1. Method

To reduce the difficulty of programming, better test the two methods, compared with the original Wumpus World made the following assumptions:

Only one wumpus

Only two pits

There are stench around Wumpus

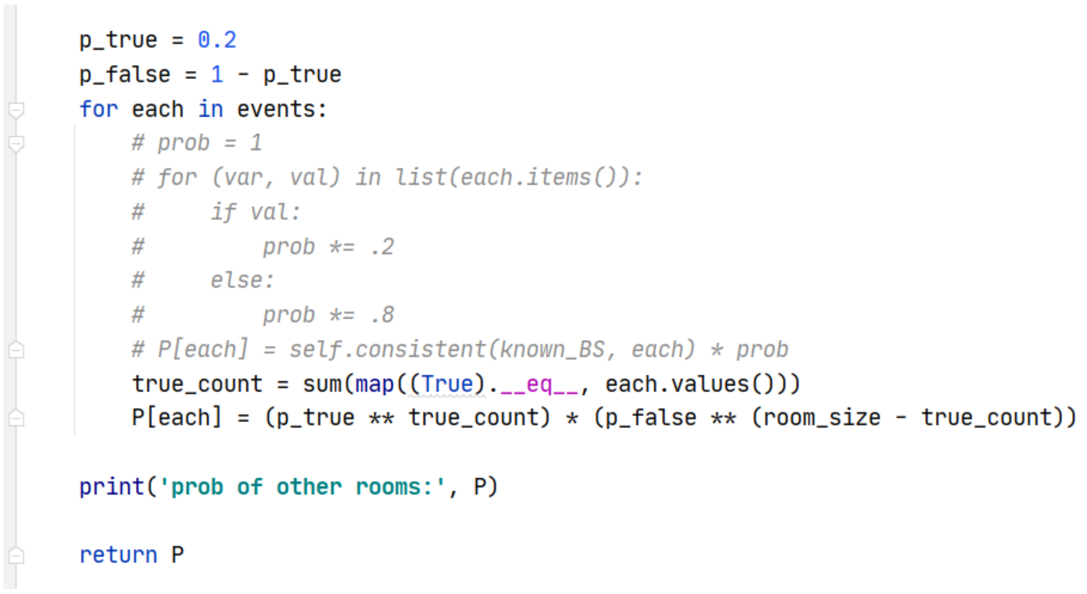
There is breeze around Pit

Agent cannot shoot

The world size is 4x4

2. Modicification

Probability_based_move.py



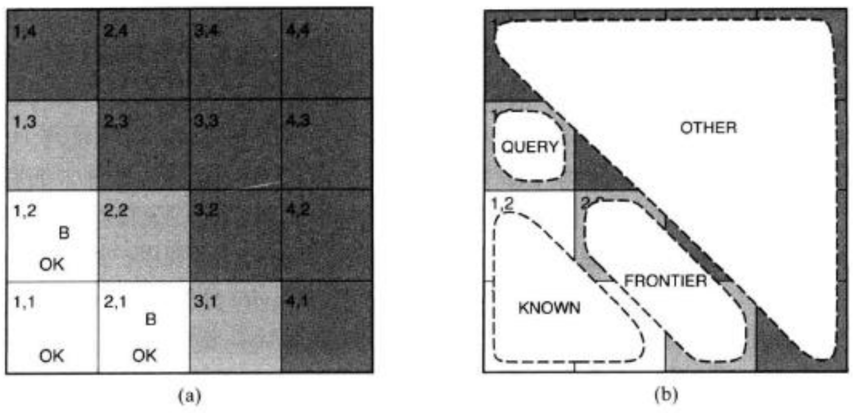
```
p_true = 0.2
p_false = 1 - p_true
for each in events:
    # prob = 1
    # for (var, val) in list(each.items()):
    #     if val:
    #         prob *= .2
    #     else:
    #         prob *= .8
    # P[each] = self.consistent(known_BS, each) * prob
    true_count = sum(map((True).__eq__, each.values()))
    P[each] = (p_true ** true_count) * (p_false ** (room_size - true_count))

print('prob of other rooms:', P)

return P
```

PitWumpus_probability_distribution()

Modified the calculation method of probability, instead of using breeze_stench, the probability of the surrounding square is used. If the surrounding blocks contain traps, the probability of safety is 0.2, otherwise it is 0.8.

Next, using the fully joint probability, calculate the probability of each square.

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4},B_{1,1},B_{1,2},B_{2,1}) = \mathbf{P}(B_{1,1},B_{1,2},B_{2,1} \mid P_{1,1},\ldots,P_{4,4})\mathbf{P}(P_{1,1},\ldots,P_{4,4}) .$$

