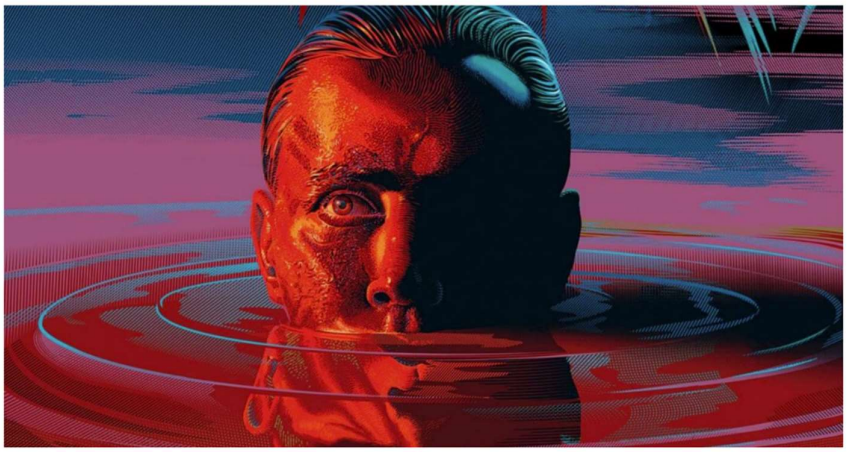
Buscando al *Coronel Kurtz*

Fundamentos de Inteligencia Artificial 23-24
Grado en *Ingeniería Matemática e Inteligencia Artificial*, ICAI, UPC



Contenido

Introducción

En la célebre película Apocalypse Now, el Capitán Willard va en búsqueda del Coronel Kurtz con la orden de encontrarlo y convencerle para que deponga su actitud y vuelva con él a Washington a rendir cuentas.

En su misión, el Capitán Willard se enfrenta a la tarea de encontrar al Coronel Kurtz dentro de un peligroso palacio, repleto de trampas letales, simas profundas e incluso monstruos (algunos de ellos, producto de su propia locura). Para seleccionar la acción en cada momento, el capitán tiene que confiar en sus sentidos (percepciones del entorno) y en las técnicas de rastreo que aprendió en el ejército (que en este caso, coincidentalmente, son las técnicas de búsqueda, lógica y probabilidad que hemos visto en clase). Comenzando en una sala inicial del palacio, sus objetivos son:

1. Evitar los obstáculos.
2. Hallar al Coronel Kurtz.
3. Dirigirse hacia la salida del palacio una vez lo encuentre.

En las secciones que siguen, detallamos los componentes del problema y la solución al problema.

Agente basado en conocimiento

Palacio del Agente basado en conocimiento

El palacio del agente basado en conocimiento consiste en una red de salas $n \times n$ de salas interconectadas. Para nuestro caso concreto, es de n = 6 salas, aunque se puede cambiar al inicializar la clase de palacio.

```python
class Palacio:
    """
    Clase de Palacio del Agente KB
    """

    def __init__(self, n: int = 6, n_precipicios=3) -> None:
        self.n = n
        self.posibles_perceptos = [
            "brisa",
            "fetido_olor",
            "resplandor",
            "pared_arriba",
            "pared_abajo",
            "pared_izquierda",
            "pared_derecha",
            "grito",
            "CK_con_CW",
        ]
        self.N_precipicios = n_precipicios

        # Se inicializa aleatoriamente las posiciones de los objetos en el palacio.
        self.shuffle()
        self.percept_indexes = {
            percept: i for i, percept in enumerate(self.posibles_perceptos)
        }
```

Figura 1. Constructor clase Palacio

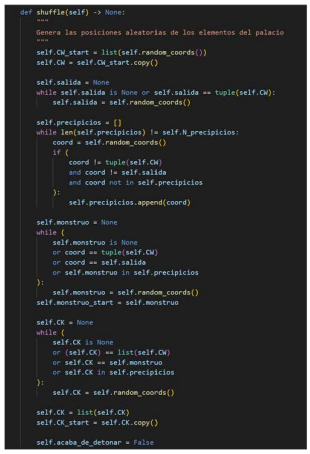
Como se puede observar en la Figura 1. Constructor clase Palacio, los perceptos en este palacio son una brisa, un fétido olor, un resplandor, pared_arriba, pared_abajo, pared_izquierda, pared_derecha, grito, CK_con_CW. Los cuales pueden tomar valores 0 o 1, ausencia y presencia de percepto respectivamente.

Los 4 primeros perceptos refieren a los distintos objetos que hay en las casillas y los percibes al estar en una casilla adyacente a alguno de estos. Percibes una brisa si estás al lado de un precipicio, un fétido olor si te encuentras al lado del monstruo, y un resplandor si la salida está adyacente a ti.

Los perceptos de las paredes toman valor 1 si el agente se encuentra en una casilla que tiene dicha pared.

Por último, grito es el percepto que se obtiene si se usa el arma y se acierta dándole al monstruo (comentado más adelante en el agente). CK_con_CW es el percepto que se obtiene si se ha encontrado al Coronel Kurtz y está con nosotros.

El constructor llama a la función shuffle (Figura 2. Función Shuffle) para generar las posiciones aleatorias de los elementos del palacio.



```python
def shuffle(self) -> None:
    """
    Genera las posiciones aleatorias de los elementos del palacio
    """
    self.CW_start = list(self.random_coords())
    self.CW = self.CW_start.copy()

    self.salida = None
    while self.salida is None or self.salida == tuple(self.CW):
        self.salida = self.random_coords()

    self.precipicios = []
    while len(self.precipicios) != self.N_precipicios:
        coord = self.random_coords()
        if (
            coord != tuple(self.CW)
            and coord != self.salida
            and coord not in self.precipicios
        ):
            self.precipicios.append(coord)

    self.monstruo = None
    while (
        self.monstruo is None
        or coord == tuple(self.CW)
        or coord == self.salida
        or self.monstruo in self.precipicios
    ):
        self.monstruo = self.random_coords()
    self.monstruo_start = self.monstruo

    self.CK = None
    while (
        self.CK is None
        or (self.CK) == list(self.CW)
        or self.CK == self.monstruo
        or self.CK in self.precipicios
    ):
        self.CK = self.random_coords()

    self.CK = list(self.CK)
    self.CK_start = self.CK.copy()

    self.acaba_de_detonar = False
```

Figura 2. Función Shuffle

Por último, cabe destacar la función que genera los perceptos para pasárselos al agente. La función get_entorno (Figura 3. Función get_entorno) devuelve los perceptos según la posición del agente.

```python
def get_entorno(self, position=None) -> list:
    """
    Genera la lista de los perceptos para determinada posición
    """
    if position is None:
        position = self.CW
    else:
        # Se traslada la posición dada por el agente a la del mapa
        position = (
            position[0] - 1 + self.CW_start[0],
            position[1] - 1 + self.CW_start[1],
        )

    brisa = (
        any(is_adyacent(position, p) for p in self.precipicios)
        or tuple(position) in self.precipicios
    )
    fetido_olor = (
        is_adyacent(position, self.monstruo) or tuple(position) == self.monstruo
    )
    resplandor = is_adyacent(position, self.salida)
    pared_arriba = position[0] == 1
    pared_abajo = position[0] == self.n
    pared_izquierda = position[1] == 1
    pared_derecha = position[1] == self.n
    grito = self.acaba_de_detonar and self.monstruo == (-100, -100)
    CK_con_CW = tuple(self.CK) == tuple(position)
    self.acaba_de_detonar = False

    return (
        brisa,
        fetido_olor,
        resplandor,
        pared_arriba,
        pared_abajo,
        pared_izquierda,
        pared_derecha,
        grito,
        CK_con_CW,
    )
```

Figura 3. Función get_entorno

Adaptación del palacio a Pygame

El objeto palacio usa la función dibujar (Figura 4. Función dibujar) para dibujar su estado en la terminal el estado del palacio según los conocimientos del agente.

```python
def dibujar(self, states, CW=None, CK=None) -> None:
    """
    Dibuja el palacio en la terminal
    """
    if CW is None:
        CW = self.CW
    else:
        CW = (
            CW[0] - 1 + self.CW_start[0],
            CW[1] - 1 + self.CW_start[1],
        )

    if CK is None:
        CK = self.CK
    else:
        # Trasladar las coordenadas dadas por el agente a las realess
        CK = (
            CK[0] - 1 + self.CW_start[0],
            CK[1] - 1 + self.CW_start[1],
        )
    dibujo = ""
    traps = {"P", "M", "S"}
    i_start, j_start = self.CW_start
    for i in range(1, self.n + 1):
        for j in range(1, self.n + 1):
            # trasladar la posicion real a la del agente
            position = (i + 1 - i_start, j + 1 - j_start)

            # obtener la información que tiene el agente de la sala a dibujar
            state = states.get(position, {})

            if len(state) == 0:
                # Si no sabe nada de la sala
                dibujo += "| xx |"
            elif len(state) < 3:
                # Si sabe algo de la sala
                doubt = list(traps - set(list(state.keys())))
                if len(doubt) == 1:
                    dibujo += f"| {doubt[0]}? |"
                else:
                    dibujo += f"|{doubt[0]}?{doubt[1]}?|"

            else:
                # Si sabe todo de la sala
                if [i, j] == list(CK):
                    if self.CK == CW:
                        dibujo += "|CWCK|"
                    else:
                        dibujo += "| CK |"
                elif [i, j] == list(CW):
                    dibujo += "| CW |"
                elif (i, j) in self.precipicios:
                    dibujo += "| P  |"
                elif (i, j) == self.monstruo:
                    dibujo += "| M  |"
                elif (i, j) == self.salida:
                    dibujo += "| S  |"
                else:
                    dibujo += "|    |"
        dibujo += "\n"
    print(dibujo)
```

