

TAREA INTEGRADORA 3: QGRAPH

JULIAN CAMILO BOLAÑOS

ARIEL EDUARDO PABÓN BOLAÑOS

JUAN FELIPE BLANCO TIGREROS

14 DE NOVIEMBRE DE 2021

ALGORITMOS Y ESTRUCTURAS DE DATOS

UNIVERSIDAD ICESI

1. INFORME DEL MÉTODO DE LA INGENIERÍA

FASE 1: Identificación del problema y requerimientos funcionales:

Contexto del problema:

En los últimos veinte años, la industria de los videojuegos se ha consolidado como un importante contribuyente a la economía mundial del entretenimiento. En 2020, esta tuvo un valor de 159.300 millones de dólares, lo cual supone un aumento del 9.3% con respecto a 2019. Lo anterior denota una diferencia enorme con respecto a la predicción que se tenía en 2016, la cual mostraba un valor total de 90.07 millones de dólares -es decir, una diferencia del 76.8% entre ambas cifras-. Además, se estima que en 2025 la industria tendrá un valor de 268.000 millones de dólares. (Statista y NewZoo, citado por WePC, 2021). Esto supone que emprender y competir en este mercado tiene un gran valor potencial, por lo que se pretende desarrollar y comercializar un videojuego para captar el interés de un público determinado.

Teniendo en cuenta lo anterior, se puede considerar el segmento de mercado al que le interesan los juegos multijugador casuales, ya que es el segundo tipo más popular según Limelight Networks en su reporte anual *The State of Online Gaming* [El estado del juego online] (2021). Los autores mencionan que el 73.6% de los encuestados juegan este tipo de juegos de forma ocasional e incluso más frecuente (pp. 10). Además, también se menciona que:

“Socializing around game play is an important aspect, with 53 percent of gamers saying they made new friends, and 36 percent responded that interactivity with other players is important” [La socialización en torno al

juego es un aspecto importante, ya que el 53% de los jugadores afirma haber hecho nuevos amigos, y el 36% respondió que la interactividad con otros jugadores es importante] (pp. 4).

Lo anterior implica que es importante satisfacer las necesidades de los jugadores que buscan socializar a través de los videojuegos.

En conclusión, la industria de los videojuegos tiene un gran valor potencial, por lo que vale la pena competir en este mercado. Por ello se pretende desarrollar y comercializar un videojuego que capte el interés del público casual que busque experiencias sociales.

Definición del problema:

Para nuestro caso, se considera conveniente apostar por el mercado meta del jugador casual y social. Por ello, se plantea el desarrollo de un videojuego que sea fácil de entender y de jugar para cualquier persona, y que tome como elemento clave la interacción social entre los participantes. En el mercado se pueden encontrar varios videojuegos que cumplen con las características mencionadas. Entre estos se tienen en cuenta a Mario Party (1998), Preguntados (2013), Among Us (2018) y skribbl.io (s.f.) para concretar el tipo de juego que se quiere desarrollar. A continuación mencionamos las características fundamentales que se hallaron:

-Estos son juegos que no tienen una curva de dificultad alta. Por ejemplo, en Mario Party y en Among Us hay minijuegos sencillos que son accesorios de la dinámica principal del juego, por lo que estos no afectan de forma significativa los resultados finales de la partida. Por otro lado, en Preguntados se hacen preguntas de múltiples temas a los participantes, por

lo que es una actividad que no requiere una destreza mecánica importante. Finalmente, si bien es cierto que el dibujo puede llegar a ser una habilidad difícil, en skribbl.io se favorece más la buena comunicación que la calidad de dibujo que se tenga.

-Las mecánicas de estos juegos tienden a enriquecer la interacción social de los participantes, ya que esta es un factor esencial para la experiencia de juego. Por ejemplo, dado que en Mario Party es el azar quien normalmente determina los resultados finales de la partida, se pone más énfasis en las decisiones que afectan el ámbito social. Por otro lado, la interacción con otros es la que determina la condición de victoria en Among Us, ya que hay una dinámica de confianza, de engaño y de cooperación constante, por lo que también favorece este ámbito. Finalmente, en skribbl.io se incentiva la interacción de los participantes por medio del dibujo y de la adivinanza.

-En estos juegos se permite la personalización de las partidas al gusto de los participantes. Por ejemplo, en Mario Party se puede escoger el número de rondas y de participantes, así como la selección de mapas, personajes y otras reglas. Por otro lado, en Among Us se permite customizar ciertos parámetros como el número de actividades que se tienen que realizar, o el tiempo de recarga de ciertas acciones. Finalmente, en Preguntados y en skribbl.io se pueden ingresar preguntas o palabras personalizadas respectivamente.

-Cada uno de los juegos mencionados tiene una personalidad bien definida, con personajes carismáticos o con elementos distintivos que los hacen destacar entre el resto de juegos de sus respectivos géneros.

Por todo lo dicho anteriormente, se quiere desarrollar un videojuego de mesa que tenga una curva de dificultad baja y que enriquezca la dinámica social de los participantes. También se debe tener la opción de personalizar la experiencia de juego y debe tener una personalidad bien definida.

El juego debe permitir que los jugadores interactúen en un mapa compuesto por casillas normales o especiales. Cada casilla debe estar conectada a otras dos como mínimo.

Hay tres tipos de casillas especiales con aristas de mayor coste (peso) que las demás. Estas pueden ser de corona, de cocodrilo, y de incentivo. La casilla de corona otorga una al responder una pregunta correctamente. La casilla de cocodrilo cambia las preguntas de todas las casillas del mapa y el coste de sus aristas. La casilla de incentivo puede tener tres funciones distintas, que son duplicar monedas, robar monedas o robar coronas. Esta requiere responder una pregunta correctamente para acceder a su función.

En cada ronda, los jugadores pueden moverse por el tablero al gastar una cierta cantidad de monedas; esta se determina con el coste de las aristas recorridas. Además, la cantidad máxima de casillas que puede recorrer un jugador está limitada por el resultado del lanzamiento de un dado de 4 caras. Todo esto quiere decir que el jugador podrá moverse lo que indique su dado o menos, dependiendo de cuántas monedas tenga en su posesión y la cantidad de casillas desee avanzar. Al finalizar el movimiento y dependiendo del tipo de casilla, se le hará una pregunta al participante. Si esta es contestada exitosamente, se le devolverá la mitad de las monedas que se gastó para llegar a esa posición.

En cada ronda se recarga un cierto número de monedas a los jugadores. Al acabar la partida se toma en cuenta el mayor número de coronas y de monedas para determinar el ganador.

