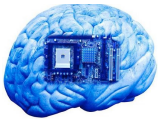
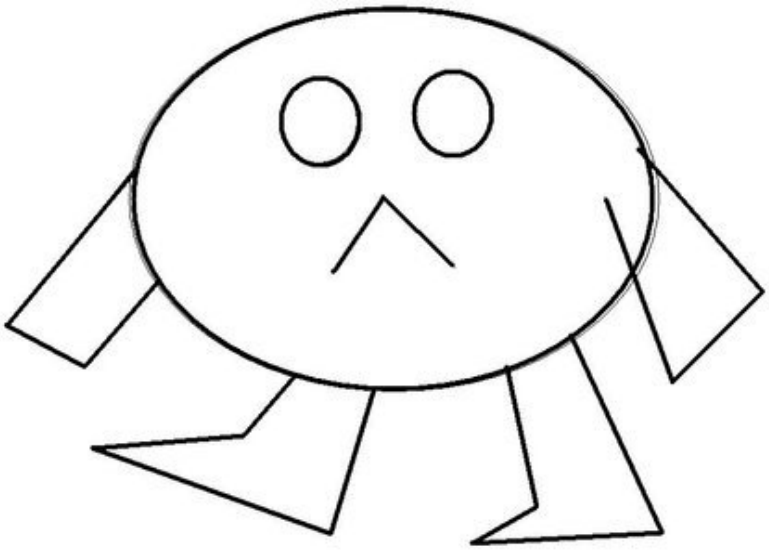
Práctica 2: Knowledge Representation (Wumpus-based)





Piedad Garrido Picazo (piedad@unizar.es)

Grado en Ing. Informática. Inteligencia Artificial 2020/2021

Contenido

Esta práctica se corresponde con el Capítulo 7 de la bibliografía básica ("*Artificial Intelligence: A Modern Approach, Chapter 7: Logical Agents*").

Objetivos de la práctica

La 2ª práctica de la asignatura se centrará en la utilización de la lógica para la resolución de problemas. El problema planteado será resuelto mediante un agente que haga uso de la lógica proposicional para representar un entorno complejo, de forma que sea capaz de inferir nuevo conocimiento a partir del conocimiento previo y de la información que se va obteniendo del propio entorno. El alumno deberá ser capaz de generar reglas de lógica proposicional que le permitan dar lugar a una base de conocimiento que será utilizada para la resolución del problema.

Agentes basados en conocimiento

El elemento central de un agente basado en conocimiento es una base de conocimiento (*knowledge base* o *KB*, en inglés), formada por una serie de sentencias o reglas. Esta base de conocimiento debe tener un método para añadir nuevas sentencias (operación TELL), y otro método que le permita consultar la información contenida (operación ASK).

La Figura 1 muestra la estructura general de un agente que emplea una base de conocimiento para decidir qué hacer ante un estado del entorno. En primer lugar, se actualiza la base de conocimiento con la información obtenida a partir de los sensores (percepciones o *perceptos*), a continuación se consulta la base para decidir la acción a realizar, y se actualiza la misma sabiendo que se va a realizar la acción indicada.

```
function KB-AGENT(percept) returns an action
    persistent: KB, a knowledge base
                t, a counter, initially 0, indicating time

    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action ← ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-PERCEPT-SENTENCE(action, t))
    t ← t + 1
    return action
```

Figura 1: Estructura general de un agente basado en conocimiento

Lógica proposicional

Las sentencias de la base de conocimiento debe expresarse de acuerdo a la sintaxis de un lenguaje de representación, de forma que todas deben ser expresiones bien formadas. Para esta práctica se va a hacer uso de un tipo de lógica simple pero suficientemente potente para el problema al que nos enfrentamos: la lógica proposicional.

