**Nombres: Joshua Lattke, Rodolfo Moreno Gutiérrez**

**Paso 1. Identificación del Problema**

**Identificación de necesidades y síntomas**

* Modo multijugador: El juego debe admitir un máximo de dos jugadores.
* Presencia de enemigos: El juego debe incluir la presencia de enemigos hostiles.
* Escenario interactivo: El juego debe estar ambientado en un escenario interactivo.
* Recogida de armas: Los jugadores deben tener la capacidad de recoger armas dispersas por el escenario.
* Selección de avatar: Los jugadores deben poder elegir su propio avatar para jugar.
* Inteligencia enemiga: Los enemigos deben mostrar comportamientos de movimiento inteligentes.
* Escenario destructible: El escenario del juego debe ser susceptible de sufrir daños o destrucción.
* Indicador de vida: El juego debe informar la vida que tiene el jugador para saber cuándo se muere.

**Identificación y definición concreta sin ambigüedad del proble**ma

El problema se centra en la creación de un videojuego interactivo que permita a dos jugadores participar en una experiencia basada en un grafo con 50 aristas y 50 vértices. El objetivo principal es implementar algoritmos gráficos fundamentales para enriquecer la jugabilidad. Estos algoritmos incluyen la búsqueda en anchura (BFS) y en profundidad (DFS) para la exploración del grafo, la determinación de caminos de peso mínimo utilizando los algoritmos de Dijkstra y Floyd-Warshall, así como la construcción de un Árbol de Recubrimiento Mínimo (MST) mediante los enfoques de Prim y Kruskal.

**TABLA DE ESPECIFICACIÓN DEL PROBLEMA DE INGENIERÍA DE SOFTWARE**

|  |  |
| --- | --- |
| CLIENTE | Icesi |
| USUARIO | Jugadores |
| REQUERIMIENTOS FUNCIONALES | RF1: Selección de jugadores  RF2: Recogida de armas  RF3: Indicar de la vida  RF4: Presencia de enemigos  RF5: Inteligencia enemiga  RF6: Escenario interactivo |
| CONTEXTO DEL PROBLEMA | Se necesita el desarrollo de un videojuego en el cual sea multijugador, el desarrollador tiene permitido crearlo como el desee. |
| REQUERIMIENTOS NO FUNCIONALES | Debe tener 50 artista y 50 vértices. Además, debe tener 3 algoritmos de grafos. |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF1: Selección de jugadores | | |
| Resumen | Se debe informar el sistema cuanto es el numero de jugadores que van a jugar. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Jugadores | Int | Debe ingresar máximo 2 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1. Ingresa al juego. 2. Debe informar cuantos jugadores van a jugar. 3. Se registra el número de jugadores. | | |
| Resultado o postcondición | . Guardar la información de número de jugadores | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| ConfirmacionJugadores | Boolean |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF2: Recogida de arma | | |
| Resumen | El jugador o jugadores deben seleccionar un arma en que se encuentren en el mapa | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Bala | Int | Numero de bala 10 |
| Arma | Int | Tiempo de uso 6 segundos |
| Apuntado | String | Se debe mover el mouse donde indica el usuario |
| BotonIzquiero | String |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1. El jugador o jugadores deben apunta 2. Con el botón | | |
| Resultado o postcondición | Eliminar los enemigos | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| ConfirmacionJugadores | Boolean |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF3: Indicar de vida | | |
| Resumen | En jugador o lo jugadores podrán informa de cuantos es la cantidad de vida que le queda para morirse. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| DañoRecibido | Int | Debe inicializar en la máxima cantidad disponible |
| TiempoRecuperacion | Float |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1. Calcular el daño que hacen los enemigos 2. Tiempo de recuperación de la vida 3. Vida recuperada totalmente | | |
| Resultado o postcondición | 1. Calcular la vida del jugador | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| IndicarlaVida | Int |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF4: Presencia de enemigos | | |
| Resumen | Los enemigos deben aparecer de forma aleatoria en el mapa, debe tener un numero aleatorio de enemigos. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| NumeroEnemigos | Int | Debe inicializar en la máxima cantidad disponible |
| AletorioMapa | Random | Debe aparecer el enemigos en cualquier punto del mapa |
|  |  |  |
|  |  |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1.Se genera el número de enemigos.  2.Se generan los enemigos en el mapa | | |
| Resultado o postcondición | 1. Visualizar los enemigos | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| NumeroEnemigos | Int | Se generan los enemigos |
|  | AletorioMapa | Random | Debe visualizar los enemigos en el mapa |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF5: Inteligencia enemiga | | |
| Resumen | Los enemigos deben buscar de una manera eficiente al jugador o los jugadores | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| DañoRecibido | Int | Debe inicializar en la máxima cantidad disponible |
| TiempoRecuperacion | Float |  |
|  |  |  |
|  |  |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1.Calcular el daño que hacen los enemigos   1. Tiempo de recuperación de la vida 2. Vida recuperada totalmente | | |
| Resultado o postcondición | 1. Calcular la vida del jugador | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| IndicarlaVida | Int |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre o identificador | RF6: Escenario indestructible | | |
| Resumen | Debe haber ciertas paredes donde se pueden destruir en el mapa para poder seguir avanzando en el mapa. | | |
| Entradas | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| ParedesDestruibles |  | Las coordenadas deben estar dentro del mapa y deben ser accesibles para el jugador. |
| ConjutoCoordenadas |  | Las coordenadas deben estar dentro del mapa y deben ser accesibles para el jugador. |
|  |  |  |
|  |  |  |
|  |  |  |  |
| Actividades generales necesarias para obtener los resultados | 1. El jugador debe encontrar las paredes y destruirlas 2. El jugador debe usar la herramienta para destruir las paredes | | |
| Resultado o postcondición | Las paredes destructibles se han destruido | | |
| Salidas | Nombre salida | Tipo de dato | Condición de selección o repetición |
| MapaModificado |  |  |
|  | MapaActualizado |  |  |
|  |  |  |  |

