DnD: WEREWOLF HUNTING

PRÁCTICA FINAL DE FUNDAMENTOS DE I.A. 13 de enero de 2023

Autor: Sergio Rodríguez Vidal

Introducción:

Este proyecto consiste en la realización de un juego tematizado en el archiconocido juego de rol "Dragones y Mazmorras" en el cual se aplicarán los conceptos aprendidos en la asignatura Fundamentos de I.A.

Objetivo:

El objetivo del juego es regresar al pueblo sin morir habiendo cazado a todos los hombres lobo. **El jugador no podrá regresar al pueblo sin haber acabado con estos**. Para ello el jugador tendrá acceso a dos agentes: **un recomendador** (lógico o bayesiano) y **un libro** (agente lógico).

Primeras decisiones de diseño:

En este apartado se comentarán las decisiones iniciales y más importantes del proyecto incluyendo la estética y el estilo de programación.

Estructura:

El **pensar cómo se estructurará el código del juego** fue un **proceso muy sesudo**. El **juego debía ser una aplicación modular**. Las razones de esto son: **mayor facilidad para editar el código y detectar errores** y **poder utilizar el código del juego en futuras ocasiones**. Es por esto que el código del juego no siempre aplicará a este en sí, si no que **podría usarse indistintamente en otros proyectos** (el caso de los agentes o las entidades, por ejemplo).

El código de la aplicación se estructura de la siguiente forma (Figura 1).

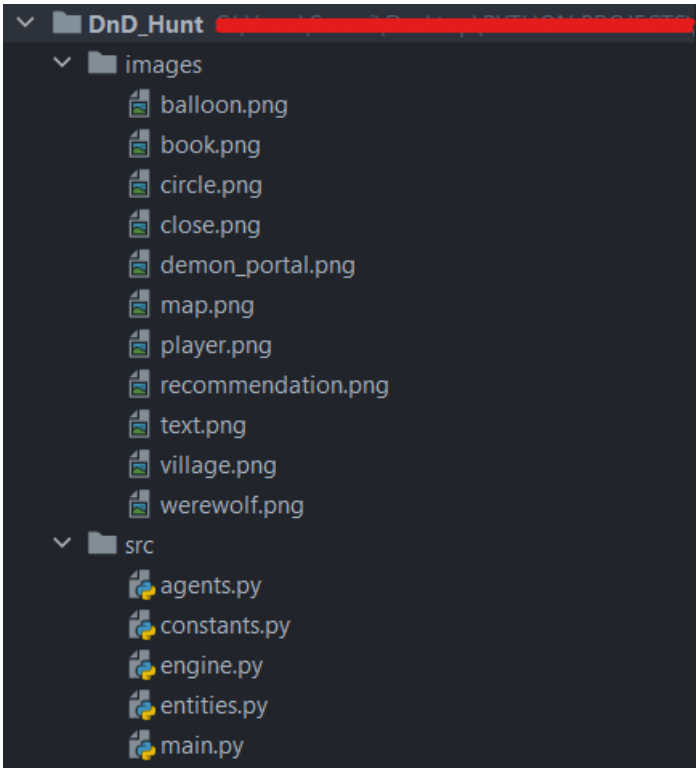

Figura 1: Directorios del juego

El directorio "**src**" es el que contiene el código fuente del juego. Este contiene varios archivos:

- **agents.py**: Este fichero contiene el código de los agentes lógicos y bayesianos.
- **constants.py**: Este fichero contiene casi todas las constantes del juego. El juego puede modificarse acorde a los gustos del usuario. Estas modificaciones se realizan cambiando los valores de este fichero (algunas constantes no se podrán modificar, estas están especificadas).
- **engine.py:** Este fichero contiene el motor del juego, por ejemplos, las clases GameState y Recommender.
- **entities.py**: Este fichero contiene todos los objetos relacionados a las entidades dentro del juego.
- **main.py:** Este fichero es el que contiene el motor gráfico del juego. Es el archivo que arranca el juego.

El directorio "**images**" contiene las imágenes que serán usadas por el motor gráfico del juego.