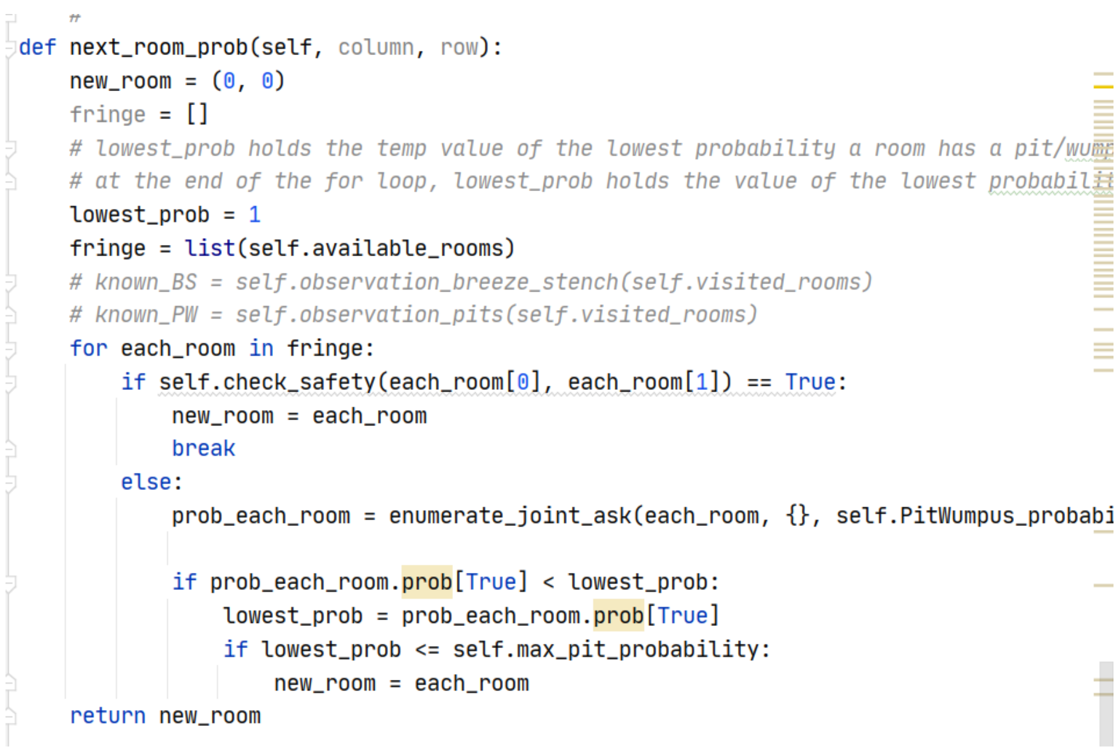


The first is the conditional probability distribution of breeze configuration for a given pit configuration; The value is 1 if the wind is adjacent to the pit, and 0 otherwise. this

The second is the prior probability of pit configuration. Each square contains a pit with a probability of 0.2, independent of the other squares; So,

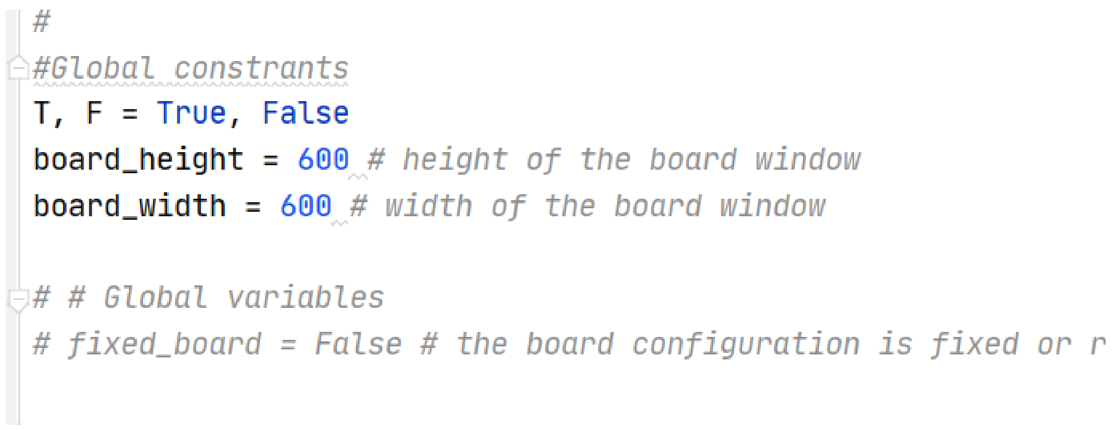
$$\mathbf{P}(P_{1,1},\ldots,P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) .$$

Next, modify next_room_prob() to comment out the known_BS and known_PW functions that were originally used



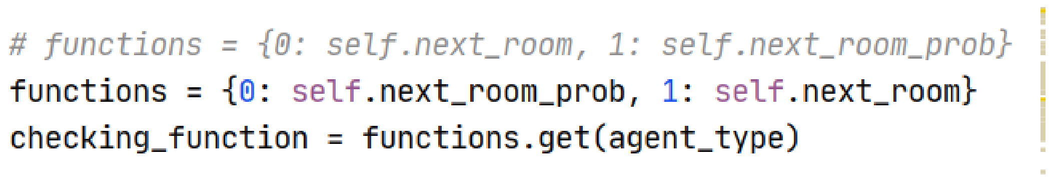
```python
    #
def next_room_prob(self, column, row):
    new_room = (0, 0)
    fringe = []
    # lowest_prob holds the temp value of the lowest probability a room has a pit/wum
    # at the end of the for loop, lowest_prob holds the value of the lowest probabilit
    lowest_prob = 1
    fringe = list(self.available_rooms)
    # known_BS = self.observation_breeze_stench(self.visited_rooms)
    # known_PW = self.observation_pits(self.visited_rooms)
    for each_room in fringe:
        if self.check_safety(each_room[0], each_room[1]) == True:
            new_room = each_room
            break
        else:
            prob_each_room = enumerate_joint_ask(each_room, {}, self.PitWumpus_probabi

            if prob_each_room.prob[True] < lowest_prob:
                lowest_prob = prob_each_room.prob[True]
                if lowest_prob <= self.max_pit_probability:
                    new_room = each_room
    return new_room
```

