جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⵉⵏ

AL AKHAWAYN UNIVERSITY

CSC 4301, Project 2:

Logical agent for the Wumpus game.

Pr. Tajjedine Rachidi

Done by: Abdelhamid El Hand

Ziad El Ismaili

Imadeddin Ouahidi

Al Akhawayn University in Ifrane

Wednsday, June 29, 2022

Table of Contents

I. Introduction:

The Wumpus game or Hunt the Wumpus as it is also called is a simple adventure game where the hunter should escape all the pits and obstacles, kill the monster named the Wumpus, and take the gold. This game might look simple and easy at first, but it requires logic and memory to successfully complete it. To be able to move in the dark cave you the hunter has sensors to help him identify where the danger is. Those sensors are Breeze, Stench, Glitter, and Scream. Those parameters indicates the presence of danger or opportunity.

Therefore, in our version of the Wumpus game we tried to capture and represent the rules and configurations of the game into a single iteration for a logical agent. The agent main purpose is to answer the queries it is asked and not try to figure out the best course of action to win the game. To win the game the player should ask questions that will ensure finding the gold, killing the Wumpus, and staying alive.

After completing the work on the agent, we will apply it on different configurations of the world and see whether our agent will be able to work properly. After testing the agent using different configurations of the world, we will tabulate and analyze the result to see how efficient our agent.

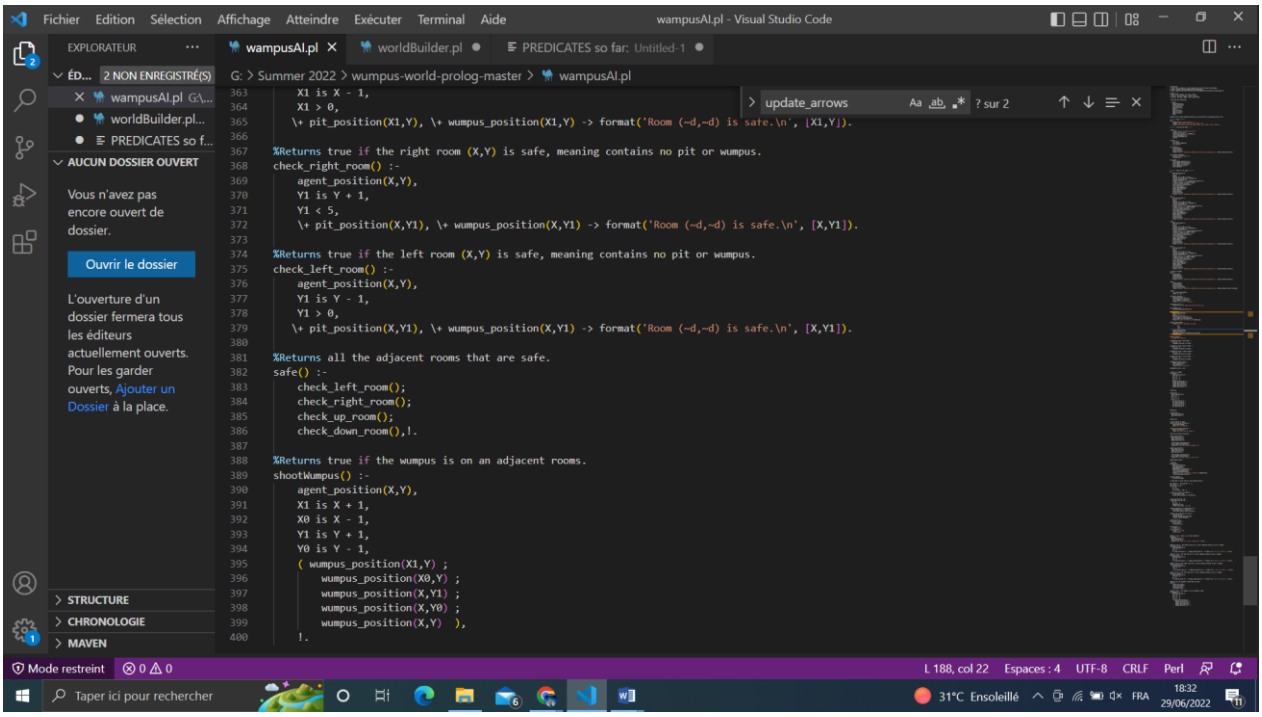
II. Key Predicates and Variables:

To have a better understanding of how the agent works we need to have a clear idea about all the predicates and variable that were used to code this agent. First, let us start by the predicates:

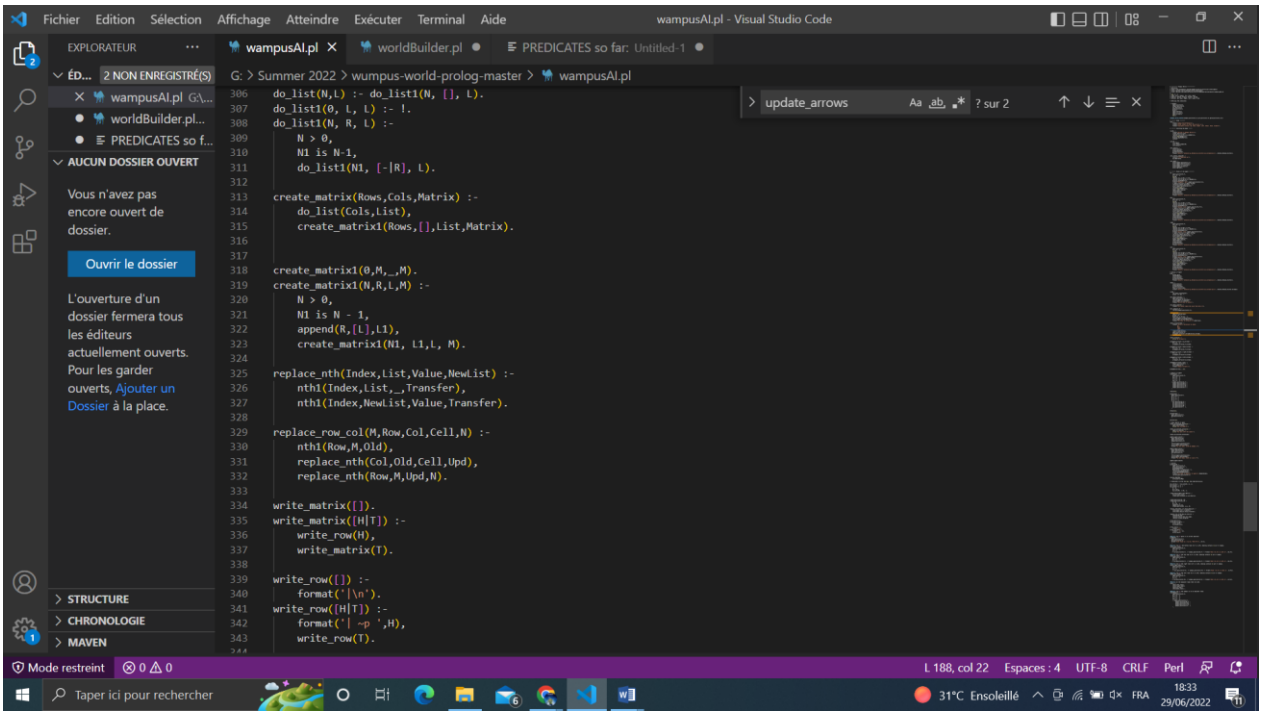
1. **Breeze():** This predicate is initialized to false, then it becomes true if and only if there is a pit in one of the adjacent rooms to the player.
2. **Pit_Position(X,Y):** this predicate return true if the position given as parameter contains a pit, otherwise return false.
3. **Wumpus_Position(X,Y):** This predicate is responsible of returning true if the position given as a parameter contains a Wumpus.
4. **Stench():** This predicate is initialized to false, then it becomes true if and only if there is a Wumpus in one of the adjacent rooms to the player.
5. **Glitter ():** This predicate requires the agent and the gold being in the same position for it to be true.
6. **Adjacent(X,Y):** This predicate gives the adjacent room to the room identified by the parameters.
7. **Grabgold():** This allows the player to grab the gold and increase the player's score.