La sintaxis de la lógica proposicional define las sentencias permitidas. Las sentencias atómicas están formadas por un único símbolo proposicional, que indican una condición que puede ser *cierta* o *falsa*, y se suelen representar con cadenas de caracteres que comienzan con mayúscula, como *P, Q*, *P12*, *Left*, etc. También existen las constantes *True* y *False*, y se pueden construir sentencias complejas usando paréntesis y conectores lógicos: *not* (¬ o ~), *and* (∧), *or* (∨), implicación o condicional (⇒), y bicondicional o equivalencia (⇔). Para determinar si una sentencia en lógica proposicional es cierta o no, deben emplearse las tablas de verdad propias de la lógica booleana que hacen referencia a estos operadores.

La librería de lógica proposicional que emplearemos (fichero *logic.py*) permite añadir sentencias a la base de conocimiento empleando los operadores anteriormente descritos, los cuales son: *not*, *and*, *or*, => y <=>, para representar cada conector lógico. También se admiten paréntesis para priorizar la evaluación.

Problema a resolver

El problema que resolveremos en esta práctica está basado en el mundo del Wumpus presentado en la bibliografía de referencia, sólo que por simplicidad no existirá un Wumpus en el mundo. En nuestro caso, tenemos un explorador que se adentra en una cueva a oscuras con una linterna que sólo consigue mostrar las inmediaciones del explorador, de forma que se mueve casi a ciegas. Por simplicidad, supondremos que la cueva está dividida en casillas o *habitaciones* (por similitud con el mundo del Wumpus original), inicialmente todas ellas están por explorar y se desconoce lo que ocultan. El explorador sabe que hay un tesoro oculto en la cueva, y también que hay peligrosos pozos en la cueva. Si embargo, también sabe que cerca de un pozo así se nota brisa conforme te acercas, y probablemente sea capaz de esquivar los pozos guiándose por la brisa hasta encontrar el tesoro. La Figura 2 muestra el estado inicial del mundo en cuestión.

Vamos a considerar que la brisa sólo se nota en las habitaciones situadas alrededor del pozo, sin tener en cuenta las diagonales. Por tanto, serían las casillas superior, inferior, izquierda y derecha de un pozo las que presenten brisa, como muestra la Figura 3. El objetivo es que el explorador sea capaz de llegar al tesoro sin caer a ningún pozo, siempre que sea posible.

