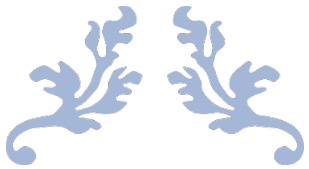

MEMORIA DE PRÁCTICAS DE SISTEMAS INTELIGENTES 2023/2024

Grupo A

ALEJANDRO, SANTANDER
ALEJANDRO RAMOS MORENO, SANTANDER ALBERTO IGLESIAS LÓPEZ
alejandro.ramos4@alu.uclm.es | santanderalberto.iglesias@alu.uclm.es

Índice

No se encontraron entradas de tabla de contenido.

No se encuentran elementos de tabla de ilustraciones.

1. Práctica 1: Búsqueda en el espacio de estados.

1.1 Introducción

Para esta práctica, utilizaremos distintos algoritmos de búsqueda en el espacio de estados para resolver el problema planteado en esta práctica. Dispondremos para ello de un 'grid' que actuará como un grafo, que será el escenario de un entorno con distintos estados en cada celda: bloqueado, libre, con personas atrapadas o peligro.

La idea principal en la resolución del problema consta de rescatar a las personas atrapadas mediante el camino generado por los distintos algoritmos que vamos a implementar. Los estados bloqueados serán innacesibles y los denotados con peligro tendrán una penalización aplicada al coste.

Los algoritmos utilizados serán:

- Búsqueda no informada: Anchura y profundidad.
- Búsqueda informada: Primero-mejor y A-Estrella.

Estos algoritmos encontrarán la ruta óptima para el rescate de estas personas.

1.2 Representación del problema

1.2.1 Clase Problema.

Para la representar el escenario del problema, utilizaremos la clase Problema. Esta misma contiene las filas, columnas, bloqueados, punto de partida, peligros y atrapados.

```python
class Problema:
    def __init__(self, problema):
        with open(problema, 'r') as file:
            self.new_dictionary = json.load(file)

        self.nfilas = self.new_dictionary['city']['rows']
        self.ncols = self.new_dictionary['city']['columns']
        self.bloqueados = set(map(tuple, self.new_dictionary['city']['blocked']))
        self.partida = self.new_dictionary['departure']
        self.peligro = set(map(tuple, self.new_dictionary['dangers']))
        self.atrapados = list(map(tuple, self.new_dictionary['trapped']))

    #Retornar la lista de sucerores
    def generarSucesores(self, nodo):
        sucesores = []
        estado_actual = nodo.estado

        acciones_posibles = ["Arriba", "Derecha", "Abajo", "Izquierda"]

        for accion in acciones_posibles:
            nuevo_estado = estado_actual.acciones(accion)
            nueva_posicion = nuevo_estado.posicion

            if (
                0 <= nueva_posicion[0] < self.nfilas
                and 0 <= nueva_posicion[1] < self.ncols
                and nueva_posicion not in self.bloqueados
            ):
                #SI la posicion es peligro el costo suma 5 en lugar de 1
                if nueva_posicion not in self.peligro:
                    nuevo_costo = nodo.costo + 1
                else:
                    nuevo_costo = nodo.costo + 5

                nuevo_nodo = Nodo(
                    estado=nuevo_estado,
                    accion=Accion(accion, nuevo_costo),
                    padre=nodo,
                    costo=nuevo_costo,
                    altura=nodo.altura + 1,
                )

                sucesores.append(nuevo_nodo)

        return sucesores
```

1. **Método "__init__(self, problema)":** Es el método constructor de la clase Problema, recibe por parámetro "problema", un diccionario que inicializamos justo debajo.
   Self.nfilas: Contiene las filas del grid.
   Self.ncols: Contiene las columnas del grid.
   Self.bloqueados: Celdas con el estado bloqueado.
   Self.partida: Celda del punto de partida.
   Self.peligro: Celdas con danger, agregan coste adicional.
   Self.atrapados: Celdas con los atrapados (objetivos).

2. **Método "generarSucesores(self, nodo)":** En este método vamos a generar los nuevos sucesores y retornarlos.

```python
def generarSucesores(self, nodo):
    sucesores = []
    estado_actual = nodo.estado #instancia de la clase estado

    acciones_posibles = ["Arriba", "Derecha", "Abajo", "Izquierda"]

    for accion in acciones_posibles:
        nuevo_estado = estado_actual.acciones(accion) #llama al metodo acciones de estado
        nueva_posicion = nuevo_estado.posicion

        if (
            0 <= nueva_posicion[0] < self.nfilas
            and 0 <= nueva_posicion[1] < self.ncols
            and nueva_posicion not in self.bloqueados
        ):
            #SI la posicion es peligro el costo suma 5 en lugar de 1
            if nueva_posicion not in self.peligro:
                nuevo_costo = nodo.costo + 1
            else:
                nuevo_costo = nodo.costo + 5

            nuevo_nodo = Nodo(
                estado=nuevo_estado,
                accion=Accion(accion, nuevo_costo),
                padre=nodo,
                costo=nuevo_costo,
                altura=nodo.altura + 1,
            )

            sucesores.append(nuevo_nodo) #añade el nuevo nodo a la lista

    return sucesores
```

Primero inicializamos un array vacío "sucesores", y creamos una instancia de la clase Estado "estado_actual".

Las acciones posibles son las definidas en la clase Estado, que veremos más adelante. Mediante un for que va a recorrer las acciones posibles, llamamos al método acciones de Estado y el algoritmo intentará ir recorriendo las

posibilidades presentes. Con un if controlamos que no se salga de las filas ni de las columnas, así como de que no avance a ninguna posición bloqueada.

