**UNIVERSITY OF SCIENCE, VNU-HCM**

**FACULTY OF INFORMATION TECHNOLOGY**



**REPORT PROJECT 2:**

**WUMPUS WORLD AGENT**

**Course:** Introduction to Artificial Intelligence

**Instructor:** Ms Nguyen Ngoc Thao, Mr Tran Nguyen Duy Minh

**Team Members:**

Thieu Quang Vinh - 23127143

Tran Hai Duc - 23127173

Pham Thanh Loc - 23127405

Table of Contents

[**1.** **Introduction to Wumpus World** 3](#_Toc206324431)

[**a.** **Problem instruction** 3](#_Toc206324432)

[**b.** **Objectives** 3](#_Toc206324433)

[**2.** **Game Visualization and Design system** 4](#_Toc206324434)

[**a.** **Game visualization** 4](#_Toc206324435)

[**b.** **System architecture and module design** 5](#_Toc206324436)

[**i.** **Design space** 5](#_Toc206324437)

[**ii.** **Screen space** 5](#_Toc206324438)

[**iii.** **Main space** 7](#_Toc206324439)

[**iv.** **Development space** 7](#_Toc206324440)

[**3.** **Knowledge base (KB) fomulation** 7](#_Toc206324441)

[**4.** **Inference engine approach** 7](#_Toc206324442)

[**a.** **Rule-based inference** 7](#_Toc206324443)

[**b.** **Probabilistic reasoning** 7](#_Toc206324444)

[**5.** **Planning algorithm desription** 7](#_Toc206324445)

[**a.** **Algorithm description** 7](#_Toc206324446)

[**b.** **Peseudo code / flowchart** 7](#_Toc206324447)

[**6.** **Experiment result** 7](#_Toc206324448)

[**a.** **Test scenarios** 7](#_Toc206324449)

[**b.** **Compare time performance** 7](#_Toc206324450)

[**c.** **Compare success rate** 7](#_Toc206324451)

[**d.** **Discussion** 7](#_Toc206324452)

[**7.** **Team member contributions** 7](#_Toc206324453)

[**8.** **Reference and citations** 7](#_Toc206324454)

1. **Introduction to Wumpus World**
   1. **Problem instruction**

Imagine being placed inside a dark cave with the goal of **finding hidden treasure** and **escaping safely**. You have **no map** of the cave and **no clear knowledge of the dangers** that lie within. How would you plan your moves to **ensure you leave with the gold in hand**?

This is the essence of the **Wumpus World problem**. In this scenario, the agent—our player—must navigate the cave by **making optimal decisions** based only on limited information and logical reasoning. Such problems are a classic example of **knowledge-based reasoning** in **Artificial Intelligence**.

In the term “**Wumpus World**,” the Wumpus represents the monster that threatens the agent, while the **World refers to the cave environment** containing gold and hazards. **The objective** of the agent is simple but challenging: to explore the cave, **collect the treasure**, and finally **climb out** with the maximum possible score.

* 1. **Objectives**

As mentioned above, the agent’s task is to escape the cave with the gold in hand while achieving the highest possible score. But what does “**the highest score**” mean in this context? The scoring system is defined as follows:

|  |  |
| --- | --- |
| **Action** | **Score** |
| Move Forward | +10 |
| Turn left / right | -1 |
| Shoot | -10 |
| Die (fall in pit or eaten by wumpus) | -1000 |
| Climb out (with gold) | +1000 |
| Climb out (without gold) | 0 |

Thus, the **danger** comes not only from the Wumpus but also from the possibility of the agent **falling into a pit**. However, the agent is given a **single arrow** that can be used to kill one Wumpus, or it may choose the safer option of climbing out without collecting the gold.

