Wumpus World

Overview

- Course CSC14003 @ 18CLC6 "Introduction to Artificial Intelligence"
- Members:
  - `18127185` Bùi Vũ Hiếu Phụng ([@alecmatts]())
  - `18127080` Kiều Vũ Minh Đức ([@kvmduc]())
- Assignment Plan
  - Phụng: build Map class, read Map, create Map, create UI, process signal and action, create a simple version of game, fix bugs
  - Đức: build Agent class, solve Map, build KB class, build Search class, Convert into Action.
- Requirements evaluation:

No.	Specifications	Completion (0.0 - 1.0)
1	Solve problem	1.0
2	Graphics	1.0
3	Map generator	1.0
4	Report	1.0

User manual

`python main.py <mode> <arguments>`

- There are 2 modes in this program
  - `random`: Run agent on a random map 10x10 that is generated with custom number of pit, wumpus, gold. Arguments are:
    - Agent position at row
    - Agent position at column
    - Number of pits
    - Number of wumpus
    - Number of Gold
      Example: `python main.py random 0 0 7 7 7`
  - `file`: Run agent on a map that is represented in a file. File must be placed in `maps/` folder with format is defined in the project description. Argument is:
    - File name
      Example: `python main.py file original.txt` which original.txt is the Figure 1 in project description
- When the game starts, you can easily see that there are 2 buttons. If you want to see how the agent acts step by step, press the STEP button. If you want the agent solve the whole map, press RUN ALL
  - You can pause at any step in RUN ALL mode by pressing STEP

Structure

Environment and programming language

- **Language:** Python3
- **Environment:** virtualenv or conda
- **Dependencies:** PyGame for map visualization
- **IDE:** VSCode, PyCharm

Actions and gamestates

- We build the agent solve map according to the environment, what the agent perceive and return an action to the environment. What agent perceive will be stored at an virtual environment at that step, we call it gamestate.
- Gamestate S_i will store all the environment around the agent. At every agent's step A_i, gamestate changes. The process will be performed similar to:

$$S_0 \rightarrow (calc) \rightarrow A_0 \rightarrow S_1 \rightarrow (calc) \rightarrow A_1 \rightarrow \ldots \text{END GAME}$$

- Initially, we build a simple version of Wumpus Simulation which will receive command from keyboard, if we hit a button, game will transform the signal to an action, finally that action will send to game. Lately, we create an agent do the same thing to human, return an signal of action to the game board.
- Action will be define with an Enum class with an unique number will represent an action. Our action will perform more human-being than some other projects, guide we had researched from GitHub, StackOverflow,... For example, player is heading to Right, if agent return Right one more time, player will go to its right node, if agent return other move than Right, player will turn to that direction (like playing video game), so forth...
- There is also a GameState enum class to define `RUNNING` or `NOT RUNNING` game.

Map

We build our Wumpus World pretty similar to tile-based game. Therefore, there are 2 main components

- Tile: which is exactly what is sound like.
    - A tile defines if there is any pit, wumpus, breeze, stench or player (or agent)
    - There are also getter and setter to change tiles status
- World:
    - The world contains a matrix of tiles as well as number of golds and wumpuses to detect if the agent has cleared the map or not
    - Because of this structure, we can directly access to a tile by its position `(row, column)` just like how we do with other matrices
- When the agent make any move, the world will automically change itself: remove a wumpus, remove a gold, move the agent,...

