

A thick, dark blue vertical bar runs along the left edge of the page, extending from the top to the bottom.

2 DE DICIEMBRE DE 2020

# TAREA INTEGRADORA 3

SANTIAGO HURTADO, SEBASTIAN MORALES, CRISTIAN MORALES

## Contenido

Tarea Integradora 3 .....	5
1. MÉTODO DE LA INGENIERÍA .....	6
2. RECOPIACION DE INFORMACION .....	8
<b>Algoritmos para usar</b> .....	9
<b>Referencias</b> .....	12
3. BÚSQUEDA DE SOLUCIONES CREATIVAS .....	13
4. TRANSFORMACIÓN DE IDEAS EN DISEÑOS PRELIMINARES .....	14
5. EVALUACIÓN DE ALTERNATIVAS .....	21
CASOS DE PRUEBA .....	25
TAD GRAFO .....	44

**En el presente documento se especifica el método de la ingeniería de la tarea 3, donde se incluye los tads del grafo, el diseño de los casos de prueba, en la misma carpeta de este documento encontrara los diagramas de clases del grafo, tanto, así como del modelo.**

## Objetivos

### Unidad 4: Grafos

- OE4.1. Explicar los conceptos básicos sobre la teoría de grafos.
- OE4.2. Modelar la información de un problema utilizando un grafo como estructura de datos.
- OE4.3. Aplicar los recorridos en profundidad y por niveles de los grafos en el contexto de un problema dado.
- OE4.4. Aplicar los algoritmos de Dijkstra y Floyd-Warshall para resolver problemas de búsqueda de caminos más cortos en el contexto de un problema dado.
- OE4.5. Aplicar los algoritmos de Prim y Kruskal para resolver problemas de árboles de recubrimiento mínimo en el contexto de un problema dado.
- OE4.6. Diseñar y construir un grafo representado por matrices de adyacencias y listas de adyacencias.
- OE4.7. Implementar los algoritmos de recorridos sobre grafos y búsqueda de caminos más cortos.
- OE4.8. Diseñar y construir las pruebas unitarias de cada uno de los grafos implementados.

### Enunciado

Usted debe desarrollar (analizar, diseñar e implementar) un programa que resuelva un problema específico que sea modelado utilizando grafos y que para su solución se apliquen al menos dos (2) de los algoritmos de grafos que se estudiarán durante el curso: Recorridos sobre Grafos (BFS, DFS), Caminos de Peso Mínimo (Dijkstra, Floyd-Warshall), Árbol de Recubrimiento Mínimo -MST- (Prim, Kruskal).

El problema debe ser definido por usted y su grupo de máximo 3 personas y los requisitos son los siguientes:

- Desarrollar 2 versiones de Grafo (su solución debe funcionar sin problema con las dos versiones, es decir, el programa debe admitir el cambio de la implementación utilizada en cualquier momento y funcionar bien indistintamente de la que se esté usando). Cada grafo debe ser desarrollado desde el TAD hasta las pruebas unitarias automáticas.
- Llevar a cabo y documentar cada una de las fases del método de la ingeniería para la solución del problema planteado.
- Documentar apropiadamente las fases de análisis y diseño con el documento de especificación de requerimientos, el diseño del TAD, diagramas de clase y objetos, y el diseño de los casos de pruebas de las pruebas unitarias automáticas.
- Su programa debe contar con una interfaz gráfica de usuario que permita utilizar las funcionalidades que responden a los requerimientos del problema.
- Implementar todos los algoritmos de grafos vistos en clase así no sean utilizados en su proyecto para la solución del problema.

Su proyecto puede ser un juego, un programa que gestione la solución de un problema de la vida real, entre otros. Otra posibilidad es que su problema sea de programación competitiva como los siguientes: [Ejemplo 1](#), [Ejemplo 2](#).

Para cada entrega, ya que todas deben estar en el mismo repositorio, no hay que entregar nuevamente la dirección del repositorio, en cambio si deben actualizar el readme.me del repositorio explicando el contenido de este, enlazando apropiadamente los documentos y explicando de qué se trata el proyecto (el nombre del proyecto ni su descripción debería hacer mención de que es un proyecto de curso, sino sólo como un desarrollo orientado a resolver el problema planteado por ustedes).

El último commit de cada entrega debe ser etiquetado con el tag MilestoneX. De esta manera, podrán seguir haciendo commits, pero se sabrá sin ambigüedad hasta donde está el desarrollo de cada una.

### Primera Entrega.

La estructura Grafo completamente analizada, diseñada, implementada (2 representaciones + BFS, DFS y Dijkstra) y probada, más una explicación detallada del problema a solucionar. Su entrega debe incluir:

1. Informe del seguimiento del Método de la ingeniería sobre el problema que están abordando.
2. Especificación del TAD Grafo. Nombre, representación, invariante, operaciones y la especificación de cada una de las operaciones en términos de entrada y salida (tal como se revisó en la Unidad 2 del curso).  
El problema por solucionar:
  - a. Enunciado suficientemente claro de **la situación problemática** que solucionarán.
  - b. Especificación de Requerimientos Funcionales del programa que darán solución al problema.
3. El **readme.md** del repositorio debe explicar brevemente (en inglés) de qué se trata el proyecto. Deben enlazar los archivos que documentan el proyecto (en formato pdf) y deben especificar las condiciones técnicas del mismo (lenguaje, sistema operativo y ambiente de desarrollo).

### Segunda Entrega.

Cristian

1. Diseño de Diagrama de Clases del TAD Grafo incluyendo las dos implementaciones (Floyd-Warshall, Prim y Kruskal) y las mejores prácticas de diseño (no olvidar desacoplamiento y generics).

Sebastian

2. Diseño de pruebas unitarias:
  - a. De las operaciones estructurales del grafo (agregar, eliminar y consultar).
  - b. De los algoritmos vistos en clase.

Santiago

3. Implementación completa de:
  - a. Los grafos incluyendo los algoritmos vistos en clase.
  - b. Las pruebas unitarias automáticas diseñadas sobre los grafos.
4. Actualizar el **readme.md** del repositorio en el cual se debe explicar brevemente (en inglés) de qué se trata el proyecto. Deben enlazar los archivos que documentan el proyecto (en formato pdf).

### Entrega Final.

Los entregables de la primera y segunda entrega más el diseño, la implementación y las pruebas unitarias del programa que da solución al problema. Actualizar el **readme.md** del repositorio en el cual se debe explicar brevemente (en inglés) de qué se trata el proyecto. Deben enlazar los archivos que documentan el proyecto (en formato pdf). Esta es la rúbrica de evaluación de la entrega final: [Rúbrica Proyecto](#).

**Video explicativo de la entrega final.** En el video deben presentar todos los integrantes del equipo. En el momento en que una persona se encuentre hablando debe visualizarse su video en una miniatura. En el video deben presentar de forma breve la documentación solicitada, haciendo énfasis en los elementos más importantes de esta entrega, al igual que una demostración de la implementación. Duración máxima del vídeo: 5 minutos.

# Tarea Integradora 3

## [Problema por resolver](https://vjudge.net/problem/UVA-13127)

<https://vjudge.net/problem/UVA-13127>

Arsène Lupin es un caballero ladrón y un maestro del disfraz; él ha sido responsable de los robos no que un individuo de mente correcta creería posible. También es, en gran medida, el hombre de las damas.

Lupin está a punto de dejar todo lo que está haciendo para ayudar a algunos de sus amigos que están planeando un robo de banco en el infame Reino de Axum: sus amigos han identificado la ubicación de los bancos que están dispuestos a robar, así como la ubicación de las comisarías de policía que sirven a la ciudad. De hecho, han elaborado un mapa de toda la ciudad en el que las carreteras bidireccionales se han detallado los sitios de conexión y los tiempos de viaje entre los sitios.

A pesar de la naturaleza criminal de sus actividades, Lupin tiene un estricto código que sigue para evitar manchar su reputación: nunca ha sido capturado por las autoridades. Con el fin de ayudar a sus amigos y al mismo tiempo, mantener su bien ganada reputación, Lupin se pregunta qué bancos pueden ser robados, así que son los más alejados de cualquier comisaría que sirva al Reino de Axum.

**Entrada** La entrada consiste en varios casos de prueba. Cada caso de prueba comienza con 4 números enteros separados en blanco

$N, M, B, P$  ( $1 \leq N \leq 1\,000$ ,  $0 \leq M$ ,  $1 \leq B \leq N$ ,  $0 \leq P < N$ ) que denotan, respectivamente, el número de sitios en la ciudad, el número de caminos en la ciudad, el número de bancos en la ciudad, y el número de las comisarías de la ciudad. Las siguientes líneas  $M$  contienen cada una tres enteros separados en blanco  $U, V, T$  ( $0 \leq U < N$ ,  $0 \leq V < N$ ,  $U \neq V$ ,  $0 \leq T \leq 10\,000$ ) que denota que hay un camino entre los sitios  $U$  y  $V$  que lleva unidades de tiempo  $T$  en tránsito. La siguiente línea contiene  $B$  en blanco y separadas por pares números de sitio que identifican la ubicación de los bancos. Si  $P \neq 0$ , entonces sigue una línea con  $P$  en blanco-separado y dos números de sitio distintivos que identifican la ubicación de las estaciones de policía. Puedes asumir que un banco y la comisaría nunca se encuentran en el mismo sitio.

## Salida

Para cada caso de prueba, saque dos líneas. La primera línea debe contener dos figuras separadas en blanco  $S, E$  denotan, respectivamente, el número de bancos más alejados de cualquier comisaría de policía y el mínimo el tiempo que tomaría el tránsito desde cualquier estación de policía a estos bancos. Si  $E$  no es un número entero, y en su lugar se pone un '\*'. La segunda línea debe contener números enteros separados por espacios en blanco, en orden ascendente, correspondientes a los sitios donde se encuentran los bancos con un tiempo mínimo desde cualquier comisaría de policía siendo exactamente unidades de tiempo  $E$ .

