Documentación de la tarea integradora 3

Martin Cifuentes, Gabriel Delgado y Felipe Barreto

Contenido

[Requerimientos 2](#_Toc87821212)

[Requerimientos no funcionales 2](#_Toc87821213)

[Diagrama de clases 2](#_Toc87821214)

[TAD 2](#_Toc87821215)

[TAD Graph 2](#_Toc87821216)

[TAD SimpleWeightedGraph 5](#_Toc87821217)

[TAD SimpleGraph 9](#_Toc87821218)

[TAD Vertex 13](#_Toc87821219)

[TAD Edge 14](#_Toc87821220)

[JUNIT 17](#_Toc87821221)

[Setup scenarios del modelo 17](#_Toc87821222)

[Diseño de casos de prueba del modelo 17](#_Toc87821223)

[Seup scenarios de las estructuras de datos 17](#_Toc87821224)

[Diseño de casos de prueba del modelo 17](#_Toc87821225)

Documentación

# Requerimientos

# Requerimientos no funcionales

# Diagrama de clases

# TAD

## TAD Graph

|  |
| --- |
| **TAD** Graph |
| Graph = {nodes=ArrayList<Vertex<T>>(); edges=ArrayList<Edge<T>>(); } |
| { inv: nodes.getSize() >= 0 && edges.getSize() >= 0} |
| Operaciones primitivas:  Graph  getNodes null -> ArrayList<Vertex<T>>  setNodes ArrayList<Vertex<T>> -> null  getEdges null -> ArrayList<Edge<T>>  setEdges ArrayList<Edge<T>> -> null  addVertex Vertex<T> -> null  addEdge Edge<T> -> null  searchByProp String -> Vertex<T>  deleteVertex T -> Vertex<T>  BFS Vertex<T> -> null  DFS null -> null  DFSVisit Vertex<T>, int -> void  Djikstra (Vertex<T>, Vertex<T>) -> ArrayList<Vertex<T>>  length Vertex<T>, Vertex<T> -> double  returnPos Vertex<T> -> int  createNodesWithProps null -> null  getNodesWithProps() null -> ArrayList<Vertex<T>>  FloydWarshall null -> double[][] |

|  |
| --- |
| Graph()  **“Constructora”**  “Constructor de la clase, crea un objeto con un arraylist de vértices y de aristas vacíos”  { pre: TRUE }    { post:  Graph = {nodes.getSize() == 0 && edges.getSize() == 0}} |

|  |
| --- |
| getNodes()  **“Analizadora”**  “Retorna la lista de vértices”  { pre: Graph = {nodes != null, edges != null}}    { post:  nodes} |
| setNodes(ArrayList<Vertex<T>> nodes)  **“Modificadora”**  “Asigna la lista de nodos enviada a través del parámetro al atributo ‘nodes’ ”  { pre: Graph={nodes != null, edges != null}}    { post:  Graph = {this.node = nodes, edges != null}} |

|  |
| --- |
| getEdges()  **“Analizadora”**  “Devuelve la lista de aristas”  { pre: Graph = {nodes != null, edges != null}}    { post:  edges} |

|  |
| --- |
| setEdges(ArrayList<Edge<T>> edges)  **“Modificadora”**  “Asigna la lista de aristas enviada a través del parámetro al atributo ‘edges’ ”  { pre: Graph ={nodes != null, edges != null}}    { post:  Graph = {nodes != null, this.edges = edges}} |

|  |
| --- |
| addVertex(Vertex<T> v)  **“Modificadora”**  “Añade un nuevo vértice a la lista de vértices”  { pre: Graph ={nodes != null, edges != null} && v != null}    { post:  Graph = {nodes.getSize() = nodes.getSize() + 1, edges != null} && nodes.get(nodes.getSize() - 1) == v } |

|  |
| --- |
| addEdge(Edge<T> e)  **“Modificadora”**  “Añade una nueva arista a la lista de aristas ”  { pre: Graph ={nodes != null, edges != null} && e != null}    { post:  Graph = {edges.getSize() = edges.getSize() + 1, nodes != null} && edges.get(edges.getSize() - 1) == e} |

|  |
| --- |
| searchByProp(String prop)  **“Analizadora”**  “Devuelve el primer vértice encontrado asociado al valor properties buscado”  { pre: Graph = {nodes ¡= null } }    { post:  Vertex<T> con properties.equals(prop) } |

