Project 2. Logic

Designing an Intelligent Wumpus World Agent

Quan Nguyen Hoang

Phuc Thai Hoang

Cuong Tran Tien

Anh Nguyen Tuan

University of Science
Faculty of Information Technology
Introduction to Artificial Intelligence

Ho Chi Minh, August 2025

Project 2. Logic

Designing an Intelligent Wumpus World Agent

Quan Nguyen Hoang
Student No. 23127106

Phuc Thai Hoang
Student No. 23127458

Cuong Tran Tien
Student No. 23127332

Anh Nguyen Tuan
Student No. 23127152

Supervisor: Minh Nguyen Tran Duy
Faculty of Information Technology

University of Science
Faculty of Information Technology
Introduction to Artificial Intelligence

Ho Chi Minh, August 2025

Acknowledgements

We would like to express our sincere gratitude to our lab instructor, Mr. *Minh Nguyen Tran Duy*, for providing us with this engaging and educational project that has significantly enhanced our understanding of search algorithms. We are equally grateful to Ms. *Thao Nguyen Ngoc* for her exceptional lectures and dedication to teaching, as her clear explanations and comprehensive coverage of Artificial Intelligence concepts have provided us with the foundational knowledge necessary to successfully complete this project. We also extend our thanks to the authors of the research papers referenced in this work, whose insights into designing logical agents have greatly informed our implementation, as well as to the creators of the stock images and graphical assets used in our GUI, which have enhanced the visual appeal and user experience of our application. This project has been a valuable learning experience, and we appreciate all those who have contributed to its success.

Contents

List of Figures

List of Tables

1 Group information

This chapter provides essential information regarding the project and group members. We began by thoroughly reviewing the project requirements (Minh Nguyen Tran Duy, 2025), dividing tasks, and assigning them to each member in a fair manner so that everyone in the group has an opportunity to develop ideas, implement solutions through coding, and conduct analysis through written documentation. We monitor team members' progress using *Google Sheets* (summarized in Table 1.2) and hold weekly meetings to evaluate and provide feedback on their work while assessing completion rates.

1.1 Group members

Our group consists of four members, details are provided in Table 1.1.

Table 1.1: *Group member information and student details.*

Student ID	Student Name
23127106	Quan Nguyen Hoang
23127458	Phuc Thai Hoang
23127332	Cuong Tran Tien
23127152	Anh Nguyen Tuan

1.2 Task distribution and descriptions

Tasks are divided and grouped into categories: *Management*, *Coding*, *Report Writing*, *Video* and *Other Tasks*, as shown in Table 1.2.

Table 1.2: *Detailed task distribution for the Wumpus World Agent project.*

Task ID	Task Description
T001	**LaTeX Document Formatting**
Writing report	*Assignee: Quan Nguyen Hoang*

Continued on the next page.

Table 1.2 continued from previous page.

Task ID	Task Description
	Ensure proper formatting, styling, and layout of the project report using LaTeX.
T002	**Environment Simulator Implementation**
Coding	*Assignee: Quan Nguyen Hoang* *Develop environment simulator for Wumpus World including map and percept generation, dynamic Wumpus movement, and visualization.*
T003	**Inference Engine Implementation**
Coding	*Assignee: Quan Nguyen Hoang* *Implement DPLL-based logic inference engine to derive safe, pit, and Wumpus cell statuses from percepts.*
T004	**Planning Module Implementation**
Coding	*Assignee: Phuc Thai Hoang* *Implement A* planner with Manhattan heuristic and risk-aware cost to generate action plans.*
T005	**Hybrid Agent Integration**
Coding	*Assignee: Anh Nguyen Tuan* *Integrate simulator, inference engine, and planner into a unified hybrid agent control loop.*
T006	**Random Agent Baseline Implementation**
Coding	*Assignee: Cuong Tran Tien* *Implement a random-action agent for baseline performance comparison.*
T007	**Advanced Setting – Moving Wumpus Module Implementation**
Coding	*Assignee: Anh Nguyen Tuan* *Extend environment simulator to move Wumpuses every five actions and update KB on percept changes.*
T008	**GUI Design and Implementation**
Coding / GUI	*Assignee: Cuong Tran Tien* *Design and develop GUI to visualize agent state, percepts, knowledge, and actions in real time.*
T009	**Update README.md**
Other tasks	*Assignee: Phuc Thai Hoang* *Create comprehensive installation and usage instructions in README.md.*
T010	**Video Recording**
Video	*Assignee: Anh Nguyen Tuan* *Record a demo video showcasing simulator, inference, planning, and GUI functionalities.*
T011	**Problem Introduction and Objectives**
Writing report	*Assignee: Phuc Thai Hoang* *Introduce the Wumpus World problem and define the project's goals and scope.*
T012	**Formulation of Knowledge Base**
Writing report	*Assignee: Quan Nguyen Hoang* *Formulate propositional logic rules for breeze, stench, pit, Wumpus, and gold percepts.*
T013	**System Architecture and Module Design**
Writing report	*Assignee: Quan Nguyen Hoang*

