PROYECTO FINAL: PRIMERA ENTREGA

LUIS FELIPE GOMEZ ANDRADE

ELIAS CALEB ESTUPIÑAN TOLOSA

OSCAN IVAN RIASCOS GUEVARA

ANDRES ARISTIZABAL

ALGORITMOS Y ESTRUCTURAS DE DATOS

2019-2

Enunciado

Sam es un apasionado aventurero que, desde niño, le gusta afrontar desafíos explorando ruinas de antiguas culturas, en donde ninguna persona o, muy pocas, se han atrevido a entrar para investigar los secretos que allí se ocultan. Para suerte de él, vive en una misteriosa isla en alguna parte del planeta, la cual está llena de misteriosos laberintos construidos por una civilización ya extinta.

De esta manera, con el fin de seguir su sueño de convertirse en el mejor explorador del mundo, Sam ha tomado la decisión de estudiar aquellos misteriosos laberintos mencionados anteriormente. Así mismo, dado que no existe información sobre cómo llegar a la salida de cada laberinto (Solo se conocen las entradas), nuestro aventurero ha llegado a la conclusión de que primero se deberá realizar o encontrar un mapa de cada laberinto, con el fin de que, tiempo después, se estudien a fondo.

Su tarea, si decide aceptarla, es construir un aplicativo que sirva para guiar a Sam durante su peligrosa expedición y mapear cada laberinto, para que, tiempo después, se calcule la ruta más rápida desde la entrada hasta la salida, por si otro explorador quisiera estudiarlo y/o recorrerlo.

Dado que cada laberinto está bajo tierra, éste se encuentra totalmente a oscuras, por lo tanto, Sam poseerá una linterna que lo ayudará a guiarse. Adicionalmente, esta linterna, única en su tipo, tiene una funcionalidad de que, cada 15 segundos, permite alcanzar una zona de iluminación 3 veces más grande que la que posee por defecto, permitiéndole, a Sam, mapear o visualizar una zona más grande.

En adición a lo anterior, se tiene conocimiento de que algunas zonas del laberinto poseen puntos de “iluminación” las cuales conservan su “luz” cuando el explorador pasa por ellas y las mapea o visualiza. Por lo tanto, se debe de agregar una funcionalidad al aplicativo de encontrar un camino desde el punto donde se encuentra Sam hasta otro punto de “iluminación”, lo cual servirá para evitar caminar en círculos y perderse aún más en las oscuras y misteriosas ruinas. Así mismo, el aplicativo debe permitir el guardar el progreso llevado por Sam en el laberinto, para que, si decide salirse de la expedición, cuando vuelva contará con sus datos previamente conseguidos. Esta opción, solo se deberá permitir cuando Sam se encuentre en uno de los puntos de “iluminación”, ya que, el volver a estos sitios es relativamente más fácil que las otras partes oscuras y tenebrosas de los laberintos.

Por último, una vez terminado un laberinto, se debe de mostrar un mensaje donde al guía (el jugador o usuario) se le indique que ha terminado de dirigir a Sam en este laberinto. También, se debe de preguntar si se desea continuar la expedición o si está ya ha terminado. Todo lo anterior para que si, por un lado, se selecciona la primera opción (continuar expedición) se debe de trasladar a Sam hasta la entrada de otro laberinto para realizar el mismo procedimiento descrito con anterioridad. Por otro lado, si se selecciona la segunda opción (terminar expedición) se debe de llevar al guía a la pantalla principal del aplicativo.

Especificación de requerimientos

1. Requerimientos funcionales:

Nombre	R1: encontrar camino más corto desde la entrada hasta la salida.
Resumen	El programa debe de estar en la capacidad de encontrar o calcular la ruta más corta, desde el punto de partida hasta el punto de salida del laberinto. Para esto, se debe de haber mapeado por completo el laberinto con anterioridad.
Entradas	• Posición del punto de partida inicial. • Posición del punto de llegada o meta.
Salida	Se mostraron en pantalla la ruta más corta desde el punto de inicio del laberinto hasta su punto de llegada o meta, pintándolo sobre el laberinto descubierto.