Tema:

La idea inicial fue basarlo acerca de la serie de Netflix Stranger Things, pero al encontrar dificultades **hallando una estética apropiada para dicho tema**, se optó por el juego de mesa que juegan los protagonistas a menudo dentro de la serie, "**Dungeons & Dragons**" o "**Dragones y Mazmorras"**.

Para caracterizar a las distintas entidades del juego se emplearon imágenes de las **miniaturas sin colorear dentro del juego** (las cuales comparten una apariencia similar entre sí, y no resulta chocante con el tablero de juego).

A raíz de este nuevo tema pudo **construirse una nueva historia** más original, **y nuevas mecánicas de juego** (todas estas conteniendo referencia a campañas del juego original).

Librerías principales empleadas:

Otro tema importante en la ideación del juego fue qué herramientas usar para el proyecto (partiendo del requisito que este debería estar programado en Python).

Pygame:

Python no es un lenguaje orientado a la creación de videojuegos. Existen otros lenguajes más potentes como C++ (Unreal) y C# (Unity) orientados hoy en día principalmente a la creación de juegos.

Pygame es un módulo de Python que realiza la tarea de un motor gráfico convencional dentro de un juego, siendo a la vez sencillo de entender y usar. Aún así es una herramienta limitada que presentó inconvenientes en la realización del proyecto, que serán exploradas en otros apartados. Aún así, Pygame permite que **un grupo más abierto de personas comprenda mejor y se entretenga más jugando al juego**, que si este hubiese jugado en una terminal de ordenador.

Numpy:

La eficiencia y rapidez dentro de un juego son importantes para mantener al usuario enganchado. Para ello, las estructuras de datos deben tener un acceso inmediato, que casualmente carecen los arrays dinámicos de Python. **Numpy permite a programadores en Python emplear estructuras de datos muy rápidas** (similares a las de C, como arrays estáticos) **y trabajar con ellas de manera eficaz** (por ejemplo, aplicar una misma operación a todos los elementos de manera instantánea).

Itertools:

Itertools es una librería preinstalada con las nuevas versiones de Python. Esta librería se emplea ocasionalmente dentro del proyecto (**algoritmo de Dijkstra y resolución lógica**).

Constants.py y Entities.py

Como se mencionó previamente, no existe una única versión del juego, **el usuario puede realizar tantos cambios desee** (dentro de lo que es modificable) **para acomodarse a su estilo de juego**.

Estos cambios se realizan dentro del archivo "constants.py". En este fichero comentaremos como un usuario puede configurar su juego.

```python
# GAME RULES #
PLAYER_HEALTH = 3
PLAYER_STARTER_WEAPON = "AXE"

WEREWOLF_HEALTH = 1
PORTAL_HEALTH = PINF  # Do not modify this

# Do not change keys withing weapons (durability, damage,...)
WEAPONS = {
    "AXE": {
        "durability": 3,
        "damage": 1,
        "miss_chance": 0.2
    },
    "PAW": {
        "durability": PINF,
        "damage": 1,
        "miss_chance": 0.
    },
    "PORTAL": {
        "durability": PINF,
        "damage": PINF,
        "miss_chance": 0.
    }
}
```

Figura 2: constants.py (parte 1)

Todas las entidades del juego son modificables. Como veremos más adelante una entidad posé un arma. **Los jugadores tienen de arma principal un hacha, "AXE", de daño 1 y durabilidad 3**. Esto significa que el jugador al atacar realiza 1 de daño y puede atacar hasta 3 veces (incluyendo las veces que el jugador falla el ataque). El **miss_chance** de un arma es la probabilidad de que el ataque no sea realizado con éxito.

