**Proyecto Final de curso**

**Integrantes:**

* Juan Fernando Giraldo
* Juan Diego Carvajal
* David Bolívar

**Método de la Ingeniería**

**Identificación del problema:**

Necesidades y síntomas del problema

* Necesidad de un sistema para medir la eficiencia de los operarios de tren en la empresa Deutsche Bahn.
* Necesidad de una representación grafica de algunos sistemas ferroviarios de Alemania.
* El programa debe de ser rápido e intuitivo para que pueda ser utilizado fácilmente sin necesidad de capacitaciones.
* El sistema debe de poder calcular la ruta mas optima para ir de una estación a otra.
* Se podrán guardar los reportes generados por el programa para posterior análisis.

Definición del problema

La mayoría de los países europeos cuentan con sistemas de metro o de tren, los cuales son muy extensos y complejos, razón por la cual existen diferentes empresas públicas o privadas que se encargan de administrar pequeños tramos de estos sistemas. Deutsche Bahn, la empresa encargada del control del sistema de trenes en la ciudad de Berlín, lo ha contratado a usted para desarrollar un simulador de la sala de control que gestiona actividad ferroviaria de esta ciudad, esto con el objetivo de capacitar a los empleados en cuanto a las habilidades que debe tener un operador de este sistema. Como ya se sabe, la ruta de los trenes es administrada desde la sala de control mediante la gestión de los rieles por donde se quiere que pase el tren. El simulador debe de estar en capacidad de cargar cualquier tipo de red ferroviaria, además debe de poder reproducir la actividad descrita previamente y adicionalmente se podrá tener más de un tren en las vías al mismo tiempo.

**Recopilación de información:**

Estructura de datos

Una estructura de datos es una forma particular de organizar datos en una computadora para que puedan ser utilizados de manera eficiente.

Grafo

En matemáticas y ciencias de la computación, un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.1​ Son objeto de estudio de la teoría de grafos.

Árbol de expansión

Dado un grafo conexo, no dirigido G. Un árbol de expansión es un árbol compuesto por todos los vértices y algunas (posiblemente todas) de las aristas de G. Al ser creado un árbol no existirán ciclos, además debe existir una ruta entre cada par de vértices.

Árbol de expansión mínima

Dado un grafo conexo, no dirigido y con pesos en las aristas, un árbol de expansión mínima es un árbol compuesto por todos los vértices y cuya suma de sus aristas es la de menor peso.

Problema del camino más corto

En la teoría de grafos, el problema del camino más corto es el problema que consiste en encontrar un camino entre dos vértices (o nodos) de tal manera que la suma de los pesos de las aristas que lo constituyen es mínima.

Matriz de adyacencia

La matriz de adyacencia es una matriz cuadrada que se utiliza como una forma de representar relaciones binarias.

Lista de adyacencia

En teoría de grafos, una lista de adyacencia es una representación de todas las aristas o arcos de un grafo mediante una lista. Si el grafo es no dirigido, cada entrada es un conjunto o multiconjunto de dos vértices conteniendo los dos extremos de la arista correspondiente. Si el grafo es dirigido, cada entrada es una tupla de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente.

Algoritmo de Dijkstra

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.

Algoritmo de Floyd-Warshall

En informática, el algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución.

Conjuntos Disyuntos

En computación, una estructura de datos para conjuntos disjuntos es una estructura de datos que mantiene un conjunto de elementos particionados en un número de conjuntos disjuntos (no se solapan los conjuntos).

Fuentes:

* <https://es.wikipedia.org/wiki/Estructura_de_datos>
* <https://jariasf.wordpress.com/2012/04/19/arbol-de-expansion-minima-algoritmo-de-kruskal/>
* <https://es.wikipedia.org/wiki/Grafo>
* <https://es.wikipedia.org/wiki/Problema_del_camino_m%C3%A1s_corto>
* <https://es.wikipedia.org/wiki/Lista_de_adyacencia>
* <https://es.wikipedia.org/wiki/Matriz_de_adyacencia>
* <https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra>
* <https://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall>
* https://es.wikipedia.org/wiki/Estructura\_de\_datos\_para\_conjuntos\_disjuntos