|  |
| --- |
| deleteVertex(T el)  **“Modificadora”**  “Elimina el vértice buscado por el “el” de tipo T”  { pre: Graph = {nodes ¡= null } }    { post: nodo eliminado && Graph = {nodes.size() – 1 && edges}} |

|  |
| --- |
| BFS(Vertex<T> init)  **“Analizadora”**  “Retorna el análisis del Breadth First Search del grafo”  { pre: Graph ={nodes != null, edges != null}}  { post: retorna un árbol n-ario BFS } |

|  |
| --- |
| DFSVisit(Vertex<T> u, int time)  **“Analizadora”**  “Realiza el procedimiento de visitar un nodo y, posteriormente, hacer los llamados recursivos”  { pre: Graph = {nodes ¡= null } && u != null }    { post: Todos los adyacentes al nodo principal y los adyacentes a esos, visitados } |

|  |
| --- |
| DFS()  **“Analizadora”**  “Retorna el análisis del Depth First Search del grafo”  { pre: Graph ={nodes != null, edges != null}}    { post: retorna un árbol n-ario DFS } |

|  |
| --- |
| djikstra(Graph<T> g, Vertex<T> init, Vertex<T> end)  **“Analizadora”**  “Retorna el camino más corto entre el vértice inicial y el final”  { pre: { pre: Graph ={nodes != null, edges != null} && }}    { post:  retorna el camino más corto entre todo par de nodos } |

|  |
| --- |
| length(Vertex<T> u, Vertex<T> v)  **“Analizadora”**  “Devuelve el peso menor entre las aristas que conecten los dos nodos enviados por parámetros”  { pre: Graph = {nodes ¡= null && edges != null} && u != null && v != null }    { post: peso de la arista con menor peso entre ambos vértices } |

|  |
| --- |
| returnPos(Vertex<T> v)  **“Analizadora”**  “Devuelve la posición de un vértice buscado en el arraylist principal de nodos”  { pre: Graph = {nodes ¡= null } && v != null }    { post: posición del vértice v || -1} |

|  |
| --- |
| createNodesWithProps()  **“Modificadora”**  “Llena, a partir del arraylist de nodos, un arraylist secundario cuyos properties sean diferentes al string vacío.”  { pre: Graph = {nodes ¡= null && edges != null}}    { post: arraylist nodesWithProps modificado } |

|  |
| --- |
| getNodesWithProps()  **“Analizadora”**  “Devuelve el arraylist nodesWithProps”  { pre: Graph = {nodes ¡= null && edges != null} }    { post: nodesWithProps} |

|  |
| --- |
| floydWarshall(Graph<T> g)  **“Analizadora”**  “Retorna el análisis de los caminos más cortos entre cada par de vértices”  {pre: { pre: Graph ={nodes != null, edges != null}}}    { post:  retorna el camino más corto entre todo par de nodos } |

## TAD SimpleWeightedGraph

|  |
| --- |
| **TAD** SimpleWeightedGraph |
| SimpleWeightedGraph = {nodes=ArrayList<Vertex<T>>(); edges=ArrayList<Edge<T>>(); } |
| { inv: nodes.getSize() >= 0 && edges.getSize() >= 0} |
| Operaciones primitivas:  SimpleWeightedGraph  getNodes null -> ArrayList<Vertex<T>>  setNodes ArrayList<Vertex<T>> -> null  getEdges null -> ArrayList<Edge<T>>  setEdges ArrayList<Edge<T>> -> null  addVertex Vertex<T> -> null  addEdge Edge<T> -> null  searchByProp String -> Vertex<T>  deleteVertex T -> Vertex<T>  BFS Vertex<T> -> null  DFS null -> null  DFSVisit Vertex<T>, int -> void  Djikstra (Vertex<T>, Vertex<T>) -> null  length Vertex<T>, Vertex<T> -> double  returnPos Vertex<T> -> int  createNodesWithProps null -> null  FloydWarshall null -> null |

|  |
| --- |
| SimpleWeightedGraph ()  **“Constructora”**  “Constructor de la clase, crea un objeto con un arraylist de vértices y de aristas vacíos”  { pre: TRUE }    { post:  SimpleWeightedGraph = {nodes.getSize() == 0 && edges.getSize() == 0}} |