Figura 4. Función dibujar

```
Posible moves: usar arma, up, down, left
DIRECCION (WASD):a
| xx || P? ||    || S  ||    ||    |
| P? ||    ||    ||    ||    ||    |
|    ||    ||    ||    || P? ||    |
|    ||    ||    || CK || xx || P? |
|    ||    || CW ||    || M  || xx |
| xx ||    ||    ||    || xx || xx |
```

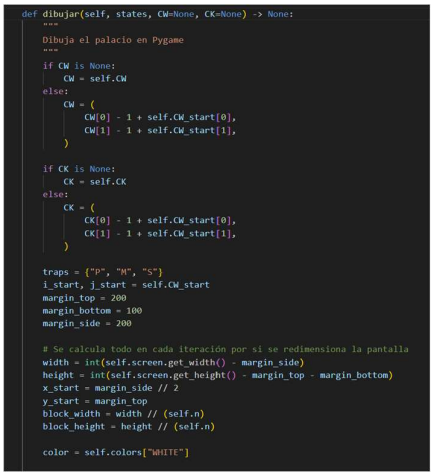
Figura 5. Ejemplo dibujar mapa en terminal (posible moves: .... es otra función)

Sin embargo, para poder dibujar el palacio en pygame, se implementó una clase adicional del palacio que heredaba de la clase principal de palacio.

```python
class PalacioPygame(Palacio):
    """
    Clase Palacio para el agente KB en Pygame
    """

    def __init__(self, screen, n=6, font=None) -> None:
        self.screen = screen

        self.font = font if font else pygame.font.Font(None, 28)
        Palacio.__init__(self, n)

        self.colors = {
            "WHITE": (255, 255, 255),
            "BLACK": (0, 0, 0),
            "GRAY": (169, 169, 169),
            "RED": (255, 0, 0),
            "BLUE": (0, 0, 255),
            "GREEN": (0, 255, 0),
        }
```

Figura 6. Clase PalacioPygame

Lo único que cambia en esta clase es la función de dibujar (Figura 7. Función dibujar de PalacioPygame). Es muy similar, pero se pinta sobre la superficie pantalla de pygame en vez de sobre la terminal.



```python
def dibujar(self, states, CW=None, CK=None) -> None:
    """
    Dibuja el palacio en Pygame
    """
    if CW is None:
        CW = self.CW
    else:
        CW = (
            CW[0] - 1 + self.CW_start[0],
            CW[1] - 1 + self.CW_start[1],
        )

    if CK is None:
        CK = self.CK
    else:
        CK = (
            CK[0] - 1 + self.CW_start[0],
            CK[1] - 1 + self.CW_start[1],
        )

    traps = {"P", "M", "S"}
    i_start, j_start = self.CW_start
    margin_top = 200
    margin_bottom = 100
    margin_side = 200

    # Se calcula todo en cada iteración por si se redimensiona la pantalla
    width = int(self.screen.get_width() - margin_side)
    height = int(self.screen.get_height() - margin_top - margin_bottom)
    x_start = margin_side // 2
    y_start = margin_top
    block_width = width // (self.n)
    block_height = height // (self.n)

    color = self.colors["WHITE"]
```

Figura 7. Función dibujar de PalacioPygame

```python
    y = y_start
    for i in range(1, self.n + 1):
        x = x_start
        for j in range(1, self.n + 1):
            text = ""

            position = (i + 1 - i_start, j + 1 - j_start)
            state = states.get(position, {})

            if len(state) == 0:
                color = self.colors["BLACK"]
                text = "xx"

            elif len(state) < 3:
                doubt = list(traps - set(list(state.keys())))
                if len(doubt) == 1:
                    text = f"{doubt[0]}?"
                else:
                    text = f"{doubt[0]}{doubt[1]}?"
                color = self.colors["GRAY"]

            else:
                if [i, j] == list(CK):
                    if self.CK == CW:
                        text = "CW-CK"
                        color = self.colors["GREEN"]
                    else:
                        text = "CK"
                        color = self.colors["GREEN"]
                elif [i, j] == list(CW):
                    text = "CW"
                    color = self.colors["BLUE"]
                elif (i, j) in self.precipicios:
                    text = "P"
                    color = self.colors["RED"]
                elif (i, j) == self.monstruo:
                    text = "M"
                    color = self.colors["RED"]

                elif (i, j) == self.salida:
                    text = "S"
                    color = self.colors["GREEN"]

                else:
                    text = ""
                    color = self.colors["WHITE"]

            rect = pygame.Rect(x, y, block_width, block_height)
            pygame.draw.rect(self.screen, color, rect)
            pygame.draw.rect(self.screen, self.colors["GRAY"], rect, 2)
            text_to_display = self.font.render(text, True, self.colors["BLACK"])
            text_rect = text_to_display.get_rect(center=rect.center)
            self.screen.blit(text_to_display, text_rect)
            x += block_width
        y += block_height
    pygame.display.update()
```

Figura 8. Continuación de la función dibujar de Palacio Pygame

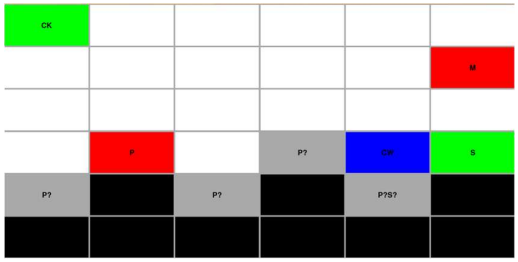


Figura 9. Ejemplo dibujar de PalacioPygame

Agente Basado En Conocimiento

Todos los agentes están basados en una clase CW (por Capitán Willard). Esta clase contiene todas las funciones básicas de interacción con el palacio.

```python
class CW:
    """
    Clase base de los agentes Capitán Willard
    """

    def __init__(self, entorno, type="user", delay=0, usos_arma=1) -> None:
        # Usos que tiene el arma del agente
        self.usos_arma = usos_arma
        # Entorno con el que interactúa el agente
        self.entorno = entorno
        self.percept_indexes = self.entorno.percept_indexes
        # Posicion inicial que tiene como referencia el agente
        # (es siempre (1,1) para él, aunque la posición real puede ser distinta)
        self.position = [1, 1]
        self.direcciones = {
            "w": "up",
            "s": "down",
            "a": "left",
            "d": "right",
            "salir": "x",
        }

        # Si interactua con usuario o recibe una lista con los pasos
        self.type = type
        self.moves_lista = []
        self.lista_index = 0

        # Tiempo entre cada paso al sacarlos de la lista moves_lista
        self.delay = delay
```

Figura 10. Constructor clase CW

```python
def salir(self) -> bool:
    return self.entorno.salir()

def get_percepts(self, position) -> list:
    """
    Consigue la lista de perceptos dada una posición
    """
    return self.entorno.get_entorno(position)

def moverse(self, direccion) -> None:
    """
    Se mueve el personaje si se lo permite el palacio
    """
    # managed_to_move es falso si no se ha podido mover por alguna pared
    managed_to_move, survived = self.entorno.mover(direccion)
    if managed_to_move:
        if direccion == "up":
            self.position[0] -= 1
        elif direccion == "down":
            self.position[0] += 1
        elif direccion == "left":
            self.position[1] -= 1
        elif direccion == "right":
            self.position[1] += 1
    return survived

def actuar(self, accion) -> tuple:
    """
    Actúa dada una acción
    """
    if accion.startswith(" ") or accion == "usar arma":
        # Hay dirección si empieza por " " y tiene algo más detrás
        direccion = (
            None if len(accion) == 1 or accion == "usar arma" else accion[1:]
        )
        direccion = self.direcciones.get(direccion, direccion)
        self.usar_arma(direccion)
        return True, True
    elif accion == "x" or accion == "salir":
        has_escapado = self.salir()
        return not has_escapado, has_escapado
    else:
        return self.moverse(accion), False
```

