Deusto Business School
Universidad de Deusto

Grado en Administración y Dirección de Empresas + Ingeniería Informática

Trabajo de Sistemas Inteligentes

NOTA: Entrega final de LightHouses AI Contest.

Autores:

Álvarez de Eulate, Javier

Atxa, Eneko

Pisón, Lander

Solozabal, Juan

Martes, a 8 de junio de 2021

Deusto
Universidad de Deusto

Índice

Tabla de Ilustraciones

1. Introducción

En el presente trabajo se ha tratado de codificar todo lo que se conceptualizó en la anterior entrega de LightHouses. Mientras que en la anterior se propuso una solución al problema de una manera meramente teórica, en este documento se ha intentado trasladar a código y poner en marcha el robot de acuerdo con nuestra propuesta.

Para ello, toda modificación ha sido implementada sobre el bot que se nos proporcionó en la asignatura como ejemplo. Es decir, el trabajo realizado en esta entrega ha consistido en la realización de un *fork* sobre el repositorio ejemplo y en la introducción del código pertinente sobre la clase *hateful.py*, teniendo como referencia la clase *randbot.py*. Toda ejecución ha sido realizada en el servicio en la nube Google colab.

2. Resumen de la entrega anterior

Hablar de que vamos a utilizar el betterManhattan, que es una modificación del clásico método Manhattan.

En la anterior entrega se concluyó que se trataría de implementar un *bot* que utilizara una versión mejorada de la distancia Manhattan para seleccionar el mejor faro a conquistar y el algoritmo A* para encontrar el camino óptimo hasta el mismo. El algoritmo A* tiene en cuenta la función heurística, la cual será la propia distancia Manhattan.

Recapitulando lo que es la distancia Manhattan ya explicada en la entrega anterior, ésta es muy utilizada en juegos que utilizan un tablero con casillas. La distancia Manhattan entre dos puntos se mide sumando el desplazamiento lateral que se debe hacer hasta el nuevo punto al desplazamiento vertical que se debe hacer hasta el nuevo punto. Por ejemplo, la distancia Manhattan entre dos puntos que se encuentran 4 casillas laterales y 5 verticales aparte, será 9.

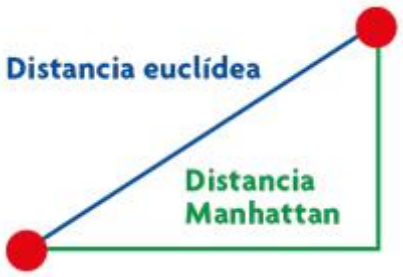

Ilustración 1: Distancia euclídea comparada con la distancia Manhattan

Aun así, se ha podido ver que la distancia Manhattan tampoco sería una heurística admisible. Utilizando la distancia Manhattan, las distancias medidas son incluso mayores que con la distancia euclídea. Por ejemplo, en el primer ejemplo expuesto, en el cual se mide la distancia a una casilla en diagonal, la distancia euclídea sería la raíz cuadrada de 2, y la distancia Manhattan 2. En muchas ocasiones, la distancia Manhattan es incluso mayor que la distancia euclídea, por lo que tampoco valdría.

Sin embargo, se ha podido encontrar la manera de obtener una heurística de distancias admisible. Para ello, se ha decidido hacer lo siguiente: se van a medir la distancia en el eje X y la distancia en el eje Y desde el punto inicial hasta el final, y se va a tomar el máximo de los dos. Así, se va a obtener el número de pasos mínimo que se necesita para llegar hasta el punto final. En el ejemplo de 4 casillas laterales y 5 verticales, la distancia medida sería 5, ya que 4 > 5. Así, se podría llegar al punto final en 5 pasos, tomando un paso vertical y 4 pasos diagonales.

Por lo tanto, la función heurística que se va a usar va a ser: $f(x) = max(x1-x0, y1-x0)$. Con esta heurística admisible, aseguraremos que A* encuentra el camino óptimo siempre. Dicho esto, cabe destacar que la complejidad temporal del algoritmo es $O(b^d)$ en el peor caso, y la complejidad espacial también es $O(b^d)$ en el peor caso. Podría darse la posibilidad de que estas características nos hicieran cambiar de opinión a la hora de implementar, pero de momento nos decantamos por A* por sus múltiples ventajas.

3. Código implementado

3.1 Cambios realizados en el método *play*

Dentro del método *play*, se han introducido algunos cambios en comparación con el método proporcionado en el *randbot.py*. En primer lugar, hemos prescindido de la opción de recargar el faro, ya que, tal y como comentamos en la anterior entrega, no recargaremos los faros que nos pertenezcan. El objetivo, en cambio, será el poseer el mayor número de faros posible, de ahí la línea de código en la que, en caso de que se verifique que no poseemos un faro, este será atacado.

