Universidad Icesi – Facultad de Ingeniería
Algoritmos y estructuras de datos
Andrés Alberto Aristizabal Pinzón

Jean Carlos Ortiz
Michael Giraldo Ruíz
Caleb Villadiego Henao

Método de la ingeniería:

Enunciado de la problemática

¡A por las esferas del Dragón!

Bandai es una empresa japonesa dedicada a la creación de juguetes y figuras de acción, en muchos de los casos posee los derechos exclusivos para la promoción de mercadería basada de series animadas; es uno de los principales auspiciantes que han ayudado a poner en marcha diferentes series de anime, entre las más famosas se encuentran Dragon Ball, Digimon, Saint Seiya, Gundam, Ultraman, Kamen Rider, Super Sentai, Power Rangers, Naruto Shippuden y One Piece. Además comercializó los Tamagotchi y está presente en otros campos, como por ejemplo, la creación y distribución de videojuegos.

En esta ocasión BanDai desea realizar un nuevo videojuego de la serie animada Dragon Ball, para ello, ha contratado sus servicios como programadores para realizar la implementación de la versión Beta del juego. Lo primero que desea la empresa es que el juego tenga un registro para una gran cantidad de usuarios, pues, se piensa mostrar esta versión creando un torneo donde se premiará el jugador con más puntaje.

El juego debe tener un panel donde se puedan seleccionar un personaje de los más reconocidos por los seguidores de la serie, el juego en su versión Beta tendrá

solamente un nivel. La temática del juego será la siguiente: Al seleccionar al personaje, el personaje aparecerá en un mapa donde se encontrarán ocultas en unos montículos de tierra las esferas del dragón.El personaje debe encontrar el camino hacia ellas en el menor tiempo posible y gastando la mínima cantidad de energía yendo de un montículo a otro, para la implementación de este juego, Bandai, le pide que use las estructuras del grafo las cuales aprendió en su materia de estructuras de datos. Adicionalmente, el escenario del juego debe contar con un Radar del Dragón el cual le dará al personaje la opción de ver en que montículo se encuentra la esfera , ahí el jugador encontrará una alternativa para poder cumplir el objetivo usando los algoritmos de recorrido mínimo del grafo(Dijkstra, Kruskal, Prim, etc…). Los caminos, esferas, montículos y ponderados deben ser puestos de forma aleatoria.Al final del juego se deberá de guardar el puntaje obtenido por el jugador. BanDai espera que el juego sea completamente funcional y se pueda mostrar todo lo dicho anteriormente

Fase 1: Identificación de necesidades y síntomas.

-La solución del problema debe entregar una interfaz dinámica.
-La solución debe brindar eficiencia en el registro de datos.
-La solución debe brindar al usuario claridad de cómo usar el juego implementado.

Descripción del problema:

La empresa BanDai desea la implementación de la versión beta de un juego de la serie reconocida Dragon Ball.

Requerimientos Funcionales:

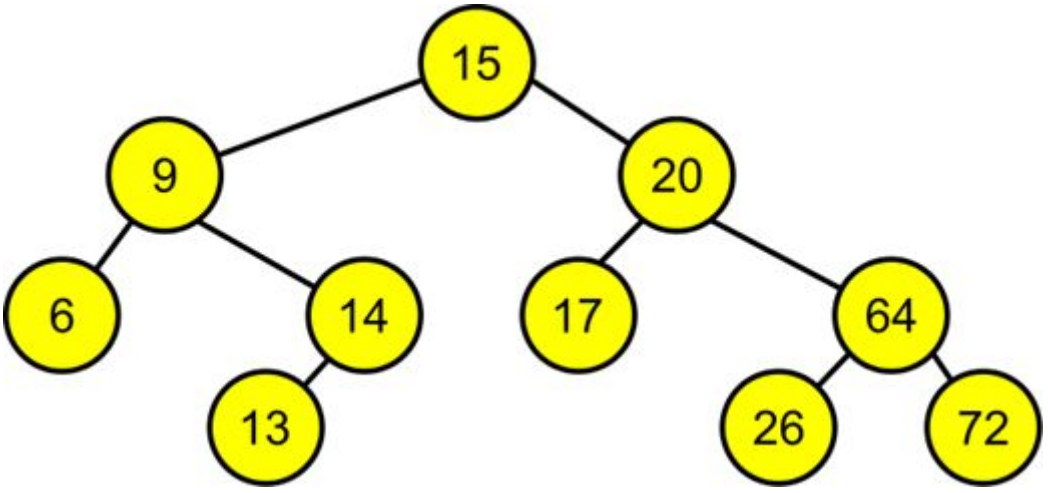
- R1 Registrar Usuarios. El programa debe permitir el registro de un nuevo usuario y almacenarlo en una cola de prioridad, donde se tendrá en cuenta el jugador con mayor puntaje.
- R2 Almacenar en tabla de puntajes. El programa permite mostrar en su interfaz una tabla con los nombres y puntajes de los usuarios en forma descendente, retorna la tabla con los usuarios.
- R3 Seleccionar Personaje. El programa debe permitir seleccionar un personaje el cual se mostrará en el inicio del juego.
- R4 Crear Mapa. El programa debe crear un mapa donde se encontrarán el personaje y los puntos generados.
- R5 Generar aleatoriamente las esferas. El programa debe distribuir aleatoriamente las 7 esferas del dragón en todo el mapa.
- R6 Encontrar Caminos. El programa permite al usuario encontrar caminos más cortos y con menos pesos, de tal manera que el personaje pueda llegar a la esfera.
- R7 Inicializar monedas. El Programa muestra monedas en todo el mapa las cuales se sumarán para que el usuario pueda ver los mejores caminos.
- R8 Mostrar Radar. En pantalla se mostrará un radar el cual llamará el metodo para mostrar el camino más corto hacia las esferas.
- R9 Generar Costo radar. El programa generará un precio para cada algoritmo que ayudará encontrar la mejor ruta o camino con menos costo.

Fase 2: Recopilación de información.

Para este proyecto se investigaron las siguientes estructuras

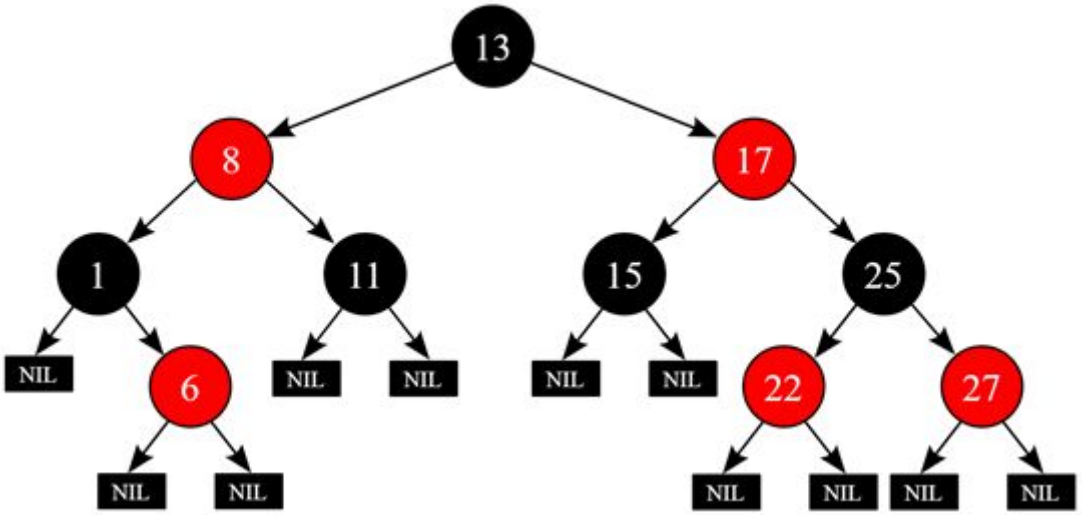
- Árbol Binario de Búsqueda (ABB):
-
- Árbol en el cual cada uno de sus nodos puede tener no más de 2 hijos. En el cual se cumple con la jerarquización de que todos los nodos a la izquierda de la raíz son menores que ella y todos los nodos a la derecha de la raíz son mayores que ella. Por lo tanto, el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores. Así, es una

buena opción para garantizar una búsqueda eficiente de los jugadores guardados.



2. Árbol Rojo y negro

Es un Árbol Binario de búsqueda que debe cumplir con las siguientes propiedades:
1 - Cada nodo es rojo o negro.
2 - La raíz es negra.
3 - Toda hoja (NIL) es negra.
4 - Si un nodo es rojo, sus dos hijos son negros. (no puede haber dos rojos consecutivos en un camino)
5 - Todocamino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros.
Así pues, de acuerdo a la propiedad 5, todo árbol rojinegro con n nodos internos tiene una altura menor o igual que 2 log(n+1). Por lo tanto, las operaciones de búsqueda pueden implementarse en tiempo O(log n) para árboles rojinegros con n nodos, así como sus funciones para insertar y eliminar.



Además para ahorrar espacio en memoria, en este árbol se crea un nodo auxiliar T.nil de color negro, al cual van apuntar todos los nodos que apuntan a un nodo NIL. Por lo tanto, se convierte en una de nuestras alternativas para facilitar la organización de jugadores y así mismo su búsqueda.

3. Arbol AVL:

Es un tipo especial de árbol binario.Es un árbol de búsqueda binaria autobalanceable, su objetivo es intentar mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente, esto con la intención de tener una complejidad O(log n) en las operaciones de búsqueda, inserción y eliminación.

Su propiedad consiste en que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa, para esto cada nodo contiene el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo, por definición, para un árbol AVL, este valor debe ser -1,0 ó 1.