Paso 2. Recopilación de Información

Se buscan diferentes alternativas para poder encontrar una mejor solución en nuestro problema. En ese caso se buscaron aplicaciones o sistemas parecidos al desarrollo de nuestro video juego llamado skulbum en para poder entretener las personas. Las 3 principales que encontramos

**1 Call of duty Warzone:** Video juego desarrollado para multiplataforma como en celular, computador y consolas. Este juego es un género de acción además es multijugador lo cual permite jugar varios jugadores a la vez.

**2.Gran Turismo:** videojuego desarrollado para la consola de PlayStation es un género de carreras donde puedes encontrar varias pistas del mundo y coches, los cuales puedes manejar y te van enseñando de como los pilotos profesionales manejan en estas pistas.

**3.** **Flappy Bird (2013):** Un juego simple en el que el jugador debe controlar un pájaro que vuela entre tuberías. El juego está compuesto por un total de 26 polígonos, desarrollado en Android.

Para poder tener un mejor desarrollo en nuestro videojuego debemos saber que tipo de estructurar se van a usar para poder implementar los grafos de manera adecuadamente.

**Recorrido de Grafos DFS**

El algoritmo recorre el grafo guardando una lista de no, en una lista guarda. Lo particular que tiene es el último nodo que agregamos a lista es el primero que vamos a ver. Lo cual nos indica que vamos a camina el grafo.

El algoritmo consiste en meter un nodo primer nodo en una lista inicial, agregar el ultimo nodo que agregamos a su lista y a sus vecinos.

Se puede usar en la siguiente funcionalidad:

Dependiendo de las condiciones del juego, puedes ajustar el objetivo del enemigo en cada ejecución del DFS. Por ejemplo, en un nivel de sigilo, el enemigo podría buscar al jugador sigilosamente, mientras que, en otro nivel, podría perseguirlo de manera agresiva.

**Caminos de Peso Mínimo (Dijkstra, Floyd-Warshall)**

El algoritmo de Floyd-Warshall es capaz de encontrar el camino más corto entre cada par de vértices en un grafo al comparar todas las rutas posibles entre estos pares. Esto se hace utilizando una matriz que almacena las distancias mínimas entre los vértices, y se va actualizando iterativamente.

Iteraciones paulatinas:

El algoritmo realiza iteraciones para calcular la distancia más corta entre todos los pares de vértices. En cada iteración, se considera un conjunto de vértices intermedios, representados por el índice 'k', y se evalúan todas las rutas posibles entre los vértices 'i' y 'j' pasando por 'k'. Se mejora gradualmente la estimación de la distancia más corta.

