Buscando al coronel Kurtz

Fundamentos de la Inteligencia Artificial

ICAI 2023-2024

IMAT 2ºB

Miguel Ángel Vallejo de Bergia

Introducción

El proyecto final de esta asignatura ha consistido en implementar una especie de "juego" en la que el usuario toma el control de un agente[1], el coronel Kurtz, y este debe buscar al capitán Willard en una serie de habitaciones.

La dificultad de este juego reside en que en dichas habitaciones puede haber trampas y un monstruo que el coronel debe evitar para no morir.

Las posiciones de dichas trampas y el monstruo deben ser inferidas por el agente, este no sabe su posición exacta, si no que a medida que avanza por las casillas va percibiendo perceptos que le ayudan a hacerse una idea de donde pueden estar.

El coronel además porta un arma capaz de dejar al monstruo fuera de combate

La única forma de que tiene el coronel de ganar es salir por la habitación de tipo salida 'S' habiendo encontrado previamente al capitán Willard.

Se han tomado 2 enfoques: el enfoque KB y el enfoque Bayesiano. Ambos explicados a continuación.

Implementación de funcionalidades básicas y aspecto visual

En cada uno de los 2 enfoques, se ha implementado lo básico de forma parecida.

En primer lugar, se han creado 2 clases: Habitacion y Capitan_Willard.

La clase Habitacion guarda información de los perceptos de dicha habitación, guarda si el capitán está o no, guarda su posición en la matriz y guarda los elementos de la misma (trampas, monstruo, salida…). Se podría ver como el 'estado' y las variables que almacena como las 'variables de estado'

La clase Capitan_Willard guarda, para cada enfoque, los perceptos percibidos y las inferencias hechas a partir de los mismos. Cabe destacar que no son iguales en el enfoque Bayesiano y en el enfoque KB.

El mapa de juego se ha implementado de forma sencilla con una matriz cuadrada. Para visualizarlo se dispone de una función visualizar_entorno(matriz) que muestra el mapa

1. He decidido no implementar el algoritmo de búsqueda

de una forma agradable a la vista, indicando además las posiciones de cada casilla.

Los distintos elementos del juego (trampas, salida, el coronel) se han ido añadiendo a la matriz en posiciones aleatorias, siguiendo unas ciertas reglas (por ejemplo, en el enfoque KB no puede haber más de una trampa en una casilla).

La función codificar_perceptos codifica los preceptos de todas las casillas en función de los elementos que hay en las mismas.

Se dispone además de una función obtener_adyacentes, que devuelve la posición de las habitaciones adyacentes en la matriz a una habitación dada.

Por último, se dispone de una función jugar que implementa el mecanismo de juego, La idea de la función es parecida para los dos enfoques.

Funcionalidades del Agente (comunes)

Las funciones moverse y salir son comunes a ambos enfoques, moverse consiste en alterar la posición del agente en el entorno siempre que sea posible (que no haya una pared) y salir consiste en salir del entorno siempre que se esté en una posición de salida.

Funcionalidades del Agente KB

El agente KB presenta una función detonar, que le permite detonar una granada que mata al monstruo si este está en una celda adyacente al mismo.

Motor de inferencia

Esta es la funcionalidad más interesante del agente KB. El motor de inferencia permite al agente inferir posiciones de los elementos a partir del percepto actual y perceptos anteriores.

En este caso se ha implementado mediante una serie de condicionales y bucles. He decidido dividirlo en dos partes fundamentales.

La primera consiste en una serie de condicionales que dan avisos en función del percepto actual. Y hacer algunas inferencias de donde pueden estar los elementos.

La segunda en un bucle que va considerando todos los perceptos(actual y anteriores) y sus casillas. Si tenemos 2 perceptos iguales 2 celdas separadas por una celda, muy probablemente en elemento esté en esa celda que las separa. Por último, si sentimos el percepto de la salida en una celda y en otra celda contigua a la misma entonces la salida está en una de las dos celdas.

Las celdas que visita el agente quedan calificadas como celdas seguras y las celdas adyacentes a una celda en la que no se percibe nada que pueda hacer sospechar también quedan calificadas como seguras.