Una vez cumpla estas condiciones, si llega a un estado de peligro se le sumará un coste por valor de 5, mientras que si es una posición libre se le sumará un coste por valor de 1. En la nueva posición, configuraremos el nuevo nodo al que ha llegado el algoritmo, asignándole su nodo padre como el anterior, su nuevo coste y se le añadirá altura por valor de 1. Esto sucederá cada vez que nos movemos de celda.

Finalmente, añadiremos el nuevo nodo a la lista de sucesores y por último se retorna la lista.

1.2.2 Clase Estado.

En la clase Estado vamos a definir la posición como una tupla de valores referenciando a la celda de la matriz donde nos encontramos, así como definir las acciones y como se mueve el agente por el entorno.

```
class Estado:
    def __init__(self, posicion):
        self.posicion = tuple(posicion)

    #Acciones de movimiento del robot
    def acciones(self, accion):
        if accion=="Arriba":
            nueva_pos = ((self.posicion[0] - 1,self.posicion[1]))
        elif accion=="Abajo":
            nueva_pos = ((self.posicion[0] + 1,self.posicion[1]))
        elif accion=="Derecha":
            nueva_pos = ((self.posicion[0], self.posicion[1] + 1))
        elif accion=="Izquierda":
            nueva_pos = ((self.posicion[0], self.posicion[1] - 1))

        return Estado(nueva_pos)
```

1. **Método "__init__(self, posición)":** Este método es el constructor de la clase, definiremos la posición como una tupla de valores para incluirse en la matriz.

2. **Método "acciones(self, acción)":** Para el métodos acciones le pasaremos por parámetro "acción", variable que nos indicará el movimiento que quiere tomar el agente.

   - **Acción=="Arriba":** restaremos por valor de 1 al primer valor de la tupla, moviéndonos hacia la celda de arriba, en función de las filas.
   - **Acción=="Abajo":** sumaremos por valor de 1 al primer valor de la tupla, moviéndonos hacia la celda de abajo, en función de las filas.
   - **Acción=="Derecha":** sumaremos por valor de 1 al primer valor de la tupla, moviéndonos hacia la celda de la derecha, en función de las columnas.
   - **Acción=="Izquierda":** restaremos por valor de 1 al primer valor de la tupla, moviéndonos hacia la celda de la izquierda, en función de las columnas.

Finalmente, retornaremos la nueva posición.

1.2.3 Clase Nodo.

En la clase Nodo implementamos la estructura de datos por las que se rigen los nodos del grid.

```python
class Nodo:
    def __init__(self, estado, accion, padre, costo, altura):
        self.estado = estado
        self.accion = accion
        self.padre = padre
        self.costo = costo
        self.altura = altura

    def __lt__(self, otro_nodo):
        # Define la comparación basada en la heurística
        return self.costo < otro_nodo.costo
```

1. **Método "__init__(self,estado,acción,padre,costo,altura)":** En el constructor de la clase vamos a pasar por parámetros las variables estado, acción, padre, costo y altura, que son las características que nos interesa conocer de cada nodo, y dentro del constructor las recogemos.

2. **Método "__lt__(self,otro_nodo)":** En este método vamos a comparar cada nodo por su coste, pues la cola de prioridad debe ir ordenada por el nodo de menor coste posible.

1.2.4 Clase Accion.

En la clase Acción solo tendremos su método constructor, al que le pasaremos el movimiento a realizar y el coste.

```python
class Accion:
    def __init__(self, movimiento, costo):
        self.movimiento = movimiento
        self.costo = costo
```

1.3 Estrategias de búsqueda

1.3.1 Clase Búsqueda

En las estrategias de búsqueda hemos implementado una clase Busqueda que, de forma futura, heredarán las clases de los distintos algoritmos que vamos a utilizar. Contiene un algoritmo general y unos métodos que mediante polimorfismo serán distintos para cada algoritmo que implementemos.

```python
class Busqueda:
    def __init__(self, problema):
        self.problema = problema
        self.total_nodos_generados = 0
        self.total_nodos_expandidos = 0
        self.total_tiempo_ejecucion = 0
        self.total_solucion_length = 0
        self.total_solucion_costo = 0

    def insertar(self, nodo, abiertos, objetivo):
        pass

    def eliminar(self, abiertos):
        pass

    def estaCerrado(self, estado, cerrados):
        return estado in cerrados
```

1. **Método "__init__(self,problema):** En el método constructor de esta clase le pasaremos por parámetro el problema a resolver e inicializaremos las variables que caracterizarán el camino obtenido, como los nodos generados, expandidos, tiempo de ejecución, longitud de la solución y su coste.

2. **Métodos "insertar(self, nodo, abiertos, objetivo)" , "eliminar(self, abiertos)" y "inicializarAbiertos(self)":** Estos métodos dejaremos un pass debido a que los modificaremos en función del algoritmo que queramos usar cuando los implementemos.

3. **Método "estaCerrado(self, estado, cerrados)":** Al método estaCerrado le pasaremos por parámetro el estado, cerrados y nos verificará si está presente el nodo en la lista cerrados, devolviendo un booleano.

```python
def buscar(self, objetivo):

    inicio_tiempo = time.time()

    # Insertar en lista de nodos abierto el nodo inicial
    nodo_inicial = Nodo(
        estado=Estado(self.problema.partida),
        accion=None,
        padre=None,
        costo=0,
        altura=0,
    )

    # abiertos = [nodo_inicial]

    abiertos = self.inicializarAbiertos()
    self.insertar(nodo_inicial, abiertos, objetivo)
    cerrados = set()

    nodos_expandidos = 0
    nodos_generados = 1
```

4. **Método "buscar(self, objetivo)":** En el método buscar, implementaremos todo el algoritmo que debe seguir nuestro agente, aunque se utilizará de una manera u otra en función del tipo de búsqueda que queramos realizar.