8. **ShootWumpus():**and will return true if the Wumpus is in one of the adjacent rooms and false if not.
9. **Safe():**will return all the rooms (X,Y) that safe from the given starting position.
10. **Check_left_room/ ceck_right_room/ check_up_room/ check_down_room:** All those predicates check one of the four adjacent rooms for both Wumpus and pit.
11. **Agent_Position(X,Y):** This predicates return the position of the hunterwhen asked.
12. We have also implemented other function related to different aspects of the game, for example: **Shoot,Update_Players_Health, Display_Map, or Climb…**

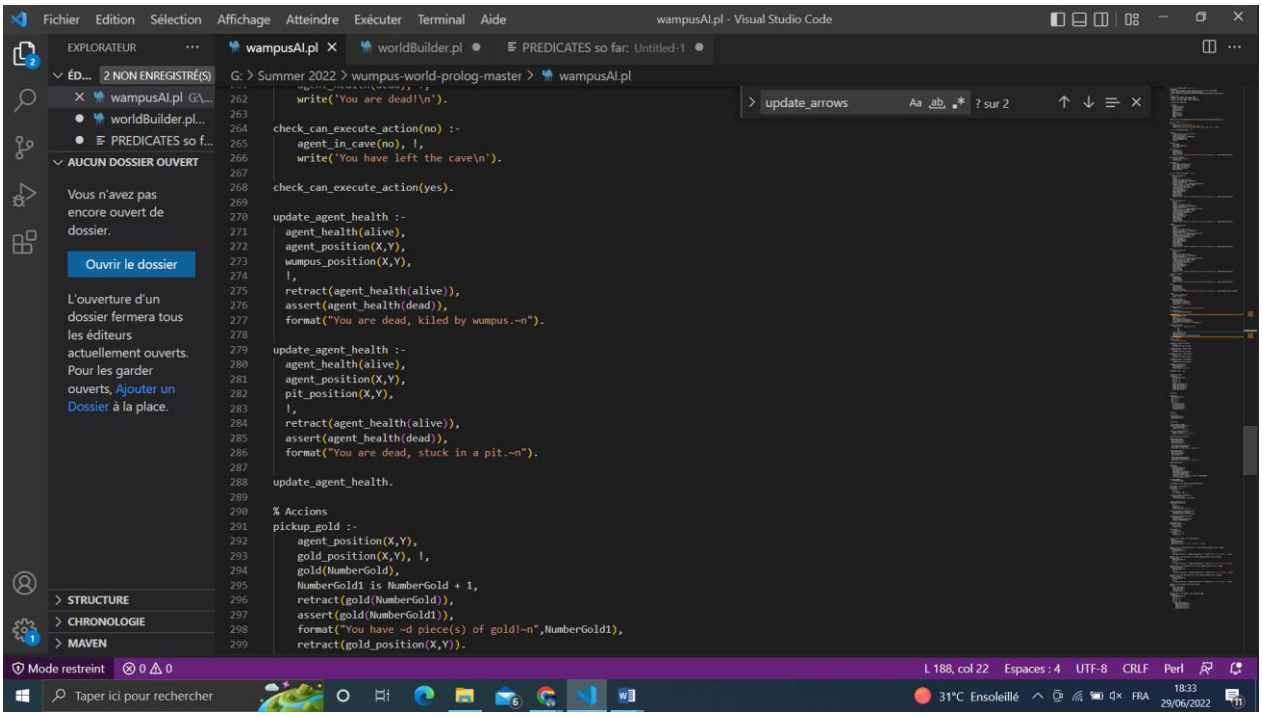
Below, find some snapshots showing the implementation of the logic behind each predicate.



```
%Returns true if the right room (X,Y) is safe, meaning contains no pit or wumpus.
check_right_room() :-
    agent_position(X,Y),
    Y1 is Y + 1,
    Y1 < 5,
    \+ pit_position(X,Y1), \+ wumpus_position(X,Y1) -> format('Room (~d,~d) is safe.\n', [X,Y1]).

%Returns true if the left room (X,Y) is safe, meaning contains no pit or wumpus.
check_left_room() :-
    agent_position(X,Y),
    Y1 is Y - 1,
    Y1 > 0,
    \+ pit_position(X,Y1), \+ wumpus_position(X,Y1) -> format('Room (~d,~d) is safe.\n', [X,Y1]).

%Returns all the adjacent rooms that are safe.
safe() :-
    check_left_room();
    check_right_room();
    check_up_room();
    check_down_room(),!.

%Returns true if the wumpus is on an adjacent rooms.
shootWumpus() :-
    agent_position(X,Y),
    X1 is X + 1,
    X0 is X - 1,
    Y1 is Y + 1,
    Y0 is Y - 1,
    ( wumpus_position(X1,Y) ;
        wumpus_position(X0,Y) ;
        wumpus_position(X,Y1) ;
        wumpus_position(X,Y0) ;
        wumpus_position(X,Y)  ),
    !.
```



```
do_list(N,L) :- do_list1(N, [], L).
do_list1(0, L, L) :- !.
do_list1(N, R, L) :-
    N > 0,
    N1 is N-1,
    do_list1(N1, [-|R], L).

create_matrix(Rows,Cols,Matrix) :-
    do_list(Cols,List),
    create_matrix1(Rows,[],List,Matrix).


create_matrix1(0,M,_,M).
create_matrix1(N,R,L,M) :-
    N > 0,
    N1 is N - 1,
    append(R,[L],L1),
    create_matrix1(N1, L1,L, M).

replace_nth(Index,List,Value,NewList) :-
    nth1(Index,List,_,Transfer),
    nth1(Index,NewList,Value,Transfer).

replace_row_col(M,Row,Col,Cell,N) :-
    nth1(Row,M,Old),
    replace_nth(Col,Old,Cell,Upd),
    replace_nth(Row,M,Upd,N).

write_matrix([]).
write_matrix([H|T]) :-
    write_row(H),
    write_matrix(T).

write_row([]) :-
    format('|\n').
write_row([H|T]) :-
    format('| ~p ',H),
    write_row(T).
```



