Universidad Rey Juan Carlos

MÓSTOLES
ETSII/ Grado de Desarrollo y Diseño de Videojuegos

PRÁCTICA 1.

PROYECTO PRÁCTICO PARTE 1

18-11-2022

Desarrollo de juegos con Inteligencia Artificial

GRUPO 3.

Daniel Capilla Sánchez
Noelia Nayara Maqueda Soto

ÍNDICE

1. DESCRIPCIÓN DEL ALGORITMO EMPLEADO PARA RESOLVER EL PROBLEMA 1

Para realizar esta práctica, se han utilizado los materiales dados por el profesorado de la asignatura, siendo estos: un paquete de Unity (conformado por escenas y scripts necesarios para la realización de la práctica, un PDF con el enunciado y un pseudocódigo).

Siguiendo el enunciado proporcionado, se ha debatido el posible algoritmo necesario para el controlador del escenario PathFinding. Este escenario está compuesto por un tablero (haciendo alusión a un laberinto) en el cuál se encuentran: una casilla de salida, una casilla de meta, un agente (personaje controlado por Inteligencia Artificial) y varios obstáculos que no puede atravesar. Los componentes más importantes del escenario son la casilla de meta y los obstáculos repartidos, ya que son los elementos que aparecen de manera aleatoria dependiendo de una semilla especificada en el **Loader**.

Teniendo todo esto en cuenta, el primer objetivo de la práctica es implementar un algoritmo de búsqueda eficiente offline que ayude al agente a encontrar la meta e ir hasta ella evitando los obstáculos desde su posición inicial.

Para cumplir con lo pedido, se ha decidido emplear el **algoritmo A***. El algoritmo A* es un algoritmo de búsqueda offline que, en caso de que su heurística sea optimista, siempre será óptimo y encontrará siempre el mejor camino. A* guía la búsqueda y exploración de los nodos en función del mínimo valor de f*, es decir, comenzará a explorar desde el nodo con menor valor de f* (el nodo que se encuentre en la primera posición de la lista abierta). Si este nodo es el nodo meta, el algoritmo lo devolverá y terminará la exploración, en caso contrario, lo expandirá y continuará la exploración de los nodos. Concretamente, se calcularán los nodos sucesores del nodo actual mientras se calcula el valor de f* en cada uno de ellos para poder insertarlos después en la lista abierta según el valor de f* creciente para que el primer nodo de la lista abierta siempre sea el de menor f* y el siguiente en ser explorado. Este proceso se repite hasta que el nodo meta sea encontrado.

Por otro lado, el algoritmo A* es un algoritmo con **heurísticas débiles**, por lo que se tiene que definir una función heurística para calcular el valor de h*, el cuál, es necesario para poder calcular el valor de f*. Para ello, el primer problema planteado en la práctica es

encontrar la forma de que el agente llegue a la casilla de meta. Para resolverlo, se ha empleado como función heurística la **distancia Manhattan**. De esta forma, aunque no se aporta del valor de h* en cada casilla del tablero, esta función heurística permite calcular manualmente estos valores basándose en la distancia entre el nodo actual (casilla de salida o el lugar donde inicia el agente) y el nodo meta (casilla de meta o GOAL).

Siguiendo con la explicación de la **distancia Manhattan**, este tipo de función heurística es muy solicitada a la hora de definir problemas de encontrar rutas (distancia entre ciudades, laberintos, etc.), además de que en esta práctica, al tener todos los operadores con coste 1, la función heurística es optimista. Esto implicaría que el algoritmo A* es óptimo, es decir, encontrará la mejor solución y, además, es completo, ya que la solución siempre la encontrará.

2. CARACTERÍSTICAS DEL DISEÑO E IMPLEMENTACIÓN

Para implementar el algoritmo A*, se ha empleado el pseudocódigo proporcionado en el Aula Virtual. En él se desarrollaba la clase *AStarSolution* que permite la búsqueda de caminos mediante el Algoritmo A* que busca el mejor camino para conducir al agente (el personaje) a la casilla de meta (GOAL).