Si el factor de equilibrio de un nodo es:

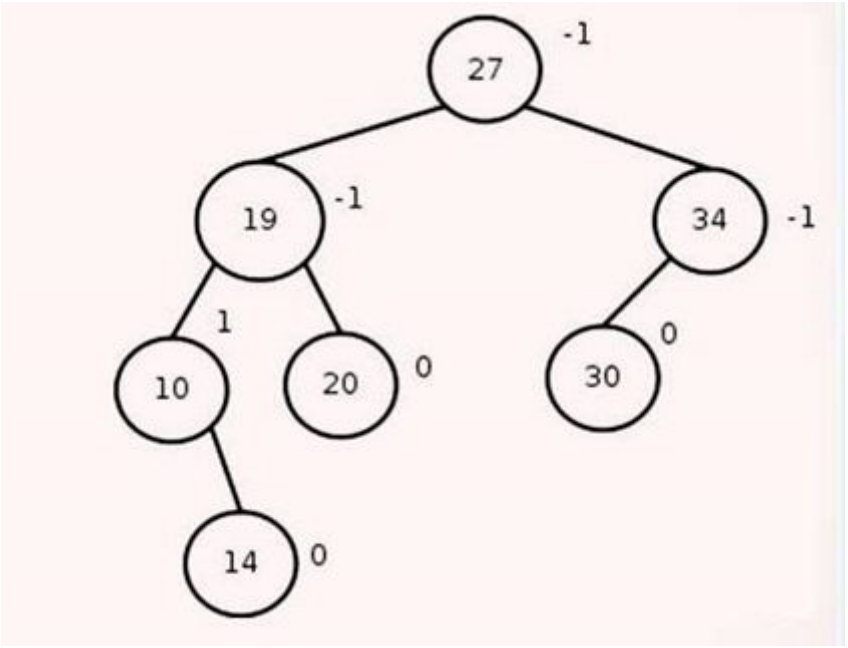
0 : el nodo está equilibrado y sus subárboles tienen exactamente la misma altura.

1 : el nodo está equilibrado y su subárbol derecho es un nivel más alto.

-1: el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

2 o mayor: es necesario equilibrar.

Para lograr el reequilibrado en un árbol AVL es necesario utilizar las operaciones de rotaciones, existen dos casos: rotación simple o rotación doble.



Por otro lado, nos encontramos con unas estructuras bastante peculiares, ya que nos ofrece una forma exclusiva de almacenar datos o un modelo a seguir para poder llevar un orden a la hora de extraer o insertar un dato, el cual son las siguientes:

4. Cola.

La cola es una estructura de dato, que permite el almacenamiento y recuperación de datos; caracterizada por usar el método FIFO el cual consiste en sacar el elemento que primero ha ingresado a la estructura, en otras palabras, si tenemos una serie de datos que se han ingresado de la siguiente forma:

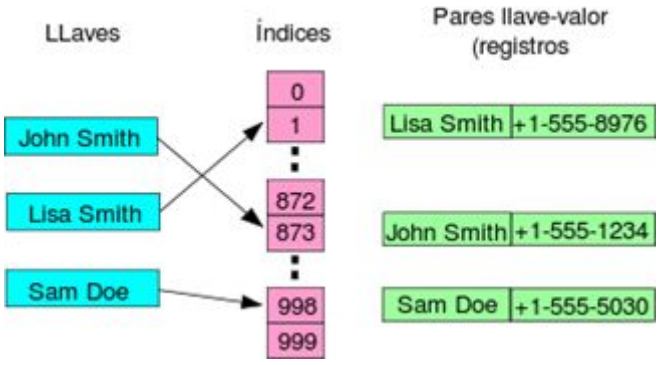
Q = { e1, e2, e3, e4…, en} donde e es igual al tipo de dato que estamos almacenando. Por ende, solo nos permitirá obtener el elemento e1 de la cola, ya que este ha sido el primer elemento insertado o agregado.

5. Pila:
La pila es una estructura de dato muy similar a la Cola, ya que nos permite almacenar y recuperar datos, pero en este caso la Pila nos ofrece el sistema LIFO, el cual consiste en poder recuperar o extraer el dato que se ha ingresado de último, en otras palabras, si tenemos una serie de datos que se han ingresado de la siguiente manera:

S = { e1, e2, e3, …., en} donde e es igual al tipo de dato que estamos almacenando.
Como se mencionó anteriormente, solo nos permite tomar o extraer el último elemento que se ha ingresado a la pila, entonces solo nos dejará obtener el elemento en de la pila S.

Una estructura muy eficiente para la búsqueda de datos es conocida como la tabla hash y una breve descripción

6. Tabla Hash: Es una estructura de datos especializada para almacenar grandes cantidades de datos, consiste en un nodo que contiene un índice y una llave, sus operaciones son bastante eficientes en especial su búsqueda, ya que con el índice único para cada elemento se puede acceder fácilmente a él.



7. Grafo

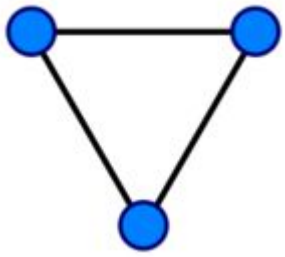
Por último cabe añadir una estructura de datos llamada grafo, el cual consiste en una serie de vértices o nodos que están conectados entre sí mediante aristas para mostrar una relación entre elementos de un conjunto.

Existen también diferentes tipos de grafos los cuales son:

Grafo Simple.
Multigrafo.
Grafo dirigido.

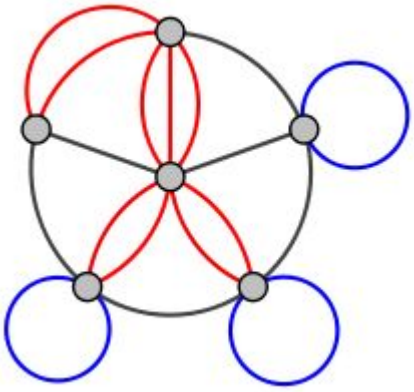
Por ahora solamente nos enfocaremos en lo que es un grafo simple y un multigrafo dirigido.

Grafo simple:



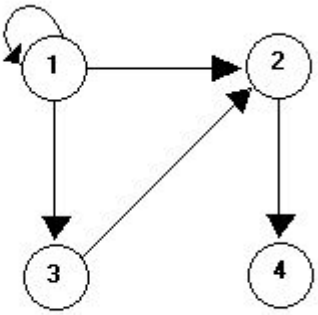
Dícese del grafo que no tiene lazos ni aristas múltiples entre sus vértices. En otras palabras un grafo simple es un grafo en los que existe a lo más una arista que une dos vértices distintos.

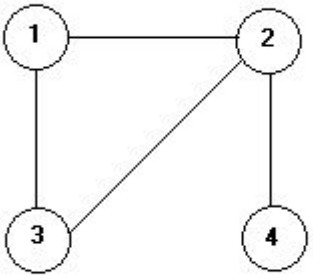
Multigrafo:
Es un grafo que está facultado para tener aristas múltiples, esto quiere decir, varias aristas que relacionan el mismo vértice o nodo. De esta forma dos nodos pueden estar conectados por más de una arista.



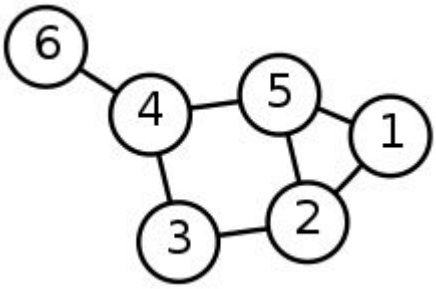
Grafo dirigido.Es aquel grafo donde las aristas tienen una dirección y sentido, es decir que la arista es una flecha que apunta hacia un nodo. Donde el vértice A conocido como origen, es la cola y el vértice B es el destino.

Para el grafo dirigido también también puede existir un multigrafo, el cual es conocido como multi dirigido o multigrafo dirigido.





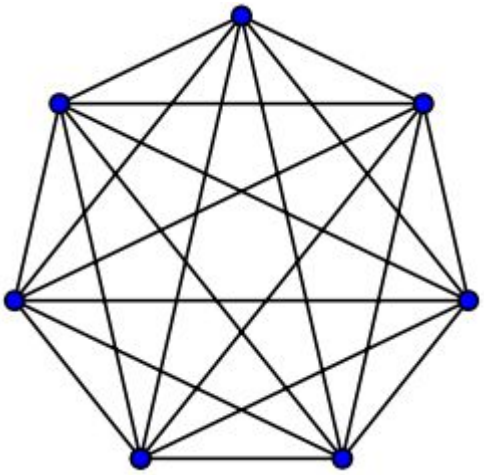
Adyacencia.
Un vértice adyacente de un vértice v en un grafo es un vértice que está conectado a v mediante una arista. Gráficamente es:



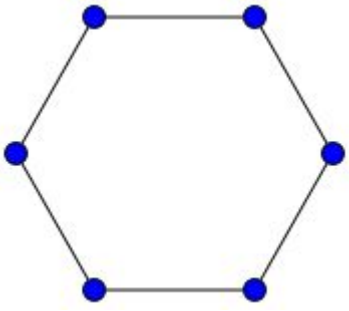
En la anterior imagen podemos decir que, el vértice 5 es adyacente a los vértices 1, 2 y 4, pero no es adyacente a los vértice 3 y 6.

Grado de un vértice.
Es el número de incidentes al vértice, es decir el número de aristas que tiene el vértice. Y se denota con el símbolo δ(v).
Tipos de grafos simples.
Aquí nos encontramos con los siguientes grafos:

Grafos completos.



Ciclos.



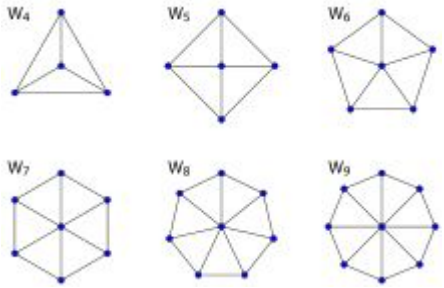
Ruedas.

Grafos completos: es un grafo simple donde cada par de vértices está conectado por una arista.
Grafo ciclo.

