|  |
| --- |
| **TAD Graph<T>** |
| Graph = {V = {v1, v2, …, vn}, E = {e1 = (vi1, vj1, w1), e2 = (vi2, vj2, w2), em = (vim, vjm, wm)}, directed, weighted}  https://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Directed.svg/160px-Directed.svg.png = vk = ek |
| Inv:   1. ∀ek ∈ E, vik ∈ V ∧ vjk ∈ V, wk > 0 2. directed = false ⇒ () 3. weighted = false ⇒ ∀ek ∈ E, wk = 1 |
| Operaciones básicas   * Graph Boolean, Boolean → Graph * addVertex Graph x Vertex → Graph * addEdge Graph x Vertex x Vertex → Graph * addEdge Graph x Vertex x Vertex x Double → Graph * removeVertex Graph x Vertex → Graph * removeEdge Graph x Vertex x Vertex → Graph * getNeighbors Graph x Vertex → List<Vertex> * getNumberOfVertices Graph → Integer * getNumberOfEdges Graph → Integer * areAdjacent Graph x Vertex x Vertex → Boolean * isInGraph Graph x T → Boolean * getEdgeWeight Graph x Vertex x Vertex → Double * setEdgeWeight Graph x Vertex x Vertex x Double → Graph * getVertices Graph → List<Vertex> * searchVertex Graph x T → Vertex * isDirected Graph → Boolean * isWeighted Graph → Boolean * bfs Graph x Vertex → Graph * dfs Graph → Graph * dijkstra Graph x Vertex → Graph * floydwarshall Graph → Double[][] * prim Graph x Vertex → Graph * kruskal Graph → List<Edge> * getEdges Graph → List<Edge> |

Operaciones

|  |
| --- |
| **Graph(Boolean directed, Boolean weighted)**  “Crea un nuevo grafo que puede o no ser dirigido o ponderado”  Pre:  Post: Graph = {V={}, E={}, directed, weighted} |
| **addVertex(Graph g, Vertex v)**  “Inserta un vértice en el grafo”  Pre: v g.V  Post: v g.V |
| **addEdge(Graph g, Vertex x, Vertex y)**  “Añade una arista de peso 1 que va de x a y. Si el grafo no es dirigido, también la añade de y a x”  Pre: x, y g.V  Post: e = (x, y, 1) g.E. Si g.directed = false, e’ = (y, x, 1) ) g.E |
| **addEdge(Graph g, Vertex x, Vertex y, Double w)**  “Añade una arista de peso w que va de x a y. Si el grafo no es dirigido, también la añade dé y a x”  Pre: x, y g.V, g.weighted = true, w > 0  Post: e = (x, y, w) g.E. Si g.directed = false, e’ = (y, x, w) ) g.E |
| **removeVertex(Graph g, Vertex v)**  “Elimina a v del grafo”  Pre: v g.V  Post: v g.V. Todos los vértices que son incidentes con v g.E |
| **removeEdge(Graph g, Vertex x, Vertex y)**  “Elimina la arista que va de x a y en el grafo”  Pre: x,y g.V, (x, y, \*) g.E  Post: e = (x, y, \*) g.E. Si g.directed = false, e’ = (y, x, \*) ) g.E |
| **getNeighbors(Graph g, Vertex x)**  “Devuelve los vértices v tal que hay una arista desde x hasta v”  Pre: x g.V  Post: vertices = {v1,v2,…,vn} : ∀vi, (x, vi, \*) g.E. |
| **getNumberOfVertices(Graph g)**  “Devuelve el número de vértices en el grafo”  Pre:  Post: n = size(g.V) |
| **getNumberOfEdges(Graph g)**  “Devuelve el número de aristas en el grafo”  Pre:  Post: n = size(g.E) |
| **areAdjacent(Graph g, Vertex x, Vertex y)**  “Devuelve si hay una arista de x a y”  Pre: x,y g.V  Post: true si y solo si (x, y, \*) g.E. |
| **isInGraph(T val)**  “Devuelve si hay un vértice con el valor dado en el grafo”  Pre:  Post: true si y solo si x g.V : value(x) = val. |
| **getEdgeWeight(Graph g, Vertex x, Vertex y)**  “Devuelve el peso de la arista que va de x a y”  Pre: x,y g.V, (x, y, \*) g.E  Post: peso = (x, y).w |
| **setEdgeWeight(Graph g, Vertex x, Vertex y, Double w)**  “Cambia el peso de la arista que va de x a y”  Pre: x,y g.V, (x, y, \*) g.E, w > 0  Post: (x, y, w) g.E |
