20)

<https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura>

[https://es.wikipedia.org/wiki/B%C3%BAsqueda\_en\_profundidad](https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad%20)

[https://www.ecured.cu/Floyd-Warshall](https://www.ecured.cu/Floyd-Warshall%20)

**3. Búsqueda de soluciones creativas**

Para la búsqueda de soluciones creativas se realiza una lluvia de ideas donde cada uno de los integrantes planteamos ideas sobre cómo podría ser el funcionamiento del software, la interfaz, los algoritmos y que estructuras de datos se podrían usar durante el desarrollo.

**Alternativa 1:** Cola de prioridad

A través de las rutas generadas por el grafo, se agregan a una cola de prioridad donde su factor de orden es el peso menor de la primera arista, es decir, el lugar más cercano a la ciudad inicial o de origen.

**Alternativa 2:** Lista enlazada

Al generarse las rutas para las ciudades, estas se agregan a una lista enlazada donde el peso menor de la primera arista tendrá relación con la segunda con peso menor, la segunda con la tercera y así sucesivamente.

**Alternativa 3:** Montículo o heap

Las rutas generadas se guardarán en un montículo o heap, con la idea de que se obtengan las rutas más cercanas a la ciudad inicial o de origen en O(1).

**Alternativa 4:** Dijsktra

La idea consiste en que se va a generar la ruta más corta o con menor peso desde la ciudad inicial hasta la ciudad terminal al cual se llegará mediante el recorrido. Es decir, se va a encontrar la mejor ruta entre todas las posibles.

**Alternativa 5:** Prim

Se encuentra un árbol de expansión mínima. Es decir, se encuentra un subconjunto de las aristas que formen un árbol que incluya todas las rutas de la ciudad inicial a las demás ciudades, donde el peso total de las aristas del árbol es el mínimo posible.

**Alternativa 6:** Kruskal

Se busca un subconjunto de aristas que, formando un árbol, incluyen todas las ciudades, incluyendo la ciudad inicial, donde el valor total de todas las aristas del árbol es el mínimo posible.

**Alternativa 7:** Floyd Warshall

La idea se basa en que se conocerá todas las rutas desde la ciudad inicial hasta las ciudades, y de las ciudades a otras ciudades y así saber el camino mínimo entre todos los vértices del grafo.

**Alternativa 8:** BFS

Al generarse las rutas de las ciudades a bombardear, se conocerán todos los “vecinos” que van desde la ciudad inicial hasta la última ciudad.

**4. Diseños preliminares**

**Ideas no factibles**

|  |  |
| --- | --- |
| **Alternativa 1** | Esta alternativa se descartó debido a que es posible que los caminos estén organizados de una forma que los caminos no estén de manera óptima. |
| **Alternativa 2** | Aunque las aristas estén organizadas por su peso de menor a mayor, esto no garantiza que el camino sea el más corto para bombardear todas las ciudades que pasen del por el punto de partida y el último punto. |
| **Alternativa 3** | Esta alternativa se descartó porque aunque se obtenga las rutas más cercanas a la ciudad inicial en O(1) esto presentará un problema para las ciudades que no estén cerca de ella porque la idea es estar conectado con todas las ciudades. |
| **Alternativa 8** | Esta alternativa se descartó pues aunque pase por todas las ciudades que se van a bombardear no garantiza que sea con el menor costo posible. |

**5. Evaluación y selección de la mejor solución.**

·  **Criterios**

* **Criterio A:** **Representación fiel al contexto del problema.**

Este criterio se basa en que tan representativa es la alternativa al contexto del problema.

o Excelente: 4 puntos

o Fiel: 3 puntos

o Regular: 2 puntos

o Mala: 1 punto

* **Criterio B: Complejidad Temporal**

Este criterio se basa en que tan eficiente es en tiempo la alternativa para la solución del problema.

