Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

Primera entrega proyecto final

Identificación de los problemas:

Enunciado 1 – Ejercicio UVa (558 – Wormholes)

En el año 2163, los agujeros de gusano fueron descubiertos. Un agujero de gusano es un túnel subespacial que conecta 2 sistemas estelares a través del espacio y el tiempo. Los agujeros de gusano tienen unas cuantas propiedades:

- Los agujeros de gusano solo tienen una dirección
- El tiempo que lleva viajar a través de un agujero de gusano es insignificante
- Un agujero de gusano tiene dos puntos finales cada uno situado en un sistema estelar
- Un sistema solar puede tener más de un agujero de gusano
- Por alguna razón desconocida, empezando desde nuestro sistema solar, siempre es posible terminar en cualquier otro sistema estelar siguiendo una secuencia de agujeros
- Entre cada par de sistemas estelares, hay al menos un agujero de gusano en cualquier dirección
- No hay agujeros de gusano que terminen en el mismo sistema estelar en el que empezaron

Todos los agujeros de gusano tienen una diferencia de tiempo constante entre sus dos puntos finales. Por ejemplo, un agujero de gusano específico puede hacer que una persona viaje 15 años en el futuro. Otro agujero de gusano puede hacer que la persona viaje 42 años en el pasado.

Una brillante física, viviendo en la tierra, quiere usar los agujeros de gusano para estudiar el Big Bang. Desde que el "Warp drive" no ha sido inventado aún, no es posible para ella viajar de un sistema estelar a otro directamente. Esto puede hacerse viajando a través de agujeros de gusano, por supuesto.

La científica quiere alcanzar un ciclo de agujeros de gusano en alguna parte del universo que la puedan hacer terminar en el pasado. A partir de viajar en este círculo repetidas veces, la científica podrá viajar hacia el pasado cuanto tiempo considere necesario para ver el inicio del universo y ver el Big Bang con sus propios ojos.

Requerimientos funcionales - Wormholes

1. Agregar sistemas solares además de la tierra que tengan como identificador único un número entero diferente de 0.
2. Agregar agujeros de gusano entre sistemas estelares diferentes, con un punto inicial, un punto final y una constante de tiempo de viaje.

3. Calcular si es posible viajar en el tiempo de manera indefinida dados:
    - La cantidad de sistemas estelares que hay
    - La cantidad de agujeros de gusano con los siguientes datos:
        - Punto inicial
        - Punto final
        - Constante de tiempo

Enunciado 2 - Ejercicio UVa(1148 - The mysterious X network)

Una de las razones por las cuales el École polytechnique está tan profundamente arraigado en la sociedad francesa es por su famosa red de camaradas - Alumnos de la misma escuela. Cuando un camarada quiere algo (dinero,trabajo, etc), puede preguntar en esta red para buscar ayuda o apoyo. En la práctica, esto significa que cuando el/ella busca otro camarada, no siempre del mismo año, seguramente encontrará camaradas intermediarios para llegar a él o ella. Note que la relación entre los camaradas es simétrica. Gracias a la magia de la red X siempre hay manera de llegar a cualquier camarada desde cualquier camarada.

Requerimientos funcionales - The mysterious X network

1. Crear una red de camaradas
2. Agregar camaradas a la red identificados con un número entero, con una lista de conexiones a otros camaradas
3. Encontrar el camino más corto de un camarada a otro

Recopilación de información:

Ambos problemas requieren una estructura que pueda modelar nodos que contienen una información (Agujeros de gusano o personas) y que están conectados entre sí representando una relación entre ellos. La mejor manera representar los problemas sería a través de grafos, se deben evaluar los posibles grafos para modelar los problemas:

- Grafo simple
- Grafo dirigido
- Grafo simple ponderado
- Grafo dirigido ponderado

Grafo simple:

Es un conjunto de objetos llamados vértices unidos por enlaces llamados aristas, que permiten representar relaciones binarias simétricas entre los vértices del conjunto.

Grafo dirigido:

Este cumple las mismas características del grafo simple a excepción que la relación entre sus nodos no es simétrica, es decir puede haber una arista que relacione de un vértice v1 a un vértice v2 pero no necesariamente habrá una relación del vértice v2 a v1, como sí pasa en el grafo simple.

Grafo simple ponderado:

Cumple todas las características del grafo simple pero ahora las aristas entre cada par de vértices tienen un atributo peso que puede indicar el costo que conlleva recorrer ese camino.