The_Wumpus_world.py



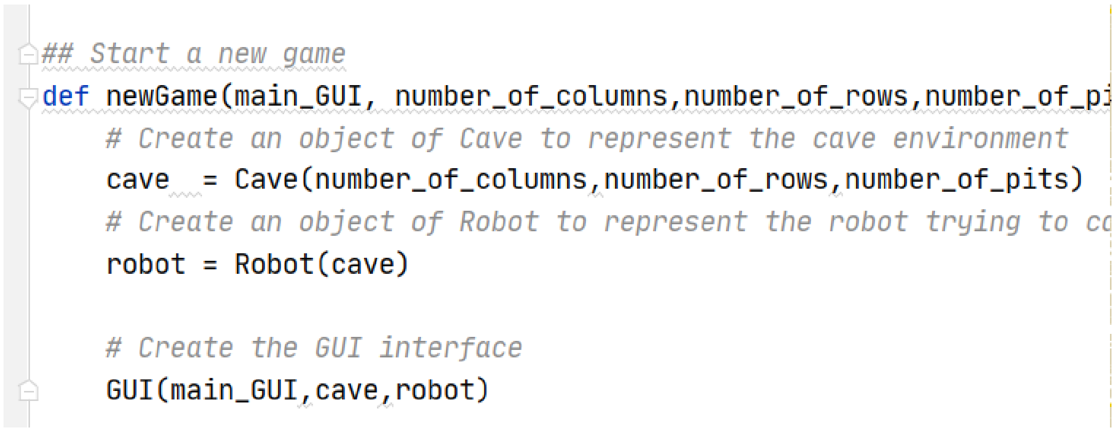
```python
#
#Global constrants
T, F = True, False
board_height = 600 # height of the board window
board_width = 600 # width of the board window

# # Global variables
# fixed_board = False # the board configuration is fixed or r
```

Remove variable fixed_board and the locations of the wumpus, pit, and gold chosen by the user for the fixed board

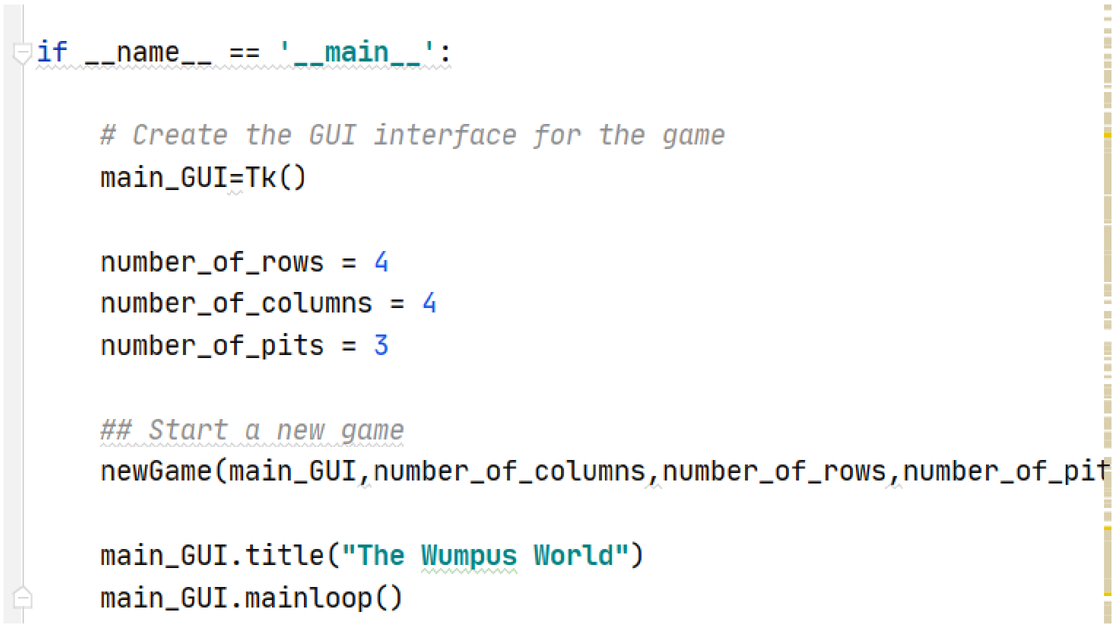


```python
# functions = {0: self.next_room, 1: self.next_room_prob}
functions = {0: self.next_room_prob, 1: self.next_room}
checking_function = functions.get(agent_type)
```

Modify the functions in step()



```
## Start a new game
def newGame(main_GUI, number_of_columns,number_of_rows,number_of_pi
    # Create an object of Cave to represent the cave environment
    cave  = Cave(number_of_columns,number_of_rows,number_of_pits)
    # Create an object of Robot to represent the robot trying to co
    robot = Robot(cave)

    # Create the GUI interface
    GUI(main_GUI,cave,robot)
```



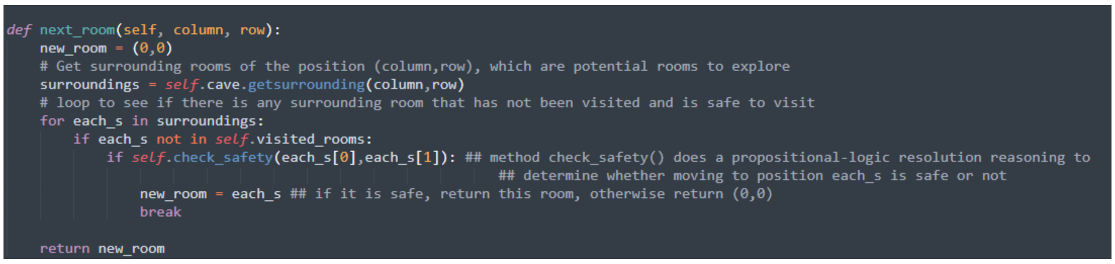
```
if __name__ == '__main__':

    # Create the GUI interface for the game
    main_GUI=Tk()

    number_of_rows = 4
    number_of_columns = 4
    number_of_pits = 3

    ## Start a new game
    newGame(main_GUI,number_of_columns,number_of_rows,number_of_pit

    main_GUI.title("The Wumpus World")
    main_GUI.mainloop()
```