Figura 11. Funciones principales de interacción con el entorno

```python
def get_next_move(self) -> str:
    """
    Consigue la próxima acción a partir de input del usuario o de la lista de acciones
    """
    if self.type == "user":
        # get input from user
        accion = input("DIRECCION (WASD):").lower()
        return self.direcciones.get(accion, accion)
    elif self.type == "lista":
        i = self.lista_index
        self.lista_index += 1

        # delay
        time.sleep(self.delay)
        return self.moves_lista[i]

def display_posible_moves(self, posible_moves) -> None:
    """
    Imprime por pantalla las posibles acciones
    """
    if posible_moves:
        print("Posible moves:", ", ".join(list(zip(*posible_moves))[0]))
    else:
        print("No possible moves")

def get_adyacent_positions(self, position=None) -> tuple:
    """
    Devuelve las posiciones adyacentes a una posición
    """
    if position is None:
        position = self.position

    up = (position[0] - 1, position[1])
    down = (position[0] + 1, position[1])
    left = (position[0], position[1] - 1)
    right = (position[0], position[1] + 1)
    return up, down, left, right
```

Figura 12. Funciones a destacar de la clase CW

Clase del agente basado en conocimiento

Esta clase, que hereda las funciones básicas de la clase CW, implementa las funciones necesarias para realizar la inferencia. No se ha hecho uso de ninguna librería adicional, como puede ser PycoSat.

```python
class CW_KB(CW):
    def __init__(self, entorno, delay=0) -> None:
        CW.__init__(self, entorno, delay=delay)
        self.KB = dict()
        self.KB_format = [
            "B",
            "O",
            "R",
            "P",
            "M",
            "S",
            "PU",
            "PD",
            "PL",
            "PR",
            "MU",
            "MD",
            "ML",
            "MR",
            "SU",
            "SD",
            "SL",
            "SR",
        ]
        # Indices de cada opción del KB
        self.indexes = {self.KB_format[i]: i for i in range(len(self.KB_format))}

        # Clausulas base
        self.base_clauses = set()
        self.set_base_clauses()

        # Estados de las celdas
        self.room_states = dict()

        # Posiciones de objetos
        self.monster_position = None
        self.salida_position = None
```

Figura 13. Constructor clase CW_KB

Primero, se eligió el formato de las cláusulas que se guardarán en el KB. Se usan vectores que pueden tener 3 valores: 0 si el valor es falso, 1 si es verdadero, y -1 para la ausencia de valor. Además, el vector consta de 18 elementos:

$[Brisa, Olor, Resplandor, Precipicio, Monstruo, Salida, Precipicio_{arriba}, Precipicio_{abajo}, Precipicio_{izquierda}, Precipicio_{derecha}, Monstruo_{arriba}, Monstruo_{abajo}, Monstruo_{izquierda}, Monstruo_{derecha}, Salida_{arriba}, Salida_{abajo}, Salida_{izquierda}, Salida_{derecha}]$

Por tanto, tenemos la cláusula:

$$Brisa \rightarrow Precipicio_{derecha} \equiv \neg Brisa \vee Precipicio_{der}$$

El vector quedaría:

$$[0, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, -1, -1, -1, -1]$$

```python
def parse_proposition(self, proposition: str) -> tuple:
    """
    Convierte una proposición de texto al formato de cada cláusula
    """
    # Regex para dividir la proposición
    pattern = r"\s(?![^(]*\))"
    splitted = re.split(pattern, proposition)

    # Creo el vector de -1
    parsed = [-1 for _ in range(len(self.indexes))]

    # Recorremos los literales
    for p in splitted:
        if p.startswith("-") or p.startswith("~") or p.startswith("~"):
            parsed[self.indexes[p[1:]]] = 0
        elif p in self.indexes:
            parsed[self.indexes[p]] = 1
    return tuple(parsed)
```

Figura 14. Función que genera el vector a partir de una cláusula (no se ha implementado la funcionalidad total para una fórmula completa)

Se han creado unas cláusulas base que han de tener todas las celdas para poder realizar la inferencia. Estas cláusulas son las imprescindibles para resolver el problema, pues el algoritmo de inferencia es lento y añadir cláusulas ralentizaría al agente.

La fórmula añadida es la siguiente:

$$Brisa \leftrightarrow (Precipicio_{arriba} \vee Precipicio_{abajo} \vee Precipicio_{izquierda} \vee Precipicio_{derec})$$

Que convirtiéndolo en CNF:

$$(Brisa \rightarrow (Precipicio_{arriba} \vee Precipicio_{abajo} \vee Precipicio_{izquierda} \vee Precipicio_{derec})) \wedge$$

$$\wedge((Precipicio_{arriba} \vee Precipicio_{abajo} \vee Precipicio_{izquierda} \vee Precipicio_{derec}) \rightarrow Brisa)$$

Sustituyendo por las implicaciones y simplificando, al final quedan las siguientes cláusulas:

$$\neg Brisa \vee (Precipicio_{arriba} \vee Precipicio_{abajo} \vee Precipicio_{izquierda} \vee Precipicio_{derecha})$$

$$Brisa \vee \neg Precipicio_{arriba}$$

$$Brisa \vee \neg Precipicio_{abajo}$$

$$Brisa \vee \neg Precipicio_{izquierda}$$

$$Brisa \vee \neg Precipicio_{derecha}$$

Las cuales son añadidas a todas las celdas para poder ser usadas durante la inferencia. Estas cláusulas se crean para todos los pares estímulo-objeto.

```python
def set_base_clauses(self) -> None:
    """
    Implementa las cláusulas base que cada celda debe tener
    """
    for a, b in [("B", "P"), ("O", "M"), ("R", "S")]:
        # B -> (PU V PD V PL V PR) and so on...
        self.base_clauses.add(
            self.parse_proposition(f"¬{a} V {b}U V {b}D V {b}L V {b}R")
        )

        # (PU V PD V PL V PR) -> B
        for k in ["U", "D", "L", "R"]:
            self.base_clauses.add(self.parse_proposition(f"{a} V ¬{b}{k}"))
```

Figura 15. Función set_base_clauses

Cuando se recorre el palacio, se añaden las cláusulas que se obtienen a partir de los perceptos.

```python
def percepts_to_knowledge(self, percepts) -> list:
    """
    Dados los perceptos, crea las cláusulas asociadas a esos perceptos
    Devuelve una lista porque es una conjunción de cláusulas
    """
    # conocimientos de perceptos de brisa, olor, resplandor
    return [
        tuple([-1 if k != i else percepts[i] for k in range(len(self.KB_format))])
        for i in range(3)
    ]

def add_knowledge_from_percepts(self, percepts, KB, position=None) -> None:
    """
    Añade conocimiento al KB a partir de los perceptos
    """
    # Si se ha percibido un grito, se elimina al monstruo
    grito = percepts[self.percept_indexes["grito"]]
    if grito:
        self.remove_from_KB(position, "O", KB)
        if self.monster_position is not None:
            self.remove_from_KB(self.monster_position, "M", KB)
            self.room_states[self.monster_position]["M"] = 0

    # Recorrer las cláusulas obtenidas a partir de los perceptos y añadirlas al KB
    for knowledge in self.percepts_to_knowledge(percepts):
        self.add_to_KB(position, knowledge, KB)
```

Figura 16. Funciones necesarias para añadir la información de los perceptos al KB

Tras añadir los perceptos, se realiza la inferencia para obtener nueva información sobre las celdas adyacentes. El agente se pregunta si cada celda adyacente tiene el monstruo, un precipicio, o la salida, esto se hace con la función ask_everything.

```python
def ask_everything(self, cell, KB, room_states) -> None:
    """
    Hace las preguntas necesarias a la celda para inferir información de las celdas adyacentes
    """
    # Actualizamos información de la celda porque igual se tiene nueva información
    # de las celdas de alrededor que no está añadida
    self.update_KB(cell, KB, room_states)

    # conseguimos celdas adyacentes
    adyacent = self.get_adyacent_positions(cell)

    # Recorremos la información de cada objeto para arriba, abajo, izquierda, derecha
    for i, q in enumerate(self.KB_format[6:]):
        # Sacamos la celda adyacente adecuada
        cell_adyacent = adyacent[(i) % 4]

        # Si no se sabe ya la respuesta, se pregunta
        if q[0] not in room_states.get(cell_adyacent, {}):
            # Preguntamos a la celda
            KB_entails_q = self.ask(cell, q, KB)

            # Si es un sí
            if KB_entails_q:
                # Ponemos propiedad a la celda adyacente
                self.set_adyacent_cells_property(
                    cell_adyacent, q[0], 1, KB, room_states
                )

            else:
                # Preguntamos lo contrario a la celda
                KB_entails_not_q = self.ask(cell, "~" + q, KB)
                # Si es un sí
                if KB_entails_not_q:
                    # Ponemos propiedad a la celda adyacente
                    self.set_adyacent_cells_property(
                        cell_adyacent, q[0], 0, KB, room_states
                    )
```