Finalmente, se podrán elegir las preguntas y los tipos de estas, el mapa, la cantidad de jugadores y de rondas antes de empezar una partida. Las preguntas y los tipos se deben como archivos CSV.

Especificación de requerimientos:

1. Requerimientos funcionales:

RF 1. Mostrar el menú de inicio. El juego debe iniciar con un menú con opciones para jugar, personalizar la partida y salir del juego.

- **RF 1.1. Mostrar el menú de personalización.** Cuando se elige jugar, se muestra un menú para personalizar la partida. Aquí se pueden elegir los bancos de preguntas, el nombre de las categorías, el tablero, la cantidad de jugadores y el número de rondas.
 - **RF 1.1.1 Mostrar el menú de selección y carga de preguntas.**
Al seleccionar los bancos de preguntas, se debe poder cargar un archivo CSV nuevo o utilizar los existentes. Cada uno tiene un listado de preguntas con enunciado, tres opciones de respuesta incorrectas y una correcta, y un número del uno al cuatro como categoría.

RF 2. Inicializar partida.

- **RF 2.1. Inicializar tablero.** Se asigna un tipo (incentivo, corona, cocodrilo), una categoría de preguntas (entero del 1 al 4) y el coste de las aristas a cada casilla del tablero. Los tipos especiales tienen aristas con costos mayores que los normales.
- **RF 2.2. Crear e inicializar jugadores.** Se crean los jugadores y se les asigna una casilla de partida y una cantidad inicial de monedas.

RF 3. Mostrar el tablero y la interfaz de usuario.

- **RF 3.1. Mostrar el tablero.** Se muestra el estado actual de las casillas y sus aristas, y la posición de los jugadores.
- **RF 3.2. Mostrar la interfaz de usuario.** Mostrar a los jugadores el número de monedas y de coronas que poseen. También se deben mostrar botones para lanzar el dado y para salir del juego, y una guía para que el jugador sepa a qué casilla puede moverse. Al finalizar la última ronda de la partida, se muestra una pantalla que indica al ganador.

RF 4. Usar las mecánicas de juego.

- **RF 4.1. Lanzar dado.** En cada turno se podrá lanzar un dado de cuatro caras, el cual determina el número máximo de casillas que un jugador puede desplazarse.
- **RF 4.2. Mover al jugador a la casilla seleccionada.** Permitir al jugador desplazarse a una casilla tras lanzar el dado. Este se puede mover a las que tengan un coste igual o menor que el número de monedas poseídas. Además, no se puede mover a las casillas que ha recorrido

en el turno anterior. Al finalizar el movimiento, se le descuenta el coste de este camino.

- **RF 4.3. Realizar preguntas.** Tras el desplazamiento de un jugador, se realiza una pregunta si este cae en una casilla de cierto tipo y categoría. Al responder correctamente, se devuelve la mitad de monedas utilizadas para llegar a la casilla.
- **RF 4.4. Recargar monedas por ronda.** Al inicio de cada ronda se recarga una cierta cantidad de monedas a los jugadores.
- **RF 4.5. Usar casillas especiales.**
 - **RF 4.5.1. Usar casillas de incentivo.** Si el jugador se desplaza a una casilla especial y responde a la pregunta correctamente, se le dará una recompensa de incentivo. Estas pueden ser duplicar o robar monedas, y robar coronas.
 - **RF 4.5.1.1 Duplicar monedas.** Se duplican las monedas del jugador actual.
 - **RF 4.5.1.2 Robar monedas.** Se escoge un jugador de la partida para robar una cierta cantidad de dinero. Si este no tiene dinero, entonces no ocurre nada.
 - **RF 4.5.1.3 Robar coronas.** Se escoge un jugador de la partida para robar una cierta cantidad de coronas. Si este no tiene coronas, entonces no ocurre nada.
 - **RF 4.5.2. Usar casilla de corona.** Al contestar correctamente una pregunta, se incrementará en una unidad el número de coronas del jugador. Posteriormente, la corona se moverá a otra parte del tablero. La casilla escogida no debe ser de tipo especial, no debe contener jugadores y no debe ser adyacente a la anterior. Los costes de las aristas cambiarán de acuerdo al

nuevo tipo de casillas que queden en el mapa; la casilla normal y la de corona pierden y ganan coste respectivamente.

- **RF 4.5.3. Usar casilla de cocodrilo.** Al desplazarse a esta casilla, se cambian las categorías y los tipos de las casillas, así como el coste de las aristas del tablero. La corona permanece estática.
- **RF 4.6. Determinar al ganador.** Al finalizar la última ronda, se determina al jugador con mayor número de coronas y de monedas como ganador. Si hay dos participantes con la misma cantidad, entonces se determinan dos ganadores.

2. *Requerimientos no funcionales:*

RNF 1. Funcionalidad con archivos CSV.

FASE 2: Recopilación de la información necesaria:

Para implementar la solución planteada, es necesario definir las estructuras de datos que se van a utilizar. En este caso vamos a darle un vistazo a la teoría de grafos.

Teoría de grafos:

Los grafos son estructuras discretas que están compuestas por vértices y aristas que los conectan entre sí. Los grafos son utilizados para modelar problemas de la vida real como redes de transferencia de datos, carreteras, entre otros. Existen distintos tipos de grafos que son los simples, los multigrafos, los pseudografos, los grafos dirigidos y los multigrafos dirigidos.

Grafos simples: Son un conjunto $G = (V, E)$ donde V es un conjunto no vacío de vértices y E es un conjunto de aristas (pares no ordenados de elementos V), las cuales no pueden poseer el mismo vértice de inicio y fin; esto es $\{\{u, v\} \mid u, v \in V, u \neq v\}$.

Multigrafos: Son un conjunto $G = (V, E)$ donde V es un conjunto no vacío de vértices y E es un conjunto de aristas (pares no ordenados de elementos V), las cuales no pueden poseer el mismo vértice de inicio y fin, y permite la repetición de las aristas.

Pseudografos: Son un conjunto $G = (V, E)$ donde V es un conjunto no vacío de vértices y E es un conjunto de aristas donde los pares sí pueden repetirse, es decir, estas pueden poseer el mismo vértice de inicio y fin; esto es $\{\{u, v\} \mid u, v \in V\}$.

Grafos dirigidos: Son un conjunto $G = (V, E)$ donde V es un conjunto no vacío de vértices y E es un conjunto de aristas que poseen pares ordenados de vértices adyacentes desde o hacia a otro.