The whole code is available on Github for more details.

Now moving on to the second part which is the different variables we used in our code.

When it comes to rooms and locations, variables X and Y refer to the X and Y coordinates representing the room.

In addition, we used yes and no to initialize different predicates.

Also, we used variable L or [H, T] to create lists that we used in multiple cases in which H is head and T is tail of the list.

NewMatrix that passes on a matrix as a parameter.

Moreover, cols and rows were passed by to identify the rows and columns of the world.

Finally, one of the interesting features of Prolog language is the ability to pass other predicates as parameters to other predicates, i.e. propagate_arrow(X, Y,ArrowDirection,Scream), assert(agent_position(X,Y1))…

Now we will visit some example to illustrate the use of those variables in our code:

```
%Returns true if gold is in current position
grabGold() :-
  agent_position(X,Y),
  gold_position(X,Y),
  format('Gold found in: (~d,~d), PICK IT!\n', [X,Y]),
  !.
```

Using X and Y to localize the agent and rooms.

```
  agent_position(X,Y),
  gold_position(X,Y), !,
  gold(NumberGold),
  NumberGold1 is NumberGold + 1,
  retract(gold(NumberGold)),
  assert(gold(NumberGold1)),
```

This predicates takes away the gold from the world after the hunter picks it up depending on the number of the gold.

```
create_matrix(Rows,Cols,Matrix) :-
    do_list(Cols,List),
    create_matrix1(Rows,[],List,Matrix).
```

In this case, we are creating the matrix based on the rows we have.