   Primeramente, vamos a insertar en la lista de abiertos nuestro nodo inicial, con sus variables inicializadas en coste y altura 0, pues es nuestro punto de partida. Inicializaremos la lista de abiertos e insertaremos los abiertos y el objetivo. Para los nodos ya visitados haremos un conjunto al que denominamos cerrados, pues a diferencia de en árboles, en grafos no volveremos a visitar el mismo nodo.

   Inicializamos nuestras variables expandidos y generados en 0 y 1 respectivamente, en dichas variables iremos almacenando los datos correspondientes en función de todas las búsquedas que realice el agente.

```
# Mientras haya nodos en la lista de nodos abiertos
while abiertos:
    # Extraigo un nodo de la lista de abiertos
    nodo_actual = self.eliminar(abiertos)

    # Compruebo si es el final
    if nodo_actual.estado.posicion == objetivo.posicion:
        # Termino y recupero el camino
        camino = []
        nodo_final = nodo_actual
        while nodo_actual.padre:
            camino.insert(0, nodo_actual.accion.movimiento)
            nodo_actual = nodo_actual.padre

        tiempo_ejecucion = time.time() - inicio_tiempo

        # Imprimir resultados
        print("Generated nodes:", nodos_generados)
        print("Expanded nodes:", nodos_expandidos)
        print("Execution time:", tiempo_ejecucion)
        print("Solution lengt:", nodo_final.altura)
        print("Solution cost:", nodo_final.costo)
        print("Solution:", camino)
        print("")

        self.total_nodos_generados += nodos_generados
        self.total_nodos_expandidos += nodos_expandidos
        self.total_tiempo_ejecucion += tiempo_ejecucion
        self.total_solucion_length += nodo_final.altura
        self.total_solucion_costo += nodo_final.costo

        return camino
```

Utilizaremos un bucle While en el que estaremos en ejecución mientras tengamos nodos abiertos hasta que lleguemos a la solución final.

Cogeremos nuestra lista de abiertos, y sacaremos el siguiente mediante la función eliminar. El nodo escogido va a depender de la implementación de eliminar en cada algoritmo de búsqueda.

Si es el nodo final, termino y creo un array vacío donde almacenaré el camino y asigno nodo final en el que me encuentro en ese momento. Para anotar el camino, recorreré mediante un While(tenga padre) todos los nodos del camino que hemos escogido. Además mediante la función time.time() obtendremos el tiempo utilizado durante la búsqueda.

Finalmente, imprimiré por pantalla los nodos generados, expandidos, tiempo de ejecución, longitud de la solución, coste y el camino obtenido.

```
            # Compruebo si no esta cerrado
            if not self.estaCerrado(nodo_actual.estado.posicion, cerrados):
                # Lo añado a los cerrados para no volver a expandirlos
                cerrados.add(nodo_actual.estado.posicion)

                # Expando, es decir, saco los sucesores
                sucesores = self.problema.generarSucesores(nodo_actual)
                for sucesor in sucesores:
                    self.insertar(sucesor, abiertos, objetivo)
                    nodos_generados += 1

                nodos_expandidos += 1

        # print("No se encontró un camino = busqueda.buscar(estado_objetivo).")
        return None

    def inicializarAbiertos(self):
        # Devuelve la estructura de datos para la lista de abiertos
        pass
```

En caso de no ser el nodo objetivo, se comprueba mediante el método estaCerrado si el nodo actual es uno abierto y de ser así lo incluyo en los nodos cerrados para no volver a expandirlo.

Ahora genera los posibles sucesores del nodo actual, y mediante un for que recorre todas estas posibilidades buscará el camino que mejor le convenga en función de la implementación de insertar en cada algoritmo de búsqueda.

1.3.2 Clase BúsquedaAnchura

Para el algoritmo de la búsqueda en anchura, heredaremos la clase Búsqueda para obtener sus métodos y variables.

La búsqueda en anchura es un algoritmo de búsqueda utilizado en grafos y estructuras de árboles. La principal característica de este algoritmo es que explora todos los nodos vecinos de un nodo actual antes de moverse hacia los nodos más distantes. Se podría describir como "explorar horizontalmente" a través del grafo o árbol.

Comenzaremos desde el nodo inicial, como no es nodo final lo meterá en la lista de visitados (cerrados) como hemos visto en el algoritmo general. La peculiaridad de anchura respecto a profundidad, es que exploraremos los nodos vecinos en horizontal antes de adentrarnos en los sucesores de cada uno. El algoritmo continuará hasta encontrar el nodo objetivo.

```python
class BusquedaAnchura(Busqueda): #FIFO

    def insertar(self, nodo, abiertos, objetivo):
        abiertos.append(nodo)

    def eliminar(self, abiertos):
        return abiertos.pop(0)

    def inicializarAbiertos(self):
        return []
```

1. **Método "insertar(self, nodo, abiertos, objetivo)":** En este método insertaremos al final de la lista el nodo, pues funciona como una cola (First-In-First-Out)

2. **Método "eliminar(self, abiertos)":** Elimina el primer elemento de la lista abiertos y lo devuelve.

3. **Método "inicializarAbiertos(self):** Inicializa la lista de abiertos.

1.3.3 Clase BúsquedaProfundidad

En el algoritmo de búsqueda en profundidad, heredaremos la clase Búsqueda.

