Universidad Pontificia de Comillas
ICAI

PROYECTO FINAL

Fundamentos de Inteligencia Artificial

Lydia Ruiz Martínez

BUSCANDO AL CORONEL KURTZ

Curso 2023-2024

Resumen

En esta memoria se presenta el proyecto final de la asignatura de Fundamentos de Inteligencia Artificial, realizado por Lydia Ruiz Martínez. Se abordan los siguientes temas:

- **Descripción de la aproximación al problema:** Se emplean técnicas de búsqueda, inferencia lógica e inferencia bayesiana estudiadas en el curso. Se desarrollan dos agentes: un agente lógico y un agente lógico bayesiano, los cuales toman decisiones de manera autónoma mediante algoritmos de búsqueda específicos para cada agente.
- **Diseño de la implementación de la interfaz gráfica:** Se presenta el diseño de la interfaz gráfica, incluyendo imágenes de los resultados obtenidos durante la ejecución del proyecto.
- **Cuentas pertinentes:** Se detallan las cuentas necesarias, proporcionando una visión clara de los cálculos realizados. Es especialmente importante leer con detenimiento los resultados obtenidos al implementar el agente bayeasiano.

La resolución del problema se ha implementado en Python. Este proyecto consta de los siguientes elementos de desarrollo software:

- Módulos: `kurtz.py` (el programa principal o main), `agente_logico.py`, `agente_bayesiano.py` y `agente_buscador.py`.
- 1369 líneas de código.
- 91 funciones.
- 9 clases.
- Librerías: colorama y random.
- 1 fichero de texto: `requirements.txt`, que especifica las dependencias (librerías de python no estándar) del proyecto.

Índice

1. Contexto

En el universo cinematográfico de "Apocalypse Now", el Capitán Willard emprende una misión encomendada para localizar y persuadir al Coronel Kurtz, instándolo a abandonar su desafiante postura y regresar a Washington para rendir cuentas. Su odisea se desarrolla en un peligroso palacio, repleto de trampas mortales, abismos profundos y criaturas monstruosas.

En este contexto, el capitán se ve obligado a depender de sus sentidos y de las técnicas de rastreo aprendidas en el ejército, coincidentemente alineadas con los conceptos de búsqueda, lógica y probabilidad abordados en la asignatura de Fundamentos de Inteligencia Artificial.

El propósito de esta memoria es analizar y desglosar los elementos clave de este desafío, equiparable a un programa donde la toma de decisiones eficiente y la resolución de obstáculos se traducen en el éxito de la misión del Capitán Willard.

A lo largo de las secciones subsiguientes, se proporcionará un análisis detallado de los componentes del problema y se explicarán las estrategias y soluciones específicas requeridas para enfrentar cada uno de los objetivos establecidos: evitar obstáculos, localizar al Coronel Kurtz y, finalmente, dirigirse hacia la salida del intrincado palacio una vez que se haya encontrado al elusivo coronel.

Figura 1: Capitán Willard en la película inspirada en el libro *Heart of Darkness*

2. Programas de los agentes

Un agente se define como una entidad capaz de percibir su entorno con la ayuda de sensores y actuar en ese entorno utilizando actuadores.

2.1. Agente lógico

2.1.1. Entorno

En el contexto de la Inteligencia Artificial, el término entorno se refiere al espacio en el que opera un agente inteligente.

Un entorno es totalmente observable si los sensores detectan todos los aspectos que son relevantes en la toma de decisiones, de lo contrario, es parcialmente observable.

En nuestro caso el palacio donde se encuentra el Coronel Kurtz consiste en una red de 6 salas interconectadas y se trata de un entorno parcialmente observable.

- Ciertas salas contienen trampas en forma de precipicios o simas, que le llevarían a una muerte segura al despeñarse. En concreto, tendremos tres precipicios, cada uno en una localización diferente.
- En una de las salas, hay un monstruo que puede acabar con el Capitán Willard. El monstruo acabaría con él si ambos se encontrasen en la misma sala. Por otra parte, si el Coronel (CK) se encuentra en una de esas celdas afectadas por la granada, él no sufrirá daño alguno.
- La salida, cuya posición no es conocida de entrada, pero que emite un claro resplandor en las celdas adyacentes.
- El Coronel Kurtz, que siempre habita una celda que no puede ser ni la del monstruo ni la de ninguno de los precipicios.

Cabe destacar que en este caso cualquier elemento puede estar en cualquier celda (excepto el Coronel Willard, que siempre comienza en (0,0), y no hay más de un elemento en una misma celda. No obstante, no puede haber precipicios en las casillas (0,1) y (1,0) simultáneamente, ya que en este caso al comenzar el juego nuestro capitán no tendría posibilidad alguna de salvarse. En nuestro juego el monstruo tampoco puede aparecer en estas posiciones para que al empezar el jugador tenga completa libertad de movimiento sin necesidad de detonar la bomba.

A continuación, se observa una configuración aleatoria del estado inicial del palacio, con 36 habitaciones en una disposición cuadrada de 6×6.

Figura 2: Configuración inicial posible

2.1.2. Acciones

El Capitán Willard puede realizar las siguientes acciones:

1. Moverse: Puede moverse arriba, derecha, izquierda o abajo, evitando obstáculos y peligros. Si el capitán entra en una sala que contiene un precipicio, se verá abocado a una muerte dolorosa, tanto si va solo como si va en compañía del Col. Kurtz. Si el capitán entra en la sala del monstruo, y este está vivo, será devorado por él inmediatamente.

2. Detonar: Como arma, el Capitán Willard porta una granada de proximidad especializada, diseñada para enfrentar a monstruos, la cual puede detonar en cualquier momento. Sin embargo, solo surtirá efecto contra el monstruo si este se encuentra en una celda contigua a la ocupada por el capitán. Las celdas contiguas se definen como aquellas ubicadas en las posiciones adyacentes a la celda del capitán, es decir, arriba, abajo, a la izquierda y a la derecha.

3. Salir: Si se encuentra en la celda de salida, el capitán puede ejecutar esta acción para abandonar el palacio y completar la misión. Si se ejecuta la acción de salir en cualquier otra celda, el estado del mundo se mantendrá como estaba.

En nuestro juego se nos pide que ingresemos una acción entre las disponibles. Para ello simplemente hay que escribir “mover”, “detonar” o “salir”.

Si seleccionamos “mover”, se nos pide que ingresemos la dirección. Para ello hay que introducir “arriba”, “abajo”, “izquierda” o “derecha”. En el caso de que queramos movernos a una celda no permitida (por ejemplo, si estamos en la pared superior y queremos movernos arriba), nos salta un mensaje diciendo que no podemos atravesar las paredes.

Por otra parte, podemos “detonar” la granada. Si el monstruo no se encuentra en una celda adyacente, aparece un mensaje diciendo que la granada no ha tenido efecto. En este caso consideramos que hay un número ilimitado de granadas.

La última acción disponible para nuestro agente es “salir”. Esta acción solo puede realizarse en la celda de salida, en caso contrario se muestra un mensaje diciedo que la acción solicitada no está disponible. Si nos encontramos en la celda de salida, hemos encontrado al Coronel Kurtz y solicitamos salir, se nos muestra un mensaje diciendo que hemos ganado. Si no lo hemos encontrado, se muestra un mensaje diciendo que nos hemos rendido. En ambos casos acabaría el juego.

2.1.3. Perceptos

El Capitán no tiene información previa sobre la disposición de los precipicios, el monstruo, el coronel o la salida. Para el posterior análisis que realizará nuestro agente lógico necesitaremos, por tanto, hacer una observación parcial del estado del entorno.

1. Una brisa: se detecta si el Capitán Willard se encuentra en una sala adyacente a otra que contiene un precipicio.
2. Un fétido olor: se percibe si el Capitán Willard está en una sala adyacente a la ubicación del monstruo.
3. Un resplandor: se observa si el Capitán Willard está en la salida o en una celda adyacente a esta.
4. Un sensor de paredes: indica la existencia de una pared que impide la ejecución de ciertos movimientos (arriba, abajo, derecha, izquierda).
5. Grito estentóreo: se escucha como resultado de detonar la granada anti-monstruo y eliminar al monstruo. Indica el fin del monstruo, y la celda del monstruo se vuelve transitable al haberlo neutralizado.
6. Reconocimiento del Col. Kurtz: cuando el Capitán Willard encuentra al Col. Kurtz, lo reconocerá y no tendrá problemas para convencerlo de unirse a él.

2.1.4. Implementación en Python del entorno, las acciones y los perceptos

Una vez explicados el entorno de nuestro palacio y las acciones y los perceptos de nuestro capitán, se muestra cómo se ha implementado lo descrito a través del lenguaje de programación de Python.