|  |
| --- |
| getNodes()  **“Analizadora”**  “Retorna la lista de vértices”  { pre: SimpleWeightedGraph = {nodes != null, edges != null}}    { post:  nodes} |
| setNodes(ArrayList<Vertex<T>> nodes)  **“Modificadora”**  “Asigna la lista de nodos enviada a través del parámetro al atributo ‘nodes’ ”  { pre: SimpleWeightedGraph ={nodes != null, edges != null}}    { post:  SimpleWeightedGraph = {this.node = nodes, edges != null}} |

|  |
| --- |
| getEdges()  **“Analizadora”**  “Devuelve la lista de aristas”  { pre: SimpleWeightedGraph = {nodes != null, edges != null}}    { post:  edges} |

|  |
| --- |
| setEdges(ArrayList<Edge<T>> edges)  **“Modificadora”**  “Asigna la lista de aristas enviada a través del parámetro al atributo ‘edges’ ”  { pre: SimpleWeightedGraph ={nodes != null, edges != null}}    { post:  SimpleWeightedGraph = {nodes != null, this.edges = edges}} |

|  |
| --- |
| addVertex(Vertex<T> v)  **“Modificadora”**  “Añade un nuevo vértice a la lista de vértices”  { pre: SimpleWeightedGraph ={nodes != null, edges != null} && v != null}    { post:  SimpleWeightedGraph = {nodes.getSize() = nodes.getSize() + 1, edges != null} && nodes.get(nodes.getSize() - 1) == v } |

|  |
| --- |
| addEdge(Edge<T> e)  **“Modificadora”**  “Añade una nueva arista a la lista de aristas ”  { pre: SimpleWeightedGraph ={nodes != null, edges != null} && e != null}    { post:  SimpleWeightedGraph = {edges.getSize() = edges.getSize() + 1, nodes != null} && edges.get(edges.getSize() - 1) == e} |

|  |
| --- |
| searchByProp(String prop)  **“Analizadora”**  “Devuelve el primer vértice encontrado asociado al valor properties buscado”  { pre: SimpleWeightedGraph = {nodes ¡= null } }    { post:  Vertex<T> con properties.equals(prop) } |

|  |
| --- |
| deleteVertex(T el)  **“Modificadora”**  “Elimina el vértice buscado por el “el” de tipo T”  { pre: SimpleWeightedGraph = {nodes ¡= null } }    { post: nodo eliminado && Graph = {nodes.size() – 1 && edges}} |

|  |
| --- |
| BFS(Graph<T> g,Vertex<T> init)  **“Analizadora”**  “Retorna el análisis del Breadth First Search del grafo”  { pre: SimpleWeightedGraph ={nodes != null, edges != null}}    { post: retorna un árbol n-ario BFS } |

|  |
| --- |
| DFS(Graph<T> g)  **“Analizadora”**  “Retorna el análisis del Depth First Search del grafo”  { pre: SimpleWeightedGraph ={nodes != null, edges != null}}    { post: retorna un árbol n-ario DFS } |

|  |
| --- |
| DFSVisit(Vertex<T> u, int time)  **“Analizadora”**  “Realiza el procedimiento de visitar un nodo y, posteriormente, hacer los llamados recursivos”  { pre: SimpleWeightedGraph = {nodes ¡= null } && u != null }    { post: Todos los adyacentes al nodo principal y los adyacentes a esos, visitados } |

|  |
| --- |
| Djikstra(Graph<T> g,Vertex<T> init, Vertex<T> end)  **“Analizadora”**  “Retorna el camino más corto entre el vértice inicial y el final”  { pre: { pre: SimpleWeightedGraph ={nodes != null, edges != null} && }}    { post:  retorna el camino más corto entre todo par de nodos } |

|  |
| --- |
| length(Vertex<T> u, Vertex<T> v)  **“Analizadora”**  “Devuelve el peso menor entre las aristas que conecten los dos nodos enviados por parámetros”  { pre: SimpleWeightedGraph = {nodes ¡= null && edges != null} && u != null && v != null }    { post: peso de la arista con menor peso entre ambos vértices } |