El motor del juego, **engine.py**, revisará las constantes dentro de este fichero, por lo que se pide no modificar las claves dentro de diccionarios.

```python
class Weapon:
    def __init__(self, name: str, owner: object):
        if name not in WEAPONS:
            raise ValueError(f'Not valid weapon "{name}".')
        self.owner = owner

        weapon_dict = WEAPONS[name]
        self.durability = weapon_dict["durability"]
        self.damage = weapon_dict["damage"]
        self.miss_chance = weapon_dict["miss_chance"]

    def broken(self):
        return self.durability < 1

    def hit(self, target):
        if not self.broken():
            if randint(0, 100) >= (self.miss_chance * 100):
                self.durability -= 1
                target.health -= self.damage
                print(f"{self.owner} dealt {self.damage} damage to {target}.")
            else:
                self.miss(target=target)
            if self.durability == 0:
                print(f"{self.owner}'s weapon broke.")
        else:
            print(f"{self.owner}'s weapon is broken. "
                  f"{self.owner} dealt 0 damage to {target}.")
```

Figura 3: Objeto arma

Este sistema **ofrece una facilidad enorme para crear nuevas armas y mecánicas**. Una idea desechada fue crear armas que puedan ser recogidas en el mapa, para cuando el jugador se quede sin arma (se le rompa). Las armas se encontrarían en ruinas, en las que el viento sopla fuerte y haría más frío en las casillas adyacentes a las armas. El arma que quería añadirse era un **Morning Star** (tipo de arma conocida dentro del juego "Dragones y Mazmorras".

Además, **nuevas mecánicas en las armas** serían sencillas de implementar como añadir una probabilidad de hacer **un golpe crítico o daño doble** con un arma.

Igualmente, la creación de nuevas entidades sería igualmente sencilla teniendo una **clase padre** denominada **Entity**.

```python
class Entity(object):
    """
    Movable in game objects with properties and actions they can perform.
    """
    _MOVESET = Union[Dict[str, Tuple[int, int, bool]], None]

    def __str__(self):
        return f"{self.type}{self.id}"

    def __init__(self, id_: int, type_: str, health: int, weapon: Weapon = None, moveset: _MOVESET = None):
        if moveset is None:
            moveset = self.empty_moveset()
        self.id = id_
        self.type = type_
        self.health = health
        self.weapon = weapon

    @staticmethod
    def empty_moveset():
        return {}

    def alive(self):
        return self.health > 0

    def hit(self, target: "Entity" or None = None):
        if self.weapon is not None:
            if target is None:
                return self.weapon.miss()
            return self.weapon.hit(target)
        else:
            return print(f'{str(self)} has no weapons.')
```

Figura 4: Clase Entidad

Ambas clases, **Weapon y Entity**, están ocultas dentro del fichero **entities.py**.

El usuario además podrá adentrarse en configuraciones más avanzadas como la **generación de mapas** o **la consideración de probabilidades**, todo ello explicado en constants.py.

Agents.py

En este apartado se explicará el **funcionamiento y decisiones de diseño tomadas** de los objetos en el módulo agents.py.

Hay dos objetos principales dentro de agents.py**, LogicalAgent y BayesianAgent**. El agente lógico **procesa expresiones lógicas y llega a conclusiones partiendo de su base de conocimientos**. El agente bayesiano **es un agente probabilístico, que contiene una matriz de "prior"** (no fija, se modificará en el uso de este agente), **y dos matrices de verosimilitudes que depende de lo que el "jugador" sienta en una coordenada**.

Agente lógico:

El agente lógico hace uso de una subclase dentro de esta llamada **Expression**. Esta subclase se encarga de convertir clausulas lógicas **a su forma CNF** por conjuntos. Esto hace que podamos manipular fácilmente expresiones lógicas que **toman forma de listas de listas**.

Para realizar esto se hace uso de las distintas propiedades lógicas y esto traducido a código **puede ser un proceso imperfecto y muy lento**. Es por ello que hay un **número límite de intentos en procesar una expresión (MAX_ATTEMPTS_EXPR_PROCCESSING)** que una vez atravesado, la función devolverá error. Pese a esto el agente es **más que capaz de procesar expresiones complejas** y más que suficiente para el **tipo de expresiones que se emplearan en este proyecto**.