Nombre	R2: encontrar camino desde el punto actual hasta un punto de iluminación.
Resumen	El programa debe de estar en la capacidad de encontrar o calcular un camino, desde el punto actual donde se encuentra Sam, hasta un punto de iluminación seleccionado en el laberinto. Para esto, se debe de haber mapeado el punto al cual se quiere llegar nuevamente.
Entradas	• Posición actual de Sam en el laberinto. • Posición del punto de iluminación al cual se quiere llegar nuevamente.
Salida	Se mostraron en pantalla el/los camino(s) desde la posición actual de Sam hasta el punto escogido en el laberinto, pintándolo sobre este (el laberinto). Una vez alcanzado dicho punto, el camino pintado desaparecerá.

Nombre	R3: mapear zona 3 veces más grande
Resumen	El programa debe de permitirle a Sam estar en la capacidad de mapear una zona 3 veces más grande de la que puede mapear por defecto. Para esto, solamente se puede utilizar cada 15 segundos y, si Sam no utiliza la habilidad, no se reiniciará el conteo de los 15 segundos hasta que la utilice.
Entradas	• Posición actual de Sam en el laberinto. • Tiempo transcurrido desde la última utilización de la habilidad.
Salida	Se mostraron en pantalla los puntos que están en el área definida anteriormente, como puntos ya descubiertos en el mapa. Posteriormente se reiniciará el contador de 15 segundos.

Nombre	R4: continuar expedición
Resumen	El programa debe de estar en la capacidad de que, una vez escogida la opción de continuar expedición, cuando se acaba de mapear un

	laberinto, lleve al jugador a la entrada de otro laberinto para realizarle el correspondiente mapeado.
Entradas	Ninguna
Salida	Se pintó en pantalla el nuevo laberinto, con Sam en el punto de partida.

Nombre	R5: terminar expedición
Resumen	El programa debe de estar en capacidad de que, una vez escogida la opción de terminar expedición, cuando se acaba de mapear un laberinto, lleve al guía o jugador al menú principal.
Entradas	Ninguna
Salida	Se mostró en pantalla la ventana del menú principal.

Nombre	R6: guardar partida
Resumen	El programa debe de estar en capacidad de guardar la partida en curso cuando el personaje (Sam) se encuentre sobre un punto de luz. Para esto, se debió haber generado y pintado un mapa en pantalla.
Entradas	• Posición actual del personaje (Sam). • Estructura de datos donde se almacena la información del mapa.
Salida	Se guardó con éxito la partida en curso.

Nombre	R7: cargar partida
Resumen	El programa debe estar en capacidad de cargar una partida, guardada con anterioridad. Para esto, se debe de poseer un apartida guardada anteriormente.
Entradas	• Ruta del archivo donde se encuentra la partida guardada.
Salida	Se cargó con éxito la partida guardada, generando y pintando el mapa guardado previamente.

2. Requerimientos no funcionales:

Nombre	R8: generar mapa
Resumen	El programa debe de estar en la capacidad de generar un mapa o laberinto a partir de un archivo de texto plano, donde se encontrará toda la información de éste (el laberinto). Para esto, el jugador debe de haber iniciado una nueva partida o haber terminado un laberinto anteriormente.
Entradas	• Ruta del archivo de texto contenedor de la información del mapa.
Salida	Se poseen los datos del mapa generado a partir de un archivo de texto plano en su correspondiente estructura de almacenamiento.

Nombre	R9: pintar mapa
Resumen	El programa debe de estar en capacidad de pintar en pantalla un mapa generado a partir de los datos leídos del archivo de texto plano. Para esto, el jugador debe de haber iniciado una nueva partida o haber terminado un laberinto anteriormente.
Entradas	• Estructura de datos donde se almacena la información del mapa.
Salida	Se visualizó en pantalla el mapa indicado.