# 1. MÉTODO DE LA INGENIERÍA

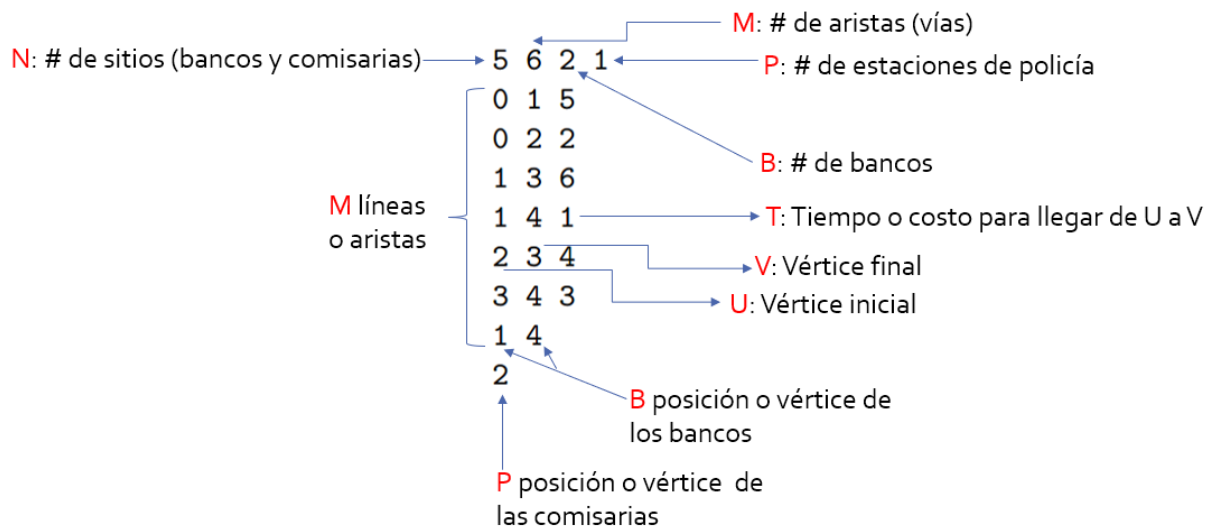
Necesidades:

- Identificar los bancos que están más lejos de las estaciones de policía.
- Comparar las distancias mínimas de los bancos a las estaciones de policía
- Analizar el formato de entrada y salida.

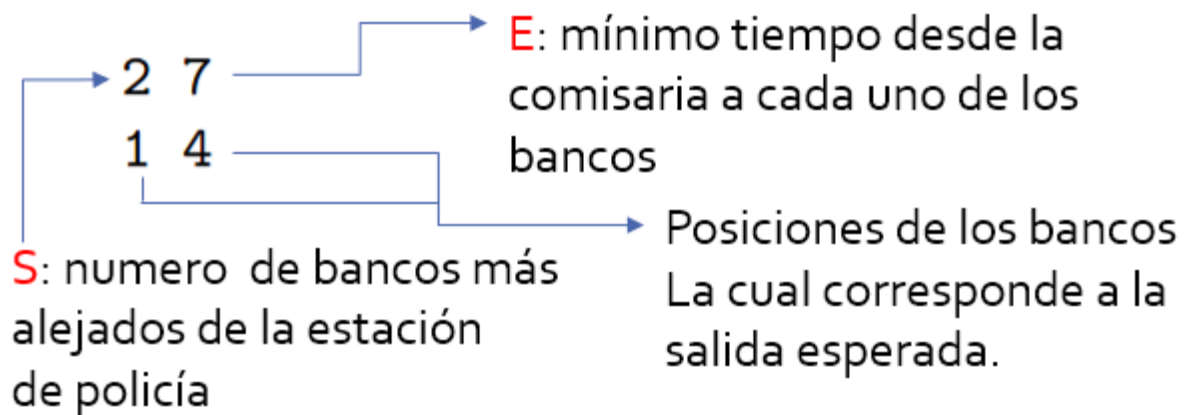
## Definición del problema

Encontrar el banco o los bancos más alejados de cualquier comisaría además de mostrar el mínimo tiempo que tomaría el tránsito desde cualquier estación de policías a estos bancos, donde el ladrón pueda efectuar un robo.

Formato de entrada



Formato salida



## **Requerimientos Funcionales**

- Encontrar el número de bancos que se encuentren más alejados de las estaciones de policías, y que tengan la mínima cantidad de recorrido desde cualquier estación de policía.
- Utilizar como mínimo dos algoritmos de grafos (camino de peso mínimo, recorrido sobre grafos y árbol de recubrimiento mínimamente)
- Diseñar una interfaz gráfica la cual permita explicar, y correr el programa desarrollado.
- Mostrar tiempo de ejecución del algoritmo.
- Permitir la carga de casos de prueba predeterminados, como cualquier otro caso valido de prueba.
- El programa debe estar desarrollado en una clase adicional para que funcione en el juez online.
- El programa debe estar desarrollado mediante un modelamiento adecuado el cual cumpla con su solución.

## **Requerimientos no Funcionales**

- Implementar la estructura de los grafos genéricos, los grafos implementados deben ser: Grafo con listas de adyacencias, Matriz de adyacencias. también se deben incluir los algoritmos de grafos para (camino de peso mínimo, recorrido sobre grafos y árbol de recubrimiento mínimo)
- Mostrar una previsualización del grafo mediante java swing o javafx, la cual muestre los puntos de las comisarias, bancos, aristas con su respectivo peso, y demás.
- El programa debe tener pruebas unitarias de las estructuras de datos.



## 2. RECOPIACION DE INFORMACION

### **Banco**

Un banco es un tipo de entidad financiera de crédito cuyo principal fin es el control y la administración del dinero, por medio de distintos servicios ofrecidos como el almacenaje de grandes cantidades de dinero, realización de operaciones financieras o la concesión de préstamos o créditos, entre otros.

La práctica habitual de un banco es la recogida de capitales de diferentes individuos o empresas que depositan su confianza y sus recursos en el mismo por medio de cuentas de ahorro o cuentas corrientes.

A la vez un banco funciona como una empresa más y cuenta con sus propios fondos en muchas ocasiones. Por supuesto, también con una idea de negocio propia a la hora de afrontar operaciones de crédito o de otros tipos en el ámbito de las finanzas.

### **Robo de bancos**

La expresión robo de un banco, atraco de un banco o asalto de un banco hace referencia a un delito que consiste en la sustracción de dinero de un banco, sea por parte de un individuo solitario o en grupo, usualmente mediante la amenaza y coacción de sus empleados para lograr su fin, empleando o no la violencia física. Se considera a aquellas personas que participaron como ladrones y atracadores de bancos.

### **Distancia recorrida**

Según la Física, la distancia recorrida puede definirse como el espacio recorrido. En este sentido, cuando un objeto móvil realiza su trayectoria lo hace recorriendo un espacio. Como tal, la distancia recorrida será, pues, el total del espacio recorrido expresado en unidades de longitud, fundamentalmente el metro.

### **Comisaría**

Se denomina **comisaría**, **delegación** o **estación de policía** al edificio de carácter permanente utilizado como cuartel general u oficina de policía. El nombre proviene del hecho de la persona al cargo de comisario.

Las comisarías normalmente están repartidas a lo largo del territorio mediante una distribución geográfica por distritos, estando cada una al cargo de la seguridad ciudadana de su zona.

Normalmente, este tipo de edificios contienen oficinas, distintos servicios, comodidades para su personal y lugares de estacionamiento para los vehículos oficiales. También pueden contener calabozos temporales para los detenidos y salas de interrogatorio.

### **Interfaz de usuario**

Conjunto de elementos que permiten la interacción entre el programa y el usuario.

# Algoritmos para usar

## ALGORITMO DE DIJKSTRA

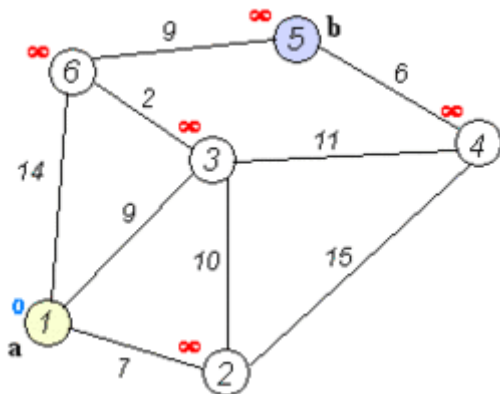
### Descripción

El algoritmo de dijkstra determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos. Algunas consideraciones:

- Si los pesos de mis aristas son de valor 1, entonces bastará con usar el algoritmo de BFS.
- Si los pesos de mis aristas son negativos no puedo usar el algoritmo de dijkstra, para pesos negativos tenemos otro algoritmo llamado Algoritmo de
- Bellmand-Ford.

### Como trabaja

Primero marcamos todos los vértices como no utilizados. El algoritmo parte de un vértice origen que será ingresado, a partir de esos vértices evaluaremos sus adyacentes, como Dijkstra usa una técnica greedy – *La técnica greedy utiliza el principio de que para que un camino sea óptimo,*



*todos los caminos que contiene también deben ser óptimos-* entre todos los vértices adyacentes, buscamos el que esté más cerca de nuestro punto origen, lo tomamos como punto intermedio y vemos si podemos llegar más rápido a través de este vértice a los demás. Después escogemos al siguiente más cercano (con las distancias ya actualizadas) y repetimos el proceso. Esto lo hacemos hasta que el vértice no utilizado más cercano sea nuestro destino. Al proceso de actualizar las distancias tomando como punto intermedio al nuevo vértice se le conoce como **relajación (relaxation)**.

