**METODO DE LA INGENIERIA**

**FASE 1: IDENTIFICACION DEL PROBLEMA**

**DESCRIPCION DEL CONTEXTO PROBLEMÁTICO**

Actualmente existen muchísimas estructuras de datos para resolver determinados problemas, el curso de algoritmos y estructuras de datos que se parte en la Universidad Icesi, ciertamente lleva a cabo la conceptualización y enseñanza de varias estructuras de datos a los estudiantes, tales como:

* **Tabla hash**
* **Colas de prioridad**
* **Montículos**
* **Arboles n-arios**
* **Arboles binarios**
* **Arboles rojo-negro**
* **Arboles balanceados**

El departamento de los tics, Emprendió una investigación hace 4 años para investigar cuál de los temas vistos en estructuras de datos era el que más causaba confusión a la hora de entender cómo se comportaba, esto se veía reflejado en las calificaciones del semestre. Se dieron cuenta que los dos temas que más causaban confusión eran:

* **Complejidad de los algoritmos.**
* **Grafos.**

**IDENTIFICACION DEL PROBLEMA**

La universidad desea que el porcentaje de ganancia de los parciales en los cuales entra uno de estos temas aumente de forma que valga la pena evidenciarse. Primeramente, se van a enfocar por solucionar la cuestión de los grafos. Por lo anterior la universidad requiere de un software educativo que, por el momento, muestre a sus estudiantes cómo se comportan los grafos, si hay éxito utilizando el software, se les implementaran a estas otras estructuras de datos, y el tema de complejidad algorítmico. La universidad contacto a la empresa **EZ-DEVELOPMENT** para el desarrollo del software que inicialmente cumpla con las siguientes funcionalidades:

* Representar gráficamente un grafo simple, un multígrafo, un pseudografo, y los dirigidos y no dirigidos.
* Mostrar el comportamiento de los métodos básicos de los grafos:
  + Recorrido en amplitud
  + Recorrido en profundidad
  + Agregar una arista entre dos vértices.

**REQUERIMIENTOS FUNCIONALES**

|  |  |
| --- | --- |
| Nombre. R1 | **Representación gráfica, grafo simple.** |
| Resumen | **Crear un grafo simple y mostrar representación gráfica de este.** |
| Entrada | **Ninguna.** |
| Resultado | **Se dibuja un grafo simple.** |

|  |  |
| --- | --- |
| Nombre. R2 | **Representación gráfica, multígrafo.** |
| Resumen | **Crear un multígrafo y mostrar representación gráfica de este.** |
| Entrada | **Ninguna.** |
| Resultado | **Se dibuja un multígrafo.** |

|  |  |
| --- | --- |
| Nombre. R3 | **Representación gráfica, pseudografo.** |
| Resumen | **Crear un pseudografo y mostrar representación gráfica de este.** |
| Entrada | **Ninguna.** |
| Resultado | **Se dibuja un pseudografo.** |

|  |  |
| --- | --- |
| Nombre. R4 | **Representación gráfica, Grafo dirigido.** |
| Resumen | **Crear un Grafo dirigido y mostrar representación gráfica de este.** |
| Entrada | **Ninguna.** |
| Resultado | **Se dibuja un grafo dirigido.** |

|  |  |
| --- | --- |
| Nombre. R5 | **Representación gráfica, multígrafo dirigido.** |
| Resumen | **Crear un multígrafo dirigido y mostrar representación gráfica de este.** |
| Entrada | **Ninguna.** |
| Resultado | **Se dibuja un multígrafo dirigido.** |

|  |  |
| --- | --- |
| Nombre. R6 | **Recorrido en amplitud** |
| Resumen | **Mostrar la animación del recorrido BFS (Búsqueda en amplitud)** |
| Entrada | **Ninguna.** |
| Resultado | **Se crea una animación mostrando el recorrido en amplitud.** |

|  |  |
| --- | --- |
| Nombre. R7 | **Recorrido en profundidad** |
| Resumen | **Mostrar la animación del recorrido DFS (Búsqueda en profundidad)** |
| Entrada | **Ninguna.** |
| Resultado | **Se crea una animación mostrando el recorrido en profundidad.** |

|  |  |
| --- | --- |
| Nombre. R8 | **Caminos de Peso Mínimo** |
| Resumen | **Mostrar la animación del recorrido realizado de los algoritmos de camino del peso mínimo (**Dijkstra, Floyd-Warshall) |
| Entrada | **Ninguna.** |
| Resultado | **Se crea una animación mostrando el recorrido de los algoritmos de camino de peso mínimo.** |

|  |  |
| --- | --- |
| Nombre. R9 | **Árbol de Recubrimiento Mínimo** |
| Resumen | **Mostrar la animación del recorrido realizado de los algoritmos de Árbol de recubrimiento Minino** (Prim, Kruskal) |
| Entrada | **Ninguna.** |
| Resultado | **Se crea una animación mostrando el recorrido de los algoritmos de Árbol de recubrimiento mínimo** |

|  |  |
| --- | --- |
| Nombre. 10 | **Agregar un nodo** |
| Resumen | **Permitir agregar un nuevo nodo al grafo.** |
| Entrada | **Ninguna.** |
| Resultado | **Agregar un nuevo nodo al grafo.** |