Continued on the next page.

Table 1.2 continued from previous page.

Task ID	Task Description
	Document the overall system architecture and design of simulator, inference engine, and planner modules.
T014	**Inference Engine Approach**
Writing report	*Assignee: Phuc Thai Hoang* *Describe the DPLL-based entailment process and handling of percept-driven KB updates.*
T015	**KB Update Strategy for Moving Wumpus**
Writing report	*Assignee: Anh Nguyen Tuan* *Explain the strategy for updating the knowledge base in response to dynamic Wumpus movements.*
T016	**Planning Algorithm Description**
Writing report	*Assignee: Cuong Tran Tien* *Provide a detailed explanation of the A* planning algorithm, heuristic design, and risk penalties.*
T017	**Experiment Results**
Writing report	*Assignee: Cuong Tran Tien* *Document and analyze comparative results between hybrid and random agents across metrics.*
T018	**Team Member Contributions**
Writing report	*Assignee: Quan Nguyen Hoang* *Summarize the contributions of each team member to project implementation and reporting.*

1.3 Group member contribution summary

Tasks listed below are identified by their corresponding task IDs, with detailed descriptions provided in Section 1.2. We have discussed and agreed on the completion rate of each member.

> **Note**
>
> The number of assigned tasks does not reflect each member's workload, but rather the difficulty and complexity of the tasks. Task distribution aims to achieve balance in workload among group members.

1.3.1 23127106 - Quan Nguyen Hoang

- **Completion rate:** *100%*
- **Tasks assigned:** T001, T002, T003, T012, T013, T018

1.3.2 23127458 - Phuc Thai Hoang

- **Completion rate:** *100%*
- **Tasks assigned:** T004, T009, T011, T014

1.3.3 23127332 - Cuong Tran Tien

- **Completion rate:** *100%*
- **Tasks assigned:** T006, T008, T016, T017

1.3.4 23127152 - Anh Nguyen Tuan

- **Completion rate:** *100%*
- **Tasks assigned:** T005, T007, T010, T015

1.4 Estimation of completion rate for each requirement

Table 1.3: *Completion rates for each grading criterion of Project 2.*

Criteria ID	Criteria	Completion rate
1	Problem formulation and knowledge base design	100%
2.1	Environment simulator implementation	100%
2.2	Inference engine implementation	100%
2.3	Planning module implementation	100%
2.4	Hybrid agent integration	100%
2.5	Advanced moving Wumpus module	100%
3	Report and analysis	100%

2

Problem introduction and objectives

2.1 Problem introduction

The **Wumpus World** is a benchmark environment for knowledge-based agents in artificial intelligence, it was introduced by Michael Genesereth and later discussed by Russell et al. (2021). It consists of an $N \times N$ grid (default $N = 8$) with cells that may contain one piece of **gold**, multiple stationary hazards: **pits** (with default probability $p = 0.2$), and K **Wumpuses** (default $K = 2$). The agent begins at an initial safe cell $(0, 0)$, facing east, with no prior knowledge of where hazards or gold lie. As it explores, the agent receives local **percepts**:

- **Stench**: when a Wumpus occupies an adjacent cell.
- **Breeze**: when a pit occupies an adjacent cell.
- **Glitter**: when gold occupies the current cell.
- **Bump**: if it attempts to move into a wall.
- **Scream**: when a Wumpus is killed by its arrow.