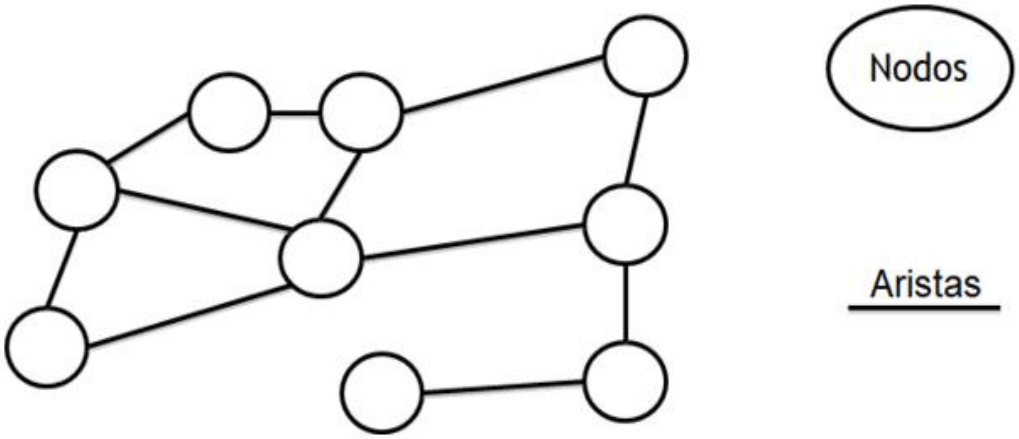
Nombre	R10: iluminar adyacentes
Resumen	El programa debe de estar en capacidad de "iluminar" los lugares adyacentes al personaje (Sam) cuando este pasa cerca o sobre ellos.
Entradas	• Posición actual del personaje (Sam) • Estructura de datos donde se almacena la información del mapa.
Salida	Se pintaron o "iluminaron" los lugares adyacentes al personaje (Sam)

Nombre	R11: verificar mapa completado
Resumen	El programa debe de estar en capacidad de verificar si el jugador ha llegado o no, a la salida del laberinto o meta. Para esto, se debió, con anterioridad, generado y pintado un nuevo mapa.
Entradas	• Posición actual del personaje (Sam). • Estructura de datos donde se almacena la información del mapa.
Salida	Se mostró en pantalla el mensaje de que ha descubierto la salida o meta. No se hubiera mostrado en caso contrario.

Nombre	R12: iluminar puntos de luz
Resumen	El programa debe estar en capacidad de iluminar los puntos que mantendrán su luz durante toda la partida. Para esto, el jugador debió haber pasado por ellos con anterioridad.
Entradas	• Posición actual del personaje (Sam). • Estructura de datos donde se almacena la información del mapa.
Salida	Se iluminaron todos los puntos de luz por los que el jugador ha pasado o paso.

TAD Grafo

Representación:



Invariante:

Dado un grafo G, existe un número asociado a G que es el mismo valor para cualquier grafo que sea isomorfo con él.

Operaciones:

- Graph ---------- Boolean, Boolean ---------- Graph
- addEdge --------- Graph, Vertex, Vertex ---------- Graph
- addEdge --------- Graph, Vertex, Vertex, Double ------ Graph
- addVertex -------- Graph, Vertex ---------- Graph
- isDirected -------- Graph ---------- Boolean
- getVerticesNumber - Graph ---------- Integer
- getEdgesNumber -- Graph ---------- Integer
- areAdjacent -------- Graph, Vertex, Vertex ---------- Boolean
- isInGraph ---------- Graph, T ---------- Boolean
- isWeighted -------- Graph ---------- Boolean
- getVertices --------- Graph ---------- List<Vertex>
- deleteEdge -------- Graph, Vertex, Vertex ---------- Graph
- deleteVertex ------ Graph, Vertex ---------- Graph
- getNeighbors ----- Graph, Vertex ---------- List<Vertex>
- getVertex --------- Graph, T ---------- Vertex
- getEdgeWeight ---- Graph, Vertex, Vertex ---------- Double
- setEdgeWeight ---- Graph, Vertex, Vertex, Double ------ Graph