La búsqueda en profundidad consiste en explorar todos los sucesores de cada nodo antes de retroceder hasta el siguiente. A diferencia de la búsqueda en anchura, busca explorar verticalmente a través de un árbol o grafo. En vez de utilizar una cola, utilizaremos una pila para seguir la política Last-In-First-Out para lograr esto.

```python
class BusquedaProfundidad(Busqueda): #LIFO

    def insertar(self, nodo, abiertos, objetivo):
        abiertos.insert(0, nodo)

    def eliminar(self, abiertos):
        return abiertos.pop(0)

    def inicializarAbiertos(self):
        return []
```

1. **Método "insertar(self, nodo, abiertos, objetivo)":** En este método insertaremos el nodo al principio de la lista, pues funciona como una pila (Last-In-First-Out)

2. **Método "eliminar(self, abiertos)":** Elimina el primer elemento de la lista abiertos y lo devuelve.

3. **Método "inicializarAbiertos(self):** Inicializa la lista de abiertos.

1.3.4 Clase BúsquedaProfundidadLimitada

Para la búsqueda en pronfudida limitada, heredaremos de la clase BusquedaProfundidad.

Esta búsqueda es una variante de la búsqueda en profundidad, donde establecemos un límite máximo en la profundidad hasta la cual el algoritmo puede explorar, esto se suele realizar para evitar ciclos infinitos (en árboles) o para controlar la cantidad de recursos que utiliza el algoritmo.

```
class BusquedaProfundidadLimitada(BusquedaProfundidad):

    def __init__(self, problema, limite=None):
        super().__init__(problema)
        self.limite = limite

    def insertar(self, nodo, abiertos, objetivo):
        if nodo.altura <= self.limite:
            abiertos.insert(0, nodo)
```

1. **Método "__init__(self, problema, limite=None)":** En el constructor de la clase vamos a pasarle el problema y un limite, establecido a None para cuando herede la Interativa no tener conflictos. El limite que le pasemos será hasta que altura bajará el agente en el grafo.

2. **Método "insertar(self, nodo, abiertos, objetivo)":** Se añadirán en cola los nodos siempre que respeten el limite impuesto.

1.3.5 Clase BúsquedaProfundidadIterativa

En la búsqueda en profundidad iterativa implementaremos un bucle para ir aumentando el limite de altura en cada iteración hasta llegar a una profundidad máxima.

Principalmente consiste una variante de la búsqueda en profundidad limitada, pero aumentando con cada iteración su límite de altura hasta una profundidad máxima establecida.

```python
class BusquedaProfundidadIterativa(BusquedaProfundidadLimitada):

    def buscar(self, objetivo):
        self.limite = 1
        while True:
            camino = super().buscar(objetivo)
            if camino:
                return camino
            else:
                self.limite += 1
```

1. **Método "buscar(self,objetivo)":** Sobreescribiremos el método buscar para establecer primero el limite en uno, para ir aumentándolo por valor de 1 en cada iteración hasta llegar al objetivo.

1.3.6 Clase PrimeroMejor

La búsqueda de primero mejor es un algoritmo de búsqueda informada que utiliza una función de evaluación para determinar qué nodo explorar primero en un grafo o árbol. A diferencia de la búsqueda en amplitud o en profundidad, la búsqueda primero mejor utiliza información heurística para tomar decisiones sobre qué camino seguir.

Principalmente buscaremos de forma voraz el siguiente nodo con menor heurística, y nos regiremos por una regla heurística que estima la cercanía al nodo objetivo.

```python
class PrimeroMejor(Busqueda):
    def insertar(self, nodo, abiertos, objetivo):
        # Utilizando PriorityQueue solo para PrimeroMejor
        abiertos.put((self.heuristica(nodo, objetivo), nodo))

    def eliminar(self, abiertos):
        # Eliminar y devolver el primer elemento de la lista (el de menor heurística)
        return abiertos.get()[1]

    def heuristica(self, nodo, objetivo):
        # Implementación de la heurística (distancia de Manhattan)
        return abs(nodo.estado.posicion[0] - objetivo.posicion[0]) + abs(nodo.estado.posicion[1] - objetivo.posicion[1])

    def inicializarAbiertos(self):
        return PriorityQueue()
```

1. **Método "insertar(self, nodo, abiertos, objetivo)":** Utilizaremos una cola de prioridad en función de la heurística que escojamos

2. **Método "eliminar(self, abiertos)":** Eliminamos y devolvemos el primer elemento de la lista, que va a tener la menor heurística.

3. **Método "heurística(self, nodo, objetivo)":** Para la heurística hemos decidido escoger la "distancia de Manhattan", que une dos puntos en un grid y su

distancia se determina por la suma de las diferencias absolutas de sus coordenadas en cada dimensión.

4. **Método inicializarAbiertos(self):** Devuelve la cola de prioridad.

1.3.7 Clase Aestrella

El algoritmo de búsqueda A estrella es otro algoritmo de búsqueda informada que utiliza una combinación de la función de costo real desde el inicio hasta el nodo actual, y la heurística que determinemos.

La suma de estas dos variables representa el costo estimado del camino menos costoso. Mientras queden nodos abiertos en la lista, el nodo con el valor de esta suma más bajo se retira de la lista, buscando que sea el nodo objetivo. Si no lo es, se abren sus sucesores y repetimos el ciclo hasta encontrar el nodo objetivo.

```python
class AEstrella(Busqueda):

    def insertar(self, nodo, abiertos, objetivo):
        # Utilizando PriorityQueue solo para AEstrella
        costoNodoHeuristica = self.f(nodo, objetivo)
        abiertos.put((costoNodoHeuristica, nodo))

    def eliminar(self, abiertos):
        return abiertos.get()[1]


    def f(self, nodo, objetivo):
        # Función f(n) = g(n) + h(n), donde g(n) es el costo acumulado y h(n) es la heurística
        return nodo.costo + self.heuristica(nodo, objetivo)

    def heuristica(self, nodo, objetivo):
        # Implementación de la heurística (distancia de Manhattan)
        return abs(nodo.estado.posicion[0] - objetivo.posicion[0]) + abs(nodo.estado.posicion[1] - objetivo.posicion[1])

    def inicializarAbiertos(self):
        return PriorityQueue()
```

1. **Método “insertar(self, nodo, abiertos, objetivo)”:** Utilizaremos una cola de prioridad en función de la heurística que escojamos, pero solo para AEstrella.