From actions such as *move*, *turn left*, *turn right*, and *shoot*, the agent must reason out the next optimal step to take. **The objective is not to discover the globally best strategy, but rather to find the gold and escape the cave with the highest possible score.**

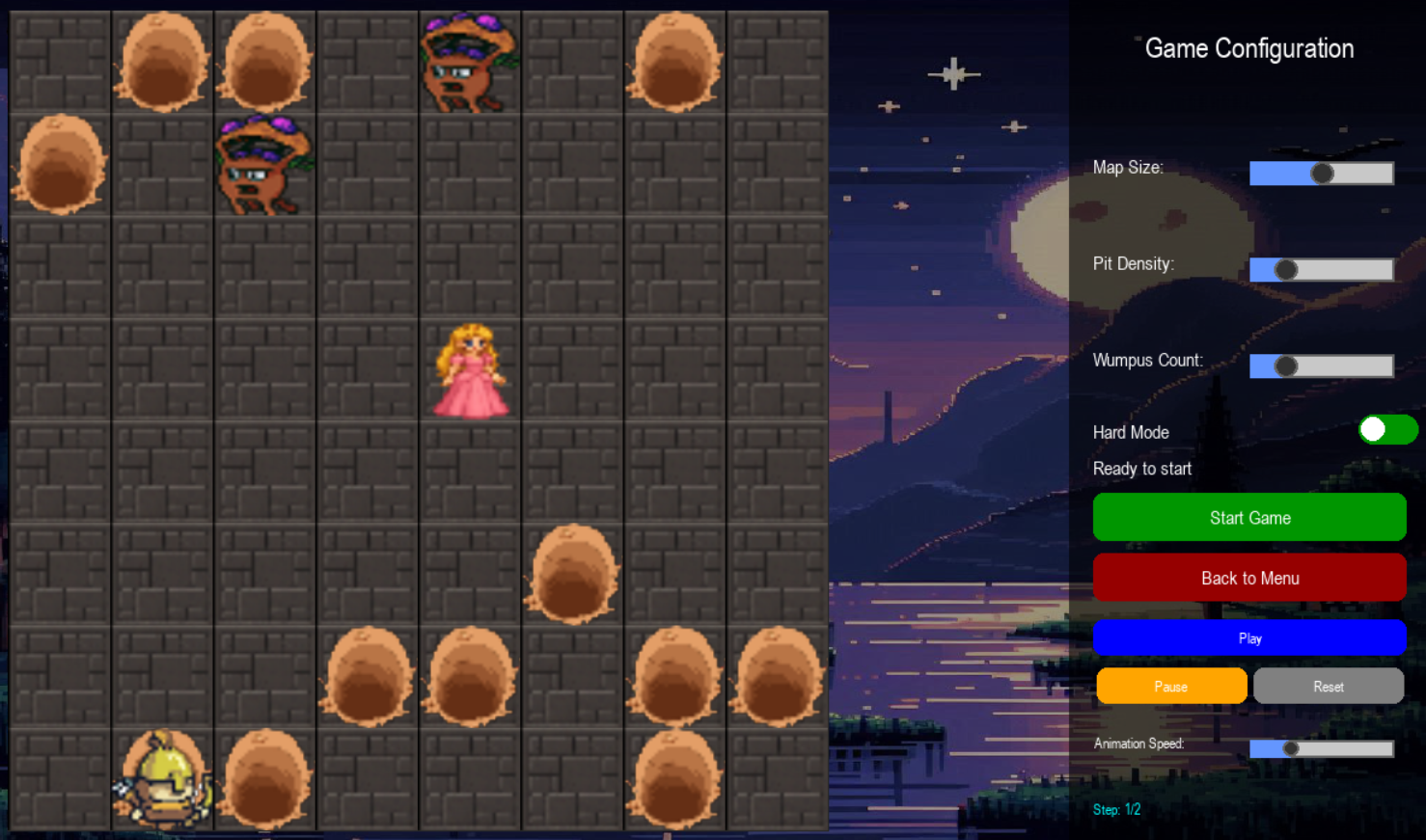
1. **Game Visualization and Design system**

Before going into the detailed definitions of the Wumpus game and the algorithms used to solve the problem, let first look at the **game interface** and provide an introduction to the **project’s code structure**.