Operaciones
Graph(Boolean d, Boolean w) " Crea un nuevo vacío indicando si es dirigido o no, si es ponderado a no" Pre: Ninguna Post: Se ha creado un grafo vacío, dirigido o no, ponderado a no.
addEdge(Graph g, Vertex v1, Vertex v2) " Añade una nueva arista al grafo g uniendo a los vértices v1 y v2 con peso predeterminado 1" Pre: Los vértices v1 y v2 pertenecen al Grafo. Post: Se ha añadido una nueva arista entre los vértices v1 y v2 con peso 1.
addEdge(Graph g, Vertex v1, Vertex v2, Double w) " Añade una nueva arista al grafo g uniendo a los vértices v1 y v2 con el peso w " Pre: Los vértices v1 y v2 pertenecen al grafo. El grafo es ponderado. El peso w para los vértices dados es mayor que cero. Post: Se añadido una nueva arista al grafo uniendo a los vértices v1 y v2 con peso w
addVertex(Graph g, Vertex v) " Añade el vértice v al grafo g" Pre: El vértice v no pertenece al grafo g Post: El vertice v pertence al grafo g

isDirected(Graph g) " Verifica si el grafo es dirigido a no " Pre: Ninguna Post: Indica si en grafo g es dirigido.
getVerticesNumber(Graph g) " Devuelve el número de vértices en el grafo g" Pre: Post: Se ha devuelto el número de vértices de g
getEdgesNumber(Graph g) " Devuelve el número de aristas en el grafo g" Pre: Post: Se ha devuelto el número de aristas de g
areAdjacent(Graph g, Vertex v1, Vertex v2) " Indica si hay una arista entre v1 y v2" Pre: Los vértices v1 y v2 pertenecen al grafo g. Post: Se ha devuelto si hay una arista entre v1 y v2.
isInGraph(T v) " Indica si el valor v pertenece a algún vértice en el grafo" Pre: Post: Se ha indicado que el valor v pertenece a algún vértice del grafo.
isWeighted (Graph g)

" Indica si el grafo g es ponderado o no"

Pre:

Post: Se ha devuelto si el grafo es ponderado o no.

---

getVertices(Graph g)

" Devuelve la lista de vértices en el grafo g"

Pre:

Post: Se ha devuelto la lista de vértices del grafo g.

---

deleteEdge(Graph g, Vertex v1, Vertex v2)

" Elimina la arista existente entre los vértices v1 y v2"

Pre: Los vértices v1 y v2 pertenecen al grafo g.
Existe una arista entre v1 y v2.

Post: Se ha eliminado la arista entre v1 y v2.

---

deleteVertex(Graph g, Vertex v)

" Elimina el vértice v del grafo g"

Pre: Los vértices v pertenecen al grafo g.

Post: Se ha eliminado el vértice v.

---

getNeighbors(Graph g, Vertex v)

" Devuelve la lista de vértices que son adyacentes al vértice pertenecientes al grafo g"

Pre: El vértice v pertenece al grafo g.

Post: Se ha devuelto la lista vértices adyacentes a v.

---

getVertex(T v)

"Devuelve el vértice con el valor v, si existe."

Pre:

Post: Si el vértice con valor v existe, se devuelve. De lo contrario se devuelve null.

getEdgeWeight(Graph g, Vertex v1, Vertex v2)

" Devuelve el peso de la arista entre v1 y v2"

Pre: Los vértices v1 y v2 pertenecen al grafo g.
Existe una arista entre v1 y v2.

Post: Se ha devuelto el peso entre v1 y v2.

