Project 02
Wumpus World

Team member:
Nguyễn Vũ Quang Minh 1651065
Phạm Thế Quyền 1651029

1. Description

The purpose of this project is to design and implement a logical search agent and AI agent for a partially-observable environment. This will be accomplished by implementing an agent that navigates through the Wumpus World.

2. New Wumpus World

We will modify the Wumpus world as such:

- The world will be limited in **10x10**.
- The Agent can appear in any Room (**xa**, **ya**) and always facing to the right. This room is the only room has the cave door.
- There may be any number of pits and gold in the world.
- There is at least **one Wumpus**.
- The agent carries **an infinite number of arrows**.
- The agent only has a limited time to explore the rooms, before the cave door collapses and the agent becomes trap inside forever. The agent only has time to visit **150** rooms.

The score is as such:

- **Add 100** points for picking up each **gold**.
- **Reduce 100** points for shooting an **arrow**.
- **Reduce 10000** points for **dying** (by being eaten by the Wumpus, falling in a pit, or being trapped inside the cave).
- **Add 10** point for **climbing out** of the cave.
- There is **no cost** for **moving** from one room to the next.

Input: the given map is represented by a matrix, which is stored in the input file, for example, **map1.txt**. The input file format is described as follows:

- The first line contains an integer **N**, which is the size of the map.
- N next lines represent the **N ´ N** map matrix. Each line contains M integers. The number at [i, j] (row i, column j) determines the state of rooms. If the room has some things or signal such as **W**umpus, **P**it, **B**reeze, **S**tench, or **A**gent, it is marked by first

capitalized character in name of each type and written next to each other. Between two adjacent rooms is separated by a space (" "). If the room is empty, it is marked by a hyphen character (-).

Output:

Level 1: Logical Search Agent

For this level, you must implement code to explore the Wumpus World and get the highest score possible, using First-Order Logic and resolution to solve it.

Project structure:

- File **logic.py**: Implement the main logic and algorithm used for this project. The function **"findPathOfGame"** in this file returns the path of the agent.
- File **graphic.py** is used to run a graphical demonstration.
- Map file is .txt file with name: map*.txt

How to run the program

- In project directory run a command: **python main.py**. The console asks a user to choose a map and input number x which is the labeled number of map files. For example, run "python main.py" and enter "1" will run the game with "map01.txt"
- When the game stops, it prints out to console the total moves and scores of the game.

Solution:

We prioritize the moving direction of agent as: Up > Right > Left > Down. We also have a matrix to store the frequency the agent go to any rooms. The idea is to choose the direction for the next move combined with the priority above: The frequency must be distributed equally for adjacent rooms to exploit information better and avoid loop.

When the next move is specified, every time the Agent enters a room, we build a knowledge base (KB) based on the state of the room the agent is inside. There are two cases to consider:

When checking the state of the current room, we have two lists:
- The first one is **"Or"** list which stores literals. The "Or" operation will be performed for all literals in that list. For example: **o = [A, B, C]** so we use **o** as **(A ∨ B ∨ C)**.
- The second one is **"And"** list which also stores literals. The "And" operation will be performed for all **negation** of literals in that list. For example, **a = [A, B, C]** so we use **a** as **(~A ∧ ~B ∧ ~C).**

The format of literal's label is: P_ij or W_ij where P presents Pit, W presents Wumpus, (i,j) is the index of the current room:

- P_i,j: This room has pit or not
- W_i,j: This room has wumpus or not

Any room (i, j) that the agent can currently stand inside is the safe room which does not have Wumpus or Pit inside. That statement is certainly true so we append two literals to **"And"** list: W_ij and P_ij immediately before checking a state of the current room. Otherwise, we gain information about the current room and adjacent rooms to update the KB like the following:

Case 1: room[i][j] = '-'

This means room (i, j) is safe and we are also sure that Wumpus and Pit are not in adjacent rooms. Therefore, we put all new literals to the "And" list.

We update KB like this:

KB = KB $\wedge$ (~P_i+1,j $\wedge$ ~P_i-1,j $\wedge$ ~P_i,j+1 $\wedge$ ~P_i,j-1 $\wedge$
~W_i+1,j $\wedge$ ~W_i-1,j $\wedge$ ~W_i,j+1 $\wedge$ ~W_i,j-1)

Notably, we also check the validity of rooms index to put appreciate literals to the list. Ant literals whose name has invalid (i, j) won't be put to list. The above sentence is just a full case when 4 rooms next to the current room are valid. Similarly for cases:

Case 2: room[i][j] = 'B'

This means room (i, j) is also safe but any adjacent rooms may have the Pit and certainly, those rooms don't have Wumpus. Therefore, we put literals related to "P" to Or list and literals related to "W" to the And list. We do the same thing with the above case for room[i][j] = 'S'.

