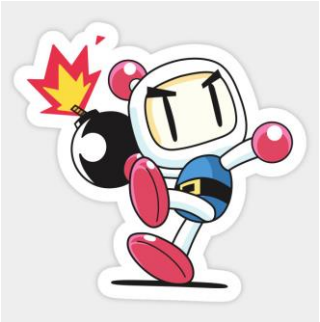
Álvaro Escobar
Luis Alfredo Rodríguez
Sebastian Correa

Método de la ingeniería

Enunciado



Bomberman es una franquicia de videojuegos estratégico-laberínticos originalmente desarrollada por Hudson Soft y actualmente por Konami.

Debido a que esta franquicia de videojuegos trabaja con problemas de tipo laberínticos, se propone diseñar un nuevo juego de Bomberman donde el personaje debe de cruzar de forma efectiva un mapa desde un punto de inicio (Salida) hasta un punto de llegada (Meta).

Ahora bien, se debe tener en cuenta que solo se podrá realizar cierta cantidad de pasos por nivel haciendo que sea un juego a su vez estratégico donde los jugadores deberán romper en algunas ocasiones unas cajas o bloques removibles mediante las bombas que puede colocar el personaje sobre la superficie del mapa o rodear obstáculos del mapa los cuales no pueden ser destruidos, de forma que se utilice la menor cantidad de pasos posibles para poder completar el nivel.

El desarrollo e implementación de este nuevo videojuego se realizará utilizando un grafo para la representación de los posibles movimientos que podrá realizar el jugador, cada casilla del mapa será un nodo del grafo recordando que algunas de estas estarán ocupadas por obstáculos.

Al colocar una bomba en alguna casilla del mapa se debe examinar hasta dónde llegará su onda expansiva y qué bloques destruirá, con cuáles se estrellara y por último en los bloques libres se debe respetar el alcance de la bomba. Es necesario aclarar también que, como este videojuego es una variante del Bomberman original, la onda expansiva de la bomba no hará daño al personaje.

Requerimientos funcionales

Nombre	R.1 Crear Partida
Resumen	Se genera una nueva partida donde el jugador tendrá 3 vidas inicialmente.
Entrada	- Nickname
Salida	Nueva Partida

Nombre	R.2 Desplazar personaje
Resumen	Permite al personaje desplazarse en el mapa en un rango de 1 casilla a la redonda, teniendo en cuenta que no podrá cruzar a través de obstáculos y disminuye el número de pasos máximos.

Entrada	- Posición de destino - Cantidad de pasos máximos
Salida	Se desplaza al personaje en la dirección indicada

Nombre	R.3 Poner Bombas
Resumen	Permite al personaje colocar una bomba sobre la casilla en la cual se encuentra (Las bombas no harán daño al personaje).
Entrada	- Posición actual
Salida	Se coloca la bomba y esta estalla después de unos segundos.

Nombre	R.4 Calcular el camino más corto
Resumen	Calcula el camino más corto entre la Salida y la Meta, ignorando los bloques removibles pero sin poder pasar por las casillas bloqueadas.
Entrada	- Salida - Meta
Salida	Retorna el número de pasos que debe hacer para realizar el recorrido por el camino más corto.

Nombre	R.5 Explotar bombas
Resumen	Permite que la bomba estalle de forma que se debe recorrer el mapa teniendo en cuenta el alcance de la bomba y las propiedades de las casillas por donde pasa su onda de expansión.
Entrada	- Posición de la bomba
Salida	La bomba estalla y realiza cambios en el mapa según las propiedades de las casillas.

Recopilación de datos

Para la solución del problema identificado se propone el uso de una estructura de datos llamada grafo. Esta se usará para poder averiguar el camino más corto desde el punto de entrada del jugador, hasta el punto de salida del mapa. En primera instancia hay que saber que es un grafo y un árbol de recubrimiento mínimo.

- **Grafo:** Es un tipo de estructura de datos no lineal que primordialmente consiste en nodos y aristas; los nodos siendo el dato a guardar en la estructura de datos y las aristas siendo las relaciones o conexiones que hay entre un par de nodos. Los grafos son usualmente usados para representar redes que pueden ser caminos por la ciudad o una red telefónica.