Grafo simple ponderado:

Cumple todas las características del grafo dirigido pero ahora las aristas entre cada par de vértices tienen un atributo peso que puede indicar el costo que conlleva recorrer ese camino.

Una vez modelado el mundo del problema se necesita resolver el problema en si, para eso se necesita recorrer todos los caminos posibles entre los pares de vértices, los algoritmos que pueden hacer esto son:

- Breadth First Search
- Dijkstra algorithm
- Bellman Ford algorithm

Breadth First Search:

Es un algoritmo que sirve para recorrer grafos, generará un árbol que muestra el camino más corto desde un vértice hasta todos los otros nodos del grafo, no tendrá en cuenta los pesos de las aristas por lo que se suele usar sobre todo en grafos no ponderados.

Dijkstra algorithm:

Encuentra el camino más corto entre un vértice raíz y el resto de vértices del grafo teniendo en cuenta los pesos de las aristas. No funciona con pesos negativos.

Tiene una complejidad temporal de $O(|E| + |V| \log |V|)$

Bellman ford algorithm

Encuentra el camino más corto entre un vértice raíz y el resto de vértices del grafo teniendo en cuenta los pesos de las aristas. En caso de haber ciclos de pesos negativos en el grafo los reporta.

Tiene una complejidad temporal de Θ (| V | | E |)

Búsqueda de soluciones - Problema 1 Wormholes:

El problema uno requiere modelar un grafo que contiene relaciones unidireccionales entre pares de vértices en donde cada relación tendrá un peso que representa el tiempo que tomará recorrer ese camino.

Para resolver el problema en cuestión se debe encontrar si hay ciclos negativos en el grafo para determinar si es posible viajar en el tiempo indefinidamente o no.

Las posibles soluciones serían:

- Grafo dirigido ponderado, Bellman Ford
- Grafo dirigido ponderado, Dijkstra
- Grafo simple ponderado, Dijkstra
- Grafo simple ponderado, Bellman Ford

Transición de la formulación de ideas a los diseños preliminares:

Debido a que los agujeros de gusanos tendrán relaciones unidireccionales se descartaran todas las soluciones que no sean con grafo dirigido ponderado.

Dejando restantes las siguiente soluciones:

- Grafo dirigido ponderado, Dijkstra
- Grafo dirigido ponderado, Bellman Ford

Evaluación y selección de la mejor solución:

Los criterios a evaluar serán la eficacia y la eficiencia con la que los algoritmos planteados (Dijkstra y Bellman Ford) resuelven el problema.

Por lo tanto los criterios serán los siguientes:

- Complejidad temporal
- Dan solución exacta al problema planteado

Ambos algoritmos pueden hallar recorrido mínimo entre un vértice y el resto de vértices en el grafo. Dijkstra como ya se sabe tiene una complejidad temporal menor a Bellman Ford.

No obstante, Dijkstra no soporta ciclos negativos mientras que Bellman Ford si, por lo que Bellman Ford es el único algoritmo capaz de resolver el problema entre los evaluados.

Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

Búsqueda de soluciones - Problema 2 The mysterious X network:

El problema dos requiere modelar un grafo conexo que contiene relaciones simétricas entre pares de vértices.

Para resolver el problema en cuestión se debe hallar el camino más corto entre un vértice raíz y un vértice objetivo.

Las posibles soluciones serían:

- Grafo simple, Dijkstra
- Grafo simple, BFS
- Grafo simple, Bellman ford
- Grafo simple, Dijkstra
- Grafo simple, BFS
- Grafo dirigido, Bellman ford

Transición de la formulación de ideas a los diseños preliminares:

Debido a que las relaciones entre cada par del vértice del grafo están planeadas para ser simétricas, es muchas más eficiente para el programador usar un grafo simple pues así solo tendrá que agregar una arista para representar la relación bidireccional, en caso de ser un grafo dirigido tendría que agregar 2 aristas por par de vértices. Por lo anterior se descartan las soluciones que impliquen usar grafos dirigidos.

Dejando restantes las siguientes soluciones:

- Grafo simple, Dijkstra
- Grafo simple, Bellman Ford
- Grafo simple, BFS

Evaluación y selección de la mejor solución:

Los criterios a evaluar serán la eficacia y la eficiencia con la que los algoritmos planteados (Dijkstra, Bellman Ford y BFS) resuelven el problema.