o Constante: 6 puntos

o Logarítmica: 5 puntos

o Lineal: 4 puntos

o Polinomial: 3 puntos

o Exponencial: 2 puntos

o Factorial: 1 punto

* **Criterio C:** **Complejidad Espacial**

Este criterio se basa en que tan eficiente en espacio de memoria es la alternativa para la solución del problema.

o Constante: 6 puntos

o Logarítmica: 5 puntos

o Lineal: 4 puntos

o Polinomial: 3 puntos

o Exponencial: 2 puntos

o Factorial: 1 punto

**Evaluación:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Alternativa** | **Criterio A** | **Criterio B** | **Criterio C** | **Total** |
| **4** | **4** | **5** | **4** | **13** |
| **7** | **3** | **3** | **3** | **9** |
| **5** | **4** | **5** | **4** | **13** |
| **6** | **3** | **5** | **4** | **12** |

Con base en los resultados obtenidos la alternativa 6 y 7 van a ser descarta y por lo tanto la alternativa 4 será la utilizada para darle solución al problema, los requerimientos 1, 2 y 3, y la alternativa 4 será la utilizada para darle solución al requerimiento 4.

**6. Preparación de informes y especificaciones.**

* **TAD**

|  |  |
| --- | --- |
| TAD de Grafos | |
|  | |
| OPERACIONES: | |
| Creación del grafo | crearGrafo(grafo) → Grafo (Constructora) |
| Inserción de vértices | agregarVertice(grafo, referenciaVertice) → Grafo (Modificadora) |
| Eliminación de vértices | eliminarVertice(grafo, referenciaVertice) → --- (Destructora) |
| Inserción de aristas | agregarArista(grafo,verticeUno,verticeDos) →Grafo (Modificadora) |
| Eliminación de aristas | eliminarArista(grafo, arista) → --- (Destructora) |
| Adyacencia | esAdyacente(Grafo,VerticeUno,VerticeDos)→boolean(Consultora) |

|  |
| --- |
| crearGrafo(): “Crea un nuevo grafo” |
| { pre: }    {post: Grafo=  } |

|  |
| --- |
| agregarVertice(Grafo, referenciaVertice): “Se agrega un vértice al grafo anteriormente creado” |
| {pre: El grafo debe estar creado, n }    {post: Grafo=  } |

|  |
| --- |
| eliminarvertice(Grafo,referenciaVertice): “Se elimina un vértice del grafo” |
| {pre: la referencia del vértice a eliminar se encuentra dentro del grafo(n)    {post:  } |

|  |
| --- |
| agregarArista(Grafo, verticeUno, verticeDos):”Se agrega una arista entre dos vértices” |
| {pre: El grafo debe de estar creado, 7,n}    {post:  } |

|  |
| --- |
| eliminarArista(Grafo, arista):”Se elimina una arista” |
| {pre: la arista a eliminar debe existir dentro del grafo}    {post:  } |

|  |
| --- |
| esAdyacente():”Permite saber si un grafo es adyacente” |
| {pre: El árbol debe de estar creado }  {post:booleano, que es True si el grafo es adyacente y False en caso contrario} |

|  |  |
| --- | --- |
| TAD de Vértice | |
|  | |
| OPERACIONES: | |
| Dar Valor | darValor()→Cadena de Texto (Consultora) |
| Dar Arista | darArista()→Arista (Consultora) |

|  |
| --- |
| darValor():”Da el valor del vertice” |
| {pre: Grafo debe estar creado, y el vértice debe estar presente dentro del grafo}    {post: Valor del vértice, el cual es una cadena de texto} |

|  |
| --- |
| darArista():”Da la arista perteneciente a un vertice” |
| {pre: Grafo debe estar creado, el vértice y la arista deben estar presentes dentro del grafo}    {post: Arista } |

|  |  |
| --- | --- |
| TAD de Arista | |
|  | |
| OPERACIONES: | |
| Es Dirigible | esDirigible()→Boolean(Consultora) |
| Dar Valor | darValor()→Cadena de Texto(Consultora) |
| Dar Costo | darCosto()→Double(Consultora) |
| Dar Origen | darOrigen()→Vertice(Consultora) |
| Dar Destino | darDestino()→Vertice(Consultora) |

|  |
| --- |
| esDirigible():”Nos permite saber si la arista seleccionada es dirigible o no” |
| {pre: Grafo debe estar creado, y la arista que se desea saber si es dirigible debe estar presente dentro del grafo}    {post: booleano que retorna True si la arista es dirigible y False en caso contrario} |

