**PRÁCTICA 1:**

**Puzle deslizante 8 piezas**

**Asignatura**:

Inteligencia Artificial para Videojuegos

**Alumnos**:

Adriana del Castillo Espejo-Saavedra

Raúl Serrano Gómez

**ÍNDICE**

1. Entorno de desarrollo
2. Descripción del problema
3. Propuesta de IA
4. Resultados y pruebas
5. Implementación
6. Referencias
7. **Entorno de desarrollo**

Hemos desarrollado la práctica con el IDE Unity en la versión 2017.3, programando en C# con Visual Studio Community 2017.

1. **Descripción del problema**

El problema consiste en desarrollar un prototipo de IA que resuelva el puzle deslizante de 8 piezas.

El prototipo muestra el puzle en su posición inicial, que es una configuración de la forma:

(1,2,3

,4,5,6,

7,8,0)

en la que el 0 representa el hueco vacío.

El usuario puede generar un puzzle aleatorio pulsando el botón de “Aleatorio”.

El usuario puede mover las piezas haciendo click sobre una pieza adyacente al vacío. El número de movimientos que puede hacer el jugador es ilimitado.

Si el jugador se atasca, puede pulsar el botón de “Reestablecer” que reestablece el estado del tablero al que tenía al principio.

El puzzle puede ser resuelto por una IA que puede resolverlo mediante dos algoritmos:

1. BFS, o búsqueda en anchura
2. DFS, o búsqueda en profundidad

El jugador elige resolver el puzzle mediante un prototipo u otro pulsando los botones “BFS” o bien “DFS”.

Si el puzzle no se puede resolver porque el número de cálculos que tiene que realizar la IA es muy elevado o bien porque dado un corte (En el caso del DFS), este no es suficiente para llegar a la configuración óptima, se mostrará un mensaje por pantalla explicando el motivo por el cual no se ha podido resolver el puzzle.

\*Estadísticas

Hay que utilizar al menos dos métodos diferentes de búsqueda no informada

Estadísticas que muestra:

○ Número de movimientos totales

○ Tiempo empleado

○ Número de nodos abiertos

○ Nivel de profundidad

○ ¿Memoria utilizada?

1. **Propuesta de IA**

Hemos optado por realizar los algoritmos BFS y DFS, que son algoritmos de búsqueda no informada, es decir, únicamente cuentan con la información que les proporciona el problema.

Los algoritmos Bfs y Dfs van hallando el recorrido mediante expansión de nodos. Estos nodos contendrán lo siguiente:

-La configuración del tablero

-Una referencia al nodo padre

-Un operador, que será aquel se ha aplicado al nodo padre para generar este nodo hijo

-El coste de la ruta desde la raíz hasta el nodo actual

Vamos hallando los nodos adyacentes de cada nodo recorrido para encontrar todas las configuraciones posibles hasta dar con la solución.

BFS

El algoritmo Bfs expande primero la raíz y a continuación expande al completo cada nivel de profundidad.

Para poder guardar la información de los nodos, utilizamos el TAD Cola, para ir guardando los nodos y una tabla hash que va guardando en este caso los tableros visitados.

DFS

El algoritmo Dfs expande hacia abajo hasta llegar al más profundo y después sube lo justo para seguir expandiendo nodos de hijos profundos. Esto lo hace mediante recursividad y vuelta atrás, métodos que nos permiten ahorrar espacio.

La estrategia no es óptima ni completa, es decir, puede haber rutas redundantes en las cuales el algoritmo profundice eternamente. Sin embargo, nosotros lo he hemos solucionado con profundidad limitada.

1. **Resultados y pruebas**

Los algoritmos son reutilizables ya que toda la implementación por dentro es independiente de a qué se está aplicando. Lo único que tendríamos que modificar es el hecho de recibir como parámetros los tableros de inicio y fin, que son matrices, y deberían ser tipos genéricos.

Tenemos implementadas dos escenas: la escena del menú, con el título, nuestros datos e instrucciones de uso; y la escena del juego, con los botones de generar el tablero aleatorio, reestablecer el tablero, y para generar los recorridos Bfs y Dfs.

Se pueden repetir pruebas de inmediato y sin límites.

Lo hemos exportado a Android para juntar nuestras prácticas y llevarlas como portfolio, y ver si funcionaba y nos ha funcionado.

Las estadísticas se muestran por pantalla: número de movimientos, tiempo tardado.

Ejemplo completo de la prueba:

No hemos implementado ninguna heurística, sino los algoritmos de búsqueda no informada porque nos parecían los más diferentes entre sí y los más complejos, ya que los de heurística, como Hamming o Manhattan solamente consistían en utilizar el algoritmo Bfs y ordenar la cola.

Hemos visto que al implementar Bfs, es un algoritmo que funciona perfectamente, sin embargo cuando se realizan muchas iteraciones el coste es tan elevado que tarda mucho.

En el caso del Dfs, hemos observado que funciona para muchas iteraciones, pero se equivoca mucho y hace muchos movimientos redundantes. Esto quedará reflejado más adelante con el ejemplo que mostraremos de Dfs.

1. **Implementación**

-Al inicio del problema, buscamos todos los gameObjects con el tag “Ficha” y los ordenamos el array por nombre.

-Inicializamos el tablero

1. **Referencias**

**Para saber como funciona el puzle deslizante, hemos utilizado:**

○ How Sliding Puzzles Work <https://entertainment.howstuffworks.com/puzzles/sliding-puzzles.htm>

Sliding Piece Puzzles

Para la representación de los números hemos utilizado estos sprites con licencia libre:

https://opengameart.org/content/numbers-blocks-set-01

**https://opengameart.org/content/background-texture-for-a-board-game**