Por lo tanto los criterios serán los siguientes:

- Complejidad temporal
- Dan solución exacta al problema planteado

La complejidad temporal de BFS es deO(|V| + |E|)

La complejidad temporal de Bellman Ford es de $\Theta (| V | | E |)$

La complejidad temporal de Dijkstra es de $O (| E | + | V | \log | V |)$

Por lo tanto el más eficiente en cuanto a complejidad temporal es BFS. Ahora bien, todos los algoritmos cumplen la misma tarea, encontrar el camino más corto entre un vértice raíz y el resto de los vértices del Grafo a excepción de que Dijkstra y Bellman Ford son usados en Grafos ponderados y BFS solo en grafos no ponderados.

Por lo anterior se concluye que, debido a que el algoritmo BFS cumple con justo lo necesario para resolver el problema y lo hace en el menor tiempo posible es el mejor algoritmo para llegar a la solución.

Diseño de estructuras de datos genéricas:

TAD Grafo		
		
{inv: |g.E| <= |g.V|*(|g.V|-1)		
Operaciones:		
crearGrafo		-> Grafo
agregarVertice	Grafo, Vertice	->Grafo
agregarArista	Grafo, Vertice, Vertice,boolean	->Grafo
agregarArista	Grafo, Vertice,Vertice,boolean, double	->Grafo
eliminarVertice	Grafo, Vertice	->Grafo
BFS	Grafo, Vertice	->Grafo
DFS	Grafo,Vertice	->Grafo
eliminarArista	Grafo,Vertice,Vertice	->Grafo
numeroVertices	Grafo	->int
numeroAristas	Grafo	->int
darPesoArista	Grafo,Vertice,Vertice	->double
modificarPesoArista	Grafo,Vertice,Vertice,double	->double
darVertices	Grafo	-> List<Vertice>
darVertice	Grafo	->

Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

esDirigido	Grafo	-> boolean
Dijsktra	Grafo	->Grafo
Bellmanford	Grafo	->Grafo, boolean
FloydWarshall	Grafo	->int[][], Grafo
Prim	Grafo	-> Grafo
Kruskal	Grafo	->Grafo

Grafo()
Crea un grafo vacío, sin vértices ni aristas
pre:
post: Grafo={V{},E{}}

agregarVertice(Grafo g, Vertice v)
Agrega un vértice a el grafo
pre: $v \notin g.V \land g \rightarrow$ estaInicializado
post: $v \in g.V$

agregarArista(Grafo g, Vértice v1, Vértice v2, boolean esDirigido)
Agrega una arista entre 2 vértices del grafo, si es dirigido entonces la relación solo queda de v1 a v2, caso contrario la relación es mutua v1 a v2 y v2 a v1. Se asume peso = 1.
pre: $v1 \in g.V \land v2 \in g.V$
post: $e1=\{v1,v2,1\} \land e2=\{v2,v1,1\} \in g.E$ si esDirigido = false $\lor$ $e = \{v1,v2,1\} \in g.E$ si esDirigido = true.

agregarArista(Grafo g, Vértice v1, Vértice v2,boolean esDirigido, double peso)
Agrega una arista entre 2 vértices del grafo, si es dirigido entonces la relación solo queda de v1 a v2, caso contrario la relación es mutua v1 a v2 y v2 a v1. Peso de la arista es igual a peso
pre: $v1 \in g.V \land v2 \in g.V$
post: $e1=\{v1,v2,peso\} \land e2=\{v2,v1,peso\} \in g.E$ si esDirigido = false $\lor$ $e = \{v1,v2,peso\} \in g.E$ si esDirigido = true.

eliminarVertice(Grafo g, Vertice v)
Elimina un vértice del grafo
pre: $v1 \in g.V$
post: $v1 \notin g.V$

BFS(Grafo g, Vertice v)
Realiza el algoritmo Breadth First Search en el grafo g a partir del vértice v
pre: $v \in g.V$
post: añade atributos vi.pred y vi.d para todo vértice conexo en el grafo g