Modify the method of calling the function in class GUI, main() and many others to simplify the call process and wrap the function

2.1 Propositional logic to find a safe grid

As shown in Figure 7.12, the resolution algorithm is used. The (KB) is first converted to CNF, and then the resolution rule is applied to the resulting sentence. A new clause is generated by resolving words containing complementary text. If the new clause has not yet appeared, it is added to the clause set until there is no new sentence that can be added or an empty clause is resolved.

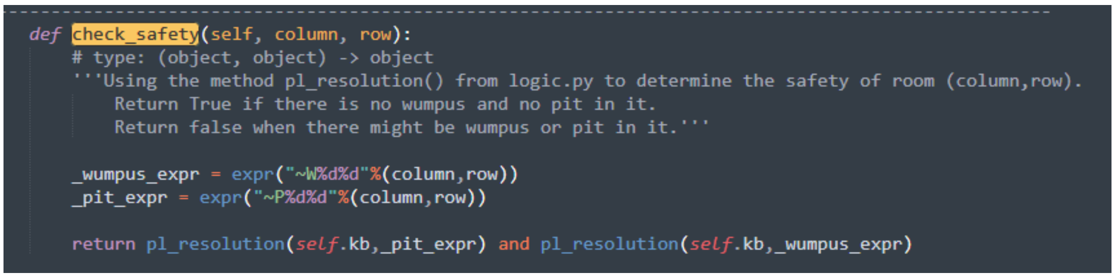
The corresponding code in the program is as follows:



```python
def next_room(self, column, row):
    new_room = (0,0)
    # Get surrounding rooms of the position (column,row), which are potential rooms to explore
    surroundings = self.cave.getsurrounding(column,row)
    # loop to see if there is any surrounding room that has not been visited and is safe to visit
    for each_s in surroundings:
        if each_s not in self.visited_rooms:
            if self.check_safety(each_s[0],each_s[1]): ## method check_safety() does a propositional-logic resolution reasoning to
                                                       ## determine whether moving to position each_s is safe or not
                new_room = each_s ## if it is safe, return this room, otherwise return (0,0)
                break

    return new_room
```

First search the surrounding grids, and pass the unvisited grids one by one to the check_safety function to check their safety. If it is safe, take the grid as the next step, otherwise return (0, 0) to indicate that there is no safe grid.

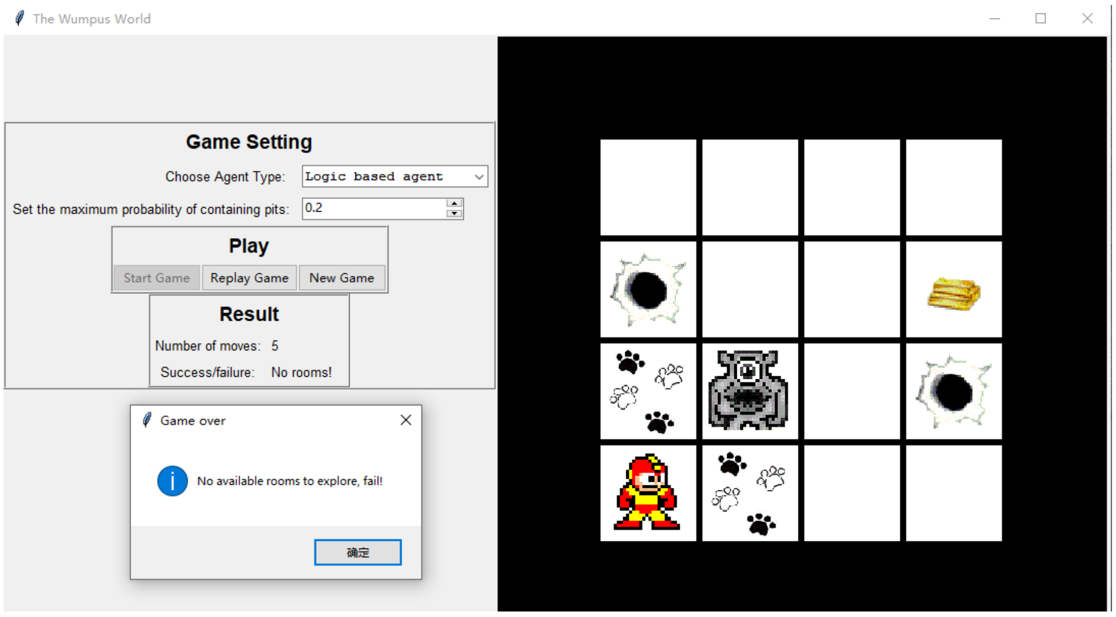
The check_safety function is as follows:



```python
def check_safety(self, column, row):
    # type: (object, object) -> object
    '''Using the method pl_resolution() from logic.py to determine the safety of room (column,row).
       Return True if there is no wumpus and no pit in it.
       Return false when there might be wumpus or pit in it.'''

    _wumpus_expr = expr("~W%d%d"%(column,row))
    _pit_expr = expr("~P%d%d"%(column,row))

    return pl_resolution(self.kb,_pit_expr) and pl_resolution(self.kb,_wumpus_expr)
```

This function can construct new sentences and solve sub-sentences.