## **ALGORITMO DE BELLMAN-FORD**

### **Descripción**

El algoritmo de Bellman-Ford determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos. La diferencia de este algoritmo con los demás es que los pesos pueden tener valores negativos ya que Bellman-Ford me permite detectar la existencia de un ciclo negativo.

### **Como trabaja**

El algoritmo parte de un vértice origen que será ingresado, a diferencia de Dijkstra que utiliza una técnica voraz para seleccionar vértices de menor peso y actualizar sus distancias mediante el paso de relajación, Bellman-Ford simplemente relaja todas las aristas y lo hace  $|V| - 1$  veces, siendo  $|V|$  el número de vértices del grafo.

Para la detección de ciclos negativos realizamos el paso de relajación una vez más y si se obtuvieron mejores resultados es porque existe un ciclo negativo, para verificar porque tenemos un ciclo podemos seguir relajando las veces que queramos y seguiremos obteniendo mejores resultados.

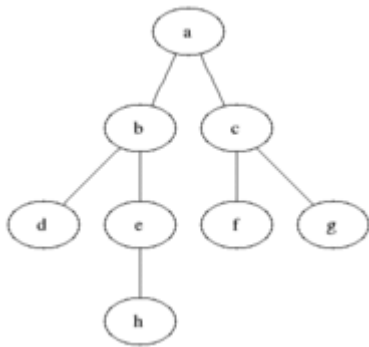
## **ALGORITMO BFS**

### **Descripción**

Este algoritmo de grafos es muy útil en diversos problemas de programación. Por ejemplo, halla la ruta más corta cuando el peso entre todos los nodos es 1, cuando se requiere llegar con un movimiento de caballo de un punto a otro con el menor número de pasos, cuando se desea transformar algo un numero o cadena en otro realizando ciertas operaciones como suma producto, pero teniendo en cuenta que no sea muy grande el proceso de conversión, o para salir de un laberinto con el menor número de pasos, etc. Podrán aprender a identificarlos con la práctica.

### **Como trabaja**

BFS va formando un árbol a medida que va recorriendo un grafo, veamos el ejemplo de la figura:



Si observan bien todo parte de un nodo inicial que será la raíz del árbol que se forma, luego ve los adyacentes a ese nodo y los agrega en una cola, como la prioridad de una cola es FIFO (primero en entrar es el primero en salir), los siguientes nodos a evaluar serán los adyacentes previamente insertados. una cosa bastante importante es el hecho de que no se pueden visitar 2 veces el mismo nodo o estado. ya que si no podríamos terminar en un ciclo interminable o simplemente no hallar el punto deseado en el menor número de pasos.

### **ALGORITMO DE FLOYD-WARSHALL**

En informática, el algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de programación dinámica.

## Referencias

[https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show\\_problem&problem=5038](https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=5038)

<https://runestone.academy/runestone/static/pythoned/Graphs/EITipoAbstractoDeDatosGrafo.html>

<https://economipedia.com/definiciones/banco.html>

[https://es.wikipedia.org/wiki/Robo\\_de\\_bancos](https://es.wikipedia.org/wiki/Robo_de_bancos)

<https://jariasf.wordpress.com/2012/03/19/camino-mas-corto-algoritmo-de-dijkstra/>

<https://jariasf.wordpress.com/2013/01/01/camino-mas-corto-algoritmo-de-bellman-ford/>

[https://es.wikipedia.org/wiki/Algoritmo\\_de\\_Floyd-Warshall](https://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall)

<https://jariasf.wordpress.com/2012/02/27/algoritmo-de-busqueda-breadth-first-search/>

### 3. BÚSQUEDA DE SOLUCIONES CREATIVAS

Cada numero corresponde a una alternativa diferente, las cuales se pueden combinar entre sí, para dar múltiples soluciones al problema, aquí están como ideas que se puede llevar a cabo al realizar el programa.

#### Algoritmo

1. Hallar las distancias mínimas de cada uno de los bancos a las comisarias usando el algoritmo de Dijkstra.
2. Floyd Warshall, encontrar la distancia mínima, entre cada par de vértices, y luego unirlos y validar las distancias de los bancos a las comisarias, además de que estos estén lo más alejados de las comisarias.
3. Bellman Ford, este algoritmo nos ayudará a encontrar la ruta más corta desde un nodo origen hacia las demás estaciones de policía, incluso si estas tienen pesos negativos, validando además la distancia más lejana del banco
4. BFS permite hallar el camino mínimo en un grafo en el menor número de interacciones, mediante un punto de inicio, este tocaría modificarlo, porque su versión inicial es cuando todas las aristas del grafo tienen peso 1.

#### Ingreso de la entrada

1. Para ingresar la entrada el usuario tiene que copiar en consola la entrada especificada en un archivo de texto.
2. Los casos de prueba se guarden en un archivo de texto plano, el cual, mediante la interfaz, al seleccionar uno, se pueda previsualizar.
3. Entrada de coordenadas a mano donde se le indique al usuario una guía para establecer con corrección una entrada, posteriormente esta se guarda como un archivo de texto.
4. Permitir la carga de un archivo de texto plano guardado en una carpeta del proyecto, con los casos bases, además de permitir que el usuario cree y valide su propio caso de prueba.

#### Interfaz

1. Una interfaz sencilla, la cual solo se tenga que digitar el caso de prueba, y en la consola me muestre la salida esperada del caso de prueba.
2. Una interfaz gráfica con java swing, la cual este acompañada de un menú el cual permita escoger los casos de prueba, además de una herramienta donde me los permita visualizar, otro punto es la previsualización del grafo.
3. Una interfaz realizada en java fx, mediante scene builder.
4. Diseño de la interfaz en código fxml.

#### Mostrar Grafo

1. La opción que después de correr un caso de prueba se realice la se previsualización del grafo en la interfaz.
2. Otro diseño, es tener como base los 3 primeros casos y dar soluciones predeterminadas mediante una imagen, cargadas en pantalla.
3. Se muestre una animación mediante un hilo, donde el ladrón vaya al banco.

4. No mostrar la visualización del grafo con sus vértices y aristas de peso.
5. Mostrarlo mediante coordenadas en consolas.

### Tiempo

1. Mostrar el tiempo en pantalla de ejecución del algoritmo, en un label.
2. No mostrar el tiempo de la ejecución del algoritmo.
3. Mostrar el tiempo mediante una animación de un reloj, mediante un hilo.

## 4. TRANSFORMACIÓN DE IDEAS EN DISEÑOS PRELIMINARES

### Algoritmo

1. Dijkstra Este algoritmo es uno de los mas eficaces en cuanto a tiempo a la hora de resolver la solución, porque con su complejidad algorítmica, el recubrimiento de los caminos más lejanos, y lentos en este caso, sería una técnica efectiva para dicha solución, identificando primero los vértices que no están conectados, para así poner en estos un \*, como especifica la salida.

```

1  método Dijkstra(Grafo,origen):
2      creamos una cola de prioridad Q
3      agregamos origen a la cola de prioridad Q
4      mientras Q no este vacío:
5          sacamos un elemento de la cola Q llamado u
6          si u ya fue visitado continuo sacando elementos de
7          marcamos como visitado u
8          para cada vértice v adyacente a u en el Grafo:
9              sea w el peso entre vértices ( u , v )
10             si v no ah sido visitado:
11                 Relajacion( u , v , w )

1  método Relajacion( actual , adyacente , peso ):
2      si distancia[ actual ] + peso < distancia[ adyacente ]
3          distancia[ adyacente ] = distancia[ actual ] + peso
4          agregamos adyacente a la cola de prioridad Q

```

2. Floyd warshall es una opción también viable porque mediante este podemos realizar lo mismo que con el Dijkstra, aquí podemos ver el pseudocódigo del algoritmo, y encontrar la distancia mínima entre cada par de vértices, este algoritmo podría ser eficaz, y ser uno que nos ayude a la solución del problema.

```
1 /* Suponemos que la función pesoArista devuelve el coste del camino que va de i a j
2   (infinito si no existe).
3 También suponemos que n es el número de vértices y pesoArista(i,i) = 0
4 */
5
6 int camino[][];
7 /* Una matriz bidimensional. En cada paso del algoritmo, camino[i][j] es el camino mínimo
8 de i hasta j usando valores intermedios de (1..k-1). Cada camino[i][j] es inicializado a
9
10 */
11
12 procedimiento FloydWarshall ()
13   para k: = 0 hasta n - 1
14
15       camino[i][j] = mín ( camino[i][j], camino[i][k]+camino[k][j])
16
17   fin para
```

3. El algoritmo de Bellman-Ford determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos. La diferencia de este algoritmo con los demás es que los pesos pueden tener valores negativos ya que Bellman-Ford me permite detectar la existencia de un ciclo negativo, se descarta, porque según UVA, la entrada para todos los casos de prueba tiene que ser enteros positivos, ya que no existe una arista con peso negativo. Aquí adjunto una imagen del algoritmo.