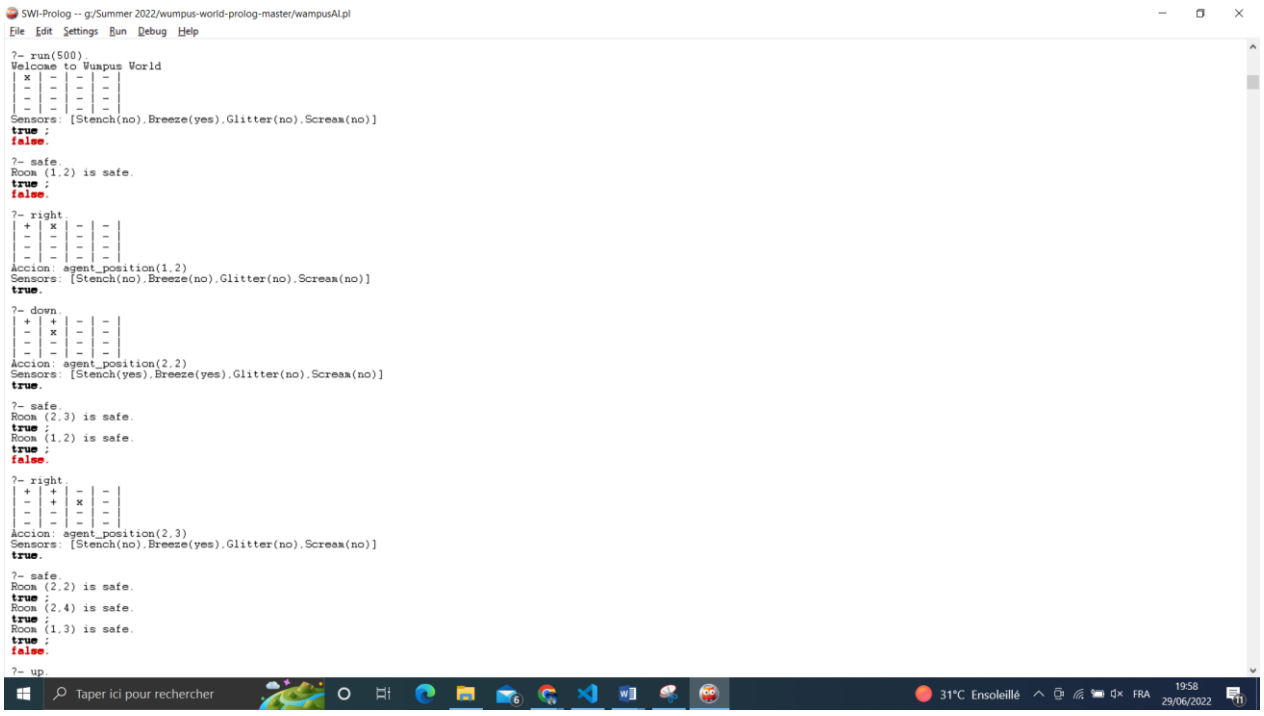
While do_list realizes creates the list based on the columns.

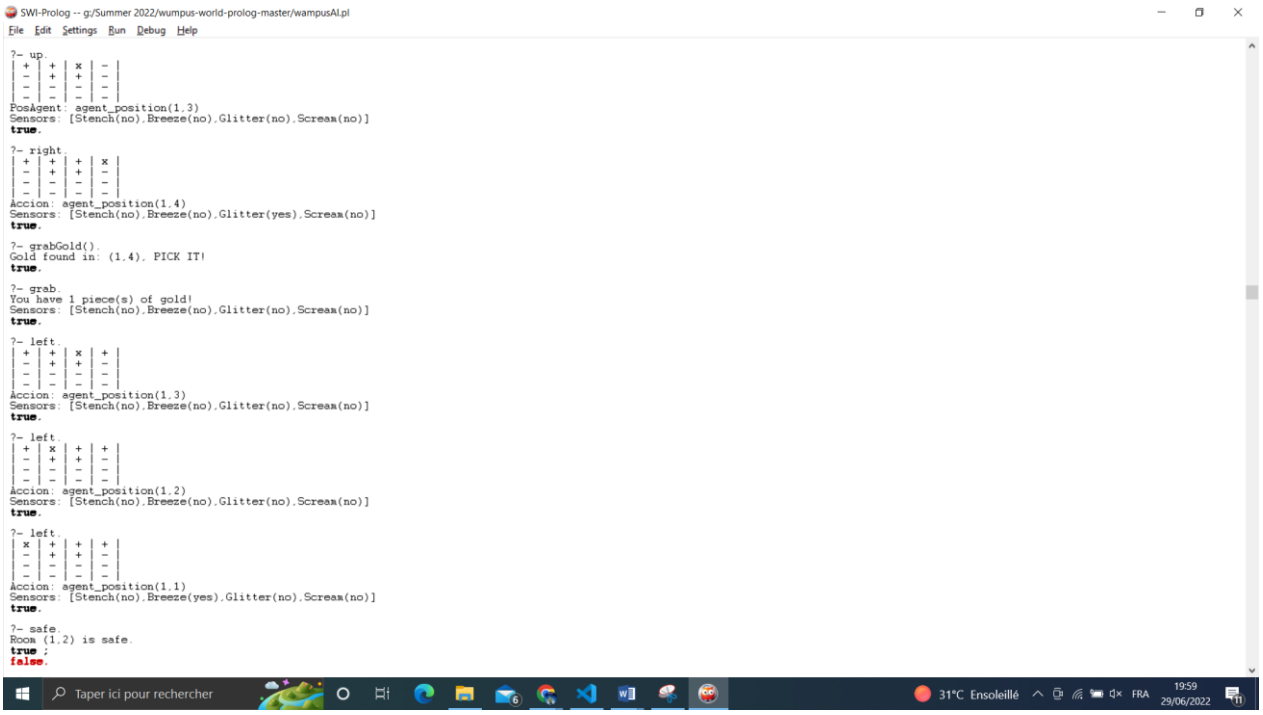
Note: The code has comments to further explain its logic.