Este proceso al ser extraordinario al juego no se explicará en este documento, pero **más información acerca de cómo funciona esta clase está dentro de la documentación del mismo código**.

```python
def ask(self, clause):
    # Step 0: Check if KB is not empty
    kb = self._kb.copy()

    # Step 1: Convert (KB ∧ ¬α) to Conjunctive Normal Form
    negated_clause = self._negate_clause(clause)
    for expr in negated_clause:
        kb[len(kb)] = expr

    combined = set()
    # Step 2: Keep checking to see if we can use resolution to produce a new clause
    while True:
        new = []
        all_combined = True
        for clause1_id, clause2_id in combinations(kb.keys(), 2):
            clause1, clause2 = kb[clause1_id], kb[clause2_id]
            if (clause1_id, clause2_id) in combined:
                continue

            all_combined = False
            resolvents = self.unify(clause1, clause2)
            combined.add((clause1_id, clause2_id))

            if not resolvents:
                return True

            if resolvents != sorted(set(clause1 + clause2)) and resolvents not in list(kb.values()):
                new.append(resolvents)
        if all_combined:
            return False
        for expr in new:
            kb[len(kb)] = expr
```

Figura 5: Algoritmo de resolución

Un gran motivo por el que el **agente lógico** es lento (más aún con dimensiones mayores del tablero) es por su algoritmo de resolución, basado en el **pseudocódigo proporcionado en el tema 4 de la asignatura Fundamentos de I.A.** El **por qué** de esto es debido a que el agente debe resolver **todas las combinaciones entre las clausulas de su base de conocimientos, y las nuevas que cree**. Además, no se ha implementado una función que permita al agente lógico avanzar con el conocimiento nuevo que aprenda, debido a que **este combinado con otras clausulas,**

resultaría en expresiones lógicas vacías afectando al resultado de la función ask. Para hallar las combinaciones se utiliza la función **combinations de itertools**. Para **reducir el tiempo del algoritmo** no se repetirán resoluciones realizadas en previas iteraciones, empleando una técnica similar a la **memorización**.

Además, para reducir aún más el tiempo, en el código principal **se hace uso de 3 agentes de estos**. Aun así, debido a que **no es un agente óptimo** este no será empleado por **el algoritmo de recomendación del proyecto**. En vez de este agente se realizará una aproximación a un agente lógico mediante uno **bayesiano de probabilidades 0.99999 y 0.00001**. Pese a esto, el agente lógico no queda desechado, sino que queda como el motor del **"libro de los divinos" (el cual garantiza la existencia de una entidad en un lugar)**.

Agente bayesiano:

Cada agente bayesiano **posé una matriz de priors y dos de verosimiltud**. ¿Por qué dos matrices de verosimilitud? Porque la **condición impuesta de que haya o no una entidad cerca**, o uno o cero, **es un experimento Bernoulli**.

El cálculo del posterior en una coordenada vendrá dado por la siguiente fórmula:

$$posterior = \frac{p(no\ siento\ nada \mid Hay\ entidad\ en\ x,y) * p(Hay\ entidad\ en\ x,y\ (prior))}{p(no\ siento \mid Entidad) * p(Entidad) + \ p(no\ siento \mid No\ entidad) * p(No\ entidad)}$$

Para que el agente bayesiano funcione correctamente, **el prior debe de actualizarse con el anterior posterior**. Para ello deben modificarse las probabilidades de las coordenadas cuyos posterior han sido calculados, **y modificar toda la matriz para que el total siga sumando lo mismo**. De esto se encarga la función **modify_prior**.

```
def modify_prior(self, dictionary):
    diff = 0
    for coord, value in dictionary.items():
        # Current value of the probability
        current_val = self.prior[coord[0], coord[1]]
        # Calculate the difference and subtract it from current probability
        diff += value - current_val
        # Update the element in the specified row and col
        self.prior[coord[0], coord[1]] = value
    # Find the minimum value of the remaining elements in the matrix
    remaining_elements = self.prior[np.logical_and(self.prior != 0,
                                                   ~np.isin(self.prior, list(dictionary.values())))]
    min_val = remaining_elements.min()
    # Calculate value of parts to be shared within remaining values
    min_count = sum(remaining_elements / min_val)
    # Use the minimum value to calculate the new values for the remaining elements
    self.prior[np.logical_and(self.prior != 0, ~np.isin(self.prior, list(
        dictionary.values())))] -= diff * remaining_elements / min_count
```

Figura 6: Modificar prior

Para que las **probabilidades no estén sesgadas** se modifican a la vez y no de forma iterativa. Esta función **halla la diferencia entre los valores previos y nuevos y la reparte entre el resto de los valores**.