A continuación, se llama al método *chooseLighthouse*, para lograr el faro más cercano desde nuestra posición y al que atacaremos próximamente. Se comprobará mediante el método *isAStarPossible* si efectivamente el faro objetivo se encuentra dentro de nuestro campo visual (no ha de olvidarse que solo se tiene información a tres casillas vista). En caso de que así sea, se aplicará el algoritmo A* para lograr el camino óptimo hacia el faro objetivo. Una vez obtenido el camino óptimo, realizará los movimientos pertinentes para llegar hasta al faro.

En caso de que el faro objetivo no se encuentre a una distancia de tres casillas, se llamará al método *getCloserToLightHouse* a fin de ir acortando las distancias, hasta que este se encuentre a una distancia suficiente como para aplicar el algoritmo A*.

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

import random, sys
import interface

class HatefulBot(interface.Bot):
    """Bot de los hateful four."""
    NAME = "HatefulBot"

    def play(self, state):
        """Jugar: llamado cada turno.
        Debe devolver una accin (jugada)."""
        cx, cy = state["position"]
        lighthouses = dict((tuple(lh["position"]), lh)
                           for lh in state["lighthouses"])
            allLh = []
        for lh in state["lighthouses"]:
            allLh.append(lh)

        # Si estamos en un faro...
        if (cx, cy) in lighthouses:
            # Si podemos hacer conexion, conectar con faro remoto vlido
            if lighthouses[(cx, cy)]["owner"] == self.player_num:
                possible_connections = []
                for dest in lighthouses:
                    # No conectar con sigo mismo
                    # No conectar si no tenemos la clave
                    # No conectar si ya existe la conexin
                    # No conectar si no controlamos el destino
                    # Nota: no comprobamos si la conexin se cruza.
                    if (dest != (cx, cy) and
                        lighthouses[dest]["have_key"] and
                        [cx, cy] not in lighthouses[dest]["connections"] and
                        lighthouses[dest]["owner"] == self.player_num):
                        possible_connections.append(dest)

                if possible_connections:
                    return self.connect(random.choice(possible_connections))

            #Si no somos duenyos, conquistar
            if lighthouses[(cx, cy)]["owner"] != self.player_num:
                return self.attack(energy)

        target = chooseLighthouse(allLh, cx, cy)
        move = []
        if isAStarPossible(allLh, cx, cy):
            path = aStar([cx, cy], target, state["view"])
            move = path[0]
        else:
            move = getCloserToLighthouse(target, cx, cy)
        # Mover aleatoriamente
        # moves = ((-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1))
        # Determinar movimientos vlidos
        # moves = [(x,y) for x,y in path if self.map[cy+y][cx+x]]

        return self.move(*move)

        #to continue...
```

Ilustración 2: Codificación del método *play*

3.2 Creación del método ***chooseLighthouses***

El objetivo de este método, cuyo código está representado en la Ilustración 3, consiste en la obtención del del faro más cercano a fin de conquistarlo. Para ello, al igual que cuando se aplique el algoritmo A*, se hará uso de la variante Manhattan para calcular las distancias a los faros. Realmente se podría calcular simplemente la distancia hacia los faros, pero, a fin de que la implementación sea consistente, también se ha utilizado esta técnica.

Así pues, se definen las variables *betterManhattan* a 9999 y *targetLh* como un *lighthouse* que a la postre servirá para referenciar el faro más cercano de acuerdo con la heurística.

En el bucle se calculará la heurística para cada uno de los faros (es decir, se calculará el máximo entre la distancia horizontal y la distancia vertical desde la posición del *bot*) y se actualizarán las variables *betterManhattan* y *targetlh* siempre que la distancia en referencia al faro en cuestión sea inferior a la previamente almacenada en la variable *betterManhattan*. Así pues, al finalizar el bucle, devolveremos el faro más cercano, con el que luego se aplicará el algoritmo A* para encontrar el camino más corto.

```
def chooseLighthouse(lighthouses, cx, cy):
        betterManhattan = 9999
        targetLh = lighthouses[0]
        for lh in lighthouses:
            if lh["owner"] != self.player_num:
                if betterManhattan != 0 and betterManhattan > max(abs(lh["position"][0]-cx), abs(lh["
                    position"][1]-cy)):
                    betterManhattan = max(abs(lh["position"][0]-cx), abs(lh["position"][1]-cy))
                    targetLh = lh
        return targetLh
```

Ilustración 3: Codificación del método chooseLighthouses

3.3 Creación del método *aStar*

El algoritmo de búsqueda A* permitirá al *bot* escoger el camino más óptimo apoyándose en una función heurística. Es decir, este algoritmo elige gracias a esta función qué rama del árbol extender con el objeto de llegar más rápido a la solución; para ayudarle en esa elección de rama se utiliza una función heurística.

La función heurística será *betterManhattan*, la cual tiene en cuenta la distancia Manhattan frente a la Euclídea y con unos ajustes que creemos que se aplicarán mejor al presente juego (por ello la hemos llamado *betterManhattan*).