Agent

Structure

As we mention above, our structure is following gamestate and action. Now we will discuss further about the structure of agent and gamestate

- **Gamestate** have a mission which is storing all the visited node and the unvisited node that is safe, it also store previous state, we use a dictionary structure to store it so we can index more convenient.
- Next we will have structure **Node**, we decide to store node name in this structure. Also, we implement this structure so it can point to it adjacent node directly follow a direction. For example, if agent is staying at Node(2,3), we get Node(2,3).right, we can get agent's right node, in this case is Node(2,4)
- As the essential element, we have **Knowledge Base (KB)**. It's the backbone of the algorithm which we decide to use **Resolution**. KB will store some logical sentences, which will describe the explored environment.
  - For the reason that we use Resolution Algorithm, so we have to operate with sentences of logic with format of CNF. We use string operate completely. For example, if Node(2,2) don't have signal of Breeze, so we can conclude that there are no pits at all four directions of the current node. Therefore, we append to KB $\neg P1,2 \land \neg P3,2 \land \neg P3,1 \land \neg P2,1$ as CNF form, so forth.
  - The KB class must have function which is can check a **query** by PL-Resolution.
- The **Agent** will have the ability to know which direction it is facing and its current state (automatically updated by itself)
- The last thing we have is **Level solver** which will cover all of structures that we mentioned above. See the full structure below:

```
LevelSolver
|   Knowledge base
|
|
└───Gamestate
|   |   Unvisited_safe []
|   |   Visited []
|   |   Dictionary of previous state (Use Node class)
|   └───
|
└───Agent
    |   CurrentNode
    |   Direction
    └───
```

Behavior

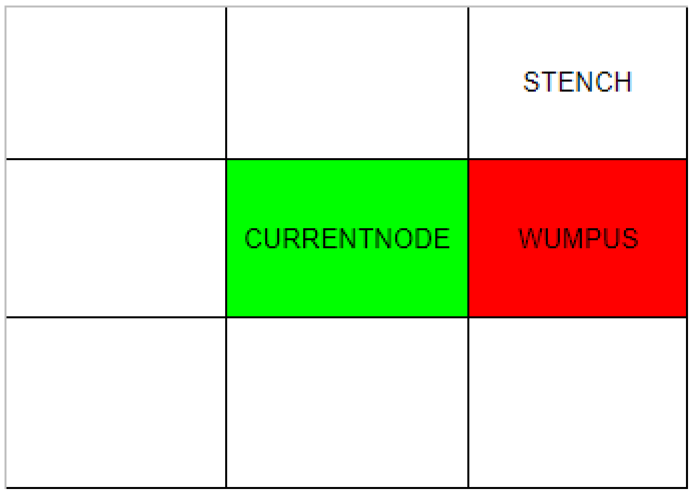
- Behavior of the agent will be processed mostly based on the knowledge base
  - At each step, the agent will sense new object/warnings such as Breeze or Stench. As we describe above, we continously add new clauses to the solver's KB and it will extract new action based on this "big bag"
  - Using Resolution here, the agent knows where is safe and where is danger
- There is a safe node list, the agent is also available to search to the next safe node reasonably because it only move to the explored node which we can be sure there is no danger there
- There are also some special moves to prevent agent losing the game and maximizing the score:
  - First, if agent can feel Breeze or Stench at the starting node, agent will climb immediately, agent won't take any risk.
  - Second, the agent will not shoot arrow randomly if it is unsure

- Otherwise, all the action that is returned must have its condition following from the True/False value of the query's check function.
- We have to remember that $KB \models \alpha$ when we can show that $KB \wedge \neg\alpha$ is satisfiable.
- So whenever want to a query from KB, we have to take the negative of itself. For example, if the agent want cho check whether Node(2,2) have Pit, this lead to CNF format is $P2,2$. If we check query $\neg P2,2$, and the function to check the query return True, that means there is $P2,2$. Our function work at the same with Breeze, Stench, Wumpus, ...etc.
- A $Node(x, y)$ are safe if it don't have $W(x, y) \wedge P(x, y)$. With KB and resolution function, we can easily check that work. After make sure it's safe, gamestate will store that node to it's memory to explore later, if the list of nodes needed to explore at State x is empty, so agent can climb out of map with no problem. That the first condition to explore map and endgame we try to focus.
- If agent sense breeze or stench, it will try to segment the source of danger. Then add into a sentence, finally add that sentence into KB. If at that node, agent don't feel breeze or stench, it will add directly into KB that all the adjacent is clear. For example, $Node(2, 2)$ is Breeze, and not Stench. So after agent get to $Node(2, 2)$, the following information will add into KB