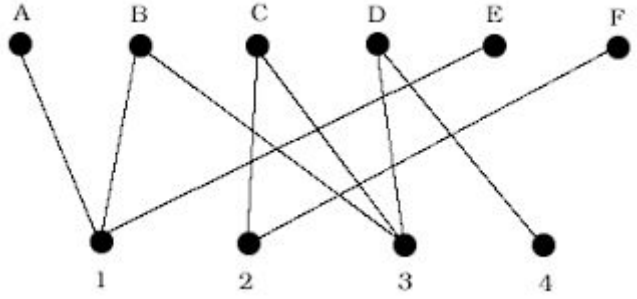
En Teoría de grafos, un Grafo ciclo o simplemente ciclo es un grafo que se asemeja a un polígono de n lados. Consiste en un camino cerrado en el que no se repite ningún vértice a excepción del primero que aparece dos veces como principio y fin del camino. Un Grafo ciclo de n vértices se denota Cn. El número de vértices en un grafo Cn es igual al número de aristas, y cada vértice tiene grado par, por lo tanto cada vértice tiene dos aristas incidentes.

Grafo Rueda

un grafo rueda (Wn), o simplemente rueda, es un grafo con n vértices que se forma conectando un único vértice a todos los vértices de un ciclo-(n-1).



Grafo Bipartito.



En resumen, hasta ahora llevamos recolectando información acerca de las diferentes alternativas que pueden ser óptimas para la resolución de nuestro problema, primordialmente se indagó sobre las siguientes 8 estructuras de datos

- Árbol ABB
    - Árbol Rojo y Negro
    - Árbol AVL
    - Cola
    - Pila
    - Tabla Hash
    - Grafos

- HashMap

No obstante, es trascendental llevar a cabo ideas centradas solamente a las condiciones de nuestro problema.

U U V = N
U n V = 0

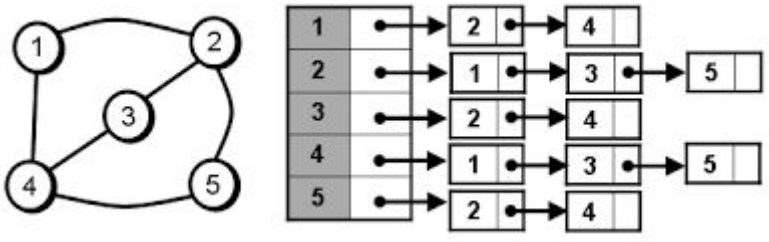
En otras palabras, si tenemos un grafo G en el cual podemos obtener dos conjuntos de vértices ya sea V1 Y V2 donde los vértices de cada conjunto no se relacionan o tiene aristas entre sí, pero si se relacionan con el otro conjunto, entonces es un grafo bipartito.

La anterior imagen sería un grafo bipartito, ya que si tenemos nuestro primer conjunto V1 con los vértices A,B,C,D,E,F y el conjunto V2 con los vértices 1,2,3,4 podemos ver que en el conjunto V1 ningún vértice se relaciona, igualmente para el V2, pero si se relacionan los conjuntos, es decir, el vértice A del conjunto V1 se relaciona o tiene una arista con el vértice 1 del conjunto V2, por tanto es un grafo bipartito.

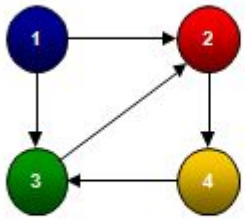
Por un lado, los grafos se pueden representar de diferentes maneras, entre ellas están:

- Lista de adyacencia.
- Matriz de adyacencia.
- Matriz de incidencia.

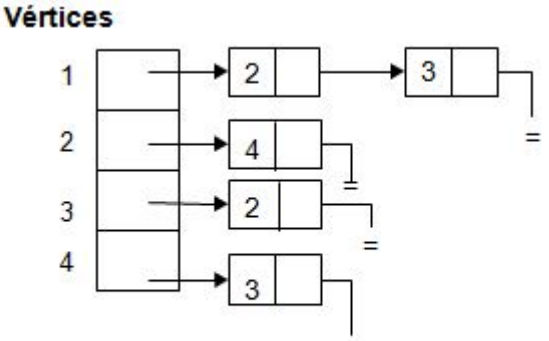
Lista de adyacencia:
Es una forma de representar un grafo sin aristas múltiples. Especifican los vértices que son adyacentes a cada uno de los vértices del grafo.



En el caso de un grafo dirigido, los vértices adyacentes son los cuales las aristas tienen la dirección hacía otro vértice e incluso hacia él mismo.



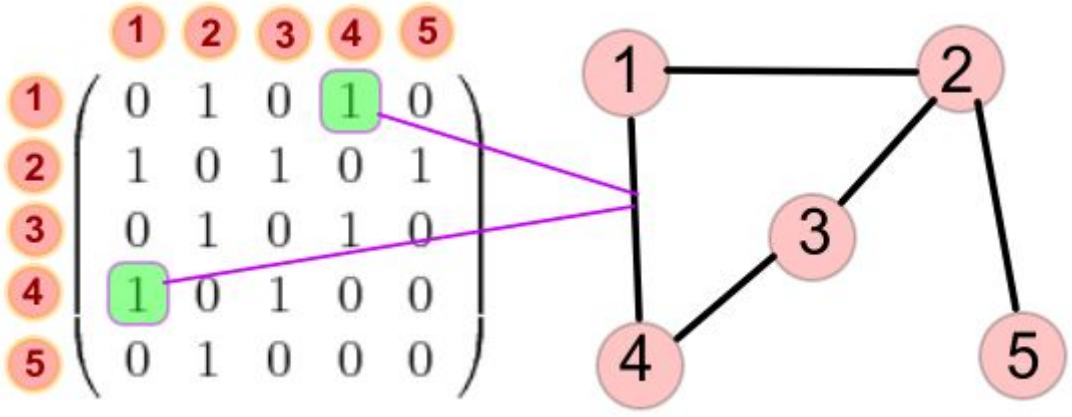
Grafos



Lista de Adyancencia

Matriz de adyacencia

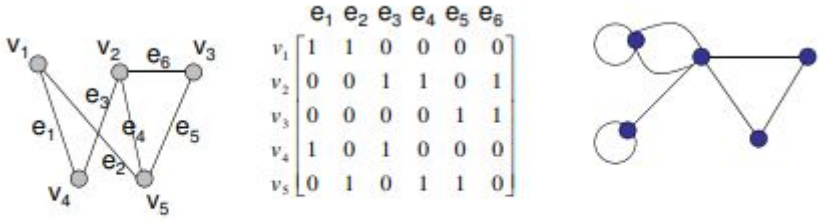
Es una matriz de tamaño n x n, donde las columnas y filas están dadas por todos los vértices del grafo y representa los vértices adyacentes de todos los vértices del grafo, si existe vértices adyacentes en un vértice se coloca 1, en caso contrario se rellena de 0.



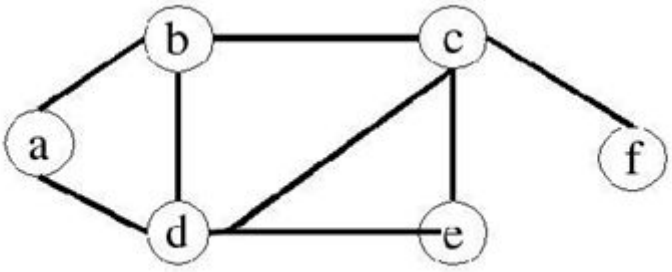
Matriz de incidencia.

Prácticamente la matriz de incidencia consta de una matriz n x m, donde las n-filas componen los vértices del grafo, y m-columnas componen el peso o la etiqueta de la arista. En el cual se observa cada arista a qué vértice está incidiendo o está

conectando y se la marca 1 si incide a algún vértice, en caso contrario se coloca 0. Esto quiere decir que se forma una matriz binaria.



Un camino del grafo una secuencia de aristas que comienzan en un vértice del grafo y
recorre ciertas aristas del grafo siempre conectando pares de vértices adyacentes.



Un camino simple de "a" a "f" = {a,b}, {b,c}, {c,f}

Un camino no simple de "a" a "f" = {a,b}, {b,c}, {c,e}, {e,d}, {d,c}, {c,f}

Un ciclo simple: {a,b}, {b,d}, {d,a}

Un ciclo no simple: {a,b}, {b,d}, {d,c}, {c,e}, {e,d},{d,a}

Figura 3: Caminos y ciclos en un grafo.

El camino es un circuito si comienza y termina en el mismo vértice y no tiene aristas repetidas, esto es, si u = v, y tiene longitud mayor a cero.

Un grafo no dirigido es conexo cuando todos sus vértices tiene una relación o una forma de comunicarse entre sí, ya sea simplemente o relacionándose a través de uno o más vértices.

Se le llama vértice de corte a aquel que al ser eliminado junto a todas las aristas incidentes en él produce un subgrafo con más componentes conexas.
Una arista de corte es aquella que cuya eliminación produce un grafo con más componentes conexas que el grafo original se llama arista de corte o puente.

Llegamos a un punto en el cuál nos preguntamos qué funciones o beneficios puede generar un grafo además de relacionar un conjunto de elementos, pues bien, nos ofrece unos diferentes métodos de búsqueda, en estos nos podemos encontrar con los siguientes:

- Búsqueda ciega o sin información
- Búsqueda heurística
- BFS
- DFS

Búsqueda ciega o sin información:
Consiste en una búsqueda en el cual no usa información sobre el problema, normalmente se realiza un recorrido exhaustivo.
Consiste en coger un vértice y el proceso de búsqueda se concibe con la construcción de un recorrido del grafo.
Búsqueda heurística:
Utiliza información acerca del problema, como los costos, pesos, etc. Esto quiere decir que posee información muy valiosa para orientar a la búsqueda.

De acuerdo a lo anterior es necesario buscar una estrategia de búsqueda, ya sea por amplitud o profundidad.

BFS:
El BFS o búsqueda por amplitud es de uno de los más simples de búsqueda de grafos.
Básicamente, se empieza con un vértice origen y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.Cabe aclarar que sirve para grafos dirigidos y no dirigidos.

Consta en principio con los nodos o vértices de color blanco para saber que no se ha visitado, una vez visitado lo cambia a color gris, y empieza a visitar sus vértices adyacentes, una vez visita todos sus vértices adyacentes cambia de color gris a negro.