The advantage of logical reasoning is to avoid death, but at the same time it is also easy to fall into a dilemma with nowhere to go.


The Wumpus World

Game Setting

Choose Agent Type: Logic based agent

Set the maximum probability of containing pits: 0.2

Play

Start Game | Replay Game | New Game

Result

Number of moves: 5

Success/failure: No rooms!

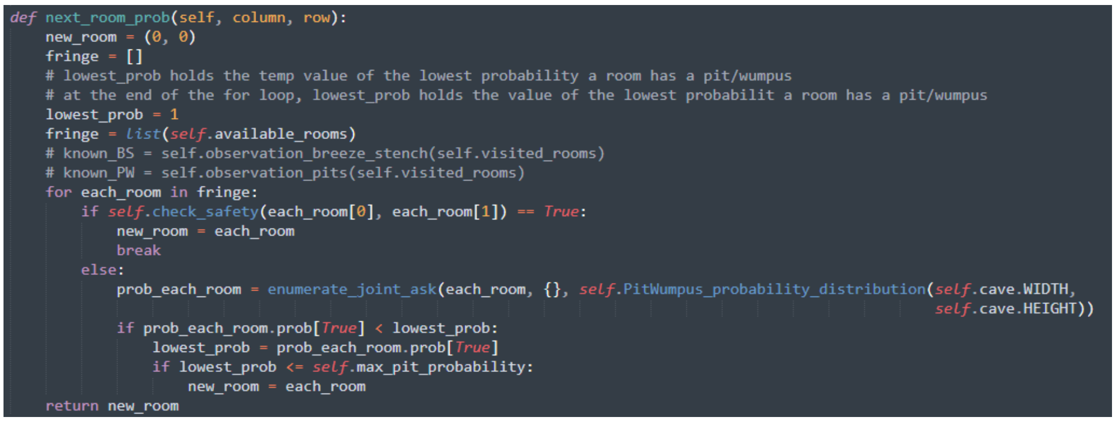
Game over

No available rooms to explore, fail!

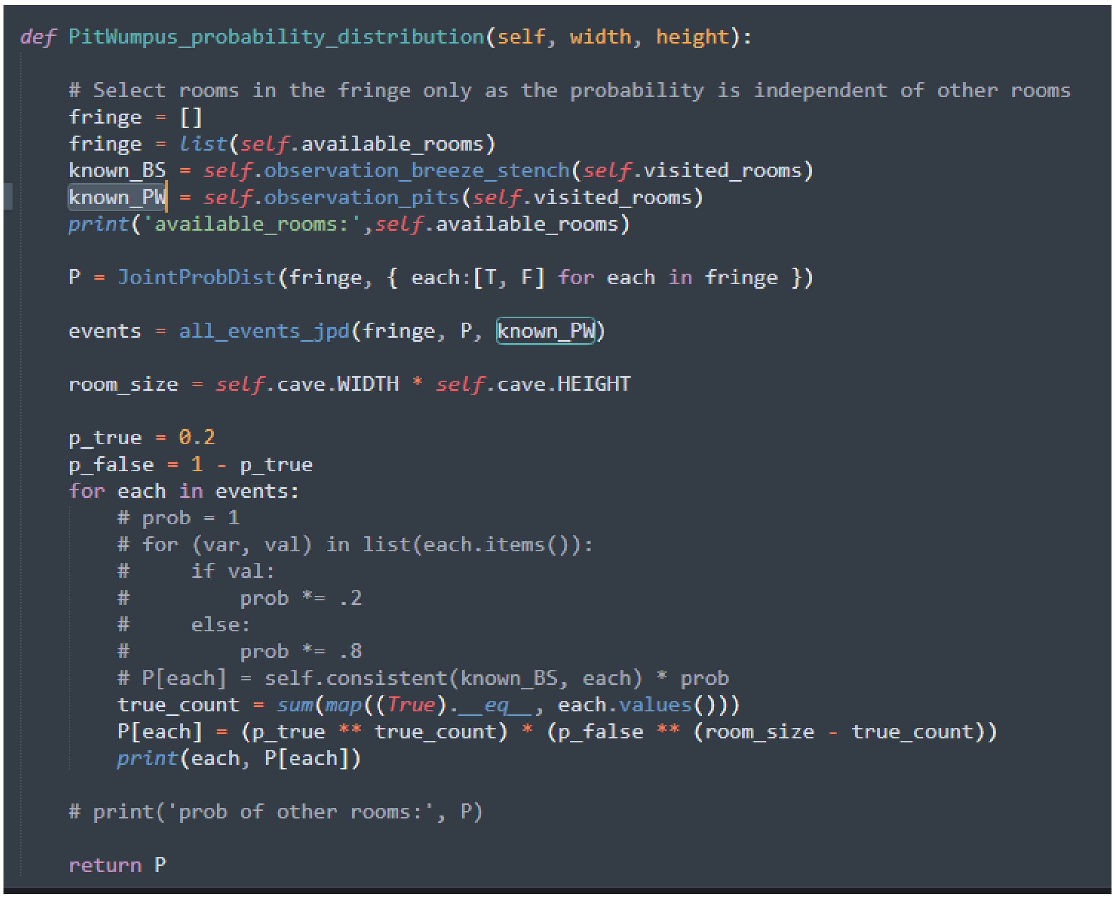
确定

2.2 Joint distribution probability to solve the surrounding grid

The method based on the joint distribution probability is used in the absence of a safety grid, as shown in the following figure:

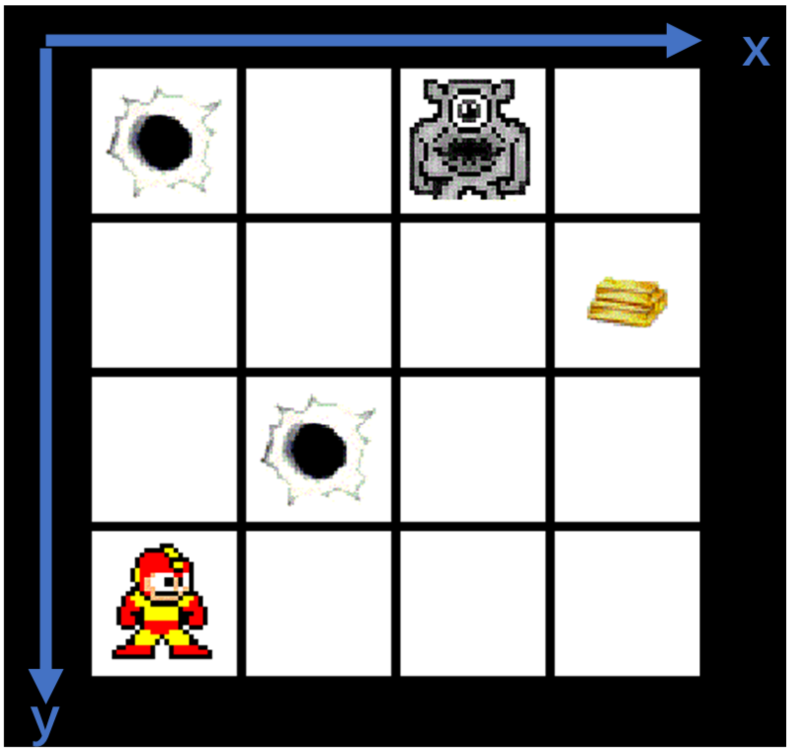

```
def next_room_prob(self, column, row):
    new_room = (0, 0)
    fringe = []
    # lowest_prob holds the temp value of the lowest probability a room has a pit/wumpus
    # at the end of the for loop, lowest_prob holds the value of the lowest probabilit a room has a pit/wumpus
    lowest_prob = 1
    fringe = list(self.available_rooms)
    # known_BS = self.observation_breeze_stench(self.visited_rooms)
    # known_PW = self.observation_pits(self.visited_rooms)
    for each_room in fringe:
        if self.check_safety(each_room[0], each_room[1]) == True:
            new_room = each_room
            break
        else:
            prob_each_room = enumerate_joint_ask(each_room, {}, self.PitWumpus_probability_distribution(self.cave.WIDTH,
                                                                                                          self.cave.HEIGHT))

            if prob_each_room.prob[True] < lowest_prob:
                lowest_prob = prob_each_room.prob[True]
                if lowest_prob <= self.max_pit_probability:
                    new_room = each_room
    return new_room
```

First calculate the joint distribution probability, as shown in the following figure:

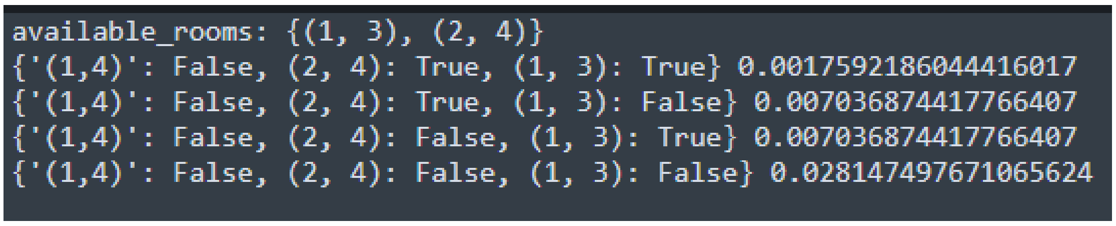


```python
def PitWumpus_probability_distribution(self, width, height):

    # Select rooms in the fringe only as the probability is independent of other rooms
    fringe = []
    fringe = list(self.available_rooms)
    known_BS = self.observation_breeze_stench(self.visited_rooms)
    known_PW = self.observation_pits(self.visited_rooms)
    print('available_rooms:',self.available_rooms)

    P = JointProbDist(fringe, { each:[T, F] for each in fringe })

    events = all_events_jpd(fringe, P, known_PW)

    room_size = self.cave.WIDTH * self.cave.HEIGHT

    p_true = 0.2
    p_false = 1 - p_true
    for each in events:
        # prob = 1
        # for (var, val) in list(each.items()):
        #     if val:
        #         prob *= .2
        #     else:
        #         prob *= .8
        # P[each] = self.consistent(known_BS, each) * prob
        true_count = sum(map((True).__eq__, each.values()))
        P[each] = (p_true ** true_count) * (p_false ** (room_size - true_count))
        print(each, P[each])

    # print('prob of other rooms:', P)

    return P
```

In this function, first obtain the known information such as the grids that have been visited, the grids of pit and wumpus, and the grids of breeze and stench. These are important conditions for calculating the risk probability.



The initial situation is calculated as follows, (1,4) is the initial position, and the default is false.