| **getVertices(Graph g)**  “Devuelve la lista de vértices del grafo”  Pre:  Post: {v1,v2,…,vn} = g.V |
| **searchVertex(T val)**  “Devuelve, si existe, el vértice con el valor dado en el grafo”  Pre:  Post: x g.V : value(x) = val. NIL si no existe. |
| **isDirected(Graph g)**  “Devuelve si el grafo es dirigido”  Pre:  Post: g.directed |
| **getWeighted(Graph g)**  “Devuelve si el grafo es ponderado”  Pre:  Post: g.weighted |
| **bfs(Graph g, Vertex s)**  “Realiza el algoritmo Breadth First Search, ajustando información para los vértices del grafo”  Pre: s g.V  Post: ∀u g.V, añade atributos u.pred y u.d, que corresponden a los añadidos por el algoritmo Breadth First Search |
| **dfs(Graph g)**  “Realiza el algoritmo Depth First Search, ajustando información para los vértices del grafo”  Pre:  Post: ∀u g.V, añade atributos u.pred, u.d y u.f, que corresponden a los añadidos por el algoritmo Depth First Search |
| **dijkstra(Graph g, Vertex s)**  “Realiza el algoritmo de Dijkstra, tomando como vértice inicial a s”  Pre: s g.V, g no tiene pesos negativos  Post: ∀u g.V, añade atributos u.pred y u.d, que corresponden respectivamente al predecesor y la distancia añadidos por el algoritmo de Dijkstra |
| **floydwarshall(Graph g)**  “Realiza el algoritmo de Floyd-Warshall sobre g”  Pre:  Post: Retorna la matriz dist, donde la posición [i,j] representa la distancia mínima para ir desde el vértice vi hasta vj |
| **prim(Graph g, Vertex r)**  “Realiza el algoritmo de Prim tomando como raíz del árbol a r, ajustando información para los vértices del grafo”  Pre: r g.V, g no es dirigido  Post: ∀u g.V, añade atributos u.pred y u.d, que corresponden respectivamente al predecesor y al key añadidos por el algoritmo de Prim |
| **kruskal(Graph g, Vertex s)**  “Realiza el algoritmo de Kruskal sobre g”  Pre:  Post: {e1, e2, …, en}, donde ei g.E son las aristas que pertenecen al MST formado por Kruskal |
| **getEdges(Graph g)**  “Devuelve la lista de aristas del árbol”  Pre: g no es dirigido  Post: {e1, e2, …, en} = g.E |

**Diseños de casos de pruebas unitarias**

*Operaciones estructurales*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 1: Verifica que el método addVertex añade correctamente un vértice al grafo | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +addVertex(Graph, Vertex): void | Existe un grafo sin vertices | Un vértice con valor 1 | El grafo tiene un vértice con valor 1 |
| Graph | +addVertex(Graph, Vertex): void | Existe un grafo con los siguientes vértices:  1, 2, 4 | Un vértice con valor 3 | El grafo tiene 4 vértices y contiene el vértice 3 |
| Graph | +addVertex(Graph, Vertex): void | Existe un grafo con los siguientes vértices:  1, 2, 4 | Un vértice con valor 4 | El grafo tiene 3 vértices (1, 2, 4). No se añadió el nuevo vértice. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 2: Verifica que el método addEdge añade correctamente una arista dirigida y ponderada al grafo | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +addEdge(Graph, Vertex, Vertex, Double): void | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7 | X= 5  Y= 7  W= 3 | 7 es vértice adyacente de 5 y su arista pesa 3. No existe una arista de 7 a 5. |
| Graph | +addEdge(Graph, Vertex, Vertex, Double): void | Existe un grafo no dirigido con los siguientes vértices:  1,2,5,7 | X= 5  Y= 7  W= 3 | 7 es vértice adyacente de 5 y su arista pesa 3. Existe una arista de peso 3 de 7 a 5. |
