Grado en Administración y Dirección de Empresas + Ingeniería Informática

Trabajo de Sistemas Inteligentes

NOTA: Entrega final de LightHouses AI Contest.

REPOSITORIO:
https://github.com/javieulate/lighthousesHateFul

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 dicha solución a código y poner en marcha el robot.

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 una copia del proyecto ejemplo[1] y en la introducción del código pertinente sobre la clase *hatefuBot.py*, teniendo como referencia la clase *randbot.py*.

Por otro lado, toda ejecución ha sido realizada en el servicio en la nube Google Colab, ya que, aunque se intentó instalar Anaconda y ejecutar el proyecto en local, se acabó desechando esta alternativa por la innumerable cantidad de dificultades que acarreaba.

En ese sentido, ya que también se han experimentado dificultades a la hora de funcionar con la herramienta de Google citada, se han adjuntado en el Readme del proyecto dos demos que certifican el correcto funcionamiento del *bot*. Una con el mapa que viene por defecto y otra con el mapa *island*. Al menos, sirve para comparar el desempeño del *bot* con "algo de inteligencia" frente al *bot* que se movía aleatoriamente.

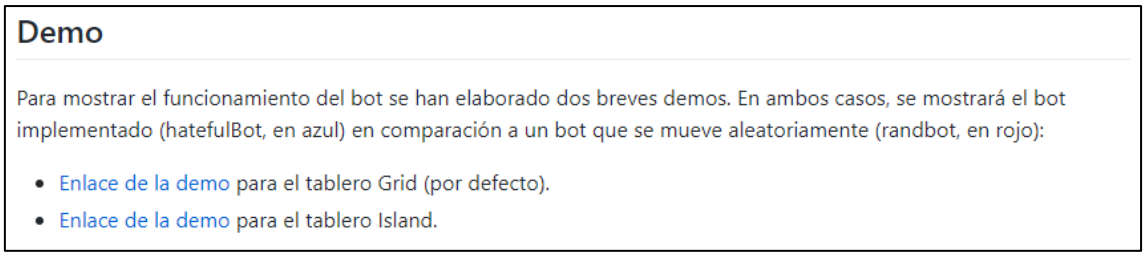

Ilustración 1: Sección Demo en Readme

[1] URL del repositorio del equipo: https://github.com/javieulate/lighthousesHateFul

2. Resumen de la entrega anterior

En la anterior entrega se concluyó que se trataría de implementar un *bot* que, en primer lugar, calculase en cada turno la distancia desde su posición a todos los faros. A continuación, escogería el faro más cercano como objetivo. Y, por último, a fin de llegar hasta este, aplicaría un algoritmo de búsqueda informado. Tras valorar varias alternativas, se decidió usar el algoritmo A*.

Sin embargo, tuvimos varias dificultades a la hora de elegir un heurístico que fuese admisible para aplicar al algoritmo. Se propusieron tanto la distancia euclídea como la distancia *Manhattan*, pero al estar el movimiento en diagonal permitido, se vio que estos no eran admisibles. Utilizando *Manhattan*, las distancias medidas eran incluso mayores que con la distancia euclídea.

En la Ilustración 2 el movimiento en diagonal, en el cual se mide la distancia a una casilla en diagonal, la distancia euclídea sería la raíz cuadrada de 5; mientras que la distancia *Manhattan* sería 3. En la mayoría de ocasiones la distancia *Manhattan* es mayor que la distancia euclídea, por lo que tampoco valdría.