**Requerimientos funcionales:**

|  |  |
| --- | --- |
| R1 | Generar grafos |
| Descripción | El programa esta en capacidad de generar cualquier grafo desde un archivo de texto |
| Entradas | Un grafo representado por un archivo de texto |
| Salidas | Representación grafica del grafo |

|  |  |
| --- | --- |
| R2 | Simulación del movimiento del tren |
| Descripción | El programa está en capacidad de simular los movimientos que hace el tren por medio de la selección del camino que se va a tomar |
| Entradas | El tren y los posibles caminos que tomar |
| Salidas | El tren con su nueva dirección |
| R3 | Reporte eficiencia |
| Descripción | El programa está en capacidad de generar un reporte de eficiencia para posteriormente analizar el desempeño de los operarios |
| Entradas | La simulación hecha por un operario |
| Salidas | Un documento de texto con la información de la simulacion |

|  |  |
| --- | --- |
| R4 | Movimiento de otros trenes |
| Descripción | El programa está en capacidad de simular el movimiento de otros trenes en el mismo sistema de rieles |
| Entradas | Los trenes y el sistema de rieles |
| Salidas | Representación gráfica de los trenes siguiendo cierta trayectoria |

|  |  |
| --- | --- |
| R5 | Camino mínimo |
| Descripción | El programa está en capacidad de calcular el camino mínimo de un punto a otro dentro del sistema de rieles |
| Entradas | El sistema de rieles y los dos puntos |
| Salidas | Una trayectoria mínima entre los dos puntos dados |

**Requerimientos no funcionales**

|  |  |
| --- | --- |
| RN1 | Interfaz amigable con el usuario |
| Descripción | El programa debe de ser lo suficientemente claro para que cualquier usuario pueda utilizarlo |

|  |  |
| --- | --- |
| RN2 | Generación de redes ferroviarias muy grandes |
| Descripción | El programa debe de poder generar grafos muy grandes |

**Búsqueda de soluciones creativas**

Lluvia de ideas:

Teniendo en cuenta la recopilación de información se realizó una lista de revisión que permite explorar las diferentes posibilidades de diseño. Gracias a esta podemos tener en cuenta todas las opciones de estructuras de datos que se investigaron durante la recopilación de información. Como se verá a continuación, las ideas obtenidas en esta fase se enfocan en la construcción del simulador.

* Implementar el sistema ferroviario como un grafo no dirigido, en el cual los vértices son las estaciones o los puntos de giro y las aristas son los rieles que conectan los vértices.

**Transformación de ideas a diseños preliminares**

Alternativa #1 Implementar el sistema ferroviario como un grafo

* Es sencillo de implementar en código
* Existen múltiples algoritmos que facilitan la solución de varios problemas
* La mayoría de los métodos de grafos tienen una complejidad relativamente baja

**Preparación de informes y especificaciones**

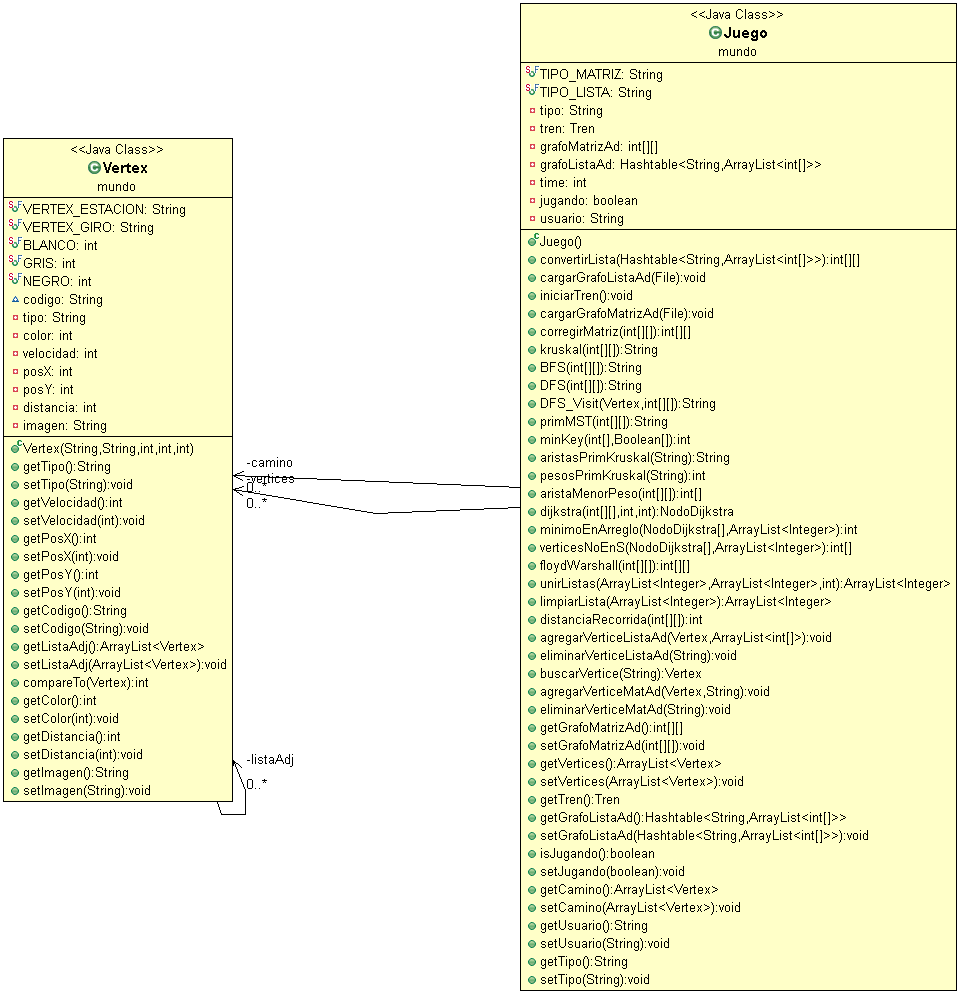
Especificaciones del problema (En términos de entrada y salida)