Recursión y definición de CaminoMinimo(i, j, k):

Para calcular el camino mínimo de 'i' a 'j' a través del vértice 'k', se utiliza una función recursiva llamada CaminoMinimo(i, j, k). Esta función compara dos opciones: el camino mínimo de 'i' a 'j' a través de 'k-1' (sin usar el vértice 'k') o el camino mínimo de 'i' a 'k' más el camino mínimo de 'k' a 'j'. Esta decisión se toma para cada valor de 'k' de manera iterativa.

La base de la recursión se encuentra en CaminoMinimo(i, j, 0), que representa el peso de la arista directa entre 'i' y 'j'. A medida que se incrementa el valor de 'k' en cada iteración, el algoritmo calcula rutas más largas y optimiza los caminos.

Proceso de finalización:

El algoritmo continúa realizando estas comparaciones y actualizaciones hasta que se haya considerado cada vértice como un posible punto intermedio ('k' alcanza 'n', donde 'n' es el número total de vértices). En este punto, habremos encontrado el camino más corto entre todos los pares de vértices usando algún vértice intermedio.

Ventajas del algoritmo de Floyd-Warshall:

Encuentra el camino más corto entre todos los pares de vértices en un solo paso.

Maneja grafos con pesos negativos siempre que no haya ciclos negativos.

Útil para determinar la existencia de ciclos negativos en un grafo ponderado.

En un videojuego podemos aplicar conectando las ubicaciones del juego.

Hacer un recorrido de donde ha recorrido el jugador

**Árbol de Recubrimiento Mínimo -MST- (Prim, Kruskal)**

El algoritmo de Prim es para encontrar un árbol recubridor mínimo en grafo conexo, no dirigido y cuyas aristas estan etiquetadas.

Lo que realiza el algoritmo es encontrar un subconjunto de artistas que forman un árbol con todos los vértices, donde le peso de todas las aristas en el árbol mínimo posible. Si en el caso del grafo no es conexo encontrar el árbol recubridor minimo para los componentes que no formen de dicho grafo conexo.

El algoritmo realiza los siguientes pasos:

1.Inicializar un árbol con un único vértice, elegido arbitrariamente, del grafo.

2.Aumentar el árbol, por un lado. Llamamos lado a la unión entre dos vértices: de las posibles uniones que pueden conectar el árbol a los vértices que no están aún en el árbol, encontrar el lado de menor distancia y unirlo al árbol.

3. Repetir el paso 2 (hasta que todos los vértices pertenezcan al árbol)

Se puede aplicar de la siguiente manera:

1. Generación de niveles: Se puede usar un MST para generar niveles o escenarios en tu juego. Por ejemplo, puedes modelar las ubicaciones importantes en el nivel como nodos y las conexiones entre ellas como aristas. Luego, al aplicar un algoritmo de MST, obtendrás una estructura de nivel que conecta todas las ubicaciones de manera eficiente.
2. Diseño de caminos o carreteras: Si un juego implica la construcción de carreteras, senderos o rutas, puedes usar un MST para diseñar rutas eficientes entre ubicaciones importantes en el juego, como ciudades o puntos de interés. El MST podría representar la red de carreteras o senderos necesarios.

Sala en una matriz representativa que Utilice el algoritmo de kruskal para encontrar los camino corto entre la salas y que este sea el recorrido de la partida.

**Paso 3. Búsqueda de Soluciones Creativas**

Como hemos planteado anteriormente en la recopilación de información podemos obtener diferentes alternativas las cuales son las siguientes:

Alternativa 1 Call of duty Warzone.

Call of Duty: Warzone es un videojuego de disparos en primera persona, perteneciente al género Battle royale gratuito, lanzado el 10 de marzo de 2020 para PlayStation 4, PlayStation 5, Xbox One, Xbox Series X|S y Microsoft Windows



Alternativa 2 Gran Turismo.

Gran Turismo es una serie japonesa de videojuegos de simulación de carreras desarrollada por Polyphony Digital. Esta saga de videojuegos se ha desarrollado exclusivamente para las consolas de PlayStation.

Imagen que contiene tabla, teléfono

Descripción generada automáticamente

Alternativa 3 Flappy Bird (2013)

Flappy Bird fue un juego para móviles desarrollado en Hanói por el desarrollador vietnamita Nguyen Hà Đông y publicado por .GEARS Studios, un pequeño desarrollador de juegos independiente, también con sede en Vietnam

Imagen que contiene Texto

Descripción generada automáticamente

Alternativa 4 Skulbum desarrollado en java.