Para ello se han creado tres clases: class Room, class Palace y class CaptainWillard, que se exponen a continuación.

```python
class Room:
    """
    Class that represents a room in the game
    """

    def __init__(self, element, color, position):
        self.element = element
        self.color = color
        self.position = position

    def __str__(self):
        return f"{self.color}{self.element}{Fore.RESET}"
```

Figura 3: class Room

La clase Palace es más compleja. Consta del constructor y de los métodos siguientes: generar el mapa, conseguir una posición aleatoria, colocar un elemento, mostrar el palacio, encontrar la posición de un elemento y detectar si una celda contiene un precipicio, al monstruo, al coronel, si es la salida o si es segura (es decir, que no contiene ni un precipicio ni al monstruo).

```python
class Palace:
    """
    Class that represents the game map (palace)
    """

    # Define special elements with their colors and quantities
    elements = [
        (Room("|:P::|", Fore.MAGENTA, (0, 0)), 3),
        (Room("|:M::|", Fore.LIGHTCYAN_EX, (0, 0)), 1),
        (Room("|:CW:|", Fore.RED, (0, 0)), 1),
        (Room("|:CK:|", Fore.BLUE, (0, 0)), 1),
    ]

    def __init__(self, dimension):
        # Initialize the palace with a given dimension
        self.dimension = dimension

        # Define base, entry, and exit cells
        self.base_cell = Room("|::::|", "", (0, 0))
        self.entry_cell = Room("|::::|", Fore.YELLOW, (0, 0))
        self.exit_cell = Room("|:S::|", Fore.GREEN, (0, 0))

        # Create a 2D board filled with base cells
        self.board = [[self.base_cell] * self.dimension for _ in range(self.dimension)]

        # Generate the map with entry, exit, and special elements
        self.generate_map()

    def generate_map(self):
        # Set the entry cell at position (0, 0)
        entry_x, entry_y = 0, 0
        self.board[entry_x][entry_y] = self.entry_cell

        # Set the exit cell at a random position
        exit_x, exit_y = self.get_random_position()
        self.board[exit_x][exit_y] = self.exit_cell

        # Place special elements on the map
        for room, count in self.elements:
            if room.element == "|:CW:|":
                self.board[0][0] = room
            elif room.element == "|:CK:|":
                self.place_element(room, count)
            else:
                self.place_other_element(room, count)
```

Figura 4: class Palace

```python
def get_random_position(self):
    # Get a random position on the map that is not occupied
    while True:
        x, y = random.randint(0, self.dimension - 1), random.randint(
            0, self.dimension - 1
        )
        if self.board[x][y] == self.base_cell:
            return x, y

def place_element(self, room, count):
    # Place a specific element on the map in random positions
    for _ in range(count):
        x, y = self.get_random_position()
        self.board[x][y] = room

def place_other_element(self, room, count):
    # Place elements (excluding specific positions) on the map in random positions
    for _ in range(count):
        while True:
            x, y = self.get_random_position()
            if (x, y) != (1, 0) and (x, y) != (0, 1):
                break
        self.board[x][y] = room

def display_palace(self):
    # Print the current state of the palace board
    for row in self.board:
        print("".join(str(room) for room in row))

def find_element_position(self, element):
    # Find the position of a specific element on the map
    for x in range(self.dimension):
        for y in range(self.dimension):
            if self.board[x][y].element == element:
                return x, y
    return None
```

Figura 5: class Palace

```python
def is_precipice(self, position):
    # Check if a specific position contains a precipice
    x, y = position
    return self.board[x][y].element == "|:P::|"

def is_monster(self, position):
    # Check if a specific position contains a monster
    x, y = position
    return self.board[x][y].element == "|:M::|"

def is_coronel(self, position):
    # Check if a specific position contains Colonel Kurtz
    x, y = position
    return self.board[x][y].element == "|:CK:|"

def is_exit(self, position):
    # Check if a specific position contains the exit
    x, y = position
    return self.board[x][y].element == "|:S::|"

def is_secure(self, position):
    # Check if a specific position is not dangerous (not a precipice or monster)
    x, y = position
    return (
        self.palace.board[x][y].element != "|:P::|"
        and self.palace.board[x][y].element != "|:M::|"
    )
```

Figura 6: class Palace

Finalmente disponemos de la clase CaptainWilliam, que consta del constructor y de los métodos que permiten realizar las acciones: mover, detonar y salir. Además hay otros métodos auxiliares llamados en los métodos anteriores: calcular nueva posición, si es movimiento válido, derrotar al monstruo y obtener las celdas adyacentes. Por último está el método más importante para el siguiente apartado: obtener los perceptos.

```python
class CaptainWillard:
    """
    Class that represents Captain Willard
    """

    def __init__(self, palace):
        # Initialize Captain Willard with starting attributes
        self.position = (0, 0)
        self.palace = palace
        self.alive = True
        self.monster_defeated = False
        self.kurtz_found = False
        self.explored_cells = []

    # Captain Willard's actions: MOVE, DETONATE, EXIT

    def move(self, direction):
        # Move Captain Willard in a given direction
        new_position = self.calculate_new_position(direction)

        if self.is_valid_move(new_position):
            # Update explored cells and handle specific cases (precipice, monster, Kurtz)
            self.explored_cells.append(self.position)
            x, y = self.position

            if (
                self.palace.board[new_position[0]][new_position[1]]
                == self.palace.base_cell
            ):
                (
                    self.palace.board[x][y],
                    self.palace.board[new_position[0]][new_position[1]],
                ) = (self.palace.base_cell, self.palace.board[x][y])

            else:
                if self.palace.is_precipice(new_position):
                    self.alive = False
                    print(
                        "¡Ahhh! Capitán Willard ha caído en un precipicio. Ha muerto."
                    )

                elif self.palace.is_monster(new_position):
                    self.alive = False
                    print(
                        "¡Ahhh! Capitán Willard ha sido devorado por el monstruo. Ha muerto."
                    )

                else:
                    self.kurtz_found = True
                    (
                        self.palace.board[x][y],
                        self.palace.board[new_position[0]][new_position[1]],
                    ) = self.palace.base_cell, Room(
                        "|CWCK|", Fore.LIGHTYELLOW_EX, (0, 0)
                    )
                    print(
                        "Está en la misma celda que en Coronel Kurtz, ahora viaja junto a él."
                    )

                self.position = new_position

        else:
            print("Movimiento no válido. ¡No puedes atravesar las paredes!")
```

Figura 7: class CaptainWillard

```python
def calculate_new_position(self, direction):
    # Calculate the new position based on the given direction
    x, y = self.position
    if direction == "arriba":
        return x - 1, y
    elif direction == "abajo":
        return x + 1, y
    elif direction == "izquierda":
        return x, y - 1
    elif direction == "derecha":
        return x, y + 1
    else:
        print("Introduzca una dirección válida.")

def is_valid_move(self, position):
    # Check if a given position is within the palace boundaries
    x, y = position
    return 0 <= x < self.palace.dimension and 0 <= y < self.palace.dimension

def detonate(self):
    # Detonate a grenade to defeat the monster if it's adjacent

    adjacent_cells = self.get_adjacent_cells()
    monster_position = self.palace.find_element_position("|:M::|")

    if monster_position in adjacent_cells:
        print("¡Boom! La granada haexplotado y has derrotado al monstruo.")
        self.defeat_monster(monster_position)
        self.monster_defeated = True
        self.get_perception()[7] == True

    else:
        print(
            "No hay monstruo cerca. La granada no ha tenido efecto. "
        )


def defeat_monster(self, monster_position):
    # Remove the defeated monster from the board
    x, y = monster_position
    self.palace.board[x][y] = self.palace.base_cell

def get_adjacent_cells(self):
    # Get the positions of cells adjacent to Captain Willard's current position
    x, y = self.position
    adjacent_cells = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
    return [cell for cell in adjacent_cells if self.is_valid_move(cell)]

def exit_palace(self):
    # Attempt to exit the palace
    if self.palace.is_exit(self.position):
        if self.kurtz_found:
            print(
                "¡Felicidades! Has completado tu misión y has encontrado a Kurtz. Has ganado."
            )
            self.alive = False
        else:
            print("Te has rendido.")
            self.alive = False
    else:
        print("Solo puedes realizar esta acción si estás en la celda de salida.")
```

Figura 8: class CaptainWillard

```python
# Captain Willard's perceptions

def get_perception(self):
    # Get perceptions based on Captain Willard's current state
    adjacent_cells = self.get_adjacent_cells()
    monster_position = self.palace.find_element_position("|:M::|")
    exit_position = self.palace.find_element_position("|:S::|")

    breezy = any(self.palace.is_precipice(cell) for cell in adjacent_cells)
    smelly = monster_position in adjacent_cells
    glowing = self.position == exit_position or exit_position in adjacent_cells
    wall_up = self.position[0] == 0
    wall_down = self.position[0] == self.palace.dimension - 1
    wall_left = self.position[1] == 0
    wall_right = self.position[1] == self.palace.dimension - 1
    shout = False
    found_kurtz = self.kurtz_found

    return [
        breezy,
        smelly,
        glowing,
        wall_up,
        wall_down,
        wall_left,
        wall_right,
        shout,
        found_kurtz,
    ]
```

Figura 9: class CaptainWillard

2.1.5. Agente lógico guiado por el usuario

En primer lugar hay que comentar que este agente se podría haber resuelto mediante librerías de python no estándar, como PicoSat. Así, a través de unas variables proposicionales, podríamos haber ido infiriendo para determinar las posibles ubicaciones de los objetos clave del juego.

A continuación se muestra un ejemplo de cómo se traduciría que hay brisa en la celda (0,0), y su implicación, ya que en ese caso es consecuencia lógica que haya un precipicio en las celdas (0,1) o (1,0). Las cláusulas se representan como conjunciones de disyunciones de Formas Normales Conjutivas. (CNF)

$$
\begin{aligned}
&B(0,0) \leftrightarrow (P(0,1) \vee P(1,0)) \\
&(B(0,0) \rightarrow (P(0,1) \vee P(1,0))) \wedge ((P(0,1) \vee P(1,0)) \rightarrow B(0,0)) \\
&(\neg B(0,0) \vee P(0,1) \vee P(1,0)) \wedge ((\neg P(0,1) \wedge \neg P(1,0)) \vee B(0,0)) \\
&(\neg B(0,0) \vee P(0,1) \vee P(1,0)) \wedge (\neg P(0,1) \vee B(0,0)) \wedge (\neg P(1,0) \vee B(0,0))
\end{aligned}
$$

Sin embargo finalmente se ha decidido implementar una clase `LogicAgent` de autoría propia, que es un agente lógico que se utiliza para la toma de decisiones en este entorno peligroso donde hay precipicios, monstruos y la presencia de un coronel. En la siguiente subsección se explicará y se mostrará el código. Para ir guardando celdas seguras, celdas donde puede estar el monstruo, etc. se han usado tuplas para evitar repetidos (además el orden no importa).

Ejemplo visual

Mostremos ahora un ejemplo del juego siendo un poco “tramposos”, con todos los elementos indicados, para demostrar el correcto funcionamiento del agente lógico (que es, en realidad, el propio capitán).