Considerar  $\text{distancia}[i]$  como la distancia más corta del vértice origen ingresado al vértice  $i$  y  $|V|$  el número de vértices del grafo.

```
1  método BellmanFord(Grafo,origen):
2      inicializamos las distancias con un valor grande
3      distancia[ origen ] = 0
4      para i = 1 hasta |V| - 1:
5          para cada arista E del Grafo:
6              sea ( u , v ) vértices que unen la arista E
7              sea w el peso entre vértices ( u , v )
8              Relajacion( u , v , w )
9      para cada arista E del Grafo:
10         sea ( u , v ) vértices que unen la arista E
11         sea w el peso entre vértices ( u , v )
12         si Relajacion( u , v , w )
13             Imprimir "Existe ciclo negativo"
15         Terminar Algoritmo

1  Relajacion( actual , adyacente , peso ):
2      si distancia[ actual ] + peso < distancia[ adyacente ]
3          distancia[ adyacente ] = distancia[ actual ] + peso
```

4. BFS, es una opción que podemos tener en cuenta, porque es como un Dijkstra, solo para aristas que no tienen peso, encontrando el camino mínimo, por otro lado, este algoritmo lo podemos usar, si se acomoda, en cuanto al peso y visita de los vértices en el grafo.

```
1  método BFS(Grafo,origen):
2      creamos una cola Q
3      agregamos origen a la cola Q
4      marcamos origen como visitado
5      mientras Q no este vacío:
6          sacamos un elemento de la cola Q llamado v
7          para cada vertice w adyacente a v en el Grafo:
8              si w no ah sido visitado:
9                  marcamos como visitado w
10                 insertamos w dentro de la cola Q
```

### **Ingreso de la entrada**

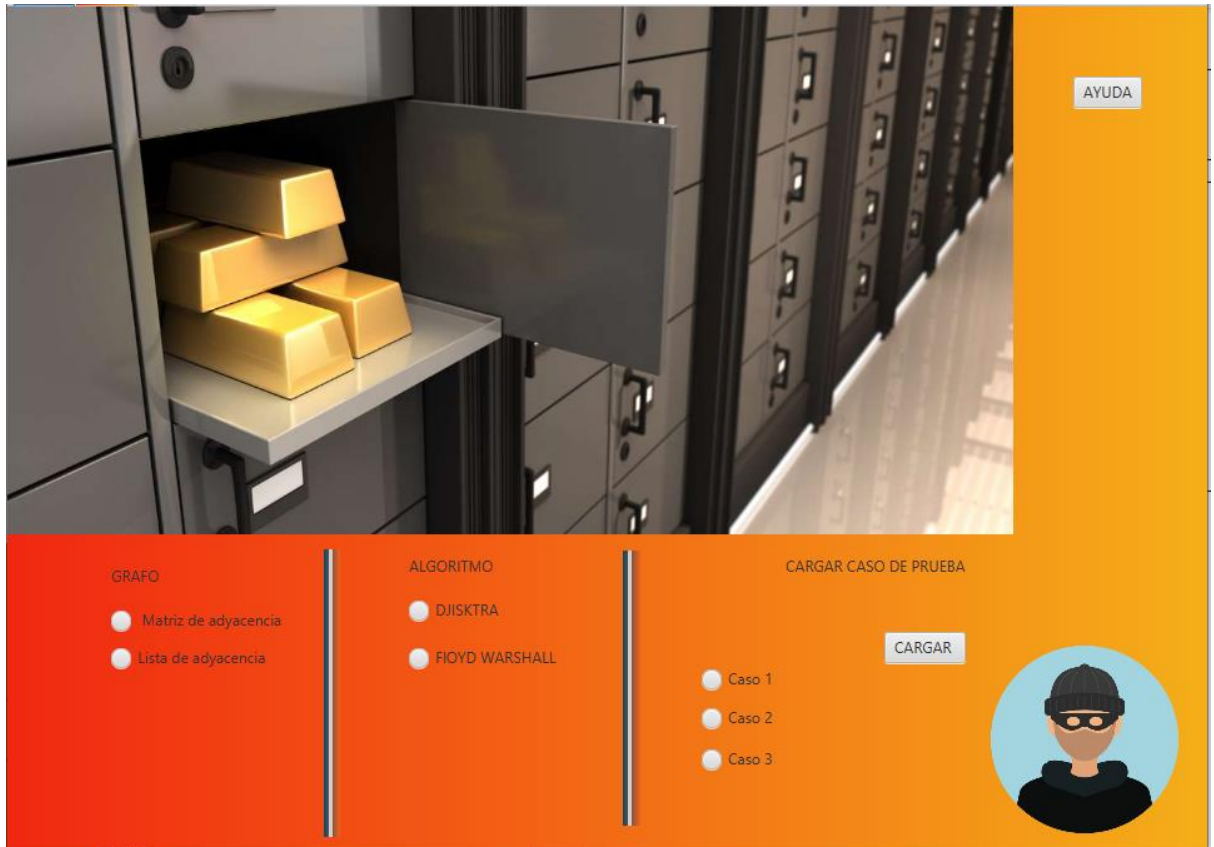
1. Para ingresar la entrada el usuario tiene que copiar en consola la entrada especificada en un archivo de texto, es una opción que descártanos porque el usuario muchas veces no conoce bien la entrada, y pueden aparecer confusiones y el programa no se ejecute bien finalmente.
2. Los casos de prueba se guarden en un archivo de texto plano, el cual, mediante la interfaz, al seleccionar uno, se pueda previsualizar es una opción buena, pero solo cumpliría para los 3 casos de pruebas cargados, porque otros no podrían ingresar, cosa que hace que no cumpla con los requerimientos.
3. Entrada de coordenadas a mano donde se le indique al usuario una guía para establecer una entrada, posteriormente esta se guarda como un archivo de texto, puede ser algo bueno para entender la entrada, pero puede traer confusiones a la hora de escribir números muy grandes, además de un tiempo.
4. Los casos de prueba están cargados en un archivo de texto plano, además que el usuario puede seleccionar un caso de prueba válido, para probar su solución, cumpliendo así con los requerimientos específicos del programa.

### **Interfaz**

1. Una interfaz sencilla, la cual solo se tenga que digitar el caso de prueba, y en la consola me muestre la salida esperada del caso de prueba, está interfaz seria solo el envío al juez en linea, pero no cumple los requerimientos mínimos.
2. Una interfaz gráfica con java swing, la cual este acompañada de un menú el cual permita escoger los casos de prueba, además de una herramienta donde me los permita visualizar, otro punto es la previsualización del grafo, además de derivar el problema para enviarlo al juez inline, es una opción viable, aunque se realice en una librería gráfica deprécate, pero conocemos una ventaja que es el dibujo del grafo, punto importante a la hora de presentar visualmente el problema.



3. Una interfaz realizada en java fx, mediante scene builder, es una opción que consideramos viable, podríamos usar ciertas cosas, pero en esta no tenemos el conocimiento para pintar un grafo en la pantalla.

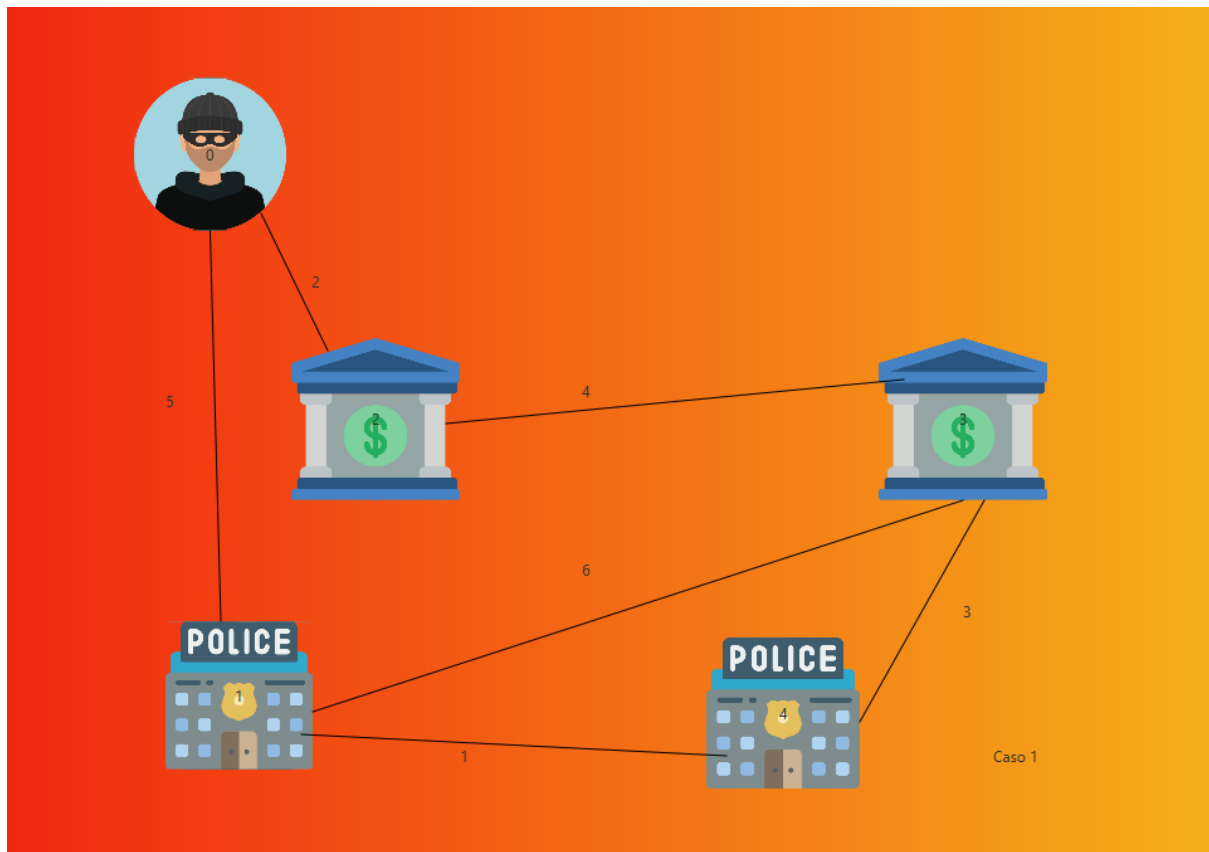


4. Diseño de la interfaz en código fxml.es una opción que nos llevaría mucho tiempo, investigar más del lenguaje, cosa que en proyectos así no podemos tomar ese riesgo.