Un videojuego desarrollado con javaFx en lenguaje de java, En el cual se pueden jugar de 2 jugadores, tiene como de objetivo eliminar todos los enemigos que se encuentren y la forma de eliminarlo es con potenciadores y bombas

**Paso 4. Transición de las Ideas a los Diseños Preliminares**

Descartamos las alternativas 1 y 2 porque tenemos la capacidad de programar juegos en multiplataforma ni en consola de PlayStation, la 3 alternativa también queda rechazada por un motivo similar y es que nosotros no tenemos la capacidad de programar en Kotlin debido a esto se selecciona la alternativa 4, por el desarrollo en java y la capacidad tenemos para hacer nuestro proyecto.

**Alternativa 4. Desarrollo de skulbum realizado en java**

-Para el planteamiento de nuestro proyecto debemos usar el lenguaje java con estructura de datos ya mencionada anteriormente en la parte de recopilación de información.

**Paso 5. Evaluación y Selección de la Mejor Solución**

**Diagrama de flujo del sistema:**

* Criterio A. Estética. La solución es agradable a la vista del usuario:
  + [2] Tiene un buen diseño.
  + [1] No es agradable.
* Criterio B. Completitud. Cumple con las funcionalidades requeridas.
  + [3] cumple con todas las funcionalidades que los usuarios necesitan.
  + [2] cumple con algunas, pero no con todas las funcionalidades requeridas.
  + [1] No cumple con las funcionalidades que se piden.
* Criterio C. Exactitud. La solución cumple con las funcionalidades empleando las estructuras de datos pertinentes.
  + [3] emplea todas las estructuras de datos pertinentes.
  + [2] emplea algunas, pero no todas.
  + [1] no emplea las estructuras pertinentes.
* Criterio D. Portabilidad. El sistema se puede usar en una gran variedad de dispositivos.
  + [2] se puede acceder desde muchos dispositivos.
  + [1] Es muy limitado y se puede acceder desde pocos dispositivos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterio A | Criterio B | Criterio C | Criterio D | Total |
| Alternativa 4 Desarrollo de Skulbum | 2 | 3 | 3 | 1 | 9 |

La mejor solución fue la alternativa 4 de Skulbum

**Definición de la tablas TAD estructuras de utilizar**

|  |
| --- |
| TAD Graph |
| < abstract object > G = (V, E)  • Note that “V” is a set of vertices.  • “E” is a set of edges.  • E ⊆ (V x V), which are ordered pairs of vertices. |
| { inv: < ∀ (vᵢ, vⱼ) ∈ E, (vⱼ, vᵢ) ∈ E >} |
| Primitive operations :  Constructor:  createGraph → Graph  Modifiers:  addVertex Graph X Element → Graph  addEdge Graph X Element X Element → Graph  removeVertex Graph X Element → Graph  removeEdge Graph X Element X Element → Graph |

|  |
| --- |
| **createGraph** |
| **creates an empty Graph { pre: TRUE } { post: Graph = { nill } }** |
|  |

|  |
| --- |
| **addVertex** |
| **adds a vertex to the graph { pre: Graph ( G ) } { post: element ∈ Graph ( G ) }** |

|  |
| --- |
| **addEdge** |
| **Given a graph, add a relationship between two elements (vertices ) of that graph. { pre: Graph ( G ) ≠ ∅ ∧ 𝑉1 ∈ 𝑉 𝑣𝑒𝑟𝑡𝑖𝑐𝑒𝑠 ∧ 𝑉2 ∈ 𝑉 𝑣𝑒𝑟𝑡𝑖𝑐𝑒𝑠 } { post: (𝑉1, , 𝑉2) ∈ E(edges) ∨ (𝑉2, , 𝑉1,) ∈ E(edges)}** |

|  |
| --- |
| **removeVertex** |
| **Removes a given element from the graph { pre: v ∈ V ∧ V ≠ ∅ } { post: v ∉ 𝑉 ∧ v ∉ G }** |

|  |
| --- |
| **removeEdge** |
| **Eliminate a relationship between two elements (vertices) of the graph. { pre: (𝑉1, , 𝑉2) ∈ E(edges) ∨ (𝑉2, , 𝑉1,) ∈ E(edges) } { post: (𝑉1, , 𝑉2) ∉ E(edges) ∧ (𝑉2, , 𝑉1,) ∉ E(edges) }** |

|  |
| --- |
| **isContained** |
| **checks whether an element is found or not in the graph { pre: G ≠ ∅} { post: True: if the element ∈ 𝐺 ∨ False: if the element ∉ G }** |

