# Wumpus World Game Simulation

*CISC6525 Artificial Intelligence*

*Final Project*

Guo Tian

## Introduction

The Wumpus world is an early computer game, which was originally created by Gregory Yob in 1972. The original version, called Hunt the Wumpus, is based on a simple hide and seek mode that requires players to discover the room with gold in a world with a dangerous monster and multiple obstacles. According to the hints gained while exploring the environment, players need to sense the location of gold and avoid the danger in the meanwhile. In this project, I explored the application of propositional logic in Artificial Intelligence by implementing truth table based entailment. This project is achieved by the following steps:

1. Task description and analysis
2. Model planing and building
3. Performance and evaluations
4. Further Study

## Task Description and Analysis

The Wumpus World game we are exploring is slight different than the traditional version. We discard the arrow-shooting function that players can kill the Wumpus once they affirm its location. The concrete game rules will be discussed below.

### Environment

- The complete world is composed with a 5x5 grid
- Players are set to start from coordinates (0,0) and facing to the east side
- Only one Wumpus and gold will be randomly located in one cell respectively
- Pits are generated randomly and can be more than one
- Game will end once agent enters the cell of Wumpus or pit

### Actuators

- Agent can move forward or backward to adjacent cells
- Agent cannot move to diagonal cells
- Once enter the cell of gold, agent can pick up gold and game ends

### Sensors

- The agent will sense stench in the cell directly adjacent to Wumpus
- The agent will sense breeze in the cell which is directly adjacent to pits
- The agent will sense if facing a wall and make a turn
- The agent will sense if he is located in a cell that gold exists

A example of Wumpus World is displayed in Figure 1:

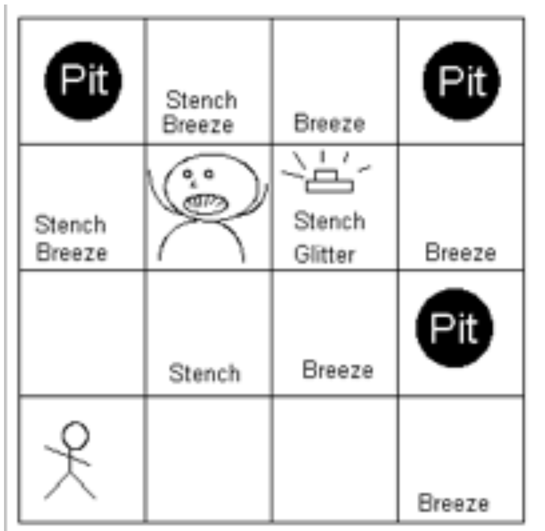


Figure 1

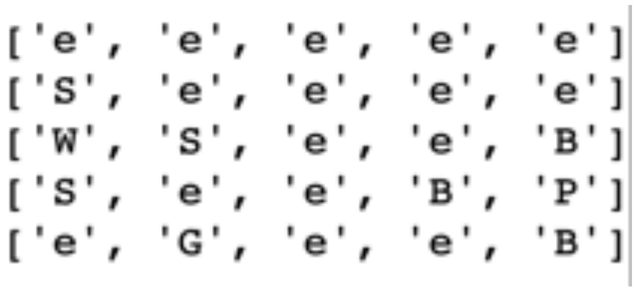
## Model Planning and Building

1. The very first step is to create a world that have randomly generates the location of Wumpus, gold, and pits. All the world generating job will be achieved in the function build_world. The world is a 5x5x5 matrix since it contains five different elements, Wumpus, gold, pits, breeze, and stench. Each element takes one layer of a two-dimensional matrix since some elements can be generated in the same cell. The matrix should possess the function of recording all of their coordinates at the same time.

One thing need be declared that Wumpus and pits can't be always located near the start point, which makes the game end easily. The cell that stores the gold also follows the rule. To make the random process more reasonable, function *random*_location completes the job to generate coordinates at a relatively high position in vertical axis. In the meanwhile, signal like breeze and stench should be simultaneously generated beside the cells of pit and Wumpus respectively. Function neighbors plays the role of distinguishing the adjacent cells of a specific location.