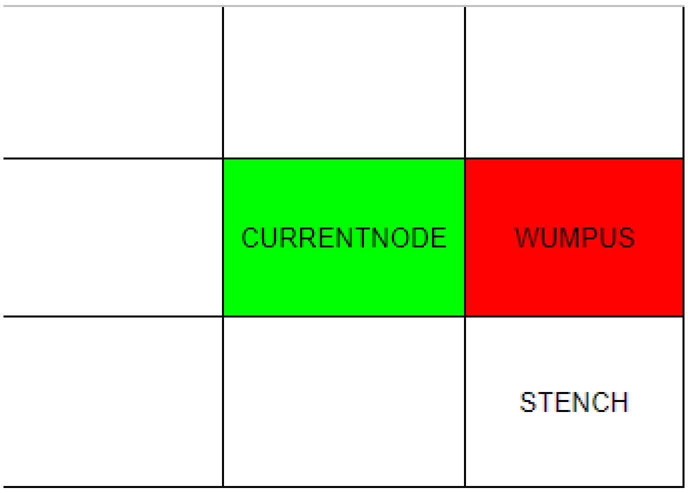
$$\begin{gathered} ['P1,2','P2,1','P3,2','P2,3'] \\ ['\neg W1,2'] \\ ['\neg W2,1'] \\ ['\neg W3,2'] \\ ['\neg W2,3'] \end{gathered}$$

- If many item in one list, that means all item will connect by **OR**($\vee$) operator in CNF, all list is connect by **AND**($\wedge$) operator in CNF. All the KB and KB solver, we try to implement by hand, so it may lead to problem that we'll discuss later.
- If agent can segment directly the $Node(x, y)$ contain a Pit, agent will record $Node(x, y)$ is danger, and it won't recognize it's any adjacent node. If agent can segment directly the $Node(x, y)$ contain a Wumpus, it will shoot that node. The condition to shoot we'll try to visualize. This is the condition to agent shoot to it's Right node:

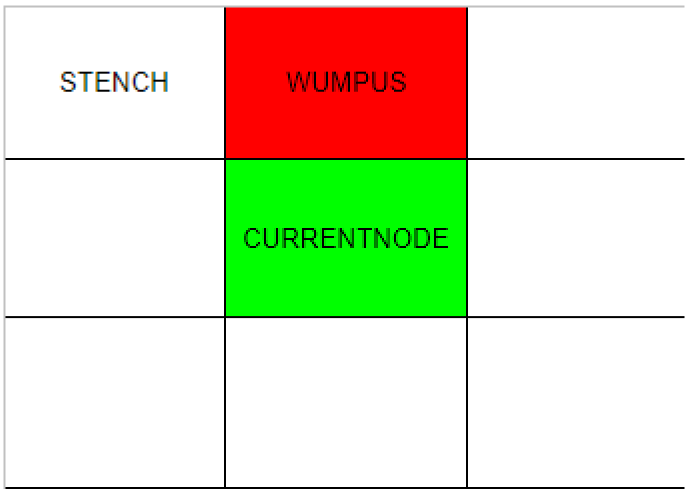
  Case 1:



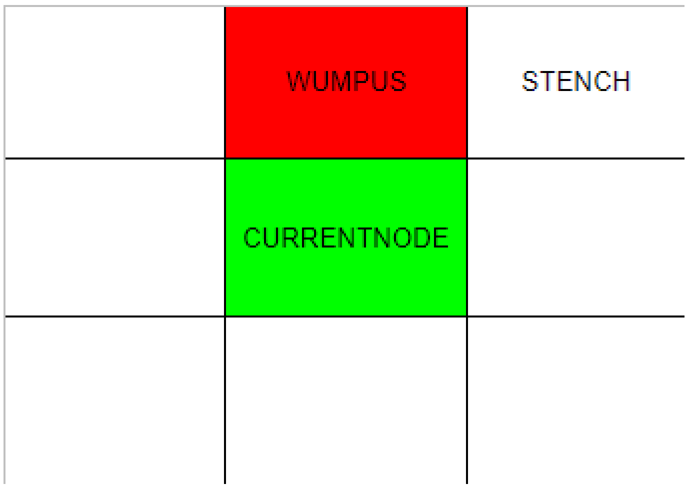
Case 2:



- If agent is at $Node(2,2)$, and $Node(2,2)$ can sense stench. Furthermore, in case 1, if agent can check Stench in $Node(1,3)$ is True and there are no Wumpus in $Node(1,2)$, then wumpus must be at $Node(2,3)$. Same as case 2, if agent can check Stench in $Node(3,3)$ is True and there are no Wumpus in $Node(3,2)$, then wumpus must be at $Node(2,3)$. Agent will perform shoot action into it's right node.
  - Same as Up node:
    Case 1:



Case 2:



  - Left node :
    Case 1:

STENCH		
WUMPUS	CURRENTNODE	

Case 2:

WUMPUS	CURRENTNODE	
STENCH		

- Finally, Down node:

  Case 1:

	CURRENTNODE	
STENCH	WUMPUS	

Case 2:

	CURRENTNODE	
	WUMPUS	STENCH

    - All we have to do is after segment the the area, we have to classify the direction we want to shoot following above case, we have to check 2 case in each direction.
    - The important thing is how to transform path into a list of move so UI can take that signal to process. Thanks to the structure that have store the Node, the direction. Each step, we pop one adjacent of current node, which is an element of the path to goal and convert it. Finally is move agent to update new state and return that action to UI. We use **UCS** find the path from current state lead to goal is the node of **unvisited**

Obstacle

- When we develop this agent, we realize is that using PL-Resolution can take $O(n^2)$ time which n is the number of clauses in the KB, to check a query (Cause it need to check all pair of KB). As the agent acts and perceives, the KB grows significantly. Since our KB class is implement by hand and solve by string compare, it leads to a problem is that the first moves are pretty fast but later the agent will extract the action very slow (2-3s).
- At first time, we try applied local search into this problem, which is, if agent kill a wumpus successfully, KB will clear all sentence related to Wumpus to resize the KB smaller, and the agent just has to concern if agent shoot don't kill the target successfully, it won't clear KB. Otherwise, all the wumpus which agent encounter will be treated as the first wumpus of the game that agent will encounter.
- This solution solve the map faster, it don't have high complexity of algorithm since it's a local search. But on the other hand, we got a local solution. In some map that we tested, agent can't get out early than we expect. For example, if agent kill successfully Wumpus at $Node(2,8)$, it will clear the KB with key Wumpus at that state, it explore some more and get a sentence is $[W1,7,W2,8]$. Now agent get confused because it don't remember that it already kill Wumpus at $Node(2,8)$, so it climb out earlier instead of kill Wumpus at $Node(1,7)$ and continue the game. This solution don't completeness
- So we decide to be greedy, it may take longer, but this algorithm is completeness. If agent kill the Wumpus successfully, it just clear the KB at that Node, it left a bunch of list in KB that never use. KB append more and more in each step (average of a game is 300 sentence in KB). It may take a while to Resolution, and to check a query.

GUI

We use Tkinter to make the GUI

- Tiles, warnings, terrains, agents are drawn by `tkinter.canvas` and stored in the list for easily manipulating
- Actions are processed similar to the demo game in project seminar
  Example: 2 RIGHT = MOVE TO THE RIGHT
- Actions will affect both the graphic and the world
- You can play the game by yourself by initializing agent as ManualAgent and uncommenting the buttom bind as well as using the corrent `mainloop()`

References

- Thanks a lot to the authors of some repositories on Github. It help us a lot in kept us in track, and gave us some idea to modify the project
- Github 1
- Github 2
- Github 3