### Mostrar Grafo

1. Es una solución viable, y mediante swing, se puede pintar el grafo, mediante las coordenadas dadas, además de que mostraría en detalle el grafo de cada ciudad y sitio.
2. Una solución viable, pero no funcionaría para todos los casos de prueba gráficamente, serían para los 3 casos de prueba.

Ejemplo:



3. Evaluamos que puede ser una opción dinámica, pero la descartamos, porque nos saldríamos algo de la solución, y contexto del problema, además sería una animación donde no se muestra el diseño del grafo como tal.
4. es una opción que descartamos, porque no cumple con nada de los requerimientos.
5. Es una opción que toca llevarla a cabo, para cumplir con una rubrica planteada, además de que, al enviar el problema al juez online, toca mediante esta forma.

## Tiempo

1. Es una opción viable, y precisa para sacar el tiempo de ejecución de los algoritmos, en el mismo método donde llamamos al algoritmo con la interfaz, por otro lado, con la fórmula de java para sacar el tiempo en milisegundos.
2. Sería una opción para descartar porque como tal, mediante el tiempo vemos la eficiencia de nuestro algoritmo, y como se evidencia el funcionamiento en cuanto a tiempo respecto a los casos de prueba.
3. Es una opción viable, pero con el tiempo, el diseño de un reloj y demás podrían afectarnos un poco en el proceso, saliéndonos del contexto de requerimientos y el problema como tal.

## 5. EVALUACIÓN DE ALTERNATIVAS

**Factible:** la solución es fácil de implementar, diseñar y analizar a la hora de realizarla.

**Eficiente:** La solución cumple con la realización de todos los requerimientos.

**Complejidad:** La solución, conjunto, a su diseño y la implementación que grado de dificultad presenta.

**Integral:** La solución abarca todos los requerimientos a realizar.

**Usabilidad y manejo:** La solución a la hora de probarlo el cliente es fácil de usar.

Escala de 1 a 5.

Solución / criterio	Factible	Eficiente	Complejidad	Integral	Usabilidad y manejo	Total	Validado
Algoritmo							
Alternativa 1	4	5	4	5	4	22	si
Alternativa 2	4	3	5	5	5	22	si
Alternativa 3	3	3	5	3	3	17	no
Alternativa 4	5	4	3	3	5	20	no
Entrada							
Alternativa 1	1	2	3	3	2	11	no
Alternativa 2	3	3	4	4	3	17	no
Alternativa 3	3	3	3	3	2	14	no
Alternativa 4	5	5	4	5	5	24	si
Interfaz							
Alternativa 1	4	4	5	4	5	22	si
Alternativa 2	4	3	5	5	5	22	si
Alternativa 3	4	4	3	3	5	19	no
Alternativa 4	2	2	1	2	1	8	no
Grafo							
Alternativa 1	3	4	4	4	3	18	si
Alternativa 2	3	2	4	3	4		no
Alternativa 3	2	2	3	2	2	11	no
Alternativa 4	1	1	1	1	2	6	no
Alternativa 5	4	4	4	4	4	20	si
Tiempo							
Alternativa 1	4	3	5	5	3	20	si
Alternativa 2	5	1	1	1	0	8	no
Alternativa 3	3	2	3	4	3	15	no

## **Estrategia y conclusión**

Este al ser un ejercicio de programación competitiva puede resultar algo complejo a la hora de mostrarlo en una interfaz, por lo tanto se desarrolla una solución precisa para dicho programa , además de ser aceptada por el juez online, evaluamos el diseño de interfaces en java, las distintas librerías gráficas, como vamos a hacer para la carga de entrada, la previsualización del grafo, y demás, como el tiempo , también algo fundamental que se pensó para la implementación del algoritmo, es cual nos sirve respecto al problema, aquí como tal vemos que el Dijkstra quedo, como también el Floyd Marshall, ya que son las mejores opciones para dicho problema, esta fueron unas de las estrategias que pensamos a la hora de llevar a cabo nuestra solución.

Identificar primero vértices disco nexos, en estos vértices estamos incluyendo la verificación de que si es comisaria o banco, posteriormente se irán contando, y si hay vértices que no se conectan con ninguna arista según las coordenadas de la entrada, retornara la salida especifica. Luego con el Dijkstra y Floyd warshall hallamos las distancias mínimas de los bancos a la comisaria, retornando el menor resultado.

Utilizar los algoritmos Dijkstra y Floyd para identificar los caminos mínimos para comparar el banco más lejano a las comisarias, y por último ver cuál es el tiempo mínimo que tomaba cualquier policía en llegar a ese banco.

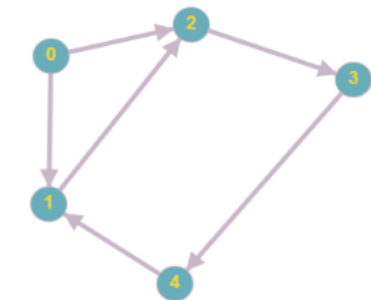

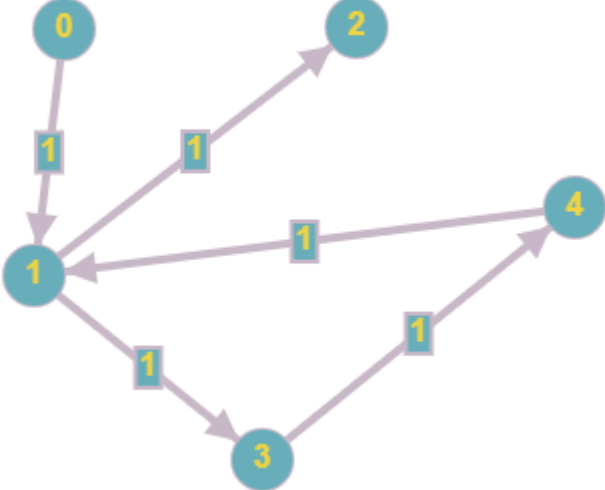




# CASOS DE PRUEBA

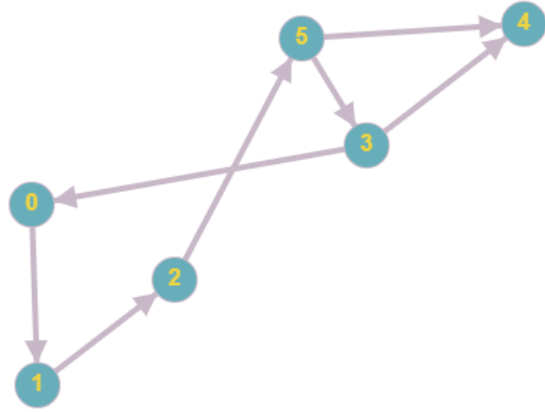
Diseño de los casos de prueba aplica tanto para grafo lista de adyacencia y grafo matriz de adyacencia. Se hace una simplificación del nombre al ponerlo en el diseño de los casos de prueba.

Escenarios

Nombre	Clase	Escenario
Setup1	GraphTest	Grafo vacio
Setup2	GraphTest	
Setup3	Graph Test	
Setup4	Graph Test	

Setup5

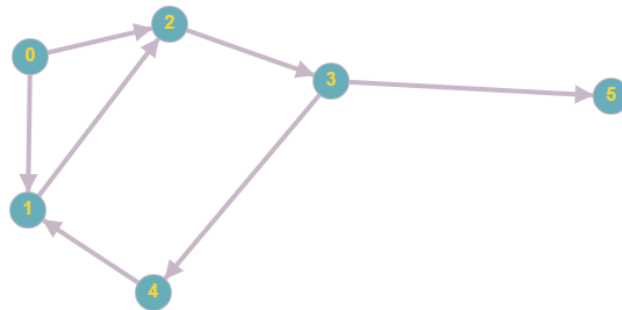
GraphTest



## Agregar

Objetivo de la prueba	Agregar un vertice al grafo cuando no existen vértices ni aristas en el grafo			
Clase	método	escenario	entrada	resultado
GrahpTest	Add1	Setup1	1	Vértice agregado correctamente

Objetivo de la prueba	Agregar un vértice al grafo cuando existen vértices conectados con aristas en el grafo			
Clase	método	Escenario	entrada	Resultado
GrahpTest	Add2	Setup2	5	Vertice agregado correctamente



Objetivo de la prueba	Agregar un vértice repetido al grafo			
Clase	método	Clase	entrada	Resultado
GrahpTest	Add3	Setup2	2	El vertice no se agrega

Objetivo de la prueba	Agregar un vértice que conecta bidireccionalmente con otro vértice			
Clase	método	Clase	entrada	Resultado
GrahpTest	Add4	Setup2	3	El vertice se agrega



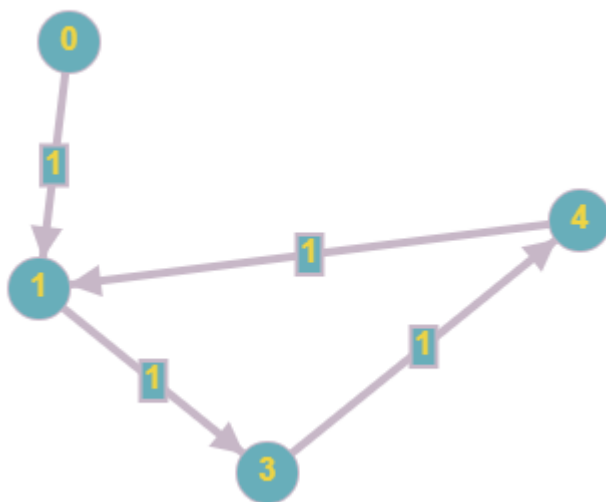
## Buscar

Objetivo de la prueba	Buscar el vértice indicado en el grafo			
Clase	método	escenario	entrada	resultado
GraphTest	testContainsVertex	Setup2	1	TRUE.

