Agentes Deliberativos Basados en Objetivos (Búsqueda Heurística):

Autores: Ismael Sagredo Olivenza

Enunciado

Vamos a ir un paso más con nuestro querido agente del Battle City. Hasta ahora, vuestro agente no disponía de suficiente información como para poder crear un plan a largo plazo. Esto cambia en la versión del del entorno, donde este nos envía en cada update a parte de la percepción (extendida con nuevos campos) el map actual del entorno.

El nuevo entorno dispone de los siguientes campos:

NEIGHBORHOOD_UP = 0, NEIGHBORHOOD_DOWN = 1, NEIGHBORHOOD_RIGHT = 2, NEIGHBORHOOD_LEFT = 3,NEIGHBORHOOD_DIST_UP = 4, NEIGHBORHOOD_DIST_DOWN = 5, NEIGHBORHOOD_DIST_RIGHT = 6, NEIGHBORHOOD_DIST_LEFT = 7, PLAYER_X = 8, PLAYER_Y = 9, COMMAND_CENTER_X = 10, COMMAND_CENTER_Y = 11, AGENT_X = 12, AGENT_Y = 13, CAN_FIRE = 14, HEALTH = 15, LIFE_X = 16, LIFE_Y = 17, TIME = 18 Antes de time se han añadido LIFE_X = 16, LIFE_Y = 17 que es la ubicación de un power up de vida que aumenta en 1 punto la vida de quien la coja.

En el mapa (que se muestra en miniatura en la imagen del juego) pueden aparecer lo sisguientes campos:

NOTHING = 0, UNBREAKABLE = 1, BRICK = 2, COMMAND_CENTER = 3, SEMI_BREKABLE = 8, SEMI_UNBREKABLE = 9

Los elementos: PLAYER = 4, SHELL = 5, OTHER_AGENT = 6, LIFE = 7 se consideran dinámicos y se mostraran en la percepción pero no en el mapa.

Los objetos que podemos ver nuevamente son los siguientes:

NOTHING = 0, UNBREAKABLE = 1, BRICK = 2, COMMAND_CENTER = 3, PLAYER = 4, SHELL = 5, OTHER_AGENT = 6 La vida no es visible por el agente (no aparece en lso 4 primeros campos de la percepcion), pero si que sabemos en todo momento donde esta en la percepción.

El resto del entorno se comporta igual salvo porque **hemos desplazado ligeramente los tiles para que los centros cuadren mejor y así facilitar seguir una ruta por el mapa**. Este desplazamiento es de 1 unidad en cada eje de forma que ahora cada celda o tile, **su centro de coordenadas está en la esquina inferir izquierda en vez de en el centro**, esto nos ayuda a determinar si un objeto está en la celda de forma fácil con las siguientes ecuaciones que ya están implementadas en la clase **BCProblem**

Hechas las presentaciones, y dado que la práctica es más compleja que al anterior

os he dado un esqueleto con TODOS para resolver. Si alguien quiere hacerla enteramente el desde el principio con su código, puede hacerlo.

Tanto si seguis la plantilla como si lo implementáis desde cero, lo que es común es lo siguiente:

- El agente ejecuta una máquina de estados y la máquina de estados debe tener al menos un estado **ExecutePlan** que ejecuta el plan que le proporciona A. *Podéis implementar A* o usar AIMA (implementación de algoritmos de búsqueda) pero la solución base que se os da es pensando en que implementáis A* vosotros.
- Hay una clase GoalMonitor que está comprobando la validez del plan y que fuerza a replanificar cuando la estrategia del agente así lo considere.
- La estrategia podría programarse mediante una función de utilidad y que sea el propio A* con su heurística, la que determine cuál de las posibles metas es mejor. Pero esto complica la heurística, así que no es necesario hacerlo para la práctica y la estrategia puede ser algo que definái vosotros por código. El monitor debe determinar cuándo se necesite replanificar que meta es la más útil en ese momento y A* asumiría esa como única posible meta e intentaría encontrar un plan para conseguirla.
- La mayoría de los estados de vuestros agente reactivo siguen siendo válidos (salvo el estado GoToCommanCenter / explorar que deber ser sustituido por **ExecutePlan**). Estos eventos que sacan al agente de **ExecutePlan** son necesario porque el agente debe reaccionar a por ejemplo, que alguien le dispare.