**Seudocódigo:**

**DFS**

class MatrixAdyacency {

boolean[][] matrizAdyacency

int numedges

ArrayList<T> vertex

MatrixAdyacency(int numedges) {

this.numedges = numedges

this.matrizAdyacency = new boolean[numedges][numedges]

this.vertex = new ArrayList<>()

}

addVertex(T vertice) {

try {

vertex.add(vertice)

} catch (Exception e) {

print("Esta lleno")

}

}

addEdge(int origin, int destiny) {

if (origin >= 0 && origin < numedges && destiny >= 0 && destiny < numedges) {

matrizAdyacency[origin][destiny] = true

} else {

print("Los índices de los vértices no son válidos.")

}

}

showMatriz() {

String msg = ""

for (int i = 0; i < numedges; i++) {

for (int j = 0; j < numedges; j++) {

msg += (matrizAdyacency[i][j] ? "1 " : "0 ")

}

msg += "\n"

}

return msg

}

toString() {

String msg = ""

for (int i=0; i < vertex.size(); i++) {

msg += (i+1) + ": " + vertex.get(i) + "\n"

}

return msg

}

seeConex(start, destiny) {

boolean[] visited = new boolean[numedges]

dfs(start, visited)

return visited[destiny]

}

hardlyConnect() {

for (int i = 0; i < vertex.size(); i++) {

for (int j = 0; j < vertex.size(); j++) {

if (i != j) {

if (!seeConex(i, j)) {

return false

}

}

}

}

return true

}

dfs(vertex, visited) {

visited[vertex] = true

for (int i = 0; i < numedges; i++) {

if (matrizAdyacency[vertex][i] && !visited[i]) {

dfs(i, visited)

}

}

}

}

**Prim**

class PrimAlgorithm<T> {

Map<T, Integer> indexes

List<Edge<T>>[] adjacencyList

PrimAlgorithm(int vertices) {

indexes = new HashMap<>()

adjacencyList = new ArrayList[vertices]

for (int i = 0; i < vertices; i++) {

adjacencyList[i] = new ArrayList<>()

}

}

addEdge(vertex1, vertex2, weight) {

int index1 = getIndex(vertex1)

int index2 = getIndex(vertex2)

adjacencyList[index1].add(new Edge<>(vertex1, vertex2, weight))

adjacencyList[index2].add(new Edge<>(vertex2, vertex1, weight))

}

minimumSpanningTree() {

List<Edge<T>> result = new ArrayList<>()

boolean[] visited = new boolean[indexes.size()]

PriorityQueue<Edge<T>> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.getWeight()))

pq.add(new Edge<>(null, indexes.keySet().iterator().next(), 0))

while (!pq.isEmpty()) {

Edge<T> edge = pq.poll()

T currentVertex = edge.getVertex2()

if (visited[getIndex(currentVertex)]) continue

visited[getIndex(currentVertex)] = true

result.add(edge)

for (Edge<T> neighbor : adjacencyList[getIndex(currentVertex)]) {

if (!visited[getIndex(neighbor.getVertex2())]) {

pq.add(neighbor)

}

}

}

return result

}

getIndex(vertex) {

if (!indexes.containsKey(vertex)) {

int index = indexes.size()

indexes.put(vertex, index)

}

return indexes.get(vertex)

}

}

**Kruskal**

class KruskalAlgorithm<T> {

Map<T, Integer> indexes

List<Edge<T>> edges

int[] parent

KruskalAlgorithm(List<Edge<T>> edges) {

this.edges = edges

this.indexes = new HashMap<>()

}

minimumSpanningTree() {

List<Edge<T>> result = new ArrayList<>()

sort(edges)

parent = new int[indexes.size()]

for (int i = 0; i < indexes.size(); i++) {

parent[i] = i

}

for (Edge<T> edge : edges) {

int index1 = getIndex(edge.getVertex1())

int index2 = getIndex(edge.getVertex2())

if (find(index1) != find(index2)) {

result.add(edge)

union(index1, index2)