|  |
| --- |
| returnPos(Vertex<T> v)  **“Analizadora”**  “Devuelve la posición de un vértice buscado en el arraylist principal de nodos”  { pre: SimpleWeightedGraph = {nodes ¡= null } && v != null }    { post: posición del vértice v || -1} |

|  |
| --- |
| createNodesWithProps()  **“Modificadora”**  “Llena, a partir del arraylist de nodos, un arraylist secundario cuyos properties sean diferentes al string vacío.”  { pre: SimpleWeightedGraph = {nodes ¡= null && edges != null}}    { post: arraylist nodesWithProps modificado } |

|  |
| --- |
| getNodesWithProps()  **“Analizadora”**  “Devuelve el arraylist nodesWithProps”  { pre: SimpleWeightedGraph = {nodes ¡= null && edges != null} }    { post: nodesWithProps} |

|  |
| --- |
| FloydWarshall(Graph<T> g)  **“Analizadora”**  “Retorna el análisis de los caminos más cortos entre cada par de vértices”  {pre: { pre: SimpleWeightedGraph ={nodes != null, edges != null}}}    { post: retorna el camino más corto entre todo par de nodos} |

## TAD SimpleGraph

|  |
| --- |
| **TAD** SimpleGraph |
| SimpleGraph = {nodes=ArrayList<Vertex<T>>(); edges=ArrayList<Edge<T>>(); } |
| { inv: nodes.getSize() >= 0 && edges.getSize() >= 0} |
| Operaciones primitivas:  SimpleGraph  getNodes null -> ArrayList<Vertex<T>>  setNodes ArrayList<Vertex<T>> -> null  getEdges null -> ArrayList<Edge<T>>  setEdges ArrayList<Edge<T>> -> null  addVertex Vertex<T> -> null  addEdge Edge<T> -> null  searchByProp String -> Vertex<T>  deleteVertex T -> Vertex<T>  BFS Vertex -> null  DFS null -> null  DFSVisit Vertex<T>, int -> void  Djikstra (Vertex, Vertex) -> null  length Vertex<T>, Vertex<T> -> double  returnPos Vertex<T> -> int  createNodesWithProps null -> null  floydWarshall null -> null |

|  |
| --- |
| SimpleGraph()  **“Constructora”**  “Constructor de la clase, crea un objeto con un arraylist de vértices y de aristas vacíos”  { pre: TRUE }    { post:  SimpleGraph = {nodes.getSize() == 0 && edges.getSize() == 0}} |

|  |
| --- |
| getNodes()  **“Analizadora”**  “Retorna la lista de vértices”  { pre: SimpleGraph = {nodes != null, edges != null}}    { post:  nodes} |
| setNodes(ArrayList<Vertex<T>> nodes)  **“Modificadora”**  “Asigna la lista de nodos enviada a través del parámetro al atributo ‘nodes’ ”  { pre: SimpleGraph={nodes != null, edges != null}}    { post:  SimpleGraph = {this.node = nodes, edges != null}} |

|  |
| --- |
| getEdges()  **“Analizadora”**  “Devuelve la lista de aristas”  { pre: SimpleGraph = {nodes != null, edges != null}}    { post:  edges} |

|  |
| --- |
| setEdges(ArrayList<Edge<T> edges)  **“Modificadora”**  “Asigna la lista de aristas enviada a través del parámetro al atributo ‘edges’ ”  { pre: SimpleGraph={nodes != null, edges != null}}    { post:  SimpleGraph = {nodes != null, this.edges = edges}} |

|  |
| --- |
| addVertex(Vertex<T> v)  **“Modificadora”**  “Añade un nuevo vértice a la lista de vértices”  { pre: SimpleGraph={nodes != null, edges != null} && v != null}    { post:  SimpleGraph = {nodes.getSize() = nodes.getSize() + 1, edges != null} && nodes.get(nodes.getSize() - 1) == v } |

|  |
| --- |
| addEdge(Edge<T> e)  **“Modificadora”**  “Añade una nueva arista a la lista de aristas ”  { pre: SimpleGraph={nodes != null, edges != null} && e != null}    { post:  SimpleGraph = {edges.getSize() = edges.getSize() + 1, nodes != null} && edges.get(edges.getSize() - 1) == e} |