De todo este trabajo se encargará el recomendador que será explicado en futuros apartados.

Engine.py

En este apartado se tocarán los fragmentos más importantes del motor del juego.

Creación del tablero

Crear el tablero parece una tarea sencilla, pero no todos los tableros hacen que el jugador pueda ganar la partida. **Con la configuración base del fichero puede darse el caso que los portales cierren paso a los objetivos del jugador** (el hombre lobo y el pueblo). ¿Cómo puede prohibirse esto, para cualquier número de portales? **Creando mapas donde exista un camino a los hombres lobo** (sin pasar antes por el pueblo y portales) y otro **de un hombre lobo al pueblo**. Para ver si este camino existe, se hace uso de una mezcla de **búsqueda en anchura y un Dijkstra no pesado** con la adición de bloquear caminos (portales o pueblo) (**Figura 7: Función para hallar caminos**). Para crear estos mapas, se probará un número determinado de veces (**BOARD_GENERATION_ATTEMPTS**) a encontrar uno que cumpla estas condiciones.

Nota: Puede darse el raro caso de que, existiendo una configuración posible para los parámetros especificados por el usuario, no se encuentre dicho mapa.

```python
def find_shortest_path_length(matrix, source, destinations, valid=lambda *args: True):
    """
    Finds the shortest paths from a source to each destination. However, the resulting paths will not
    necessarily have to contain each destination (this is not a TSP implementation). This function uses
    an unweighted version of Dijkstra's algorithm.
    :parameter matrix: Matrix containing nodes and paths.
    :parameter source: Starting point of the Algorithm.
    :parameter destinations: Array containing all destinations.
    :parameter valid: Function that determines if a node is valid or not.
    :return: Length of the obtained path.
    """
    # Dictionary of distances to each node
    distances = {}
    # Set all distances to infinity
    for i in range(BOARD_HEIGHT):
        for j in range(BOARD_WIDTH):
            distances[(i, j)] = PINF

    # Set distance to source node to 0
    distances[source] = 0

    # Initialize heap queue and path and visited dictionaries. Path dictionary contains
    # the path to each node
    queue, visited, path = [], {}, {source: [source]}

    heappush(queue, (0, next(counter := count()), source))

    # Iterate over queue until all destinations have been visited
    while queue and (destinations_left := destinations.copy()):
        # Pop first element of the queue
        curr_distance, _, curr_node = heappop(queue)
        if curr_node in visited:
            continue
        visited[curr_node] = True

        # Remove destination if visited
        if curr_node in destinations:
            destinations_left.remove(curr_node)

        # Iterate through neighbours
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            neighbour = (curr_node[0] + dx, curr_node[1] + dy)
            if not out_of_bounds(matrix, (neighbour[0], neighbour[1])):
                # Skip neighbour if not valid
                if not valid(matrix, neighbour[0], neighbour[1]):
                    continue
                distance = curr_distance + 1
                # If new distance is lower than previous new distance, modify
                # neighbour's distance
                if distances[neighbour] > distance:
                    distances[neighbour] = distance
                    # Push neighbour to queue
                    heappush(queue, (distance, next(counter), neighbour))
                    path[neighbour] = path[curr_node] + [neighbour]
    return get_distance(destinations, path)
```