* 1. **Game visualization**



***Image 2.a.1****: Menu screen. Include* ***2 buttons****: click* ***play*** *to change to game screen,* ***exit*** *to quit the game.*



***Image 2.a.2****: Game screen. Include* ***2 sides****:* ***Left side*** *displays* ***game’s animation*** *step by step. Meanwhile, the* ***right side*** *displays the* ***game’s UI configuration*** *to adjust the game components, such as* ***Map size*** *(min = 4, max = 8),* ***Pit*** *quantity is calculate by its* ***density*** *(let the size of the map mutiply with the density, min = 0.1, max = 0.5), the* ***number of wumpus*** *(min = 1, max = 5). The list of the button includes:* ***Start Game*** *– to solve the current map display;* ***Back to menu*** *– back to menu screen;* ***Play*** *– play the solving’s animation;* ***Pause*** *– stop the solving’s animation;* ***Reset*** *– reset the animation display. User can also adjust the speed of the animation.*

* 1. **Link video demo**

**LINK**

* 1. **System architecture and module design**

This project is organized into five main areas:

* **Assets**: stores resources such as images and sounds.
* **Design**: contains code related to the user interface, including buttons, sliders, and toggles.
* **Development**: includes code for game logic and algorithm implementation.
* **Screen**: manages the different screens of the game.
* **Main** and **main\_process**: contain the main functions responsible for launching the program. Notify that **main\_process** is used for running the test version of the game quickly.
  + 1. **Design space**

This is where the game’s UI components are programmed. Among them, three key UI elements have the greatest impact on the game: **Text**, **Button**, and **Slider**.

* **Text**: Uses the system font **Arial**. The arguments passed to this object include content, position, size, style, and color. Based on these, methods are **implemented for rendering text** and updating its content.
* **Button**: Takes arguments similar to Text (content, position, size), but requires additional design work to make it visually appealing. This includes background, rounded corners, hover effects (lighter color), click effects (darker color), and an ***onClick*** function to **notify** the system when the **button is pressed** (**mouse click within its area**).
* **Slider**: Used to **adjust game parameters** such as map size and the number of Wumpus. In addition to position and size, the slider requires **min, max, and default values** for user configuration. The implementation also includes a rectangular track and a draggable circle to indicate the current value.
  + 1. **Screen space**

This area manages the player interface screens, implemented using the **State Design Pattern** to allow smooth transitions between pages via the **set\_screen** function.

The most important interface in this project is the **GameScreen**, which directly connects to the algorithm. Referring back to **Figure 2.a.2**, this screen is divided into **two parts**: the **game animation part** and the **algorithm configuration** **part**. In general, the right panel provides input adjustments, while the left panel visualizes the game.

To simplify implementation, the inputs and outputs of the algorithm must be clearly defined.

* **Inputs**: parameters such as map size, pit density, and the number of Wumpus. These are provided to the solver tool, which then builds and executes the algorithm.
* **Outputs**: information returned by the solver to be displayed on the game screen. The three most important outputs are:
  1. The list of actions taken by the Agent.
  2. The list of actions taken by the Wumpus.
  3. The final score obtained after the algorithm finishes.

The algorithmic details will be presented in later sections. Here, we explain how the UI is rendered from the solver’s output. In other words, the solver’s **output** serves as the **input** to the rendering module. Here is an example of the solver output:

* **Score**: -1009
* **Agent actions**: [‘move’, ‘turn\_left’, ‘turn\_right’, ‘shoot’, ...]
* **Wumpus actions**:

[

['stay', 'stay', 'stay', 'stay', 'E', ...]

['stay', 'stay', 'stay', 'stay', 'N', 'stay', …]

]

**Rendering routine:** execute the game step by step by applying each action of both the Agent and the Wumpus. For every received action, update state, then draw the corresponding change on the screen.

**Render Agent:** four primary actions: move, turn\_left, turn\_right, shoot. Maintain two state variables: agent\_pos and agent\_dir.

* **On move:** update agent\_pos by adding the unit vector of agent\_dir.
* **On turn\_left / turn\_right:** rotate agent\_dir accordingly.
* **Shooting animation** is **not implemented** in this project.