DFS:
El DFS es un algoritmo de búsqueda por profundidad que permite recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme.
Expande un vértice hasta que no se pueda más y regresa. Este proceso continúa hasta que se descubran todos los vértices alcanzables desde el vértice original.
De igual forma, el DFS sigue el mismo proceso del BFS, colorea los vértices para indicar su estado.

Un grafo ponderado Son aquellos en los que se asigna un número a cada una de las aristas.
Uno de los problema de los grafos ponderados es el de determinar un camino de longitud mínima entre dos vértices de una red. Más concretamente, se define la longitud de un camino en un grafo ponderado como la suma de los pesos de las aristas de ese camino.

Existen algunos algoritmos que nos ayudan a determinar el camino mínimo entre dos vértices.

- Dijkstra
- Floyd - Warshall

Dijkstra:

Consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene..

Floyd - Warshall.
Es un algoritmo de análisis sobre grafos que permite encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución, constituyendo un ejemplo de programación

dinámica. Cabe aclarar que este algoritmo trabaja con la matriz D inicializada con las distancias directas entre todo par de nodos, es decir la matriz de pesos.
Compara todos los posibles caminos a través del grafo entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo V3 comparaciones, lo hace mejorando una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima.

Un grafo también nos ofrece la oportunidad de encontrar árboles recubridores mínimos en un grafo conexo no dirigido y cuyas aristas están etiquetadas.
En realidad, ¿Qué es un árbol recubridor mínimo?
Un árbol recubridor mínimo de un grafo, es un subgrafo que tiene que ser un árbol y contender todos los vértices del grafo inicial. Por lo que existen algunos algoritmos que nos ayudan a encontrarlos:

- Algoritmo de Prim
- Algoritmo Kruskal

Algoritmo de Prim
El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

El algoritmo podría ser informalmente descrito siguiendo 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 pertenecen al árbol)

UNIVERSIDAD ICESI

Algoritmo de Kruskal

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo.

El algoritmo de Kruskal es un ejemplo de algoritmo voraz que funciona de la siguiente manera:

1. se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado
2. se crea un conjunto C que contenga a todas las aristas del grafo
3. mientras C es no vacío:
    - eliminar una arista de peso mínimo de C
    - si esa arista conecta dos árboles diferentes se añade al bosque, combinando los dos árboles en un solo árbol
    - en caso contrario, se desecha la arista.

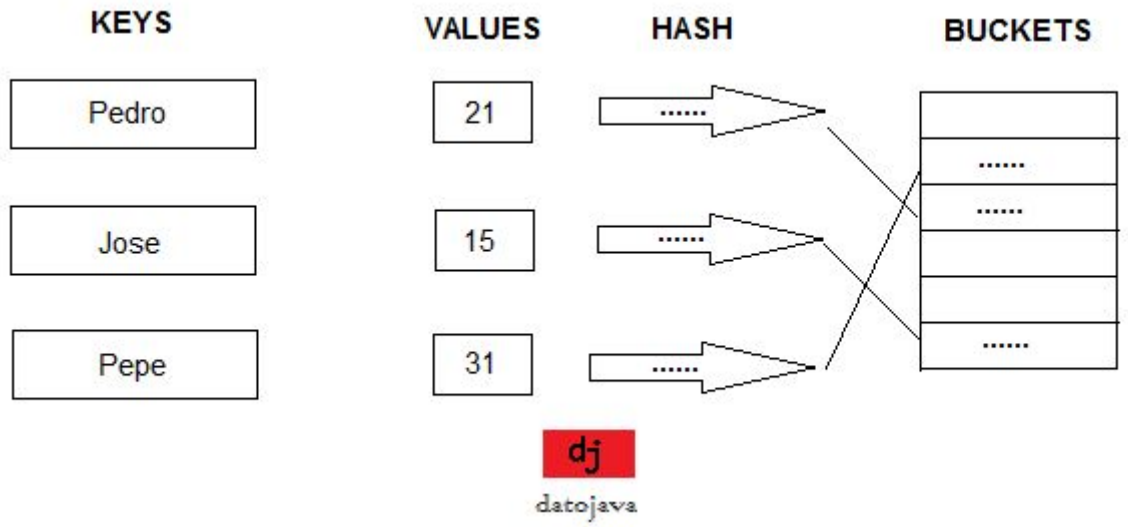
Al acabar el algoritmo, el bosque tiene un solo componente, el cual forma un árbol de expansión mínimo del grafo.

En un árbol de expansión mínimo se cumple:

la cantidad de aristas del árbol es la cantidad de nodos menos uno (1).

8. HashMap

HashMap es otra estructura de datos que almacena datos asociando una llave a un valor, sirve para muchas cosas y tiene ciertas características que la definen, por ejemplo, no permite key duplicados, cada key tiene que estar asociado a un valor como máximo, si agregas un key que ya existe sobrescribe el valor del key anterior, sólo permite Object types lo que quiere decir que no puedes poner un valor primitivo.



9. Sprites

Los sprites (del inglés, "duendecillos") fueron popularizados por Jay Miner. Se trata de un tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado (MSX, Atari 400/800, Commodore 64 y Commodore Amiga fueron unos de los pocos ordenadores que soportan sprites reales) sin cálculos adicionales de la CPU. A menudo son pequeños y parcialmente transparentes, dejándoles así asumir otras formas a la del rectángulo, por ejemplo puntero de ratón en ordenadores de la serie Amiga. Esto es posible ya que se podía hacer una operación lógica OR o AND entre la imagen original y otra, y dejar el resultado en otra posición de memoria. De esta manera era sencillo cambiar colores o eliminar el fondo mediante una "máscara". Típicamente, los sprites son usados en videojuegos para crear los gráficos de Por ejemplo, los personajes de un *Pac-Man* podrían ser sprites. Generalmente son utilizados para producir una animación, como un personaje corriendo, alguna expresión facial o un movimiento corporal.

Con el paso del tiempo el uso de este término se extendió a cualquier pequeño mapa de bits que se dibuje en la pantalla, incluso si tiene que moverlo todo el procesador central y no cuenta con hardware especializado. Esto ocasiona que a diferencia de ordenadores que soportan sprites reales no se puedan mezclar

resoluciones (por ejemplo mostrar un sprite de 256 colores y alta resolución en una pantalla de 16 colores y baja resolución), la cpu tenga que calcular las transparencias y no vayan sincronizados con el refresco vertical de pantalla (por lo cual el hardware gráfico que no dispone de sprites reales necesita aplicar técnicas de doble buffer).

Fase 3: Búsqueda de soluciones creativas.

En resumen, hasta ahora llevamos recolectando información acerca de las diferentes alternativas que pueden ser óptimas para la resolución de nuestro problema, primordialmente se indagó sobre las siguientes 8 estructuras de datos

- Árbol ABB
- Árbol Rojo y Negro
- Árbol AVL
- Cola
- Pila
- Tabla Hash
- Grafos
- HashMap

No obstante, es trascendental llevar a cabo ideas centradas solamente a las condiciones de nuestro problema.

En primer lugar, el problema está basado en la interfaz gráfica y en la capacidad de poder representar caminos cortos ya que el avatar necesita encontrarlos para poder

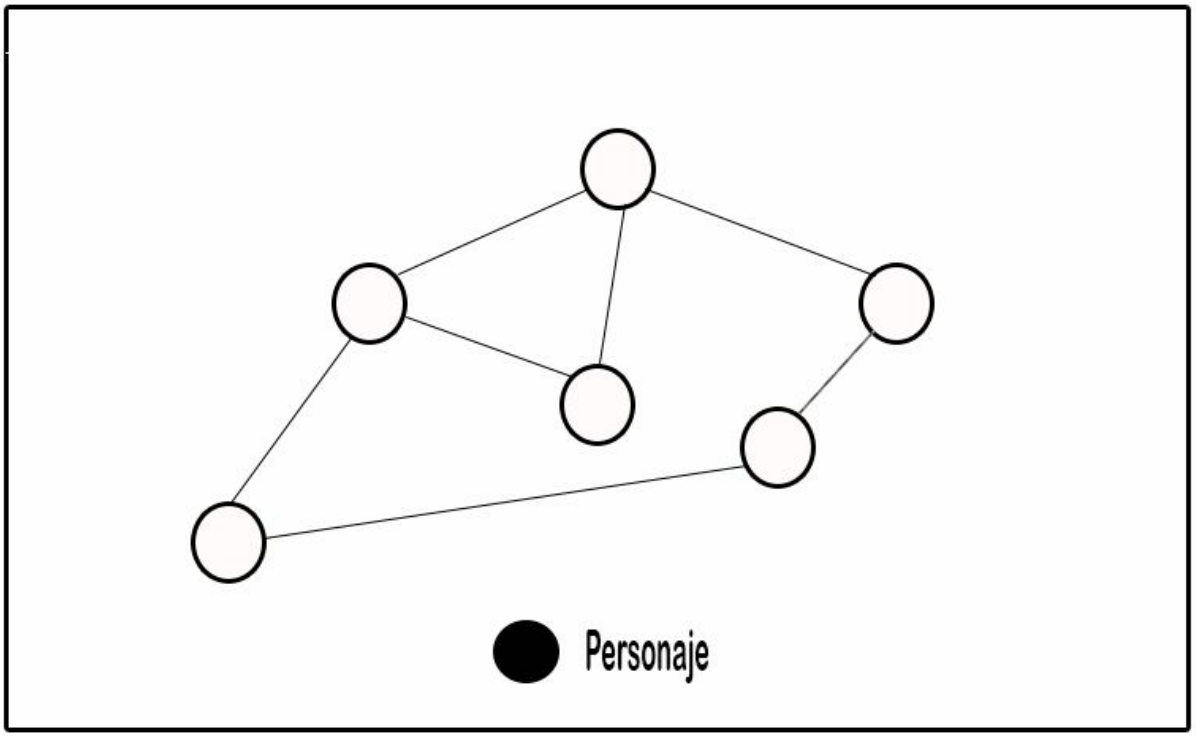
tener la mejor puntuación, Es necesario que el programa implementa un mundo donde el personaje pueda moverse de la mejor forma, además debe poder almacenar una gran cantidad de jugadores en las que solo nos interesa saber el ganador del torneo. Para esto, se ha decidido tener las siguientes alternativas:

Alternativa 1:

Almacenamiento de datos: Para esta alternativa se decidió usar la estructura de datos de árboles rojiNegros, pues estos nos brindan una gran capacidad de memoria y su capacidad de búsqueda gracias a su método de nivelación , nos permite encontrar datos de una manera ordenada y efectiva.