| Graph | +addEdge(Graph, Vertex, Vertex, Double): void | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7 | X= 5  Y= 5  W= 8 | Existe una arista de 5 a 5 (bucle) de peso 8. |
| Graph | +addEdge(Graph, Vertex, Vertex, Double): void | Existe un grafo no dirigido con los siguientes vértices:  1,2,5,7 | X= 5  Y= 5  W= 8 | Existe una arista de 5 a 5 (bucle) de peso 8. |
| Graph | +addEdge(Graph, Vertex, Vertex, Double): void | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7  Y las siguientes aristas  (1, 2, 3)  (1, 5, 6)  (5, 2, 3)  (7, 5, 5) | X= 5  Y= 7  W= 3 | Los vértices adyacentes a 5 son 2 y 7, y la arista de 5 a 7 pesa 3. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 3: Verifica que el método removeVertex elimina correctamente un vértice del grafo, y por ende todas las conexiones a este. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +removeVertex(Graph, Vertex): void | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7  Y las siguientes aristas  (1, 2, 3)  (1, 5, 6)  (5, 2, 3)  (7, 5, 5)  (5, 7, 3) | El valor del vértice es 2 | El único vértice adyancente de 1 es 5. El único vértice adyacente de 5 es 7 |
| Graph | +removeVertex(Graph, Vertex): void | El mismo que el anterior | El valor del vértice es 1 | No existe el vértice con valor 1. |
| Graph | +removeVertex(Graph, Vertex): void | El mismo que el anterior | El valor de vértice es 5 | El único vértice adyancente de 1 es 2.  7 es un vértice aislado. |
| Graph | +removeVertex(Graph, Vertex): void | Existe un grafo no dirigido con los siguientes vértices  1,2,3,4  Y las siguientes aristas  (1,2,1)  (2,3,1)  (3,4,1)  (4,1,1) | El valor del vértice es 2 | El único vértice adyance a 1 es 4. El único vértice adyacente de 3 es 4. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 4: Verifica que el método removeEdge elimina correctamente una artista del grafo | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +removeEdge(Graph, Vertex, Vertex): void | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7  Y las siguientes aristas  (1, 2, 3)  (1, 5, 6)  (5, 2, 3)  (7, 5, 5)  (5, 7, 3)  (1, 1, 8) | X= 1  Y= 2 | Lo únicos vértices adyacentes de 1 son 5, 1. |
| Graph | +removeEdge(Graph, Vertex, Vertex): void | El mismo que el anterior | X=5  Y=7 | Existe una arista de 7 a 5, pero no de 5 a 7. |
| Graph | +removeEdge(Graph, Vertex, Vertex): void | Existe un grafo no dirigido con los siguientes vértices  1,2,3,4  Y las siguientes aristas  (1,2,1)  (2,3,1)  (3,4,1)  (4,1,1) | X=1  Y=2 | No existe una arista que conecte de 1 a 2, ni de 2 a 1. Existen los vértices 1 y 2. |
| Graph | +removeEdge(Graph, Vertex, Vertex): void | El mismo que el anterior | X=3  Y=4 | No existe una arista que conecte de 3 a 4 ni de 4 a 3. Existen los vértices 3 y 4. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 5: Verifica que el método searchVertex devuelve el vértice dado su valor si este se encuentra en el grafo. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +searchVertex (T): Vertex | Existe un grafo no dirigido con los siguientes vértices  1,2,3,4  Y las siguientes aristas  (1,2,1)  (2,3,1)  (3,4,1)  (4,1,1) | Val= 1 | Retorna un vértice de valor 1 cuyos vértices adyacentes son 2 y 4. |
| Graph | +searchVertex (T): Vertex | El mismo que el anterior | Val=5 | Retorna null |