Multigrafos dirigidos: Son un conjunto $G = (V, E)$ donde V es un conjunto no vacío de vértices y E es un conjunto de aristas que poseen pares ordenados de vértices adyacentes desde o hacia a otro, e igualmente permite los bucles en su composición.

Los grafos poseen una serie de algoritmos que son importantes para resolver ciertos tipos de problemas. A continuación se listan la búsqueda por amplitud y por profundidad, el algoritmo de Dijkstra y Floyd Warshall.

Búsqueda por amplitud: Este es uno de los algoritmos más simples de búsqueda en grafos. Se empieza con un vértice s de origen y se exploran todos los que son adyacentes a él. Cuando acaba, se computa la distancia desde el inicio a cada vértice alcanzable. Este algoritmo tiene una complejidad temporal de $O(V + E)$.

Búsqueda por profundidad: Es un algoritmo que permite recorrer un grafo al expandirse por cada una de las aristas de un nodo. Se empieza con un vértice s de origen y se continúa con los otros que se van descubriendo. Si al acabar existen nodos sin explorar, se selecciona alguno de ellos y se repite el proceso. Este algoritmo tiene una complejidad temporal de $O(V + E)$.

Algoritmo de Dijkstra: Es un algoritmo muy conocido por su capacidad para hallar los caminos mínimos desde un nodo origen hacia el resto de los nodos de un grafo ponderado. Consiste en tener una lista con las distancias de cada vértice del grafo, otra con los predecesores para cada uno y una cola de prioridad mínima. El procedimiento comienza desde el primer nodo de la cola, encontrando la distancia de él a todos sus adyacentes, editando la lista de distancias y predecesores. Una vez se acaba una iteración se ajusta la cola de prioridad con base en las distancias de los nodos y se repite el proceso. Este algoritmo posee una complejidad temporal de $O(V E \log V)$, siendo n el número de vértices del grafo y E sus aristas.

Algoritmo de Floyd Warshall: Permite encontrar el camino de coste mínimo entre todos los pares de vértices de un grafo ponderado (pesos enteros). Se empieza con una matriz de $n \times n$ cuyas filas y columnas son los vértices, y se toma en cuenta una función minPath . Esta última retorna el menor camino posible entre nodos i y j usando vértices de un conjunto K . Al finalizar, se

obtiene una matriz con los caminos de coste mínimo. La complejidad temporal de este algoritmo es $O(n^3)$, donde n es la cantidad de vértices.

FASE 3: Búsqueda de soluciones creativas:

Se toman el rango de movimiento del jugador y el desplazamiento con menor coste como los problemas principales del juego. Luego, se formulan las posibles soluciones con la técnica de lluvia de ideas.

Modelado del tablero.

Se desea modelar el tablero de tal forma que se pueda implementar los costes de los movimientos junto con otras mecánicas listadas en los requerimientos funcionales.

Alternativa 1. Matrices: Se plantea utilizar una matriz para representar el tablero de juego, en la que las posiciones tienen una casilla asociada.

Alternativa 2. Grafos: Se plantea utilizar grafos para representar el tablero de juego.

Alternativa 3. Matriz de listas enlazadas: Se plantea utilizar una matriz de listas enlazadas para representar el tablero de juego. Cada elemento es una casilla que tiene enlaces a otras, y por cada una de ellas hay un coste asociado.

Calcular el rango de movimiento de un jugador.

Se desea calcular el rango de movimiento de un jugador teniendo en cuenta la distancia de un vértice a otro.

Alternativa 1. Búsqueda por amplitud: Realizar una búsqueda por amplitud para determinar la distancia de la casilla a las demás.

Alternativa 2. Prueba y error: Ensayar cada uno de los caminos posibles desde la casilla inicial para encontrar a los demás nodos. Luego, se verifica si el camino es menor o igual al rango de movimiento. Este proceso tiene una complejidad temporal de $O(n!)$.

Calcular el desplazamiento con menor coste a la casilla seleccionada.

Se desea calcular el desplazamiento con menor coste al vértice al que el jugador elige desplazarse.

Alternativa 1. Algoritmo de Dijkstra: Emplear el algoritmo de Dijkstra para hallar los caminos de menor coste desde la casilla inicial hasta las demás.

Alternativa 2. Medición por tiempo: Medir el tiempo que se demora el jugador en recorrer cada uno de los caminos que dan hasta la casilla escogida. Luego, se toma el camino con menor tiempo para desplazar al jugador.

Alternativa 3. Algoritmo Floyd-Warshall: Emplear el algoritmo de Floyd-Warshall para hallar los caminos más cortos entre todos los pares de casillas del tablero.

FASE 4: Transición de la formulación de ideas a diseños preliminares:

En el desarrollo de esta fase se busca descartar las ideas que no resultan factibles para la solución del problema o que son imposibles de implementar. Con esto en mente, descartamos:

A. La alternativa 3 para el **modelado del tablero**, ya que acceder a la información de una casilla tiene una complejidad temporal de $O(n^2)$. Esto no resulta factible para un tablero de grandes dimensiones, ya que se espera utilizar esta información de forma recurrente. Lo anterior plantea un problema que puede afectar la experiencia de usuario de forma significativa.

B. La alternativa 2 para el **rango de movimiento**, ya que este proceso tendría una complejidad temporal de $O(n!)$. Esto no resulta factible para un tablero de grandes dimensiones por las mismas razones del punto anterior.

C. La alternativa 2 para el **cálculo del desplazamiento con menor coste**, porque no se espera que solucione el problema al tener en cuenta algo tan variable como el tiempo de ejecución.

La revisión cuidadosa de las otras alternativas nos conduce a lo siguiente:

Modelado del tablero:

Alternativa 1. Matrices:

- En este caso, se utiliza una matriz para representar el tablero de juego. Cada una de las posiciones tienen una casilla asociada, y estas tienen un atributo de coste. Lo anterior se emplea para determinar el movimiento del jugador.
- El juego se jugaría en una cuadrícula de $n \times n$ que brinda un amplio espacio de acción para los jugadores. Esto puede llegar a ser confuso o poco interesante para el tipo de jugador que se busca.
- Se crea un tablero para cada ronda jugada.

Alternativa 2. Grafos:

- En este caso, se utilizan grafos para representar el tablero de juego. Cada vértice tiene una casilla asociada y las aristas representan el coste de movimiento para el jugador.
- El juego se jugaría en un “mapa” que limita el espacio de acción del jugador, lo cual puede presentarle opciones más claras de movimiento a los participantes.
- Se crea un tablero para cada ronda jugada.

Calcular el rango de movimiento de un jugador.