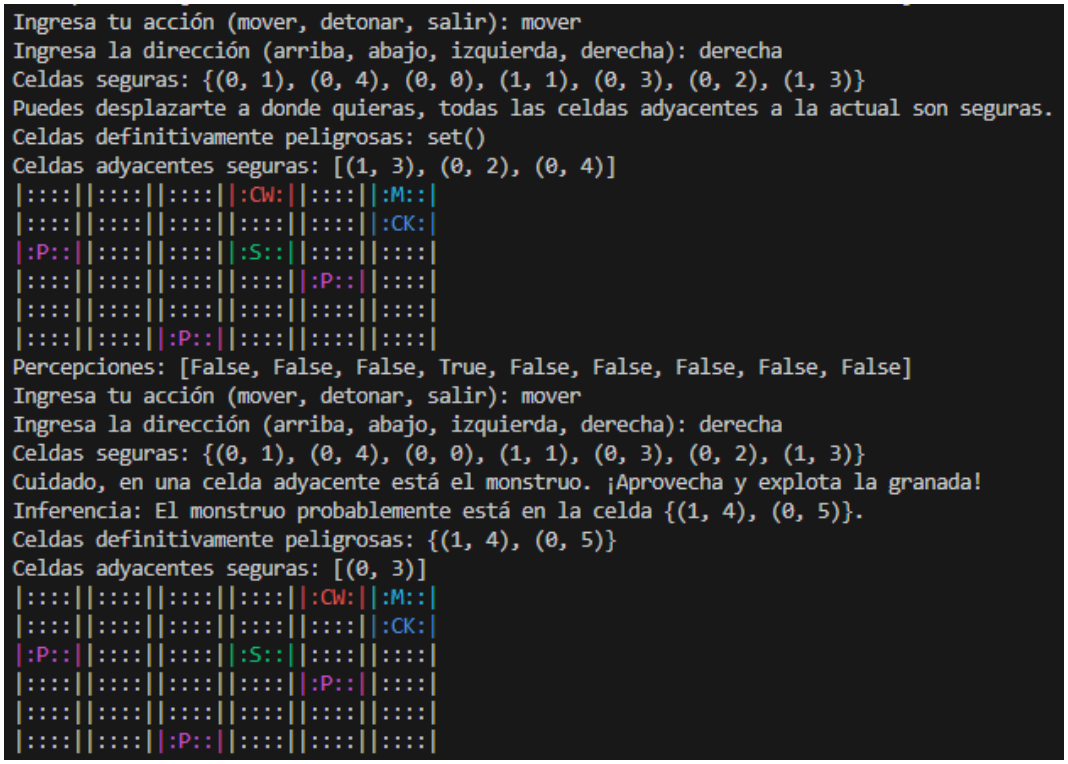


Figura 10: Ejemplo de juego

En este ejemplo donde ya hemos avanzado hasta la celda (0,3), se nos muestran las celdas seguras (hasta el momento, que se corresponden con las celdas ya exploradas) y las celdas adyacentes seguras (en este caso todas las adyacentes).

Si nos desplazamos a la derecha y nos encontramos en la celda (0,4), se puede observar que estamos al lado del monstruo, que se encuentra en la celda (0,5). En este caso se nos recomiento que explotemos la granada. Puesto que ya hemos explorado la celda (0,3), no se añade a las celdas posibles del monstruo, ya que nuestro agente infiere que ahí no puede estar. Siguiendo la recomendación, detonaríamos la granada y derrotaríamos al monstruo.

Cabe destacar que este agente lógico únicamente aconseja al jugador sobre las celdas seguras o exploradas, las celdas donde pueden estar los precipicios o el monstruo, las celdas definitivamente peligrosas (se van acumulando a lo largo de la partida, son aquellos donde se infiere que hay algún precipicio o el monstruo) y las celdas adyacentes seguras. No obstante, el usuario es libre de moverse por el mapa sin seguir las recomendaciones del agente lógico.

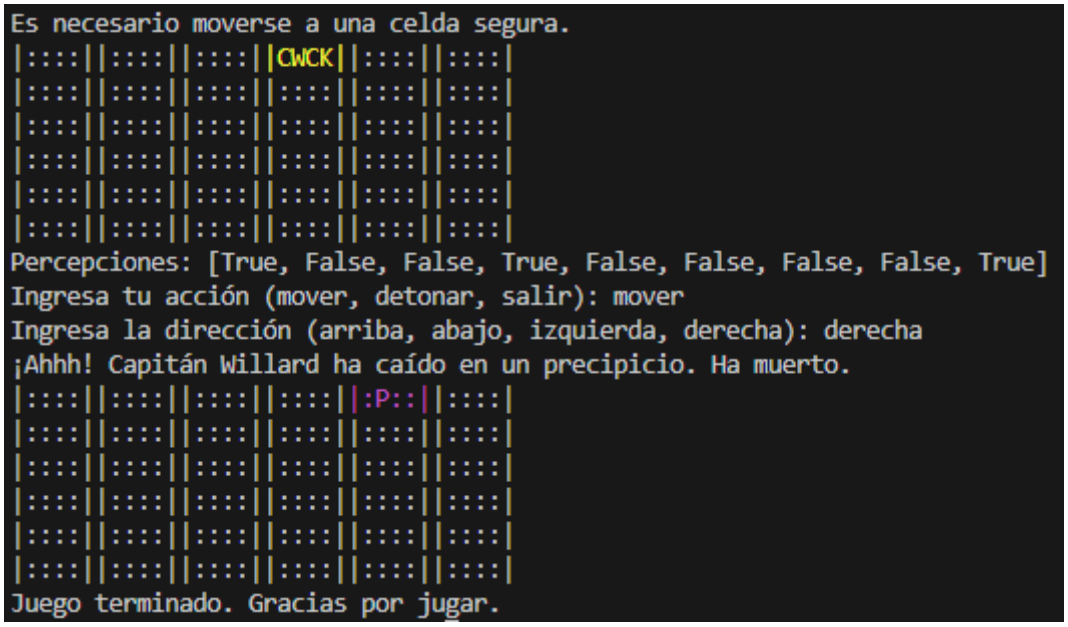


Figura 11: Otro ejemplo de juego

En este caso, comentaremos dos cosas:

La primera es que el juego definitivo tiene este aspecto: el Capitán Willard se va moviendo por el mapa y la celda donde se encuentra es de color rojo (únicamente se puede visualizar dónde se encuentra el capitán). Una vez que encuentra al Coronel Kurtz, pasa a ser de color amarillo. En caso de que muera porque cae en un precipicio, se colorea la celda donde ha muerto de color morado. Si muere devorado por el monstruo, se colorea de azul. Se ha programado el main de forma que el jugador puede ver tanto el palacio completo como el palacio mostrando únicamente la celda donde se encuentra el capitán. Esto se debe a que así se puede comprobar fácilmente el correcto funcionamiento de los agentes. Para jugar en modo "jugador", basta con eliminar las llamadas a la función display player del main.

En segundo lugar, se puede observar que el agente lógico nos comenta la necesidad de movernos a una celda segura, ya que en caso contrario puede que perdamos la partida. No le hemos hecho caso y por lo tanto el Capitán Willard ha caído en un precipicio, ya que como se ha comentado, el jugador es el que decide la acción que va a tomar.

2.1.6. Implementación en Python del agente lógico

Empecemos comentando los atributos del Agente Lógico:

- `secure_cells`: Un conjunto que representa las celdas seguras. Inicialmente, contiene solo la celda inicial (0, 0).
- `knowledge_base`: Un diccionario que almacena información sobre la ubicación probable de precipicios, monstruos y el coronel. Las claves son strings ("Precipicios", "Monstruo", "Coronel") y los valores son conjuntos de celdas. Se corresponde con la KB (base de conocimientos) del agente.
- `capitan`: Un objeto que representa al capitán en el juego. El capitán tiene métodos para obtener percepciones, celdas adyacentes y su posición actual.

Además, nuestro agente cuenta con los siguientes métodos:

- `__init__(self, capitan)`: Constructor de la clase que inicializa los atributos con valores iniciales. Recibe un parámetro `capitan`, que es un objeto que representa al capitán.
- `update_kb(self, cell, perception)`: Actualiza la base de conocimientos basándose en las percepciones del capitán. Actualiza la ubicación probable de precipicios, monstruos y el coronel.
- `is_safe_cell(self, cell)`: Verifica si una celda es segura, es decir, no es peligrosa (no hay precipicio ni monstruo).
- `most_probable_precipice_location(self)`: Devuelve las ubicaciones más probables de precipicios según la base de conocimientos.
- `most_probable_monster_location(self)`: Devuelve las ubicaciones más probables del monstruo según la base de conocimientos.
- `definitely_dangerous_cells(self)`: Devuelve las celdas que son definitivamente peligrosas, es decir, contienen precipicios o monstruos.
- `logic_agent(self)`: Proporciona recomendaciones lógicas basadas en percepciones y la base de conocimientos. Imprime información sobre celdas seguras, ubicación probable del coronel, posibilidad de movimiento seguro, advertencias sobre precipicios o monstruos, etc.

Finalmente vamos a explicar el funcionamiento del Agente Lógico.

Para usar esta clase, se crea una instancia de `LogicAgent` pasando un objeto `capitan` como parámetro. Luego, se invoca el método `logic_agent()` para obtener recomendaciones lógicas basadas en la información actualizada de la base de conocimientos y las percepciones del capitán.