In the standard setting, hazards and gold remain fixed. An advanced scenario introduces *moving* Wumpuses, which relocate randomly to an adjacent cell every five agent actions (Minh Nguyen Tran Duy, 2025). This dynamic behavior requires the agent to continuously update its beliefs and adapt its plan as Wumpus positions change.

2.2 Objectives

The goal is to design and implement a **hybrid agent** to grab the gold and return to $(0, 0)$, climb out safely and maximizing score [1], in which it integrates:

[1] The agent may successfully grab the gold and climb out safely or die, since our design involves risk-taking behavior when no safe, unvisited cells remain.

1. **Environment simulator**
   - Randomly generate maps with adjustable grid size, pit density, and Wumpus count.
   - Produce accurate percepts (*stench, breeze, glitter, bump, scream*) and support *moving* Wumpuses.
   - Visualize both the true world state and the agent's internal map.
2. **Inference engine**
   - Encode breeze–pit and stench–Wumpus relationships in **propositional logic** (CNF).
   - Implement a **DPLL-based** solver [2] to infer which cells are safe or not.
   - Continuously revise the knowledge base after each percept and after Wumpus movements.
3. **Planning module**
   - Implement optimal path-finding (here we use **A***) balancing path cost, risk, and expected utility.
   - Incorporate risk penalties for entering unknown cells.
   - Replan as new inferences become available.
4. **Hybrid agent integration**
   - Combine percept processing, logical inference, and planning into a unified decision loop.
   - Execute actions (*move, turn, grab, shoot, climb*) to retrieve gold and return safely.
   - Handle collisions (pit or Wumpus) and scoring to maximize utility.
5. **Baseline and evaluation**
   - Implement a random agent for comparison.
   - Conduct experiments over some maps to measure success rate, average score and decision efficiency.
6. **Deliverables**
   - Well-implemented source code.
   - A demo video illustrating the GUI.
   - A comprehensive report covering all things mentioned above.

[2] Later on we will explain why we choose this over other inferencing algorithms.

3

Formulation of Knowledge Base rules using Propositional Logic

In this chapter, we detail how our agent's `InferenceEngine` constructs a **propositional logic Knowledge Base (KB)** from sensory input, enabling inferencing procedures later on. We begin by defining the grid-indexed **propositional symbols** for hazards ($P_{x,y}$, $W_{x,y}$), percepts ($B_{x,y}$, $S_{x,y}$, $L_{x,y}$), and gold ($G_{x,y}$). We then derive the **rules/ axioms** needed for the Wumpus World, following Minh Nguyen Tran Duy (2025) and Russell et al. (2021).

In our `InferenceEngine` implementation, those clauses are programmatically asserted into the KB via calls to `KnowledgeBase.tell()`, and entailment queries are issued via `KnowledgeBase.ask()`.

3.1 Propositional symbols

We introduce the following propositional symbols for each cell (x, y) in the grid:

- $P_{x,y}$: "there is a pit in cell (x, y)."
- $W_{x,y}$: "there is a Wumpus in cell (x, y)."
- $B_{x,y}$: "agent perceives a breeze in cell (x, y)."
- $S_{x,y}$: "agent perceives a stench in cell (x, y)."
- $G_{x,y}$: "there is gold in cell (x, y)."
- $L_{x,y}$: "agent perceives a glitter in cell (x, y)."

Let the (Manhattan) neighbors of (x, y) be[1]

$$N(x, y) = \{(i, j) \mid |i - x| + |j - y| = 1,\ 0 \le i, j < \text{size}\}. \tag{3.1}$$

[1] Each cell has at most 4 neighbors (adjacent cells) in 4 directions.

3.2 KB axioms

Below are the KB axioms—general knowledge of how the Wumpus World works.

Note: Implementation details for these rules will be discussed in next chapters.