|  |  |
| --- | --- |
| Nombre. R11 | **Eliminar un nodo** |
| Resumen | **Permitir eliminar un nodo del grafo.** |
| Entrada | **Ninguna.** |
| Resultado | **Eliminar un nodo del grafo.** |

|  |  |
| --- | --- |
| Nombre. R12 | **Agregar una arista** |
| Resumen | **Permitir agregar una nueva arista entre nodos.** |
| Entrada | **Ninguna.** |
| Resultado | **Agregar una nueva arista entre dos nodos.** |

|  |  |
| --- | --- |
| Nombre. R13 | **Eliminar una arista** |
| Resumen | **Permitir eliminar una arista existente.** |
| Entrada | **Ninguna.** |
| Resultado | **Eliminar arista entre nodos.** |

**FASE 2: RECOPILACION DE LA INFORMACION**

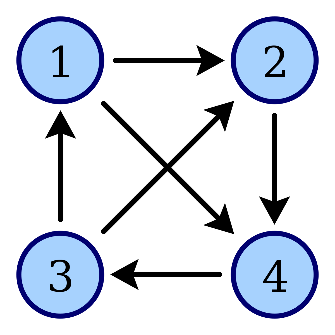
**MARCO TEORICO**

**GRAFO**

* Son estructuras discretas que constan de vértices y aristas que conectan entre sí esos vértices. Existen distintos tipos de grafos, que se diferencian entre sí por el tipo y el número de aristas que pueden conectar cada par de vértices.
* Ejemplos de problemas que se representan por medio de grafos:
  + Una red de carreteras que conecta ciudades.
  + Una red eléctrica.
* ¿Qué contiene un grafo?
  + **VERTICES**
    - Constituyen uno de los elementos que forman un grafo.
    - Como ocurre con el resto de las ramas de las matemáticas, a la teoría de grafos no le interesa qué son los vértices. Pueden verse simplemente como objetos.
    - Diferentes situaciones en las que pueden identificarse objetos y relaciones que satisfacen la definición de grafo pueden verse como tales y así aplicar la teoría en mención.
  + **ARISTAS**
    - Son las conexiones que existen en los vértices de un grafo. Una arista puede ser **dirigida** o **no dirigida.**
    - Sea **E** una arista dirigida entre el vértice **A** y **B**, con dirección **B**, quiere decir que se puede ir del vértice **A** al **B**, pero no del vértice **B** al **A**.
    - Sea **E** una arista no dirigida entre el vértice **A** y **B**, sin dirección, quiere decir que se del vértice **A** se puede llegar al vértice **B**, y viceversa.

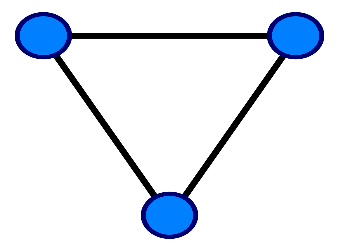
**GRAFO DIRIGIDO**

* Un grafo dirigido es un grafo en cual, si existe una arista entre un vértice **A** y un vértice **B**, esta arista tendrá una dirección final, ya sea **A** o **B**.



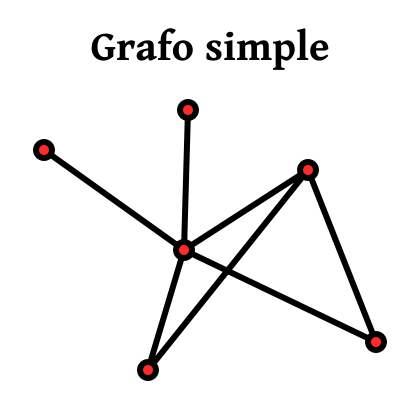
**GRAFO NO DIRIGIDO**

* Un grafo dirigido es un grafo en cual, si existe una arista entre un vértice **A** y un vértice **B**, esta arista no tendrá dirección final, y se podrá mover tanto del vértice **A** hasta el vértice **B**, como del vértice **B** hasta el vértice **A**.



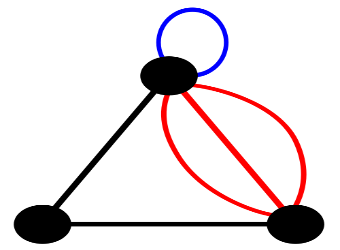
**GRAFO SIMPLE**

* Un grafo simple **G = (V, E)** consta de **V**, un conjunto no vacío de vértices, y de **E**, un conjunto de pares no ordenados de elementos de **V**. A estos pares se les llama aristas, Es decir un grafo en cual un vértice no tiene más de una arista hacia otro vértice.



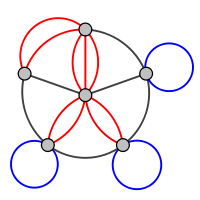
**MULTIGRAFO**

* Un multígrafo **G = (V,E)** consta de un conjunto **V** de vértices, un conjunto **E** de aristas y una función f de **E en {{u,v} | u,v є V, u ≠ v}.** Se dice que las aristas **e1** y **e2** son aristas múltiples o paralelas si **f(e1) = f(e2),** Es decir un grafo en cual un vértice puede tener cero o varias aristas hacia otro vértice.