Este diseño modular facilita la extensión y modificación del comportamiento del agente lógico en un entorno peligroso.

```python
class LogicAgent:
    """
    Class that represents a logical agent
    """

    def __init__(self, capitan):
        # Initialize the logical agent with secure cells and a knowledge base
        self.secure_cells = {(0, 0)}
        self.knowledge_base = {"Precipicios": set(), "Monstruo": set(), "Coronel": set()}
        self.capitan = capitan

    def update_kb(self, cell, perception):
        # Update the knowledge base based on perceptions
        if perception[0]:
            possible_prec = self.capitan.get_adjacent_cells()
            for precipice_cell in possible_prec:
                if (
                    precipice_cell not in self.secure_cells
                    and precipice_cell != self.capitan.position
                ):
                    self.knowledge_base["Precipicios"].add(precipice_cell)
        elif perception[1]:
            possible_monster = self.capitan.get_adjacent_cells()
            for monster_cell in possible_monster:
                if (
                    monster_cell not in self.secure_cells
                    and monster_cell != self.capitan.position
                ):
                    self.knowledge_base["Monstruo"].add(monster_cell)
        elif not (perception[0] or perception[1]):
            if cell not in self.secure_cells:
                self.secure_cells.add(cell)
                adjacent_secure_cells = self.capitan.get_adjacent_cells()
                self.secure_cells.update(adjacent_secure_cells)
        elif perception[8]:
            self.knowledge_base["Coronel"].add(cell)

    def is_safe_cell(self, cell):
        # Check if a cell is safe (not dangerous)
        return cell in self.secure_cells

    def most_probable_precipice_location(self):
        # Get the most probable locations of precipices
        return self.knowledge_base["Precipicios"]

    def most_probable_monster_location(self):
        # Get the most probable locations of the monster
        return self.knowledge_base["Monstruo"]

    def definitely_dangerous_cells(self):
        # Get cells that are definitely dangerous (precipice or monster)
        all_dangerous_cells = self.knowledge_base["Precipicios"].union(
            self.knowledge_base["Monstruo"]
        )
        return all_dangerous_cells - self.secure_cells
```

Figura 12: class LogicAgent

```python
def logic_agent(self):
    # Provide logical recommendations based on perceptions and knowledge base
    perceptions = self.capitan.get_perception()
    current_cell = self.capitan.position

    self.update_kb(current_cell, perceptions)

    print(f"Celdas seguras: {self.secure_cells}")
    if self.knowledge_base["Coronel"]:
        print(f"Se ha inferido que el coronel está en: {self.knowledge_base['Coronel']}")

    if not (perceptions[0] or perceptions[1]):
        print(
            "Puedes desplazarte a donde quieras, todas las celdas adyacentes a la actual son seguras."
        )

    if perceptions[0]:
        print(f"Cuidado, en una celda adyacente hay un precipicio.")
        print(
            f"Inferencia: El precipicio probablemente está en la celda {self.most_probable_precipice_location()}."
        )

    if perceptions[1]:
        print(f"Cuidado, en una celda adyacente está el monstruo. ¡Aprovecha y explota la granada!")
        print(
            f"Inferencia: El monstruo probablemente está en la celda {self.most_probable_monster_location()}."
        )

    print(f"Celdas definitivamente peligrosas: {self.definitely_dangerous_cells()}")

    if self.capitan.position not in self.secure_cells:
        print("Es necesario moverse a una celda segura.")
    else:
        adjacent_cells = self.capitan.get_adjacent_cells()
        safe_adjacent_cells = [
            cell for cell in adjacent_cells if self.is_safe_cell(cell)
        ]
        print(f"Celdas adyacentes seguras: {safe_adjacent_cells}")
```

Figura 13: class LogicAgent

Se muestra un último ejemplo del juego. El agente lógico ha actualizado sus celdas seguras e inferido que la celda (3,0) es definitivamente peligrosa (hay un precipicio). Al llegar a la celda de salida tras haber rescatado al coronel y realizar la acción de salir, se nos felicita por haber ganado y termina el juego.

```
Percepciones: [False, False, True, True, False, False, False, False, True]
Ingresa tu acción (mover, detonar, salir): salir
¡Felicidades! Has completado tu misión y has encontrado a Kurtz. Has ganado.
Celdas seguras: {(0, 1), (1, 2), (2, 1), (0, 0), (3, 1), (1, 1), (5, 3), (2, 0), (5, 1), (4, 2), (0, 2), (3, 3), (2, 2), (1, 0), (3, 2), (1, 3), (5, 2)}
Puedes desplazarte a donde quieras, todas las celdas adyacentes a la actual son seguras.
Celdas definitivamente peligrosas: {(3, 0)}
Celdas adyacentes seguras: [(1, 1), (0, 0), (0, 2)]
Juego terminado. Gracias por jugar.
```

Figura 14: Últimos movimientos del juego

2.2. Agente lógico bayesiano

2.2.1. Entorno

En este caso el palacio donde se encuentra el Coronel Kurtz sigue siendo una red de 6 salas interconectadas y se trata de un entorno parcialmente observable. No obstante, el Capitán Willard recuerda que antes de entrar en el palacio le avisaron de una serie de detalles que hacer que el entorno varíe ligeramente con respecto al apartado anterior.

- Ciertas salas siguen conteniendo tres trampas, aunque se identifican de manera distinta. Hay una trampa de fuego (F), otra con pinchos (P) y otra con dardos venenosos (D). Cada una tiene un estímulo o percepto distinto asociado. Además, en este caso pueden estar unas junto a otras, incluso todas en la misma posición. Así, para cada trampa, hay cuatro combinaciones diferentes. Por ejemplo, la trampa de fuego puede ser única en una celda, pero también puede estar junto con la trampa de pinchos, junto con la de dardos, o las tres trampas en la misma posición.

- En una de las salas, sigue habiendo un monstruo que puede acabar con el Capitán Willard, pero ahora puede estar en la misma celda que el Coronel Kurtz o que la salida. Por lo tanto para cada uno de estos tres elementos vuelve a haber cuatro combinaciones posibles.

Cabe destacar que en este caso cualquier elemento puede estar en cualquier celda (excepto el Coronel Willard, que siempre comienza en (0,0), y puede haber más de un elemento en una misma celda como ya se ha explicado. No obstante, no puede haber precipicios en las casillas (0,1) y (1,0) simultáneamente, ya que en este caso al comenzar el juego nuestro capitán no tendría posibilidad alguna de salvarse.

A continuación, se observa una configuración aleatoria del estado inicial del palacio, con 36 habitaciones en una disposición cuadrada de 6×6.

Figura 15: Configuración inicial posible

En la figura superior se puede observar que en una misma celda se encuentran las trampas de pinchos y dardos, lo cual no podía pasar en el entorno anterior asociado al agente lógico.

2.2.2. Acciones

El Capitán Willard puede las mismas acciones:

1. Moverse: Puede moverse arriba, derecha, izquierda o abajo, evitando obstáculos y peligros. Si el capitán entra en una sala que contiene un trampa, se verá abocado a una muerte dolorosa, tanto si va solo como si va en compañía del Col. Kurtz. Si el capitán entra en la sala del monstruo, y este está vivo, será devorado por él inmediatamente.

2. Detonar: Como arma, el Capitán Willard porta una cerbatana con un dardo somnífeo de un solo tiro. Tiene que tener cuidado y detonarla cuando la probabilidad de que el monstruo esté en una celda adyacente sea muy alta, ya que si por ejemplo el monstruo se encuentra en la misma celda que el Coronel Kurtz y hemos malgastado el dardo, el capitán no tiene posibilidad alguna de salvarse.

3. Salir: Si se encuentra en la celda de salida, el capitán puede ejecutar esta acción para abandonar el palacio y completar la misión. Si se ejecuta la acción de salir en cualquier otra celda, el estado del mundo se mantendrá como estaba.

Para realizar las acciones hay que introducir la acción deseada de la misma forma que antes.

2.2.3. Perceptos

Por último antes de estudiar nuestro agente lógico bayeasiano, vamos a indicar los nuevos perceptos disponibles. En este caso tendremos los mismos estímulos que antes con una única diferencia: en lugar de tener un percepto asociado a un precipicio, tenemos tres estímulos diferentes para cada trampa. Si nos encontramos en una celda adyacente a la que tiene la trampa de fuego, percibiremos olor a queroseno; si nos encontramos en una celda adyacente a la que tiene la trampa de pinchos, notaremos el suelo crujir y si nos encontramos en una celda adyacente a la que tiene la trampa de dardos, observaremos unos cables en el suelo.

2.2.4. Implementación en Python del entorno, las acciones y los perceptos

Una vez explicados el entorno de nuestro palacio y las acciones y los perceptos de nuestro capitán en esta segunda forma de jugar, se muestra cómo se ha implementado lo descrito a través del lenguaje de programación de Python.

Para ello se han seguido utilizando las tres clases de antes: class Room, class Palace y class CaptainWillard, con una serie de modificaciones. Puesto que Class Room no varía, mostremos las restantes.

En la Class Palace, definimos nuevas celdas en el constructor, ya que hay muchas celdas posibles nuevas al poder tener hasta tres elementos en la misma celda. Esta clase cuenta con los métodos siguientes: generar el palacio, conseguir posiciones aleatorias, mostrar el palacio, encontrar la posición de un elemento dado y otras funciones auxiliares que permiten saber si en una celda concreta se encuentra algún precipicio (o trampa), el monstruo, el coronel, la salida, o si es segura (es decir, no están ni el monstruo ni las trampa)

```python
class Palace:
    """
    Class that represents the game map (palace)
    """

    def __init__(self, dimension):
        # Constructor for Palace class
        self.dimension = dimension

        # Define different types of rooms with specific elements and colors
        self.base_cell = Room("|::::|", "", (0, 0))
        self.entry_cell = Room("|::::|", Fore.YELLOW, (0, 0))
        self.exit_cell = Room("|:S::|", Fore.GREEN, (0, 0))
        self.CW_cell = Room("|:CW:|", Fore.RED, (0, 0))
        self.D_cell = Room("|:D::|", Fore.MAGENTA, (0, 0))
        self.F_cell = Room("|:F::|", Fore.MAGENTA, (0, 0))
        self.P_cell = Room("|:P::|", Fore.MAGENTA, (0, 0))
        self.monster_cell = Room("|:M::|", Fore.LIGHTCYAN_EX, (0, 0))
        self.CK_cell = Room("|:CK:|", Fore.BLUE, (0, 0))

        self.exit_monster_cell = Room("|:SM:|", Fore.LIGHTCYAN_EX, (0, 0))
        self.exit_ck_cell = Room("|SCK:|", Fore.BLUE, (0, 0))
        self.monster_ck_cell = Room("|MCK:|", Fore.BLUE, (0, 0))
        self.exit_monster_ck_cell = Room("|SCKM|", Fore.BLUE, (0, 0))

        self.F_D_cell = Room("|:FD:|", Fore.MAGENTA, (0, 0))
        self.F_P_cell = Room("|:FP:|", Fore.MAGENTA, (0, 0))
        self.D_P_cell = Room("|:DP:|", Fore.MAGENTA, (0, 0))
        self.F_D_P_cell = Room("|FDP:|", Fore.MAGENTA, (0, 0))

        # Initialize the game board with the base cell and generate the map
        self.board = [[self.base_cell] * self.dimension for _ in range(self.dimension)]

        # Generate the map with entry, exit, and special elements
        self.generate_map()
```