Except these environmental elements, more arguments should be declared before in the initial function. To ensure the game is continuing, agent should be alive and gold should not be picked yet. What's more, the initial position is (0,0) and the first action is moving forward facing to the right.

2. Since the game is not shown in graphic, we need find a way to display the Wumpus world and make it easy to trace the agent. The 5x5x5 world matrix will be converted into text type in a two-dimensional matrix. The function return_world is built to complete the task. ‘e’ indicates the cell is empty and the rest letters mean the cell contains Wumpus, Gold, Breeze, and Stench. A sample of a random Wumpus World is shown in Figure 2.



```
['e', 'e', 'e', 'e', 'e']
['S', 'e', 'e', 'e', 'e']
['W', 'S', 'e', 'e', 'B']
['S', 'e', 'e', 'B', 'P']
['e', 'G', 'e', 'e', 'B']
```

Figure 2

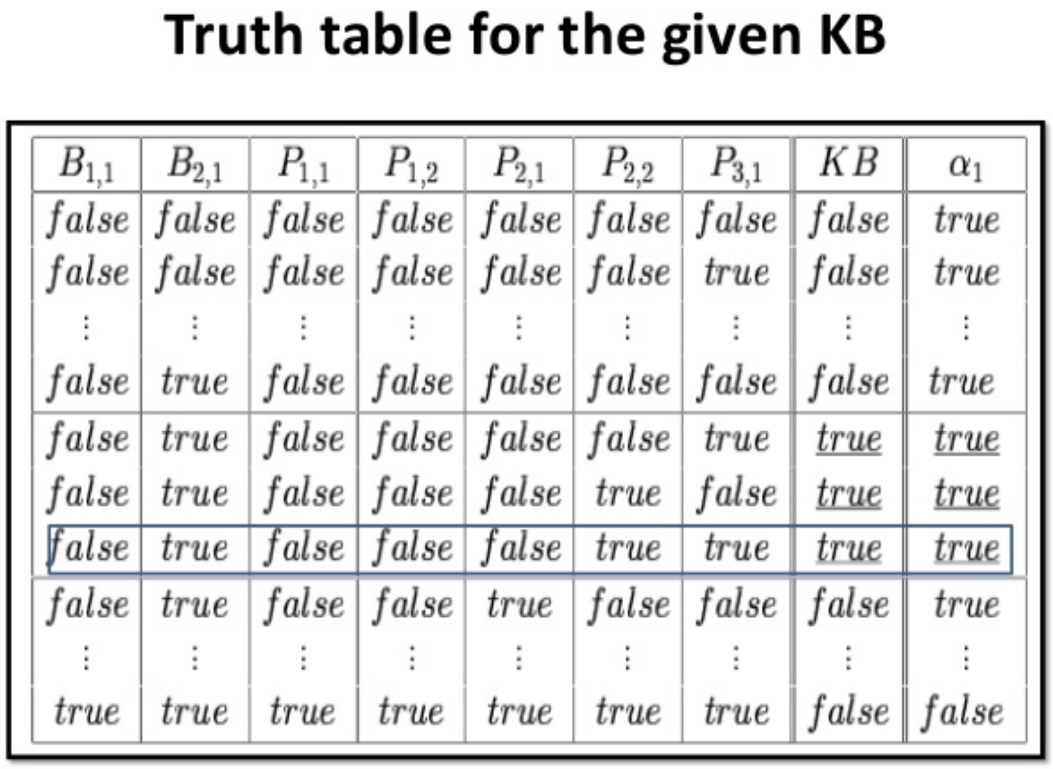
3. Once the agent enters the world, the most significant step is to perceive the environment. Function _do takes charge of the task to sense if the agent enters a cell with Wumpus or pit and informs the game ends if so. Besides, the game also ends if the agent senses a gold, which means gold-picked returns True, the agent wins.

Another important information should be perceived as well that danger signal, breeze or stench, will be recorded to knowledge base for future decision making. The signal is recorded in the format of ‘S01’, ‘B12’. This information goes into truth table and simultaneously update the condition of other cells’ arguments.