Alternativa 1. Búsqueda por amplitud:

- En este caso, se realiza una búsqueda por amplitud para determinar la distancia de la casilla a las demás. Luego, se compara con un valor máximo de rango (determinado por el dado) para descartar las que no están disponibles en el desplazamiento.
- Este proceso tiene una complejidad temporal de $O(V + E)$, donde V es el conjunto de casillas y E el conjunto de aristas.

Calcular el desplazamiento con menor coste a la casilla seleccionada.

Alternativa 1. Algoritmo de Dijkstra:

- Emplear el algoritmo de Dijkstra para hallar los caminos de menor coste desde la casilla inicial hasta las demás. Luego, se utilizan para hacer el movimiento del jugador.

- Este proceso tiene una complejidad temporal de $O(n \cdot E \log n)$, siendo n el número de casillas del tablero y E el conjunto de aristas.
- Este algoritmo se realiza tantas veces como turnos en una partida.

Alternativa 3. Algoritmo Floyd-Warshall:

- En este caso, se emplea el algoritmo de Floyd-Warshall para hallar los caminos más cortos entre todos los pares de casillas del tablero.
- Esto serviría con pesos enteros, es decir, con valores positivos y negativos.
- El proceso tiene una complejidad temporal de $O(n^3)$, donde n es la cantidad de casillas.
- Este algoritmo se realiza al principio de la partida y cada vez que se pisa la casilla de cocodrilo.

FASE 5: Evaluación de ideas y selección de la mejor solución:

En esta fase se quieren definir los criterios que nos permitirán evaluar y escoger alguna de las alternativas de solución planteadas. A continuación enumeramos los escogidos, junto con un valor que describe su deseabilidad:

Criterios de Evaluación:

Criterio A. Eficiencia: Se prefiere una solución con mejor eficiencia que las otras consideradas.

Criterio B. Pertinente. Se prefiere una solución que se asemeje más a los requerimientos funcionales y a la problemática. Esta puede ser:

1. Pertinente.

2. No es pertinente.

Evaluación:

Modelado del tablero:

	Criterio A	Criterio B
Alternativa 1	$O(1)$	No es pertinente
Alternativa 2	$O(V + E)$	Pertinente

Desplazamiento con menor coste:

	Criterio A	Criterio B
Alternativa 1	#Turnos * $O(n E \log n)$	Pertinente
Alternativa 3	#Cambios de tablero * $O(n^3)$	Pertinente

Soluciones decididas:

Para el caso del **modelado del tablero**, se tiene que la alternativa 1 tiene mejor complejidad temporal que la alternativa 2 a la hora de hacer búsquedas. Sin embargo, esta última es más pertinente de implementar, ya que se asemeja más al tipo de tablero que se quiere tener en el juego. Por ende, se toma la alternativa 2 como solución a este problema.

Por su parte, para el caso del **desplazamiento con menor coste** nos resulta más factible la alternativa 1, ya que es más eficiente para el caso de uso esperado del juego.

2. DISEÑOS TAD

TAD SimpleGraph			
SimpleGraph = {vertices = <V>, edges = <E>}			
{inv: <i>vertices</i> ∈ <i>List</i> ; { <i>v</i> <i>v</i> ∈ <i>V</i> → <i>v</i> ≠ null ∧ <i>v</i> ∈ SimpleVertex}; <i>edges</i> ∈ <i>Matrix</i> ; { <i>e_{ij}</i> <i>i, j</i> ∈ Z_0^+ ∧ <i>e_{ij}</i> ∈ <i>edges</i> , <i>e_{ij}</i> ≥ 0 } }			
Operaciones primitivas:			
- createSimpleGraph			→
SimpleGraph			
- BFS	SimpleGraph x Value		→
SimpleGraph			
- DFS	SimpleGraph		→
SimpleGraph			
- Dijkstra	SimpleGraph x Value		→List
- FloydWarshall	SimpleGraph		→Matrix
- Kruskal	SimpleGraph		→Tree
- degreeOf	SimpleGraph x Vertex		→Value
- getVertices	SimpleGraph		→List
- getEdges	SimpleGraph		→List x List

- | | | |
|------------------|-------------------------------------|----------|
| - containsVertex | SimpleGraph x Value | →Boolean |
| - setVertices | SimpleGraph x List | → |
| SimpleGraph | | |
| - setEdges | SimpleGraph x Matrix | → |
| SimpleGraph | | |
| - addVertex | SimpleGraph x Value | → |
| SimpleGraph | | |
| - addEdge | SimpleGraph x Value x Value x Value | → |
| SimpleGraph | | |

createSimpleGraph()

Crea un grafo simple vacío. El listado de vértices se define como vacío y la matriz de aristas *edges* es vacía.

{pre: True}

{pos: SimpleGraph: {vertices = <v>, edges = <e>}}

BFS(simpleGraph, value)

Realiza una búsqueda por anchura en el grafo *simpleGraph*, partiendo como base del vértice con valor *value*.

{pre: value ≠ null ∧ simpleGraph ≠ null}

{pos: SimpleGraph: {vertices = <v>, edges = <e>}}

DFS(*simpleGraph*)

Realiza una búsqueda por profundidad en el grafo *simpleGraph*.

{pre: *simpleGraph* ≠ null}

{pos: SimpleGraph: {vertices = <v>, edges = <e>}} }

Dijkstra(*simpleGraph*, *value*)

Busca el camino mínimo entre el vértice con valor *value* y todos los demás vértices del grafo *simpleGraph*. Retorna un arreglo *prev* con todos los vértices previos para llegar a dicho vértice partiendo desde *value*. El tamaño de *prev* es acorde a la cantidad de vértices y sus índices dependen de los valores de los mismos (El vértice previo para llegar hasta *i* partiendo desde *value* es el valor de *prev* en *i*).

{pre: *value* ≠ null ∧ *simpleGraph* ≠ null}

{pos: *prev*}

Kruskal(*simpleGraph*)

Encuentra el árbol de recubrimiento mínimo *tree* del grafo *simpleGraph*.

{pre: *simpleGraph* ≠ null}

{pos: *tree*}

FloydWarshall(*simpleGraph*)

Encuentra el valor del camino mínimo entre todos los pares de vértices del grafo.

Retorna una matriz cuadrada *Dist* con esta información.

{pre: simpleGraph \neq null}

{pos: Dist}

degreeOf(simpleGraph, vertex)

Calcula el grado g de un vértice *vertex* del grafo *simpleGraph*.

{pre: vertex \neq null \wedge simpleGraph \neq null}

{pos: g}

getVertices(simpleGraph)

Retorna los vértices del grafo *simpleGraph*.