Figura 16: class Palace

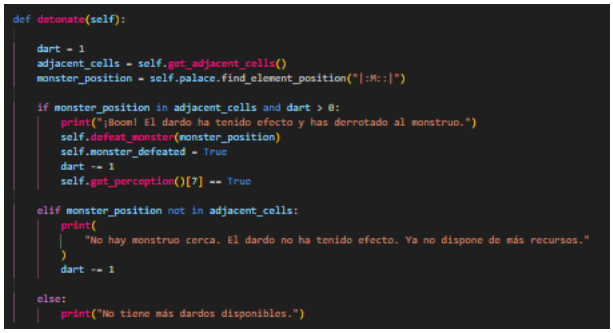
```python
def generate_map(self):
    # Method to generate the game map by placing elements on the board
    entry_x, entry_y = 0, 0
    self.board[entry_x][entry_y] = self.entry_cell

    self.board[0][0] = self.CW_cell

    f_x, f_y = self.get_random_position()
    self.board[f_x][f_y] = self.F_cell

    d_x, d_y = self.get_random_position()
    self.board[d_x][d_y] = self.D_cell

    p_x, p_y = self.get_random_position()
    self.board[p_x][p_y] = self.P_cell

    if (f_x, f_y) == (d_x, d_y) and (f_x, f_y) == (p_x, p_y):
        self.board[f_x][f_y] = self.F_D_P_cell

    elif (f_x, f_y) == (d_x, d_y):
        self.board[f_x][f_y] = self.F_D_cell

    elif (f_x, f_y) == (p_x, p_y):
        self.board[f_x][f_y] = self.F_P_cell

    elif (d_x, d_y) == (p_x, p_y):
        self.board[d_x][d_y] = self.D_P_cell

    exit_x, exit_y = self.get_second_random_position()
    self.board[exit_x][exit_y] = self.exit_cell

    monster_x, monster_y = self.get_second_random_position()
    self.board[monster_x][monster_y] = self.monster_cell

    ck_x, ck_y = self.get_second_random_position()
    self.board[ck_x][ck_y] = self.CK_cell

    if (exit_x, exit_y) == (monster_x, monster_y) and (exit_x, exit_y) == (
        ck_x,
        ck_y,
    ):
        self.board[monster_x][monster_y] = self.exit_monster_ck_cell

    elif (exit_x, exit_y) == (monster_x, monster_y):
        self.board[monster_x][monster_y] = self.exit_monster_cell

    elif (exit_x, exit_y) == (ck_x, ck_y):
        self.board[exit_x][exit_y] = self.exit_ck_cell

    elif (monster_x, monster_y) == (ck_x, ck_y):
        self.board[monster_x][monster_y] = self.monster_ck_cell
```

Figura 17: class Palace

```python
def get_random_position(self):
    # Helper method to get a random position on the board
    while True:
        x, y = random.randint(0, self.dimension - 1), random.randint(
            0, self.dimension - 1
        )
        if (x, y) != (0, 0) and (x, y) != (1, 0) and (x, y) != (0, 1):
            return x, y

def get_second_random_position(self):
    # Helper method to get a second random position on the board
    while True:
        x, y = random.randint(0, self.dimension - 1), random.randint(
            0, self.dimension - 1
        )
        if (x, y) != (0, 0) and (
            self.board[x][y] != self.F_D_cell
            and self.board[x][y] != self.F_P_cell
            and self.board[x][y] != self.D_P_cell
            and self.board[x][y] != self.F_D_P_cell
            and self.board[x][y] != self.F_cell
            and self.board[x][y] != self.P_cell
            and self.board[x][y] != self.D_cell
        ):
            return x, y

def display_palace(self):
    print("CONFIGURACIÓN ALEATORIA DEL ESTADO INICIAL DEL PALACIO")
    # Print the current state of the palace board
    for row in self.board:
        print("".join(str(room) for room in row))

def find_element_position(self, element):
    # Method to find the position of a specific element on the board
    for x in range(self.dimension):
        for y in range(self.dimension):
            if self.board[x][y].element == element:
                return x, y
    return None

# Various methods to check the type of a room at a given position
def is_precipice(self, position):
    x, y = position
    return (
        self.board[x][y].element == "|:F::|"
        or self.board[x][y].element == "|:FD:|"
        or self.board[x][y].element == "|:DP:|"
        or self.board[x][y].element == "|FDP:|"
        or self.board[x][y].element == "|:FP:|"
        or self.board[x][y].element == "|:P::|"
        and self.board[x][y].element == "|:D::|"
    )
```

Figura 18: class Palace

En la Class CaptainWillard solo se han cambiado dos funciones con respecto al ejercicio anterior, la de detonar (ya que ahora hay un único dardo) y la de obtener los perceptos (ahora hay más y están asociados a más celdas por poder tener varios elementos en una misma celda).



```
def detonate(self):

    dart = 1
    adjacent_cells = self.get_adjacent_cells()
    monster_position = self.palace.find_element_position("|:M::|")

    if monster_position in adjacent_cells and dart > 0:
        print("¡Boom! El dardo ha tenido efecto y has derrotado al monstruo.")
        self.defeat_monster(monster_position)
        self.monster_defeated = True
        dart -= 1
        self.get_perception()[7] == True

    elif monster_position not in adjacent_cells:
        print(
            "No hay monstruo cerca. El dardo no ha tenido efecto. Ya no dispone de más recursos."
        )
        dart -= 1

    else:
        print("No tiene más dardos disponibles.")
```

Figura 19: class CaptainWillard

```
# Captain Willard's perceptions

def get_perception(self):
    adjacent_cells = self.get_adjacent_cells()

    fire_positions = [
        self.palace.find_element_position("|:F::|"),
        self.palace.find_element_position("|:FD:|"),
        self.palace.find_element_position("|:FP:|"),
        self.palace.find_element_position("|FDP:|"),
    ]
    spikes_positions = [
        self.palace.find_element_position("|:P::|"),
        self.palace.find_element_position("|:FP:|"),
        self.palace.find_element_position("|:DP:|"),
        self.palace.find_element_position("|FDP:|"),
    ]
    darts_positions = [
        self.palace.find_element_position("|:D::|"),
        self.palace.find_element_position("|:FD:|"),
        self.palace.find_element_position("|:DP:|"),
        self.palace.find_element_position("|FDP:|"),
    ]

    monster_positions = [
        self.palace.find_element_position("|:M::|"),
        self.palace.find_element_position("|:SM:|"),
        self.palace.find_element_position("|MCK:|"),
        self.palace.find_element_position("|SCKM|"),
    ]

    exit_positions = [
        self.palace.find_element_position("|:S::|"),
        self.palace.find_element_position("|:SM:|"),
        self.palace.find_element_position("|SCK:|"),
        self.palace.find_element_position("|SCKM|"),
    ]

    fire = any(position in adjacent_cells for position in fire_positions)
    spikes = any(position in adjacent_cells for position in spikes_positions)
    darts = any(position in adjacent_cells for position in darts_positions)
    smelly = any(position in adjacent_cells for position in monster_positions)
    glowing = (
        any(position in adjacent_cells for position in exit_positions)
        or self.position in exit_positions
    )
    wall_up = self.position[0] == 0
    wall_down = self.position[0] == self.palace.dimension - 1
    wall_left = self.position[1] == 0
    wall_right = self.position[1] == self.palace.dimension - 1
    shout = False
    found_kurtz = self.kurtz_found

    return [
        fire,
        spikes,
        darts,
        smelly,
        glowing,
        wall_up,
        wall_down,
        wall_left,
        wall_right,
        shout,
        found_kurtz,
    ]
```

Figura 20: class CaptainWillard

2.2.5. Agente lógico bayesiano guiado por el usuario

En esta ocasión el Capitán Willard puede tomar riesgos que le permitan resolver el problema accediendo a salas cuyo nivel de riesgo sea bajo.

Tratando de comprender de forma más explícita su situación, el Capitán decide usar inicialmente un prior uniforme para representar su falta de conocimiento sobre la posición de cualquiera de los elementos del palacio. Podemos poner esta creencia previa en una expresión compacta para detallar el prior (o distribución a priori) de la posición de los diferentes elementos. En concreto, para cada elemento τ (usaremos este símbolo para representar los distintos elementos, es decir, M , F , P , D, S o CK) y para cada posición (i, j) en el palacio, el cual cuenta con un total de N celdas (para este caso, $N = 6 \times 6$), tendremos que el prior es

$$p(\tau_{ij}) = \frac{1}{N-1},$$

donde el $N - 1$ viene dado de que el Capitán ha entrado ya en el palacio, visitando la primera celda.