- **Minimum Spanning Tree (MST):** También llamado en español como árbol de recubrimiento mínimo. Es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Cada arista tiene asignado un peso proporcional entre ellos, que es un número representativo de algún objeto; y se usa para asignar un peso total al árbol recubridor mínimo computando la suma de todos los pesos de las aristas del árbol en cuestión. El MST se da en grafos conexos y no dirigidos.

Los siguientes algoritmos permiten recorrer el grafo de 2 formas distintas, por amplitud y profundidad:

- **Depth First Search (DFS):** Es un algoritmo recursivo para buscar todos los nodos de un grafo, ya que este recorre todo el grafo. La implementación genérica de este algoritmo pone los nodos en dos categorías: visitados o no visitados, con el fin de evadir ciclos tediosos si se da un ciclo repetitivo. Su funcionamiento es el siguiente:
    a. Comenzar poniendo cualquiera de los vértices del grafo en la parte superior de una pila.
    b. Tomar el elemento superior de la pila y agrégalo a la lista visitada.
    c. Crear una lista de los nodos que tienen adyacencia a ese vértice para después agregar los que no están en la lista visitada en la parte superior de la pila.
    d. Repetir los pasos (b) y (c) hasta que la pila esté vacía.

- **Breadth First Search (BFS):** Es un algoritmo de desplazamiento en el que debe comenzar desde un nodo seleccionado y recorrer el grafo por capas, explorando primero los nodos vecinos (nodos que están conectados con aristas al nodo seleccionado). Luego debe seguir hacia los nodos vecinos del siguiente nivel. Como sugiere el nombre de este algoritmo, se debe cruzar el grafo horizontalmente de la siguiente manera:
    a. Primero se debe mover horizontalmente y visitar todos los nodos de la capa actual.
    b. Ir a la siguiente capa y repetir el paso (a).

A continuación algunos de los algoritmos más conocidos y regularmente utilizados para calcular el camino más corto en un grafo:

- **Dijkstra's Algorithm:** Es un algoritmo que sirve para la determinación del camino más corto desde un vector de origen dado, hasta un vector final, teniendo en cuenta que las aristas tienen un peso, este algoritmo tomará la decisión de cuál es el camino más corto entre todos los caminos posibles que hay en un grafo. Adicionalmente, el peso de las aristas no puede ser negativo.

- **Floyd-Warshall Algorithm:** Es un algoritmo que calcula la ruta más corta de todos los pares, lo que significa que calcula la ruta más corta entre todos los pares de nodos en un grafo, siempre y cuando el grafo no contiene ningún ciclo o aristas con peso negativo.

- **Bellman-Ford Algorithm:** Este algoritmo calcula el camino más corto desde un nodo origen dado, hasta el nodo final en un grafo dirigido. En este algoritmo el peso de las aristas puede ser negativo.

- **Prim Algorithm:** Es un algoritmo que permite encontrar el árbol de mínimo recubrimiento de un grafo conexo, no dirigido y cuyas aristas tienen un peso. Donde

el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol de mínimo recubrimiento para uno de los componentes conexos que forman dicho grafo no conexo.

- **Kruskal Algorithm:** Es un algoritmo para encontrar un árbol de recubrimiento mínimo en un grafo conexo y ponderado. Esto quiere decir que 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. Si el grafo no es conexo, entonces busca un bosque expandido mínimo.

Búsqueda de soluciones creativas

Teniendo en cuenta que la forma más eficiente de resolver el problema es con la estructura de datos grafo y que esta cuenta con muchas utilidades, además de diferentes algoritmos, se plantean los siguientes algoritmos para calcular el camino mínimo como posibles soluciones para la implementación del aplicativo:

➔ **Dijkstra's:** Este algoritmo tiene como restricción que no funciona para grafos con pesos negativos en sus aristas. Cabe destacar que este algoritmo puede retornar un arreglo de distancias donde se puede observar el peso mínimo de un nodo elegido inicialmente a un nodo objetivo. Si se desea, no se retorna el arreglo de distancias, ya que pueden haber varias rutas cuya distancia sea la misma, por eso otra opción de retorno puede ser la ruta más corta como tal o "arreglo de padres", que es una lista de nodos (también puede representarse como una pila).