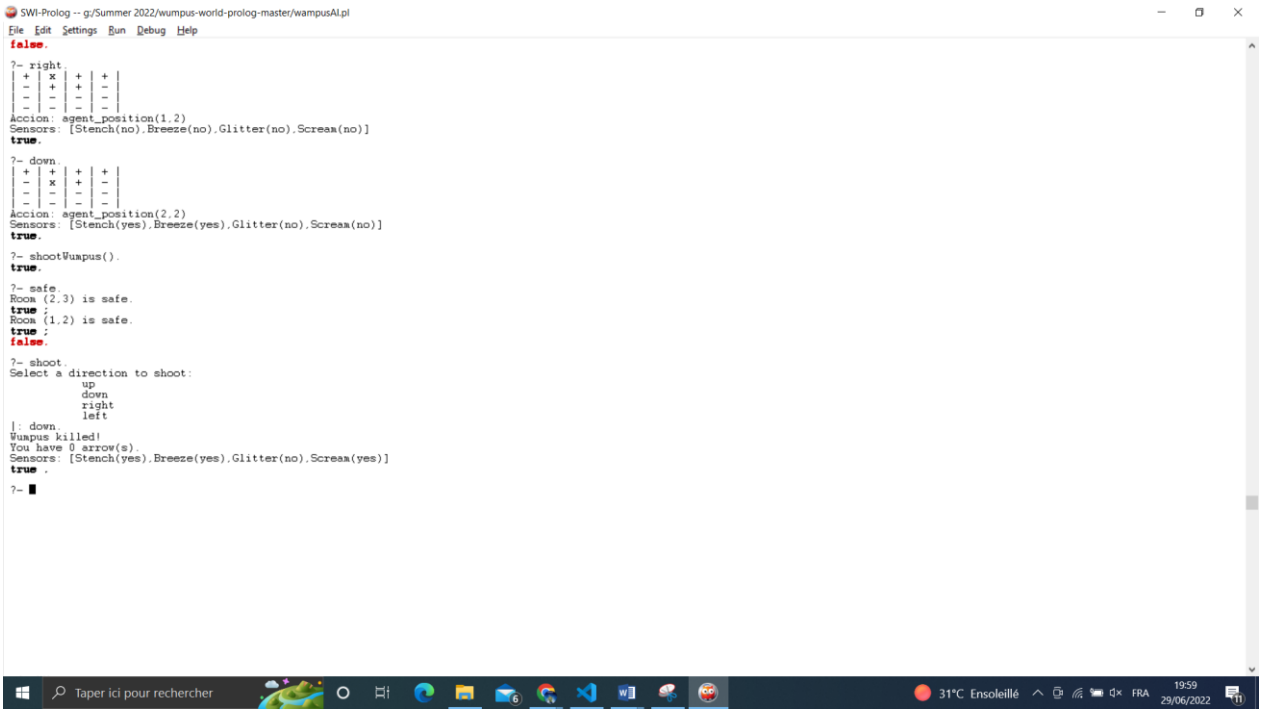
III. Testing Phase:

As we said before, we need to play the game using the agent and evaluate its performance based on the outcome of each round with each configuration.

```
buildWorld(500) :-
    asserta(pit_position(2,1)),
    asserta(pit_position(3,1)),
    asserta(pit_position(3,3)),
    asserta(pit_position(4,2)),
    asserta(gold_position(1,4)),
    asserta(wumpus_position(3,2)).
```







This shows that our agent is capable of the following things:

1. Showing the safe rooms from Wumpus and Pits.
2. Detecting where the pits are located in adjacent rooms.
3. Returning true if the Wumpus is in adjacent room
4. Returning true if the gold is in the same room as the player

5. Grabbing the gold from the world
6. Killing the Wumpus if the player chooses right direction
7. Other functionalities that weren't demonstrated in this particular case.

Moving to another example:

```
buildWorld(504) :-
    asserta(pit_position(1,2)),
    asserta(pit_position(1,4)),
    asserta(pit_position(3,2)),
    asserta(pit_position(4,2)),
    asserta(pit_position(4,3)),
    asserta(pit_position(4,4)),
    asserta(gold_position(2,3)),
    asserta(wumpus_position(1,2)).
```

```
?- run(504).
Welcome to Wumpus World
| x | - | - | - |
| - | - | - | - |
| - | - | - | - |
| - | - | - | - |
Sensors: [Stench(yes),Breeze(yes),Glitter(no),Scream(no)]
true ;
false.

?- safe.
Room (2,1) is safe.
true.

?- shootWumpus().
true.

?- shoot.
Select a direction to shoot:
            up
            down
            right
            left
|: right.
Wumpus killed!
You have 0 arrow(s).
Sensors: [Stench(yes),Breeze(yes),Glitter(no),Scream(yes)]
true .

?- down.
| + | - | - | - |
| x | - | - | - |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(2,1)
Sensors: [Stench(no),Breeze(no),Glitter(no),Scream(no)]
true.

?- safe.
Room (2,1) is safe.
true ;
Room (2,3) is safe.
true ;
false.

?- right.
| + | - | - | - |
| + | + | x | - |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(2,3)
Sensors: [Stench(no),Breeze(no),Glitter(yes),Scream(no)]
true.

?- grabGold().
Gold found in: (2,3), PICK IT!
true.

?- grab.
You have 1 piece(s) of gold!
Sensors: [Stench(no),Breeze(no),Glitter(no),Scream(no)]
true.
```

Another example:

```
buildWorld(507) :-
    asserta(pit_position(2,2)),
    asserta(pit_position(2,3)),
    asserta(pit_position(3,1)),
    asserta(pit_position(3,3)),
    asserta(gold_position(4,4)),
    asserta(wumpus_position(1,2)).
```

```
?- run(507).
Welcome to Wumpus World
| x | - | - | - |
| - | - | - | - |
| - | - | - | - |
| - | - | - | - |
Sensors: [Stench(yes),Breeze(no),Glitter(no),Scream(no)]
true ;
false.

?- shootWumpus().
true.

?- shoot.
Select a direction to shoot:
            up
            down
            right
            left
|: right.
Wumpus killed!
You have 0 arrow(s).
Sensors: [Stench(yes),Breeze(no),Glitter(no),Scream(yes)]
true .

?- safe.
Room (1,2) is safe.
true ;
Room (2,1) is safe.
true.

?- right.
| + | x | - | - |
| - | - | - | - |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(1,2)
Sensors: [Stench(no),Breeze(yes),Glitter(no),Scream(no)]
true.

?- safe.
Room (1,1) is safe.
true ;
Room (1,3) is safe.
true ;
false.
```

```
?- right.
| + | + | x | - |
| - | - | - | - |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(1,3)
Sensors: [Stench(no),Breeze(yes),Glitter(no),Scream(no)]
true.

?- safe.
Room (1,2) is safe.
true ;
Room (1,4) is safe.
true ;
false.

?- right.
| + | + | + | x |
| - | - | - | - |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(1,4)
Sensors: [Stench(no),Breeze(no),Glitter(no),Scream(no)]
true.

?- safe.
Room (1,3) is safe.
true ;
Room (2,4) is safe.
true.

?- down.
| + | + | + | + |
| - | - | - | x |
| - | - | - | - |
| - | - | - | - |
Accion: agent_position(2,4)
Sensors: [Stench(no),Breeze(yes),Glitter(no),Scream(no)]
true.

?- safe.
Room (1,4) is safe.
true ;
Room (3,4) is safe.
true.

?- down.
| + | + | + | + |
| - | - | - | + |
| - | - | - | x |
| - | - | - | - |
Accion: agent_position(3,4)
Sensors: [Stench(no),Breeze(yes),Glitter(no),Scream(no)]
true.

?- safe.
Room (2,4) is safe.
true ;
```

```
?- safe.
Room (2,4) is safe.
true ;
Room (4,4) is safe.
true.

?- down.
| + | + | + | + |
| - | - | - | + |
| - | - | - | + |
| - | - | - | x |
Accion: agent_position(4,4)
Sensors: [Stench(no),Breeze(no),Glitter(yes),Scream(no)]
true.

?- grabGold().
Gold found in: (4,4), PICK IT!
true.

?- grab.
You have 1 piece(s) of gold!
Sensors: [Stench(no),Breeze(no),Glitter(no),Scream(no)]
true.
```