Nombre: Programa de computador que sea capaz de simular un sistema ferroviario y además se debe poder controlar la trayectoria de un tren

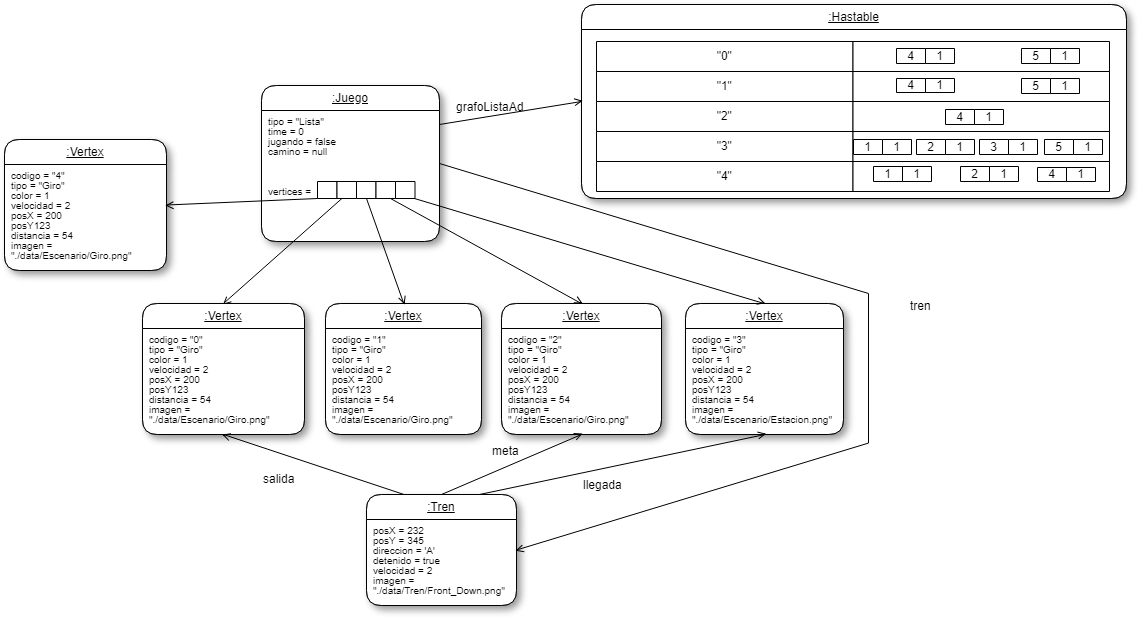
Entradas: Un sistema ferroviario representado por un grafo en un archivo de texto.

Salidas: Representación grafica de este sistema y además el tren.