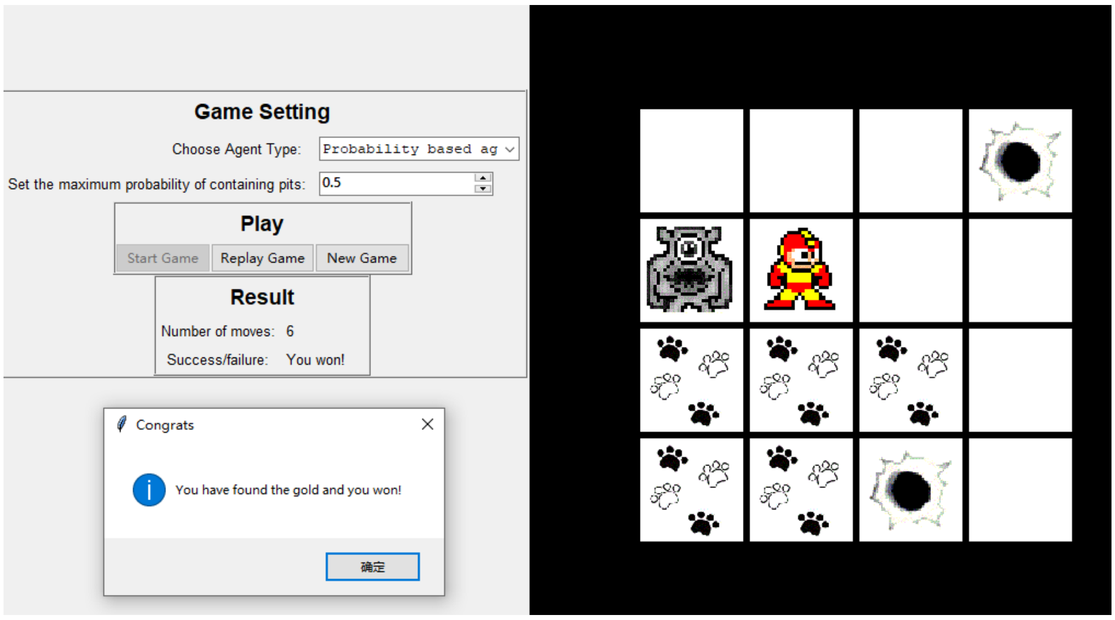
```
available_rooms: {(1, 3), (2, 4)}
{'(1,4)': False, (2, 4): True, (1, 3): True} 0.0017592186044416017
{'(1,4)': False, (2, 4): True, (1, 3): False} 0.007036874417766407
{'(1,4)': False, (2, 4): False, (1, 3): True} 0.007036874417766407
{'(1,4)': False, (2, 4): False, (1, 3): False} 0.028147497671065624
```

3 Experiment

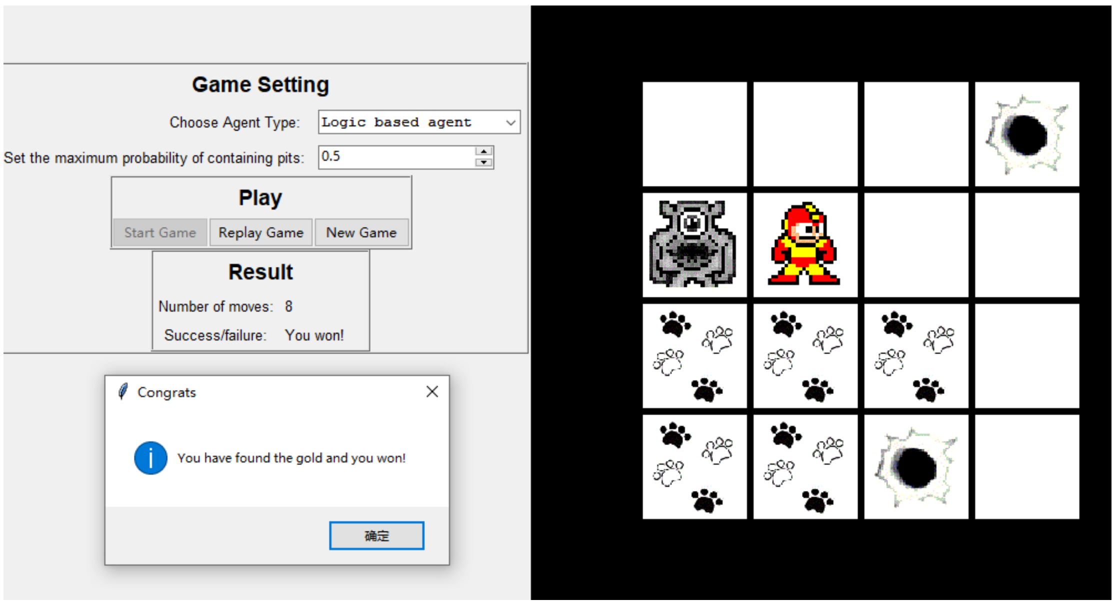
I modified the program so that the hybrid probabilistic agent and logic agent can be compared in the same world. The following is their comparison result:

3.1 Hybrid probabilistic agent is faster

The logical agent is slower than the hybrid probabilistic agent, because if there is no safe grid in front, it needs to go back and find it, and the probability can be directly calculated by the probability of the surrounding grid.