2. **Método “eliminar(self, abiertos)”:** Eliminamos y devolvemos el primer elemento de la lista, que va a tener la menor heurística.

3. **Método “heurística(self, nodo, objetivo)”:** Para la heurística hemos decidido escoger la “distancia de Manhattan”, que une dos puntos en un grid y su distancia se determina por la suma de las diferencias absolutas de sus coordenadas en cada dimensión. A diferencia de primero mejor, calcularemos la suma del coste acumulado en el nodo más la heurística.

4. **Método inicializarAbiertos(self):** Devuelve la cola de prioridad.

1.4 Evaluación experimental

instance-5-5-t1

Algoritmo	Generados	Expandidos	Tiempo ejec.	Profundidad	Coste
Anchura	15	7	0.0	4	16
Profundidad	25	12	0.0	12	12
P.Limitada	19	9	0.0	4	16
P.Iterativa	17	8	0.001	4	16
PrimeroMejor	9	4	0.0	4	16
AEstrella	29	14	0.0009	12	12

instance-50-75-563-4-563-2023

instance-50-75-563-4-563-2023

Algoritmo	Generados	Expandidos	Tiempo ejec.	Profundidad	Coste
Anchura	2414	701	0.007	22	34
Profundidad	3560	1049	0.012	928	1622
P.Limitada	4218	1243	0.01	288	474
P.Iterativa	2831	829.5	0.0047	87	166
PrimeroMejor	104	31	0.001	29	44
AEstrella	477	141	0.00627	30	31

1.5 Conclusiones

Como conclusión de esta práctica, podemos observar que en problemas más complejos podríamos tener soluciones gigantescas si utilizamos búsqueda no informada, algo que nos solucionan las búsquedas informadas. Sin embargo, estas búsquedas requieren una heurística válida y son más complejas de implementar.

4. Práctica 2: Aprendizaje por refuerzo

a. Introducción

En el desarrollo de esta práctica, abordaremos la implementación de dos destacados algoritmos de aprendizaje por refuerzo: Q-learning e iteración de valores. Estos algoritmos están diseñados para descubrir y optimizar la política óptima en un entorno específico. En el contexto de este ejercicio práctico, nos enfrentamos a la resolución de un desafío que representa una ciudad modelada como una cuadrícula. Esta ciudad cuenta con un punto de partida, áreas de peligro, zonas de alto riesgo y nuestros objetivos atrapados, cada una de las cuales posee valores asociados. El objetivo es abordar eficientemente la dinámica de este entorno para encontrar la estrategia más efectiva y segura.

b. Representación del problema

En el código correspondiente a esta práctica, se ha implementado dos algoritmos de aprendizaje por refuerzo, específicamente Q-learning e iteración de valores, con el propósito de abordar y resolver un problema particular. El objetivo es hallar una política óptima en un entorno predefinido. A continuación, una breve descripción de la estructura y organización de este código.

i. Clase Estado

La clase "Estado" representa el estado actual de nuestro robot en el entorno definido. Al instanciar la clase, se le proporciona la posición inicial y, opcionalmente, el problema en el que se encuentra.

1. **Método “acciones”**: el método “acciones” que permite al robot realizar movimientos en el entorno, especificados por un código de acción. Estos movimientos incluyen arriba, derecha, abajo e izquierda.

   Dentro del método "acciones", se verifica la validez de la nueva posición resultante del movimiento, considerando los límites del entorno y posibles obstáculos definidos en el problema. En caso de que la nueva posición sea válida, se devuelve un nuevo objeto de la clase "Estado" con la posición actualizada. Si la nueva posición está fuera de los límites o es bloqueada, se devuelve el estado actual sin cambios.

```python
class Estado:
    def __init__(self, posicion, problema=None):
        self.posicion = posicion
        self.problema = problema

    #Acciones de movimiento del robot
    def acciones(self, accion):

        if accion==0: # Arriba
            nueva_pos = ((self.posicion[0] - 1,self.posicion[1]))
        elif accion==1: # Derecha
            nueva_pos = ((self.posicion[0], self.posicion[1] + 1))
        elif accion==2: # Abajo
            nueva_pos = ((self.posicion[0] + 1,self.posicion[1]))
        elif accion==3: #Izquierda
            nueva_pos = ((self.posicion[0], self.posicion[1] - 1))

        # Verificar límites y bloqueados
        if (0 <= nueva_pos[0] < self.problema.nfilas and 0 <= nueva_pos[1] < self.problema.ncols
            and list(nueva_pos) not in self.problema.bloqueados):
            return Estado(nueva_pos)
        else:
        # Si la nueva posición está fuera de los límites o es bloqueada, devolver el estado actual
            return self
```

ii. Clase Color

La clase “Color” en este código proporciona códigos de formato ANSI para cambiar el color y estilo del texto en la salida de la consola. Cada color está representado por constantes, como **AMARILLO**, **MAGENTA**, **NEGRO**, **ROJO**, **VERDE**, **BLANCO**, y **AZUL**, cada una con su respectivo código ANSI.