To check the neighbors are safe or not, return_safe function reads the coordinate of the agent’s location, and returns a safe list including its adjacent cells and the cell it has visited. In the meanwhile, a danger list will be generated once dangerous cell is determined depending on the result of truth table.

4.Once the agent grasps the basic information of the environment, it will make plan for the next step. Its top priority is not only avoiding dangerous place, but rather find out the shortest path to a safe cell where gold could be stored there. According to the agent’s knowledge of the world, function shortest_path can find the shortest path to the next safe house. At the same time, function path_plan will give specific instruction on following actions need be taken to achieve target.

5. The most significant part of making the proposition logical agent capable of planning a safe path is to construct the agent's knowledge base. The knowledge base should possess the function of implementing simple search routines. I completed this task with the help of a python API, AIMA, which refers to *Artificial Intelligence: A modern Approach* by Stuart Russell and Peter Norvig. Code file logic.py, from AIMA, implements first-order (FOL) logic language components and DPLL (Davis--Putnam-Logemann-Loveland), an improved backtracking algorithm. It targets to build a Wumpus World truth table for enumerating propositions for cells around the agent's current location. A Wumpus World truth table is shown in Figure 3.



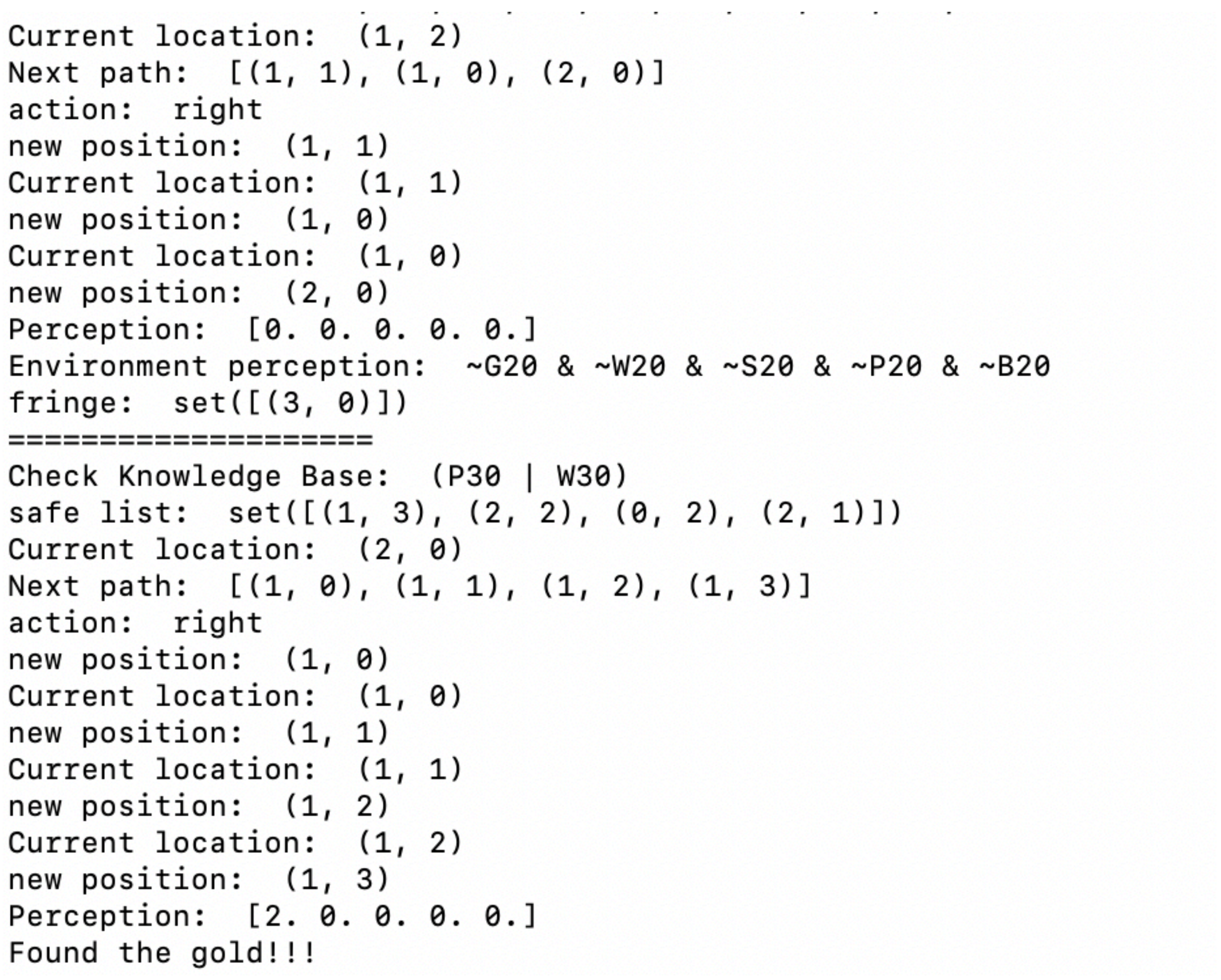
**Truth table for the given KB**

| $B_{1,1}$ | $B_{2,1}$ | $P_{1,1}$ | $P_{1,2}$ | $P_{2,1}$ | $P_{2,2}$ | $P_{3,1}$ | $KB$ | $\alpha_1$ |
|---|---|---|---|---|---|---|---|---|
| *false* | *false* | *false* | *false* | *false* | *false* | *false* | *false* | *true* |
| *false* | *false* | *false* | *false* | *false* | *false* | *true* | *false* | *true* |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| *false* | *true* | *false* | *false* | *false* | *false* | *false* | *false* | *true* |
| *false* | *true* | *false* | *false* | *false* | *false* | *true* | *true* | *true* |
| *false* | *true* | *false* | *false* | *false* | *true* | *false* | *true* | *true* |
| *false* | *true* | *false* | *false* | *false* | *true* | *true* | *true* | *true* |
| *false* | *true* | *false* | *false* | *true* | *false* | *false* | *false* | *true* |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| *true* | *true* | *true* | *true* | *true* | *true* | *true* | *false* | *false* |

Figure 3

## Performance and Evaluation

To evaluate the performance of this model, I plan to run the entire game 100 times automatically and record each round's final result, which indicates the cause of game over; the path that agent has passed through, and the steps taken in total. When running the game, each previous and current location will be displayed, as well as the environmental perception and action. This information can help me track if the game runs normally. A sample record of game is shown in Figure 4.