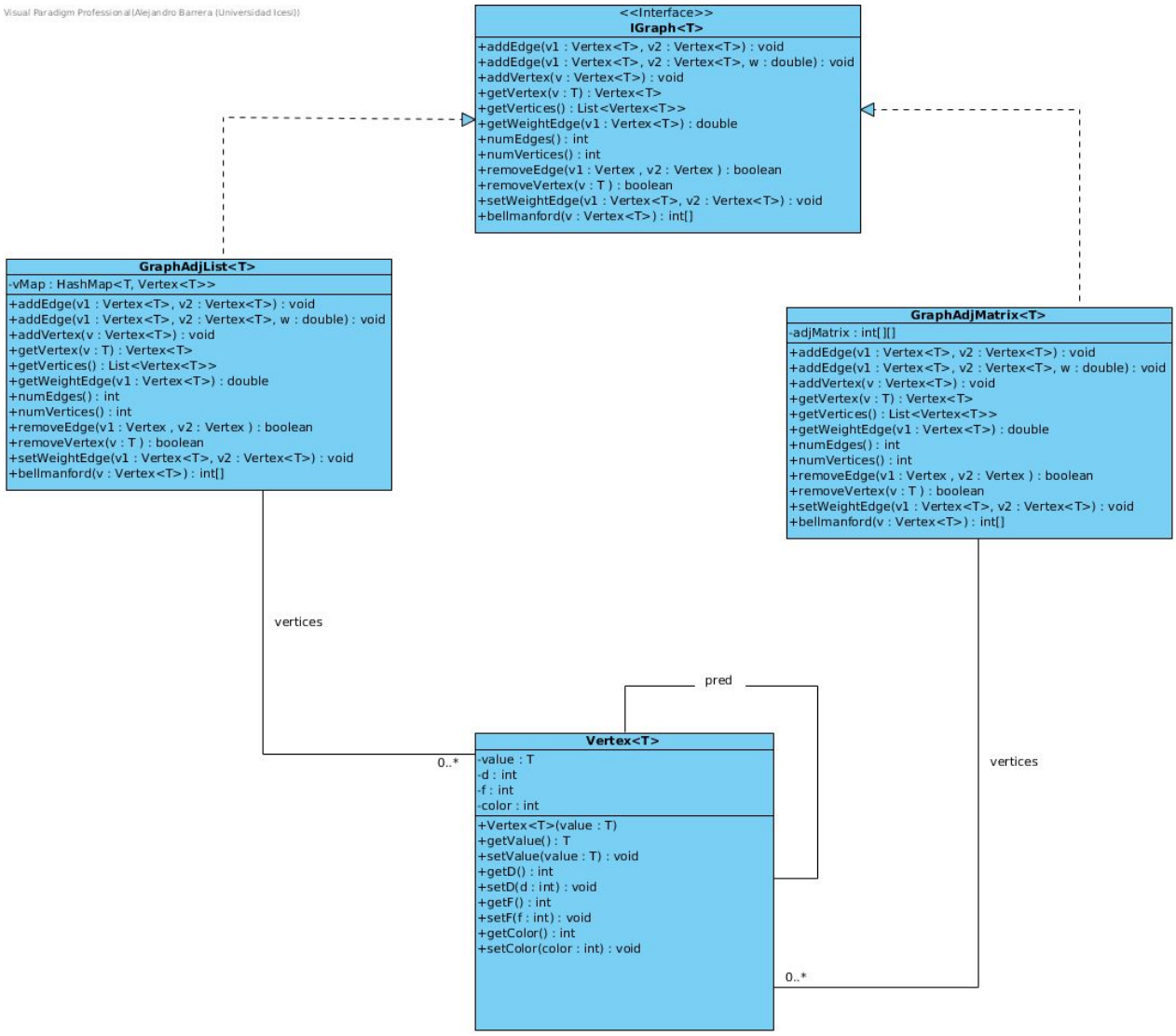
DFS(Grafo g, Vertice v)
Realiza el algoritmo Depth First Search