Figura 17: Función ask_everything

```python
def update_KB(self, cell, KB, room_states) -> None:
    """
    Actualiza la información de la celda a partir de las celdas de alrededor
    """
    positions = {0: "U", 1: "D", 2: "L", 3: "R"}

    # Ponemos la posición como limpia, a excepción de la salida
    self.set_clear(cell, KB, room_states, also_S=False)

    # Recorremos las celdas adyacentes
    for p, adyacent in enumerate(self.get_adyacent_positions(cell)):
        # Se consigue el estado de la celda adyacente, está vacío si no se ha añadido antes
        state = room_states.get(adyacent, {})

        # Se recorren los objetos del estado
        for s, v in state.items():
            proposition = s + positions[p]
            if v == 0:
                proposition = "¬" + proposition

            # Se añade la información al KB de la celda
            self.add_to_KB(cell, proposition, KB)
```

Figura 18. Función update_KB

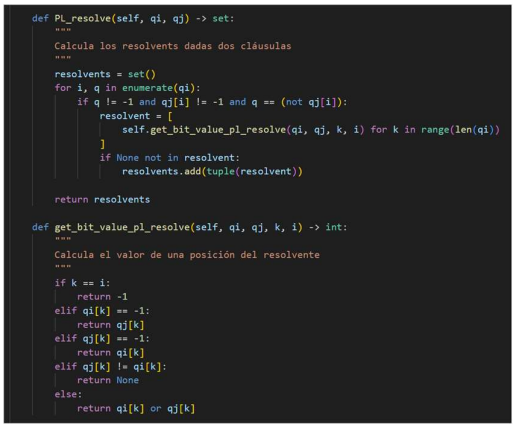
```python
def set_adyacent_cells_property(
    self, adyacent_position, property, value, KB, room_states
) -> None:
    """
    Establece una propiedad de la celda adyacente
    """
    # Negamos la cláusula si es necesario
    proposition = property if value else "¬" + property

    # Añadimos al KB
    self.add_to_KB(adyacent_position, proposition, KB)

    # Establecemos el estado de la celda
    if adyacent_position not in room_states:
        room_states[adyacent_position] = dict()
    room_states[adyacent_position][property] = value

    # Si es monstruo o salida, guardamos la posición
    if property == "M" and value == 1:
        self.monster_position = adyacent_position
    elif property == "S" and value == 1:
        self.salida_position = adyacent_position
```

Figura 19. Función set_adyacent_cells_property

Esta inferencia se realiza con el algoritmo de resolución visto en clase. El cual busca la cláusula vacía entre los resolventes obtenidos aplicando PL_resolve a todas las parejas de cláusulas.

```python
def ask(self, room, alpha, KB) -> bool:
    """
    Applies resolution and answers if KB entails alpha
    """
    if KB is None:
        KB = self.KB
    if room not in KB:
        self.add_to_KB(room, KB)
    clauses = set(KB[room])

    # Añade el alpha negado
    clauses.add(self.parse_proposition(self.not_clause(alpha)))

    # Algoritmo de resolución
    new = set()
    while True:
        clause_list = list(clauses)

        for i, ci in enumerate(clause_list):
            for cj in clause_list[i + 1 :]:
                resolvents = self.PL_resolve(ci, cj)
                if -len(self.indexes) in [sum(r) for r in resolvents]:
                    return True
                new = new | resolvents

        if new <= clauses:
            return False
        clauses = clauses | new
```

Figura 20. Función ask



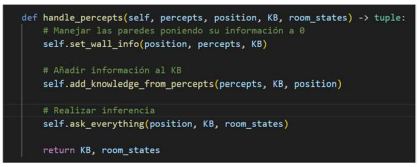
```python
def Pl_resolve(self, qi, qj) -> set:
    """
    Calcula los resolvents dadas dos cláusulas
    """
    resolvents = set()
    for i, q in enumerate(qi):
        if q != -1 and qj[i] != -1 and q == (not qj[i]):
            resolvent = [
                self.get_bit_value_pl_resolve(qi, qj, k, i) for k in range(len(qi))
            ]
            if None not in resolvent:
                resolvents.add(tuple(resolvent))

    return resolvents

def get_bit_value_pl_resolve(self, qi, qj, k, i) -> int:
    """
    Calcula el valor de una posición del resolvente
    """
    if k == i:
        return -1
    elif qi[k] == -1:
        return qj[k]
    elif qj[k] == -1:
        return qi[k]
    elif qj[k] != qi[k]:
        return None
    else:
        return qi[k] or qj[k]
```

Figura 21. PL_resolve

Con todo esto, lo que hace el agente en el bucle principal (Figura 22. Función perceive_and_act) es conseguir los perceptos para su posición, usarlos para actualizar toda su base de conocimientos y realizar la inferencia (Figura 23. Función handle_percepts), dibujar el estado del entorno actual, conseguir los posibles movimientos (Figura 25), obtener la siguiente acción y realizarla.

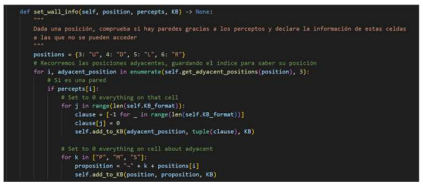
```python
def perceive_and_act(self) -> tuple:
    # Poner como segura la celda inicial
    self.set_clear(self.get_position(), self.KB, self.room_states)
    ok = True
    while ok:
        position = self.get_position()
        # Conseguir perceptos
        percepts = self.get_percepts(position)

        # Actualizar todo con los perceptos
        self.handle_percepts(percepts, position, self.KB, self.room_states)

        # Dibujar entorno
        self.dibujar()

        # Obtener los posibles movimientos
        posible_moves = self.get_posible_moves(percepts, position)
        self.display_posible_moves(posible_moves)

        # Obtener la siguiente acción
        accion = self.get_next_move()

        # Obtener resultado de dicha acción
        ok, victory = self.actuar(accion)

    if victory:
        print("ENHORABUENA!!! Has conseguido salir")
    else:
        print("Has perdido")

    return ok, victory
```

Figura 22. Función perceive_and_act



```python
def handle_percepts(self, percepts, position, KB, room_states) -> tuple:
    # Manejar las paredes poniendo su información a 0
    self.set_wall_info(position, percepts, KB)

    # AÑadir información al KB
    self.add_knowledge_from_percepts(percepts, KB, position)

    # Realizar inferencia
    self.ask_everything(position, KB, room_states)

    return KB, room_states
```

Figura 23. Función handle_percepts



```python
def set_wall_info(self, position, percepts, KB) -> None:
    """
    Dada una posición, comprueba si hay paredes gracias a los perceptos y declara la información de estas celdas
    a las que no se pueden acceder
    """
    positions = {3: "U", 4: "D", 5: "L", 6: "R"}
    # Recorremos las posiciones adyacentes, guardando el índice para saber su posición
    for i, adyacent_position in enumerate(self.get_adyacent_positions(position), 3):
        # Si es una pared
        if percepts[i]:
            # Set to 0 everything on that cell
            for j in range(len(self.KB_format)):
                clause = [-1 for _ in range(len(self.KB_format))]
                clause[j] = 0
                self.add_to_KB(adyacent_position, tuple(clause), KB)

            # Set to 0 everything on cell about adyacent
            for k in ["W", "M", "S"]:
                proposition = "~" + k + positions[i]
                self.add_to_KB(position, proposition, KB)
```

Figura 24. Función set_wall_info

```python
def get_posible_moves(
    self, percepts, position=None, KB=None, room_states=None, CK_con_CW=False
) -> list:
    """
    Consigue la lista de posibles movimientos
    """
    if position is None:
        position = self.get_position()

    if room_states is None:
        room_states = self.room_states

    if KB is None:
        KB = self.KB

    # Lista de posibles acciones
    actions = []

    positions = {3: "up", 4: "down", 5: "left", 6: "right"}

    # Mira si se puede salir
    if (percepts[self.percept_indexes["CK_con_CW"]] or CK_con_CW) and room_states[
        position
    ].get("S", 0):
        actions.append(("salir", (None, None)))

    # Comprueba si se puede usar el arma
    if percepts[self.percept_indexes["fetido_olor"]]:
        actions.append(("usar arma", position))

    # Recorre las adyacentes y comprueba que no sean paredes y sean seguras
    for i, adyacent_position in enumerate(self.get_adyacent_positions(position), 3):
        if not percepts[i] and self.is_safe(adyacent_position, room_states):
            actions.append((positions[i], adyacent_position))

    return actions
```

Figura 25. Función get_posible_moves

Cabe destacar que el agente no tiene información sobre cuántos precipicios hay, ni salidas, ni monstruos. Por lo que, aunque sepa que hay un monstruo en una sala, puede seguir teniendo dudas sobre si hay en otra sala o no.