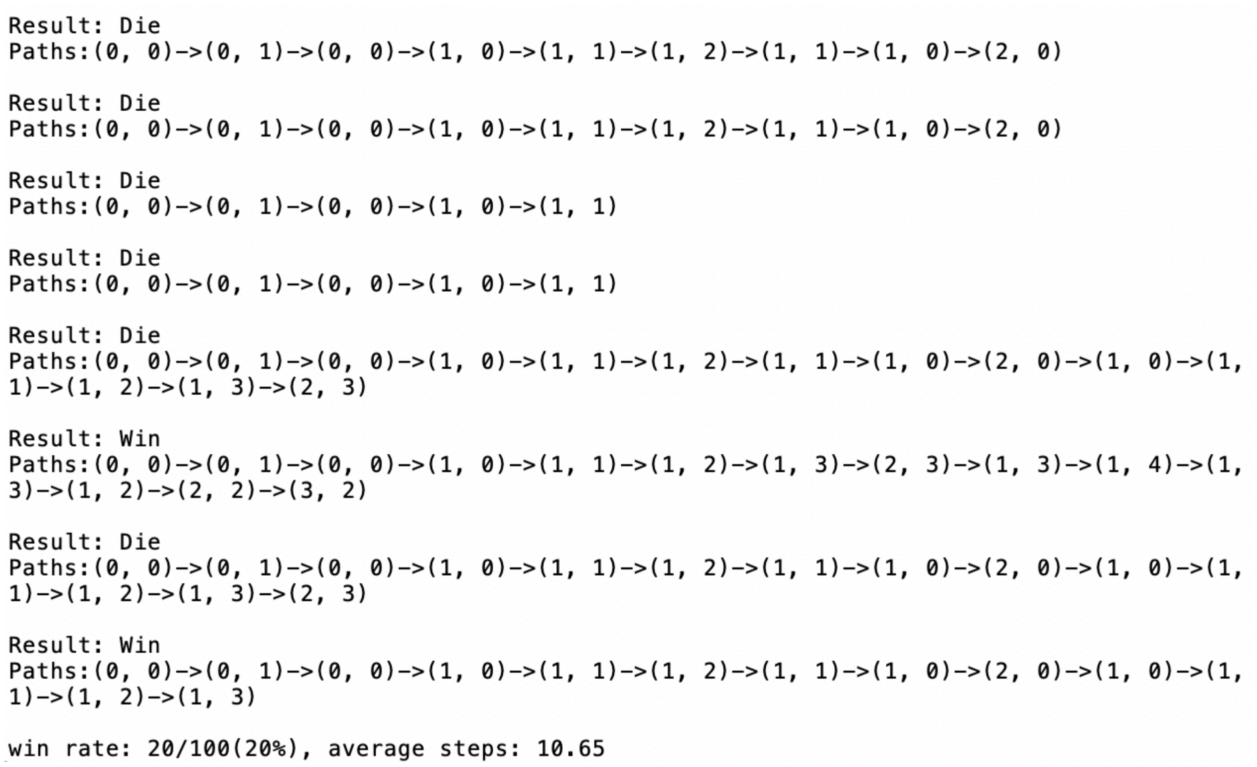


```
Current location:  (1, 2)
Next path:  [(1, 1), (1, 0), (2, 0)]
action:  right
new position:  (1, 1)
Current location:  (1, 1)
new position:  (1, 0)
Current location:  (1, 0)
new position:  (2, 0)
Perception:  [0. 0. 0. 0. 0.]
Environment perception:  ~G20 & ~W20 & ~S20 & ~P20 & ~B20
fringe:  set([(3, 0)])
====================
Check Knowledge Base:  (P30 | W30)
safe list:  set([(1, 3), (2, 2), (0, 2), (2, 1)])
Current location:  (2, 0)
Next path:  [(1, 0), (1, 1), (1, 2), (1, 3)]
action:  right
new position:  (1, 0)
Current location:  (1, 0)
new position:  (1, 1)
Current location:  (1, 1)
new position:  (1, 2)
Current location:  (1, 2)
new position:  (1, 3)
Perception:  [2. 0. 0. 0. 0.]
Found the gold!!!
```

Figure 4

As the figure shows, each step's perception and proposition logic write-in and read will help the agent plan the next path to a safe cell. However, sometimes the safe cell the truth table determines can be deceiving since the agent could not grasp complete information of a cell, which means the agent still have chance to enter a cell with Wumpus or pit.

In order to evaluate the performance of my model, I create a log file to record each round's result, either win or die, and the path the agent has passed through. After 100 runs, the log file will also record the success rate and the steps taken in average. A part of the log file is captured below:



```
Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 1)->(1, 0)->(2, 0)

Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 1)->(1, 0)->(2, 0)

Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)

Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)

Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 1)->(1, 0)->(2, 0)->(1, 0)->(1,
1)->(1, 2)->(1, 3)->(2, 3)

Result: Win
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 3)->(2, 3)->(1, 3)->(1, 4)->(1,
3)->(1, 2)->(2, 2)->(3, 2)

Result: Die
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 1)->(1, 0)->(2, 0)->(1, 0)->(1,
1)->(1, 2)->(1, 3)->(2, 3)

Result: Win
Paths:(0, 0)->(0, 1)->(0, 0)->(1, 0)->(1, 1)->(1, 2)->(1, 1)->(1, 0)->(2, 0)->(1, 0)->(1,
1)->(1, 2)->(1, 3)

win rate: 20/100(20%), average steps: 10.65
```

Since the agent has a large opportunity to die in cells that near the initial point, the building world function should still be improved to avoid entering dangerous locations at the beginning. Besides, the winning rate is only 15% which means agent still faces difficulties finding the gold and keeping away from pits and Wumpus, depending on knowledge base. The Wumpus world knowledge base function can also be modified since it takes relative long time to extract information from it that taking a single piece of information out costs about 2 seconds in average. It’s inefficient in time consumption because the space complexity is too high as the grid expands.