Objetivo de la prueba	Buscar un vértice que no existe en el grafo			
Clase	método	escenario	entrada	resultado
GraphTest	testContainsVertex	Setup2	10	FALSE

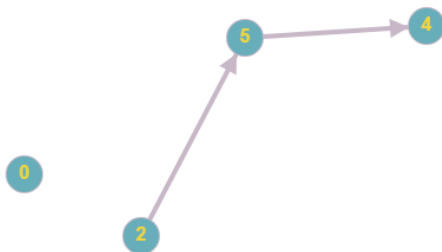
## Eliminar

Objetivo de la prueba	Eliminar el vertice indicado en el grafo			
Clase	método	escenario	entrada	resultado
GraphTest	testDeleteVertex1	Setup4	2	True

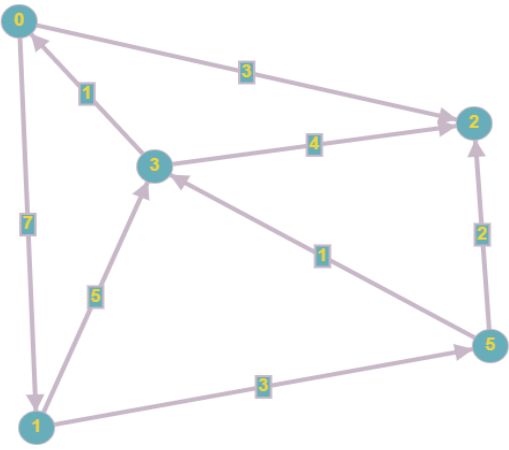
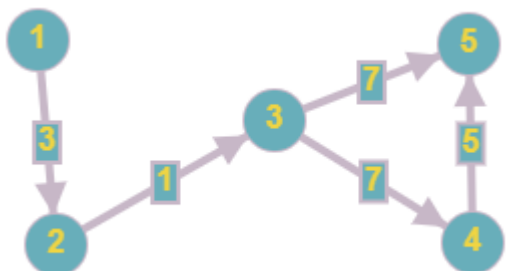
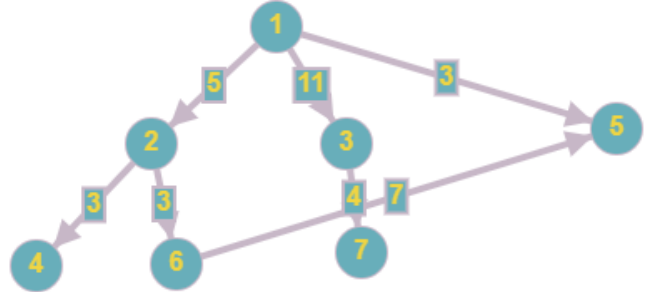


Objetivo de la prueba	Eliminar un vértice que no existe en el grafo			
Clase	método	escenario	entrada	resultado
GraphTest	<u>testDeleteVertex2</u>	Setup2	48	FALSE

Objetivo de la prueba	Eliminar más de un vértice que existe en el grafo			
Clase	método	escenario	entrada	resultado
GraphTest	<u>testDeleteVertex3</u>	Setup2	1,3	TRUE



Escenarios algoritmos

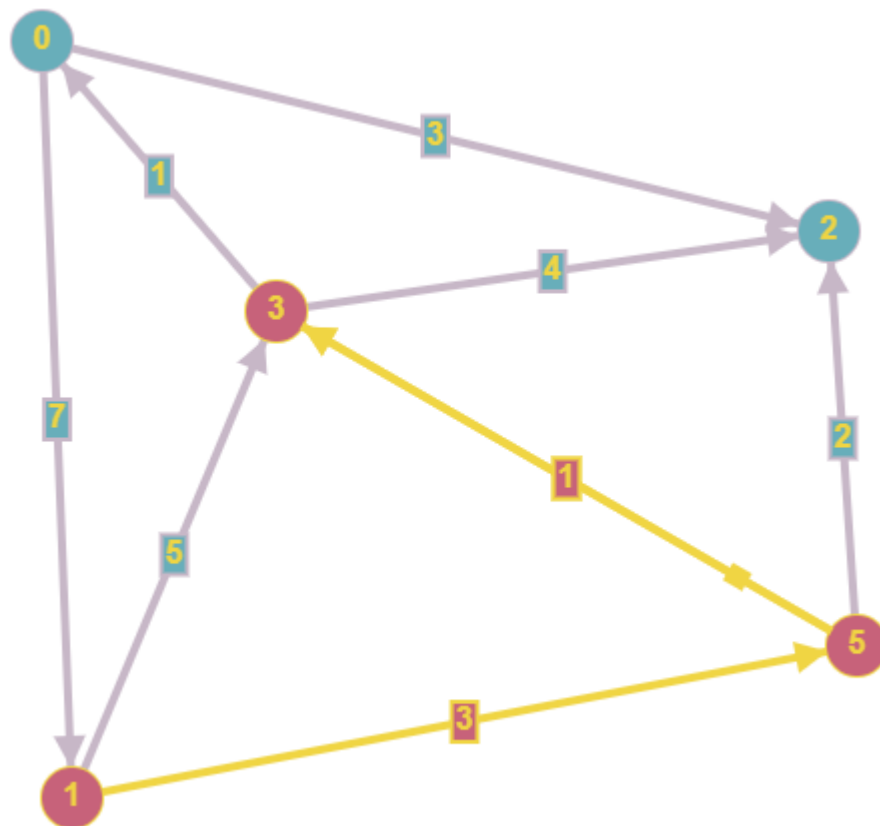
Nombre	Clase	Escenario
Setup1	AlgoritmTest	
Setup2	AlgoritmTest	
Setup 3	AlgoritmTest	

Setup4	AlgoritmTest	<pre> graph TD     0((0)) --- 6  1((1))     0 --- 6  2((2))     0 --- 5  4((4))     1 --- 8  2     1 --- 3  3((3))     3 --- 2  5((5))     4 --- 1  6((6))     6 --- 3  7((7)) </pre>
Setup5	AlgoritmTest	<pre> graph TD     0((0)) --- 8  2((2))     1((1)) --- 3  2     1 --- 8  5((5))     2 --- 9  4((4))     2 --- 2  5     2 --- 7  6((6))     3((3)) --- 6  5     4 --- 5  6     5 --- 8  7((7))     6 --- 8  7     5 --- 7  6 </pre>



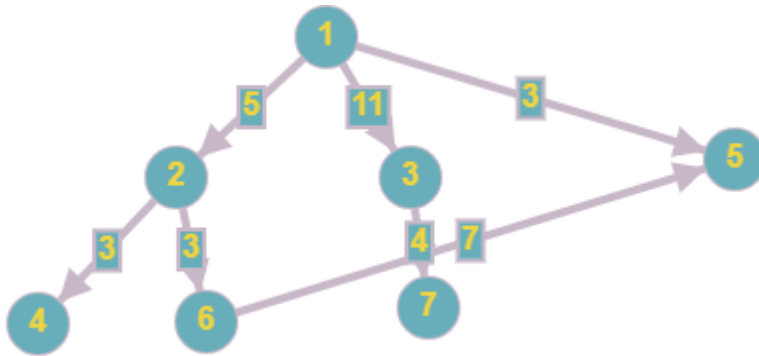
## Algoritmo Dijkstra

Objetivo de la prueba	Encontrar el camino más corto desde 1 a 3			
Clase	método	escenario	entrada	resultado
AlgoritmTest	TestDijkstra	Setup1	Grafo con 5 vertices conectados por aristas las cuales tienen determinado peso (ver gráfico)	1=>5=>3



## Algoritmo DFS

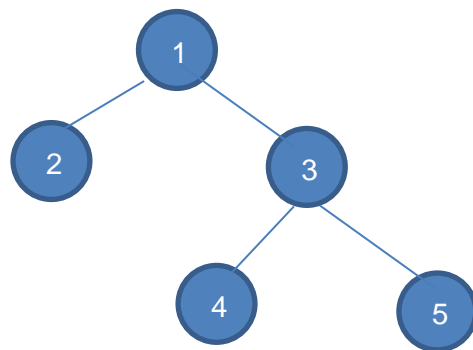
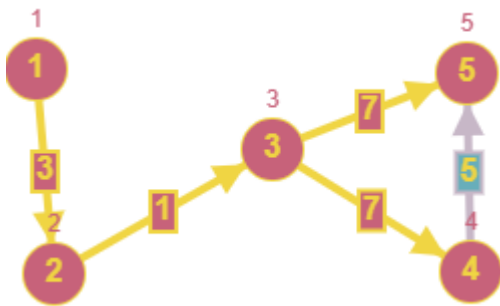
Objetivo de la prueba	Buscar y examinar los vértices de un grafo mediante el DFS			
Clase	método	escenario	entrada	resultado
AlgorithmTest	TestDFS	Setup3	Grafo con 7 vértices conectados, el cual el recorrido inicia en 1.	1,2,4,6,5,3,7