Teniendo en cuenta la información que conocemos acerca de los estímulos, el modelo de verosimilitud será el siguiente:

$$p(e_{\tau_{ij}}|\tau_{k\ell}) = \begin{cases} 1 & \text{si } (k,\ell) \in \text{adj}(i,j) \cup \{(i,j)\}, \\ 0 & \text{en otro caso.} \end{cases}$$

donde (i, j) y (k, ℓ) representan posiciones en el palacio, y $\text{adj}(i, j)$ señala las celdas adyacentes a la celda (i, j) (por ejemplo, $\text{adj}(0, 0) = \{(0, 1), (1, 0)\}$). Es decir, la probabilidad de recibir el estímulo asociado a cada elemento será 1 si estamos en una celda adyacente al origen del mismo, y 0 para el resto de las celdas (por ejemplo, el Capitán solo percibirá el olor a queroseno si está al lado de la trampa de fuego, nunca en otro lado). Además, la propia celda de cada elemento presentará el estímulo también.

Teniendo esto en cuenta, el capitán va recorriendo el palacio actualizando sus creencias gracias a la regla de Bayes

$$p(\tau_{k\ell}|e_{\tau_{ij}}) = p(e_{\tau_{ij}}|\tau_{k\ell}) \cdot p(\tau_{k\ell}) \cdot \frac{1}{p(e_{\tau_{ij}})}.$$

Como ejemplo, imaginemos que en la celda $(0, 0)$ no hemos percibido que el suelo cruja, y por lo tanto sabemos que no hemos percibido el estímulo de la trampa de pinchos, lo cual, según la notación anterior, podemos escribir como $\neg e_{P_{00}}$, usando $e_{P_{ij}}$ para denotar el estímulo de pinchos (P) en la celda (i, j). Usando toda esta información, el posterior sobre las posiciones de la trampa de pinchos asociado a esta ausencia de estímulo en la celda $(0, 0)$ es:

$$P(p_{k\ell}|\neg e_{P_{00}}) = P(\neg e_{P_{00}}|p_{k\ell}) \cdot P(p_{k\ell}) \cdot \frac{1}{P(\neg e_{P_{00}})} = \frac{1}{N-3},$$

para todas las celdas que no sean la $(0, 0)$ y sus celdas adyacentes.

Ahora vamos a mostrar un ejemplo de la interfaz asociada al juego de este agente para adentrarnos en el siguiente apartado, donde vamos a explicar detalladamente qué cuentas debe hacer nuestro agente lógico bayesiano para ir calculando mapas de probabilidad. Es importante destacar que en la versión jugador no se mostraría el palacio, como ya se ha explicado antes.

En nuestro caso se ha decidido representar mapas de probabilidades que tienen asociados los cálculos al realizar el posterior sobre los elementos siguientes: fuego, pinchos, dardos, monstruo y salida. El coronel se encuentra en la posición (0,0) y no percibe ningún estímulo asociado a los elementos anteriores, por lo que la probabilidad de que haya algún elemento en las celdas adyacentes es 0, y para el resto de celdas la probabilidad pasa a ser $\frac{1}{33}$.

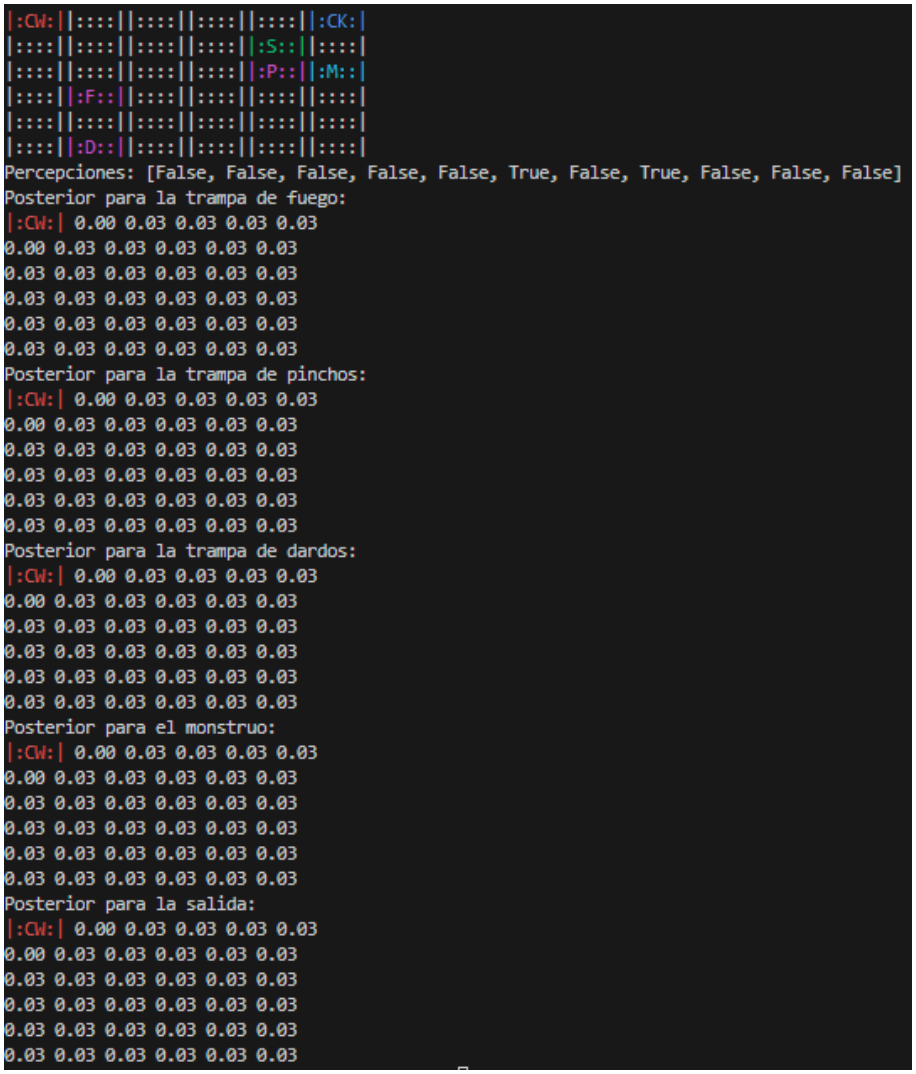


Figura 21: class CaptainWillard

Como podemos observar, hay una trampa de fuego en (3,1). Para ver cómo reacciona nuestro agente bayesiano, nos desplazamos a la posición (2,1).

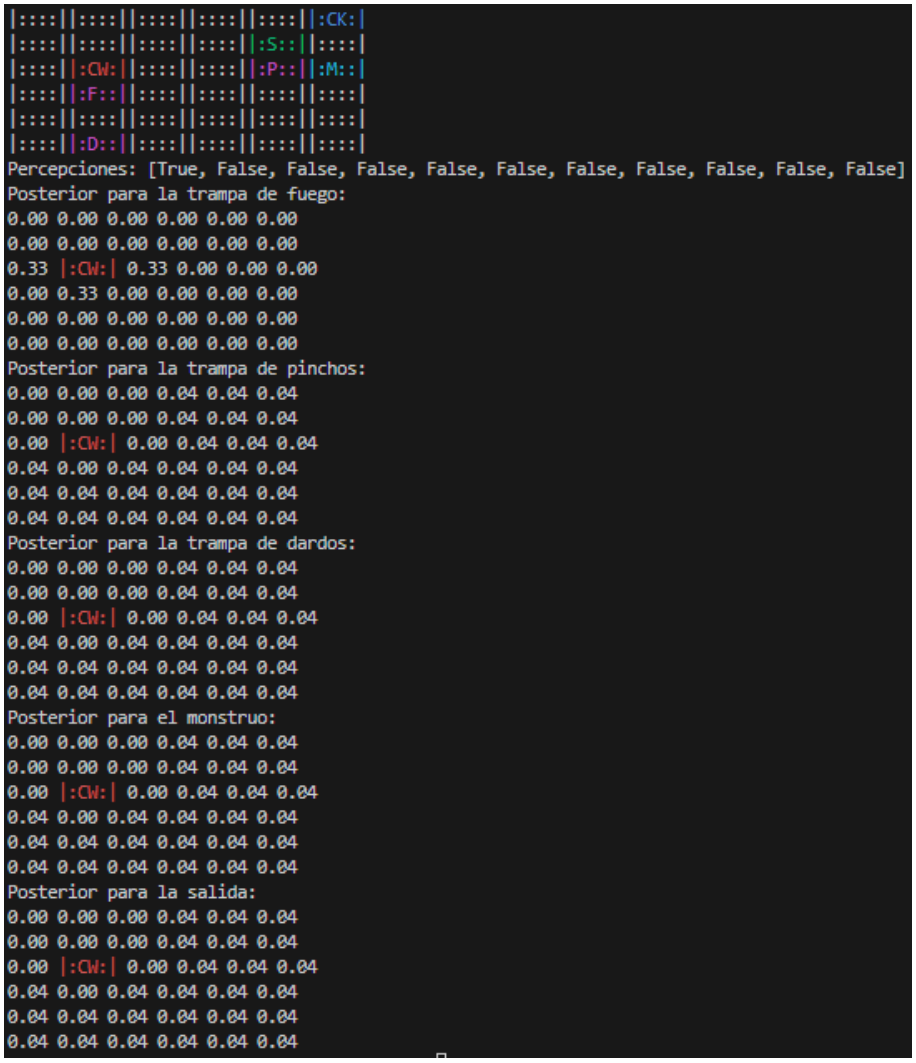


Figura 22: class CaptainWillard

Como se puede observar, se han ido actualizando todos los mapas. Puesto que el agente ha percibido el estímulo asociado al fuego, sus casillas adyacentes tienen una probabilidad de 0.33, y el resto 0.00. Para el resto de elementos ahora hay varias casillas con probabilidad 0.00 (concretamente 10), y el resto tienen una probabilidad asociada de $\frac{1}{26}$.

¿Pero cómo se van actualizando las probabilidades con cada acción?

2.2.6. Implementación en Python del agente lógico bayesiano

Vamos a explicar cúal es la lógica de nuestro agente bayesiano y cómo va infiriendo las probabilidades en cada caso. Puesto que usa la misma lógica, vamos a ejemplificarlo con la trampa de fuego.