**Render Wumpus:** similar idea, but Wumpus movement is not tied to a facing direction. Use a variable (e.g., wumpus\_movement) to track each Wumpus’s step.

* **If action is stay:** no update.
* **If action is one of N, E, S, W** (North, East, South, West): update the position with the corresponding delta.
  + 1. **Main space**

This file contains the program’s runtime functions. It uses **double buffering**: draw each layer once into an off-screen back buffer, then swap it to the front buffer on each tick to display. This reduces flicker and keeps frame timing consistent.

There is an alternate main entry point, **main\_process**, used to test how the algorithm behaves.

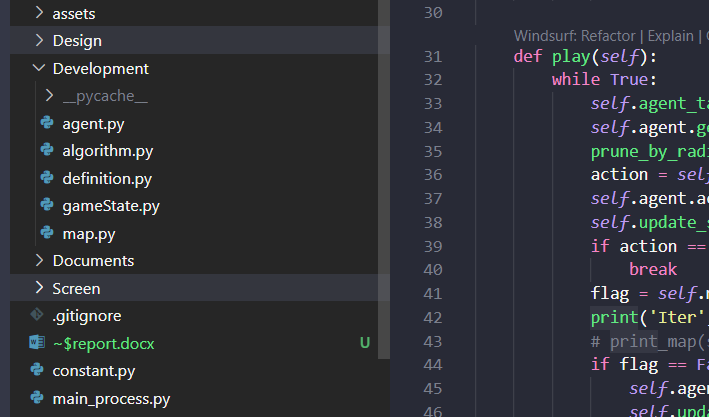
There is also a **constants** file that centralizes and **manages all immutable configuration values** used by the program.

* + 1. **Development space**

If Section 2.b.ii explains how to use the solver’s output, this section is where the solver itself is built. The algorithmic details will be covered later; here we introduce how the game objects are structured.

This section contains five key programming areas:

* **agent**: manages the agent entity.
* **map**: manages the agent’s environment; tracks Wumpus, pits, gold, etc.
* **algorithm**: implements reasoning/planning modules that update the agent’s knowledge and yield actions.
* **definitions**: declares literals/predicates for map cells to support the solver.
* **gameState**: manages results and the algorithm lifecycle (start, run, finish).



***Image 2.b.iv.1****: Directory structure and object management*

After covering the project introduction, the UI, and the code structure. Now this is the suitable moment to move on to the most important part: the algorithms.

1. **Knowledge base (KB) fomulation** 
   1. **Element Definition – Literal**

**Map**

The cave map the agent must explore to find gold. It stores the locations of Wumpus, pits, and gold. It is modeled as an **N × N** grid of **literals** – **boolean facts about each cell** (e.g., There is no wumpus at location x, y ¬Wumpus(x,y)).

At the begin of the solver running, the map computes and assigns ground-truth facts for every cell.

**Wumpus and Stench**

A Wumpus is a lethal monster. It emits a **stench** in adjacent cells.

When the agent is adjacent to a Wumpus, it perceives a stench.

In this report we use **W** for **Wumpus** and **S** for **Stench**. In a 2D grid, a **cell has S if at least one of its four neighbors (N,E,S,W) contains W**.

And now a literal can be display as: ¬W(x, y) There is no wumpus at location x, y.

**Pit and Breeze**

A pit is another lethal hazard. Pits cause a breeze in adjacent cells. We use P for Pit and B for Breeze. In the grid, cells next to a pit have B (breeze).

**Gold and Glitter**

Gold is the objective. The agent **perceives glitter** **only when it stands in a cell containing gold**, at which point it can perform **grab**.

Entering a gold cell implies the Glitter percept; after grabbing, mark the gold as collected.

**Scream and Bump**

**Heard** when the agent **fires its (single) arrow** and a **Wumpus is killed**. The scream confirms a kill along the shot direction.

Occurs when the agent walks into a wall. After a bump the agent can infer the map boundary.