Agente Bayesiano

Palacio del agente bayesiano

La clase del palacio para el agente bayesiano es muy similar al del agente basado en conocimiento, pero cambian los estímulos, los perceptos (Figura 26) y la disposición de las trampas, dado que ahora hay algunas que se pueden encontrar en la misma celda.

```python
class PalacioBayes:
    """
    Clase del Palacio para el agente Bayesiano
    """

    def __init__(self, n: int = 6) -> None:
        self.n = n
        self.posibles_perceptos = [
            "estimulo_fuego",
            "estimulo_pinchos",
            "estimulo_dardos",
            "estimulo_monstruo",
            "estimulo_salida",
            "pared_arriba",
            "pared_abajo",
            "pared_izquierda",
            "pared_derecha",
            "grito",
            "CK_con_CW",
        ]

        self.percept_indexes = {
            percept: i for i, percept in enumerate(self.posibles_perceptos)
        }
```

Figura 26. Constructor clase PalacioBayes

```python
def get_entorno(self, position=None) -> list:
    """
    Genera la lista de los perceptos para determinada posición
    percept(s) = [eF?, eP?, eD?, eM?, eS?, Pared↑?, Pared↓?, Pared←?, Pared→?, Grito?]
    """
    if position is None:
        position = tuple(self.CW)
    else:
        # Se traslada la posición dada por el agente a la del mapa
        position = (position[0] + self.CW_start[0], position[1] + self.CW_start[1])

    indice_fuego = self.indices_trampas["fuego"]
    indice_pinchos = self.indices_trampas["pinchos"]
    indice_dardos = self.indices_trampas["dardos"]
    eF = (
        is_adyacent(position, self.precipicios[indice_fuego])
        or position == self.precipicios[indice_fuego]
    )
    eP = (
        is_adyacent(position, self.precipicios[indice_pinchos])
        or position == self.precipicios[indice_pinchos]
    )
    eD = (
        is_adyacent(position, self.precipicios[indice_dardos])
        or position == self.precipicios[indice_dardos]
    )
    eM = is_adyacent(position, self.monstruo) or position == self.monstruo
    eS = is_adyacent(position, self.salida) or position == self.salida
    pared_arriba = position[0] == 1
    pared_abajo = position[0] == self.n
    pared_izquierda = position[1] == 1
    pared_derecha = position[1] == self.n
    grito = self.acaba_de_detonar and self.monstruo == (-100, -100)
    CK_con_CW = tuple(self.CK) == tuple(position)
    self.acaba_de_detonar = False

    return (
        eF,
        eP,
        eD,
        eM,
        eS,
        pared_arriba,
        pared_abajo,
        pared_izquierda,
        pared_derecha,
        grito,
        CK_con_CW,
    )
```

Figura 27. Función get_entorno

La función de dibujar también es distinta, en la terminal se muestra la suma de probabilidades de todos los mapas, y en pygame la probabilidad de cada objeto. Se ha vuelto a crear otro objeto que hereda lo básico del anterior para añadir la funcionalidad de dibujar en pygame.

```python
def dibujar(self, mapas, position=None) -> None:
    """
    Dibuja el palacio en la terminal
    """
    if position is None:
        position = self.CW
        start = self.CW_start
        p1, p2 = position
        position = (p1 - start[0], p2 - start[1])

    suma = np.sum(mapas, axis=0)
    i, j = position

    dibujo = ""
    for k in range(self.n):
        for l in range(self.n):
            if (k, l) == (i, j):
                if self.CW == self.CK:
                    dibujo += "|CW  CK|"
                else:
                    dibujo += "|  CW  |"
            elif (k, l) == (self.salida[0] - 1, self.salida[1] - 1):
                dibujo += f"|S {suma[k, l]:0.02f}|"
            else:
                dibujo += f"| {suma[k, l]:0.02f} |"
        dibujo += "\n"
    print(dibujo)
```

Figura 28. Función dibujar

```
Posible moves: up, left, right
DIRECCION (WASD):d
| 0.00 || 0.00 || 0.00 || 0.00 || 0.00 || 0.06 |
|S 1.00|| 0.00 || 0.00 || 0.00 ||  CW  || 0.50 |
| 0.00 || 0.00 || 0.00 || 1.00 || 0.50 || 0.06 |
| 0.00 || 1.00 || 0.00 || 0.06 || 0.06 || 0.06 |
| 0.06 || 0.06 || 0.06 || 0.06 || 0.06 || 0.06 |
| 0.06 || 0.06 || 0.06 || 0.06 || 0.06 || 0.06 |
```

Figura 29. Ejemplo función dibujar

```python
def dibujar(self, mapas, position=None, umbral=None) -> None:
    """
    Dibuja el palacio en Pygame
    """
    if position is None:
        position = self.CW
        start = self.CW_start
        p1, p2 = position
        position = (p1 - start[0], p2 - start[1])

    if umbral == None:
        umbral = 0.2

    margin_top = 200
    margin_bottom = 100
    margin_side = 200

    width = int(self.screen.get_width() - margin_side)
    height = int(self.screen.get_height() - margin_top - margin_bottom)

    x_start = margin_side // 2
    y_start = margin_top
    block_width = width // (self.n)
    block_height = height // (self.n)

    displacement_x = (block_width // 2) * 0.6
    displacement_y = (block_height // 2) * 0.7

    color = self.colors["WHITE"]

    suma = np.sum(mapas, axis=0)
    i, j = position
```

Figura 30. Función dibujar en PalacioBayesPygame

```
y = y_start
for k in range(self.n):
    x = x_start
    for l in range(self.n):
        text = ""
        texts_to_draw = []

        rect = pygame.Rect(x, y, block_width, block_height)
        color = (
            self.colors["WHITE"]
            if suma[k, l] == 0
            else self.colors["YELLOW"]
            if suma[k, l] < umbral
            else self.colors["RED"]
            if suma[k, l] == 1
            else self.colors["ORANGE"]
        )
        if (k, l) == (i, j):
            if self.CW == self.CK:
                text = "CW CK"
                color = self.colors["GREEN"]
            else:
                text = "CW"
                color = self.colors["BLUE"]
            texts_to_draw.append((text, rect.center))

        else:
            if mapas[3][k, l] > 0:
                pos = change_pos(rect.center, -displacement_x, -displacement_y)
                texts_to_draw.append((f"M:{mapas[3][k, l]:0.02f}", pos))
            if mapas[0][k, l] > 0:
                pos = change_pos(rect.center, displacement_x, -displacement_y)
                texts_to_draw.append((f"F:{mapas[0][k, l]:0.02f}", pos))

            if mapas[1][k, l] > 0:
                pos = change_pos(rect.center, -displacement_x, displacement_y)
                texts_to_draw.append((f"P:{mapas[1][k, l]:0.02f}", pos))

            if mapas[2][k, l] > 0:
                pos = change_pos(rect.center, displacement_x, displacement_y)
                texts_to_draw.append((f"D:{mapas[2][k, l]:0.02f}", pos))

            if mapas[4][k, l] > 0:
                texts_to_draw.append((f"S:{mapas[4][k, l]:0.02f}", rect.center))
                if mapas[4][k, l] >= 0.1:
                    color = self.colors["GREEN"]

        pygame.draw.rect(self.screen, color, rect)
        pygame.draw.rect(self.screen, self.colors["GRAY"], rect, 2)

        for t, p in texts_to_draw:
            render_text_cell(t, p, self.screen, self.font)

        x += block_width
    y += block_height
pygame.display.update()
```

Figura 31. Bucle de función dibujar de PalacioBayesPygame

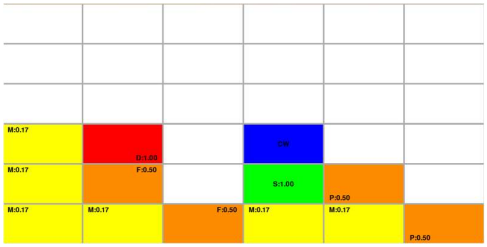


Figura 32. Ejemplo dibujar en pygame

Agente Bayesiano

El agente bayesiano, a diferencia del basado en conocimiento, usa probabilidad para inferir las posiciones de los objetos en el mapa.

Para ello, se crean mapas de probabilidad para todos los posibles objetos. Estos mapas se inicializan con un prior. El capitán, al no saber nada de la posición de los objetos, les da la misma probabilidad a todas las celdas (exceptuando en la que comienza), por lo que se usa una distribución uniforme (Figura 33).