## Algoritmo BFS (Búsqueda en anchura)

Objetivo de la prueba	Buscar y examinar los vértices de un grafo mediante el bfs			
Clase	método	escenario	entrada	resultado
AlgorithmTest	TestBFS	Setup2	Grafo con 5 vertices	1,2,3,4,5

			conectados por aristas las cuales tienen determinado peso (ver gráfico)	
--	--	--	---	--



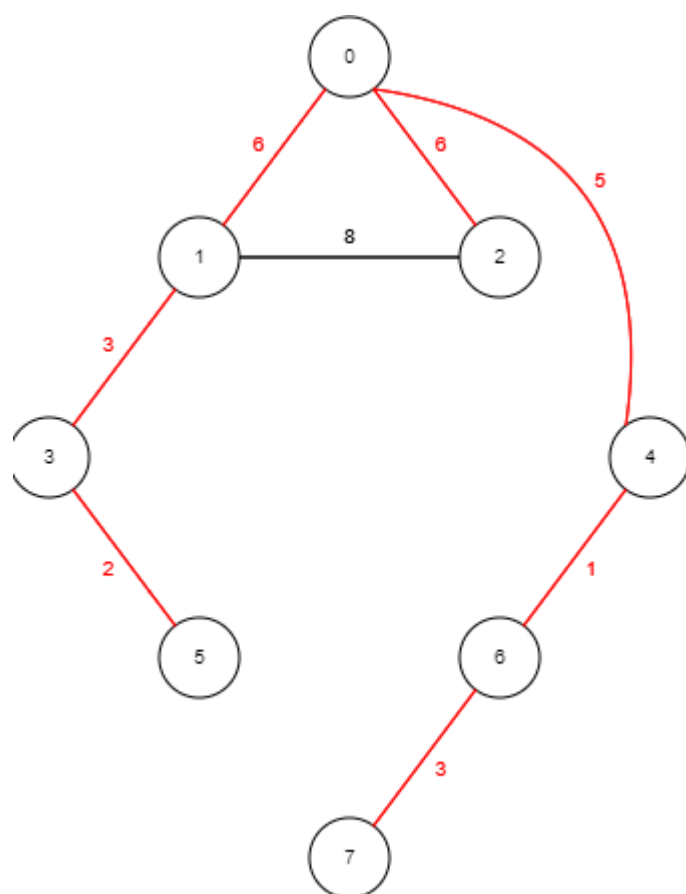
Algoritmo Floyd Warshall El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución.

Objetivo de la prueba	Encontrar el peso entre todos los pares de vértices en una única ejecución.			
Clase	método	escenario	entrada	resultado
AlgorithmTest	TestFloyd warshall	Setup1	1, 3	4

Algoritmo Prim encontrar un árbol recubridor mínimo en un grafo conexo, **no** dirigido y cuyas aristas están etiquetadas.  
 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.

Objetivo de la prueba	Encontrar 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			
Clase	método	escenario	entrada	resultado
AlgorithmTest	TestPrim	Setup4	Vertice 4  grafo	(VER IMAGEN)  5,3,1,0,2,4,6,7

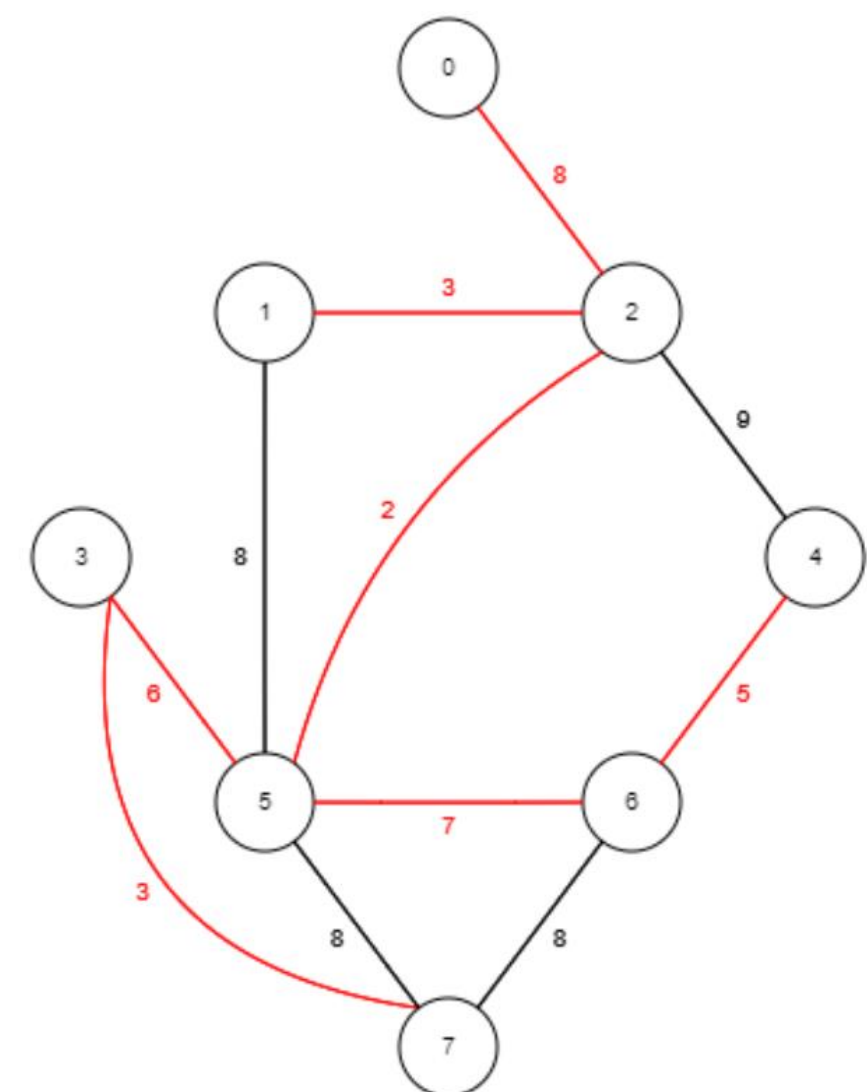
Vertex	Known	Cost	Path
0	T	5	4
1	T	6	0
2	T	6	0
3	T	3	1
4	T	0	-1
5	T	2	3
6	T	1	4
7	T	3	6



### Algoritmo Kruskal

busca un subconjunto de aristas 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

Objetivo de la prueba	busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo.			
Clase	método	escenario	entrada	resultado
AlgorithmTest	TestKruskal	Setup5	grafo	(VER IMAGEN)  0,2=8 1,2=3 2,5= 2 3,5=6 3,7=3 5,6=7 6,4=5