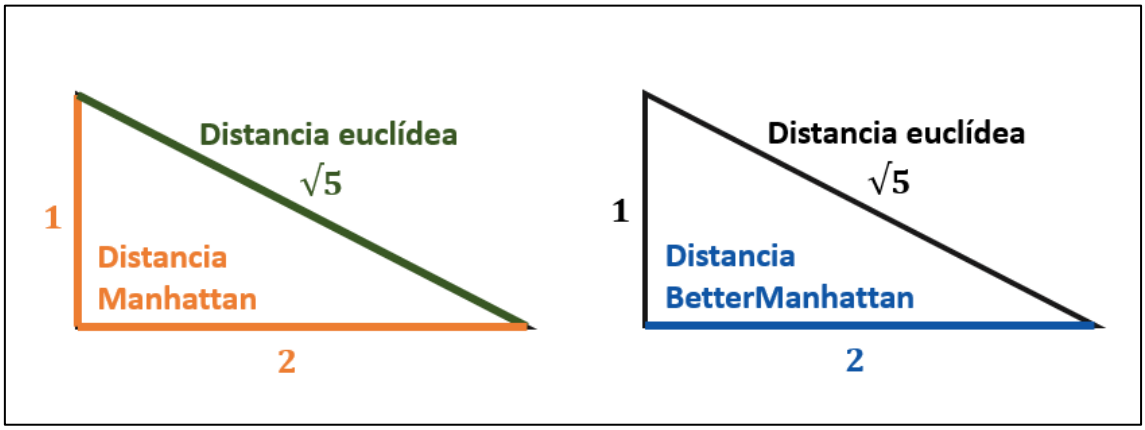

Ilustración 2: Distancia Euclídea vs Manhattan vs BetterManhattan

Así pues, decidimos "adaptar" la distancia *Manhattan* para que fuera admisible. Por ello, se decidió hacer lo siguiente: se iban a medir la distancia en el eje X y la distancia en el eje Y desde el punto inicial hasta el final, tomando como heurístico el máximo entre los dos (en el caso anterior, la distancia *betterManhattan* sería 2). Así, se iba a obtener el número de pasos mínimo que se necesitase para llegar hasta el punto final. Por lo tanto, la función heurística que se acordó utilizar iba a ser: *f(x) = max(x1-x0, y1-x0)*. Con esta heurística admisible, asegurábamos que A* fuese a encontrar el camino óptimo.

En resumen, se decidió dividir el problema en dos pasos: la elección del faro a atacar, por un lado; y, por otro lado, la ejecución del algoritmo A* para "dibujar" el camino hacia el faro.

3. Código implementado

Para la codificación de la solución planteada se han aplicado cambios en el método *play*, que es el que se encarga de orquestar el funcionamiento del *bot.* Desde dicho método se han llamado a otros métodos que se han programado específicamente para la elaboración de una tarea.

En ese sentido, se han creado métodos para la elección del faro más cercano (método *chooseLighthouses)*, para la ejecución del algoritmo A* (método aStar), para el movimiento del *bot* cuando no se encuentra a una distancia de tres casillas o menos del faro objetivo (método *getCloserToLighthouse*) e incluso para verificar si el faro se encuentra a una distancia de tres casillas o menos, y así aplicar el algoritmo A* o no (método *isAStarpossible*).

A continuación se detallan los métodos implementados.

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. El objetivo, en cambio, consistirá en poseer el mayor número de faros posible, de ahí las líneas de código 40 y 41 de la Ilustración 3, en las que, en caso de que se verifique que no poseemos un faro y se disponga de energía, el faro en cuestión será atacado.

A continuación, se llama al método *chooseLightHouse*, para identificar el faro más cercano desde nuestra posición y al que atacaremos próximamente. Este método devuelve dos coordenadas (x e y), razón por la que se recoge la solución en dos parámetros, xLh e yLh.

Una vez obtengamos las coordenadas de la posición del faro más cercano, estas serán enviadas, junto con las coordenadas de la posición del *bot*, al método *getCloserToLightHouse*, que será el encargado de devolver el siguiente movimiento a realizar por parte del *bot*.