Manejo del escenario: Para el manejo del escenario decidimos usar una imagen fija que se adaptara a la pantalla del dispositivo, dentro de la pantalla se pondrán los elementos los cuales el personaje recorrerá.

Puesta en escena: para la implementación de los montículos y esferas se decide usar un ArrayList donde se agregaran los objetos y se les asignará una posición en el escenario



Movimiento: Para el movimiento se trabajará a través de hilos, el personaje podrá moverse por todo el escenario activando un hilo cada vez que se dé click en la pantalla, de esta forma el personaje se moverá a través del escenario.

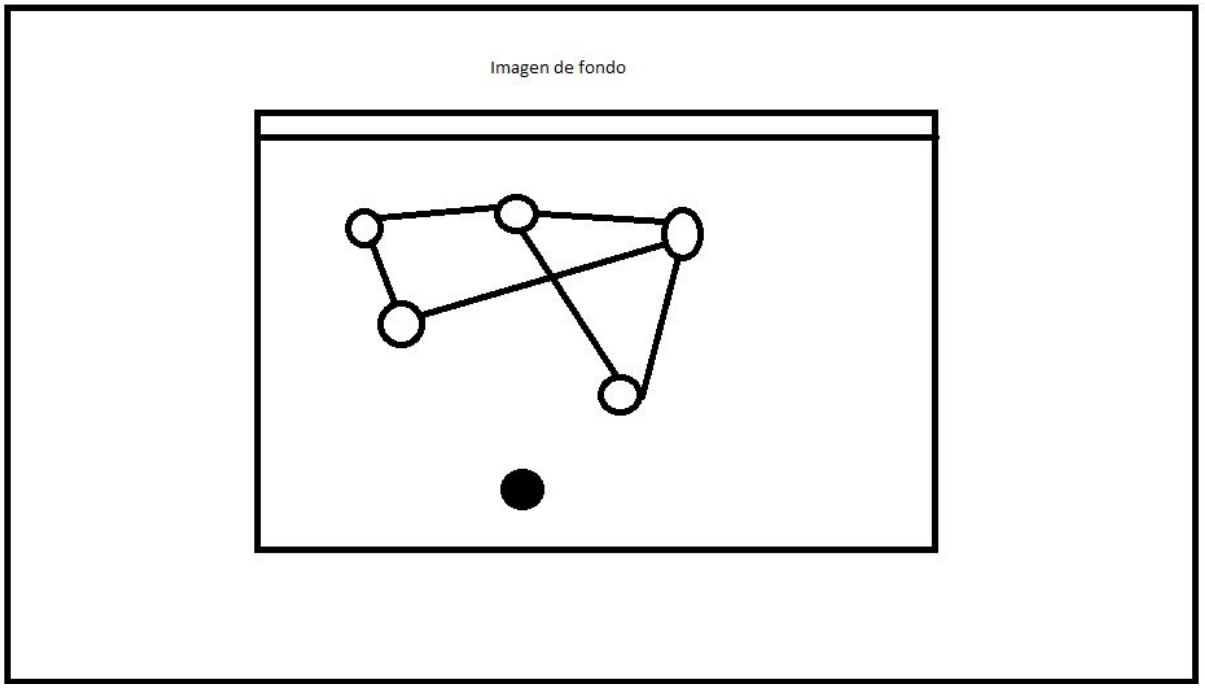
Implementación de animaciones: Las animaciones del juego serán manejadas a través de imágenes .gif, las cuales pueden crear cierto dinamismo en la interfaz, esto nos ayudará a mejorar la interacción con el usuario.

Alternativa 2:

Almacenamiento de datos: Para esta alternativa se decide usar una cola de prioridad. Una **cola de prioridad** es un tipo de dato abstracto similar a una cola en la que los elementos tienen adicionalmente, una *prioridad* asignada.[12] En este caso, la prioridad será el jugador con mayor puntaje.

Manejo de escenario: para el manejo de escenario se pensó hacer un mapa grande que se adecue al tamaño de la pantalla. Para esto es necesario tener una imagen lo suficientemente grande para irla adecuando. A continuación se mostrará como funcionaria.

Puesta en escena: para la implementación de los montículos y esferas se decide usar la estructura Grafo, el cual nos permite tener distintos nodos (los cuales serán las esferas del dragón) y unos recorridos, que serán la cantidad de energía que gastará gokú.



Para el movimiento : El personaje se desplaza mediante los key Events, al desplazarse el personaje hacia la derecha, el mundo (imagen de gran tamaño) se moverá hacia el lado contrario, de esta forma, da la perspectiva al usuario que el escenario se mueve, aunque no sea así.

Implementación de animaciones: Las animaciones del juego serán manejadas a través de sprites, estarán asociados a un KeyEvent que darán un cambio en el personaje, la animación no será un hilo, será solamente un cambio de posición.

Fase 4: Transición de ideas de diseños preliminares:

Revisión de las alternativas

Alternativa 1:

-El espacio del escenario impide la jugabilidad y la limita a un espacio muy pequeño.
-Los gif's no son editables.
-La clase ArrayList a la larga no es eficiente a la hora de buscar el ganador.

Alternativa 2:
-La cola de prioridad tiene un tope que puede ser perjudicial a la hora de agregar una gran cantidad de usuarios.

Fase 5: Evaluación de la mejor solución:

Criterios:

- Criterio A. Eficiencia. Se prefiere una solución con mejor eficiencia que las otras consideradas. La
  eficiencia puede ser:
  [4] Constante
  [3] Mayor a constante
  [2] Logarítmica

[1] Lineal

- Criterio B. Compatibilidad. Se prefiere una solución que sea compatible con todos los sistemas a evaluar.
  [3] Compatible con todos los sistemas
  [2] Compatible con algunos sistemas
  [1] No compatible.

- Criterio C. Eficiencia en la búsqueda. Se prefiere una solución que sea eficiente en el momento de buscar y seleccionar datos.La eficiencia puede ser.
  [4] Constante
  [3] Mayor a constante
  [2] Logarítmica
  [1] Lineal

- Criterio D. Manejo de caminos: Se prefiere una solución que sea más gráfica y visible que las otras soluciones. La solución puede ser:
  [3] Clara gráficamente
  [2]Poco clara gráficamente
  [1]Irrelevante.

	Criterio A	Criterio B	Criterio C	Criterio D	Total
Alternativa 1	3	3	2	2	10
Alternativa 2	4	3	4	3	14

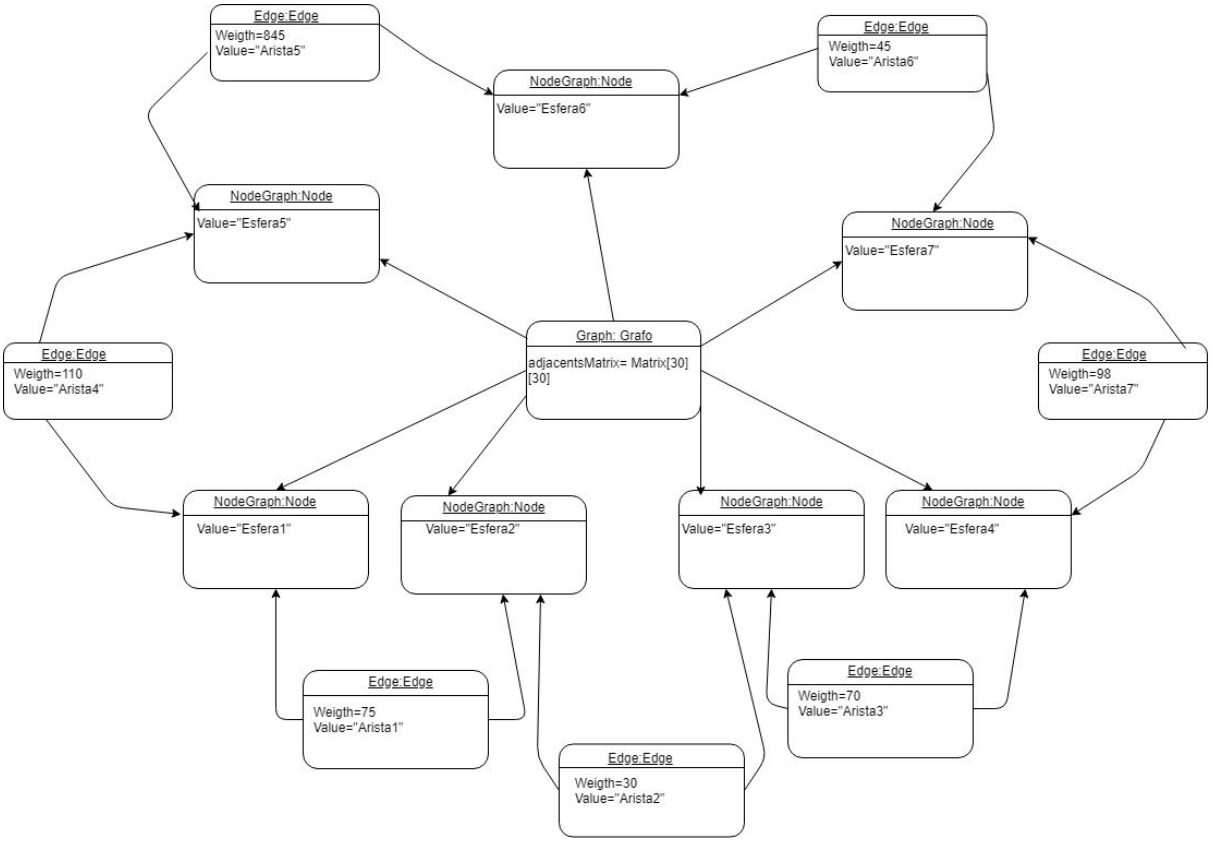
Selección

De acuerdo con la evaluación anterior se debe seleccionar la Alternativa 2, ya que obtuvo la mayor puntuación de acuerdo con los criterios definidos. Se debe tener en cuenta que hay que hacer un manejo adecuado del criterio en el cual la alternativa fue peor evaluada que la otra alternativa.