El estímulo asociado al fuego está en la primera posición de la lista de perceptos, luego si recibimos que es False (no hay fuego en una celda adyacente), el agente bayesiano inferirá de la siguiente forma. Añadirá una probabilidad de 0.00 de que el fuego esté en su casilla o en una casilla adyacente. Para el resto de casillas, la probabilidad de que el fuego esté en esa posición se calcula como sigue,

$$P(F_{ij}) = \frac{1}{N - E},$$

Donde N representa el número de celdas (en nuestro caso 36) y E representa la suma del número de celdas exploradas y las adyacentes de las exploradas. Para el primer ejemplo del mapa hemos visto que teníamos una celda explorada (0,0) y sus adyacentes son (1,0) y (0,1). Para ese caso,

$$P(F_{ij}) = \frac{1}{36 - 3} = \frac{1}{33} = 0{,}03,$$

como ya se ha comprobado.

La otra posibilidad es que percibamos True, y en este caso la implementación es muy sencilla. Ningún elemento está repetido. Por ejemplo, no puede ser que haya fuego en la celda (2,4) y en la (5,1) en una misma partida. Por lo tanto, cuando percibamos el percepto asociado al fuego sabremos que todas las celdas que no sean adyacentes tienen una probabilidad e 0.00 de contener la trampa de fuego. Ahora bien, para calcular las probabilidades de las adyacentes, si son nulas, no hay que cambiar nada. En caso de no ser nulas, la probabilidad se calcula como

$$P(F_{ij}) = \frac{1}{A - Z},$$

Donde A representa el número de adyacentes (en el ejemplo de la página 26, sería 4) y Z representa el número de celdas adyacentes nulas (en el ejemplo de la página 26, sería 1, ya que la celda superior se ha explorado y tiene probabidad 0.00 de tener fuego). Para este caso,

$$P(F_{ij}) = \frac{1}{4 - 1} = \frac{1}{3} = 0{,}33$$

Por último, se puede observar que para el resto de elementos las probabilidades no nulas tras haber visitado las celdas (0,0), (0,1), (1,1), y estando en (2,1); es decir, moviéndonos a la derecha y luego dos veces seguidas hacia abajo, es de 0.04. Esto se debe a que al aplicar la fórmula,

$$P(F_{ij}) = \frac{1}{36 - 10} = \frac{1}{26} = 0{,}04$$

Empecemos comentando los atributos del Agente Bayesiano:

- `palace`: Representa el palacio o entorno en el que el agente opera.
- `capitan`: Objeto que simboliza al capitán del juego. Posee métodos para obtener percepciones, celdas adyacentes y su posición actual.
- `F_matrix, P_matrix, D_matrix, M_matrix, S_matrix`: Matrices que almacenan las distribuciones posteriores para la trampa de fuego, trampa de pinchos, trampa de dardos, monstruo y salida respectivamente.
- Conjuntos:

  `F_visited_cells, P_visited_cells, D_visited_cells, M_visited_cells, S_visited_cells`: Almacenan las celdas visitadas relacionadas con cada trampa o entidad.

Además, nuestro agente cuenta con los siguientes métodos:

- `__init__(self, palace, capitan)`: Constructor de la clase que inicializa los atributos. Recibe como parámetro un objeto `capitan`.
- `posterior_F(self)`, `posterior_P(self)`, ...: Métodos para calcular las distribuciones posteriores para la trampa de fuego, trampa de pinchos, trampa de dardos, monstruo y salida respectivamente.
- `display_palace_prob(self, element, position, perception, captain_position)`: Método que muestra las distribuciones posteriores para diferentes elementos en el palacio.

El agente utiliza la información de percepciones y celdas adyacentes proporcionadas por el capitán para actualizar las distribuciones posteriores de trampas y entidades en el palacio. Cada método `posterior_X()` calcula la distribución posterior para el elemento X basándose en la información actualizada. El método `display_palace_prob` imprime las distribuciones posteriores para cada elemento en el palacio.

Este diseño modular permite una fácil extensión y modificación del comportamiento del agente en un entorno peligroso, ya que las distribuciones posteriores se calculan por separado para cada elemento. Para usar la clase, se crea una instancia de `BayesianLogicAgent` pasando un objeto `capitan` como parámetro y luego se invoca el método `display_palace_prob` para visualizar las distribuciones posteriores actualizadas.

```python
class BayesianLogicAgent:
    """
    Class that represents a bayesian knowledge-based agent
    """

    def __init__(self, palace, capitan):
        self.palace = palace
        self.capitan = capitan
        self.F_matrix = [
            [None] * self.palace.dimension for _ in range(self.palace.dimension)
        ]
        self.P_matrix = [
            [None] * self.palace.dimension for _ in range(self.palace.dimension)
        ]
        self.D_matrix = [
            [None] * self.palace.dimension for _ in range(self.palace.dimension)
        ]
        self.M_matrix = [
            [None] * self.palace.dimension for _ in range(self.palace.dimension)
        ]
        self.S_matrix = [
            [None] * self.palace.dimension for _ in range(self.palace.dimension)
        ]

        self.F_visited_cells = set()
        self.P_visited_cells = set()
        self.D_visited_cells = set()
        self.M_visited_cells = set()
        self.S_visited_cells = set()

    def posterior_F(self):
        # Method to calculate the posterior distribution for the fire trap
        percepts = self.capitan.get_perception()
        adjacent_cells = self.capitan.get_adjacent_cells()
        self.F_visited_cells.update(adjacent_cells)

        if percepts[0]:
            percept = True
            adjacent_null_cells = []
            adjacent_non_null_cells = []
            for cell in adjacent_cells:
                x, y = cell[0], cell[1]
                if self.F_matrix[x][y] == 0.00:
                    adjacent_null_cells.append(cell)
                else:
                    adjacent_non_null_cells.append(cell)
            for x, row in enumerate(self.palace.board):
                for y, room in enumerate(row):
                    self.F_matrix[x][y] = 0.00
            for cell in adjacent_non_null_cells:
                x, y = cell[0], cell[1]
                self.F_matrix[x][y] = 1 / len(adjacent_non_null_cells)

        else:
            for x, row in enumerate(self.palace.board):
                for y, room in enumerate(row):
                    if (x, y) in self.F_visited_cells:
                        self.F_matrix[x][y] = 0.0
                    else:
                        self.F_matrix[x][y] = 1 / (
                            (self.palace.dimension) ** 2 - len(self.F_visited_cells)
                        )

        return self.F_matrix
```

Figura 23: class BayesianLogicAgent

```python
# Method to display the posterior distributions for different elements
def display_palace_prob(self, element, position, perception, captain_position):
    posterior_distribution_F = self.posterior_F()
    print("Posterior para la trampa de fuego:")
    for i, row in enumerate(self.palace.board):
        for j, room in enumerate(row):
            if (i, j) == captain_position:
                print(str(room), end=" ")
            else:
                print(f"{posterior_distribution_F[i][j]:.2f}", end=" ")
        print()

    posterior_distribution_P = self.posterior_P()
    print("Posterior para la trampa de pinchos:")
    for i, row in enumerate(self.palace.board):
        for j, room in enumerate(row):
            if (i, j) == captain_position:
                print(str(room), end=" ")
            else:
                print(f"{posterior_distribution_P[i][j]:.2f}", end=" ")
        print()

    posterior_distribution_D = self.posterior_D()
    print("Posterior para la trampa de dardos:")
    for i, row in enumerate(self.palace.board):
        for j, room in enumerate(row):
            if (i, j) == captain_position:
                print(str(room), end=" ")
            else:
                print(f"{posterior_distribution_D[i][j]:.2f}", end=" ")
        print()

    posterior_distribution_M = self.posterior_M()
    print("Posterior para el monstruo:")
    for i, row in enumerate(self.palace.board):
        for j, room in enumerate(row):
            if (i, j) == captain_position:
                print(str(room), end=" ")
            else:
                print(f"{posterior_distribution_M[i][j]:.2f}", end=" ")
        print()

    posterior_distribution_S = self.posterior_S()
    print("Posterior para la salida:")
    for i, row in enumerate(self.palace.board):
        for j, room in enumerate(row):
            if (i, j) == captain_position:
                print(str(room), end=" ")
            else:
                print(f"{posterior_distribution_S[i][j]:.2f}", end=" ")
        print()
```

Figura 24: class BayesianLogicAgent

2.3. Agente de búsqueda

2.3.1. Funcionamiento del algoritmo BFS

En el proyecto se ha implementado un agente buscador que utiliza el algoritmo de BFS. Si seleccionamos la opción de jugar con un agente de búsqueda, el usuario solo tiene que esperar a que el agente encuentre la solución. En esta ocasión se ha creado un mapa sencillo donde hay un palacio de dimensión 6. El Capitán Willard siempre empieza en la posición (0,0) y el Coronel Kurtz se posiciona en una casilla aleatoria de entre las 35 restantes.

Primero vamos a comentar en qué consiste este algoritmo y en el último apartado de esta memoria se explicará cómo se ha programado en Python y se mostrará el código.

La búsqueda en anchura o BFS es un algoritmo básico de búsqueda en inteligencia artificial. El objetivo es recorrer los nodos de forma creciente, además de utilizar el FIFO, que es un acrónimo que significa "primero en entrar, primero en salir". El algoritmo parte de una raíz, explora todos sus posibles descendientes y luego elige otra raíz de esos descendientes para continuar con el mismo proceso hasta encontrar una solución. A diferencia de otros algoritmos como greedy o A*, se trata de un algoritmo de búsqueda sin información, así que se exploran todos los nodos a una profundidad dada antes de pasar a la siguiente profundidad. No tiene en cuenta ninguna heurística o función de evaluación para dirigir su búsqueda.

Algunas aplicaciones prácticas son la búsqueda de rutas en un laberinto, la resolución de rompecabezas, o la navegación en mapas, aunque hay que tener en cuenta que mayor complejidad temporal y espacial en comparación con algunos algoritmos de búsqueda informada en ciertos casos.