## Appendix

***Code file logic.py from folder AIMA was adapted from Artificial Intelligence: Modern Approach (Third Edition) by Stuart Russel and Peter Norvig.
***The code of project is shown below. Use command python2 main.py to run the whole code. The detail information will be recorded in file log.txt.

```
###CISC6525 Artificial Intelligence
###Final Project
###Author: Guo Tian
###Date: Dec.15 2018
from aima.logic import expr, dpll_satisfiable
import pdb
import random
import time
import copy
import numpy
import sys

#modify python's maximum recursive times
sys.setrecursionlimit(10000)

#define action, facing list and corresponding number
action_list = {
   'forward': 0,
   'left': 1,
   'right': 2,
   'pick': 3,
   }
action_prin = {
   0 : 'forward',
   1 : 'left',
   2 : 'right',
   3 : 'pick'
}

facing_list = {
   'up': 0,
   'right': 1,
   'down': 2,
   'left': 3
   }

map_list = ['G', 'W', 'S', 'P', 'B']
```

```
#function that defines the surrounding cells locations
def neighbors(i, j):
    li = set()
    if (i - 1) in range(5):
        li.add((i-1, j))
    if (i + 1) in range(5):
        li.add((i+1, j))
    if (j - 1) in range(5):
        li.add((i, j-1))
    if (j + 1) in range(5):
        li.add((i, j+1))
    return li
#function transfers world map from numeric to text
def return_world(world):
    world_c = numpy.chararray((5, 5))
    world_c[:] = 'e'
    for i in range(5):
        for j in range(5):
            if world[0][i][j]==2 and world[1][i][j]==0 and world[3][i][j]==0:
                world_c[i][j]='G'
            if world[1][i][j]==2 and world[0][i][j]==0 and world[3][i][j]==0:
                world_c[i][j]='W'
            if world[3][i][j]==2 and world[0][i][j]==0 and world[1][i][j]==0:
                world_c[i][j]='P'
            if world[0][i][j]==2 and world[1][i][j]==2 and world[3][i][j]==0:
                world_c[i][j]='G+W'
            if world[0][i][j]==2 and world[1][i][j]==0 and world[3][i][j]==2:
                world_c[i][j]='G+P'
            if world[1][i][j]==2 and world[0][i][j]==0 and world[3][i][j]==2:
                world_c[i][j]='W+P'
    return world_c

class Agent:

    #random locations of wumpus and gold
    def random_location(self):
        r = random.randint(2, 19)
        if r <= 3:
            i = 1
            j = r+1
        else:
           i = (r + 4) % 5
           j = (r + 1) % 5

        return (i, j)
```

```
def __init__(self):
    self.alive = False
    self.gold_picked = False
    self.all_pos = set()
    for i in range(5):
        for j in range(5):
            self.all_pos.add((i, j))

      #initial all the parameters and random the wumpus world
def build_world(self):
    self.alive = True
    self.gold_picked = False
    self.pos = (-1, 0)
    self.facing = facing_list['right']
    self.visited = set()
    self.danger = set()
    self.safe = set()
    self.fringe = set()
    self.plan = [action_list['forward']]
    self.world = numpy.zeros((5, 5, 5))
    self.known_world = numpy.ones((5, 5, 5))
    self.kb = WumpusKB()

      #random the location of Gold
    i, j = self.random_location()
    self.world[0][i][j] = 2
      #random the location of Wumpus
    i, j = self.random_location()
    self.world[1][i][j] = 2
      #locate the cells of stench
    for pos in neighbors(i, j):
        self.world[2][pos[0]][pos[1]] = 2

    for i in range(5):
        for j in range(5):
            if not ((i == 0) and (j == 0)):
        #random the location of pit
                if random.randint(1, 20) <= 2:
                    self.world[3][i][j] = 2
            #locate the cells of breeze
                    for pos in neighbors(i, j):
                        self.world[4][pos[0]][pos[1]] = 2
    self.paths = []
    self.end = False
    #return self.world
#function defines the agent
```

```
def wumpus_agent(self, percept):
    if len(self.safe) > 0:
        source = self.safe
    elif len(self.fringe) > 0:
        source = self.fringe
    else:
        source = self.danger
    goal = source.pop()
    self.plan = self.path_plan(self.shortest_path(goal))