|  |
| --- |
| searchByProp(String prop)  **“Analizadora”**  “Devuelve el primer vértice encontrado asociado al valor properties buscado”  { pre: Graph = {nodes ¡= null } }    { post:  Vertex<T> con properties.equals(prop) } |

|  |
| --- |
| deleteVertex(T el)  **“Modificadora”**  “Elimina el vértice buscado por el “el” de tipo T”  { pre: Graph = {nodes ¡= null } }    { post: nodo eliminado && Graph = {nodes.size() – 1 && edges}} |

|  |
| --- |
| BFS(Graph g, Vertex<T> init)  **“Analizadora”**  “Retorna el análisis del Breadth First Search del grafo”  { pre: SimpleGraph={nodes != null, edges != null}}    { post: retorna un árbol n-ario BFS} |

|  |
| --- |
| DFS(Graph<T> g)  **“Analizadora”**  “Retorna el análisis del Depth First Search del grafo”  { pre: SimpleGraph={nodes != null, edges != null}}    { post: retorna un árbol n-ario DFS} |

|  |
| --- |
| searchByProp(String prop)  **“Analizadora”**  “Devuelve el primer vértice encontrado asociado al valor properties buscado”  { pre: SimpleGraph = {nodes ¡= null } }    { post:  Vertex<T> con properties.equals(prop) } |

|  |
| --- |
| deleteVertex(T el)  **“Modificadora”**  “Elimina el vértice buscado por el “el” de tipo T”  { pre: SimpleGraph = {nodes ¡= null } }    { post: nodo eliminado && Graph = {nodes.size() – 1 && edges}} |

|  |
| --- |
| Djikstra(Graph <T> g,Vertex<T> init, Vertex<T> end)  **“Analizadora”**  “Retorna el camino más corto entre el vértice inicial y el final”  { pre: { pre: SimpleGraph={nodes != null, edges != null} && }}    { post:  retorna el camino más corto entre init y end} |

|  |
| --- |
| length(Vertex<T> u, Vertex<T> v)  **“Analizadora”**  “Devuelve el peso menor entre las aristas que conecten los dos nodos enviados por parámetros”  { pre: SimpleGraph = {nodes ¡= null && edges != null} && u != null && v != null }    { post: peso de la arista con menor peso entre ambos vértices } |

|  |
| --- |
| returnPos(Vertex<T> v)  **“Analizadora”**  “Devuelve la posición de un vértice buscado en el arraylist principal de nodos”  { pre: SimpleGraph = {nodes ¡= null } && v != null }    { post: posición del vértice v || -1} |

|  |
| --- |
| createNodesWithProps()  **“Modificadora”**  “Llena, a partir del arraylist de nodos, un arraylist secundario cuyos properties sean diferentes al string vacío.”  { pre: SimpleGraph = {nodes ¡= null && edges != null}}    { post: arraylist nodesWithProps modificado } |

|  |
| --- |
| getNodesWithProps()  **“Analizadora”**  “Devuelve el arraylist nodesWithProps”  { pre: SimpleGraph = {nodes ¡= null && edges != null} }    { post: nodesWithProps} |

|  |
| --- |
| FloydWarshall(Graph<T> g)  **“Analizadora”**  “Retorna el análisis de los caminos más cortos entre cada par de vértices”  {pre: { pre: SimpleGraph={nodes != null, edges != null}}}    { post: retorna el camino más corto entre todo par de nodos } |

## TAD Vertex

|  |
| --- |
| **TAD** Vertex |
| Vertex = {value= T; properties = String} |
| {inv: properties !=null } |
| Operaciones primitivas:  Vertex (T, String) -> Vertex  getValue null -> T  setValue T -> null  getProperties null -> String  setProperties String -> null |

|  |
| --- |
| Vertex(T, String)  **“Constructora”**  “Constructor de la clase, crea un objeto con un objeto y una cadena, el objeto indica el valor del nodo y la cadena sus propiedades o características”  { pre: (T, String) }    { post:  Vertex = {value != null ; properties != null}} |

|  |
| --- |
| getValue()  **“Analizadora”**  “Retorna el valor del vertice”  { pre: Vertex = { value != null }}    { post:  value} |
| setValue(T newValue)  **“Modificadora”**  “Asigna al valor del vertice, el valor enviado a través del parámetro al atributo ‘newValue”  { pre: Vertex = { value != null }}    { post:  Vertex = {this.value = newValue }} |
| getProperties()  **“Analizadora”**  “Retorna el valor de properties”  { pre: Vertex = { properties != null }}    { post:  properties } |
| setProperties(T newProperties)  **“Modificadora”**  “Asigna al valor del properties, el valor enviado a través del parámetro al atributo ‘newProperties”  { pre: Vertex = { properties!= null }}    { post:  Vertex = {this. properties = newProperties }} |