In order not to make the report long and redundant, I will summarize the results and findings of the tests in a table.

	Win Case	**Gold Not Found**	**Wumpus Not Killed**	**Loss Case**	**Unwinnable**
Number of Cases	53	19	17	27	10
Probability %	66.25	23.75	21.25	33.75	12.5

- By trying around 80 random configuration, the agent finished with an estimated win-rate of 66%, which is good for a single step logical agent that only answers queries and does not suggest paths. Due to some limitations that will be discussed later on, the win-rate is not higher than 2/3. To win you have to kill the Wumpus, but you may take or not take the gold.

- The agent was very efficient in (76%) finding the gold, and the few cases he couldn't achieve the task were because the gold was unreachable.

- Not being able to kill the Wumpus in this case refers to missing the arrow because we had more than one certain direction to shoot at. Yet, this rate is low and can be easily reduced further with some modifications.

- Unwinnable cases represent the cases that no matter what you do, you cannot win. If the Wumpus is separated from, the hunter by pits or the player is trapped between pits from the start… Those cases nonetheless are rare and cannot be solved.

- Finally, Loss-rate represents all the cases we couldn't win either because it is impossible or we missed the Wumpus. 1/3 of the cases result in loss, but with few modifications we can almost eliminate the cases where we miss the Wumpus.

The numbers are just an estimation of how well the agent will perform. Also, in the cases of missing the Wumpus we are not using the player's observation or memory to win that is why we sometimes miss.

IV. Analysis and Limitations:

1. Analysis:

The agent performs well in various aspects related to the Wumpus game. The agent is very efficient in the process of finding gold. Also, the agent answers accurately the queries. There is no worry to fall in a pit or be killed by the Wumpus if we use safe. Additionally, it gives correct answer if you ask about the Wumpus and it is in an adjacent cell. Shooting, grabbing, or moving work very well. It requires almost no effort and thinking from the player, he only needs to follow the directions the agents gives based on the queries. By having a human contribution in these games the result are slightly biased.

2. Limitations:

The agent in fact has few limitations that can be adjusted in the future.

- When it comes to identifying where exactly the Wumpus is, sometimes the agent is not enough and the player needs to remember some details about the previous rounds to be able to find the Wumpus.
- The agent answers very well the queries, but to win the game the player needs to do some thinking himself and memorize what happened in the previous rounds.

- When we query for safe, we get false when the sub-predicates of safe return false, but in few cases false stops us from checking all the adjacent cases (checking only 2 from 4 rooms).
- We need to add a way for the agent to perform better inference so that we don't need to think or memorize any information at all.
- The cases that actually beat our agent are the ones that are unwinnable either because the Wumpus is not reachable or the player is trapped.

V. Summary:

The agent was able to perform very well and can be even better with small help.

We were able to get 66.25% as an approximate win-rate for this particular agent.

The agent is able to win in most configurations easily and without help from the player.

The agent is efficient when it comes to answering the requested queries.

Nonetheless, we have few limitations that can be fixed in more time.