



Universidad  
Rey Juan Carlos

## Práctica 2: Agentes lógicos

### Test de modificaciones

#### Inteligencia Artificial

10 de noviembre - Curso 2023/2024

## Introducción

En este documento se proponen una serie de modificaciones al contenido de la Práctica 2. Se trabajará sobre la versión completa de la práctica incluida en el repositorio alojado en el GitLab de la EIF, que podrá ser descargado a través del siguiente comando:

```
git clone https://gitlab.eif.urjc.es/\[nombre\_usuario\]/\[nombre\_repositorio\].git
```

- Las modificaciones propuestas deben incorporarse sobre el archivo `logicPlan.py`, no siendo necesario aplicar ningún cambio sobre otros archivos. Una vez completada, la versión **modificada** del archivo anteriormente mencionado debe entregarse a través del espacio de entrega habilitado en el Aula Virtual.
- Se dispone de **1h 30 minutos** para abordar las modificaciones requeridas.
- El código tiene que ir obligatoriamente comentado explicando su funcionalidad. Debe ser legible y estar debidamente tabulado.
- Se utilizarán sistemas anticopia y se podrá requerir explicación individual de la práctica en caso de duda.

## 1. Lógica en el mundo de Wumpus (3,5 ptos)

Como se ha visto en clase de teoría, el mundo de Wumpus es un juego utilizado como ejemplo para ilustrar la resolución de problemas lógicos. En él, hay un personaje que se mueve por un tablero formado por una cuadrícula en busca de una pila de oro. En algunas de las casillas del tablero, hay peligros que hacen perder la partida si el personaje se adentra en ellas.

En este caso, vamos a simplificar el juego, suponiendo que el único peligro mortal para el personaje son los pozos (*pits*, en inglés). Cuando el personaje está adyacente a una casilla en la que hay un pozo, detecta una brisa (*breeze*, en inglés).

El objetivo de este ejercicio es implementar un ejemplo que permita llevar a cabo inferencia en lógica proposicional con las herramientas que se han usado en la práctica. Se parte de la situación mostrada en la figura:

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 P?	2,2 P?	3,2	4,2
1,1 V B	2,1 A OK	3,1 P?	4,1

El personaje empezó en la casilla (1,1) y detectó una brisa. A continuación, se desplazó a la casilla (2,1), donde no detectó nada (ni cayó por un pozo).

Queremos razonar sobre la existencia de pozos en las casillas (1,2), (2,2) y (3,1) usando los tipos y expresiones manejados en la práctica. Para ello, se pide:

1. Crear una función `noneIsTrue`, similar a las desarrolladas en la pregunta 2 de la práctica, que devuelva una sola expresión (Expr) en CNF que es verdadera si y solo si ninguna de las expresiones en la lista de entrada es verdadera (1 punto)
2. Crear una función `logicWumpus` sin argumentos de entrada. En ella, en primer lugar, deben crearse los símbolos `Pit_1_1`, `Pit_1_2`, `Pit_2_1`, `Pit_2_2`, `Pit_3_1`, `Breeze_1_1`, y `Breeze_2_1`. Nota: los números en estos símbolos hacen referencia a las coordenadas x e y, no al tiempo, puesto que son elementos estáticos (0,5 puntos)
3. A continuación, dentro de la misma función, crear una KB que represente los siguientes hechos mediante una única instancia de Expr (1 punto):
  - Hay brisa en (1,1) si y sólo si hay pozo en (1,2) o hay pozo en (2,1)
  - Hay brisa en (2,1) si y solo si hay pozo en (1,1) o hay pozo en (2,2) o hay pozo en (3,1)
  - No hay pozo en (1,1)
  - No hay pozo en (2,1)
  - Hay brisa en (1,1)
  - No hay brisa en (2,1)
4. Finalmente, también dentro de `logicWumpus`, llevar a cabo las llamadas a las funciones necesarias para deducir, a partir de esta KB (1 punto):
  - Si, con toda seguridad, hay pozo en (1,2).
  - Si, con toda seguridad, puede descartarse que haya pozo en (1,2).
  - Si existe al menos un modelo en el que no hay pozo en (1,2) ni en (2,2). Se pide

usar, obligatoriamente, la función `noneIsTrue` desarrollada en el apartado 1.

Usar `print()` para mostrar el resultado de las llamadas a dichas funciones. Se valorará que se comenten brevemente los resultados (como comentarios de Python dentro de la función)

Para probar ese ejercicio, se puede abrir una sesión del intérprete de Python, importar las funciones de `logicPlan.py` y llamarlas desde allí. Por ejemplo:

```
> python3
>>> from logicPlan import *
>>> logicWumpus()
>>> exit()
```

## 2. Encontrar todas las esquinas con lógica (3,5 pts)

Queremos diseñar un agente lógico similar a los desarrollados en la práctica que maneje a Pacman para cumplir el objetivo ya tratado en la práctica 1 de visitar todas las esquinas del laberinto. Para ello, se pide:

1. Crear una función `cornerVisitingPlan`, análoga a `positionLogicPlan`, donde implementaremos la funcionalidad requerida. Se recomienda comenzar desde una copia de `positionLogicPlan`. Se valorará que se razone brevemente en un comentario dentro de la función por qué tiene sentido partir de `positionLogicPlan` (0,5 pts).
2. Dentro de `cornerVisitingPlan`, crear una variable que almacene las cuatro coordenadas de las esquinas, dependiendo del laberinto. Nótese que los laberintos siempre tienen una capa externa de muros que empieza en (0,0), que no puede ser visitada por el Pacman y, por tanto, no debería usarse aquí. Podría ser útil echar un vistazo a las variables `width` y `height` empleadas en la función original `positionLogicPlan`, que contienen las dimensiones de la parte interna (intramuros) del laberinto (0,5 pts).
3. Modifica el *Goal Assertion* para que compruebe si Pacman ha estado en cada una de las esquinas en, al menos, un instante de tiempo hasta el `t` actual. Una sentencia para representar el nuevo *Goal Assertion* puede ser aquella que contiene símbolos para la posición del Pacman en cada una de las cuatro esquinas en cada instante de tiempo desde `0` hasta `t` (2,5 pts):

```
[(Pacman está en esquina 1 en el tiempo 0) ○ (Pacman está en esquina 1 en el tiempo 1) ○ ...
(Pacman está en esquina 1 en el tiempo t)] y
[(Pacman está en esquina 2 en el tiempo 0) ○ (Pacman está en esquina 2 en el tiempo 1) ○ ...
(Pacman está en esquina 2 en el tiempo t)] y ...
[(Pacman está en esquina 4 en el tiempo 0) ○ (Pacman está en esquina 4 en el tiempo 1) ○ ...
(Pacman está en esquina 4 en el tiempo t)].
```

(Los saltos de línea se han introducido sólo por legibilidad).

Notas:

- Aquí, **t** se refiere a la variable que itera sobre `range(50)` en la práctica original.
- Esta sentencia contiene  $(t+1) \times 4$  símbolos, pero puede construirse en unas pocas líneas de Python usando los bucles apropiados y las funciones `conjoin` y `disjoin`.
- Es importante resaltar que, para un correcto funcionamiento, en el paso **t** la sentencia debe contener símbolos para los tiempos 0, 1, 2... **t**, incluyendo este último.

Para probar esta parte, basta ejecutar:

```
python3 pacman.py -l tinyCorners -p LogicAgent -a fn=cornerVisitingPlan
```

No se recomienda probar en laberintos más grandes como `mediumCorners` porque el tiempo de ejecución pasa a ser muy elevado.