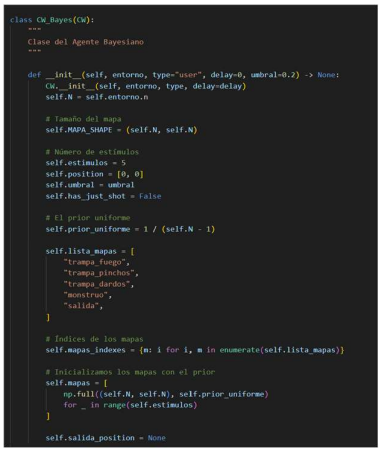


Figura 33. Constructor clase CW_Bayes

Cuando el Capitán se mueve, realiza inferencia aplicando la regla de Bayes. Para ello, usa la siguiente función de verosimilitud.

$$p(e_{ij}^{\tau}|\tau_{k\ell}) = \begin{cases} 1 & (k,l) \in adj(i,j) \cup \{(i,j)\}, \\ 0 & \text{otherwise.} \end{cases}$$

Es decir, la probabilidad de percibir un estímulo es 1 si el objeto está en las adyacentes, así como si la trampa se encuentra en la propia celda en la que se encuentra (aunque ya estaría muerto) y 0 si está en otras (Figura 34). Con esta función verosimilitud obtenida, se crea una matriz con el valor de la verosimilitud para cada celda, la cual se usará para aplicar la regla de Bayes (Figura 35).

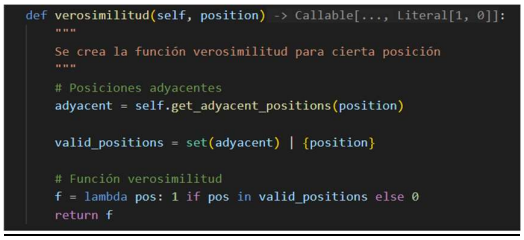


Figura 34. Función verosimilitud

Figura 35. Función create_mapa_verosimilitud

Para actualizar cada mapa, se aplica la regla de Bayes:

$$p(\tau_{k\ell}|e^{\tau}_{ij}) = \frac{p(e^{\tau}_{ij}|\tau_{k\ell}) \cdot p(\tau_{k\ell})}{p(e^{\tau}_{ij})}.$$

Es decir, multiplicamos el prior (o la distribución de probabilidad de las celdas actuales), lo multiplicamos por la verosimilitud y la dividimos por la suma de todo eso.

```python
def Bayes(self, mapa_verosimilitud, mapa_prior) -> np.array:
    """
    Aplica Bayes
    """
    # Calcula numerador
    numerador = mapa_verosimilitud * mapa_prior

    # El denominador es la integral (suma) del numerador
    denominador = np.sum(numerador)

    # Para no dividir por cero
    if denominador == 0:
        return numerador

    # Se devuelve la fracción
    return numerador / denominador
```

Figura 36. Función Bayes.

Una vez se tiene el nuevo valor, se actualiza el mapa. Se hace esto con todos los mapas y todos los perceptos (Figura 37).

```python
def actualizar_mapa(
    self, index, percept, mapas, position, p_verosimilitud=None
) -> None:
    """
    Se actualizan las probabilidades de un mapa a partir de la posición y el percepto
    """
    mapa_verosimilitud = self.create_mapa_verosimilitud(
        position, percept, p_verosimilitud
    )
    mapa_prior = mapas[index]
    mapas[index] = self.Bayes(mapa_verosimilitud, mapa_prior)

def actualizar_mapas(self, percepts, mapas, position) -> None:
    """
    Se actualizan todos los mapas del agente dados los perceptos para una posición
    """
    # Se recorren los perceptos y se actualizan los mapas asociados a cada uno
    for i, percept in enumerate(percepts[: self.estimulos]):
        self.actualizar_mapa(i, percept, mapas, position)

    # Si se oye un grito, se se pone a 0 la probabilidad de monstruo en todas las casillas
    if percepts[self.percept_indexes["grito"]]:
        mapas[self.mapas_indexes["monstruo"]] = np.zeros(self.MAPA_SHAPE)
```

Figura 37. Funciones actualizar_mapas y actualizar_mapa

Ahora que se tiene definido el método de inferencia, el bucle principal del agente es prácticamente idéntico al del agente KB (Figura 38). Se obtienen los estímulos para su posición, se manejan los perceptos (se actualizan los mapas con Bayes) (Figura 39) y se obtienen las próximas jugadas (Figura 40) para mostrarlas y obtener la siguiente acción. Los posibles movimientos se obtienen considerando un umbral para determinar su seguridad. Si la suma de las probabilidades de los mapas usados para las trampas y el monstruo es menor que este umbral, se consideran seguras (Figura 40).

```python
def perceive_and_act(self) -> tuple:
    """
    Bucle principal del agente
    """
    ok = True
    while ok:
        # Conseguir perceptos y manejarlos
        percepts = self.get_percepts(self.get_position())
        self.handle_percepts(percepts)

        # Dibuja entorno
        self.dibujar()

        # Obtener los posibles movimientos
        posible_moves = self.get_posible_moves(
            percepts, self.get_position(), self.mapas, self.umbral
        )

        # Mostralos
        self.display_posible_moves(posible_moves)

        # Obtener la siguiente acción
        accion = self.get_next_move()

        # Obtener resultado de dicha acción
        ok, victory = self.actuar(accion)

    if victory:
        print("ENHORABUENA!!! Has conseguido salir")
    else:
        print("Has perdido")
    return ok, victory
```

Figura 38. Bucle principal

```python
def handle_percepts(self, percepts, position=None, mapas=None) -> None:
    """
    Usa los perceptos para actualizar los mapas
    """
    if position is None:
        position = self.get_position()

    if mapas is None:
        mapas = self.mapas

    # Actualiza los mapas
    self.actualizar_mapas(percepts, mapas, position)

    # Si se sabe con certeza la localización de la salida, se guarda la posición
    if np.count_nonzero(mapas[self.mapas_indexes["salida"]] == 1) == 1:
        i, j = np.where(mapas[self.mapas_indexes["salida"]] == 1)
        self.salida_position = (i[0], j[0])
```

Figura 39. Función handle_percepts

```python
def get_posible_moves(
    self, percepts, position, mapas, umbral, CK_con_CW=False
) -> list:
    """
    Consigue la lista de posibles movimientos
    """
    positions = {5: "up", 6: "down", 7: "left", 8: "right"}

    # Lista de posibles acciones
    actions = []

    # Mira si se puede salir
    if (percepts[self.percept_indexes["CK_con_CW"]] or CK_con_CW) and mapas[
        self.mapas_indexes["salida"]
    ][position[0], position[1]] == 1:
        actions.append(("salir", (None, None)))

    # Recorre las adyacentes y comprueba que no sean paredes y sean seguras
    for i, adyacent_position in enumerate(self.get_adyacent_positions(position), 5):
        if not percepts[i] and self.is_safe(adyacent_position, mapas, umbral):
            actions.append((positions[i], adyacent_position))

    return actions

def is_safe(self, adyacent_position, mapas, umbral) -> bool:
    """
    Establece si una posición es segura a partir del umbral
    """
    p_morir = self.get_probabilidad_de_morir(adyacent_position, mapas)
    return p_morir < umbral

def get_probabilidad_de_morir(self, position, mapas) -> float:
    """
    Calcula la probabilidad de morir en una posición a partir de la suma de todos los mapas de probabilidad de las trampas y monstruo
    """
    i, j = position
    probabilidad_de_morir = np.sum(mapas[: self.mapas_indexes["salida"]], axis=0)[
        i, j
    ]
    return probabilidad_de_morir
```

Figura 40. Funciones get_posible_moves, is_safe, get_probabilidad_de_morir

Agentes de Búsqueda

Clase SearchAgent

Para guiar a los agentes por los palacios, se ha creado una clase con la lógica de un agente de búsqueda (Figura 41).

```python
class SearchAgent:
    """
    Clase del Agente de búsqueda
    """

    def __init__(self) -> None:
        self.queue = Queue()
```

Figura 41. Constructor SearchAgent

Este agente hace uso de una cola prioridad para obtener el siguiente nodo que se va a expandir. La cola consiste en una lista ordenada de menor a mayor, de la cual se van extrayendo los nodos con menor coste. Además, tiene otra lista donde se guardan los nodos extraídos.

```python
class Queue:
    def __init__(self) -> None:
        self.queue = []
        self.visited = []

    def add(self, node, exist_ok=False, visited_ok=False) -> None:
        """
        Añade un nodo a la cola
        """
        if (exist_ok or not self.exists(node)) and (
            visited_ok or not self.exists(node, True)
        ):
            i = 0
            position_founded = False

            # Busca la posición del nodo (está ordenada de menor a mayor)
            while i < len(self.queue) and not position_founded:
                if node.cost < self.queue[i].cost:
                    position_founded = True
                else:
                    i += 1

            # Añade el nodo en su posición
            self.queue.insert(i, node)

    def add_from_list(self, list, exist_ok=False, visited_ok=False) -> None:
        """
        Añade una lista de nodos a la cola
        """
        for node in list:
            self.add(node, exist_ok, visited_ok)

    def get(self) -> Node:
        """
        Función para obtener el siguiente de la muestra
        """
        o = self.queue.pop(0)
        self.visited.append(o)
        return o

    def is_empty(self) -> bool:
        """
        Comprueba si la cola está vacía o no
        """
        return len(self.queue) == 0

    def exists(self, object, visited=False) -> bool:
        """
        Comprueba si el nodo existe en la cola o en los nodos visitados
        """
        if visited:
            to_check = self.visited
        else:
            to_check = self.queue
        for o in to_check:
            # Comprueba que el nuevo coste no sea superior al existente
            if object.position == o.position and object.cost >= o.cost:
                return True
        return False
```