**PSEUDOGRAFO**

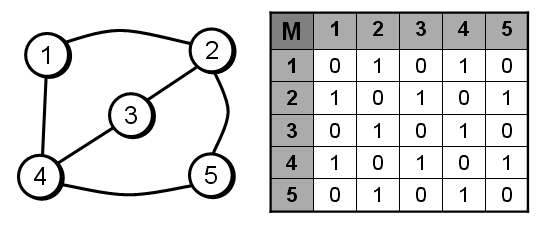
* Un pseudografo **G = (V,E)** consta de un conjunto **V** de vértices, un conjunto **E** de aristas y una función **f** de **E** en **{{u,v} | u,v є V }**. Una arista e es un bucle, o lazo, si **f(e) = { u, u }** **= { u }** para algún **u** є **V**.



**FORMAS DE REPRESENTACION DE UN GRAFO**

**MATRIZ DE ADYACENCIA**

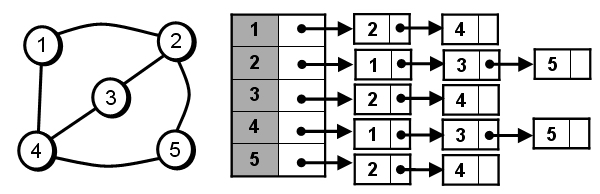
* Sea **G = (V,E)** es un grafo simple con **|V|=n.** Enumeramos los vértices de G de manera arbitraria como **v1,v 2,…,v n**. La matriz de adyacencia **A (o A G)** de G con respecto a este listado de los vértices es la matriz booleana **nxn** que tiene un **1** en la posición **(i,j)**, si **vi** y **vj** son adyacentes, y tiene un **0** en la posición **(i,j)**, si **v** y **v** no son adyacentes.



* Si el grafo es no dirigido, su matriz de adyacencia será simétrica.

**LISTA DE ADYACENCIAS**

* Cuando el grafo es dirigido, una representación por listas de adyacencia enumera todos los vértices finales de las aristas que salen de cada uno de los vértices del grafo.



**FASE 3: BUSQUEDA DE SOLUCIONES CREATIVAS**

* **CREACION DEL GRAFO**

Estrategia usada para la generación de ideas: Lluvia de ideas.

**Situación 1) ESTRUCTURA DE LA ARISTA**

**Idea (1)**

* + - Inicialmente se pensó que las aristas iban a contener los siguientes atributos:
      * **Nodo final.**
      * **Nodo inicial.**
      * **Peso.**
      * **Si es dirigida o no.**
      * **Dirección arista.**
      * **Llave.**

Los anteriores atributos dado que toda arista siempre va tener un nodo final y un nodo inicial, independientemente de que la arista sea o no sea dirigida.

**Idea (2)**

* + - En la segunda idea se remodelo la idea de arista y sus componentes, al inicio se planteaba que la arista tendría nodo inicial y nodo final, pero nos dimos cuenta que no es necesario, ¿Por qué?, dado que nuestros nodos tendrán una lista de adyacencias, sea un nodo A qué tiene de lista de adyacencia a los siguientes nodos { B, C, E}, quiere decir que este tendrá una arista hacia ellos, y el nodo A será el nodo inicial, de manera que si creamos la arista se estaría volviendo a repetir el nodo inicial, esto produciría un aumento de la complejidad espacial de manera innecesaria. De este modo la estructura de la arista quedaría de la siguiente manera:
      * **Nodo final.**
      * **Peso.**
      * **Llave.**

**Idea (3)**

* + - En el tercer planteamiento, se modelo la arista como se había propuesto inicialmente, con un nodo final y nodo inicial. Lo anterior a razón de que es responsabilidad del grafo tener conocimiento de todas las aristas existentes en el mapeo del grafo, por lo tanto, este debe tener acceso a todas, no solamente los nodos incidentes en esta.
      * **Nodo final.**
      * **Nodo inicial**
      * **Peso.**
      * **Llave.**

**Situación (2) ESTRUCTURA DEL GRAFO**

**Idea (1)**

* + - Hubo dificultad para saber cuál sería la mejor manera de definir si el grafo es dirigido o no. Inicialmente se modelo la estructura del grafo de la siguiente forma:
      * **Contenedora de todos los nodos que estarían en el grafo.**
      * **Lista de adyacencia**
      * **Matriz de adyacencia**
      * **Un valor que indicara si el grafo era dirigido o no.**

**Idea (2)**

* + - En esta segunda idea por medio de **herencia** e **interfaces** se planteó lo siguiente:
      * **Utilizar dos clases de grafo, dirigido y no dirigido.**

**Idea (3)**

* + - En nuestra tercera idea, se siguió utilizando herencia. Dado que el grafo se desea representar de dos formas, se establecido el siguiente modelo:
      * **Utilizar una clase grafo, abstracta, esta únicamente tendrá la contenedora de nodos**
      * **Utilizar la clase lista de adyacencia como primera forma de representación del grafo, esta heredara de la clase grafo. Implementara todos los métodos propuestos los cuales son:**
        + **Dijkstra**
        + **Prim**
        + **Kruskal**
        + **DFS**
        + **BFS**