}

}

return result

}

getIndex(vertex) {

if (!indexes.containsKey(vertex)) {

int index = indexes.size()

indexes.put(vertex, index)

}

return indexes.get(vertex)

}

find(i) {

while (i != parent[i]) {

parent[i] = parent[parent[i]]

i = parent[i]

}

return i

}

union(x, y) {

int rootX = find(x)

int rootY = find(y)

parent[rootX] = rootY

}

}

**Pruebas unitarias**

**Paso 6. Preparación de Informes y Especificaciones**

\*\*Análisis de Complejidad Temporal para las Operaciones: \*\*

DFS:

Agregar vértice (addVertex): O(1)

Agregar arista (addEdge): O(1)

Ver matriz (showMatriz): O(V^2) donde V es el número de vértices.

Ver si dos vértices están conectados (seeConex): En el peor caso, O(V + E) donde V es el número de vértices y E es el número de aristas.

Verificar conectividad total (hardlyConnect): O(V^3) en el peor caso.

Prim:

Agregar arista (addEdge): O(1)

Árbol de expansión mínima (minimumSpanningTree): En el peor caso, O((V + E) log V) donde V es el número de vértices y E es el número de aristas.

Kruskal:

Inicialización: O(V log V)

Árbol de expansión mínima (minimumSpanningTree): En el peor caso, O(E log E) donde E es el número de aristas.

Estos análisis proporcionan una idea de cómo aumenta el tiempo de ejecución a medida que el número de vértices (V) y aristas (E) en el grafo aumenta. Las complejidades mencionadas son las peores posibles y pueden variar en la práctica dependiendo de la estructura del grafo y la implementación del algoritmo.

**Class DFS**

**Pruebas para DFS:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Método** | **Escenario** | **Valores de Entrada** | **Resultado Esperado** |
| **seeConex** | **Verificar si dos vértices están conectados en un grafo a través de DFS** | **Vértices iniciales, grafo con conexiones definidas** | **Devuelve true/false según conexión** |
| **hardlyConnect** | **Verificar si un grafo está completamente conectado a través de DFS** | **Grafo con múltiples vértices y conexiones definidas** | **Devuelve true si conectado, false si no conectado** |
| **seeConex (2)** | **Verificar conexión entre dos vértices en un grafo desconectado** | **Vértices no conectados, grafo con conexiones limitadas** | **Devuelve false** |
| **hardlyConnect (2)** | **Verificar conectividad en un grafo desconectado** | **Grafo con vértices aislados y sin conexiones** | **Devuelve false** |

**Pruebas para Prim:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Método** | **Escenario** | **Valores de Entrada** | **Resultado Esperado** |
| **addEdge** | **Agregar una arista a la lista de adyacencia en el algoritmo Prim** | **Vértices y peso de la arista** | **Arista agregada correctamente** |
| **minimumSpanningTree** | **Encontrar el árbol de expansión mínima en un grafo usando el algoritmo Prim** | **Grafo con vértices y aristas definidas** | **Árbol de expansión mínima correctamente encontrado** |
| **minimumSpanningTree (2)** | **Encontrar el árbol de expansión mínima en un grafo grande** | **Grafo grande con múltiples vértices y aristas** | **Árbol de expansión mínima correctamente encontrado** |

**Pruebas para Kruskal:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Método** | **Escenario** | **Valores de Entrada** | **Resultado Esperado** |
| **minimumSpanningTree** | **Encontrar el árbol de expansión mínima en un grafo usando el algoritmo Kruskal** | **Grafo con vértices y aristas definidas** | **Árbol de expansión mínima correctamente encontrado** |
| **minimumSpanningTree (2)** | **Encontrar el árbol de expansión mínima en un grafo desconectado** | **Grafo desconectado con varios componentes** | **Árbol de expansión mínima correctamente encontrado** |
| **minimumSpanningTree (3)** | **Encontrar el árbol de expansión mínima en un grafo grande** | **Grafo grande con múltiples vértices y aristas** | **Árbol de expansión mínima correctamente encontrado** |

**Paso 7. Implementación del Diseño**

Implementación en lenguaje de programación

Lista para implementar:

Especificación de subrutina

|  |  |
| --- | --- |
| Nombre |  |
| Descripcion |  |
| Entrada |  |
| Retorno |  |
|  |  |

Construcción

Escritura del codigo en lenguaje de programación en java