## TAD Edge

|  |
| --- |
| **TAD** Edge |
| Edge = {weight = int; origin = Vertex; destiny = Vertex;} |
|  |
| Operaciones primitivas:  Edge (Vertex<T>,Vertex<T>) -> Edge<T>  getWeight null -> int  setWeight int -> null  getOrigin Vertex<T -> null  setOrigin null -> Vertex<T  getDestiny Vertex<T> -> null  setDestiny null -> Vertex<T> |

|  |
| --- |
| Edge (Vertex<T>,Vertex<T>)  **“Constructora”**  “Constructor de la clase, crea un objeto con dos vertices, una indica el origen de la arista y la otra su destino”  { pre: (Vertex<T>,Vertex<T>) }    { post:  Edge = {origin != null; destiny != null}} |

|  |
| --- |
| getWeight()  **“Analizadora”**  “Retorna el valor del peso”  { pre: Edge = { weight!= null }}    { post:  value} |
| setWeight(int newWeight)  **“Modificadora”**  “Asigna al valor del peso de la arista, el valor enviado a través del parámetro al atributo ‘newWeight”  { pre: Edge = { weight != null }}    { post:  Edge = {this.weight = newWeight }} |
| getOrigin()  **“Analizadora”**  “Retorna el valor de origen”  { pre: Edge = { origin != null }}    { post:  origin } |
| setOrigin(Vertex<T> newOrigin)  **“Modificadora”**  “Asigna al valor de origin, el valor enviado a través del parámetro al atributo ‘newOrigin”  { pre: Edge = { origin != null }}    { post:  Edge = {this. origin = newOrigin }} |
| getDestiny()  **“Analizadora”**  “Retorna el valor de destino”  { pre: Edge = { destiny != null }}    { post:  properties } |
| setDestiny(Vertex<T> newDestiny)  **“Modificadora”**  “Asigna al valor de destiny, el valor enviado a través del parámetro al atributo ‘newDestiny”  { pre: Edge = { destiny != null }}    { post:  Edge = {this. destiny = newDestiny }} |

# JUNIT

## Setup scenarios del modelo

## Diseño de casos de prueba del modelo

## Seup scenarios de las estructuras de datos

|  |  |  |
| --- | --- | --- |
| Nombre | Clase | Escenario |
| setupScenario1 | VertexTest |  |
| setupScenario1 | EdgeTest |  |
| setupScenario1 | GraphTest |  |
| setupScenario2 | GraphTest |  |
| setupScenario3 | GraphTest |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el constructor de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| VertexTest | Constructor | setupScenario1 | value = 5;  properties = “a”; | Crea un objeto Vertex |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el constructor de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| VertexTest | Constructor | setupScenario1 | value = 5;  properties = “a”; | Crea un objeto Vertex |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el método agregar vertice de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| GraphTest | addVertex | setupScenario1 | vertex = new Vertex<Integer>(5,"a"); | Agrega un objeto Vertex |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el método agregar arista de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| GraphTest | addEdge | setupScenario1 | edge = new Edge<Integer>(5,"a"); | Agrega un objeto Edge |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el método Dijkstra de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| GraphTest | Djikstra | setupScenario2 | Vertex v= new Vertex<Integer>(0, "");  Vertex v= new Vertex<Integer>(4, "");  Donde los Vertex pertenece a un grafo específico | Encuentra el camino más efectivo para llega de un nodo a otro |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el constructor de la clase funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de entrada | Resultado |
| Edge Test | FloysWarshall | setupScenario3 | Vertex v= new Vertex<Integer>(2, "");  Donde el Vertex pertenece a un grafo específico | Encuentra el camino más efectivo para llega de un nodo a todos los demás |

## Diseño de casos de prueba del modelo