pre: v ∈ g.V post: añade atributos vi.pred y vi.d y vi.f para todo vértice conexo en el grafo g
eliminarArista(Grafo g, Vertice v1, Vertice v2) Elimina la arista e = {v1,v2} pre: e ∈ g.E post: e ∉ g.E
numeroVertices(Grafo g) Retorna el numero de vertices en el grafo pre: post:
numeroAristas(Grafo g) Retorna el número de aristas en el grafo pre: post:
darPesoArista(Grafo g, Vertice v1, Vertice v2) Retorna el peso de la arista que va desde v1 hasta v2 pre: v1,v2 ∈ g.V post:
modificarPesoArista(Grafo g, Vertice v1, Vertice v2, double peso) Modifica el peso de la arista que va desde v1 hasta v2 pre: v1,v2 ∈ g.V post: e = {v1,v2,peso}, e ∈ g.E
darVertices(Grafo g) Retorna la lista de vertices que tiene el grafo g pre: g != NIL post: List<Vertice> = {v1,v2,....,vn} ∈ g.V
darVertice(Grafo g, E element) Retorna un vertice del grafo g buscandolo por su identificador pre: g != NIL post: v ∈ g.V , v.value = element
Dijsktra(Grafo g, Vértice v1) Encuentra el camino mas corto de un vértice a todos los demas vertices del arbol pre g != NIL , v1 ∈ g.V post: añade atributos vi.pred y vi.d para todo vértice conexo en el grafo g
Bellmanford(Grafo g, Vértice v1) Encuentra el camino mas corto de un vértice a todos los demas vertices del arbol, soporta pesos negativos a diferencia de Dijsktra Retorna falso en caso de encontrar un ciclo negativo

Retorna true en caso contrario pre: g != NIL, v1 ∈ g.V post: añade atributos vi.pred y vi.d para todo vértice conexo en el grafo g **FloydWarshall(Grafo g)** Encuentra el camino minimo entre cada par de vertices conexos en el grafo pre: g != NIL post: Matriz de adjacencia del grafo que contiene la distancia entre vertices
Prim(Grafo g, Vértice v1) Encuentra el arbol de recubrimiento minimo del grafo g pre: g != NIL, v1 ∈ g.V, esConexo(g) = true post: añade atributos vi.pred para todo vértice conexo en el grafo g
Kruskal(Grafo g) Encuentra el bosque de recubrimiento minimo del grafo g, pre: g != NIL, v1 ∈ g.V post: añade atributos vi.pred para todo vértice conexo en el grafo g

Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

Diagrama de clases



Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

Diseño de pruebas unitarias

Prueba 1: Verifica si el metodo agregarVertice de la clase grafo funciona correctamente

Clase	**Método**	**Escenario**	**Entrada**	**Salida**
Grafo	+addVertex(Vertex):void	Grafo vacio	Vértice con valor 4	El grafo tiene un solo vértice con valor 4
Grafo	+addVertex(Vertex):void	Grafo con un conjunto de vértices V = { 10, 2, 5}	Vértice con valor 6	El grafo ahora tiene un conjunto de vértices Vx ={10,2,5,6}

Prueba 2: Verifica que el método agregarArista funcione correctamente

Grafo	+addVertex(Vertex,Vertex,boolean):void	Grafo con 2 vértices v1 = 1 v2 = 2 g.V = { v1, v2}	Arista entre v1 y v2 sin peso y no dirigida esDirigido = false	Hay una arista e1 = (1,2,1) e2 =(2,1,1) peso siendo igual a 1
Grafo	+addVertex(Vertex,Vertex,double,boolean):void	Grafo con 2 vértices v1 = 1 v2 = 2 g.V = {1,2}	Arista no dirigida entre v1 y v2 con peso w = 5 esDirigido = false	Hay una arista e1 =(1,2,5) e2 =(2,1,5)
Grafo	+addVertex(Vertex,Vertex,double,boolean): void	Grafo con 2 vértices v1 = 1 v2 = 2 g.V = {1,2}	Arista dirigida entre v1 y v2 con peso w = 5 esDirigido = true	Hay una arista e =(1,2,5)

Prueba 3: Verifica que el método eliminarVertice funcione correctamente

Grafo	+removeVertex(T):void	Grafo con 1 vértice v1 = 4	Eliminar vértice con valor = 4 t = 4	Grafo sin vértices
Grafo	+removeVertex(T):void	grafo no dirigido no ponderado con 2 vértices v1 = 4 v2 = 2 e1 = (4,2,1) e2 = (2,4,1)	Eliminar vértice con valor 4 T t = 4	Grafo que contiene únicamente v2 con ninguna arista asociada