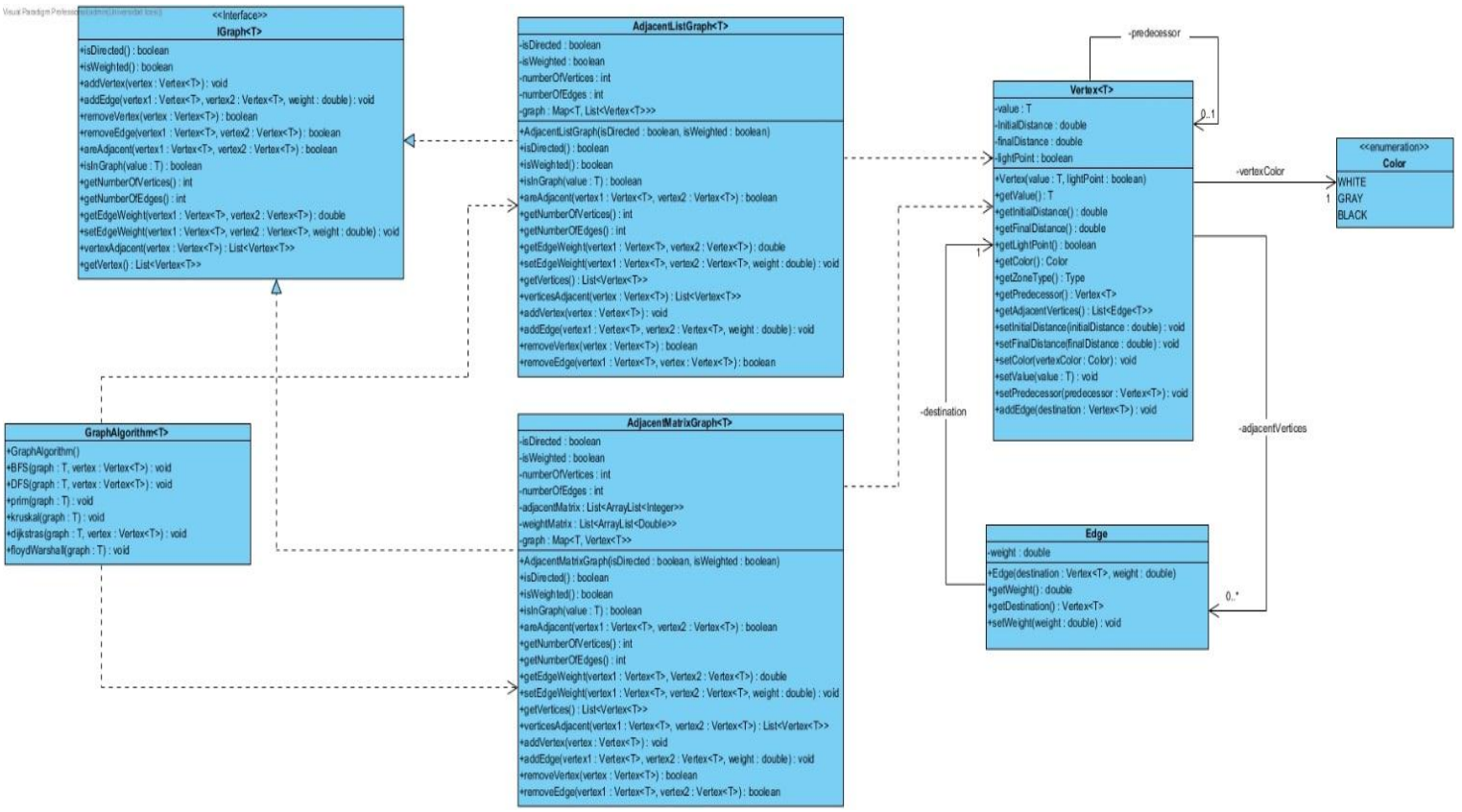
setEdgeWeight(Graph g, Vertex v1, Vertex v2, Double w)

" Cambia el peso de la arista entre los vértices v1 y v2"