{pre: simpleGraph \neq null}

{pos: simpleGraph.vertices}

getEdges(simpleGraph)

Retorna las aristas del grafo *simpleGraph*.

{pre: simpleGraph \neq null}

{pos: simpleGraph.edges}

containsVertex(simpleGraph, value)

Retorna True si el grafo *simpleGraph* contiene un vértice con el valor *value*.

{pre: *simpleGraph* ≠ null}

{pos: True si contiene el valor, False de lo contrario}

setVertices(*simpleGraph*, *vertices*)

Define el listado de vértices *vertices* como el conjunto de vértices del grafo *simpleGraph*.

{pre: *vertices* ≠ null ∧ *simpleGraph* ≠ null}

{pos: SimpleGraph: {vertices = <*vertices*>, edges = <*e*> } }

setEdges(*simpleGraph*, *edges*)

Define la matriz de aristas *edges* como la matriz de adyacencia del grafo *simpleGraph*.

{pre: *edges* ≠ null ∧ *simpleGraph* ≠ null}

{pos: SimpleGraph: {vertices = <*v*>, edges = <*edges*> } }

addVertex(*simpleGraph*, *value*)

Añade un vértice *vertex* con valor *value* al listado de vértices de *simpleGraph*, al igual que añade una columna a la matriz de adyacencia *edges*.

{pre: *value* ≠ null ∧ *simpleGraph* ≠ null}

{pos: SimpleGraph: {vertices = <*v*>, edges = <*e*> } }

addEdge(simpleGraph, value1, value2, weight)

Añade una arista de peso *weight* entre los vértices con valor *value1* y *value2*. En caso de que la arista ya exista, sobrescribe el valor de la anterior con la nueva.

{pre: $value1 \neq null \wedge value2 \neq null \wedge weight \geq 0 \wedge simpleGraph \neq null$ }

{pos: SimpleGraph: {vertices = <v>, edges = <e>}}

TAD SimpleVertex

SimpleVertex = {value = <v>, id = <i>, dist = <d>}

{inv:

value, id $\neq null$

dist ≥ 0

}

Operaciones primitivas:

- createSimpleVertex	Value x Value	→ SimpleVertex
- getValue	SimpleVertex	→ Value
- getId	SimpleVertex	→ Value
- getDist	SimpleVertex	→ Value
- setValue	SimpleVertex x Value	→ SimpleVertex
- setId	SimpleVertex x Value	→ SimpleVertex
- setDist	SimpleVertex x Value	→ SimpleVertex

createSimpleVertex(v, i)

Crea un vértice con valor v y con id i . *dist* se inicia en null.

{pre: $v \neq \text{null} \wedge i \neq \text{null}$ }

{pos: SimpleVertex: {value = $\langle v \rangle$, id = $\langle i \rangle$, dist = $\langle \text{null} \rangle$ }}

getValue(simpleVertex)

Retorna el valor del vértice *simpleVertex*.

{pre: $\text{simpleVertex} \neq \text{null}$ }

{pos: $\text{simpleVertex.value}$ }

getId(simpleVertex)

Retorna el id del vértice *simpleVertex*.

{pre: $\text{simpleVertex} \neq \text{null}$ }

{pos: simpleVertex.id }

getDist(simpleVertex)

Retorna la distancia del vértice *simpleVertex*.

{pre: $\text{simpleVertex} \neq \text{null}$ }

{pos: simpleVertex.dist }

setValue(simpleVertex, val)

Define el valor *val* como el valor del vértice *simpleVertex*.

{pre: $val \neq \text{null} \wedge \text{simpleVertex} \neq \text{null}$ }

{pos: *simpleVertex*: {value = <val>, id = <i>, dist = <d>}} }

setId(*simpleVertex*, idn)

Define *idn* como el nuevo id del vértice *simpleVertex*.

{pre: $\text{idn} \neq \text{null} \wedge \text{simpleVertex} \neq \text{null}$ }

{pos: *simpleVertex*: {value = <v>, id = <idn>, dist = <d>}} }

setDist(*simpleVertex*, distn)

Define la distancia *distn* como la nueva distancia del vértice *simpleVertex*.

{pre: $\text{distn} \neq \text{null} \wedge \text{simpleVertex} \neq \text{null}$ }

{pos: *simpleVertex*: {value = <v>, id = <i>, dist = <distn>}} }

TAD ListGraph

ListGraph = {vertices = <V>}

{inv:

$V \subseteq \text{List}; \{v \mid v \in V, v \neq \text{null}\};$

}

Operaciones primitivas:

- createListGraph		→ListGraph
- BFS	ListGraph x Value	→ListGraph
- DFS	ListGraph	→ListGraph
- Dijkstra	ListGraph x Value	→List
- FloydWarshall	ListGraph	→Matrix
- Kruskal	ListGraph	→Tree
- degreeOf	ListGraph x Vertex	→Value
- getVertices	ListGraph	→List
- containsVertex	ListGraph x Value	→Boolean
- setVertices	ListGraph x List	→ListGraph
- addVertex	ListGraph x Value	→ListGraph
- addEdge	ListGraph x Value x Value x Value	→ListGraph

createListGraph()

Crea un listGraph vacío. El listado de vértices se define como vacío.

{pre: True}

{pos: listGraph: {vertices = <v>}}

BFS(listGraph, value)

Realiza una búsqueda por anchura en el grafo *listGraph*, partiendo como base del vértice con valor *value*.

{pre: value ≠ null ∧ listGraph ≠ null}

{pos: listGraph: {vertices = <v>}}

DFS(listGraph)

Realiza una búsqueda por profundidad en el grafo *listGraph*.

{pre: listGraph \neq null}

{pos: listGraph : {vertices = <v>} }

Dijkstra(listGraph, value)

Busca el camino mínimo entre el vértice con valor *value* y todos los demás vértices del grafo *listGraph*. Retorna un arreglo *prev* con todos los vértices previos para llegar a dicho vértice partiendo desde *value*. El tamaño de *prev* es acorde a la cantidad de vértices y sus índices dependen de los valores de los mismos (El vértice previo para llegar hasta *i* partiendo desde *value* es el valor de *prev* en *i*).

{pre: value \neq null \wedge listGraph \neq null}

{pos: prev}

Kruskal(listGraph)

Encuentra el árbol de recubrimiento mínimo *tree* del grafo *listGraph*.

{pre: listGraph \neq null}

{pos: tree}

FloydWarshall(listGraph)

Encuentra el valor del camino mínimo entre todos los pares de vértices del grafo.
Retorna una matriz cuadrada *Dist* con esta información.