Figura 42. Clase Queue

Cabe destacar que, en la clase nodo, a parte de guardar el coste, los pasos y el nodo padre, se guarda también información sobre si ese nodo estaba con el Coronel Kurtz o no (Figura 43).

```python
class Node:
    def __init__(self, position, action, father=None, cost=0, steps=0) -> None:
        self.position = position
        self.father = father
        self.action = action
        self.cost = cost
        self.steps = steps
        self.CK = self.father.CK if self.father else False

    def __repr__(self) -> str:
        return f"({self.cost}, {self.position}, {self.action})"
```

Figura 43. Clase Node

La función más relevante del objeto SearchAgent es la función de evaluación (Figura 44), esta determina el comportamiento del agente de búsqueda.

```python
    def evaluation_function(self, position, parent_node) -> tuple:
        """
        Función de evaluación del agente de búsqueda
        """
        count = 0

        if parent_node.CK:
            # Apremiamos que tenga al Coronel Kurtz
            count -= 50

        heuristic = 0
        if self.salida_position is not None and parent_node.CK:
            # Apremiamos que se haya encontrado la salida y se tenga al Coronel
            count -= 100

            # Si se tiene al Coronel y se conoce la salida, se convierte en A*
            heuristic = self.distance(self.salida_position, position)

        # Sumamos un paso
        steps = parent_node.steps + 1

        # Calculamos el coste
        cost = steps + count + heuristic
        return cost, steps
```

Figura 44. Función evaluation_function

En un principio, simplemente se van guardando los pasos para alcanzar cada nodo (se suma uno a los pasos del nodo padre), esto hace que se comporte como un BFS, pues la cola los ordena de menor a mayor. Pero, además, en caso de empate, el más reciente se coloca el último de los nodos empatados (la condición de parada es que el siguiente coste sea mayor) (Figura 42).

En cuanto se encuentre con el Coronel, se le descuentan 50 puntos al coste del nodo, esto lo que hace es volver a comenzar a buscar con BFS pero desde la posición del Coronel Kurtz.

En el caso en el que se sepa la localización de la salida y se encuentre al Coronel, se vuelve a descontar 100 puntos y el algoritmo pasa a ser A*, con una heurística que es la distancia de manhattan (Figura 45), una heurística admisible.

```python
    def distance(self, position1, position2) -> int:
        """
        Distancia de manhattan entre dos puntos
        """
        q1, q2 = position1
        p1, p2 = position2
        if q1 is None or p1 is None:
            return 0
        return abs(q1 - p1) + abs(q2 - p2)
```

Figura 45. Función distance

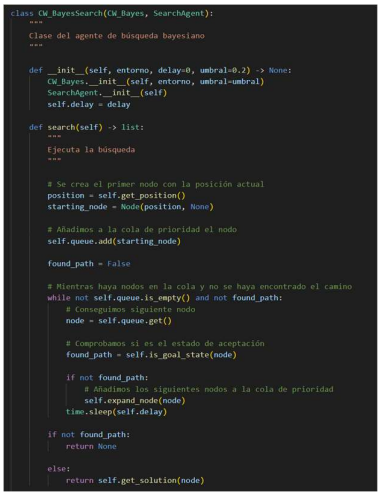
A pesar de usar BFS y A*, el agente no está garantizado a encontrar la solución óptima, esto es debido a que puede dejar una celda vacía en duda, la rodee antes de que hubiese un nodo que liberase esa casilla, dejando libre un camino más corto.

Clases CW_Search

Heredando de la clase de SearchAgent, se han creado dos nuevas clases, CW_KB_Search y CW_BayesSearch.

Estas implementan dos nuevas funciones, search y expand_node. Las cuales son prácticamente idénticas en ambos agentes.

La función search crea el primer nodo y lo añade a la cola, seguidamente se entra en un bucle donde se extrae el siguiente nodo de la cola, se comprueba si es el estado de aceptación y, si no lo es, se expande el nodo.



```python
class CW_BayesSearch(CW_Bayes, SearchAgent):
    """
    Clase del agente de búsqueda bayesiano
    """

    def __init__(self, entorno, delay=0, umbral=0.2) -> None:
        CW_Bayes.__init__(self, entorno, umbral=umbral)
        SearchAgent.__init__(self)
        self.delay = delay

    def search(self) -> list:
        """
        Ejecuta la búsqueda
        """

        # Se crea el primer nodo con la posición actual
        position = self.get_position()
        starting_node = Node(position, None)

        # Añadimos a la cola de prioridad el nodo
        self.queue.add(starting_node)

        found_path = False

        # Mientras haya nodos en la cola y no se haya encontrado el camino
        while not self.queue.is_empty() and not found_path:
            # Conseguimos siguiente nodo
            node = self.queue.get()

            # Comprobamos si es el estado de aceptación
            found_path = self.is_goal_state(node)

            if not found_path:
                # Añadimos los siguientes nodos a la cola de prioridad
                self.expand_node(node)
            time.sleep(self.delay)

        if not found_path:
            return None

        else:
            return self.get_solution(node)
```

Figura 46. Función Search de clase CW_BayesSearch

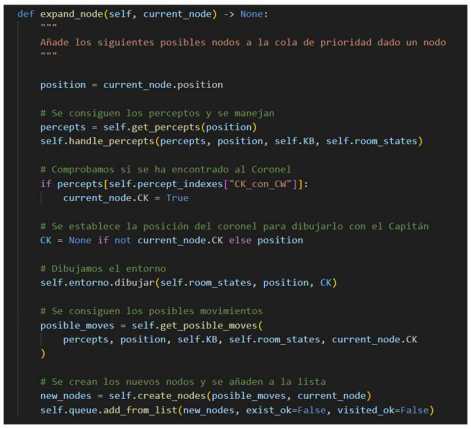
```python
class CW_KB_Search(CW_KB, SearchAgent):
    """
    Clase del agente de búsqueda basado en conocimiento
    """

    def __init__(self, entorno, delay=0) -> None:
        CW_KB.__init__(self, entorno)
        SearchAgent.__init__(self)
        self.delay = delay

    def search(self) -> list:
        """
        Ejecuta la búsqueda
        """

        # Se crea el primer nodo con la posición actual
        starting_node = Node(
            self.get_position(),
            None,
        )

        # Ponemos la habitación como segura
        self.set_clear(
            starting_node.position,
            self.KB,
            self.room_states,
        )

        # Añadimos a la cola de prioridad el nodo
        self.queue.add(starting_node)

        found_path = False

        # Mientras haya nodos en la cola y no se haya encontrado el camino
        while not self.queue.is_empty() and not found_path:
            # Conseguimos siguiente nodo
            node = self.queue.get()

            # Comprobamos si es el estado de aceptación
            found_path = self.is_goal_state(node)

            if not found_path:
                # Añadimos los siguientes nodos a la cola de prioridad
                self.expand_node(node)
            time.sleep(self.delay)

        if not found_path:
            return None
        else:
            return self.get_solution(node)
```

Figura 47. Función Search de clase CW_KB_Search

La función que expande el nodo también es prácticamente idéntica en los dos agentes. Se consiguen los perceptos, se manejan para actualizar la información, y se buscan las posibles acciones para añadirlas la cola. Cabe destacar que a la cola se añaden los nodos con exist_ok = False y visited_ok = False (Figura 49), lo que provoca el comportamiento de BFS al no añadir nodos ya visitados.

```python
def expand_node(self, current_node) -> None:
    """
    Añade los siguientes posibles nodos a la cola de prioridad dado un nodo
    """

    position = current_node.position

    # Se consiguen los perceptos y se manejan
    percepts = self.get_percepts(position)
    self.handle_percepts(percepts, position, self.mapas)

    # Comprobamos si se ha encontrado al Coronel
    if percepts[self.percept_indexes["CK_con_CW"]]:
        current_node.CK = True

    # Dibujamos el entorno
    self.entorno.dibujar(self.mapas, position)

    # Se consiguen los posibles movimientos
    posible_moves = self.get_posible_moves(
        percepts, position, self.mapas, self.umbral, current_node.CK
    )

    # Se crean los nuevos nodos y se añaden a la lista
    new_nodes = self.create_nodes(posible_moves, current_node)
    self.queue.add_from_list(new_nodes, False, False)
```