Si seguís la plantilla estos son los ficheros que debéis modificar (donde hay TODOS)

AStar.py:

```
def GetPlan(self):
    findGoal = False
    #TODO implementar el algoritmo A*
    #cosas a tener en cuenta:
    #Si el número de sucesores es 0 es que el algoritmo no ha encontrado una solución,
    #Hay que invertir el path para darlo en el orden correcto al devolverlo (path[::-1]
    #GetSucesorInOpen(sucesor) nos devolverá None si no lo encuentra, si lo encuentra
    #es que ese sucesor ya está en la frontera de exploración, DEBEMOS MIRAR SI EL NUEV
    #SI esto es asi, hay que cambiarle el padre y setearle el nuevo coste.
    self.open.clear()
    self.precessed.clear()
    self.open.append(self.problem.Initial())
    path = []
    #mientras no encontremos la meta y haya elementos en open....
    #TODO implementar el bucle de búsqueda del algoritmo A*
    return path
```

```
    #nos permite configurar un nodo (node) con el padre y la nueva G
    def _ConfigureNode(self, node, parent, newG):
        node.SetParent(parent)
        node.SetG(newG)
        #TODO Setearle la heuristica que está implementada en el problema. (si ya la tenía s

    #reconstruye el path desde la meta encontrada.
    def ReconstructPath(self, goal):
        path = []
        #TODO: devuelve el path invertido desde la meta hasta que el padre sea None.
        return path
```

BCNode.py:

```
    def IsEqual(self,node):
        #TODO: dos nodos son iguales cuando sus coordenadas x e y son iguales.
        return False
```

BCProblem.py:

```
    #Calcula la heuristica del nodo en base al problema planteado (Se necesita reimplementar
    def Heuristic(self, node):
        #TODO: heurística del nodo
        print("Aqui falta ncosas por hacer :) ")
        return 0

    #Genera la lista de sucesores del nodo (Se necesita reimplementar)
    def GetSucessors(self, node):
        successors = []
        #TODO: sucesores de un nodo dado
        print("Aqui falta ncosas por hacer :) ")
        return successors

    #se utiliza para calcular el coste de cada elemento del mapa
    @staticmethod
    def GetCost(value):
        #TODO: debes darle un coste a cada tipo de casilla del mapa.
        return sys.maxsize
```

GoalMonitor.py:

```
    #determina si necesitamos replanificar
    def NeedReplaning(self, perception, map, agent):
        if self.recalculate:
            self.lastTime = perception[AgentConsts.TIME]
            return True
        #TODO definir la estrategia de cuando queremos recalcular
        #puede ser , por ejemplo cada cierto tiempo o cuanod tenemos poca vida.
        return False
```

```
    #selecciona la meta mas adecuada al estado actual
    def SelectGoal(self, perception, map, agent):
        #TODO definir la estrategia del cambio de meta
        print("TODO aqui faltan cosas :)")
        return self.goals[random.randint(0,len(self.goals))]
```

GoalOrientedAgent.py:

```
    #método interno que encapsula la creació nde un plan
    def _CreatePlan(self,perception,map):
        #currentGoal = self.problem.GetGoal()
        if self.goalMonitor != None:
            #TODO creamos un plan, pasos:
            #-con gualMonito, seleccionamos la meta actual (Que será la mas propicia => def
            #-le damos el modo inicial _CreateInitialNode
            #-establecer la meta actual al problema para que A* sepa cual es.
            #-Calcular el plan usando A*
            print("TODO aqui faltan cosas :)")
        return self.aStar.GetPlan()

    #no podemos iniciarlo en el start porque no conocemos el mapa ni las posiciones de los
    def InitAgent(self,perception,map):
        #creamos el problema
        #TODO inicializamos:
        # - creamos el problema con BCProblem
        # - inicializamos el mapa problem.InitMap
        # - inicializamos A*
        # - creamos un plan inicial
        print("TODO aqui faltan cosas :)")
        goal1CommanCenter = None
        goal2Life = self._CreateLifeGoal(perception)
        goal3Player = self._CreatePlayerGoal(perception)
        self.goalMonitor = GoalMonitor(self.problem,[goal1CommanCenter,goal2Life,goal3Player
```

Test: * Debe superar los dos test de la práctica anterior. Debe ir derecho a por los objetivos (puede haber algún titubeo por redondeo) * Debe ser mucho mas inteligente contra el jugador

Mejoras: * Limpiar el path para que solo tenga en cuenta los giros. Es decir, cuando nos devuelva el plan este plan lo podemos optimizar, dejando solo los nodos del path donde realmente cambiamos de dirección esto hace que sea más eficiente y que tenga menos errores de llegada a los centros de los nodos. * Cualquier otra cosa que se os ocurra para que el comportamiento del agente sea inmisericorde y despiadado :)