Agregar:

boolean addVertex(V v);

caso 1:

entrada: v de tipo V que no se encuentra en el grafo.

salida: true

caso 2:

entrada: v de tipo V que ya se encuentra en el grafo.

salida: false

boolean addEdge(V u, V v);

caso 1:

entrada: u y v, de la misma clase. grafo no ponderado. u no pertenece al grafo.

salida: ElementNotFoundException

caso 2:

entrada: u y v, de la misma clase. grafo no ponderado. v no pertenece al grafo.

salida: ElementNotFoundException

caso 3:

entrada: u y v, de la misma clase. grafo no ponderado. Grafo no dirigido.

salida: true

caso 4:

entrada: u y v, de la misma clase. grafo no ponderado. Grafo dirigido.

salida: true

caso 5:

entrada: u y v iguales y de la misma clase. grafo no ponderado. Grafo no dirigido.

salida: true

caso 6:

entrada: u y v iguales y de la misma clase. grafo ponderado. Grafo no dirigido.

salida: WrongEdgeTypeException

caso 7:

entrada: u y v iguales y de la misma clase. grafo no ponderado. Grafo dirigido.

salida: true

boolean addEdge(V u, V v, double w);

caso 1:

entrada: u y v, de la misma clase. grafo ponderado. peso w que los une. u no pertenece al grafo.

salida: ElementNotFoundException

caso 2:

entrada: u y v, de la misma clase. grafo ponderado. peso w que los une. v no pertenece al grafo.

salida: ElementNotFoundException

caso 3:

entrada: u y v, de la misma clase. Grafo no ponderado. peso w que los une.

salida: WrongEdgeTypeException

caso 4:

entrada: u y v, de la misma clase. Grafo ponderado y dirigido. peso w que los une.

salida: true

caso 5:

entrada: u y v, de la misma clase. Grafo ponderado y no dirigido. peso w que los une.

salida: true

caso 6:

entrada: u y v, de la misma clase. U y v iguales. grafo ponderado. Grafo no dirigido.

salida: true

eliminar:

boolean removeVertex(V v);

caso 1: v no esta en el grafo.

Entrada: v

Salida: false

caso 2: v esta en el grafo.

Entrada: v

Salida: true

boolean removeEdge(V u, V v);

caso 1: no existe un borde de u a v. u, v pertenecen al grafo. Grafo dirigido.

Entrada: u, v

Salida: false

caso 2: no existe un borde entre u y v. u, v pertenecen al grafo. Grafo no dirigido.

Entrada: u, v

Salida: false

caso 3: existe un borde entre u y v. u, v pertenecen al grafo. Grafo no dirigido.

Entrada: u, v

Salida: true

caso 4: existe un borde de u a v. u, v pertenecen al grafo. Grafo dirigido.

Entrada: u, v

Salida: true

caso 5: u no pertenece al grafo.

Entrada: u, v

Salida: Element not found exeption

caso 6: v no pertenece al grafo.

Entrada: u, v

Salida: Element not found exeption

consultar:

List<V> vertexAdjacent(V u);

caso 1: u no pertenece al grafo.

Entrada: u

Salida: ElementNotFoundException

caso 2: u pertenece al grafo y tiene por lo menos un vertice adyaciente

Entrada: u

Salida: List<V> adjacentVertices

caso 3: u pertenece al grafo y no tiene vertices adyacentes

Entrada: u

Salida: null

boolean areConnected(V u, V v);

caso 1: u no pertenece al grafo.

Entrada: u, v

Salida: ElementNotFoundException

caso 2: v no pertenece al grafo.

Entrada: u, v

Salida: ElementNotFoundException

caso 3: u y v estan conectados.

Entrada: u, v

Salida: true

caso 4: u y v no estan conectados.

Entrada: u, v

Salida: false

double[][] weightMatrix();

caso 1: la matriz se retorna correctamente

Entrada: void

Salida: double[][] matrizAdyacencia

boolean isDirected();

caso 1: la matriz es dirigida

Entrada: void

Salida: true

caso 2: la matriz es no dirigida

Entrada: void

Salida: false

int getIndex(V u);

caso 1: u pertenece al grafo

Entrada: u

Salida: int index

caso 2: u no pertenece al grafo

Entrada: u

Salida: ElementNotFoundException

int getVertexSize();

caso 1: el grafo tiene al menos un vertice

Entrada: void

Salida: int vertices

caso 2: el grafo no tiene vertices

Entrada: void

Salida: 0

Map<V, Integer> getVertices();

caso 1: el programa devuelve el mapa

Entrada: void

Salida: Map<V, Integer> vertices

ArrayList<Edge> getEdges();

caso 1: el programa devuelve la lista de bordes

Entrada: void

Salida: ArrayList<Edge> edges

Algorithms:

List<V> bfs(IGraph<V> g, V u) throws ElementNotFoundException

caso 1: u no pertenece al grafo

entrada: g, u

salida: ElementNotFoundException

caso 2: u pertenece al grafo

entrada: g, u

salida: List<V> list

List<V> dfs(IGraph<V> g, V u) throws ElementNotFoundException

v pertenece, v no per

caso 1: u no pertenece al grafo

entrada: g, u

salida: ElementNotFoundException

caso 2: u pertenece al grafo

entrada: g, u

salida: List<V> list

List<V> traversal(IGraph<V> g, V v, ICollection<V> ds) throws ElementNotFoundException

caso 1: v no pertenece al grafo

entrada: g, v, ds

salida: ElementNotFoundException

caso 2: v pertenece al grafo

entrada: g, v, ds

salida: List<V> list

double[][] dijkstra(IGraph<V> g, V s)

caso 1: s no pertenece al grafo

entrada: g, s

salida: ElementNotFoundException

caso 2: s pertenece al grafo

entrada: g, s

salida: double[][]

caso 3: g tiene pesos negativos

entrada: g, s

salida: WrongEdgeTypeException

void initializeSingleSource(int indexOfS, double[][] shortestPath, double[][] w)

caso 1: w tiene pesos negativos

entrada: indexOfS, shortestPath, w

salida: WrongEdgeTypeException

double[][] floydWarshall(IGraph<V> g)

caso 1: g no es ponderado

entrada: g

salida: WrongGraphType

int[] prim(IGraph<V> g, V s)

s no pertenece, g es dirigido, g es no ponderado

caso 1: g no es ponderado

entrada: g, s

salida: WrongGraphType

caso 2: g es dirigido

entrada: g, s

salida: WrongGraphType

caso 3: s no pertenece a g

entrada: g, s

salida: ElementNotFoundException

Set<Edge> kruskal(IGraph<V> g)

caso 1: g no es ponderado

entrada: g, s

salida: WrongGraphType

caso 2: g no es conexo

entrada: g, s

salida: WrongGraphType