In this project the **agent** **will know the size** at the begin of the game, so **bump effect will not be explained in this report**.

**Agent**

The agent has **four** primitive actions: **move, turn left, turn right, shoot**. When it grabs gold, update the state to mark that cell’s gold as collected. Obviously, the agent’s target is always to **grab gold** and **climb out** safely.

* 1. **Definition foudation** 
     1. **Rules**

A **rule** is an “**if–then**” **statement** that links what the agent senses to what might be in nearby cells.

* If there is Breeze in cell (x,y), then **at least** one neighbor has a Pit.
* If there is no Breeze in (x,y), then **none** of the neighbors has a Pit.
* A cell is **Safe** if it has **no Pit** and **no Wumpus**.
  + 1. **Agent knowledge base**

The **agent’s knowledge base** is its **memory of rules plus facts it has observed**. At each step the agent:

* **Records** what it perceives in the **current cell** (e.g., Breeze, Stench, Glitter),
* Marks cells as Visited, updates Safe cells using the rules,
* Keeps state like HaveArrow or HaveGold.

Then the agent **asks the KB** simple questions: which neighbor cells are safe, should it shoot, should it grab, or should it leave. The planner uses these answers to pick the next action that maximizes score.

More detail about the rule and knowledge base and its implements will be explained the follow part of the report.

1. **Inference engine approach**

**Notes from the previous sections**: The Agent **cannot directly access** detailed information about the Map. Instead, it **only receives percept signals from the Map** and knows the overall size of the map.

The KB of the Agent is organized in **Conjunctive Normal Form (CNF)**. More specifically, a CNF sentence is a **conjunction (AND) of clauses**, where each clause is a **disjunction (OR) of Literals**.

A **Literal** is represented as a class with the following attributes:

* **name:** the symbol (e.g., wumpus, pit, breeze, …)
* **pos:** a coordinate pair (y, x) that defines its position
* **negate:** a boolean flag to indicate whether it is a negated literal
* **at\_step:** the timestep at which the literal was observed (important for handling dynamic updates when the Wumpus moves).

**Inference rule**: The system uses **model checking**. The model’s process follow these steps:

* **Step 1:** **Build** all **possible states** and **positions relevant** to **decision-making** (e.g., current cell, adjacent cells, wumpus in front).
* **Step 2:** For each literal under consideration, collect all related propositional variables (from the KB).
* **Step 3:** Generate truth assignments (models) for these variables. For each assignment:
  + If any clause evaluates to False, discard the assignment.
  + If all clauses are True, then the KB holds under that assignment, and the literal takes its corresponding truth value.
* **Step 4:** If all satisfying models agree on the literal’s value → return **SAFE** or **UNSAFE**. Otherwise → **UNKNOWN**.

**Dynamic KB**:

* Since **Wumpus** and **Stench** can **change over time**, **at\_step** is used to **only keep the latest information**.
* Clauses with outdated dynamic literals are removed, while static literals remain in the KB.
* This avoids contradictions (e.g., having both A and ¬A at the same time, which would make KB unsatisfiable).

**Interaction between Agent and Map**

* **Percepts:** Map receives the Agent’s position, checks the state of the current and adjacent cells, and returns percepts as a **list of Literals**.
* **KB update:** Agent records these percepts, translates them into clauses, and integrates them into its KB. Dynamic literals are deduplicated so that only the most recent timestep is kept.
* **Pruning:** The Agent prunes its **KB by radius**, keeping all static clauses but removing dynamic clauses outside a certain range.
* **Decision-making:** The Agent selects which literals to test, uses **truth-table entailment** to classify cells, and keeps track of recent breezes to remain cautious.
* **Action selection:** Based on the classified cell statuses, the Agent generates possible actions and picks the optimal move (e.g., move, grab, shoot, climb out).
* **Map update:** The Map applies the chosen action, updates the game state, and moves the Wumpus every 5 steps.
* **Loop:** If the game is not over (win/loss), return to Step 1.