Figura 2: Estado inicial del mundo del tesoro escondido

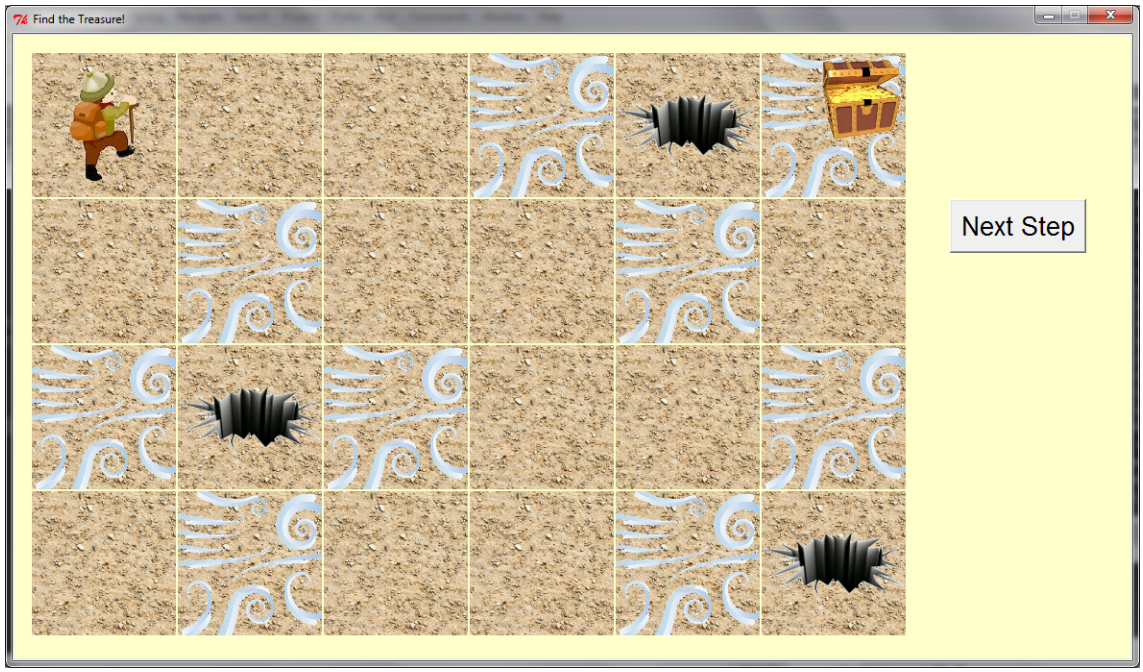


Figura 3: Ejemplo de mundo en el que se aprecia la posición del tesoro, los pozos y la brisa alrededor de los mismos

A la hora de diseñar un agente capaz de completar la tarea en cuestión, nos basaremos en el esquema general de un agente capaz de resolver el problema del Wumpus mediante lógica proposicional, como muestra la Figura 4.

function PL-WUMPUS-AGENT (*percept*) **returns** an *action*
inputs: *percept*, a list, [*stench*, *breeze*, *glitter*]
static: *KB*, a knowledge base, initially containing the "physics" of the wumpus world
x, *y*, *orientation*, the agent's position (init. [1, 1]) and orientation (init. *right*)
visited, an array indicating which squares have been visited, initially *false*
action, the agent's most recent action, initially *null*
plan, an action sequence, initially empty

update *x*, *y*, *orientation*, *visited* based on *action*
if *stench* **then** TELL(*KB*, $S_{x,y}$) **else** TELL(*KB*, $\neg S_{x,y}$)
if *breeze* **then** TELL(*KB*, $B_{x,y}$) **else** TELL(*KB*, $\neg B_{x,y}$)
if *glitter* **then** *action* ← *grab*
else if *plan* is nonempty **then** *action* ← POP(*plan*)
else if for some fringe square [*i, j*], ASK(*KB*, $(\neg P_{i,j} \wedge \neg W_{i,j})$) is *true* **or**
for some fringe square [*i, j*], ASK(*KB*, $(P_{i,j} \vee W_{i,j})$) is *false* **then do**
plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([*x*,*y*], *orientation* , [*i*, *j*], *visited*))
action ← POP(*plan*)
else *action* ← a randomly chosen move
return *action*

Figura 4: Esquema general del agente para resolver el problema del Wumpus

No obstante, puesto que nuestro problema es ligeramente distinto al original, partiremos de una versión modificada del esquema anterior para reflejar las nuevas condiciones, tal y como muestra la Figura 5.

function PL-TREASURE-AGENT (*percept*) **returns** an *action*
inputs: *percept*, a list, [*breeze*, *glitter*]
static: *KB*, a knowledge base, initially containing the "physics" of the treasure world
x, *y*, the agent's position (initially [0, 0])
visited, an array indicating which squares have been visited, initially *empty*
plan, an action sequence, initially *empty*

if *current room* not in *visited* **then**
update *visited* with *current room*
if *breeze* **then** TELL(*KB*, $B_{x,y}$) **else** TELL(*KB*, $\neg B_{x,y}$)
if *agent alive* **then** TELL(*KB*, $\neg P_{x,y}$)
if *glitter* **then** *action* ← *grab*
else if *plan* is nonempty **then** *action* ← POP(*plan*)
else if for some fringe square [*i, j*], ASK(*KB*, $(\neg P_{i,j})$) is *true* **then do**
plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([*x*,*y*], [*i*, *j*], *visited*))
action ← POP(*plan*)
else if for some fringe square [*i, j*], ASK(*KB*, $(P_{i,j})$) is *false* **then do**
plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([*x*,*y*], [*i*, *j*], *visited*))
action ← POP(*plan*)
else do
destination ← a randomly chosen fringe square [*i, j*]
plan ← A*-GRAPH-SEARCH(ROUTE-PROBLEM([*x*,*y*], *destination*, *visited*))

```
    action ← POP(plan)
update x, y based on action
return action
```