**Diagrama de clases**

****

**Diagrama de objetos**



**Implementación del diseño**

Tareas por implementar

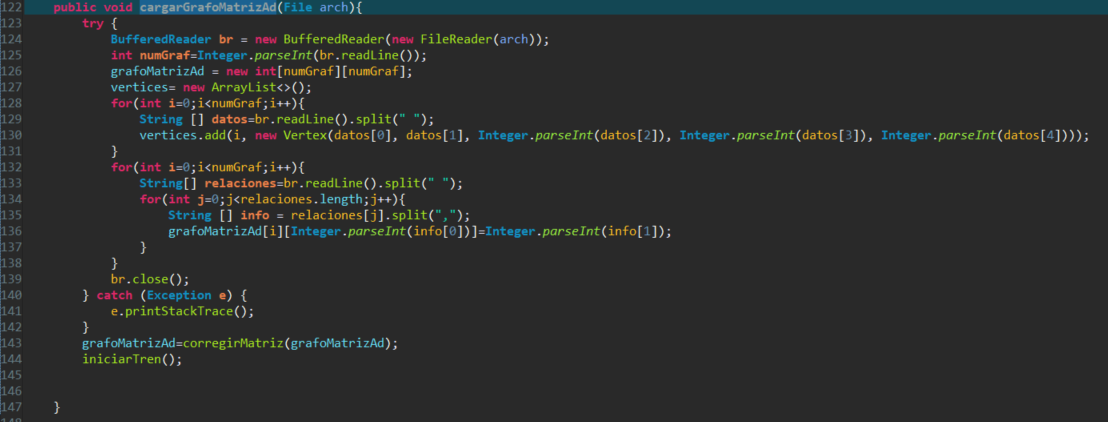
* Hacer la estructura de datos vértice
* Representar la estructura de datos grafo como una matriz de adyacencia
* Representar la estructura de datos grafo como una lista de adyacencia
* Hacer la estructura de datos conjunto disyunto
* Hacer la clase que represente el simulador y desarrollar todas las funciones necesarias
* Hacer la clase que represente los trenes
* Desarrollar una interfaz funcional que cumpla con las especificaciones necesarias

**Construcción**

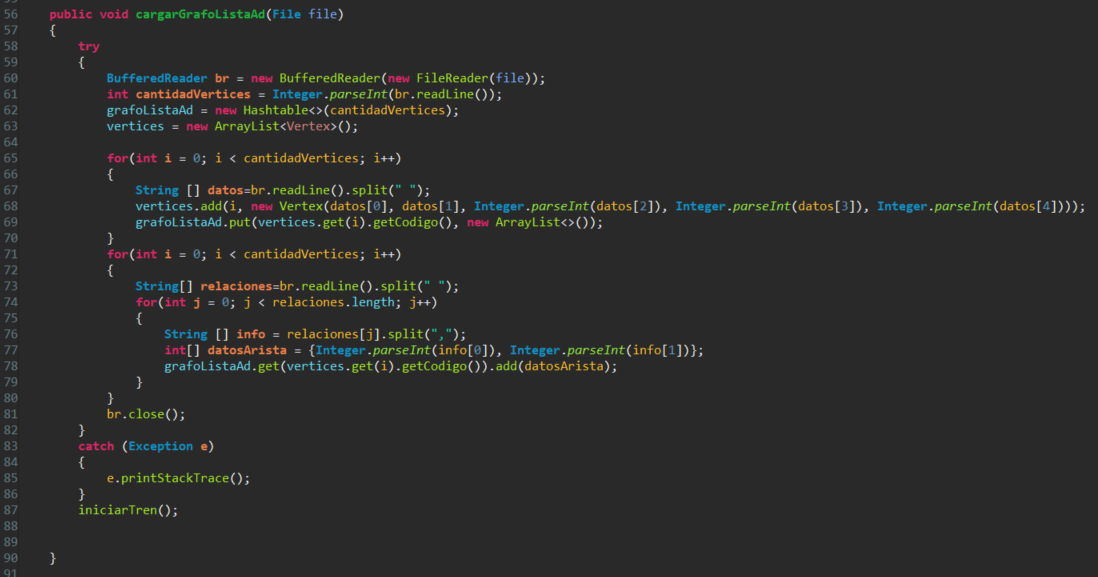
Vértice



Matriz de adyacencia



Lista de adyacencia



Conjunto disyunto



**Diseño TAD**

Grafo

|  |
| --- |
| **TAD**  Graph |
| Simple Graph:    Graph where:  Circumferences are vertex or nodes.  Segments of lines between vertex or nodes are edges |
| Invariants: |
| Operations:   * Eliminar * Consultar * CrearGrafo |