Este motor permite inferir correctamente, en la mayoría de los casos, las posiciones exactas de todos los distintos elementos del entorno, aunque a veces esto no sea necesario para ganar.

Funcionalidades del Agente Bayesiano

El agente Bayesiano presenta una función disparar_dardo, que dada una dirección dispara un dardo a la casilla adyacente que está en dicha dirección. Si el monstruo esta en esa casilla, el monstruo muere, si no continua con vida.

Mapas de probabilidad

Es la funcionalidad en la que se basa el agente Bayesiano. Para cada movimiento, se muestran unos mapas de probabilidad (Heatmaps) que indican para cada casilla la probabilidad de que haya un elemento.

En este caso se han creado 3 tipos de 'celdas seguras': libres de trampas, libres de monstruos y libres de salida. Estas celdas son aquellas en las que el agente sabe con total seguridad que no hay una trampa, un monstruo o una salida, bien por que no ha percibido ningún estímulo proveniente de las celdas adyacentes, en cuyo caso pasarían a ser seguras, o bien por que ha estado en dicha celda y sigue vivo.

Para los mapas de probabilidad se han distinguido 2 casos.

El primero y trivial es el caso en el que se percibe un estímulo. Se consideran las celdas adyacentes no seguras y se calcula la probabilidad de que haya algo en una celda adyacente no segura como 1 entre el numero de celdas adyacentes no seguras.

```python
# Caso 1: El capitán percibe estímulo
def modificar_mapa1(self,mapa:list,tipo:str):
    adyacentes = f.obtener_adyacentes(self.posicion[0],self.posicion[1],n)
    adyacentes_inseguras = []
    for adyacente in adyacentes:
        if tipo == 'tr':
            if adyacente not in self.seguras_trampas:
                adyacentes_inseguras.append(adyacente)
        elif tipo == 'm':
            if adyacente not in self.seguras_monstruo:
                adyacentes_inseguras.append(adyacente)
        elif tipo == 's':
            if adyacente not in self.seguras_salida:
                adyacentes_inseguras.append(adyacente)
    num_adyacentes_inseguras = len(adyacentes_inseguras)
    try:
        probabilidad = 1/num_adyacentes_inseguras
    except ZeroDivisionError:
        probabilidad = None
    for adyacente_insegura in adyacentes_inseguras:
        mapa[adyacente_insegura[0]][adyacente_insegura[1]] = probabilidad
    mapa[self.posicion[0]][self.posicion[1]] = 'CW'

    return mapa

if percepto_actual[0] == 1 or percepto_actual[1] == 1 or percepto_actual[2] == 1: # Trampas
    for i in range(3):
        if percepto_actual[i] == 1:
            modificados[i] = 1
            mapa_trampas = modificar_mapa1(self,mapa_trampas,'tr')
        mapas_mostrar['mapa_trampas'] = mapa_trampas

if percepto_actual[3] == 1 and modificados[3] == 0: # Monstruo
    modificados[3] = 1
    mapa_monstruo = modificar_mapa1(self,mapa_monstruo,'m')
    mapas_mostrar['mapa_monstruo'] = mapa_monstruo

if percepto_actual[4] == 1: # Salida
    modificados[4] = 1
    mapa_salida = modificar_mapa1(self,mapa_salida,'s')
    mapas_mostrar['mapa_salida'] = mapa_salida
```

(caso 1)

En un segundo caso se considera la posibilidad de que el agente no perciba estímulo, por lo que se puede inferir que el elemento no está en las celdas adyacentes y por tanto solo puede estar en las no adyacentes. Para este caso se ha empleado el teorema de bayes. (posterior = prior*likelyhood / constant) . Se ha calculado la probabilidad de elemento en cualquier casilla dado que no se ha percibido estímulo en una casilla dada.

En este caso se ha considerado:

n = 6

prior = 1/(n*n) – número de casillas seguras

el prior es la probabilidad de elemento sin condicionar, es la probabilidad de que el elemento (1) esté en una de las casillas posibles (n*n) – número de casillas seguras

likelihood = 1 o 0

La probabilidad de estimulo en una casilla dada una casilla condicionado a que en dicha casilla hay elemento es 1 si la casilla es adyacente y 0 si no lo es.

constant = probabilidad de no percibir estimulo en posición actual =

= 1 – probabilidad de percibir estimulo en posición actual =

= número casillas posibles adyacentes con estimulo / casillas_ totales posibles que pueden presentar estimulo

Donde,

numero casillas posibles adyacentes con estimulo = numero de casillas adyacentes + 1

casillas totales posibles que pueden presentar estimulo = n*n

```python
# Caso 2: El capitán NO percibe estímulo
N = n*n
def modificar_mapa2(self,mapa:list,tipo:str):
    adyacentes = f.obtener_adyacentes(self.posicion[0],self.posicion[1],n) # En este caso todas las adyacentes son seguras
    # Considerando que las casillas con elemento también presentan el estímulo:
    if tipo == 'tr':
        seguras = self.seguras_trampas
    elif tipo == 'm':
        seguras = self.seguras_monstruo
    elif tipo == 's':
        seguras = self.seguras_salida
    numero_casillas_posibles_adyacentes_con_estimulo = len(adyacentes) +1 # casos favorables
    casillas_totales_posibles_que_pueden_presentar_estimulo = N  # casos totales
    prior = 1/(N-len(list(set(seguras)))) # Probabilidad de elemento en la posicion i j
    probabilidad_de_percibir_estimulo_en_posicion_actual = numero_casillas_posibles_adyacentes_con_estimulo/casillas_totales_posibles_que_pueden_presentar_es
    probabilidad_de_no_percibir_estimulo_en_posicion_actual =  1 - probabilidad_de_percibir_estimulo_en_posicion_actual
    posterior = (1*prior)/probabilidad_de_no_percibir_estimulo_en_posicion_actual # Posterior para las no adyacentes, probabilidad de elemento dada la ausen
    for i in range(n):
        for j in range(n):
            if (i,j) not in seguras:
                mapa[i][j] += posterior
    mapa[self.posicion[0]][self.posicion[1]] = 'CW'
    return mapa
for i in range(3): # trampas
    if percepto_actual[i] == 0 and modificados[i] == 0:
        mapa_trampas = modificar_mapa2(self,mapa_trampas,'tr')
    mapas_mostrar['mapa_trampas'] = mapa_trampas
if percepto_actual[3] == 0 and modificados[3] == 0: # monstruo
    mapa_monstruo = modificar_mapa2(self,mapa_monstruo,'m')
    mapas_mostrar['mapa_monstruo'] = mapa_monstruo
if percepto_actual[4] == 0 and modificados[4] == 0: # salida
    mapa_salida = modificar_mapa2(self,mapa_salida,'s')
    mapas_mostrar['mapa_salida'] = mapa_salida
c = 1
for mapa in mapas_mostrar.keys():
    if c == 1:
        print(f'Celdas sin trampa: {self.seguras_trampas}')
    elif c==2:
        print(f'Celdas sin monstruo: {self.seguras_monstruo}')
    else:
        print(f'Celdas sin salida: {self.seguras_salida}')
    mapa_name = mapa.split(('_'))[1]
    print(f'Heatmap {mapa_name}')
    f.visualizar_entorno(mapas_mostrar[mapa])
    c+=1
```

(caso 2)

Como se puede ver, se crean y muestran 3 mapas distintos para cada caso. Uno con las trampas, otro con la salida y otro con el monstruo.

Estos mapas permiten al jugador tener una idea de donde pueden estar los elementos.

Conclusiones del trabajo

Se han creado 2 agentes distintos con 2 enfoques distintos. En ambos casos el jugador determina cual es la mejor acción en base a los resultados inferidos bien por el motor de inferencia o bien por los mapas de probabilidad.

El enfoque KB permite inferir posiciones con mayor seguridad, el enfoque Bayesiano se basa en un modelo probabilístico.

El enfoque KB considera todos los perceptos anteriores para hacer inferencias, es decir usa su base de conocimientos. El enfoque Bayesiano se basa en las celdas seguras y en el percepto actual para determinar las probabilidades de los elementos.