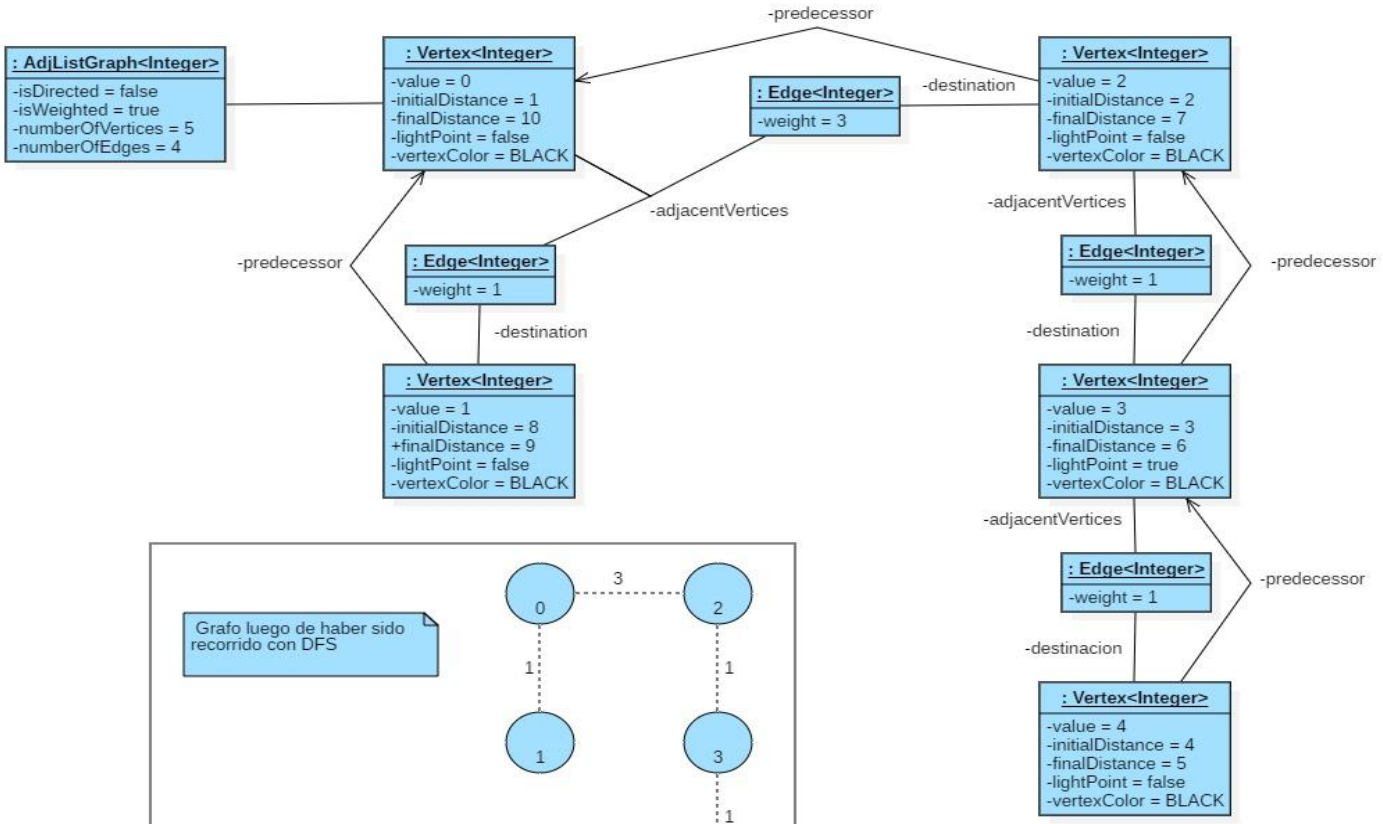
Pre: Los vértices v1 y v2 pertenecen al grafo g.
Existe una arista entre v1 y v2.
El peso w es mayor a cero