UNIVERSIDAD ICESI

Fase 6 Preparación de informe y reparaciones.

Diagrama De Objetos(Imagen full en carpeta Docs del Proyecto)



UNIVERSIDAD ICESI

Diagrama De clases del modelo

UNIVERSIDAD ICESI

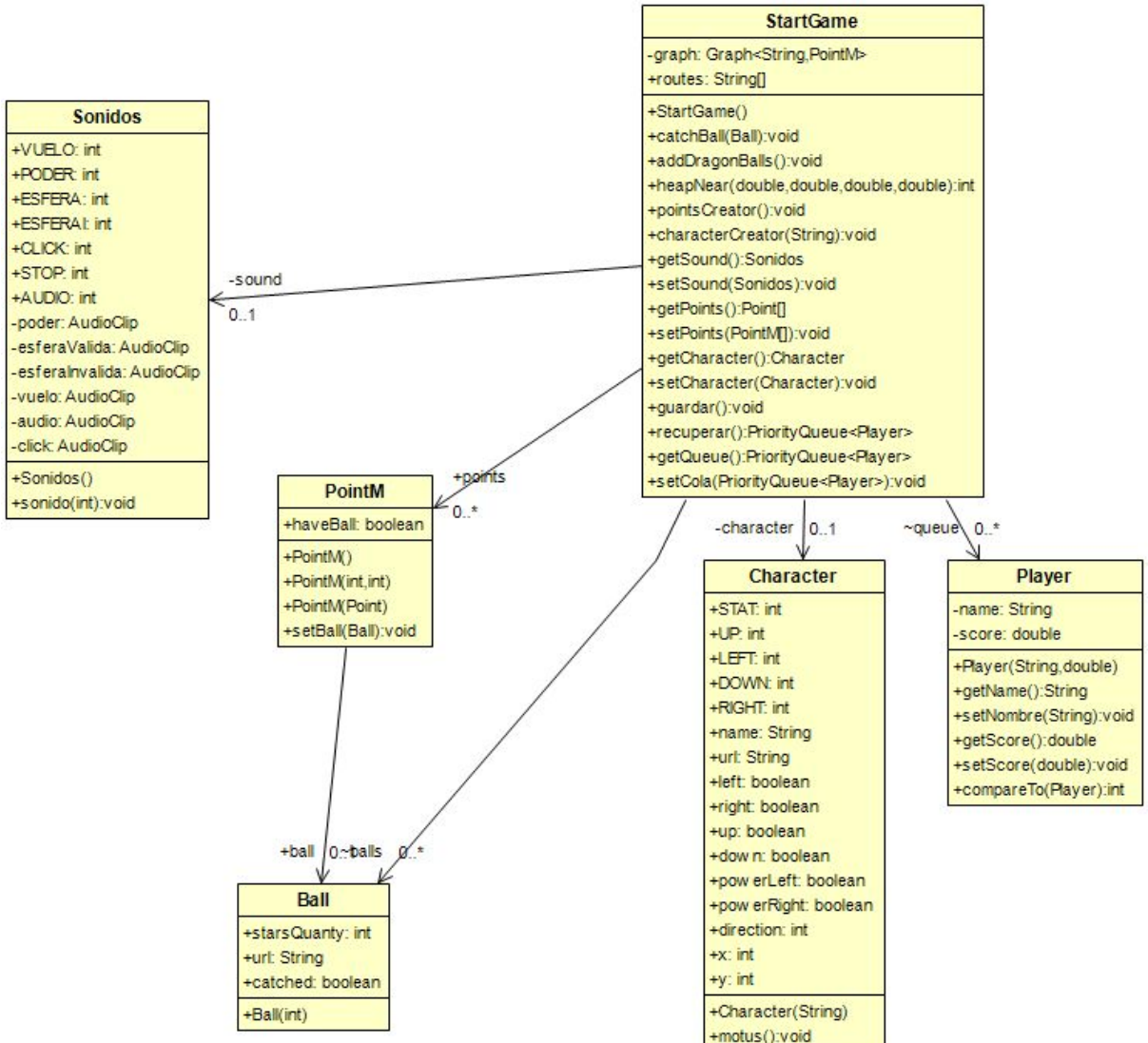
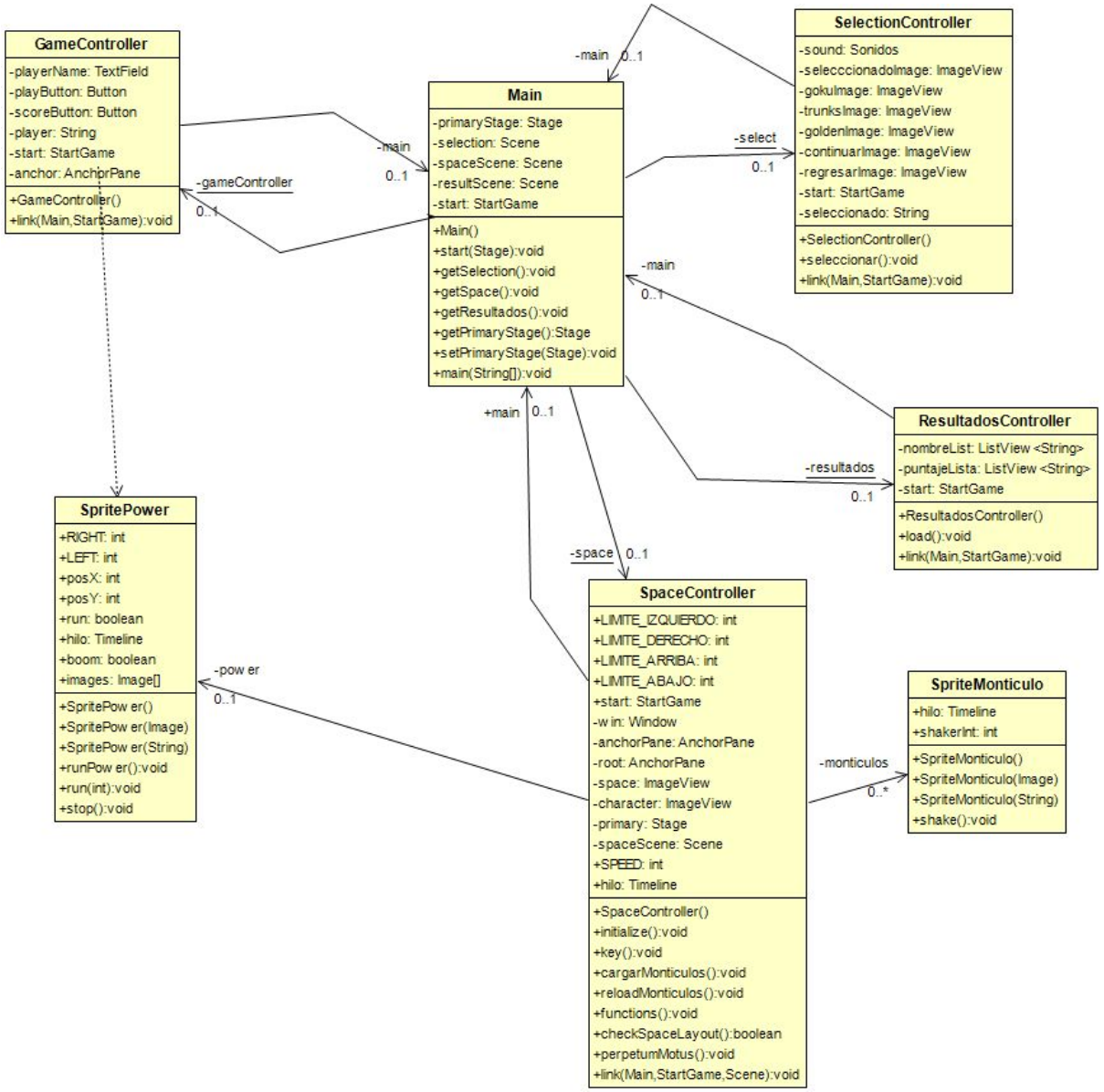


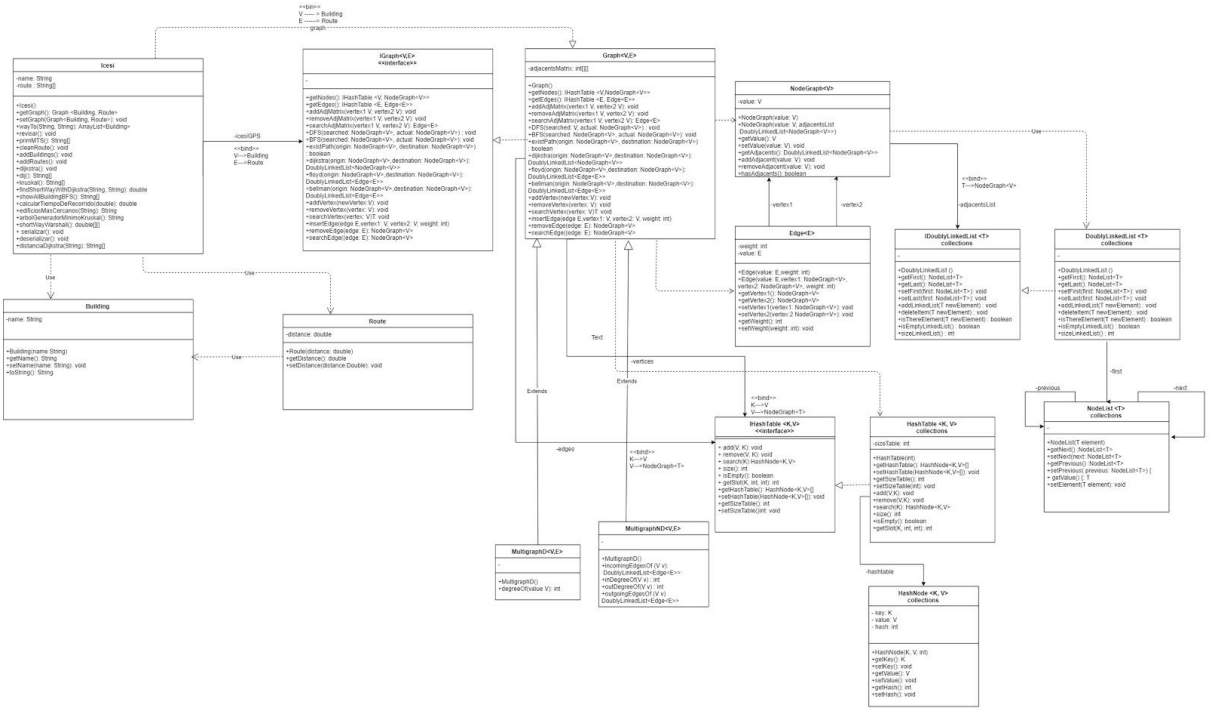
Diagrama de clases de la interfaz

UNIVERSIDAD ICESI



UNIVERSIDAD ICESI

Diagrama de la Colección(Sin modelo)(En la carpeta Docs, se ve claramente)