|  |
| --- |
| CrearGrafo()  “Crea un grafo con el primer vértice dado”  {pre: TRUE}  {post: Grafo con n vértices si la lista no está vacía. FALSE si la lista está vacía} |

|  |
| --- |
| Agregar()  “Agrega un vértice dado al grafo”  {pre: }  { post: Grafo con el vértice agregado} |

|  |
| --- |
| Eliminar()  “Elimina un vértice que tenga el código de vértice dado”  {pre: }  { post: TRUE si elimina el vértice. FALSE si no lo elimina} |

|  |
| --- |
| Modificar()  “Busca un vértice que tenga el código de vértice dado”  {pre: }  {post: TRUE si encuentra el vértice. FALSE si no lo encuentra } |

**Diseño de pruebas unitarias**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: buscarVertice(String codigo)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se busca un vértice, de acuerdo a un código dado** | **1** | **grafoMatrizAd =**  **{{0,0,0,1,1},**  **{0,0,0,1,1},**  **{0,0,0,1,0},**  **{1,1,1,0,1},**  **{1,1,0,1,0}}**  **Código = “3”** | **Se retorna el vértice con el mismo código al dado por parámetro, si no se encontró se retorna null** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: agregarVerticeListaAd(Arraylist<int[]> adyacencias)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se busca agregar un vértice al grafo** | **2** | **Adyacencias = { [1,5 4,5] }** | **Se ha agredado el vértice a la lista de vertices y se ha agregado su lista de adyacencia a las listas de adyacencia del grafo** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: agregarVerticeMatAd(Vertex vertice, String adyacencia)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se busca agregar un vértice al grafo** | **3** | **Vertex nuevo = new Vertex("34", Vertex.VERTEX\_ESTACION, 2, 200, 200);**  **Adyacencias=** **"0,50"** | **Se ha agredado el vértice a la lista de vertices y se ha agregado sus adyacencias a la matriz, además la matriz es ahora de n+1** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: eliminar(String codigo)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se busca eliminar un vértice de acuerdo a un código dado** | **2** | **Código = “3”** | **Si se encontró un vértice con el mismo código se ha borrado de la lista de vertices y sus aristas de las listas de adyacencia del grafo.** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: eliminar(String codigo)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se busca eliminar un vértice de acuerdo a un código dado** | **3** | **Código = “0”** | **Si se encontró un vértice con el mismo código se ha borrado de la lista de vertices y sus adyacencias de la matriz, además ahora su tamaño es n-1** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: kruskal(int[][] grafoMatrizAd)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica kruskal a un grafo implementado con matriz de adyacencias para encontrar las aristas que forman el árbol generador mínimo y su peso de recorrido** | **1** | **grafoMatrizAd =**  **{{0,0,0,1,1}**  **,{0,0,0,1,1}**  **,{0,0,0,1,0}**  **,{1,1,1,0,1}**  **,{1,1,0,1,0}}** | **Se retorna un arreglo de caracteres en cuyo índice 0 se encuentran las aristas que formar el árbol de expansión mínima y en cuyo índice 1 se encuentra la sumatoria de los pesos de dichas aristas** |
| **2** | **Se aplica kruskal a un grafo implementado con listas de adyacencia para encontrar las aristas que forman el árbol generador mínimo y su peso de recorrido** | **2** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **Se retorna un arreglo de caracteres en cuyo índice 0 se encuentran las aristas que formar el árbol de expansión mínima y en cuyo índice 1 se encuentra la sumatoria de los pesos de dichas aristas** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: prim(int[][] grafoMatrizAd)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica prim a un grafo implementado con matriz de adyacencias para encontrar las aristas que forman el árbol generador mínimo y su peso de recorrido** | **1** | **grafoMatrizAd =**  **{{0,0,0,1,1},**  **{0,0,0,1,1},**  **{0,0,0,1,0},**  **{1,1,1,0,1},**  **{1,1,0,1,0}}** | **Se retorna un arreglo de caracteres en cuyo índice 0 se encuentran las aristas que formar el árbol de expansión mínima y en cuyo índice 1 se encuentra la sumatoria de los pesos de dichas aristas** |