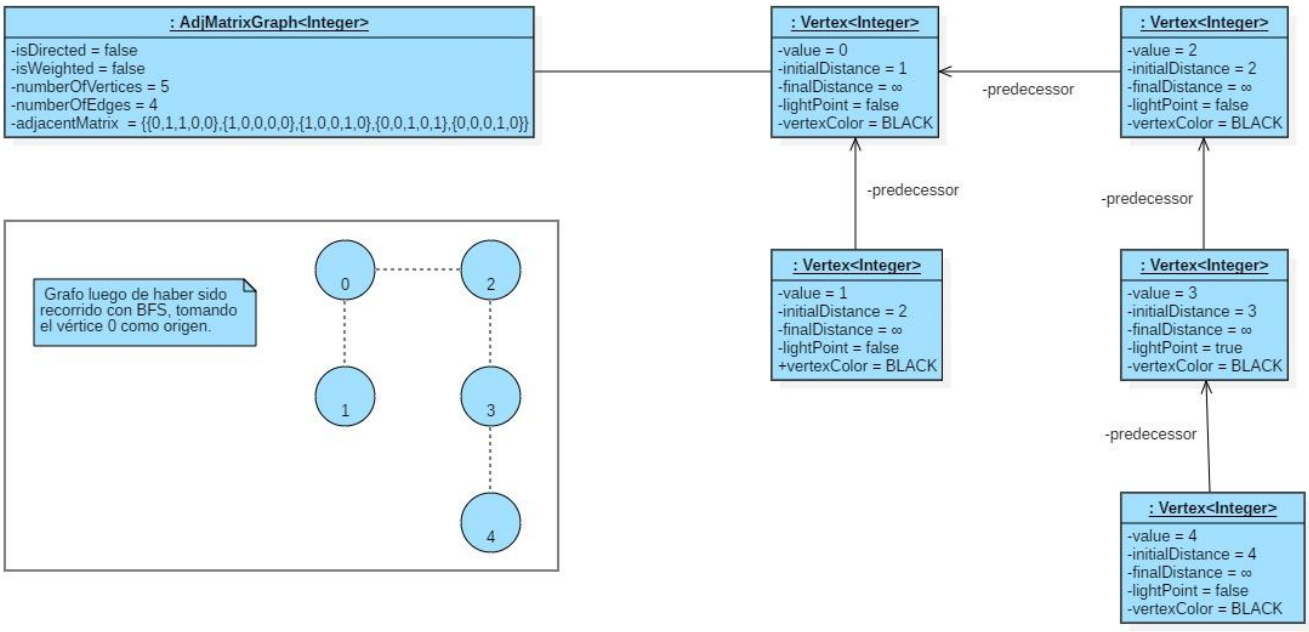
Post: Se ha cambiado el paso de la arista entre v1 y v2.

Diagrama de clases



Diagramas de Objectos





Diseño de Pruebas unitarias

Objetivo: Verificar que el método isDirected devuelve el valor de verdad correcto dependiendo del tipo de grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	isDirected	Se creó un grafo en el cual las aristas tienen un sentido definido.	Grafo especificado en la siguiente matriz Adyacencia. Cuyas filas y columnas están enumeradas del 1 al 3. $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$	True. El grafo es dirigido y por tanto el método retornara verdadero.
Graph	isDirected	Se creó un grafo, donde las aristas son relaciones simétricas y no apuntan a ningún sentido.	Grafo especificado en la siguiente matriz Adyacencia. Cuyas filas y columnas están enumeradas del 1 al 3. $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$	False. EL grafo de entrada no es dirigido, puesto que conserva la adyacencia.

Objetivo: Verificar que el método isWeigthed retorna el valor de verdad de acuerdo con el tipo de grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	isWeighted	Se creó un grafo donde cada par de vértice tiene una arista que contiene un peso el cual representa la distancia entre ellos	Grafo que contiene los siguientes vértices. A1, A2, A3. Donde ir de A1 hasta A2 la arista tiene un peso de 5, de A2 a A3 es 2 y de A3 a A1 es 10.	True. El grafo es ponderado, es decir sus aristas tienen un peso o un valor que representa el costo de ir de un vértice a otro.
Graph	isWeighted	Se creo un grafo donde cada par de vértices contiene una artista no ponderada	Grafo que contiene los siguientes vértices. A1, A2, A3. Donde ir de un vértice a cualquiera de ellos no tiene ningún costo.	False. El grafo no es ponderado, porque ir de un vértice a otro no representa ningún costo.

Objetivo: Verificar que el método addVertex añade correctamente un vértice al grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	checkAddVertex	Existe un grafo sin vértices.	Vértice con valor 0	True. El vértice se agregó correctamente al grafo. Ahora la estructura contiene un vértice con valor 0.
Graph	checkAddVertex	Existe un grafo que contiene los siguientes vértices: 1,2,3,4	Vértice con valor 5	True. Se agregó correctamente el grafo con valor 5. Por lo tanto el grafo ahora tiene 5 vértices.