3.2.1 Breeze–Pit biconditional

A breeze is perceived in (x, y) if and only if at least one adjacent cell contains a pit:

$$B_{x,y} \longleftrightarrow \bigvee_{(i,j)\in N(x,y)} P_{i,j}. \tag{3.2}$$

In conjunctive normal form (CNF), this yields:

$$\neg B_{x,y} \vee \bigvee_{(i,j)\in N(x,y)} P_{i,j} \tag{3.3}$$

$$\bigwedge_{(i,j)\in N(x,y)} \big(B_{x,y} \vee \neg P_{i,j}\big). \tag{3.4}$$

3.2.2 Stench–Wumpus biconditional

Similarly, a stench is perceived exactly when one or more neighboring cells contain a Wumpus:

$$S_{x,y} \longleftrightarrow \bigvee_{(i,j)\in N(x,y)} W_{i,j}, \tag{3.5}$$

with CNF clauses:

$$\neg S_{x,y} \vee \bigvee_{(i,j)\in N(x,y)} W_{i,j} \tag{3.6}$$

$$\bigwedge_{(i,j)\in N(x,y)} \big(S_{x,y} \vee \neg W_{i,j}\big). \tag{3.7}$$

3.2.3 Gold–Glitter biconditional

The glitter percept corresponds directly to the presence of gold:

$$L_{x,y} \longleftrightarrow G_{x,y}, \tag{3.8}$$

encoded in CNF as:

$$\neg L_{x,y} \vee G_{x,y} \tag{3.9}$$

$$L_{x,y} \vee \neg G_{x,y} \tag{3.10}$$

In implementation, we omit the gold–glitter biconditional because when the agent observes a glitter, it immediately fixes the gold's location, making the CNF clauses redundant.

3.2.4 Global and initialization constraints

- **Start cell is always safe.** The initial cell $(0, 0)$ is free of hazards:

$$\neg P_{0,0}, \quad \neg W_{0,0}.$$

- **Exactly one gold.** To enforce a unique gold location:

$$\bigvee_{x,y} G_{x,y} \quad \text{and} \quad \bigwedge_{(x,y)\neq(i,j)} (\neg G_{x,y} \vee \neg G_{i,j}). \tag{3.11}$$

  In implementation, we drop the CNF encoding for this rule, since once the agent perceives a glitter in some cell, it immediately fixes the gold's location and can ignore every other cells.
- **Exactly K Wumpus.** The number of Wumpus K is known, we can add:

$$\sum_{x,y} W_{x,y} = K \tag{3.12}$$

  via standard at-least and at-most cardinality encodings, but this is not necessary. In our implementation, we omit these to avoid CNF blow-up, instead the agent simply keeps an internal counter of how many Wumpi it believes remain, and updates that count when it fires an arrow and hears a scream.

3.3 Remarks on multiple pits and wumpus

Because propositional logic does not support quantifiers, we assign each potential pit and each potential Wumpus its own local biconditional clause. The conjunction of these biconditionals over all cells suffices to capture any number of pits or Wumpuses without FOL-style quantification, while still enabling sound and complete inference from percepts alone.

4

System architecture and module design

The **Wumpus World agent** is implemented through a clear separation of concerns across several collaborating modules. This architecture ensures that environment simulation, knowledge representation, logical inference, planning, and agent control remain decoupled for ease of development, testing, and extension. The system is designed to be robust, supporting both a static environment and an advanced setting with a moving Wumpus.

4.1 Module design and class definitions

Below we summarize each module's purpose, its core attributes, and methods. The goal is to convey how functionality is designed rather than delve into exhaustive coding details.

4.1.1 `Environment` module

The `Environment` class simulates the Wumpus world grid, manages its state, and enforces its rules. It is the single source of truth for the world's configuration.

Attributes:

- `size`: The dimension of the N x N grid (default N=8).
- `pit_positions`, `wumpus_positions`, `gold_position`: Sets and tuples storing the locations of hazards and treasure.
- `agent_state`: A dataclass object tracking the agent's current coordinates, direction, score, and status flags (`alive`, `has_gold`, `has_arrow`).
- `moving_wumpus_mode`: A boolean flag that enables the advanced dynamic environment.
- `agent_action_count`: An integer that counts agent actions to trigger Wumpus movement every 5 steps in the advanced mode.