Figura 7: Función para hallar caminos

GameState

El objeto GameState es el que **graba el estado actual del juego y permite mover las distintas entidades.** Además, tienen otras funciones como **comprobar si quedan hombres lobo, o si el juego ha finalizado y de qué forma.**

```python
class GameState:
    def __init__(self):
        """
        This class represent the current state of the game.
        """
        self.board, self.entities = generate_board()

    class IlegalMove(Exception):
        """
        Exception raised when a specific move is not allowed within the game rules.
        Example: A movement to a coordinate out of bounds.
        """
        def __init__(self, message="Specified move is ilegal"):
            super().__init__(message)

    def move_entity(self, entity_str: str, move: str):
        e = self.entities[entity_str]
        coords = np.where(self.board == entity_str)
        x_curr, y_curr = int(coords[0]), int(coords[1])
        moveset = e.MOVESET

        # Check if move is legal. Currently, the only illegal moves are being out of bounds or
        # not being on the entity's moveset.
        if move in moveset and not out_of_bounds(self.board, (x := x_curr + moveset[move][0],
                                                              y := y_curr + moveset[move][1])):
            if (d_str := self.board[x, y]) in self.entities:
                d = self.entities[d_str]
                # If entity attacks...
                if moveset[move][2] and not isinstance(d, DemonPortal):
                    x, y = self.battle(attacker=e, defender=d, position=(x_curr, y_curr, x, y))
                # Otherwise...
                else:
                    x, y = self.battle(attacker=d, defender=e, position=(x, y, x_curr, y_curr))
                return x, y, True
            else:
                # If entity attacks...
                if moveset[move][2]:
                    e.hit(target=None)
                # Finally, move entity
                self.board[x, y] = entity_str
                self.board[x_curr, y_curr] = "--"
                return x, y, False
        else:
            raise self.IlegalMove
```

Figura 8: Estado de juego

La función **move_entity tiene la capacidad de mover cualquier entidad del juego siempre y cuando el movimiento sea legal.** ¿Qué es un movimiento legal? **Que esté dentro del conjunto de movimientos de la entidad, y no salga del tablero**.

Nota: Previamente, se pensó realizar una función donde los hombres lobo pudieran moverse, pero dicha decisión fue cancelada ya que interferiría con la predicción de los agentes (sobre todo la del agente lógico).

Recommender

Existen dos versiones del recomendador: **un recomendador lógico y otro bayesiano**. Por eficiencia del código ya se comentó en un apartado previo que **se realiza una aproximación a un agente lógico mediante un agente bayesiano de probabilidades extremas**. Pese a esto, la configuración actual juego y función están pensado para que el **recomendador sea bayesiano**, ya que se ha encontrado los parámetros exactos para tener **un recomendador lo**

suficientemente atrevido y seguro. ¿Qué significa esto? **El agente no reincidirá en volver a casillas visitadas siempre que encuentre peligro, lo que impide la aparición de bucles.** Esto se consigue añadiendo una serie de **thresholds** (en constants.py se puede encontrar más información acerca de estos). Estos thresholds son **más complejos de definir con probabilidades más extremas**, lo que lleva a la aparición de bucles inevitablemente.

```python
def recommend(self, gs: GameState, percepts: Dict[str, bool]):
    x, y = gs.where_is_player()
    probabilities = {}

    for agent_type, agent_obj in self.agents.items():
        agent_obj.modify_prior({(x, y): 0})
        feel = percepts[ENTITY_PERCEPT[agent_type]]
        probabilities[agent_type] = dict()
        for (dx, dy) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            adj_x, adj_y = x + dx, y + dy
            if not out_of_bounds(gs.board, (adj_x, adj_y)):
                # Step 1: Modify likelihood based on percepts
                agent_obj.modify_likelihood0(adj_x, adj_y, feel)
                agent_obj.modify_likelihood1(adj_x, adj_y, feel)
                # Step 2: Calculate posteriors
                posterior = agent_obj.calculate_posterior(adj_x, adj_y)
                probabilities[agent_type][(adj_x, adj_y)] = posterior
        # Step 3: Update prior
        agent_obj.modify_prior(probabilities[agent_type])
    # print(probabilities)    # Remove the hashtag to display probabilities

    if gs.are_werewolves():
        # Search minimum probability
        mean_values = {coord: (sum(probabilities[key].get(coord, 0) for key in probabilities
                                   ) / len(probabilities)) for coord in
                       set.union(*(set(val) for val in probabilities.values()))}
        min_coord = min(mean_values, key=mean_values.get)

        for value in mean_values.keys():
            if self.agents['demon_portal'].prior[value[0], value[1]] == 0:
                continue
            if mean_values[value] < THRESHOLD_MEAN:
                min_coord = value
        return min_coord
    else:
        def custom_key(coord):
            if probabilities['demon_portal'][coord] < THRESHOLD:
                return probabilities['village'][coord], -probabilities['demon_portal'][coord]
            else:
                return -PINF, -PINF

        best_coordinate = max(probabilities['village'], key=custom_key)
        return best_coordinate
```