Objetivo: Verifica que el método addEdge añade correctamente una arista dirigida y ponderada al grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	addEdge	Existe un grafo dirigido con los siguientes vértices: 1,2,5,7	Vertex1 = 5 Vertex2 = 1 Vertex3 = 8	5 es vértice adyacente de 1 y su arista pesa 3.
Graph	addEdge	Existe un grafo dirigido con los siguientes vértices: 1,2,5,7	Vertex1 = 5 Vertex2 = 1 Vertex3 = 8	1 no es vértice adyacente de 5.
Graph	addEdge	Existe un grafo dirigido con los siguientes vértices: 1,2,5,7	Vertex1 = 5 Vertex2 = 5 Vertex3 = 12	Existe una arista de 5 a 5 la cual corresponde a un bucle de peso 12.
Graph	addEdge	Existe un grafo dirigido con los siguientes vértices: 1,2,5,7		Los vértices adyacentes a 5 son 2 y 7

		Y las siguientes aristas. (1,2,3). (1,5,6). (5,2,3). (5,7,5).		

Objetivo: Verifica que el método removeVertex elimina correctamente un vértice del grafo y las conexiones a este.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	removeVertex	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. Del mismo modo las siguientes aristas. (1,2,3) (1,3,6) (2,3,5) (3,4,3) (4,2,1) (4,1,7) Nota: Los elementos de color rojo son subjetos de modificación.	El valor del vértice a eliminar es 2.	El único vértice adyacente de 1 es 3, del mismo modo de 4 es 1.
Graph	removeVertex	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10)	El valor del vértice a eliminar es 3.	Se elimino correctamente el vértice 5. Por tanto, se modifica el grafo de la siguiente manera: 5 es el único vértice adyacente de 4. 1 es un vértice aislado.

Objetivo: Verificar que el método removeEdge elimina correctamente una arista del grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	removeEdge	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)	Vertex1 = 2 Vertex2 = 3	El único vértice adyacente de 2 es 4. Por otro lado, el único vértice adyacente de 3 es 4.
Graph	removeEdge	Misma escena que el anterior	Vertex1 = 1 Vertex2 = 5	Lanza la excepción, especificando que no existe una arista que conecte de 1 a 5.

Objetivo: Verifica que el método areAdjacent retorna un booleano si dos vértices son adyacentes en el grafo.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	areAdjacent	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)	Vertex1 = 1 Vertex2 = 2	True. Los vértices 1 y 2 son adyacentes.
Graph	areAdjacent	Mismo escenario del anterior	Vertex1 = 1 Vertex2 = 4	False. Los vértices 1 y 4 no son adyacentes.

Objetivo: Verificar que el método isInGraph retorna correctamente si una estructura es un grafo o no.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	isInGraph	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)		True. La estructura es un grafo, ya que: G = (V, E). Donde: V es un conjunto de vértices. E es un conjunto de aristas que relacionan estos vértices.

Objetivo: Verificar que el método getNumberOfVertices retorne el número de vértices.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	getNumberOfVertices	Existe un grafo dirigido con los siguientes vértices 1,2,3,4.	Vertex1 = 1; Vertex2 = 2; Vertex3 = 3; Vertex4 = 4;	El número de vértices que hay en el grafo son 4.
Graph	getNumberOfVertices	No se agregó ningún vértice al grafo.		El número de vértices que existen en la estructura es 0.

Objetivo: Verificar el método getNumberEdge retorna correctamente el valor de verdad.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	getNumberOfEdges	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)	Contiene las siguientes aristas. (1,2,3) (2,3,5) (3,4,8) (4,5,10)	El número de aristas que existen en el grafo son 5.
Graph	getNumberOfEdges	No se agregó ninguna arista al grafo.		El número de aristas en el grafo es 0.

Objetivo: Verificar el método getEdgeWeight retorna el peso de una arista.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	getEdgeWeight	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)	Vertex1 = 1 Vertex2 = 2	El peso de ir desde la arista 1 a la 2 es 3.

Objetivo: Verificar el método vertexAdjacent retorna correctamente la lista de vértices adyacentes a el.				
Clase	Método	Escenario	Valores de entrada	Resultado
Graph	vertexAdjacent	Existe un grafo dirigido con los siguientes vértices 1,2,3,4. (1,2,3) (2,3,5) (3,4,8) (4,5,10) (2,4,7)	Vertex = 2	Los vértices adyacentes a 2 son 4 y 3.