Complejidad Temporal

- Para matriz de adyacencia tiene complejidad $O(V^2)$.
- Para lista de adyacencia tiene complejidad $O(E \log V)$.

➔ **Floyd-Warshall:** Este algoritmo tiene como restricción que no se puede implementar en la representación del grafo como lista de adyacencias. A diferencia de Dijkstra, este algoritmo muestra la ruta más corta de un nodo cualquiera, a otro nodo cualquiera, es por esto que la representación del grafo necesaria en este caso debe ser la matriz de adyacencia. Además, una ventaja del algoritmo de Floyd-Warshall es que este puede ser utilizado por grafos que presenten aristas cuyo peso o ponderación sean de valor negativo. La complejidad temporal es $O(V^3)$.

➔ **Prim:** Este algoritmo tiene como restricción que solamente puede ser implementado en grafos que sean no dirigidos ponderados. Además el algoritmo no busca como tal el camino más corto sino que retorna el MST (Árbol de recubrimiento mínimo) del grafo, que es un subgrafo como tal del grafo original. Además, es requisito para usar este algoritmo que el grafo sea conexo, puesto que al retornar un árbol de recubrimiento mínimo se está asegurando que todos los nodos van a ser recubiertos.

Complejidad Temporal

- Para matriz de adyacencia tiene complejidad $O(V^2)$.

- Para lista de adyacencia tiene complejidad O(E log V) usando montículos binarios y O(E + V log V).

➔ **Kruskal:** Este algoritmo tiene como restricción que solamente puede ser implementado en grafos que sean no dirigidos ponderados. Además el algoritmo no busca como tal el camino más corto si no que calcula el MST (Árbol de recubrimiento mínimo) del grafo. A diferencia del algoritmo de Prim, el algoritmo de Kruskal no necesariamente es para grafos conexos, puesto que este puede retornar una lista de árboles (un bosque) o varios subgrafos, mientras que Prim solo retorna uno.

Complejidad Temporal

- Para matriz de adyacencia tiene complejidad $O(V^2)$.
- Para lista de adyacencia tiene complejidad O(E log V).

Además de los algoritmos anteriores se debe reflexionar sobre cuáles algoritmos se utilizaran para solucionar el problema de las bombas al explotar para las cuales se necesita realizar un recorrido completo en el grafo, los algoritmos son los siguientes:

➔ **Breadth First Search (BFS):** En este caso, la búsqueda en anchura o BFS como es popularmente conocida, no es una buena opción para el proyecto puesto que si se utilizara esta estrategia, al poner una bomba la explosión recorrería las adyacencias de las adyacencias del nodo donde se puso la bomba, y así sucesivamente. Es decir, el BFS realizaría algo totalmente opuesto a lo que se busca, que es que la bomba tenga una dirección y un camino fijo que recorrer.

➔ **Depth First Search (DFS):** Para el caso que corresponde a este proyecto, es el mejor algoritmo de recorrido que se pueda usar. Lo anterior debido a que, hablando ya en términos específicos de este proyecto, se necesita que cada adyacencia del nodo donde se ponga una bomba haga un recorrido en profundidad hasta que se encuentre con un muro que se pueda romper, un muro que no se pueda romper, hasta donde lo permita el alcance de la explosión, o hasta que se encuentre con el personaje principal Bomberman, en su defecto.

Selección de la mejor solución

Ante los algoritmos descritos en la sección de búsqueda de soluciones creativas se pueden encontrar algunas restricciones, ventajas y desventajas. Para seleccionar los algoritmos adecuados o la mejor solución del problema, se van a descartar los algoritmos vistos en el curso de estructuras de datos por su uso y/o restricciones.

- El algoritmo de Prim se utiliza para encontrar el árbol/subgrafo de recubrimiento mínimo para un grafo conexo, es decir, no es de utilidad para solucionar los problemas inicialmente descritos en este documento, pues se busca encontrar la ruta más corta de una meta a una llegada, no recorrer todos los nodos presentes en el grafo.

- El algoritmo de Kruskal tiene una utilidad similar a la de Prim, solo que como se mencionaba en el apartado de búsqueda de soluciones creativas, el grafo no necesariamente conexo, y tiene varios posibles retornos.