Aristas que deja fuera

2 —<sup>9</sup> 4

1 —<sup>8</sup> 5

5 —<sup>8</sup> 7

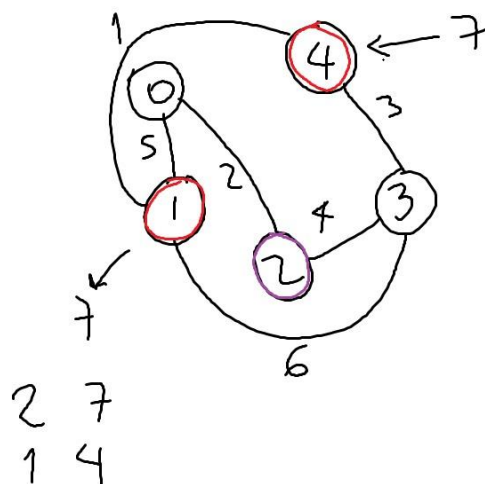
6 —<sup>8</sup> 7

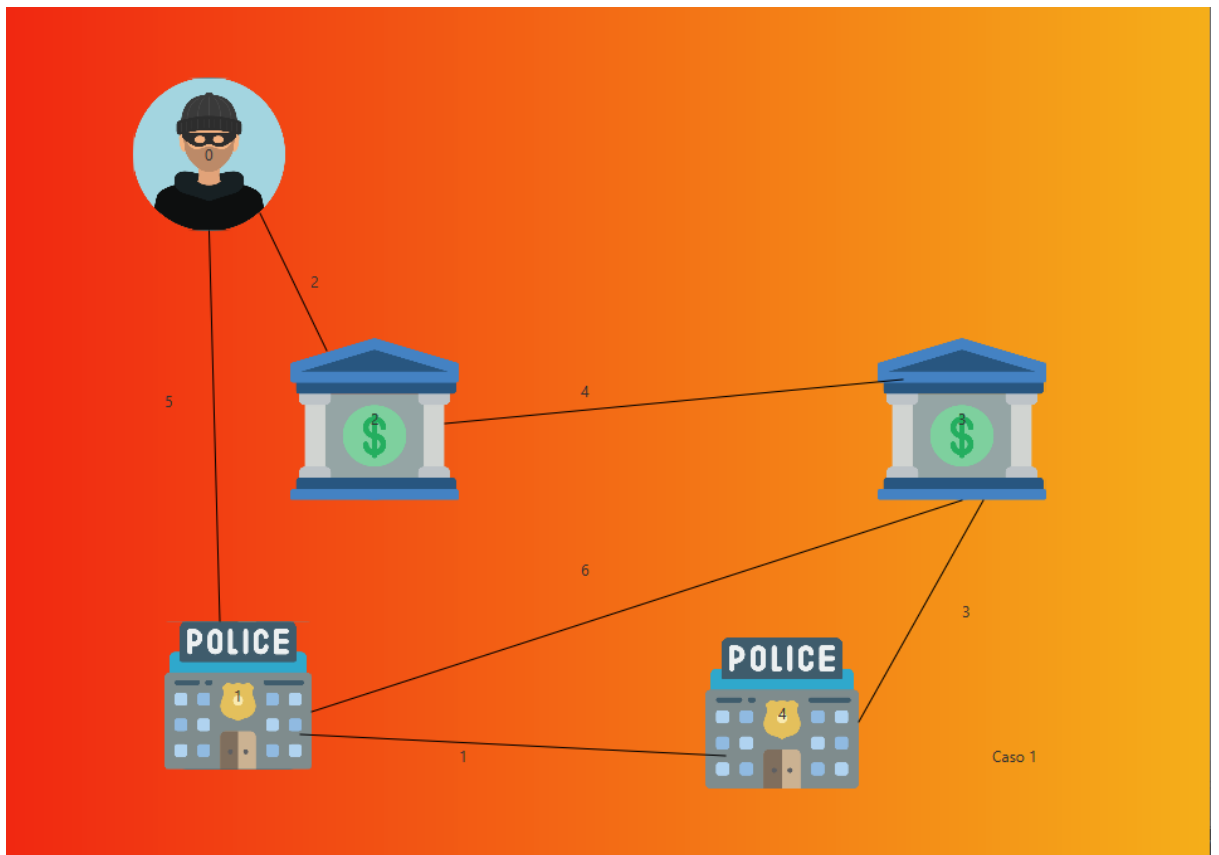
## Pruebas del modelo

Nombre	Clase	Escenario
Setup1	Solución	5 6 2 1 0 1 5 0 2 2 1 3 6 1 4 1 2 3 4 3 4 3 1 4 2
Setup 2	Solución	5 4 2 1 0 1 5 0 2 2 1 3 6 2 3 4 1 4 2
Setup3	Solución	5 6 2 2 0 1 5 0 2 2 1 3 6 1 4 1 2 3 4 3 4 3 1 4 2 3

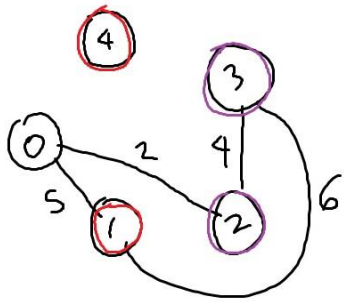


Objetivo de la prueba	Se prueba el caso 1.			
Clase	método	escenario	entrada	resultado
Solucion	Case1	Setup1	<pre> 5 6 2 1 0 1 5 0 2 2 1 3 6 1 4 1 2 3 4 3 4 3 1 4 2 </pre>	<pre> 2 7 1 4 </pre>

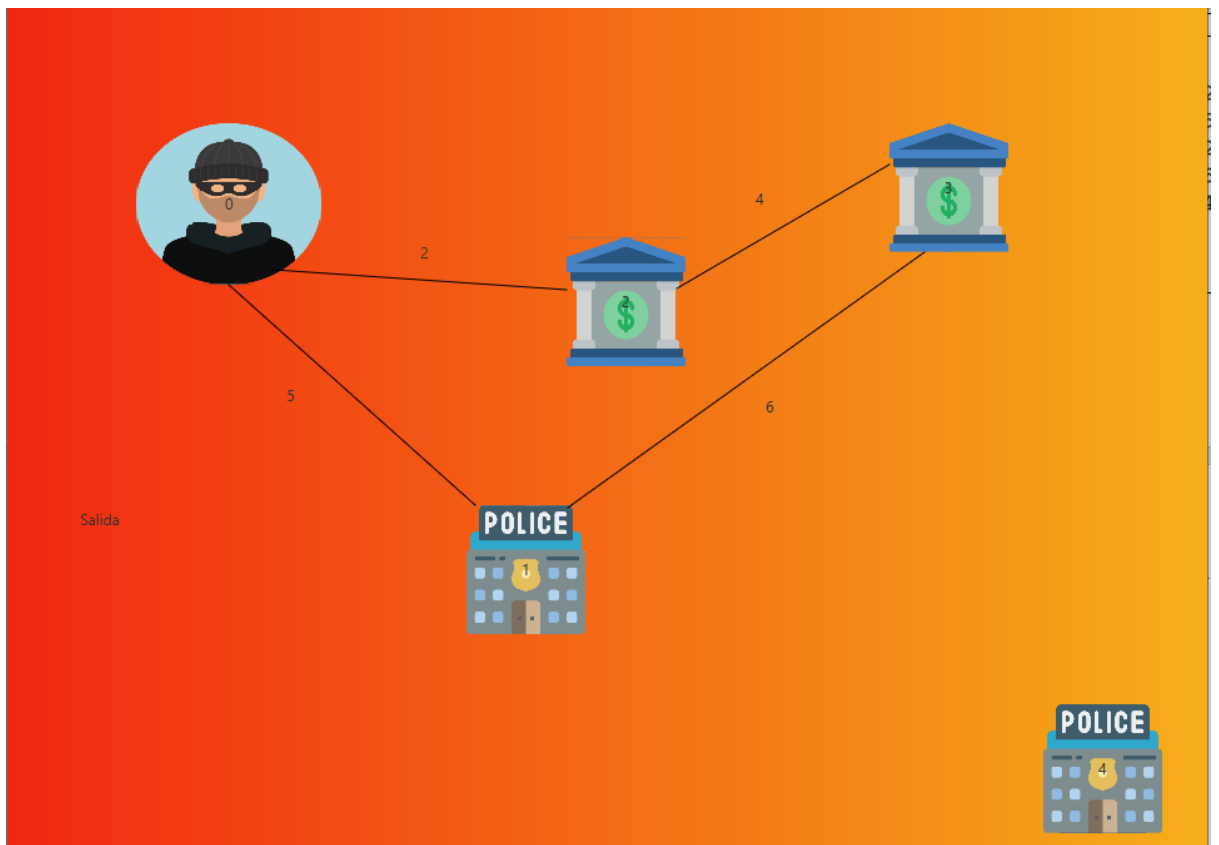




Objetivo de la prueba	Se prueba el caso 2			
Clase	método	escenario	entrada	resultado
Solucion	Case2	Setup2	<pre> 5 4 2 1 0 1 5 0 2 2 1 3 6 2 3 4 1 4 2 </pre>	<pre> 1 * 4 </pre>



1 \*  
4



Objetivo de la prueba	Se prueba el caso 3			
Clase	método	escenario	entrada	resultado
Solucion	Case3	Setup3	5 6 2 2 0 1 5 0 2 2 1 3 6 1 4 1 2 3 4 3 4 3 1 4 2 3	1 4 1

# TAD GRAFO

El tipo abstracto de datos (TAD) grafo está definido como sigue:

- Grafo () crea un grafo nuevo y vacío.
- agregarVertice(vert) agrega una instancia de Vertice al grafo.
- agregar Arista (deVertice, aVertice) agrega al grafo una nueva arista dirigida que conecta dos vértices.
- agregar Arista (de Vértice, a Vértice, ponderación) agrega al grafo una nueva arista ponderada y dirigida que conecta dos vértices.
- obtenerVertice(claveVert) encuentra el vértice en el grafo con nombre claveVert.
- obtener Vértices () devuelve la lista de todos los vértices en el grafo. Y devuelve True para una instrucción de la forma vertice in grafo, si el vértice dado está en el grafo, False de lo contrario.

TAD GRAFO		
<div> <div> <p>Grafo no dirigido</p> </div> <div> <p>Grafo dirigido</p> </div> </div>		
Invariante		
Operaciones primitivas		
Operación	Entradas	Salida
Crear Grafo	Vértice	Grafo
Añadir vértice	Vértice, Vértice	Grafo
Añadir arista con peso	Vérticeinicial, Vérticefinal, int	Grafo
Añadir arista	Vértice	vértice
Obtener Vértice	Grafo	Lista de todos los vértices del grafo
Borrar vértice	Vértice	Boolean
Borrar arista	VérticeInicial, vértice Final	Boolean
Son adyacentes	Vértice1, vértice2	Boolean

Nombre	Crear grafo (vértice)
Descripción	Crea un grafo con determinadas aristas y vértices adyacentes entre sí.
Precondición	No hay precondición
Postcondición	Se crea un grafo

Nombre	Agregar vértice (vértice)
Descripción	Agrega un vértice al grafo
Precondición	No hay precondición
Postcondición	Se agrega un vértice en la estructura del grafo.

Nombre	Añadir una arista (vértice, vértice)
Descripción	Agrega una arista(conexión) entre dos vértices determinados.
Precondición	Existir al menos 2 o más vértices en el grafo para conectarlos
Postcondición	Se crea una arista, la cual está conectada con un par de vértices determinados.

Nombre	Añadir arista con peso (vértice, vértice, peso(int))
Descripción	Crea un grafo con determinadas aristas y vértices adyacentes entre sí, con un peso definido.
Precondición	El peso no es número negativo. El peso es un número entero Deben existir al menos 2 vértices en el grafo
Postcondición	Se crea una arista la cual se relaciona con los dos vértices, está tiene un peso determinado.

Nombre	Obtener vértice (grafo)
Descripción	Se obtiene un vértice en específico o el conjunto de vértices en el grafo
Precondición	Deben existir un grado con n vértices
Postcondición	Se retorna una lista de vértices existentes en el grafo

Nombre	Borrar vértice (vértice)
Descripción	Se borra el vértice especificado
Precondición	Debe existir el vértice a borrar
Postcondición	Se actualizan los vértices del grafo

Nombre	Borrar arista (vértice, vértice)
Descripción	Elimina la arista o conexión que hay entre dos vértices.
Precondición	Deben existir los vértices y la arista que se eliminará.
Postcondición	Se elimina la arista que relaciona los vértices especificados.

Nombre	Son adyacentes (vértice, vértice)
Descripción	Determina si los dos vértices especulados se relacionan por una arista.
Precondición	La estructura tiene vértices y aristas.
Postcondición	Se válida si el par de vértices son adyacentes o no.