**Esta podrá darse de dos formas, lista de adyacencia dirigida y lista de adyacencia no dirigida, de este modo tendríamos en cuenta los dos tipos principales de grafos.**

* + - * **Utilizar la clase matriz de adyacencia como otra forma de representación del grafo, esta heredara de la clase abstracta grafo. Implementara todos los métodos propuestos los cuales son:**
        + **Dijkstra**
        + **Prim**
        + **Kruskal**
        + **DFS**
        + **BFS**
        + **Floyd-Warshall.**

**Esta clase podrá darse de dos formas, matriz de adyacencia dirigida y matriz de adyacencia no dirigida. Con lo anterior tendríamos en cuenta los dos tipos principales de grafo.**

**FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES**

* **DESCARTE DE IDEAS NO FACTIBLES**

**Situación (1)**

|  |  |
| --- | --- |
| IDEA NO FACTIBLE | JUSTIFICACION |
| 1 | Dado que la arista tiene un nodo inicio y un nodo final aumentara la complejidad espacial de los algoritmos de forma innecesaria, ¿Por qué?, Supongamos que existe un **nodo A** con una lista de adyacencia que contiene los siguientes vértices **{B, C, W, X}**, esto quiere decir que el **nodo A** tiene una arista hacia esos nodos, donde el inicio de la arista siempre será el **nodo A**, de manera innecesaria se estaría usando el espacio del nodo inicio, ya que sabemos que todas las aristas que salgan hacia sus nodos adyacentes van a iniciar desde el **nodo A.** |
| 2 | A razón de que el grafo debe tener conocimiento de todas las conexiones que en el subyacen, es importante que se sepa desde donde inicia la arista **nodoSource** y hasta dónde va la arista **nodoEnd**. |

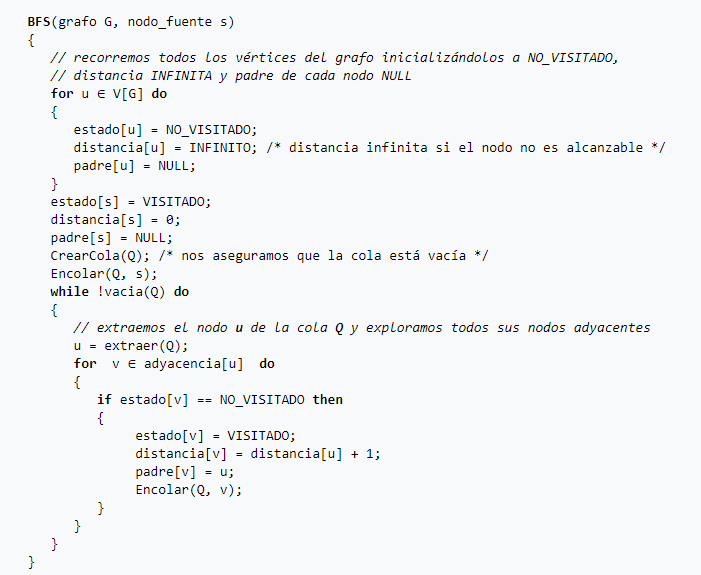
**Situación (2)**

|  |  |
| --- | --- |
| IDEA NO FACTIBLE | JUSTIFICACION |
| 1 | La idea de que un atributo **boolean** defina si el grafo será o no dirigido, no es suficiente para abarcar todas las demás estructuras que contiene un grafo. Si un grafo es dirigido la función de agregar un puente va ser totalmente distinta que agregar un puente entre un grafo no dirigido. Por lo anterior esta idea se descarta. |
| 2 | Dado que en este modelo no se tienen en cuenta las principales representaciones del grafo, las cuales son, matriz de adyacencia y lista de adyacencia, se descarta. |

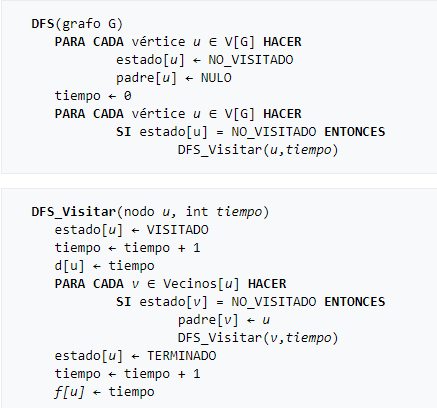
**Seudocódigo algoritmos que se utilizaran:**

Recorridos sobre Grafos (BFS, DFS):

* BFS:

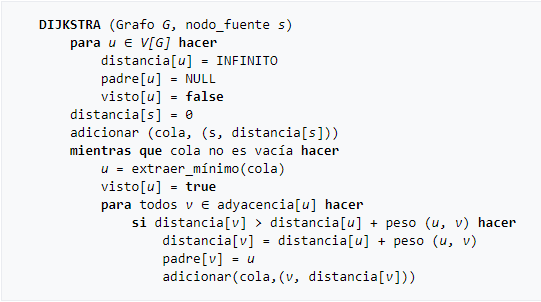


* DFS:

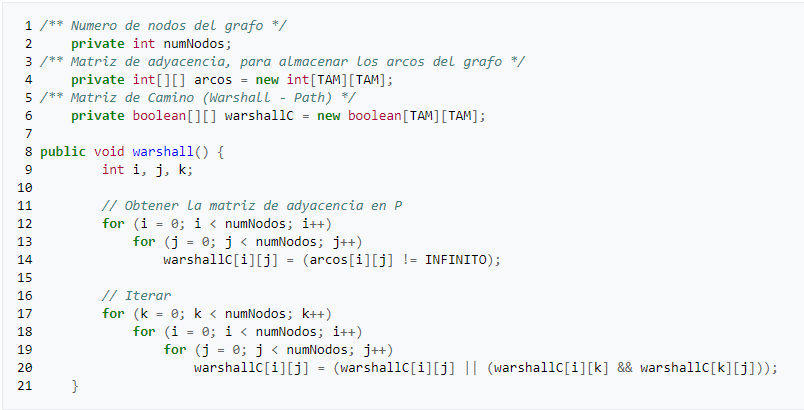


Caminos de Peso Mínimo (Dijkstra, Floyd-Warshall)

* Dijkstra:

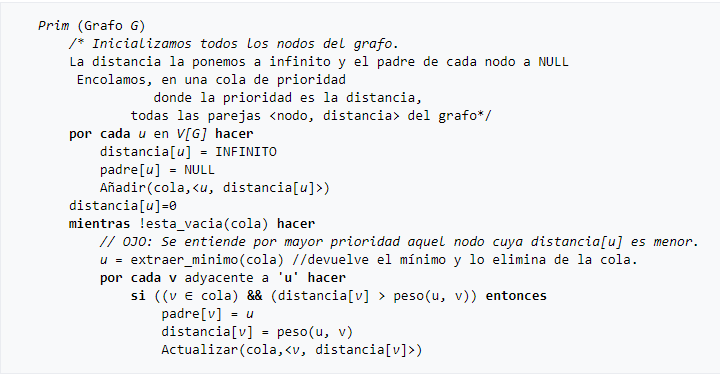


* Floyd-Warshall:

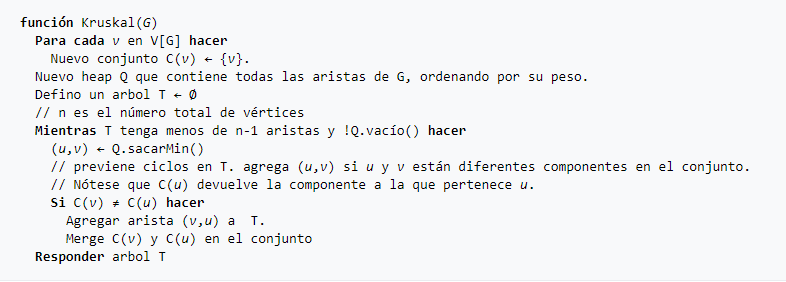


Árbol de Recubrimiento Mínimo -MST- (Prim, Kruskal).

* Prim:



* Kruskal:



**FASE 5: EVALUACION Y SELECCIÓN DE LA MEJOR SOLUCION**

**CRITERIOS ELEGIDOS**

* **Situación (1)**
  + **Complejidad Espacial** 
    - **[5].** La estructura de la arista solo contendrá un nodo final, por lo tanto, solo se guardará un espacio de memoria para un nodo final.
    - **[4].** La estructura de la arista contendrá un nodo final y un nodo inicial, por lo tanto, se guardarán dos espacios de memoria.
  + **Responsabilidad del grafo y facilidad de representar en la interfaz** 
    - **[4].** La estructura de la arista solo contendrá un nodo final, por lo tanto, solo se guardará un espacio de memoria para un nodo final, pero el grafo no conocerá todas las aristas que en el subyacen. Por lo anterior se tendría que acceder a cada nodo el grafo para revisar cuales son las conexiones de este.
    - **[5].** La estructura de la arista contendrá un nodo final y un nodo inicial, por lo tanto, se guardarán dos espacios de memoria. Del modo anterior el grafo si conocerá todas las conexiones o aristas que en el subyacen y de este modo a la hora de pintarse en la interfaz solo va requerir conocer la estructura donde se almacenan todas las aristas.
* **Situación (2)**
  + **Niveles de representación del grafo**
    - **[3].** La estructura del grafo se podrá representar por medio de lista de adyacencia y matriz de adyacencia.
    - **[2].** La estructura de la arista se podrá representar por matriz de adyacencia o por lista de adyacencia.
    - **[1].** La estructura del grafo no tendrá ningún tipo de representación, todas las operaciones se realizarán sobre él.

**SITUACION (1)**

|  |  |  |
| --- | --- | --- |
| **Idea** | **Complejidad espacial** | **Responsabilidad del grafo y facilidad de representar en la interfaz** |
| 3 | 4 | 5 |

De este modo, la solución analizada y escogida estratégicamente que nos resolverá la situación #1 en la cual estaba la problemática de como estructurar la arista es la **solución/idea #3**, dado que el grafo tendrá conocimiento de todo el grafo, pero habrá un grado de complejidad espacial mayor.