Update KB:

KB = KB $\wedge$ ((P_i+1,j $\vee$ P_i-1,j $\vee$ P_i,j+1 $\vee$ P_i,j-1) $\wedge$
~W_i+1,j $\wedge$ ~W_i-1,j $\wedge$ ~W_i,j+1 $\wedge$ ~W_i,j-1)

With room[i][j] = 'S', the update would be:

KB = KB $\wedge$ ((W_i+1,j $\vee$ W_i-1,j $\vee$ W_i,j+1 $\vee$ W_i,j-1) $\wedge$
~P_i+1,j $\wedge$ ~P_i-1,j $\wedge$ ~P_i,j+1 $\wedge$ ~P_i,j-1)

Case 3: room[i][j] = 'BS' or room[i][j] == 'BSG' (or any swapping of characters)

This means room(i, j) is safe but any adjacent rooms can contain the Pit or Wumpus and we are not sure about that. Therefore, we put literals of both "W" and "P" to Or list.

Update KB with a sentence:

KB = KB $\wedge$ (P_i+1,j $\vee$ P_i-1,j $\vee$ P_i,j+1 $\vee$ P_i,j-1 $\vee$
W_i+1,j $\vee$ W_i-1,j $\vee$ W_i,j+1 $\vee$ W_i,j-1)

For case 1, the room is “totally” safe, means it is empty or an escape room so after update a KB, we just need to find the suitable next move among adjacent rooms. However, for cases 2, 3, we need to check which adjacent rooms are safe to go in. By using inference based on updated KB, we can define which rooms don’t contain both Wumpus and Pit and put it in a temporary safe list. If that safe list is empty, we return to the previous room we entered to find another way. Otherwise, for each suitable next moves with the priority: Up > Right > Left > Down, if exists room is in the safe list, we choose that room to go next immediately.

If the agent doesn’t move, means it cannot find any safe rooms to move further so we end discovery and let the agent moves back to the escape room. We have a BFS function to find the way home from the current room to escape room with the opened rooms (the rooms have been passed)

Level 2: Try With Learning

In this level, my group tried to research some machine learning to apply to Wumpus World. One of the algorithms we use for this report is **Q-Learning**. We assume that the reader has basic knowledge of reinforcement learning. **Q-Learning** is a technique to assess which actions should follow based on an **action-value function**. This function determines the value when in a certain state and follows a certain action in that state.

We have the **function Q** takes the input as a table and an action, and then returns the expected reward for that action (and all subsequent actions) in this state. Before exploring the game, Q will give a **Q-table** with the **same fixed value** (optional). After that, when exploring the map of Wumpus game more, Q will give us a better and better approximation of the value of **action a** (left, right, up, down, shoot an arrow) at the **state s**. We **update the Q function** when we move.

The expression explains this very clearly. It shows how we update Q’s value based on the reward received from the game:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Source: Wikipedia

Ignore the **discount factor** by replacing it by **1**. You have to remember that Q represents the total number of rewards when choosing Q and these actions are optimal.

We will explain the expression from left to right. When we use action at the state st, we update the value of **Q(st, at)** by adding an expression to it. This expression includes:

- **Learning rate:** this variable shows how you want to update values. When alpha is close to 0, we will not update actively. When alpha is close to 1, we simply replace the old value with the newly updated value.
- **Reward rt:** It is the reward we get when we use the action at the state st. So we add this reward to our old estimates.
- **maxQ(st+1, a):** We also add a predictable reward in the future, which is the maximum reward can be achieved before any action can be done at state st+1.
- **-α*Q(st, a):** Finally, we subtract the value of Q to ensure the difference in prediction only increases or decreases (of course this difference is multiplied by alpha)

For now, we have estimated the value for each pair of states - actions, we can choose which action to take based on our strategy (no need to choose an action which gives the best reward), we should choose randomly action to build the Q table with all optimize Q value.

In Wumpus World, we can use Q-Learning to find the value of each position in the Q-table represent the map and the value of actions (up, down, left, right, shoot an arrow) at each location by running all maps randomly and many times. Therefore, we can use the strategy based on Q-Learning to choose what the agent really has to do at each step.

Reference:

https://forum.machinelearningcoban.com/t/tutorial-implement-cac-thuat-toan-reinforcement-learning-ddpg-bai-1/3276

https://www.cc.gatech.edu/~bhroleno/cs6601/wumpus.pdf