Por las razones anteriormente mencionadas, los algoritmos de Prim y Kruskal no pertenecen a la selección de la mejor solución.

Ahora, los algoritmos de ruta de peso mínimo o ruta más corta:

- El algoritmo de Dijkstra utiliza un nodo de salida para buscar la ruta más corta hasta un nodo de llegada, lo que se asemeja bastante al problema inicialmente descrito de encontrar la ruta en la cual Bomberman tenga que usar la menor cantidad de pasos posible. Además, se cumple con la restricción de que no hayan aristas cuyo peso sea negativo.

- El algoritmo de Floyd-Warshall sirve como algoritmo de ruta más corto para cualquier nodo salida usando cualquier nodo como destino, es decir, una matriz de adyacencia. Para este caso no es una buena alternativa pues lo que se busca es ser específicos eligiendo un nodo como salida y un nodo como destino para la ruta más corta. Además su complejidad es demasiado elevada respecto a la complejidad de Dijkstra.

Por las razones anteriormente mencionadas, el algoritmo de Dijkstra es la mejor solución para encontrar la ruta más corta que debería usar Bomberman de manera que el número de pasos que dé para llegar a su destino sea el menor, teniendo en cuenta las aplicaciones de ambos algoritmos, sus restricciones y la complejidad correspondiente a Dijkstra de $O(E \log V)$ u $O(V^2)$, dependiendo de la representación del grafo, frente a la complejidad $O(V^3)$ de Floyd-Warshall.

Finalmente, los algoritmos de recorridos:

- El algoritmo BFS haría que para cada adyacencia del nodo o vértice donde se ponga la bomba, se recorran sus demás adyacencias, es decir que la bomba tendría una explosión diferente a la que se busca inicialmente, por lo que se vería algo así:

- El algoritmo DFS es el mejor algoritmo de recorrido para solucionar el problema de la explosión de la bomba, puesto que se asemeja al problema inicial y cumple algunas de sus características. Las otras características se pueden controlar con condicionales, como por ejemplo detener la explosión al encontrar un muro que no se pueda romper. El algoritmo adaptado a las características del juego y las condiciones de parada de la explosión, harían que la explosión se viera algo así:

Por lo tanto el mejor algoritmo de recorrido y el que más se ajusta al problema es la búsqueda en profundidad o DFS.

Decisión Final: por todo lo anteriormente expuesto, se ha decidido que la mejor manera (y la que se va a implementar, por supuesto) de encontrar la ruta más corta es usando el algoritmo de Dijkstra. Respecto a la manera de simular la explosión de la bomba, el mejor algoritmo es DFS y por consiguiente será el que se va a implementar en la solución del problema.

Diseño de pruebas unitarias

Clase	**Método**	**Escenario**	**Entrada**	**Resultado**
Graph	addVertex();	setUp1()	Elemento que se quiere agregar al grafo.	Se agrega un elemento al grafo, por lo tanto el tamaño del mismo debe ser 1 y al buscarlo en el grafo se debe encontrar el elemento agregado.

Clase	**Método**	**Escenario**	**Entrada**	**Resultado**
Graph	addEdge();	setUp2()	Dos elementos existentes en el grafo, el primero es de donde viene la arista y el segundo elemento es a dónde va la misma.	Se crea una arista entre el elemento 1 y el elemento 2, por lo tanto al buscar si existe la conexión debe ser verdadero.

Clase	Método	Escenario	Entrada	Resultado
Graph	removeVertex();	setUp2()	Elemento que se quiere eliminar del grafo.	Se elimina un elemento del grafo, por lo tanto el tamaño del mismo debe ser menor y al buscarlo en el grafo no se puede encontrar, al igual que sus aristas correspondientes.

Clase	Método	Escenario	Entrada	Resultado
Graph	removeEdge();	setUp2()	Dos elementos que estén el grafo. El primero que es donde empieza arista y el segundo elemento que es a dónde va la arista.	Se elimina la arista que está entre los dos nodos. Por lo que en la lista de adyacencias o matriz de adyacencia no debe aparecer esa conexión.