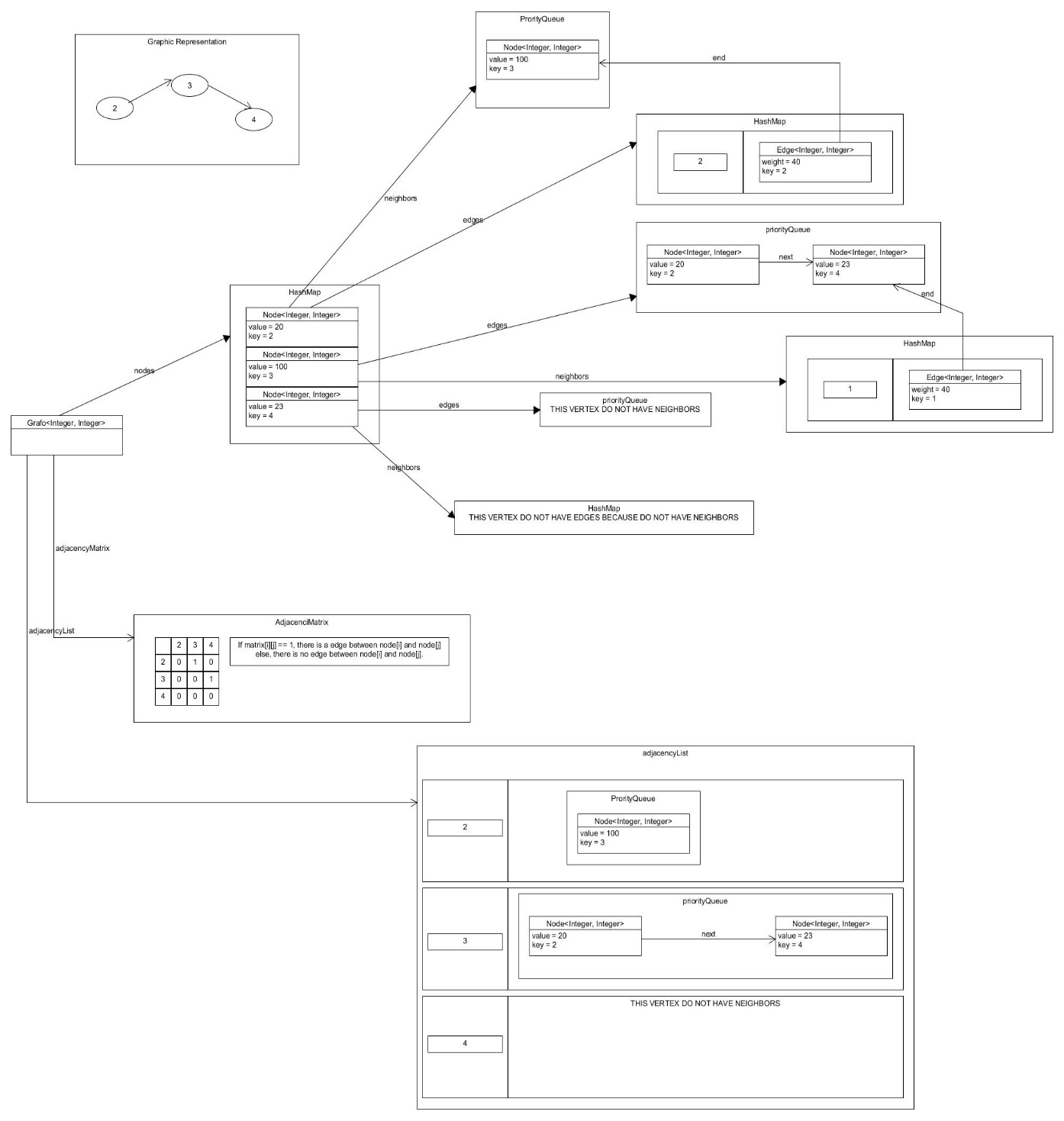
**SITUACION (2)**

|  |  |
| --- | --- |
| **Idea** | **Niveles de representación del grafo** |
| 3 | 3 |

A razón de que con esta solución el grafo se podrá representar de las dos formas más importantes, se utilizará para solucionar la problemática #2 la cual era sobre el planteamiento de la estructura del grafo.

**FASE 6: PREPARACION DE INFORME Y ESPECIFICACIONES.**

**DIAGRAMA DE OBJETOS**



**DIAGRAMA DE CLASES**

**Acá toca meter 4 o 5 diagramas de clase o qué?**

**Clase = Graph**

N = Node

SetupStage1:

* Se crea un objeto de la clase grafo<Integer, Integer> con un HashMap <Integer, Node> nodes con 10 nodos.
* adjacencyMatrix = new AdjacencyMatrix = int[10][10]
* adjacencyList = new AdjacencyList = HashMap<Integer, Node> (Contendrá los 10 nodos cada uno con sus respectivos nodos adyacentes)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 1** | **Objetivo de la prueba:** Comprobar que el método addNode agrega satisfactoriamente al grafo y sus representaciones.    **Firma del método:** public void addNode(Node newNode) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | addNode(node : N) | setupStage1() | newNode : Node    nuevo nodo que va a agregar. | Nodo agregado satisfactoriamente al HashMap nodes.    Se agrega una nueva posición a la matriz de adyacencia y a la lista de adyacencias. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 2** | **Objetivo de la prueba:** Comprobar que el método removeNode(k keyNode)  Elimina satisfactoriamente nodos del grafo y sus representaciones.    **Firma del método:** public void removeNode(K keyNode) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | removeNode(Key: k) | setupStage1() | keyNode : k    Llave del nodo que se quiere eliminar. | Nodo eliminado satisfactoriamente del grafo y sus representaciones. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 3** | **Objetivo de la prueba:** Comprobar que el método addEdge(k keyNodeStart, k keyNodeEnd) añade una arista que comienza en el nodo con llave keyNodeStart y el node keyNodeEnd    **Firma del método:** public void addEdge(K keyNodeStart, K keyNodeEnd) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | addEdge(Key: k1, Key: k2) | setupStage1() | KeyNodeStart representa el nodo desde el cual se va crear la arista y KeyNodeEnd representa el nodo al cual llega la arista | Se crea una arista entre los dos nodos y se actualizan sus diferentes representaciones. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 4** | **Objetivo de la prueba:** Comprobar que el método removeEdge(K keyNode, K keyEdge ) remueve la arista con llave keyEdge del nodo con llave keyNode    **Firma del método:** public void removeEdge (K keyNode, K keyEdge) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | removeEdge(Key: k1, Key: k2) | setupStage1() | keyNode representa la llave del nodo que tiene la arista, mientras keyEdge representa la llave de la arista que se quiere eliminar. | Arista eliminada satisfactoriamente del nodo y las diferentes representaciones del grafo. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 5** | **Objetivo de la prueba:** Comprobar que el método DFS(K keyNodeStart)  Recorre el grafo de manera exitosa, haciendo un recorrido a profundidad.    **Firma del método:** public void DFS(K keyNodeStart) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | DFS (Key: k1) | setupStage1() | keyNodeStart, llave del valor donde se va iniciar el recorrido DFS | Comprobar que todos los nodos del grafo fueron recorridos de manera exitosa. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 6** | **Objetivo de la prueba:** Comprobar que el método BFS(K keyNodeStart)  Recorre el grafo de manera exitosa, haciendo un recorrido en amplitud.    **Firma del método:** public void BFS(K keyNode, K keyEdge) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | BFS (Key: k1) | setupStage1() | keyNodeStart, llave del valor donde se va iniciar el recorrido BFS | Comprobar que todos los nodos del grafo fueron recorridos de manera exitosa. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 7** | **Objetivo de la prueba:** Comprobar que el método updateAdjacencyMatrix ()  actualiza la matriz de adyacencia cuando sea necesario (se agrega o elimina un nodo u arista)    **Firma del método:** public void updateAdjacencyMatrix () | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | updateAdjacencyMatrix () | setupStage1() |  | La matriz de adyacencia se actualizara de manera exitosa cuando se agrega u elimine un nodo o arista |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 8** | **Objetivo de la prueba:** Comprobar que el método updateAdjacencyList ()  actualiza la lista de adyacencia cuando sea necesario (se agrega o elimina un nodo u arista)    **Firma del método:** public void updateAdjacencyList () | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Graph | updateAdjacencyList () | setupStage1() |  | La lista de adyacencia se actualizara de manera exitosa cuando se agrega u elimine un nodo o arista |

**Clase = Node**

SetupStage1:

Neighbors = new PriorityQueue<Node<V,K>> (con 10 nodos )

Edges = new HashMap<K, Edge>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 1** | **Objetivo de la prueba:** Comprobar que el método addEdge(K keyEdge ,Node node) añade una arista al HashMap edges de aristas.    **Firma del método:** public void addEdge ( K keyEdge, Node node) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Node | addEdge(K k, Node node) | setupStage1() | keyEdge representa la llave de la nueva arista y node el nodo al cual llega la arista | Se agrega una arista entre con llave keyEdge al HashMap de aristas . |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 2** | **Objetivo de la prueba:** Comprobar que el método removeEdge(K keyEdge)  Remueve la arista del hashMap de aristas.    **Firma del método:** public void removeEdge(K keyEdge) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| Node | removeEdge(K key) | setupStage1() | KeyEdge representa la llave de la arista que se quiere eliminar. | Se elimina la arista del hashMap de aristas. |