{pre: listGraph \neq null}

{pos: Dist}

degreeOf(listGraph, vertex)

Calcula el grado g de un vértice *vertex* del grafo *listGraph*.

{pre: vertex \neq null \wedge listGraph \neq null}

{pos: g }

getVertices(listGraph)

Retorna los vértices del grafo *listGraph*.

{pre: listGraph \neq null}

{pos: listGraph.vertices}

containsVertex(listGraph, value)

Retorna True si el grafo *listGraph* contiene un vértice con el valor *value*.

{pre: listGraph \neq null}

{pos: True si contiene el valor, False de lo contrario}

setVertices(listGraph, vertices)

Define el listado de vértices *vertices* como el conjunto de vértices del grafo *listGraph*.

{pre: $vertices \neq null \wedge listGraph \neq null$ }

{pos: $listGraph : \{vertices = \langle vertices \rangle\}$ }

addVertex(listGraph, value)

Añade un vértice *vertex* con valor *value* al listado de vértices de *listGraph*.

{pre: $value \neq null \wedge simpleGraph \neq null$ }

{pos: $SimpleGraph : \{vertices = \langle v \rangle\}$ }

addEdge(listGraph, value1, value2, weight)

Añade una arista de peso *weight* entre los vértices con valor *value1* y *value2*. En caso de que la arista ya exista, sobrescribe el valor de la anterior con la nueva.

{pre: $value1 \neq null \wedge value2 \neq null \wedge weight \geq 0 \wedge listGraph \neq null$ }

{pos: $listGraph : \{vertices = \langle v \rangle\}$ }

TAD ListVertex

ListVertex = {value = $\langle v \rangle$, id = $\langle i \rangle$, prev = $\langle pi \rangle$, dist = $\langle d \rangle$, edges = $\langle E \rangle$ }

{inv:

value, id $\neq null$

$dist \geq 0$

$edges \in List; \{e | e \in E \rightarrow e \in ListEdge \wedge e \neq null\}$

}

Operaciones primitivas:

- createListVertex	Value x Value	→ListVertex
- getValue	ListVertex	→Value
- getId	ListVertex	→Value
- getDist	ListVertex	→Value
- getEdges	ListVertex	→ListEdge
- setValue	ListVertex x Value	→ListVertex
- setId	ListVertex x Value	→ListVertex
- setDist	ListVertex x Value	→ListVertex
- setEdges	ListVertex x List	→ListVertex
- addAdjacent	ListVertex x ListVertex	→ListVertex

createListVertex(v, i)

Crea un vértice con valor v y con id i . $dist$ se inicia en null y la lista de adyacentes como vacía.

{pre: $v \neq null \wedge i \neq null$ }

{pos: listVertex: {value = $\langle v \rangle$, id= $\langle i \rangle$, dist = $\langle null \rangle$, edges = $\langle a \rangle$ } }

getValue(listVertex)

Retorna el valor del vértice $listVertex$.

{pre: listVertex ≠ null}

{pos: listVertex.value}

getId(listVertex)

Retorna el id del vértice *listVertex*.

{pre: listVertex ≠ null}

{pos: listVertex.id}

getDist(listVertex)

Retorna la distancia del vértice *listVertex*.

{pre: listVertex ≠ null}

{pos: listVertex.dist}

getEdges(listVertex)

Retorna la lista de vértices aristas de *listVertex*.

{pre: listVertex ≠ null}

{pos: listVertex.edges}

setValue(listVertex, val)

Define el valor *val* como el valor del vértice *listVertex*.

{pre: $val \neq \text{null} \wedge \text{listVertex} \neq \text{null}$ }

{pos: listVertex: {value = <val>, id = <i>, dist = <d>, edges = <e>}} }

setId(listVertex, idn)

Define *idn* como el nuevo id del vértice *listVertex*.

{pre: idn \neq null \wedge listVertex \neq null}

{pos: listVertex: {value = <v>, id = <idn>, dist = <d>, edges = <e>}} }

setDist(listVertex, distn)

Define la distancia *distn* como la nueva distancia del vértice *listVertex*.

{pre: distn \neq null \wedge listVertex \neq null}

{pos: listVertex: {value = <v>, id = <i>, dist = <distn>, edges = <e>}} }

setDist(listVertex, en)

Define la lista *aen* como la nueva lista de aristas del vértice *listVertex*.

{pre: en \neq null \wedge listVertex \neq null}

{pos: listVertex: {value = <v>, id = <i>, dist = <d>, edges = <en>}} }

addEdge(listVertex, newEdge)

Añade una nueva arista al listado de aristas del vértice *listVertex*.

{pre: newEdge \neq null \wedge listVertex \neq null}

{pos: listVertex: {value = <v>, id = <i>, dist = <d>, edges = <e>}} }

TAD ListEdge

ListEdge = {beginning = <v1>, end = <v2>, weight = <w>}

inv = {
beginning, *end* \neq null
weight \geq 0
}

Operaciones primitivas:

- createListEdge	ListVertex x ListVertex x Value	→ListEdge
- getBeginnig	ListEdge	→ListVertex
- getEnd	ListEdge	→ListVertex
- getWeight	ListEdge	→Value
- setBeginning	ListEdge x ListVertex	→ListEdge
- setEnd	ListEdge x ListVertex	→ListEdge
- setWeight	ListEdge x Value	→ListEdge

createListEdge(beg, end, w)

Crea una arista con inicio *beg* y final *end*. El peso de dicha arista es *w*.

{pre: *beg*, *end* \neq null \wedge *beg*, *end* \in ListVertex \wedge *w* \geq 0}

{pos: listEdge: {beginning = <beg>, end = <end>, weight = <w>}} }

getBeginning(listEdge)

Retorna el vértice *beginning* de la arista *listEdge*.

{pre: listEdge \neq null}

{pos: listEdge.beginning }

getEnd(listEdge)

Retorna el vértice *end* de la arista *listEdge*.

{pre: listEdge ≠ null}

{pos: listEdge.end }

getWeight(listEdge)

Retorna el peso *weight* de la arista *listEdge*.

{pre: listEdge ≠ null}

{pos: listEdge.weight}

setBeginning(listEdge, beg)

Define *beg* como el nuevo vértice inicial de la arista *listEdge*.

{pre: beg, listEdge ≠ null ∧ beg ∈ ListVertex}

{pos: listEdge: {beginning = <beg>, end = <v2>, weight = <w>}} }

setEnd(listEdge, end)

Define *end* como el nuevo vértice final de la arista *listEdge*.