Methods:

- `reset()`: Initializes a new game by calling internal helpers to randomly generate pits, Wumpuses, and gold, and resets the agent's state.
- `get_percept()`: Returns a `Percept` object for the agent's current location, indicating the presence of *Stench, Breeze, Glitter, Bump,* or *Scream*.
- `execute_action(action)`: Updates the world state based on the agent's action. It adjusts the score (-1 for movement, -10 for shooting, +1000 for a successful climb), checks for agent death, and returns the resulting percept. In moving Wumpus mode, it calls `_move_wumpuses()` after every 5 actions.
- `_move_wumpuses()`: An internal method that handles the movement logic for all Wumpuses, including random direction choice and collision rules (a Wumpus stays in place if its intended move is invalid).
- `display()`: Renders a text-based representation of the true world state, showing all components for debugging and visualization.

4.1.2 `MapKnowledge` module

This module encapsulates the agent's internal representation of the world, which is built and refined over time.

Attributes:

- `grid`: A dictionary mapping '(x, y)' coordinates to `Cell` objects. Each `Cell` tracks its inferred `status` (`UNKNOWN`, `WUMPUS`, `PIT`, `SAFE`), its `visited` status, and the last perceived percepts (`stench`, `breeze`).
- `num_wumpus`: A counter for the number of Wumpuses believed to be alive.

Methods:

- `update_after_visit(x,y,percept)`: Marks a cell as visited, updates its status to `SAFE`, and records the latest percepts received at that location.
- `update_cell_status(x,y,status)`: Sets a cell's inferred status based on deductions from the `InferenceEngine`.
- `reset_wumpus_knowledge()`: A crucial method for the moving Wumpus setting. It resets all stench information and reverts the status of any non-visited cells from `SAFE` back to `UNKNOWN` if they are adjacent to other unknown cells, effectively forcing a re-evaluation of safety.
- `display_agent_view()`: Shows the agent's current knowledge map, providing a visual representation of its beliefs for debugging.

4.1.3 `InferenceEngine` module

This module performs logical reasoning using a propositional logic knowledge base (KB) to deduce the state of unvisited cells.

Attributes:

- `knowledge`: A reference to the agent's `MapKnowledge`.
- `kb`: A `KnowledgeBase` object storing facts and rules as a set of clauses in Conjunctive Normal Form (CNF).
- `processed_cells`: A set of coordinates for cells whose percepts have already been added to the KB, preventing redundant axiom generation.

Methods:

- `run_inference()`: Adds new facts from newly visited cells to the KB by creating biconditional sentences (e.g., $B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$). It then queries the KB to determine if adjacent, unvisited cells are provably `SAFE`, a `PIT`, or a `WUMPUS`, and updates `MapKnowledge` accordingly.
- `_add_biconditional(...)`: A helper to encode the relationship between a percept and its potential causes into CNF and add it to the KB.
- `reset_kb()`: Clears the entire knowledge base and the set of processed cells. This is called every 5 moves in the dynamic setting to discard outdated Wumpus-related knowledge.

4.1.4 `Planner` module

The planner is responsible for generating sequences of actions to achieve a goal, such as reaching a specific cell.

Attributes: The agent's `MapKnowledge` and the grid size.

Methods:

- `find_path(start, goal)`: Implements the A* search algorithm to find an optimal path. The path cost is not uniform; the `_get_action_cost` method assigns a higher cost for moving into `UNKNOWN` cells, making the planner inherently risk-averse.
- `find_least_risky_unknown()`: An exploration strategy used when no guaranteed safe cells are available. It evaluates all `UNKNOWN` cells, estimating their risk based on percepts in neighboring squares. It returns the cell with the lowest estimated risk, using Manhattan distance as a tie-breaker.

4.1.5 `HybridAgent` module

This is the main agent class, orchestrating the perception-reasoning-action loop by integrating all other modules.