Sin embargo, hay una última comprobación. Puede que el movimiento devuelto por dicho método no sea posible en el tablero. Por ejemplo, en el tablero *island*, hay varias casillas por las que no se puede pasar. De ese modo, se comprueba que el movimiento devuelto por el método se encuentra, de acuerdo con la disposición del mapa en ese momento determinado, dentro de los movimientos disponibles. En caso de que así sea, el método *play* devolverá el movimiento especificado por el método *getCloserToLightHouse*. En cambio, si el movimiento no puede llevarse a cabo porque no lo permite el tablero, el método elegirá un movimiento aleatorio dentro de todos los movimientos disponibles. De esta forma, la probabilidad de que el *bot* se quede atascado en el tablero será menor, ya que sin esta comprobación la probabilidad de que suceda es muy alta.

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

import random, sys, math
import interface

class HatefulBot(interface.Bot):
    """Bot que juega utilizando la distancia betterManhattan."""
    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"])

        # Si estamos en un faro...
        if (cx, cy) in lighthouses:
            # Probabilidad 60%: 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
            energy = state["energy"]
            if lighthouses[(cx, cy)]["owner"] != self.player_num and energy != 0:
                return self.attack(energy)

        allLh = []
        move = [0,0]
        for lh in state["lighthouses"]:
            allLh.append(lh)

        xLh, yLh = chooseLighthouse(self, allLh, cx, cy)
        move = getCloserToLighthouse(xLh, yLh, cx, cy)
        moves = ((-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1))
        # Determinar movimientos validos
        moves = [(x,y) for x,y in moves if self.map[cy+y][cx+x]]
        # Comprobacion para que no se quede atascado el bot
        if tuple(move) not in moves:
            move = random.choice(moves)

        return self.move(*move)

if __name__ == "__main__":
    iface = interface.Interface(HatefulBot)
    iface.run()
```

Ilustración 3: Codificación del método *play*

3.2 Creación del método *chooseLighthouses*

El objetivo de este método, cuyo código está representado en la Ilustración 4, 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 distancia *Manhattan* modificada para calcular las distancias a los faros. Se podría simplemente calcular la distancia hacia los faros, pero, a fin de que la implementación sea consistente, también se ha utilizado esta técnica.

Así pues, se inicializa la variable *betterManhattan* a 9999 y se define *targetLh* como un *lighthouse*, ya 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, e devolverá la posición del faro más cercano.

```
1  def chooseLighthouse(self, lighthouses, cx, cy):
2      betterManhattan = 9999
3      targetLh = lighthouses[0]
4      for lh in lighthouses:
5          xLh, yLh = lh["position"]
6          if lh["owner"] != self.player_num:
7              if max(abs(xLh-cx), abs(yLh-cy)) != 0 and betterManhattan > max(abs(xLh-cx), abs(yLh-
                   cy)):
8                  betterManhattan = max(abs(xLh-cx), abs(yLh-cy))
9                  targetLh = lh
10     return targetLh["position"]
```

Ilustración 4: Codificación del método chooseLighthouses

3.3 Creación del método *aStar*

El algoritmo de búsqueda A* permitirá al *bot* escoger el camino 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*). Sin embargo, tal y como se puede observar en la Ilustración 5, no se ha codificado el algoritmo A* en el presente proyecto debido a que no se ha contado con el tiempo suficiente. Esto será comentado con mayor profundidad en el apartado de Conclusión.

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

Ilustración 5: 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.*

Dado que no se ha podido desarrollar la función para A*, tanto esta función como la anterior quedarían pendientes para futuros desarrollos.

```
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 6: 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.

En un inicio, el objetivo de este método consistía en hacer mover al *bot* hasta que se encontrara a una distancia de tres casillas del faro objetivo (pues no ha de olvidarse que no se cuenta con la situación de todo el tablero) y, una vez en ese rango de distancia, aplicar el algoritmo A*.