Nodo

Para poder utilizar este pseudocódigo, se ha creado una clase auxiliar denominada *Nodo* para poder representar y crear los nodos del árbol de búsqueda. En esta clase se representan las propiedades necesarias de los nodos para su correcta presentación. Las características son las siguientes:

```
public class Nodo
{
    public CellInfo coord; //Coordenadas del nodo actual
    public Nodo padre; //Nodo padre
    public Locomotion.MoveDirection ProducedBy; //Movimiento que ha hecho el padre
    public float dist; //Distancia al nodo final con el coste
    public float distaciaAcumulada;
```

CellInfo coord: Coordenadas del nodo actual.

Nodo padre: Nodo padre de los hijos.

Locomotion.MoveDirection ProduceBy: Calcula el movimiento del agente a través del tablero. Para ello almacena el movimiento que hizo el padre para obtenerlo (up, Right, Down, Left).

float dist: Variable tipo float para almacenar la distancia (f*).

float distanciaAcumulada: Almacena el coste total de las acciones de los nodos.

Para crear los nodos se necesita de un constructor (*Nodo*) que toma como propiedades un objeto de tipo *CellInfo* llamada *v* que guardará las coordenadas y un objeto de tipo *Nodo* llamado *padre*. La *distanciaAcumulada* se inicializará solo si existe un padre (el primer nodo no tiene padre) y será la suma del coste actual del nodo más todo el coste anterior de sus padres.

```
public Nodo(CellInfo v, Nodo father)
{
    coord = v;
    padre = father;
    if(padre != null) //El primer nodo no tiene padre
    {
        distaciaAcumulada = v.WalkCost + padre.distaciaAcumulada; //Suma de los costes: el actual y el anterior
    }
```

Se crea una función denominada DistanciaM que toma como valor un objeto del tipo *CellInfo* llamado *destinoFinal*, este parámetro le proporciona a la función la posición exacta en la que se encuentra la casilla de meta o GOAL. Aquí, se calcula la función heurística mediante la distancia Manhattan. Utilizando la variable de tipo float *dist* inicializada anteriormente, se almacenará la distancia que hay entre el agente y la meta. El cálculo se realizará sumando los valores absolutos de las diferencias de filas y columnas entre las casillas de salida y meta. También se suma el coste total del paso entre las casillas.

```
public void DistanciaM(CellInfo destinoFinal)
{
    dist = Mathf.Abs(coord.ColumnId - destinoFinal.ColumnId) + Mathf.Abs(coord.RowId - destinoFinal.RowId) + distaciaAcumulada;
}
```