{pre: end, listEdge ≠ null ∧ end ∈ ListVertex}

{pos: listEdge: {beginning = <v1>, end = <end>, weight = <w>}} }

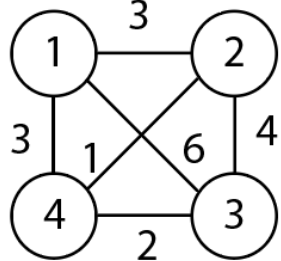
setWeight(listEdge, wn)

Define *wn* como el nuevo peso de la arista *listEdge*.

{pre: listEdge ≠ null ∧ $wn \geq 0$ }

{pos: listEdge: {beginning = <v1>, end = <v2>, weight = <wn>}}

3. DISEÑO DEL DIAGRAMA DE CLASES**4. DISEÑO DE LOS CASOS DE PRUEBA**

Nombre	Clase	Escenario
SGSC1	SimpleGraphTest	Grafo vacío, sin vértices o aristas.
SGSC2	SimpleGraphTest	Grafo con 1 vértice. Este tiene su valor asignado a 1.
SGSC3	SimpleGraphTest	

Objetivo de la Prueba: Verificar si se añade un vértice al grafo con sus aristas vacías.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	addVertex	SGSC1	value = 1	La lista de vértices del arreglo ahora posee uno con valor = 1 y la lista de aristas posee una lista cuyo primer valor es 0.
Simple Graph	addVertex	SGSC2	value = 3	La lista de vértices del arreglo ahora posee dos con valor = 1 y 3 respectivamente, igualmente, la lista de aristas posee dos listas cuyos valores son un número máximo y por último un peso igual a 0.
Simple Graph	addVertex	SGSC3	value = 5	La lista de vértices del arreglo ahora posee 5 elementos con valor = 1, 2, 3, 4 y 5 respectivamente, igualmente, la lista de aristas posee dos listas cuyos valores son números máximos y pesos iguales a 0.

Objetivo de la Prueba: Verificar si se añade un vértice al grafo con sus aristas vacías.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	deleteVertex	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices.
Simple Graph	deleteVertex	SGSC2	Un vértice existente en el grafo.	Se elimina el vértice dado de la lista de vértices y se eliminan las aristas desde y hacia a este. En este caso se tiene un grafo vacío como resultado.
Simple Graph	deleteVertex	SGSC3	Un vértice existente en el grafo.	Se elimina el vértice dado de la lista de vértices y se eliminan las aristas desde y hacia a este. Al actualizar la

				matriz de adyacencia, esta tiene una fila y una columna menos.
--	--	--	--	--

Objetivo de la Prueba: Verificar si se añade una arista entre dos vértices con un peso dado

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	addEdge	SGSC1	Dos vértices existentes en el grafo. weight = 10;	No hay cambios en la estructura del grafo ni en sus vértices.
Simple Graph	addEdge	SGSC2	Dos vértices existentes en el grafo. weight = 10;	No hay cambios en la estructura del grafo ni en sus vértices.
Simple Graph	addEdge	SGSC3	Dos vértices existentes en el grafo. weight = 10;	Se asigna una arista con peso 10 entre el vértice 1 y el vértice 2. La matriz de adyacencia cambia este valor en la posición (i, j) teniendo en cuenta los IDs de los vértices.

Objetivo de la Prueba: Verificar si se asignan las distancias y predecesores correctos a cada vértice.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	breadthFirstSearch	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.

Simple Graph	breadthFirstSearch	SGSC2	Primer y único vértice de la lista	Distancia del único vértice = 0. Sigue sin predecesor.
Simple Graph	breadthFirstSearch	SGSC3	Segundo vértice en la lista con valor = 2	Distancias = [1,0,1,1] Predecesores = [2, nulo,2,2]

Objetivo de la Prueba: Verificar si se añaden los tiempos adecuados a cada vértice según el recorrido por profundidad.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	depthFirstSearch	SGSC1		No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.
Simple Graph	depthFirstSearch	SGSC2		timestamps = [1,2]
Simple Graph	depthFirstSearch	SGSC3		timestamps = [[1,8],[2,7],[3,6],[4,5]]

Objetivo de la Prueba: Verificar si se halla el camino mínimo desde un vértice hasta todos los demás.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	dijkstra	SGSC1	Un vértice cualquiera	[null]
Simple Graph	dijkstra	SGSC2	Primer y único vértice de la lista	[null]
Simple Graph	dijkstra	SGSC3	Segundo vértice en la	[1, null, 3, 1]

			lista con valor = 2	
--	--	--	------------------------	--

Objetivo de la Prueba: Verificar si se encuentran los caminos mínimos entre todos los vértices

Clase	Método	Escenario	Valores de Entrada	Resultado																									
Simple Graph	floydWarshall	SGSC1		No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.																									
Simple Graph	floydWarshall	SGSC2		Se calculan los caminos de peso mínimo utilizando como base una copia de la matriz de adyacencia, asignando el resultado en minimumWeightPaths. En este caso el resultado es una matriz de 1x1 con un valor de 0.																									
Simple Graph	floydWarshall	SGSC3		<p>Se calculan los caminos de peso mínimo utilizando como base una copia de la matriz de adyacencia, asignando el resultado en minimumWeightPaths.</p> <table> <tr> <td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr> <td>1</td><td>0</td><td>3</td><td>5</td><td>3</td></tr> <tr> <td>2</td><td>3</td><td>0</td><td>3</td><td>1</td></tr> <tr> <td>3</td><td>5</td><td>3</td><td>0</td><td>2</td></tr> <tr> <td>4</td><td>3</td><td>1</td><td>2</td><td>0</td></tr> </table>		1	2	3	4	1	0	3	5	3	2	3	0	3	1	3	5	3	0	2	4	3	1	2	0
	1	2	3	4																									
1	0	3	5	3																									
2	3	0	3	1																									
3	5	3	0	2																									
4	3	1	2	0																									

Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	prim	SGSC1		
Simple Graph	prim	SGSC2		
Simple Graph	prim	SGSC3		

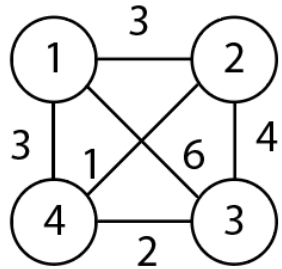
Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	kruskal	SGSC1	Nada	null
Simple Graph	kruskal	SGSC2	Nada	null
Simple Graph	kruskal	SGSC3	Nada	[2,4,1][3,4,2][1,4,3][1,2,3]

Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Simple Graph	degreeOf	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.