Figura 48. Función expand_node de agente bayesiano



```python
def expand_node(self, current_node) -> None:
    """
    Añade los siguientes posibles nodos a la cola de prioridad dado un nodo
    """

    position = current_node.position

    # Se consiguen los perceptos y se manejan
    percepts = self.get_percepts(position)
    self.handle_percepts(percepts, position, self.KB, self.room_states)

    # Comprobamos si se ha encontrado al Coronel
    if percepts[self.percept_indexes["CK_con_CW"]]:
        current_node.CK = True

    # Se establece la posición del coronel para dibujarlo con el Capitán
    CK = None if not current_node.CK else position

    # Dibujamos el entorno
    self.entorno.dibujar(self.room_states, position, CK)

    # Se consiguen los posibles movimientos
    posible_moves = self.get_posible_moves(
        percepts, position, self.KB, self.room_states, current_node.CK
    )

    # Se crean los nuevos nodos y se añaden a la lista
    new_nodes = self.create_nodes(posible_moves, current_node)
    self.queue.add_from_list(new_nodes, exist_ok=False, visited_ok=False)
```

Figura 49. Función expand_node de agente KB

Agentes en Pygame

Para poder usar los agentes en pygame, se han creado unas clases que añaden una función para hacer solamente un movimiento en vez del bucle principal explicado anteriormente. La función un movimiento consigue los perceptos para su posición, los maneja y muestra las posibles acciones. La función hacer siguiente movimiento toma una acción, la ejecuta para obtener su resultado y llama a la función un movimiento. También se ha adaptado la función display_posible_moves para pygame.

Cabe destacar que en el constructor ya se llama a un movimiento para hacer la primera inferencia y poder mostrar todo adecuadamente en pantalla.

```python
class CW_KB_Pygame(CW_KB):
    def __init__(
        self,
        entorno,
        screen,
        delay=0,
    ) -> None:
        CW_KB.__init__(self, entorno, delay=delay)
        self.screen = screen
        self.set_clear(self.get_position(), self.KB, self.room_states)
        self.un_movimiento()

    def un_movimiento(self) -> None:
        position = self.get_position()
        percepts = self.get_percepts(position)

        self.handle_percepts(percepts, position, self.KB, self.room_states)
        posible_moves = self.get_posible_moves(percepts, position)
        self.display_posible_moves(posible_moves)

    def display_posible_moves(self, posible_moves) -> None:
        if posible_moves:
            text = "Posible moves: " + ", ".join(list(zip(*posible_moves))[0])

        else:
            text = "No possible moves"

        text_to_display = pygame.font.Font(None, 60).render(text, True, (255, 255, 255))
        text_rect = text_to_display.get_rect(center=(self.screen.get_width() // 2, 150))
        rect = pygame.Rect(self.screen.get_width() // 2 - 425, 115, 850, 70)
        pygame.draw.rect(self.screen, (0, 0, 0), rect)
        self.screen.blit(text_to_display, text_rect)

    def hacer_siguiente_movimiento(self, accion) -> tuple:
        ok, victory = self.actuar(accion)
        self.un_movimiento()
        return ok, victory

    def dibujar(self) -> None:
        self.entorno.dibujar(self.room_states)
```

Figura 50. CW_KB_Pygame

```python
class CW_Bayes_Pygame(CW_Bayes):
    def __init__(self, entorno, screen, delay=0, umbral=0.2) -> None:
        CW_Bayes.__init__(self, entorno, delay=delay, umbral=umbral)
        self.screen = screen
        self.un_movimiento()

    def un_movimiento(self) -> None:
        percepts = self.get_percepts(self.get_position())
        self.handle_percepts(percepts)

        posible_moves = self.get_posible_moves(
            percepts, self.get_position(), self.mapas, self.umbral
        )

        self.display_posible_moves(posible_moves)

    def hacer_siguiente_movimiento(self, accion) -> tuple:
        ok, victory = self.actuar(accion)
        self.un_movimiento()
        return ok, victory

    def dibujar(self) -> None:
        self.entorno.dibujar(self.mapas, umbral=self.umbral)

    def display_posible_moves(self, posible_moves) -> None:
        if posible_moves:
            text = "Posible moves: " + ", ".join(list(zip(*posible_moves))[0])

        else:
            text = "No possible moves"

        text_to_display = pygame.font.Font(None, 60).render(text, True, (255, 255, 255))
        text_rect = text_to_display.get_rect(center=(self.screen.get_width() // 2, 150))
        rect = pygame.Rect(self.screen.get_width() // 2 - 425, 115, 850, 70)
        pygame.draw.rect(self.screen, (0, 0, 0), rect)
        self.screen.blit(text_to_display, text_rect)
```

Figura 51. CW_Bayes_Pygame

Las clases usadas en el main de pygame son unas que juntan el SearchAgent y las de pygame, para poder usar una única clase para la búsqueda y manejarlo usando pygame (Figura 52).

```python
class CW_BayesSearch_Pygame(CW_Bayes_Pygame, CW_BayesSearch):
    def __init__(self, entorno, screen, delay=0.2, umbral=0.2) -> None:
        CW_BayesSearch.__init__(self, entorno, delay)
        CW_Bayes_Pygame.__init__(self, entorno, screen, delay, umbral=umbral)


class CW_KB_Search_Pygame(CW_KB_Pygame, CW_KB_Search):
    def __init__(self, entorno, screen, delay=0.2) -> None:
        CW_KB_Search.__init__(self, entorno, delay=0)
        CW_KB_Pygame.__init__(self, entorno, screen, delay)
```

Figura 52. Clases usadas finalmente en pygame

Ejemplo de juego

En este apartado se va a mostrar cómo se usa la aplicación para interactuar con los agentes. A pesar de que también se pueden usar en la terminal, las explicaciones van a estar dirigidas para pygame principalmente. El juego de pygame se puede ejecutar en kurtz.py, mientras que los juegos de las terminales se encuentran en la función main de cada fichero con la clase dada.

Menú de juego

El menú principal (Figura 53) muestra una interfaz sencilla. En la parte de arriba se presentan cinco botones. Los dos primeros son para seleccionar el agente que se desea usar.

El tercer botón, Encontrar Solución, sirve para ejecutar el agente de búsqueda (se usará el seleccionado por los dos botones anteriores). El agente buscará el camino, mostrando su proceso por pantalla. Una vez encontrado, mostrará paso por paso la solución. Este proceso no se puede interrumpir.

El cuarto botón, Cambiar Palacio, generará otra distribución de palacio distinto, mientras que el quinto, Reiniciar Palacio, volverá a colocar al agente y todos los elementos del mapa en su posición inicial.

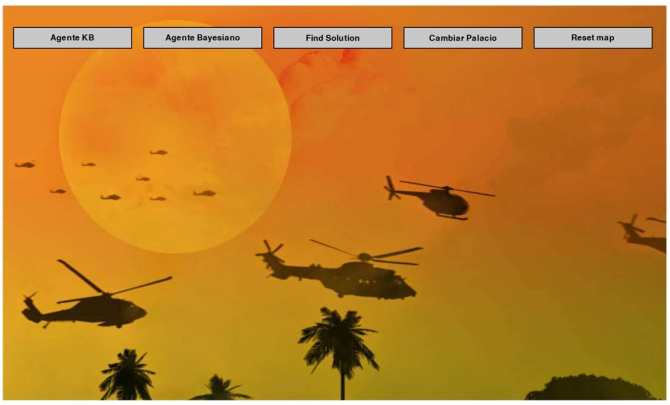


Figura 53. Menú de pygame

En la representación, las celdas con el Coronel Kurtz están siempre vacías, a no ser que se encuentre con el Capitán Willard, pues este solamente encuentra al Coronel si se encuentran en la misma sala.

Instrucciones

Para desplazarse por el mapa, bastará con presionar las flechas del teclado o las teclas WASD (en el caso de la terminal es solamente WASD).

Para usar el arma, se presiona el espacio. Si se requiere introducir una dirección, pygame lo pedirá por pantalla (Figura 54), la cual se obtendrá a partir de la siguiente tecla presionada, mientras que en la terminal se debe incluir la dirección después del espacio. Por ejemplo, si se quiere disparar hacia arriba, habrá que introducir " w".

Figura 54. Pygame pidiendo la dirección para disparar

Por último, para escapar (salir), se debe presionar la tecla Enter, mientras que en la terminal es pulsando la "x" o escribiendo "salir". Esta acción sólo se podrá realizar si el agente está seguro de que la salida se encuentra en esa posición (debe descartar el resto de posiciones posibles para la salida anteriormente).