A*

En la clase AStarSolution, que hereda de la clase AbstractPathMind, es la clase que implementa el algoritmo A* que calcula el camino hasta la meta (mediante la función Search()) y, después, obtiene los movimientos que el agente debe realizar a lo largo del tablero para lograr su objetivo (GetNextMove()). Para poder utilizar esta clase proporcionada en el pseudocódigo se han utilizado la clase auxiliar *Nodo* y algunos métodos auxiliares explicados posteriormente.

```
// declarar Stack de Locomotion.MoveDirection de los movimientos hasta llegar al objetivo
private Stack<Locomotion.MoveDirection> currentPlan = new Stack<Locomotion.MoveDirection>(); //Pila de direcciones
```

```
public override void Repath()
{
    currentPlan.Clear();
}
```

```
public override Locomotion.MoveDirection GetNextMove(BoardInfo board, CellInfo currentPos, CellInfo[] goals)
{
    // si la Stack no está vacía, hacer siguiente movimiento
    if (currentPlan.Any())
    {
        // devuelve siguiente movimiento
        return currentPlan.Pop();

    }

    // calcular camino, devuelve resultado de A*
    var searchResult = Search(board, currentPos, goals); //Devuelve un nodo que tienen padres

    // recorre searchResult and copia el camino a currentPlan
    while (searchResult.padre != null)
    {
        currentPlan.Push(searchResult.ProducedBy);
        searchResult = searchResult.padre;
    }

    // returns next move (pop Stack)
    if (currentPlan.Any())
        return currentPlan.Pop();

    return Locomotion.MoveDirection.None;

}
```

```
private Nodo Search(BoardInfo board, CellInfo start, CellInfo[] goals)
{
    // crea una lista vacía de nodos
    var open = new List<Nodo>();

    // nodo inicial
    var n = new Nodo(start, null);

    // añade nodo inicial a la lista
    open.Add(n);

    // mientras la lista no esté vacia
    while (open.Any())
    {
        // mira el primer nodo de la lista

        // si el primer nodo es goal, returns current node
        if (open[0].coord == goals[0])
        {
            return open[0];
        }
        else
        {
            var hijos = open[0].coord.WalkableNeighbours(board);
            // expande vecinos (calcula coste de cada uno, etc)y los añade en la lista
            for (var i = 0; i < hijos.Length; i++)
            {
                //Revisa si las posiciones vecinas son transitables
                if (hijos[i] != null)
                {
                    var nHijos = new Nodo(hijos[i], open[0]); //Creamos el nuevo hijo
                    nHijos.DistanciaM(goals[0]); //Calcula f*= WalkCost + h*(distancia Manhattan)
                    nHijos.ProducedBy = (Locomotion.MoveDirection)i; //Indica el movimiento que ha hecho el
                                                                     //padre para tenerle. (Se ha cambiado CellInfo)
                    open.Add(nHijos); //Añadimos los hijos a la lista abierta
                }
            }
            open.RemoveAt(0); //Elimina al padre no meta
            open = open.OrderBy(idx => idx.dist).ToList(); //Linq Ordena la lista de manera ascendente. Con el ToList()
                                                           //lo trasformamos de IOrderedEnumerable a lista
        }
    }
    return null;
}
```

En el código de arriba se desarrolla el algoritmo A*. Con *var open* creamos la lista abierta del algoritmo. Inicializamos el primer nodo que no tiene padre, con las posiciones iniciales del character *n = new Nodo(star, null).* Y lo añadimos a la lista abierta. Se empieza el bucle para expandir a los hijos y encontrar la solución. La variable hijos almacena las coordenadas de los vecinos (hijos). Para expandir a todos los hijos posibles, primero se crea un nodo con las coordenadas de uno de los vecinos y su padre. Luego calcula la distancia Manhattan y el movimiento que ha hecho su padre con *ProducedBy* y por último se añade a la lista abierta. Para que se guardará de manera correcta los movimientos del padre, se han cambiado los valores del array en los que se guardan los WalkableNeighbours y así el índice coincide con los movimientos de los vecinos. Se elimina al primero de la lista y se ordena la lista de manera ascendente usando la librería Linq con el método *OrderBy()*. Con este algoritmo tenemos control sobre la función heurística que se utiliza (en este caso la distancia Manhattan).

3. DISCUSIÓN SOBRE LOS RESULTADOS OBTENIDOS

Al inicio de la práctica, el personaje como tenía implementado un controlador que generaba posiciones *ProducedBy* (Locomotion.Movedirection) de manera aleatoria, no encontraba la solución de una manera óptima. Sin embargo con el algoritmo A* podemos encontrarla de una manera óptima utilizando el menor número de pasos posibles. Cuando el agente se encuentra con un vecino que es un muro, gracias a *WalkableNeighbours* que nos da un valor null en esa dirección, ese nodo ya no se añade a la lista (if(hijos[i] != null)) ahorrando espacio en memoria. En el peor de los casos, la meta está en la otra punta del mapa, la complejidad será exponencial. Mientras que en el mejor de los casos, la meta está contigua al inicio y sin obstáculos, el algoritmo se ejecutará en tiempo lineal. Es muy importante que se acumulen los costes ya que varía el número de pasos y la ruta obtenidos.

El algoritmo A* es de los mejores métodos de búsqueda ya que se trata de un espacio no dinámico y accesible para el agente. Una búsqueda Online también hubiese valido y además hubiese reducido la complejidad.