1. **Método “estiloTexto”:** el método llamado estiloTexto, toma un texto y un color como argumentos y devuelve el texto formateado con el color especificado. Este enfoque se utiliza para resaltar diferentes elementos al mostrar la tabla Q en la consola, mejorando la legibilidad y comprensión de la información presentada.

```python
class Color:
    AMARILLO = "\033[93m"
    MAGENTA = "\033[95m"
    NEGRO = "\033[30m"
    ROJO = "\033[91m"
    VERDE = "\033[92m"
    BLANCO = "\033[97m"
    AZUL = "\033[38;5;45m"

    def estiloTexto(texto, color):
        return f"{color}{texto}{Color.BLANCO}"
```

iii. Clase Problema

La clase Problema en este código se encarga de cargar la información de los problema definidos mediante un archivo JSON. Al instanciar la clase con la ruta del archivo, el constructor lee el archivo JSON y extrae datos esenciales, como el número de filas y columnas en el entorno, las posiciones bloqueadas, la posición de inicio, las zonas de peligro, las zonas de peligro fatal y los objetivos atrapados.

1. **Método "finales":** esta clase proporciona un método llamado finales que devuelve listas de posiciones asociadas a objetivos alcanzables y zonas de peligro fatal.

```python
class Problema:
    def __init__(self, problema):
        with open(problema, 'r') as file:
            self.nuevo_diccionario = json.load(file)

        self.nfilas=self.nuevo_diccionario["city"]["rows"]
        self.ncols=self.nuevo_diccionario["city"]["columns"]
        self.bloqueados=self.nuevo_diccionario["city"]["blocked"]
        self.partida=self.nuevo_diccionario["departure"]
        self.peligro=self.nuevo_diccionario["dangers"]
        self.peligro_fatal=self.nuevo_diccionario["fatal_dangers"]
        self.atrapados=self.nuevo_diccionario["trapped"]


    def finales(self):
        lista_buenos = [[fila, columna] for fila, columna, recompensa in self.atrapados]
        lista_malos = [[fila, columna] for fila, columna, recompensa in self.peligro_fatal]

        return lista_buenos, lista_malos
```

Clase Agente

La clase Agente representa el agente de aprendizaje por refuerzo que utiliza los algoritmos Q-learning e iteración de valores para abordar un problema específico. Al instanciar la clase con un objeto de la clase Problema, el agente adquiere conocimiento sobre el entorno. La clase Agente incluye métodos para construir y actualizar la tabla Q, realizar el aprendizaje mediante el método Q-learning, ejecutar la iteración de valores, y calcular y asignar recompensas.

1. **Metodo "Qtabla":** se encarga de inicializar la tabla Q utilizada en los algoritmo de Q-learnign e iteracion de valores. Inicializa la tabla Q con valores iniciales de cero, asignando recompensas específicas para los nodos finales, tales como aquellos asociados a los objetivos atrapados y a las zonas de peligro fatal.

2. **Método "recompensa":** tiene como función principal determinar la recompensa asociada a una posición específica en el entorno del problema. Este método verifica si la posición a la que el agente se dirige es un nodo decisivo, como puede ser una zona de peligro (dangers), un objetivo atrapado (trapped), o una zona de peligro fatal (fatal_dangers). En caso afirmativo, el método devuelve una recompensa diferente a la recompensa por defecto, ajustando el valor según el tipo de nodo decisivo.

3. **Método "formato":** se encarga de organizar los datos de la tabla Q de manera más estructurada y legible para su presentación en la consola. Este método intenta mejorar la presentación de los valores de la tabla Q, aplicando un formato específico con un número definido de decimales.

4. **Método "mostrarQTabla":** imprime la tabla Q en la consola, aplicando un formato visual que destaca diferentes elementos mediante el uso de colores.

Estructuras claves para la Resolución:

1. **Q-tabla:** Un componente central en el algoritmo Q-learning e iteración de valores, la Q-tabla es una matriz que almacena valores que representan la utilidad esperada de realizar acciones en estados particulares de un entorno. Cada celda de la matriz contiene información sobre la calidad de una acción en un estado específico, y estas valoraciones se actualizan durante el proceso de aprendizaje para mejorar la toma de decisiones del agente.

2. **Función de Recompensa**: Encargada de calcular la recompensa asociada a una posición en la ciudad, la función "recompensa" es esencial para actualizar los valores en la Q-tabla.

3. **Iteraciones de Aprendizaje:** El método "busqueda" lleva a cabo iteraciones de aprendizaje por refuerzo. En cada iteración, el agente toma decisiones basadas en la Q-tabla y ajusta los valores correspondientes. Se recomienda realizar más iteraciones para problemas más grandes, pero esto puede aumentar significativamente el costo computacional.

Algoritmo Q-learning

El algoritmo Q-learning es un método de aprendizaje por refuerzo que permite a un agente aprender a tomar decisiones óptimas en un entorno desconocido. A continuación, se describe brevemente el algoritmo:

1. **Inicialización de la Q-Tabla:**
   - Se crea una tabla Q, que es una matriz donde las filas representan los estados posibles y las columnas representan las acciones posibles.
   - Inicialmente, todos los valores de la tabla Q se establecen en cero.

2. **Iteraciones de Aprendizaje:**
   - El agente interactúa con el entorno realizando acciones y recibiendo recompensas.
   - En cada iteración, el agente selecciona una acción en función de la política de exploración/explotación, que determina si el agente elige una acción aleatoria (exploración) o la mejor acción conocida (explotación).
   - El agente ejecuta la acción, observa la recompensa y la nueva posición del estado, y actualiza la tabla Q utilizando la ecuación de actualización Q.