Figura 9: Función de recomendación.

El recomendador realiza lo explicado previamente en el apartado de **Agente bayesiano. Para cada agente calcula los posteriors a base de lo que percibe del entorno en un punto del tablero, y modifica el prior con las probabilidades recién calculadas**. A partir de esto toma una decisión. Si sigue habiendo hombres lobos, **escogerá la coordenada no visitada con el mínimo índice de peligro** (si el peligro es muy alto volverá a la anterior). Si no, buscará la coordenada **con mayor probabilidad que sea un pueblo, y menor que sea un portal**.

Acerca del recomendador, este está pensado para que todas las **decisiones** que se tomen sean **dinámicas**, o sea el jugador cambie de casilla. Además, **el recomendador evitará a toda costa toparse con hombres lobo, aunque exista la necesidad de cazarlos**. Para conocer las probabilidades exactas de cada entidad en cada punto, **solo habrá que eliminar el "#" de la línea de código marcada**. Estas se imprimirán por terminal, por la complejidad de transmitir esto por el motor gráfico.

¿Cómo se juega?

Figura 10: Tablero de juego

Para explicar la interfaz del juego haremos referencia a la **Figura 10.** El **personaje** controlado por el usuario viene indicado en el **cuadrado azul,** mientras que la recomendación es el **amuleto** dentro del rectángulo **naranja**.

El jugador podrá **moverse** con las teclas **W, A, S y D.** Si este se mueve fuera del tablero, el **programa imprimirá por pantalla que dicho movimiento es ilegal**. Para atacar empleará **las flechas del teclado (que indican la dirección del ataque)**.

Es complejo ajustar que un mensaje se muestre por pantalla (en la interfaz gráfica) durante un tiempo, es por esto por lo que **el usuario deberá también observar la terminal para observar distintos mensajes** (historia, instrucciones, batallas, mensajes y fin de juego). En **rojo** vienen marcadas las **vidas restantes del jugador, y sus sentidos** que se iluminaran cuando sientan algo.

Preguntar al libro

En **amarillo** viene remarcado el libro al que el usuario podrá acceder para comprobar las entidades a su alrededor. Cuando se cliquea en este, el juego se pausa por instantes. En **verde** viene el menú que se muestra al usuario al abrir el libro, un **rectángulo y un texto (respuesta)**. Al hacer **click en el rectángulo (se marca en color amarillo)** el usuario podrá escribir **una pregunta** y al deseleccionar el **rectángulo (se marca en color negro)** se mandará esta pregunta al agente lógico.

¿Cómo se pregunta al libro? Se introducirá un texto de la siguiente forma: Primero **una letra mayúscula** (**W para hombres lobo, D para portales y V para pueblo)** y segundo unas coordenadas entre corchetes. **MUY IMPORTANTE:** Las coordenadas dadas marcan **primero la columna y segundo la fila**. Al ser una matriz, un array de arrays, primero habrá que seleccionar el **array, o sea la columna**. En la **Figura 10** el **jugador** se encontraría en la coordenada **[1, 0]** y el **amuleto** en la **[0,0].**

Para **cerrar el libro**, el **jugador deberá cliquear de nuevo en este para reanudar la partida**.

Mensajes por terminal

Como ya se explicó previamente, **por terminal se mostrará información importante** como la historia, las instrucciones o las batallas. En las **batallas** se mostrará el daño que hace el jugador y a qué, o si este ha fallado el golpe. **También las muertes si hay algún muerto**.