Figura 5: Esquema modificado del agente para resolver el problema de la búsqueda del tesoro

El alumno deberá realizar dos acciones principales para completar la práctica:

1) La base de conocimiento proporcionada, inicialmente vacía, debe completarse con un conocimiento básico sobre el entorno que relacione la presencia de brisa y de pozos en la cueva. Por unificar la notación, se empleará el símbolo *BXY* para indicar la presencia de brisa en la habitación (*X*, *Y*), y *PXY* para indicar que hay un pozo en la casilla (*X*, *Y*). Por ejemplo, *B12*, *P04*, etc.

Con estos símbolos deberán crearse sentencias que relacionen la brisa y los pozos. Por uniformidad, se deberá crear una sentencia por cada casilla, de forma que en la parte izquierda (antecedente) se encuentre la presencia de brisa (por ejemplo, *B00*), y en la parte derecha (consecuente) se indicará las casillas donde debería haber un pozo (por ejemplo, *P01* o *P10*). Estas sentencias se introducirán en la base de conocimiento en el método *init_kb* de la clase *PLTreasureAgent* del fichero *agents.py*. Para completarlo, se implementará el método *get_adjacent_rooms* de la misma clase.

2) Deberá implementarse el código correspondiente a la Figura 5 en el método *program* de la clase *PLTreasureAgent*. Este método debe hacer uso de la función *a_star* del fichero *search.py* para hallar el camino entre habitaciones, que debe incluir una heurística (como ya sabéis) para dirigir la búsqueda hacia el objetivo. En este caso, se debe calcular la distancia en línea recta (euclídea) entre las coordenadas de ambas habitaciones.

Ficheros de la práctica

La práctica está compuesta por los siguientes ficheros:

- `logic.py`: contiene la clase *PropKB* que representa la base de conocimiento de lógica proposicional a utilizar en la práctica, con sus métodos *ask* y *tell*.
- `search.py`: contiene las funciones para realizar la búsqueda, tal y como se implementaron en la Práctica 1. El alumno debe rellenar el código de la siguiente función:
  - `h_distance`
- `agents.py`: contiene las clases que representan tanto al agente aleatorio como al agente basado en lógica proposicional. El alumno debe rellenar el código de los siguientes métodos:
  - `init_kb`
  - `get_adjacent_rooms`
  - `program`
- `treasureworld.py`: contiene la interfaz gráfica del problema, y sirve como programa principal de la práctica, es decir, es el fichero que se debe ejecutar.

Entrega de la práctica

Deberán completare las siguientes tareas para considerar la práctica entregada:

1. Completar el código Python necesario para que el agente basado en conocimiento sea capaz de encontrar el tesoro siempre que sea posible, y mostrar cómo es capaz de resolver el problema base planteado.
2. Estudiar diferentes partidas con solución y sin ella, variando la posición y el número de pozos. Mostrar cuándo ha sido el agente capaz de encontrar el tesoro, razonando las razones por las que lo ha conseguido (o no).
3. (Opcional) Modificar la estructura del agente para que incluya una nueva acción, *GetOut*, que será la elegida por el agente cuando todas las casillas que pueda explorar estén ocupadas a ciencia cierta por pozos, de forma que no tenga que elegir una muerte segura y escape de la cueva.

La entrega de esta práctica se realizará en una única modalidad a través del ADD (plataforma Moodle). El alumno deberá realizar una memoria indicando cómo se ha realizado la implementación y los resultados obtenidos por los diferentes algoritmos. Se deberá entregar un fichero comprimido (ZIP y/o RAR) con el código de la práctica y la memoria realizada en formato PDF. La fecha límite de entrega será el **12/11/2020**.