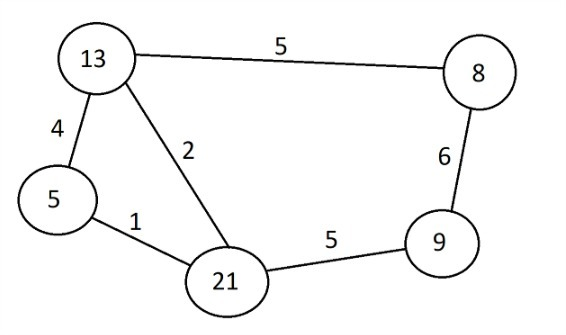
|  |
| --- |
| darValor():”Da el valor de la arista” |
| {pre: Grafo debe estar creado, y la arista debe estar presente dentro del grafo}    {post: Valor de la arista, el cual es una cadena de texto} |

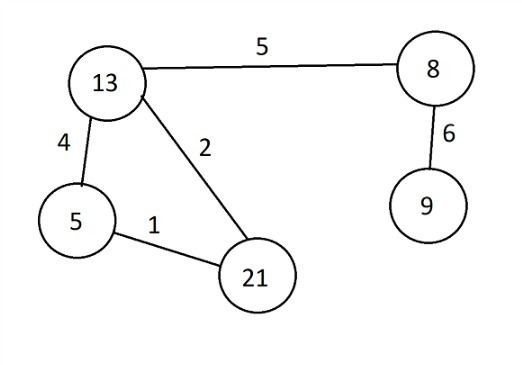
|  |
| --- |
| darCosto():”Da el costo de la arista” |
| {pre: Grafo debe estar creado, y la arista a la cual se le desea calcular el costo debe estar presente dentro del grafo}    {post: Costo de la arista, el cual es un valor de tipo double} |

|  |
| --- |
| darOrigen():”Da el vértice de origen de la arista” |
| {pre: Grafo debe estar creado, y la arista debe estar presente dentro del grafo}    {post: Vertice, el cual es el origen de dicha arista} |

|  |
| --- |
| darDestino():”Da el vértice de destino de la arista” |
| {pre: Grafo debe estar creado, y la arista debe estar presente dentro del grafo}    {post: Vértice que hace referencia al destino de la arista} |

* **Diseño de pruebas unitarias**





Los anteriores grafos fueron utilizados para las pruebas realizadas en cada uno de los métodos excepto en los métodos prim y kruskal, los cuales serán presentados más adelante .

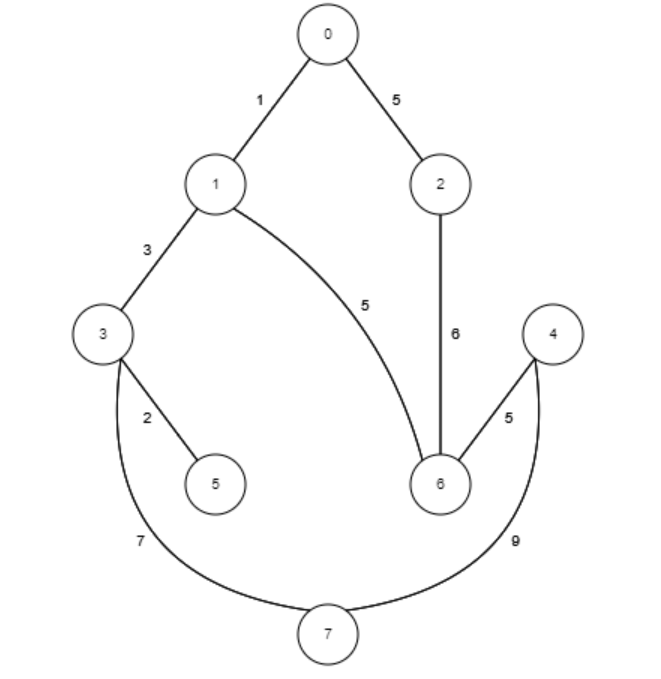
Los algoritmos de los grafos fueron evaluados tanto para los grafos por listas como para los grafos por matriz.