Clase	Método	Escenario	Entrada	Resultado
Graph	containsVertex();	setUp2()	Elemento que se quiere buscar en el grafo.	Retorna verdadero si el elemento está en el grafo y retorna falso de ser caso contrario.

Clase	Método	Escenario	Entrada	Resultado

Graph	isEdge();	setUp2()	Dos elementos que se cree que hay una arista entre ellos.	Si existe la arista entre los dos elementos retorna verdadero, en caso contrario retorna falso.

Clase	**Método**	**Escenario**	**Entrada**	**Resultado**
Graph	bfs();	setUp2()	Elemento inicial en el cual el algoritmo va a recorrer el grafo	Recorre el grafo

Clase	**Método**	**Escenario**	**Entrada**	**Resultado**
Graph	dfs();	setUp2()	Elemento inicial en el cual el algoritmo va a recorrer el grafo.	Se espera que el algoritmo recorra el grafo de manera Pre-Order

Diagrama de clases

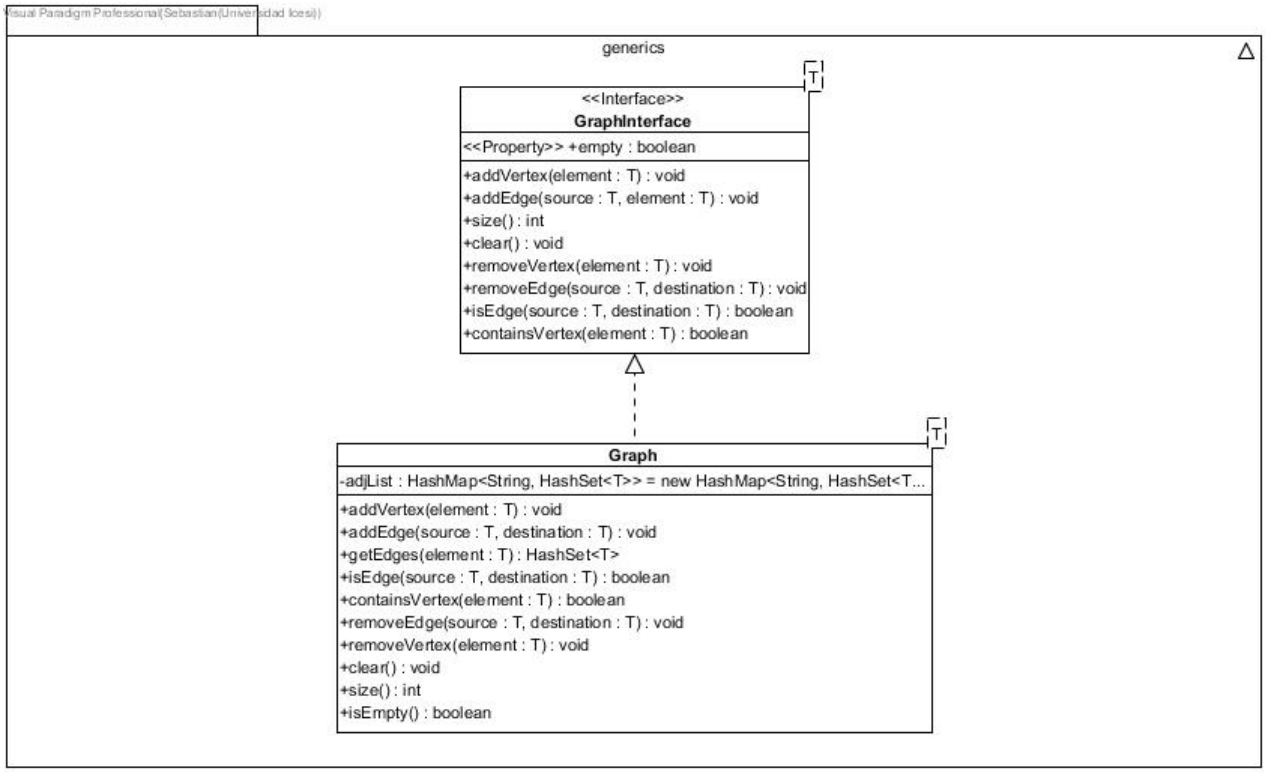


Diagrama de objetos