Game Setting
Choose Agent Type: Probability based ag
Set the maximum probability of containing pits: 0.5
Play

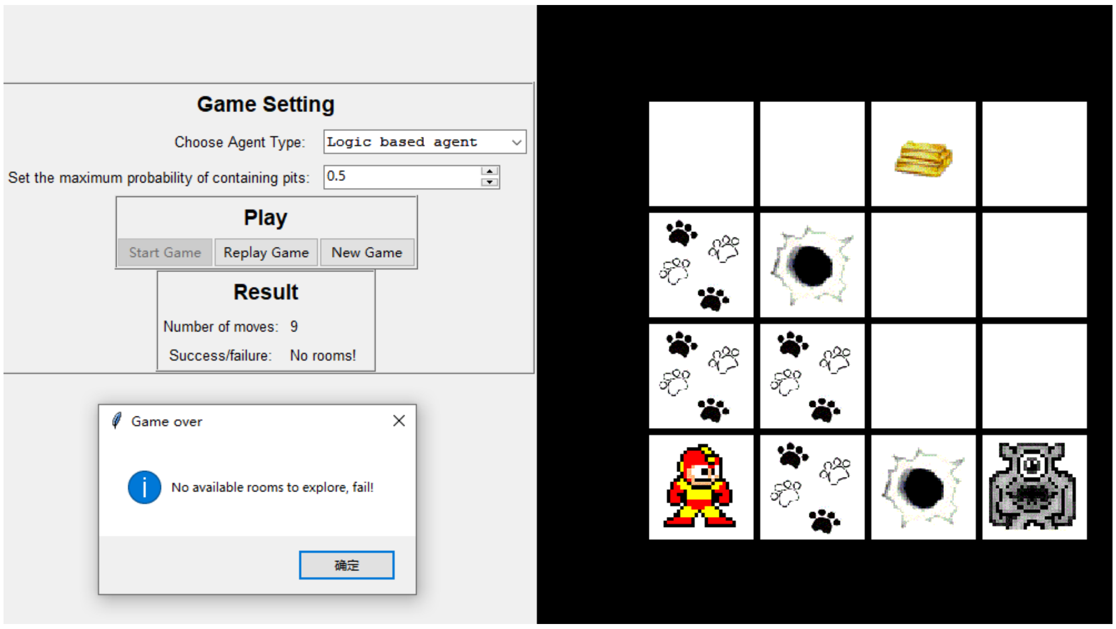
Result
Number of moves: 6
Success/failure: You won!
Congrats
You have found the gold and you won!

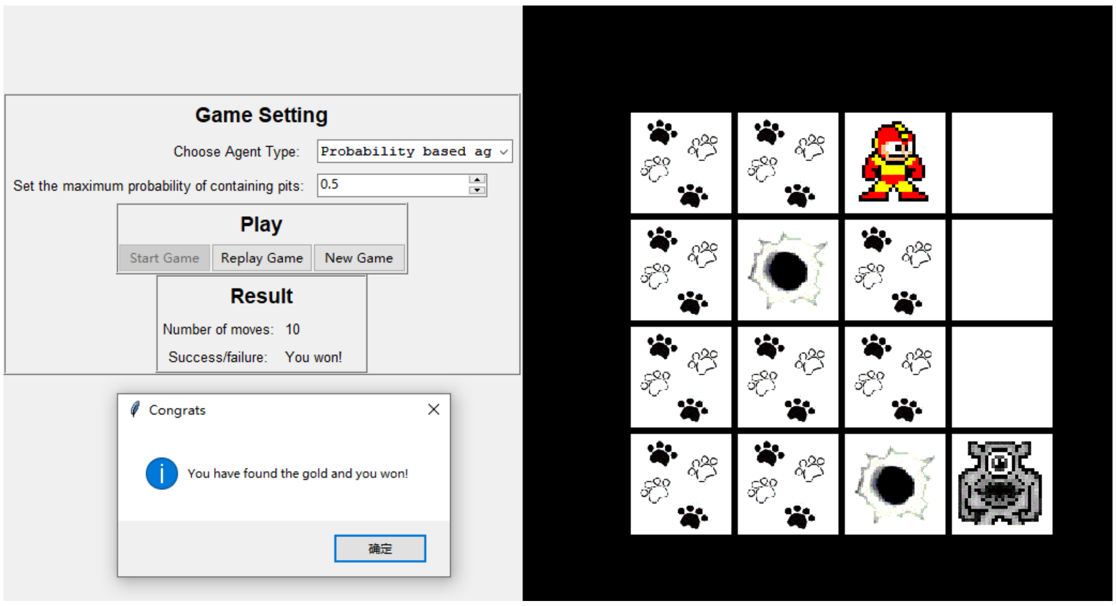

Game Setting
Choose Agent Type: Logic based agent
Set the maximum probability of containing pits: 0.5
Play

Result
Number of moves: 8
Success/failure: You won!
Congrats
You have found the gold and you won!

3.2 Hybrid probabilistic agent explores more possibilities

As mentioned in 2.1, the logic agent can only judge whether a grid is safe, so when there is no safe grid, it will not be able to move. The hybrid probabilistic agent is different. Even if there is no safety grid, it will choose a grid with reasonable probability to try. Although sometimes it will die, but there will also be times of success. With the goal of getting gold, hybrid probabilistic agent is better.


Game Setting
Choose Agent Type: Logic based agent
Set the maximum probability of containing pits: 0.5
Play
Result
Number of moves: 9
Success/failure: No rooms!
Game over
No available rooms to explore, fail!


Game Setting
Choose Agent Type: Probability based ag
Set the maximum probability of containing pits: 0.5
Play
Result
Number of moves: 10
Success/failure: You won!
Congrats
You have found the gold and you won!