|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: BFS() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se crea un arraylist de vértices y se compara con el arraylist que retorna el método BFS | Grafo creado | Todos los vértices comparados se encuentran en la misma posición dentro del arraylist |

|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: DFS() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se crea un arraylist de vértices y se compara con el arraylist que retorna el método DFS | Grafo creado | Todos los vértices comparados se encuentran en la misma posición dentro del arraylist. |

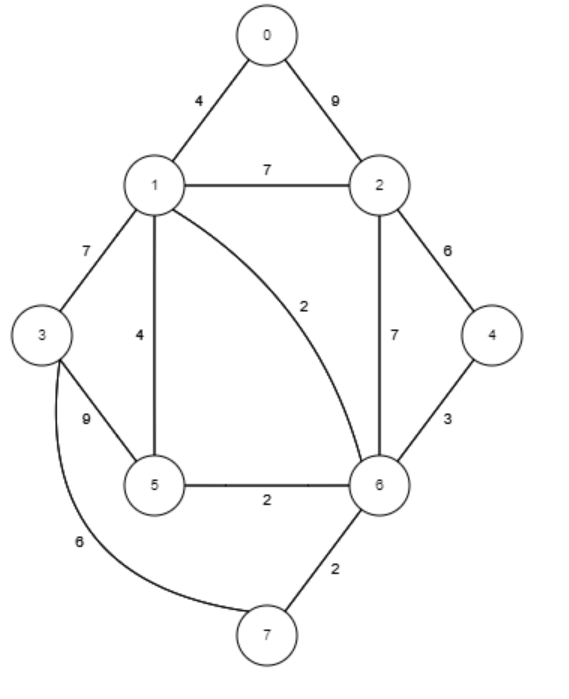
|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: Dijkstra() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se crea un arreglo de double que corresponde a la distancia mínima entre un vértice y otro y se compara con el arreglo de double que retorna el método Dijkstra | Grafo Creado | Todos las distancias comparadas se encuentran en la misma posición dentro del arreglo |

|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: floydMarshall() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se comprueba que todas las filas de la matriz floyd warshall, cada una corresponde al Dijkstra de cada vértice al que le corresponde cada fila | Grafo creado | Una matriz de floyd warshall que en sus filas, posee el menor recorrido del vértice al que le corresponde la fila a todos los demás vértices |



Este grafo se utilizó para realizar las pruebas del método prim, tanto para el grafo por matriz como para el grafo por lista

|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: prim() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se comprueba que el camino más corto es igual a lo que nos retorna el método prim, desde un vértice inicial. | Grafo creado | La prueba es correcta, puesto que al comparar ambos valores se comprueba que son iguales. |



Este grafo se utilizó para realizar las pruebas del método kruskal, tanto para el grafo por matriz como para el grafo por lista

|  |  |  |  |
| --- | --- | --- | --- |
| Clase: MethodsGraphs | | Método: prim() | |
| Caso # | Descripción de la prueba | Estado Inicial | Resultado |
| 1 | Se comprueba que el camino más corto es igual a lo que nos retorna el método kruskal. | Grafo creado | La prueba es correcta, puesto que al comparar ambos valores se comprueba que son iguales. |

* **Diagrama de clases de la solución.**

**7. Implementación de la solución.**

[**https://github.com/MrAlien98/AEDProject**](https://github.com/MrAlien98/AEDProject)