3. Ecuación de Actualización Q
   - La tabla Q se actualiza utilizando la siguiente ecuación:

$$Q(s, a) = (1 - \alpha) * Q(s, a) + \alpha * [r + \gamma * \max(Q(s', a'))]$$

donde:
- **Q(s, a)** es el valor actual en la tabla Q para el estado “s” y la acción “a”.
- **α** es la tasa de aprendizaje.
- **r** es la recompensa obtenida por realizar la acción.
- **γ** es el factor de descuento que pondera la importancia de las recompensas futuras.

- s' es el próximo estado.
- max(Q(s', a')) representa el valor máximo en la tabla Q para las acciones posibles en el próximo estado.

4. Exploración/Explotación:
- Durante la exploración, el agente elige acciones de manera aleatoria para descubrir nuevas experiencias.
- Durante la explotación, el agente selecciona la acción con el mayor valor Q conocido.

5. Convergencia:
- El proceso de iteraciones continúa hasta que el agente alcanza un nivel satisfactorio de aprendizaje o un número predeterminado de iteraciones.

Algoritmo de iteración de valores

El algoritmo iteración de valores busca aprender una política óptima que maximice la recompensa total a largo plazo. La actualización incremental de la tabla Q permite al agente adaptarse y mejorar sus decisiones a medida que interactúa con el entorno.

El algoritmo de iteración de valores es una técnica utilizada en aprendizaje por refuerzo para encontrar la política óptima en un entorno. Aquí se describe brevemente el algoritmo:

1. Inicialización de la Función de Valor:
- Se inicia con una estimación inicial de la función de valor para cada estado en el entorno.

2. Iteraciones de Actualización:
- El algoritmo realiza iteraciones para mejorar gradualmente las estimaciones de los valores de los estados.
- En cada iteración, se calcula una nueva estimación para el valor de cada estado basándose en los valores de los estados adyacentes y las recompensas asociadas a las transiciones.

3. Ecuación de Actualización de Valor:
- La ecuación de actualización de valor generalmente toma la forma:

$$V(s) = \max [R(s, a) + \gamma * V(s')]$$

donde:
- V(s) es el valor del estado actual s.
- R(s, a) es la recompensa de realizar la acción “a” en el estado “s”.
- γ es el factor de descuento que pondera las recompensas futuras.
- V(s') es el valor del próximo estado “s'” alcanzado después de tomar la acción “a”.

4. Convergencia:

- El proceso de iteración se repite hasta que los valores convergen y dejan de cambiar significativamente entre iteraciones.

5. Política Óptima:
- Una vez que los valores han convergido, la política óptima se puede derivar eligiendo las acciones que maximizan el valor en cada estado.

El algoritmo de iteración de valores busca mejorar sistemáticamente las estimaciones de la función de valor de los estados, lo que a su vez permite determinar una política que maximiza la recompensa total esperada en el entorno. Este enfoque es particularmente útil cuando se tiene un modelo completo del entorno, es decir, cuando se conocen las transiciones y recompensas asociadas a cada acción y estado.

Evaluación experimental:

Q-learning:

```
                elif [i, j] == self.problema.partida:
                    print(Color.estiloTexto(elemento, Color.VERDE), end=" ")
                else:
                    print(elemento, end=" ")
            print()


problema = Problema("Aprendizaje-por-refuerzo/initial-rl-instances/larger-rl.json")
agente = Agente(problema)
qTablaResultante = agente.qLearning(0.8, -0.04, 0.1, 0.5, 1000) # gamma, r, alfa, epsilon, iteraciones
# qTablaResultante = agente.iteracionValores(0.8, -0.5, 1000) # gamma, r, iteraciones
agente.mostrarQTabla(qTablaResultante)
```

```
PROBLEMAS 1  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  COMENTARIOS

PS C:\Users\santa\OneDrive - misena.edu.co\Documentos\Programación\Clases\Sistemas inteligentes\Practica\Practica-1-y-2> & C:/Users/santa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/santa/OneDrive - misena.edu.co/Documentos/Programación/Clases/Sistemas inteligentes/Practica/Practica-1-y-2/Aprendizaje-por-refuerzo/Q-Learning.py"
Tiempo de ejecución:  0.08303666114807129
['8.11', '41.41', '11.72', '8.98'] ['25.00']                    ['9.22', '17.68', '12.29', '42.88'] ['0.00', '0.00', '0.00', '0.00'] ['0.03', '0.77', '-0.00', '-0.00'] ['0.17', '7.68', '0.00', '0.00'] ['-0.00', '25.30', '0.00', '1.00'] ['30.00']
['27.15', '35.96', '21.96', '25.88'] ['45.00', '28.73', '28.73', '28.73'] ['30.41', '20.41', '21.31', '35.96'] ['10.47', '1.17', '6.69', '28.14'] ['0.01', '0.17', '0.30', '7.76'] ['0.00', '0.00', '0.00', '0.00'] ['-0.00', '0.43', '0.00', '-0.01'] ['10.26', '0.00', '-0.00', '0.00']
['28.47', '28.73', '18.23', '22.76'] ['35.96', '22.94', '22.94', '22.94'] ['28.45', '18.16', '17.91', '28.73'] ['11.56', '5.15', '12.65', '22.94'] ['0.24', '0.01', '4.05', '14.19'] ['-0.01', '-0.01', '-0.01', '1.07'] ['-0.01', '-0.00', '-0.01', '0.03'] ['0.42', '0.00', '0.00', '0.00']
['22.88', '22.94', '14.57', '18.27'] ['28.73', '18.31', '18.31', '18.31'] ['22.94', '14.56', '14.61', '22.94'] ['17.79', '10.64', '11.54', '18.31'] ['4.49', '2.05', '3.78', '14.57'] ['-0.02', '1.01', '0.28', '8.48'] ['0.00', '0.00', '0.00', '0.00'] ['0.00', '0.00', '0.00', '0.00']
['18.31', '17.80', '10.61', '12.92'] ['22.94', '14.61', '14.61', '14.61'] ['18.31', '11.65', '11.65', '18.31'] ['14.61', '9.28', '9.28', '14.61'] ['10.46', '4.06', '5.19', '11.65'] ['2.37', '0.15', '1.95', '7.36'] ['-0.02', '-0.02', '-0.02', '1.70'] ['-0.02', '-0.02', '-0.02', '0.09']
['10.98', '14.54', '2.82', '4.51'] ['18.31', '11.31', '9.32', '10.16'] ['14.61', '8.76', '7.27', '13.63'] ['11.65', '6.35', '6.09', '11.17']
 ['5.55', '2.11', '0.60', '9.03'] ['0.00', '0.00', '0.00', '0.00'] ['-0.02', '-0.02', '-0.02', '-0.02'] ['0.00', '0.00', '0.00', '0.00']
['8.37', '-0.01', '-19.51', '1.16'] ['13.49', '4.95', '-18.78', '1.20'] ['11.05', '1.85', '0.92', '5.66'] ['8.89', '0.12', '-30.88', '3.49']
 ['1.29', '-0.02', '-0.02', '2.42'] ['-0.02', '-0.02', '-0.01', '-0.02'] ['-0.01', '-0.01', '-0.01', '-0.01'] ['-0.01', '-0.00', '-2.70', '-
```