| Graph | +searchVertex (T): Vertex | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7  Y las siguientes aristas  (1, 2, 3)  (1, 5, 6)  (5, 2, 3)  (7, 5, 5)  (5, 7, 3)  (1, 1, 8) | Val=2 | Retorna un vértice de valor 2 que no tiene vértices adyacentes. |
| Graph | +searchVertex (T): Vertex | El mismo que el anterior | Val=1 | Retorna un vértice con valor 1 cuyos vértices adyacentes son 1, 2, 5. |
| Graph | +searchVertex (T): Vertex | El mismo que el anterior | Val=8 | Retorna null. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 6: Verifica que el método areAdjacent retorna true si dos vertices son adyacentes en el grafo. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +areAdjacent(Graph, Vertex, Vertex): boolean | Existe un grafo no dirigido con los siguientes vértices  1,2,3,4  Y las siguientes aristas  (1,2,1)  (2,3,1)  (3,4,1)  (4,1,1) | X=1  Y=2 | Retorna true. |
| Graph | +areAdjacent(Graph, Vertex, Vertex): boolean | El mismo que el anterior | X=1  Y=3 | Retorna false |
| Graph | +areAdjacent(Graph, Vertex, Vertex): boolean | Existe un grafo dirigido con los siguientes vértices:  1,2,5,7  Y las siguientes aristas  (1, 2, 3)  (1, 5, 6)  (5, 2, 3)  (7, 5, 5)  (5, 7, 3)  (1, 1, 8) | X=1  Y=2 | Retorna true |
| Graph | +areAdjacent(Graph, Vertex, Vertex): boolean | El mismo que el anterior | X=2  Y=1 | Retorna false |
| Graph | +areAdjacent(Graph, Vertex, Vertex): boolean | El mismo que el anterior | X=1  Y=1 | Retorna true |

*Algoritmos de recorridos*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 7: Verifica que el método bfs crea correctamente el árbol bf para encontrar el camino más corto en materia de aristas desde un vértice dado. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +bfs(Graph, Vertex): void | Se tiene el siguiente grafo no dirigido: | El valor del vértice es u | El árbol de predecesores queda como se sigue con u como raíz:  Imagen que contiene objeto  Descripción generada con confianza alta |
| Graph | +bfs(Graph, Vertex): void | Se tiene el siguiente grafo no dirigido: | El valor del vértice es 3 | La raíz es 3, su hijo izquierdo es 2 y su hijo derecho es 4. El hijo izquierdo de 2 es 1 y el derecho es 5 |
| Graph | +bfs(Graph, Vertex): void | Un grafo que tiene 3 vértices: 3, 4, 5 | El valor del índice es 3 | El árbol bf solo está conformado por la raíz 3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 8: Verifica que el método dfs crea un bosque df que provee información acerca de la estructura del grafo | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +dfs(Graph): void | Se tiene el siguiente grafo dirigido no ponderado | Ninguna | El siguiente bosque df, donde solo las aristas sombreadas son las pertenecientes a los árboles: |
| Graph | +dfs(Graph): void | Se tiene un grafo no dirigido con los siguientes vértices:  1,2,3,4  El grafo no posee aristas | Ninguna | El bosque df está compuesto por 4 árboles donde cada vértice del grafo es la raíz un árbol DF. |
| Graph | +dfs(Graph): void | Se tiene el siguiente grafo no dirigido: | Ninguna | El bosque df solo está compuesto por un árbol que en realidad puede verse de la siguiente manera, donde es una secuencia de números y los números en los paréntesis son los timestamps:  1(1/10)  2(2/9)  3(3/8)  4(4/7)  5(5/6) |

*Algoritmos de camino mínimo*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 9: Verifica que el método Dijkstra encuentra el camino más corto desde un vértice a otro. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | Dijkstra(Graph, Vertex): void | Tomado del libro de Matemática discreta y sus aplicaciones. | El vértice tiene valor Dallas | El camino mínimo entre Dallas y Boston cuesta 1500 |