Complejidad Espacial:

- Mejor caso: $d(b-1)$
- Caso medio: $\frac{1}{2}(b-1)$
- Peor caso: $d(b-1)$

Complejidad Temporal:

- Mejor caso: bd
- Caso medio: $\frac{1}{2}b^d$
- Peor caso: b^d

2.3.2. Implementación en Python del agente de búsqueda

En esta ocasión se nos dibujará la situación inicial del palacio, donde el capitán se encuentra en la posición (0,0) y el coronel en una celda aleatoria, en este caso está en la posición (5,4).

Una vez que el agente buscador encuentre al Coronel Kurtz, nos saldrá un mensaje comunicándonos que se ha llegado al estado objetivo y se muestra la ruta a seguir para acabar en esa posición, indicando en cada paso el estado y la acción que debe realizar.

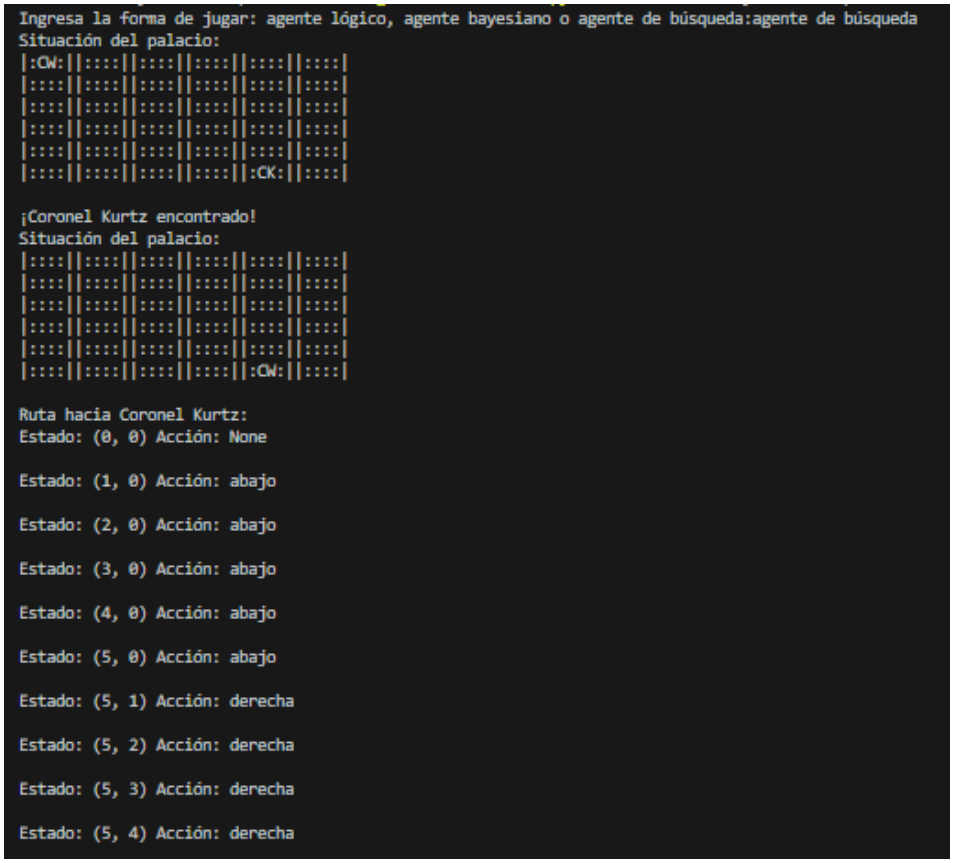


Figura 25: Ejemplo de juego

Para implementar esta forma de jugar, se han creado tres clases: class Node, class Palace y class AgentBFS. A continuación se explican los detalles más importantes de estas.

La class Node representa un estado en el espacio de búsqueda. Almacena información como el estado actual, el nodo padre, la acción tomada y el costo acumulado.

La class Environment representa el entorno bidimensional en el que se mueve el agente. Inicializa un tablero de cierta dimensión con un agente ("CW") y un objetivo "CK") en posiciones aleatorias. Proporciona métodos para imprimir el estado del entorno y actualizar la posición del agente.

Por último la clase AgentBFS representa un agente que utiliza el algoritmo de BFS para buscar una solución. Inicializa la frontera con el nodo de inicio y establece el estado objetivo. Define acciones posibles (movimientos) y genera nodos hijos basados en esas acciones. Comprueba si se ha alcanzado el estado objetivo. Proporciona métodos para imprimir la ruta de solución y extraer la ruta de solución.

El método destacado de la clase AgentBFS es solve, que realiza la búsqueda BFS para encontrar la solución. Explora nodos en la frontera, generando y agregando nodos hijos. Cuando encuentra el nodo objetivo, imprime la solución y el estado final del entorno.

```python
class Node:
    """
    Class that represents a state in the search space
    """
    def __init__(self, state, parent=None, action=None, cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost

    def __str__(self):
        return f"Estado: {self.state} Acción: {self.action}\n"


class Environment:
    """
    Class that represents the 2D grid where the agent moves
    """
    def __init__(self, dimension=6):
        # Initialize the environment with a given dimension
        self.dimension = dimension
        self.tablero = [[" "] * dimension for _ in range(dimension)]
        self.start = (0, 0)
        self.goal = (0, 0)
        self.cw = None
        self.colocar_elementos()

    def colocar_elementos(self):
        # Place the goal and agent (CW) in random positions on the grid
        self.goal = self.generar_posicion_aleatoria()
        self.start = (0, 0)
        while self.start == self.goal:
            self.goal = self.generar_posicion_aleatoria()
        self.cw = self.start
        self.tablero[self.start[0]][self.start[1]] = "CW"
        self.tablero[self.goal[0]][self.goal[1]] = "CK"

    def generar_posicion_aleatoria(self):
        # Generate a random position within the grid
        return random.randint(0, self.dimension - 1), random.randint(
            0, self.dimension - 1
        )

    def imprimir(self, solution=None, frontier=None, explored=None):
        # Print the current state of the environment
        print("Situación del palacio:")
        for i, row in enumerate(self.tablero):
            for j, col in enumerate(row):
                if col == "CK":
                    print("|:CK:|", end="")
                elif col == "CW":
                    print("|:CW:|", end="")
                else:
                    print("|::::|", end="")
            print()
        print()

    def actualizar_posicion_cw(self, nueva_posicion):
        # Update the position of the agent in the environment
        old_pos = self.cw
        self.tablero[old_pos[0]][old_pos[1]] = " "
        self.cw = nueva_posicion
        self.tablero[nueva_posicion[0]][nueva_posicion[1]] = "CW"
```

Figura 26: class Node y class Environment

```python
class AgentBFS:
    """
    Class that represents an agent using Breadth-First Search
    """
    def __init__(self, environment):
        # Initialize the agent with the starting node in the frontier
        self.frontier = [Node(state=environment.start)]
        self.explored = set()
        self.goal_state = environment.goal
        self.environment = environment

    def actions(self, node):
        # Define possible actions (movements) for the agent
        row, col = node.state
        possible_actions = [
            ("arriba", (row - 1, col)),
            ("abajo", (row + 1, col)),
            ("izquierda", (row, col - 1)),
            ("derecha", (row, col + 1)),
        ]
        return possible_actions

    def result(self, node):
        # Generate child nodes based on the agent's actions
        children = []
        height = len(self.environment.tablero)
        width = len(self.environment.tablero[0])

        for action, (r, c) in self.actions(node):
            if 0 <= r < height and 0 <= c < width:
                children.append(Node(state=(r, c), parent=node, action=action))

        return children

    def is_goal(self, node):
        # Check if the agent has reached the goal state
        return node.state == self.goal_state

    def print_solution(self, goal_node):
        # Print the solution path from the start to the goal state
        solution_path = self.extract_solution_path(goal_node)
        print("Ruta hacia Coronel Kurtz:")
        for node in solution_path:
            print(node)
        print()

    def extract_solution_path(self, goal_node):
        # Extract the solution path by traversing back through parent nodes
        path = []
        current_node = goal_node
        while current_node:
            path.insert(0, current_node)
            current_node = current_node.parent
        return path

    def solve(self):
        # Perform Breadth-First Search to find the goal state
        while self.frontier:
            current_node = self.frontier.pop(0)
            self.explored.add(current_node.state)

            if self.is_goal(current_node):
                print("¡Coronel Kurtz encontrado!")
                self.environment.actualizar_posicion_cw(current_node.state)
                self.environment.imprimir()

                self.print_solution(current_node)
                return

            for child in self.result(current_node):
                if child.state not in self.explored and child not in self.frontier:
                    self.frontier.append(child)
```

Figura 27: class BFS

Bibliografía

[1] *Búsqueda en anchura*. Disponible en `https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura`.

[2] *Inteligencia Artificial conceptos básicos, un poco de opiniones y aprendiendo TypeScript*. Disponible en `https://www.linkedin.com/pulse/inteligencia-artificial-conceptos-b%C3%A1sicos-un-poco-de-y-palma-s%C3%A1nchez/?originalSubdomain=es`.

[3] *Introducción a la inferencia Bayesiana con Python*. Disponible en `https://relopezbriega.github.io/blog/2017/05/21/introduccion-a-la-inferencia-bayesiana-con-python/`.

[4] *CS50's Introduction to Artificial Intelligence with Python*. Disponible en `https://cs50.harvard.edu/ai/2023/notes/1/`.

[5] *Using Logic to Hunt the Wumpus*. Disponible en `http://www.sista.arizona.edu/~clayton/courses/ai/projects/wumpus/`.

También han sido de gran ayuda para realizar este proyecto final los materiales utilizados en la asignatura de Fundamentos de Inteligencia Artificial del segundo curso del Grado en Ingeniería Matemática e Inteligencia Artificial de la Universidad Pontificia de Comillas. Estos se basan principalmente en las transparencias realizadas por los profesores de esta asignatura Andrés Occhipinti, Simón Rodríguez y Mario Castro.