Sin embargo, debido a que no se ha contado con tiempo suficiente para codificar el método de aStar, finalmente se ha implementado toda la funcionalidad referente al movimiento en el método *getCloserToLightHouse*. Para ello se han elaborado una serie de condiciones *if* que valoran 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]

Como ya se ha mencionado anteriormente, en caso de que el movimiento establecido por este método no sea posible debido a las restricciones del tablero, en el método *play* se le indica al *bot* que realice un movimiento aleatorio, con la esperanza de que este movimiento consiga desatascar al *bot* y prosiga en la conquista de faros y en la realización de conexiones.

```
def getCloserToLighthouse(xLh, yLh, cx, cy):
    if cx < xLh:
        if cy < yLh:
            #return upright
            return [1,1]
        if cy > yLh:
            #return downright
            return [1,-1]
        if cy == yLh:
            #return right
            return [1,0]
    if cx > xLh:
        if cy < yLh:
            #return upleft
            return [-1,1]
        if cy > yLh:
            #return downleft
            return [-1,-1]
        if cy == yLh:
            #return left
            return [-1,0]
    if cx == xLh:
        if cy < yLh:
            #return up
            return [0,1]
        if cy > yLh:
            #return down
            return [0,-1]
```

Ilustración 7: Codificación del método getCloserToLighthouse

4. Conclusión

Tras haber explicitado los métodos y cambios a nivel de código, tratados en los apartados anteriores, se deben mencionar una serie de conclusiones acerca del trabajo realizado. Primeramente, cabe destacar que el tiempo disponible para realizar esta entrega no ha sido el deseado, principalmente, por razones académicas (exámenes, otros proyectos, etc.) y organizacionales (tuvimos complicaciones en la comprensión del trabajo, por ejemplo). Como consecuencia, el resultado final conseguido no es el que se había planificado al inicio de la asignatura, llegando a esperar una implementación de un robot con un algoritmo de búsquedas completamente codificado y funcional. Ejemplo de ello es la no implementación del algoritmo de búsqueda A*, el cual se expuso en la anterior entrega que era el mejor método de *pathfinding* para nuestro robot.

Adicionalmente, se debe de realizar una especial mención a la gran variedad de problemas afrontados a la hora de implantar el proyecto en la plataforma Google Colab. Tal y como se ha explicitado en el apartado introductorio, el grupo de trabajo se ha visto obligado a utilizar esta herramienta online, dado que la instalación de Anaconda en nuestros equipos resultaba problemática. No obstante, Google Colab no ha resultado ser herramienta fácil de usar, puesto que a la hora de probar nuestros avances en el proyecto se resaltaban fallos sin ningún tipo de feedback, llegando a retrasar el avance del proyecto a causa de no saber qué aspecto corregir. Obviamente, el grupo de trabajo sabía que detrás de esos fallos existía un fallo de código propio que había que cambiar, pero acostumbrados a la retroalimentación de los entornos de desarrollo que habitualmente utilizamos (Pycharm, Eclipse, etc.) sí que hemos echado en falta esta característica.

Más concretamente, los métodos creados que más fallos han ocasionado son los referentes a *chooseLightHouses* y *getCloserToLighthouse*. Debido a que no se ha dispuesto de tiempo suficiente de implementar A*, el grupo de trabajo ha invertido todos sus esfuerzos en conseguir que la implementación de estos dos métodos funcionase, a fin de aportar parte de la funcionalidad ideada en la entrega anterior.

En conclusión, si bien es cierto que ha quedado cierto sabor amargo por no haber podido finalizar el robot con toda la funcionalidad planificada en la primera entrega, sí que existe un grado alto de satisfacción por haber conseguido modificar el comportamiento del robot, parcialmente al menos, mejorándolo sustancialmente en comparación al robot primigenio, tal y como se puede observar en las demos incluidas del proyecto. En el caso de haber podido invertir la misma cantidad de tiempo que a la primera entrega, el grupo de trabajo está seguro de que hubiera conseguido un resultado notablemente mejor.

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