Simple Graph	degreeOf	SGSC2	Un vértice existente en el grafo.	Se cuenta la cantidad de vértices adyacentes que tiene la entrada. En este caso se retorna 0.
Simple Graph	degreeOf	SGSC3	Un vértice existente en el grafo.	Se cuenta la cantidad de vértices adyacentes que tiene la entrada. En este caso se retorna 3 para cualquiera de ellos.

Nombre	Clase	Escenario
LGSC1	ListGraphTest	Grafo vacío, sin vértices o aristas.
LGSC2	ListGraphTest	Grafo con 1 vértice. Este tiene su valor asignado a 1.
LGSC3	ListGraphTest	

Objetivo de la Prueba: Verificar si se añade un vértice al grafo con sus aristas vacías.				
Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	addVertex	SGSC1	value = 1	La lista de vértices del arreglo ahora posee uno con valor = 1 y la lista de aristas posee una lista cuyo primer valor es 0.

ListGraph	addVertex	SGSC2	value = 3	La lista de vértices del arreglo ahora posee dos con valor = 1 y 3 respectivamente, igualmente, la lista de aristas posee dos listas cuyos valores son un número máximo y por último un peso igual a 0.
ListGraph	addVertex	SGSC3	value = 5	La lista de vértices del arreglo ahora posee 5 elementos con valor = 1, 2, 3, 4 y 5 respectivamente, igualmente, la lista de aristas posee dos listas cuyos valores son números máximos y pesos iguales a 0.

Objetivo de la Prueba: Verificar si se añade un vértice al grafo con sus aristas vacías.				
Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	deleteVertex	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices.
ListGraph	deleteVertex	SGSC2	Un vértice existente en el grafo.	Se elimina el vértice dado de la lista de vértices y se eliminan las aristas desde y hacia a este. En este caso se tiene un grafo vacío como resultado.
ListGraph	deleteVertex	SGSC3	Un vértice existente en el grafo.	Se elimina el vértice dado de la lista de vértices y se eliminan las aristas desde y hacia a este. Al actualizar la matriz de adyacencia, esta tiene una fila y una columna menos.

Objetivo de la Prueba: Verificar si se añade una arista entre dos vértices con un peso dado

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	addEdge	SGSC1	Dos vértices existentes en el grafo. weight = 10;	No hay cambios en la estructura del grafo ni en sus vértices.
ListGraph	addEdge	SGSC2	Dos vértices existentes en el grafo. weight = 10;	No hay cambios en la estructura del grafo ni en sus vértices.
ListGraph	addEdge	SGSC3	Dos vértices existentes en el grafo. weight = 10;	Se asigna una arista con peso 10 entre el vértice 1 y el vértice 2. La matriz de adyacencia cambia este valor en la posición (i, j) teniendo en cuenta los IDs de los vértices.

Objetivo de la Prueba: Verificar si se asignan las distancias y predecesores correctos a cada vértice.

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	breadthFirstSearch	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.
ListGraph	breadthFirstSearch	SGSC2	Primer y único vértice de la lista	Distancia del único vértice = 0. Sigue sin predecesor.
ListGraph	breadthFirstSearch	SGSC3	Segundo vértice en la lista con valor = 2	Distancias = [1,0,1,1] Predecesores = [2, nulo,2,2]

Objetivo de la Prueba: Verificar si se añaden los tiempos adecuados a cada vértice según el recorrido por profundidad.

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	depthFirstSearch	SGSC1		No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.
ListGraph	depthFirstSearch	SGSC2		timestamps = [1,2]
ListGraph	depthFirstSearch	SGSC3		timestamps = [[2,7],[1,8],[3,6],[4,5]]

Objetivo de la Prueba: Verificar si se halla el camino mínimo desde un vértice hasta todos los demás.

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	dijkstra	SGSC1	Un vértice cualquiera	[null]
ListGraph	dijkstra	SGSC2	Primer y único vértice de la lista	[null]
ListGraph	dijkstra	SGSC3	Segundo vértice en la lista con valor = 2	[]

Objetivo de la Prueba: Verificar si se encuentran los caminos mínimos entre todos los vértices

Clase	Método	Escenario	Valores de Entrada	Resultado
-------	--------	-----------	--------------------	-----------

ListGraph	floydWarshall	SGSC1		No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.
ListGraph	floydWarshall	SGSC2		Se calculan los caminos de peso mínimo utilizando como base una copia de la matriz de adyacencia, asignando el resultado en minimumWeightPaths. En este caso el resultado es una matriz de 1x1 con un valor de 0.
ListGraph	floydWarshall	SGSC3		Se calculan los caminos de peso mínimo utilizando como base una copia de la matriz de adyacencia, asignando el resultado en minimumWeightPaths.

Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	prim	SGSC1		
ListGraph	prim	SGSC2		
ListGraph	prim	SGSC3		

Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
-------	--------	-----------	--------------------	-----------

ListGraph	kruskal	SGSC1		
ListGraph	kruskal	SGSC2		
ListGraph	kruskal	SGSC3		

Objetivo de la Prueba: Verificar si se retorna apropiadamente la lista con todas las aristas del grafo.

Clase	Método	Escenario	Valores de Entrada	Resultado
ListGraph	degreeOf	SGSC1	Un vértice existente en el grafo.	No hay cambios en la estructura del grafo ni en sus vértices. Tampoco se hace ningún procedimiento.
ListGraph	degreeOf	SGSC2	Un vértice existente en el grafo.	Se cuenta la cantidad de vértices adyacentes que tiene la entrada. En este caso se retorna 0.
ListGraph	degreeOf	SGSC3	Un vértice existente en el grafo.	Se cuenta la cantidad de vértices adyacentes que tiene la entrada. En este caso se retorna 3 para cualquiera de ellos.

5. BIBLIOGRAFÍA

Limelight Networks (2021), *The State of Online Gaming [El estado del juego online]*.

https://img03.en25.com/Web/LLNW/%7B46b8ee89-c7f0-46b7-a5b3-0976ef1a3253%7D_2021-MR_SOOG_8.5x11_V06.pdf

Marcus Bromander (2018), *Among Us* (Versión para computadora) [Videojuego]. Innersloth.

Mario Party (Nintendo 64) [Videojuego]. (1998). Nintendo.

Preguntados (Versión para móvil) [Videojuego]. (2013). Etermax.

ticedev(s.f.) *skribbl.io* (Versión web) [Videojuego]. ticedev.

WePC (2021), *Video Game Industry Statistics, Trends and Data In 2021* [Estadísticas de la industria del videojuego].
<https://www.wepc.com/news/video-game-statistics/>