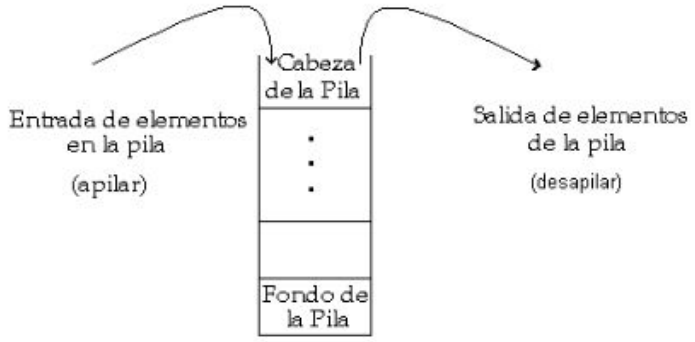


TAD’s

Tabla de representaciones

Modificadora	Push, Pop,Delete
Constructora	LlenarPila
Analizadora	IsEmpty,toList

TAD Pila



{inv: el primer elemento a agregar es el último elemento que sale de la pila}

Operaciones:

• push	Pila X Nodo	→	Pila
• pop	Pila	→	Nodo
• peek	Pila	→	Nodo
• isEmpty	Pila	→	Booleano
• delate	Pila	→	Null
• toList	Pila	→	NodoxNodo

CrearPila()

Se encarga de estructurar la pila para su funcionamiento

{ pre: TRUE }

{ post: }

UNIVERSIDAD ICESI

EsVacia(Pila p)
Informa si la pila está o no vacia
{ pre: Pila p}
{ post: <True: p = null || False: p != null > }

Apilar(Pila p, Elemento e)
Se añade un elemento a la pila. Se añade el final de esta.
{ pre: cola != null || p = < e1, e2, e3, ... en> }
{ post: <e1, e2, e3, ...en, e> }

Desapilar(Pila p)
Saca de la pila el elemento insertado más rencientemente
{ pre: cola != null || p = < e1, e2, e3, ... en> }
{ post: <e1, e2, e3, ...en-1> }

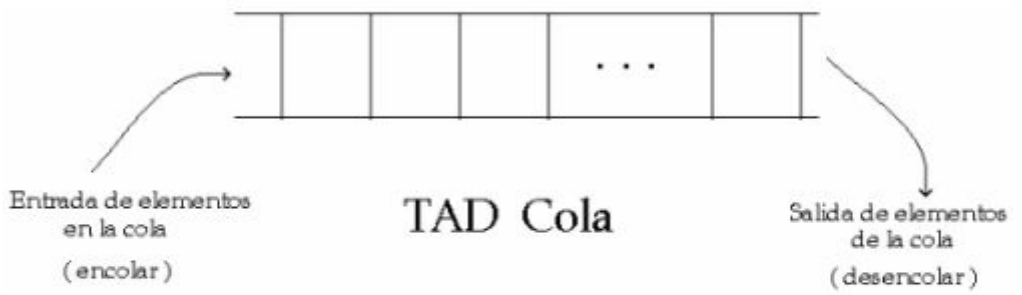
Tope(Pila p)
Recupera el valor del elemento que está al tope.
{ pre: cola != null || p = < e1, e2, e3, ... en> }
{ post: <en> }

DestruirPila(Pila p)
Destruye la pila p liberando la memoria que está ocupada.
{ pre: Pila p}
{ post: <-> }

Modificadora	Offer,poll
Constructora	crearCola,
Analizadora	peek,size,isEmpty

UNIVERSIDAD ICESI

TAD Cola



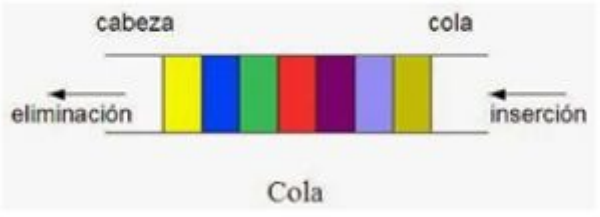
{invariante: el primer nodo a ingresar en la cola es el primer nodo que sale de la cola }

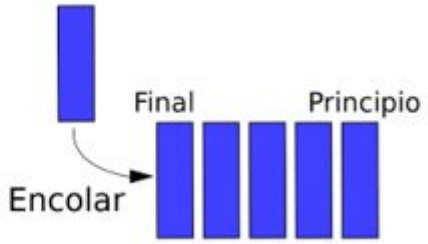
Operaciones:

• offer	Cola X Nodo	→	Cola
• poll	Cola	→	Nodo
• peek	Cola	→	Nodo
• size	Cola	→	Integer
• isEmpty	Cola	→	Booleano

esVacia()
Verifica si la cola está vacia
{ pre: TRUE }
{ post : TRUE si está vacia. FALSE si no está vacia }

CrearCola()
Se encarga de estructurar la cola para su funcionamiento
{ pre: TRUE }
{ post: }



Encolar(elemento)
Se añade un elemento a la cola. Se añade el final de esta.
{ pre: cola != null }
{ post:  }

Frente()
Devuelve el elemento frontal de la cola, es decir, el primer elemento que entro
{ pre : cola != null && primero != null }
{post : retorno del primer elemento de la cola }

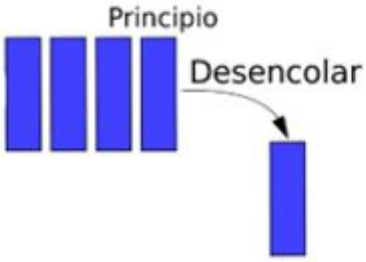
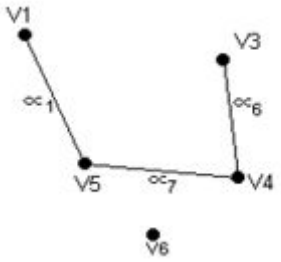
Desencolar()
Obtiene el primero de la cola y lo elimina.
{ pre: primero != null && cola != null }
{ post: 

Tabla de conversiones

Modificadora	(1)
Constructora	(2)
Analizadora	(3)

TAD GRAFO

UNIVERSIDAD ICESI

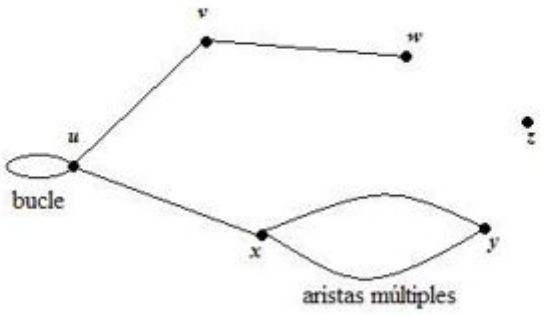


{inv: Los extremos de las aristas deben estar conectadas a un vértice }

Operaciones :

(1) CrearGrafo		→ Grafo
(2)AgregarVertice	Grago X Vertice	→ GrafoXBooleano
(2)EliminarVertice	Grafo X Vertice	→ Grafo X Booleano
(2)AgregarArista	Grafo X Vertice1X Vertice2 X peso	→ GrafoX Booleano
(1) EliminarArista	Grafo X Vertice1X Vertice2	→ GrafoX Booleano
(3) BFS	Grafo X Vertice	→ GrafoX IntegerX Arbol
(3) DFS	Grafo X Vertice	→ GrafoX IntegerX Arbol
(3) Dijkstra	Grafo X Vertice1X Vertice2	→ GragoX Lista de Integers
(3)FloydWarshall	Grafo	→ Grafo X Matriz
(3)Prim	Grafo	→ GrafoXArbol
(3)Kruskal	Grafo	→ GrafoXArboles
(3)DarAdyacentes	Grafo X Vertice	→GrafoX array de Vertices

CrearGrafo()
Se encarga de estructurar el Grafo para su funcionamiento
{ pre: TRUE }
{ post: }



AgregarVertice(Vertice)
“Agrega el vértice pasado por parámetro al grafo” {pre: {Vertices = <Vn>, Aristas =<An>}} Pre puede ser {Vertices = <V0, …, Vn>, Aristas = <A0,…, An> {post: { Vertices = <V0, …, Vn, Vn+1>}}

AgregarArista(Vertice1, Vertice2, Distancia)
“Agrega la arista pasada por parámetro entre dos vértices dentro del grafo” {pre: Vertices = <V0, V1, …, Vn>, Aristas = <A0, A1,…, An>} {post: Aristas = < A0, A1, …, An,An+1}

EliminarVertice(Vertice)
“Elimina el vertice pasado por parámetro” {pre: Vertices = <V0, V1, …, Vn>, Aristas = <A0, A1,…, An>} {post: { Vertices = <V0, …, Vn, Vn+1>}}

EliminarArista(Vertice1, Vertice2)