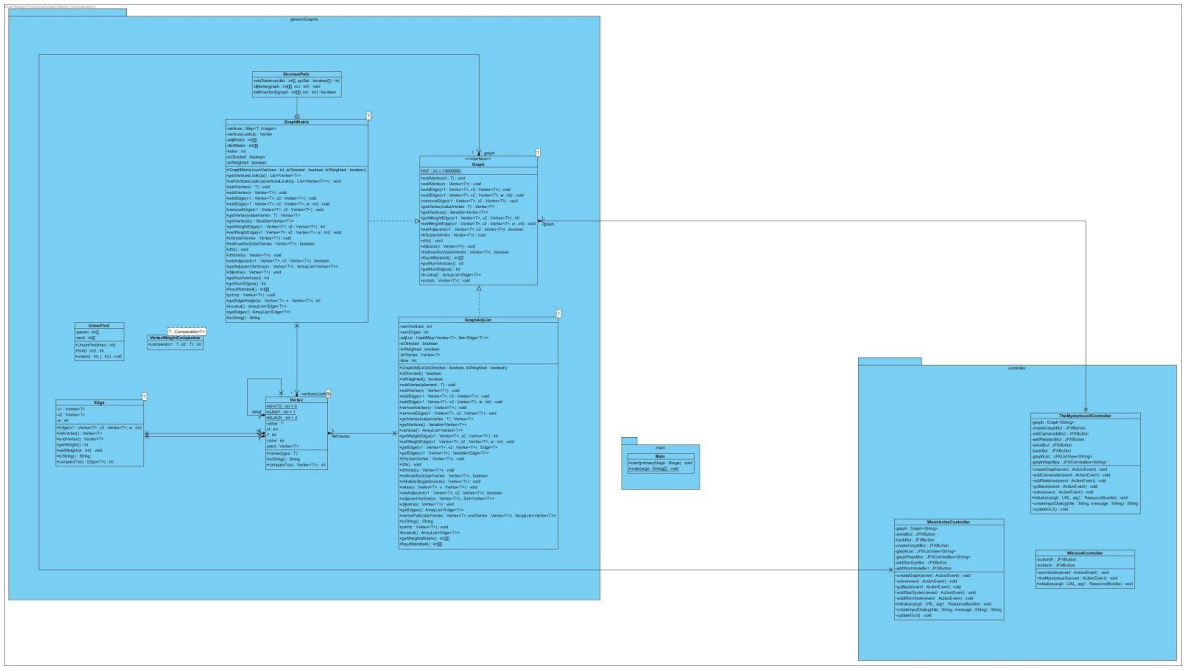
Prueba 4: Verifica que el metodo eliminar arista funcione correctamente

Grafo	+removeEdge(Vertex,Vertex):void	Grafo con 2 vértices y una arista que los relaciona v1 = 1 v2 = 4 e = (1,4,1)	Eliminar arista entre v1 y v2	Grafo con 2 vértices no conexos
Grafo	+removeEdge(Vertex,Vertex):void	Grafo con 3 vértices noDirigido = true v1 = 2 v2 = 4 v3 = 6 e1 = (2,4,1) e2=(4,2,1)	Eliminar arista entre v1 y v2	Grafo con 3 vértices no conexos
Grafo	+removeEdge(Vertex,Vertex):void	Grafo con 3 vértices noDirigido = false v1 = 2 v2 = 4 v3 = 6 e1 = (2,4,1) e2=(4,2,1)	Eliminar arista entre v1 y v2	Grafo con 3 vértices y una arista e1 = (4,2,1)

Prueba 5: Verificar que el recorrido BFS se esté realizando correctamente

Grafo	+bfs(Vértice):void	Grafo con 4 vértices 1,2,3,4 y 3 aristas e1 = (1,2,1) e2 =(2,3,1) e3 = (3,4,1)	Realizar recorrido bfs desde el vértice 3	Queda un árbol con raíz 3, qué recorrido en preorden se veria asi {3,2,4,1}G
Grafo	+bfs(Vértice):void	Grafo con 4 1,2,3,4 vértices y 3 aristas e1 = (1,2,1) e2 = (1,3,1) e3= (1,4,1)	Realizar recorrido bfs desde el vértice 3	Se recorre el grafo, la distancia entre 3 y 1 es 1, entre 3 y 2 es 2 y entre 3 y 4 es 2

Diagrama de clases de la solución



El diagrama se puede encontrar en la carpeta “Docs” dentro de la entrega del proyecto

Alejandro Barrera Lozano
Algoritmos y Estructuras de datos

Nota: Hay 2 soluciones diferentes por cada problema planteado, Las soluciones mandadas a al juez online, que no usan clases genéricas y reciben y dan información por consola (se encuentran en la carpeta “uva Solutions” programadas tanto en java como en python)

Las soluciones que usan clases genéricas e interfaz grafica en javaFX están en la carpeta “project”.

Bibliografía:

UVa - Wormholes

https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=499

UVa - The mysterious X network

https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=3589