    action = self.plan.pop()
    print "action: ", action_prin[action]
    return action

def main_func(self):

    if self.alive and (not self.gold_picked):
        if len(self.plan) > 0:
            action = self.plan.pop()
        else:
            i, j = (self.pos[0], self.pos[1])
            percept = self.world[i][j]
            action = self.wumpus_agent(percept)

        self._do(action)
#find the safe list based on knowledge base
def return_safe(self, i, j):
    new = neighbors(i, j) - self.visited
    new -= self.safe
    new -= self.danger
    new_safe = set()
    new_danger = set()
    if (self.world[2][i][j] == 0) and (self.world[4][i][j] == 0):
        self.fringe -= set((i, j))
        new_safe |= new
    else:
        self.fringe |= new
    print "fringe: ", self.fringe
    fringe = self.fringe.copy()
    if len(self.visited) > 1:
        for x in fringe:
            #print(x)
            if self.kb.ask(expr('(P%s%s | W%s%s)' % (x*2))):
                new_danger.add(x)
                self.fringe.remove(x)
            elif self.kb.ask(expr('(~P%s%s & ~W%s%s)' % (x*2))):
                new_safe.add(x)
                self.fringe.remove(x)
    if len(new_safe) > 0:
        self.safe |= (new_safe)
    if len(new_danger) > 0:
        self.danger |= new_danger

    print "safe list: ", self.safe
#function that executes moves for agents and records new positions
def _do(self, action):
    facing_dict = {
            facing_list['up']: (0, 1),
            facing_list['right']: (1, 0),
            facing_list['down']: (0, -1),
            facing_list['left']: (-1, 0),
            }
    if action is action_list['forward']:
        dx, dy = facing_dict[self.facing]
        new_pos = (self.pos[0]+dx, self.pos[1]+dy)
        print "new position: ", new_pos
        self.paths.append(new_pos)
        if new_pos in neighbors(*self.pos):

            if new_pos not in self.visited:
                i, j = new_pos
                percept = self.world[i][j]
                print "Perception: ",percept

                self.visited.add((i, j))
                if (i, j) in self.safe:
                    self.safe.remove((i, j))
                self.known_world[i][j] = percept

                if percept[0] == 2:
                    self.gold_picked = True
                    self._do(action_list['pick'])
                    print "Found the gold!!!"
                    self.end = True
                elif (percept[1] == 2) or (percept[3] == 2):
                    self.alive = False
                    print "Unfortunately the agent died"
                    self.end=True
                else:
                    c = 0
                    know = []
                    #save perception of environment into knowledge base
                    for x in percept:
                        if x == 2:
                            know.append("%s%s%s" % (map_list[c], i, j))
                        elif x == 0:
                            know.append("~%s%s%s" % (map_list[c], i, j))
                        c += 1
                    if len(know) > 0:
                        print "Environment perception: ", ' & '.join(know)
                        self.kb.tell(expr(' & '.join(know)))
                    self.return_safe(i, j)
            self.pos = new_pos
            print "Current location: ", self.pos
    elif action is action_list['left']:
        self.facing = (self.facing - 1) % 4
    elif action is action_list['right']:
        self.facing = (self.facing + 1) % 4
    elif action is action_list['pick']:
        i, j = (self.pos[0], self.pos[1])
        self.world[0][i][j] = 0
        self.gold_picked = True
#find the shortest path to the goal destination
def shortest_path(self, goal):
    g = self.visited
    v = {}
    for x in g:
        v[x] = 625
    v[goal] = 0
    def v_neighbors(i, j, s):
        li = []
        for t in neighbors(i, j):
             if (t in self.visited) and (t not in s):
                li.append(t)
        return li
    def get_path(v):
        path = []
        a = self.pos
        t = v[a]
        while t > 1:
            for x in v_neighbors(a[0], a[1], s):
                if v[x] < t:
                    t = t - 1
                    path.append(x)
                    a = x
                    break
        return path
    s = [goal]
    while (len(s) > 0):
        u = s.pop()
        for x in v_neighbors(u[0], u[1], s):
            if v[x] > v[u] + 1:
                v[x] = v[u] + 1
                s.append(x)

    pa = get_path(v)
    pa.append(goal)
    print "Next path: ", pa
    return pa
#function that returns the actions need be taken to the target path
def path_plan(self, path):
    plan = []
    prev = self.pos
    prev_facing = self.facing
    for pos in path:
        dx = pos[0] - prev[0]
        dy = pos[1] - prev[1]
        dy_dict = {
            -1: 'down',
            1: 'up',
            0: 'up' #dummy item
            }
        dx_dict = {
            -1: 'left',
            1: 'right',
            0: dy_dict[dy]
            }
        facing = facing_list[dx_dict[dx]]
        d = facing - prev_facing
        if d != 0:
            c = abs(d)
            if c > 2:
                d = -d
                c = c % 2
            if d < 0:
                direction = action_list['left']
            elif d > 0:
                direction = action_list['right']
            plan.extend([direction] * c)
            prev_facing = facing
        plan.append(action_list['forward'])
        prev = pos
    plan.reverse()
    #print "plan: ", plan
    return plan
```

```
class WumpusKB():
    #A KB for wumpus world

    def __init__(self, sentence=None):
        self.k = expr('~P00 & ~W00')

        li = []
        for i in range(5):
            for j in range(5):
                for l, r in (('B', 'P'), ('S', 'W')):
                    left = "%s%s%s" % (l, i, j)
                    right_list = []
                    for s, t in neighbors(i, j):
                        right_list.append("%s%s%s" % (r, s, t))
                    li.append("(%s <=> (%s))" % \
                             (left, ' | '.join(right_list)))
        e = expr(' & '.join(li))
        self.tell(e)

        # one and only one wumpus
        li = ['W%s%s' % (i, j) for i in range(5) for j in range(5)]
        e = expr(' | '.join(li))
        self.tell(e)

        li = ['(~W%s%s | ~W%s%s)' % \
              (i, j, x, y)
              for i in range(5) \
              for j in range(5) \
              for x in range(5) \
              for y in range(5) \
              if not ((i == x) and (j == y))]
        e = expr(' & '.join(li))

        self.tell(e)
    #function that save perception to knowledge base
    def tell(self, s):
        self.k &= s
    #function that extracts information from truth table
    def ask(self, s):
        print "=================="
        print "Check Knowledge Base: ", s
        start = time.time()
        r = dpll_satisfiable(self.k & ~s)
        #print "cost: ", time.time() - start
        if r is False:
            return True
        else:
            return False
#main function that runs the game and saves records to a log file
if __name__ == '__main__':
        b = Agent()
        paths = []
        pick = []
        steps = []
        #modify the number in the parentheses to run games multiple times
        for i in range(2):
                maps = []
                b.build_world()
                while not b.end:
                        b.main_func()
                pick.append(b.gold_picked)
                maps.append(return_world(b.world))
                paths.append(b.paths)
                steps.append(len(b.paths))
                print(maps)
        with open("logs.txt", 'w+') as f:
                for i in range(len(paths)):
                        f.write("Result: {}\n".format('Win' if pick[i] else 'Die'))
                        f.write("Paths:" + "->".join([str(j) for j in paths[i]]) + '\n\n')
                        #f.write()
                f.write("win rate: {}/{}({:.0f}%), average steps: {:.2f}".format(sum(pick),
len(pick), sum(pick)*100.0/len(pick), numpy.mean(steps)))
```