| Graph | Dijkstra(Graph, Vertex): void | El mismo que el anterior | El vértice tiene valor San Francisco | El camino mínimo entre San Francisco y Dallas cuesta 1500 |
| Graph | Dijkstra(Graph, Vertex): void | El mismo que el anterior | El vértice tiene valor Chicago | El camino mínimo entre Chicago y Los Ángeles es la arista que los conecta. |
| Graph | Dijkstra(Graph, Vertex): void | Existe el siguiente grafo no dirigido con los siguientes vértices y las siguientes aristas:  1,2,3,4  (1,2,4)  (1,3,2) | El vértice tiene el valor 1 | No existe un camino mínimo entre 1 y 4. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 10: Verifica que el método Floyd-Warshall encuentra el camino mínimo entre todos los vértices. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | Floydwarshall(): double [][] | Se tiene el siguiente grafo: | Ninguna | El siguiente grafo, representado por una matriz de la siguiente manera: |
| Graph | Floydwarshall(): double [][] | Se tiene el siguiente grafo: | Ninguna | La matriz resultante sería:   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | | 1 | 0 | 5 | 6 | 8 | | 2 | inf | 0 | 1 | 3 | | 3 | inf | 5 | 0 | 2 | | 4 | inf | 3 | 4 | 0 | |
| Graph | Floydwarshall(): double [][] | Se tiene el siguiente grafo dirigido: | Ninguna | La matriz resultante sería:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | A | B | C | D | E | | A | 0 | 32 | 12 | 44 | inf | | B | 10 | 0 | 22 | 54 | inf | | C | 30 | 20 | 0 | 32 | inf | | D | inf | inf | inf | 0 | inf | | E | 7 | 39 | 19 | 51 | 0 | |
| Graph | Floydwarshall(): double [][] | Un grafo con tres vértices: 1, 2, 3 | Ninguna | La diagonal de la matriz es de 0 y el resto de la matriz es inf. |

*Algoritmos de árbol de recubrimiento mínimo*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 11: Verifica que el método Prim crea el árbol generador mínimo de un grafo dado. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +prim(): void | Se tiene el siguiente grafo no dirigido | Ninguna | El árbol resultante es el siguiente resaltado con verde: |
| Graph | +prim(): void | Se tiene el siguiente grafo no dirigido | Ninguna | El árbol generador mínimo es el siguiente  Imagen que contiene texto, mapa, camión  Descripción generada con confianza muy alta |
| Graph | +prim(): void | Se tiene el siguiente grafo no dirigido | Ninguna | El árbol generador mínimo es el siguiente |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 11: Verifica que el método Kruskal crea el árbol generador mínimo, o el bosque generador mínimo, de un grafo dado. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +kruskal(Graph): void | El del primer caso de prueba de la prueba 10. | Ninguna | El del primer caso de prueba de la prueba 10. |
| Graph | +kruskal(Graph): void | El del segundo caso de prueba de la prueba 10. | Ninguna | El del segundo caso de prueba de la prueba 10. |
| Graph | +kruskal(Graph): void | El del tercer caso de prueba de la prueba 10. | Ninguna | El del tercer caso de prueba de la prueba 10. |
| Graph | +kruskal(Graph): void | Se tiene el siguiente grafo no conexo, descrito en el escenario 1 del final de la página. | Ninguna | Se generan los siguientes árboles |

Escenario 1:

