**PRIMERA ENTREGA PROYECTO FINAL**

**MARIA CAMILA LENIS RESTREPO**

**JUAN SEBASTIÁN PALMA GARCÍA**

**JAVIER ANDRÉS TORRES REYES**

**ALGORITMOS Y ESTRUCTURAS DE DATOS**

**ANDRÉS ALBERTO ARISTIZABAL**

**2018-2**

**PRIMERA ENTREGA PROYECTO FINAL**

**Enunciado**

Sarah Winchester era una anciana millonaria heredera del imperio Winchester, una de las empresas más importantes de fabricación de rifles en Estados Unidos, después de que su marido William Wirt Winchester muriera en 1881, heredando así una fortuna de más de 20 millones de dólares. Cuenta la leyenda que después del fallecimiento de su esposo consultó una médium que le dijo que lo espíritus de todas las personas asesinadas con un rifle Winchester vendrían a buscarla con sed de venganza.

De esta manera, con el fin de hacer que los espíritus se perdieran y la dejaran en paz construyó una de las mansiones más grandes y tétricas del planeta. La mansión estuvo en constante remodelación desde 1884 hasta su muerte en 1922, donde constantemente se la añadían nuevas puertas que conducían a otras habitaciones, sótanos, pasillos, baños y demás lugares, todo sin usar un solo plano; por lo tanto, nadie sabe cómo es realmente la mansión Winchester, que actualmente es un sitio turístico donde se les advierte a sus visitantes “Si te separas del grupo, no te aseguramos que te podamos encontrar”, un completo laberinto.

Su tarea es construir un aplicativo que sirva de guía para el recorrido turístico realizado por la mansión Winchester. De esta forma se sabrá desde qué habitación se puede entrar a otras, y cómo se puede volver a alguna parte. Se debe tener en cuenta que no necesariamente si se pasa de la habitación 1 a la 2 se puede pasar de la habitación 2 a la 1, e incluso muchas habitaciones conducen no tienen salida.

Dada una posición en la mansión se debe encontrar el camino más rápido hasta la salida, para que así los espíritus no puedan hacer perder a más personas. También se espera que desde algún punto de la mansión se encuentre el camino más corto, es decir que pase por menos habitaciones, hacía otro punto, buscando así no caminar en círculos y perderse aún más en el gigantesco monumento.

Adicionalmente, a una hora determinada se cierra la mansión, por lo tanto, se desea transmitir un mensaje a todos los rincones de la casa de manera que llegue de la forma más rápida posible y así todas las personas puedan evacuar.

Como la mansión Winchester sigue habitada por espíritus, constantemente se añaden nuevas habitaciones, pasillos y demás lugares que estarán conectados con otros, a la vez que algunos por mal estado han de ser demolidos. Se requiere entonces que se permita añadir y eliminar espacios de la casa, de manera que el modelo de esta sea lo más actualizado posible.

La mansión está llena de tesoros escondidos que muchas veces la gente suele apoderarse de ellos, es por esto se desea llevar un registro de los tesoros encontrados para así llevarlos al museo y no permitir que las personas se apoderen de ellos.

 Figura 1: Mansión Winchester. *Enunciado basado en una historia real.*

**Especificación de requerimientos**

1. Dada una habitación de la mansión se debe encontrar el camino más rápido, en minutos, desde esa habitación hasta la salida. Si la habitación no tiene salida, de debe mostrar un mensaje de advertencia.
2. El sistema debe encontrar el camino que pase por menos habitaciones desde un punto a otro de la mansión. El usuario debe ingresar el punto de partida y el de llegada, y recibe una secuencia de habitaciones incluyendo el punto de partida y el de llegada.
3. El sistema debe transmitir el mensaje de cierre a todos los rincones de la casa, de manera que este llegue de la manera más rápida posible teniendo en cuenta lo que se demora cruzar de una habitación a otra, desde la entrada de la mansión.
4. Añadir una habitación a la mansión. La nueva habitación debe contener el indicador, las habitaciones a las cuales se puede llegar a través de ella, y las habitaciones de las cuales se puede llegar a ella.
5. Dado el indicador de la habitación se debe eliminar la habitación del mapa. Si la habitación contenía tesoros, estos deben quedar en el registro de tesoros encontrados.
6. Dado el indicador de la habitación se deben registrar tesoros encontrados. Se debe añadir el nombre y el valor del tesoro y la habitación a la cual pertenece. Si la habitación es eliminada el tesoro quedará solo en el registro y será enviado al museo.
7. Visualizar los tesoros encontrados, ya sea que aún pertenezcan a la habitación o que pertenezcan al museo. Se debe mostrar su nombre, valor, habitación a la que pertenece o, en su defecto, que pertenece al museo.

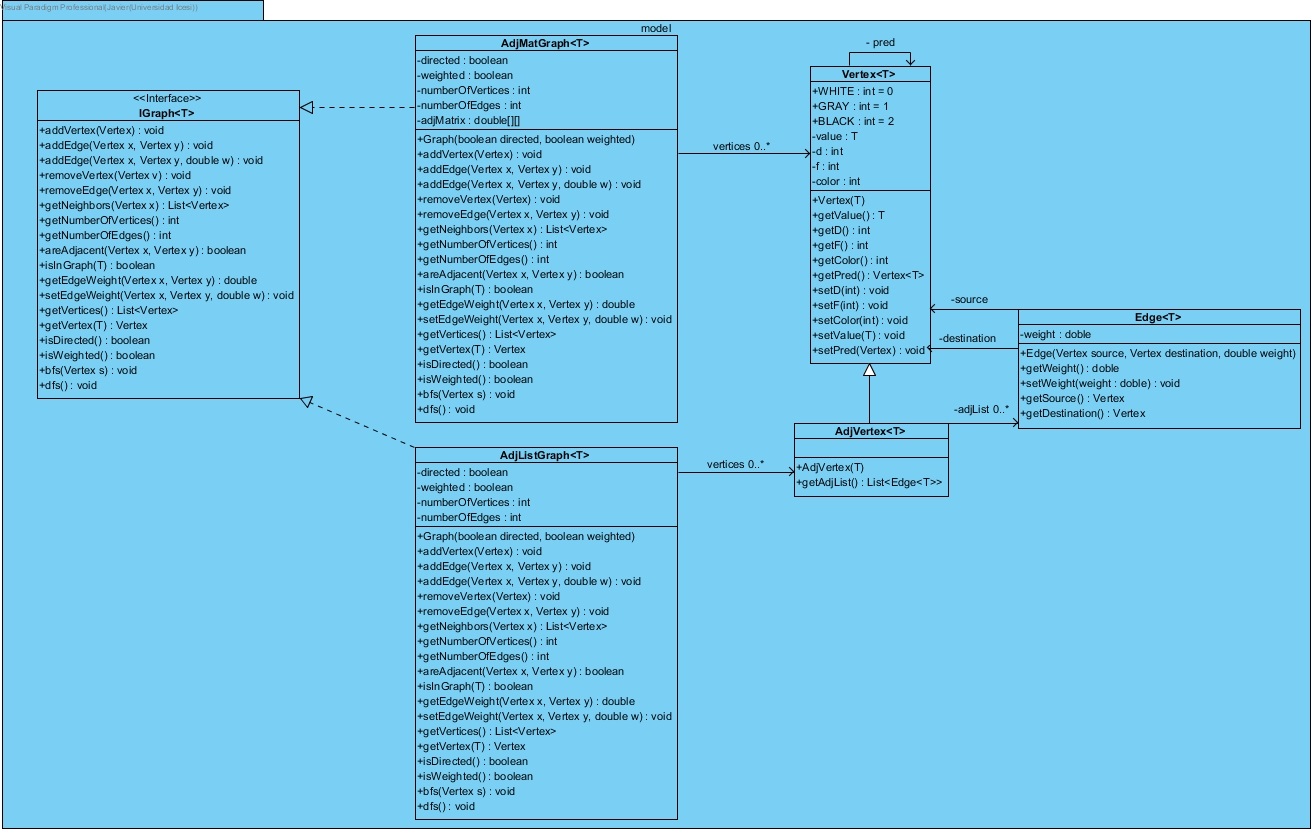
**TAD Grafo**

|  |
| --- |
| **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> * getVertex Graph x T → Vertex * isDirected Graph → Boolean * isWeighted Graph → Boolean * bfs Graph x Vertex → Graph * dfs Graph → Graph |

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 |
| **getVertex(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, Vertex s)**  “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 |

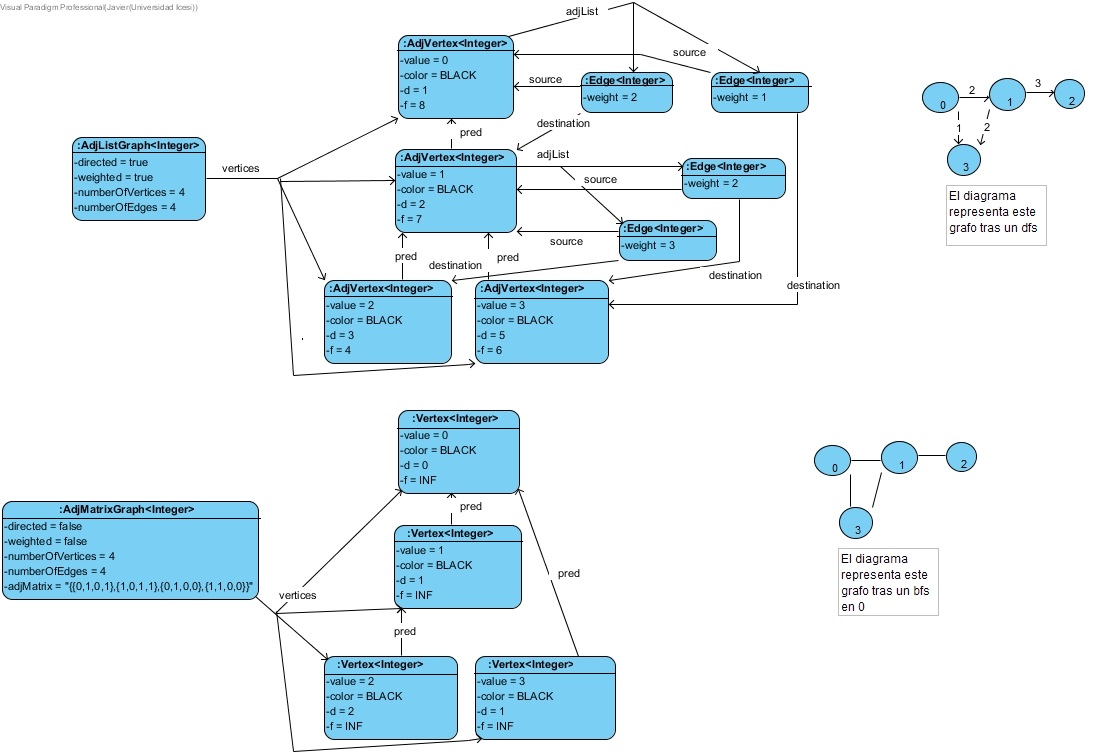
**Diagrama de clases**

****

Todos los diagramas e informes se encuentran en el siguiente repositorio de GitHub: <https://github.com/camilaleniss/AEDFinalProject>

Los diagramas de encuentran dentro del directorio *Bibliografía/Diagramas/*

**Diagrama de objetos**

****

**Diseño 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 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 en el grafo ni las aristas  (1, 5, 6)  (1, 2, 3) |
| 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=3  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 1 y 2. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 5: Verifica que el método getVertex devuelve el vértice dado su valor si este se encuentra en el grafo. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +getVertex(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 | +getVertex(T): Vertex | El mismo que el anterior | Val=5 | Retorna null |
| Graph | +getVertex(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 | +getVertex(T): Vertex | El mismo que el anterior | Val=1 | Retorna un vértice con valor 1 cuyos vértices adyacentes son 1, 2, 5. |
| Graph | +getVertex(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: | 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 |  | 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 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 8: 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 entre Dallas y Boston es:  Dallas- Nueva York- Boston |
| Graph | Dijkstra(Graph, Vertex): void | El mismo que el anterior | El vértice tiene valor San Francisco | El camino mínimo entre San Francisco y Dallas es:  San Francisco- Los Ángeles- Dallas |
| 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. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prueba 9: Verifica que el método Prim recubre todo el árbol con el mensaje de salida. | | | | |
| Clase | Método | Escenario | Entrada | Resultado |
| Graph | +prim(Graph, Vertex): void |  | El valor del vértice es D | El grafo resultante es el siguiente: |

**Bibliografía**

Anónimo. (Desconocido). “Algoritmo de Prim”. Tomado de: <https://sites.google.com/site/complejidadalgoritmicaes/prim>

Miller, B. (2014, julio 2) “El tipo abstracto de datos grafo”. Tomado de: <http://interactivepython.org/runestone/static/pythoned/Graphs/ElTipoAbstractoDeDatosGrafo.html>

Peréz, L. (Desconocido). “¿Visitarías una mansión embrujada de 2 hectáreas? Conoce la historia de la Mansión Winchester”. Tomado de: <https://www.vix.com/es/mundo/190470/visitarias-una-mansion-embrujada-de-2-hectareas-conoce-la-historia-de-la-mansion-winchester>