**Diseño de pruebas para representación en lista de adyacencia dirigida y no dirigida.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 1** | **Objetivo de la prueba:**Comprobar que el método addEdge(K keySource, K keyDestiny, K keyEdge, **double** weight) añade una arista.    **Firma del método:**public void  addEdge(K keySource, K keyDestiny, K keyEdge, **double** weight) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | addEdge(K keySource, K keyDestiny, K keyEdge, **double** weight)) | **generalScenary()** | **keySource:** Llave del nodo desde se quiere iniciar la arista  **keyDestiny:** Llave de la dirección de la arista  KeyEdge: llave de la arista  **Weight:** El peso de la arista | Se agrega una arista entre el nodo con llave keySource y keyDestiny, como es un grafo no dirigido, el nodo con llave **keySource** debe aparecer en las adyacencias de **keyDestiny** y de forma contraria también. |
| **AdjacencyList** | addEdge(K keySource, K keyDestiny, K keyEdge, **double** weight)) | **generalScenary()** | **keySource:** Llave del nodo desde se quiere iniciar la arista  **keyDestiny:** Llave de la dirección de la arista  KeyEdge: llave de la arista  **Weight:** El peso de la arista | Se agrega una arista entre el nodo con llave **keySource** y el nodo con llave **keyDestiny**. Como el grafo es dirigido, solo deberá aparecer el nodo con la llave **keyDestiny** en las adyacencias de el nodo con llave **keySource,** no se debe dar la situación contraria. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 2** | **Objetivo de la prueba:**Comprobar que el método addNode(Node<V, K> newNode)    **Firma del método: public** **void** addNode(Node<V, K> newNode) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | addNode(Node<V, K> newNode) | **generalScenary()** | **newNode**: El nodo que se creó, y se desea agregar al grafo. | Se agrega a la hash de nodos el nodo **newNode.** |
| **AdjacencyList** | addNode(Node<V, K> newNode) | **generalScenary()** | **newNode:** El nodo que se creó, y se desea agregar al grafo. | Se agrega a la hash de nodos el nodo **newNode.** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 3** | **Objetivo de la prueba:**Comprobar que el método bfs(Node<V,K> source)    **Firma del método: public** HashMap<K,K> bfs(Node<V,K> source) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | **bfs(Node<V,K> source)** | **generalScenary()** | **source**: Nodo desde se quiere iniciar el recorrido en anchura. | Un **HashMap<K,K>** donde estará almacenado cada nodo con su llave, y el valor será la llave del padre en la ruta. |
| **AdjacencyList** | **bfs(Node<V,K> source)** | **generalScenary()** |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 4** | **Objetivo de la prueba:**Comprobar que el método dfs(Node<V,K> source)    **Firma del método: public** HashMap<K,K> dfs(Node<V,K> source) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | dfs(Node<V,K> source) | **generalScenary()** | **source:** Nodo desde se quiere iniciar el recorrido en profundidad. | Un **HashMap<K,K>** donde estará almacenado cada nodo con su llave, y el valor será la llave del padre en la ruta. |
| **AdjacencyList** | dfs(Node<V,K> source) | **generalScenary()** | **source:** Nodo desde se quiere iniciar el recorrido en profundidad. | Un **HashMap<K,K>** donde estará almacenado cada nodo con su llave, y el valor será la llave del padre en la ruta. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 6** | **Objetivo de la prueba:**Comprobar que el método prim(Node<V,K> source) retorne el hash de padres de la ruta.  **Firma del método: public** HashMap<K,K> prim(Node<V,K> source) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | **prim(Node<V,K> source)** | **generalScenary()** | **source:** Nodo desde se quiere iniciar el recorrido en profundidad. | Un **HashMap<K,K>** donde estará almacenado cada nodo con su llave, y el valor será la llave del padre en la ruta. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 7** | **Objetivo de la prueba:**Comprobar que el método kruscal() retorne el hash de padres de la ruta.  **Firma del método: public** HashMap<K,K> kruscal() | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirected** | **kruscal()** | **generalScenary()** | **Ninguna** | Un **HashMap<K,K>** donde estará almacenado cada nodo con su llave, y el valor será la llave del padre en la ruta. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba No. 5** | **Objetivo de la prueba:**Comprobar que el método dijsktra(Node<V,K> source) retorne el hash con las mínimas distancias desde eso nodo hacia los demás.    **Firma del método: public** HashMap<K, Double> dijsktra(Node<V,K> source) | | | |
| **Clase** | **Método** | **Escenario** | **Valores de entrada** | **Resultado** |
| **AdjacencyListUndirecte**d | **dijsktra(Node<V,K> source)** | **generalScenary()** | source: Nodo desde se quieren conocer las distancias mínimas. | Un **HashMap<K,Double**> donde estará almacenado cada nodo con su llave, y el valor será la distancia mínima que hay entre el nodo source y el nodo con la llave. |
| **AdjacencyList** | **dijsktra(Node<V,K> source)** | **generalScenary()** | source: Nodo desde se quieren conocer las distancias mínimas hacia los nodos a los cuales el nodo **source** pueda llegar. | Un **HashMap<K,Double**> donde estará almacenado cada nodo con su llave, y el valor será la distancia mínima que hay entre el nodo source y el nodo con la llave. |

**FASE 7: IMPLEMENTACION**

* La implementación se encuentra en el siguiente repositorio:
  + https://github.com/KingSatur/Proyecto-Estructuras.git

**BIBLIOGRAFIA**

* Diapositivas dispuestas en moddle para el entendimiento de los grafos.
  + Introducción a Grafos Escuela de Ingeniería de Sistemas y Computación Universidad del Valle.
  + Grafos Representación, Isomorfismos y Caminos Escuela de Ingeniería de Sistemas y Computación Universidad del Valle.
  + Matemáticas Discretas. **Richard Johnsonbaugh.**