**Pseudo code for more illustration**

|  |
| --- |
| **function play(map, agent):**  while true:  ***# 1. Percepts***  percepts = step1\_percepts(map, agent)  ***# 2) KB update***  step2\_update\_kb(agent, percepts)  ***# 3) Pruning***  step3\_prune(agent, radius = 3)  ***# 4) Decision-making***  status\_map = step4\_decide(agent)  ***# 5) Action selection***  action = step5\_choose\_action(mode)  actions.append(action) ***# support save result***  score = update\_score() ***# support save result***  ***# 6) Map update***  flag = step6\_map\_update(action)  ***# 7) Loop control***  if flag == False:  actions.append(die)  return score, actions |

1. **Planning algorithm desription (Lộc)**
2. **Experiment result (Đức)** 
   1. **Test scenarios and Comparison System**

In this section, **two algorithms** are compared: one that **selects moves randomly** and one that **selects moves using a heuristic**.

The evaluation is based on **two criteria**: the **average path length** and the **success rate** of each algorithm.

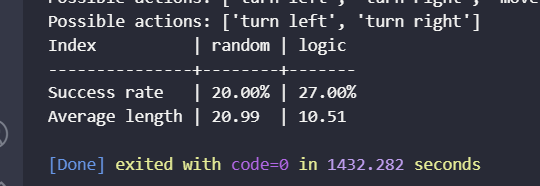
To ensure fairness in the evaluation, the following conditions are required:

* **Common parameters**: both algorithms must use the **same map size**, **the same number of wumpus**, **the same number of pits**, and the **same mode** in which wumpus are able to move.
* **Common evaluation system**: the assessment tool applied to both algorithms must operate in the same way.

Comparison System: **Averaging method**

* Fix the map parameters. Pick a **trial count times**.
* For each strategy, run times independent trials on fresh maps.
* After each trial, record: success or failure and number of actions taken.
* After all trials per strategy:
  + **Success rate**: count the successes and divide by the number of trials.
  + **Avarage Solution length**: add all action counts and divide by the number of trials.

Here is a result when choose **trial count times = 100**:



* 1. **Discussion and insight**

**Observation.**

The heuristic planner shows a higher success rate and a shorter average path than the random baseline.

**Why the success rate is higher.**

Heuristics prioritize safe cells and known fronts. Random picks arbitrary neighbors and **enters** **hazards more often**. The heuristic also backtracks to previously safe cells instead of pushing into unknown high-risk cells. It can still fail if a moving Wumpus steps into a cell believed safe, or if the policy chooses to exit when no safe progress is provable, **but deaths are rarer overall**.

**Why the average path is shorter.**

Heuristics guide exploration toward promising cells and avoid aimless wandering. They reduce loops, dead ends, and unnecessary visits. **Random exploration wastes steps on detours and revisits**, so it typically needs many more actions to reach gold or to terminate.

1. **Team member contributions**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Member** | **Student ID** | **Role** | **Key task** | **Overall Contribution** |
| **Thieu Quang Vinh** | 23127143 | Game system design; baseline **random** strategy | 1) Define game architecture and state flow.  2) Implement random action policy and integration with scoring.  3) Wire percept→KB hooks in the main loop.  4) Draft core system sections of the report. | 100% |
| **Tran Hai Duc** | 23127173 | UI/UX and evaluation framework | 1) Build UI components (Text, Button, Slider) and screens.  2) Render solver outputs on GameScreen.  3) Implement comparison harness and metrics (success rate, avg length).  4) Write the report except Sections 4–5. | 100% |
| **Pham Thanh Loc** | 23127405 | Heuristic planner (**A\*** for optimal pathfinding | 1) Implement logic/heuristic strategy with A\* (cost, heuristic, tie-breaking) and integrate with choose\_action("logic").  2) Write report part 5 | 100% |

1. **Reference and citations**

This project benefited from the Group 7 “RushHour” Project 1, whose UI system patterns informed our design of **main.py** and the **Screen framework**.

We also consulted ChatGPT for specific UI implementation details, including:

* Persisting Wumpus traces,
* Recording gold pickup locations,
* Rendering Wumpus movement.