Iteración de valores:

```
                    print(Color.estiloTexto(elemento, Color.VERDE), end=" ")
                else:
                    print(elemento, end=" ")
            print()


problema = Problema("Aprendizaje-por-refuerzo/initial-rl-instances/larger-rl.json")
agente = Agente(problema)
# qTablaResultante = agente.qLearning(0.8, -0.04, 0.1, 0.5, 1000) # gamma, r, alfa, epsilon, iteraciones
qTablaResultante = agente.iteracionValores(0.8, -0.5, 1000) # gamma, r, iteraciones
agente.mostrarQTabla(qTablaResultante)
```

```
PROBLEMAS 1  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  COMENTARIOS

pData/Local/Programs/Python/Python311/python.exe "c:/Users/santa/OneDrive - misena.edu.co/Documentos/Programación/Clases/Sistemas inteligent
es/Practica/Practica-1-y-2/Aprendizaje-por-refuerzo/Q-Learning.py"
Tiempo de ejecución:  0.08303666114807129
['8.11', '41.41', '11.72', '8.98'] ['25.00']                                  ['9.22', '17.68', '12.29', '42.88'] ['0.00', '0.00', '0.00', '0.00'] [
'0.03', '0.77', '-0.00', '-0.00'] ['0.17', '7.68', '0.00', '0.00'] ['-0.00', '25.30', '0.00', '1.00'] ['30.00']
['27.15', '35.96', '21.96', '25.88'] ['45.00', '28.73', '28.73', '28.73'] ['30.41', '20.41', '21.31', '35.96'] ['10.47', '1.17', '6.69', '28
.14'] ['0.01', '0.17', '0.30', '7.76'] ['0.00', '0.00', '0.00', '0.00'] ['-0.00', '0.43', '0.00', '-0.01'] ['10.26', '0.00', '-0.00', '0.00'
]
['28.47', '28.73', '18.23', '22.76'] ['35.96', '22.94', '22.94', '22.94'] ['28.45', '18.16', '17.91', '28.73'] ['11.56', '5.15', '12.65', '2
2.94'] ['0.24', '0.01', '4.05', '14.19'] ['-0.01', '-0.01', '-0.01', '1.07'] ['-0.01', '-0.00', '-0.01', '0.03'] ['0.42', '0.00', '0.00', '0
.00']
['22.88', '22.94', '14.57', '18.27'] ['28.73', '18.31', '18.31', '18.31'] ['22.94', '14.56', '14.61', '22.94'] ['17.79', '10.64', '11.54', '
18.31'] ['4.49', '2.05', '3.78', '14.57'] ['-0.02', '1.01', '0.28', '8.48'] ['0.00', '0.00', '0.00', '0.00'] ['0.00', '0.00', '0.00', '0.00'
]
['18.31', '17.80', '10.61', '12.92'] ['22.94', '14.61', '14.61', '14.61'] ['18.31', '11.65', '11.65', '18.31'] ['14.61', '9.28', '9.28', '14
.61'] ['10.46', '4.06', '5.19', '11.65'] ['2.37', '0.15', '1.95', '7.36'] ['-0.02', '-0.02', '-0.02', '1.70'] ['-0.02', '-0.02', '-0.02', '0
.09']
['10.98', '14.54', '2.82', '4.51'] ['18.31', '11.31', '9.32', '10.16'] ['14.61', '8.76', '7.27', '13.63'] ['11.65', '6.35', '6.09', '11.17']
 ['5.55', '2.11', '0.60', '9.03'] ['0.00', '0.00', '0.00', '0.00'] ['-0.02', '-0.02', '-0.02', '-0.02'] ['0.00', '0.00', '0.00', '0.00']
['8.37', '-0.01', '-19.51', '1.16'] ['13.49', '4.95', '-18.78', '1.20'] ['11.05', '1.85', '0.92', '5.66'] ['8.89', '0.12', '-30.88', '3.49']
 ['1.29', '-0.02', '-0.02', '2.42'] ['-0.02', '-0.02', '-0.01', '-0.02'] ['-0.01', '-0.01', '-0.01', '-0.01'] ['-0.01', '-0.00', '-2.70', '-
0.01']
```