| **2** | **Se aplica prim a un grafo implementado con lista de adyacencia para encontrar las aristas que forman el árbol generador mínimo y su peso de recorrido** | **1** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **Se retorna un arreglo de caracteres en cuyo índice 0 se encuentran las aristas que formar el árbol de expansión mínima y en cuyo índice 1 se encuentra la sumatoria de los pesos de dichas aristas** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: BFS(int[][] grafoMatrizAd)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica BFS a un grafo implementado con matriz de adyacencias para encontrar las aristas que forman el árbol de expansión mínimo** | **1** | **grafoMatrizAd =**  **{{0,0,0,1,1}**  **,{0,0,0,1,1}**  **,{0,0,0,1,0}**  **,{1,1,1,0,1}**  **,{1,1,0,1,0}}** | **String árbol = "(0,3) (0,4) (3,1) (3,2) "**  **Se retorna una cadena de texto con parejas que indican conexiones: (Padre, hijo).** |
| **2** | **Se aplica BFS a un grafo implementado con listas de adyacencia para encontrar las aristas que forman el árbol de expansión mínimo** | **2** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **String árbol = “(0,1) (0,2) (0,3) (0,4) (0,5) (0,6)”**  **Se retorna una cadena de texto con parejas que indican conexiones:**  **(Padre, hijo).** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: DFS(int[][] grafoMatrizAd)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica DFS a un grafo implementado con matriz de adyacencias para encontrar las aristas que forman el árbol de expansión mínimo** | **1** | **grafoMatrizAd =**  **{{0,0,0,1,1}**  **,{0,0,0,1,1}**  **,{0,0,0,1,0}**  **,{1,1,1,0,1}**  **,{1,1,0,1,0}}** | **String árbol = “(0,3) (3,1) (1,4) (3,2) / "**  **Se retorna una cadena de texto con parejas que indican conexiones: (Padre, hijo). En caso de ser un bosque, separara los múltiples arboles con “ / “** |
| **2** | **Se aplica DFS a un grafo implementado con listas de adyacencia para encontrar las aristas que forman el árbol de expansión mínimo** | **2** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **String árbol = "(0,1) (1,2) (2,3) (3,4) (4,5) (5,6) / "**  **Se retorna una cadena de texto con parejas que indican conexiones: (Padre, hijo). En caso de ser un bosque, separara los múltiples arboles con “ / “** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: dijkstra(int [][] adyacencia,int inicio,int llegada)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica Dijkstra a un grafo implementado con matriz de adyacencias para encontrar el camino minimo desde inicio hasta llegada** | **3** | **grafoMatrizAd = {{0,99999,99999,57,45}**  **,{99999,0,99999,45,28}**  **,{99999,99999,0,57,99999}**  **,{57,45,57,0,60}**  **,{45,28,99999,60,0}}** | **Se retorna un vértice de tipo Dijkstra el cual contiene el camino y además el peso de este camino** |
| **2** | **Se aplica Dijkstra a un grafo implementado con lista de adyacencias para encontrar el camino minimo desde inicio hasta llegada** | **2** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **Se retorna un vértice de tipo Dijkstra el cual contiene el camino y además el peso de este camino** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase: Juego** | | **Método: floydWarshall(int[][] ad)** | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | **Se aplica Floyd-Warshall a un grafo implementado con matriz de adyacencias para encontrar el camino minimo desde todos los vertices hasta todos los vertices** | **3** | **grafoMatrizAd = {{0,99999,99999,57,45}**  **,{99999,0,99999,45,28}**  **,{99999,99999,0,57,99999}**  **,{57,45,57,0,60}**  **,{45,28,99999,60,0}}** | **Se retorna una matriz de vertices tipo Dijkstra en la cual todas sus posiciones tienen un peso y un camino** |
| **2** | **Se aplica Floyd-Warshall a un grafo implementado con lista de adyacencias para encontrar el camino minimo desde todos los vertices hasta todos los vertices** | **2** | **Aristas = { 2,5**  **2,5 6,5**  **0,5 1,5 3,5 4,5**  **2,5 5,5**  **2,5 5,5 6,5**  **3,5 4,5**  **1,5 4,5 }** | **Se retorna una matriz de vertices tipo Dijkstra en la cual todas sus posiciones tienen un peso y un camino** |