```
def aStar(start, goal, grid):
        return [[1,0]]
```

Ilustración 4: Codificación del método aStar

3.4 Creación de *isAStarPossible*

Este método auxiliar es de utilidad para realizar la comprobación de si se puede aplicar el algoritmo A* o no. Por lo tanto, el resultado de este método será un booleano:

- Si hay un faro que está en el campo de visión (a 3 o menos casillas de distancia desde la posición del *bot*) devuelve *true*.
- En caso contrario, devuelve *false.*

```
def isAStarpossible(lighthouses, cx, cy):
        for lh in lighthouses:
            if(max(abs(lh["position"][0]-cx), abs(lh["position"][1]-cy)<=3)):
                return true
        return false
```

Ilustración 5: Codificación del método isAStarPossible

3.5 Creación del método *getCloserToLightHouse*

Por último, el objetivo de este método es dirigir el movimiento hacia el faro *target* que se pretende alcanzar, y más adelante, conquistar. Para ello se han elaborado una serie de condiciones *if* que realicen la comparación entre la posición actual del *bot* y la posición del faro.

En primer lugar, se realiza la comparación sobre el eje de abscisas, para luego comprobar el eje de ordenadas:

- Si la posición del *bot* en el eje de abscisas es menor que el del faro, o dicho de otra manera, se encuentra a la izquierda del faro:
  - Si la posición del *bot* en el eje de ordenadas es menor que el del faro (debajo): El *bot* se mueve a la derecha y hacia arriba [1,1]
  - Si la posición del *bot* en el eje de ordenadas es mayor que el del faro (encima): El *bot* se mueve a la derecha y hacia abajo [1,-1]
  - Si la posición del *bot* en el eje de ordenadas es igual que el del faro (misma altura): El *bot* se mueve a la derecha [1,0]
- Si la posición del *bot* en el eje de abscisas es mayor que el del faro, o dicho de otra manera, se encuentra a la derecha del faro:
  - Si la posición del *bot* en el eje de ordenadas es menor que el del faro (debajo): El *bot* se mueve a la izquierda y hacia arriba [-1,1]
  - Si la posición del *bot* en el eje de ordenadas es mayor que el del faro (encima): El *bot* se mueve a la izquierda y hacia abajo [-1,-1]
  - Si la posición del *bot* en el eje de ordenadas es igual que el del faro (misma altura): El *bot* se mueve a la izquierda [-1,0]
- Si la posición del *bot* en el eje de abscisas es igual que el del faro:
  - Si la posición del *bot* en el eje de ordenadas es menor que el del faro (debajo): El *bot* se mueve hacia arriba [0,1]
  - Si la posición del *bot* en el eje de ordenadas es mayor que el del faro (encima): El *bot* se mueve hacia abajo [0,-1]

```
1   def getCloserToLighthouse(target, cx, cy):
2       if cx < target[0]:
3           if cy < target[1]:
4               #return upright
5               return [1,1]
6           if cy > target[1]:
7               #return downright
8               return [1,-1]
9           if cy == target[1]:
10              #return right
11              return [1,0]
12      if cx > target[0]:
13          if cy < target[1]:
14              #return upleft
15              return [-1,1]
16          if cy > target[1]:
17              #return downleft
18              return [-1,-1]
19          if cy == target[1]:
20              #return left
21              return [-1,0]
22      if cx == target[0]:
23          if cy < target[1]:
24              #return up
25              return [0,1]
26          if cy > target[1]:
27              #return down
28              return [0,-1]
```

Ilustración 6: Codificación del método getCloserToLighthouse

4. Conclusión

- Hablar de la falta de tiempo. Que el resultado no ha sido el que nos hubiese gustado al inicio de la asignatura, además de que hay una gran diferencia en cuanto al tiempo dedicado a la primera entrega comparándolo con el tiempo dedicado la segunda.

5. Bibliografía

A, Tile costs and heuristic; How to approach*. (2014). Recuperado el 21 de abril de 2021, de Stack Exchange - Game Development: https://gamedev.stackexchange.com/questions/64392/a-tile-costs-and-heuristic-how-to-approach