“Elimina la arista pasada por parámetro entre dos vértices dentro del grafo” {pre: Vertices = <V0, V1, …, Vn>, Aristas = <A0, A1,…, An>} {post: Aristas = <a, a0, A1, …, An-1}

BFS(Vertice)
“Recorre el grafo respecto a su anchura empezando por el vértice pasado por parámetro, colocando a este como padre de sus adyacentes y así sucesivamente” {pre: {Vértices = <V0, V1, …, Vn>, Aristas = <a0, a1, …, an>, V E Vértices del grafo}} {post: Un árbol y un entero indicando el total de vértices recorridos}

DFS()

“Recorre el grafo por profundidad asignando como padre el vértice adyacente por el que llegó” {pre: {Vertices = <V0, V1, …, Vn>, Aristas = <A0, A1, …, An>}} {post: Un bosque de objetos (vértices) donde cada árbol son los objetos conexos}

Dijkstra(Vertice, Vertice)
“Se encarga de encontrar el camino más corto entre los dos vértices” {pre: {Vértices = <v0, v1, …, Vn>, Aristas = <A0, A1, …,An> V1 & V2 E Vértices}} {post: Lista de enteros que indican el camino más corto y esto enteros el peso de las aristas de un lado a otro}

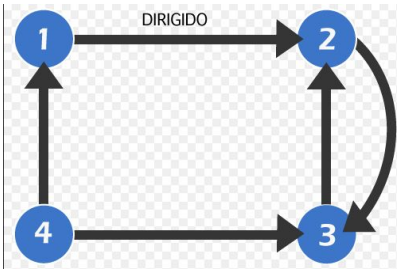
FloydWarshall()

“Busca el camino más corto entre todos los vértices” {pre: Vertices = {<V0, V1, …, Vn>, Aristas = <A0, A1, …, An>}} {post: Una matriz donde cada posición indica el peso mínimo entre los caminos de Vi a Vj}

Adyacentes(Vértice)
“Devuelve un arreglo con los vértices adyacentes a él” {pre: {Vertices = <v0, v1, …, vn>, Aristas = <a0, a1, …, am>} Vértice E Vertices} {post: Arreglo de vértices}

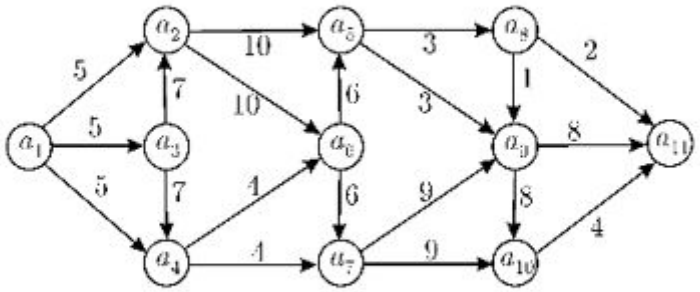
Modificadora	(1)

Constructora	(2)
Analizadora	(3)

TAD MultiGrafo Dirigido

{Inv: Las aristas tienen un sentido definido. Los extremos de las aristas deben estar conectadas a un vertice}
Operaciones: (2)CrearGrafo → Grafo (2) AgregarVertice Grago X Vertice → GrafoXBooleano (1)EliminarVertice Grafo X Vertice → Grafo X Booleano (2)AgregarArista Grafo X Vertice1X Vertice2 X peso → GrafoX Booleano (1)EliminarArista Grafo X Vertice1X Vertice2 → GrafoX Booleano

UNIVERSIDAD ICESI

CrearGrafo()
Se encarga de estructurar el Grafo para su funcionamiento
{ pre: TRUE }



AgregarArista(Grafo, Elemento, Elemento, Entero)
Se añade una arista a un vértice junto a su peso.
{ pre: Grafo != null && Elemento != null }
{ post: Se ha agregado arista exitosamente. }

BorrarArista(Grafo, Elemento, Elemento)
Modifica el grafo eliminando una arista de un vértice.
{ pre: Grafo != null && Element != null}
{ post : TRUE si está vacio. FALSE si no está vacio }

BorrarNodo(Grafo, Elemento)
Se encarga de suprimir un vértice del grafo.
{ pre: Grafo != nulll && Elemento != null }
{ post : Se ha eliminado correctamente. Error en caso contrario}

BorrarArista(Grafo, Elemento, Elemento)
Modifica el grafo eliminando una arista de un vértice.
{ pre: Grafo != null && Element != null}
{ post : TRUE si está vacio. FALSE si no está vacio }

Diseño de pruebas:

Clase	Método	Escenario	Valores	Resultado
GraphTest	addVertexTest()	Se instanció un objeto de tipo Grafo.	Se agregan los siguientes esferas al grafo: vertex=" vertice1" key="0"	Los objetos han sido agregados al grafo de forma adecuada, cada vértice del grafo contiene la información correcta
GraphTest	removeVertexTest()	Se instanció un objeto de tipo Grafo, se crearon 5 vértices y 8 aristas y se agregar	Key="0"	El objeto fue eliminado exitosamente ya no hace parte del grafo

		on al grafo.		
GraphTest	searchTest()	Se instanció un objeto de tipo Grafo .	esfera=” A”	El objeto ha sido encontrado en el grafo.
GraphTest	insertEdgeTest	Se instanció un objeto de tipo Grafo .	Vertex1=” vertice0” Key1=”0” vertex2=” vertice1” key2=”1” edge=6.0 keyEdge=”0”	Los objetos han sido agregados al grafo de forma adecuada, la arista del grafo contiene los vértices correctos
GraphTest	removeEdgeTest	Se instanció un objeto de tipo Grafo .	Vertex1=” vertice0” Key1=”0” vertex2=” vertice1” key2=”1” edge=6.0 keyEdge=”0”	El objeto fue eliminado exitosamente ya no hace parte del grafo

GraphTest	areAdjacentsTest	Se instanció un objeto de tipo Grafo, se crearon 5 vértices y 8 aristas y se agregaron al grafo.	Key=”0” Key2=”1” Edge=6.0 keyEdge=”0”	El resultado es true, dado que es un nodo no dirigido para ambos sentidos el resultado es correcto
GraphTest	dikstraTest	Se instanció un objeto de tipo Grafo, se crearon 5 vértices y 8 aristas y se agregaron al grafo.	Way1=8+10 Way2=14+13 Way3=14+9+12 Way4=8+20+12	Se comparan todos los posibles caminos de un vertive inicio a un vértice final para verificar que Dijkstra retono el de menor valor

GraphTest	floydWarshallTest	Se instanció un objeto de tipo Grafo, se crearon 5 vértices y 8 aristas y se agregaron al grafo.		Se buscan dos caminos corto de un vértice a otro con dijkstra y se revisa la matriz de Floyd para saber si encontró los caminos minimos hacia todos los vertices del grafo
GraphTest	PrimTest	Se instanció un objeto de tipo Grafo, se crearon 5 vértices y 8 aristas y se agregaron al grafo.	menorEdge=7 arrayD=dijkstra desde el vertice de la arista 7	El camino hallado por dijkstra desde la arista de menor peso debe ser igual al camino hallado por prim

Fase 7: Implementación del diseño
Lista de tareas a implementar.
-Crear Grafo
-Mostrar personaje
-Mostrar Mapa
-Mostrar Esferas
-Mostrar radar del Dragón
-Mostrar Grafo
-Mostrar Ganador
-Mostrar lista de Usuarios

Nombre	**Crear Grafo**	public StartGame() throws IOException {
		sound = new Sonidos();
		queue = new PriorityQueue<Player>();
		balls = new ArrayList<Ball>();
		points = new PointM[30];
		routes= new String[30];
		graph= new Graph<String,PointM>();
		// r

Descripción	Este método crea un grafo con vértices y aristas generadas aleatoriamente, que posteriormente será mostrado en la interfaz	
Entrada	Vértices, Aristas	
Salida	Grafo conexo	

Nombre	**Mostrar personaje**	public void perpetumMotus() {
		hilo = new Timeline(new KeyFrame(Duration.ZERO, e -> {
		
		start.getCharacter().motus();
		
		character.setImage(new Image(start.getCharacter().url));

Descripción	Este método muestra en la interfaz con el usuario, el personaje que se moverá por el mapa (espacio)	
Entrada	ruta Imagen	
Salida	Personaje seleccionado, con su	

	respectiva imagen de movimiento	character.setLayoutX(start.getCharacter().x);
		
		character.setLayoutY(start.getCharacter().y);
		
		}), new KeyFrame(Duration.millis(SPEED)));
		
		hilo.setCycleCount(Animation.INDEFINITE);
		hilo.play();
		}

Nombre	**Mostrar Mapa**	public boolean checkSpaceLayout() {
		boolean what= false;
		if (space.getLayoutX()<LIMITE_IZQUIERDO&&space.getLayoutX()>LIMITE_DERECHO&&space.getLayoutY()<LIMITE_ARRIBA&&space.getLayoutY()>LIMITE_ABAJO) {
		what= true;
		}
		return what;
		}

Descripción	El método permite mostrar un mapa donde puede moverse un personaje ya seleccionado	
Entrada	ruta imagen	
Salida	se muestra correctamente el mapa	

Nombre	**Mostrar Esferas**	
Descripción	En la interfaz se podrá ver las esferas una vez estén reveladas, estas son 7 y están por todo el mapa	
Entrada	Ruta de imagen	
Salida	Se muestra correctamente la esfera con ruta seleccionada	

Nombre	**Mostrar radar del Dragón**	
Descripción	Al finalizar la partida del juego, se muestra el grafo que se debió haber cumplido en el juego dentro del radar	
Entrada	Vértices, Aristas	
Salida	Grafo conexo mostrado en el radar	

Nombre	**Mostrar Ganador**	
Descripción	Al final se muestra el mejor puntaje de entre los usuarios	
Entrada	Lista de usuarios	
Salida	Se mostró con éxito	

Nombre	**Mostrar lista de Usuarios**	
Descripción	Se muestra la lista de los usuarios ya registrados	
Entrada	Lista de usuarios	
Salida	Se mostró con éxito	