Attributes: References to the `Environment`, `MapKnowledge`, `InferenceEngine`, and `Planner`; the agent's internal `state`; and an `action_plan` queue (a list of `Actions`).

Methods:

- `run()`: The main control loop that repeatedly calls `think()`, executes an action from the `action_plan`, updates its state, and checks for termination conditions.

- `think()`: The agent's cognitive center. It follows a strict hierarchy of objectives: 1) `GRAB` if on gold, 2) `CLIMB` if at (0,0) with gold, 3) execute the next action if a plan already exists, or 4) generate a new plan if the current one is empty.
- `_plan_exploration()`: Defines the agent's exploration strategy. It first attempts to find a path to the nearest unvisited `SAFE` cell. If none exist, it falls back to a "gambling" strategy by calling `find_least_risky_unknown()` to choose a target.
- `_update_state(action, percept)`: Updates the agent's internal state (position, direction, etc.) after an action is executed.

4.1.6 `RandomAgent` module

A baseline agent provided for comparison. It acts without logical inference or planning.

Attributes: A simplified set of attributes mirroring the `HybridAgent` but without inference or planner references.

Methods:

- `think()`: A simple decision-maker. It performs essential actions like grabbing gold, but for all other situations, it calls `_choose_random_action()` to pick a move, ignoring all percepts except for glitter.
- `_choose_random_action()`: A helper method that returns a uniformly random, legal action.
- `_update_state()`: Updates the agent's internal state after action execution.

4.2 Hybrid agent reasoning loop

The `HybridAgent.run()` method drives the core decision-making cycle. This loop integrates perception, inference, and a multi-layered planning process. It proceeds as follows:

1. **Perception:** The agent retrieves the current percepts (*stench, breeze, glitter, bump, scream*) from its location by calling `Environment.get_percept()`.
2. **Knowledge update:** It immediately updates its internal map by calling `MapKnowledge.update_after_visit()`, marking the current cell as visited and safe, and recording the new percepts.
3. **Logical inference:** The agent executes `InferenceEngine.run_inference()`. The engine uses the new percepts to add facts to its KB and queries it to deduce the status of adjacent, unvisited cells, refining the agent's knowledge map.
4. **Goal assessment:** The agent first checks for time-sensitive or critical objectives. If an existing plan is in progress, it continues with it. Otherwise, it assesses:
    - If `glitter` is perceived, the action plan is cleared and set to [`GRAB`].
    - If the agent has the gold and is at the starting cell (0,0), the plan is set to [`CLIMB`].

- **Shooting:** If there are no remaining unvisited safe cells, the agent attempts to form a plan to shoot a known Wumpus by calling `_plan_shoot()`. This makes hunting a primary strategy when safe exploration is no longer an option.

5. **Planning:** If the action plan is still empty after the checks, the agent formulates a new plan:
   - If carrying gold, it calls `Planner.find_path()` to find the safest route back to (0,0).
   - If exploring, it follows a clear hierarchy within `_plan_exploration()`:
     (a) First, it seeks a path to the nearest known `SAFE` and unvisited cell.
     (b) If none are found, but a Wumpus location is known, it calls `_plan_align_wumpus()` to move to a safe position from which it can shoot later.
     (c) Only if neither of the above options is available does it fall back to a "gambling" strategy, using `find_least_risky_unknown()` to plan a path to the most promising `UNKNOWN` cell.
6. **Action execution (and Knowledge reset when needed):** The agent pops the next action from its plan and sends it to `Environment.execute_action()`. After the action, it updates its own internal state via `_update_state()`. It then performs conditional knowledge resets:
   - If the action was `SHOOT`, a targeted reset is triggered via `inference_engine.reset_kb_after_shoot()` to update beliefs along the arrow's path.
   - If in moving Wumpus mode, a broader reset is triggered via `inference_engine.reset_kb()` after every 5 actions.
7. **Termination Check:** The loop continues until the agent dies, successfully climbs out, or runs out of planned actions and cannot formulate a new one (i.e., it is stuck and has no more moves).

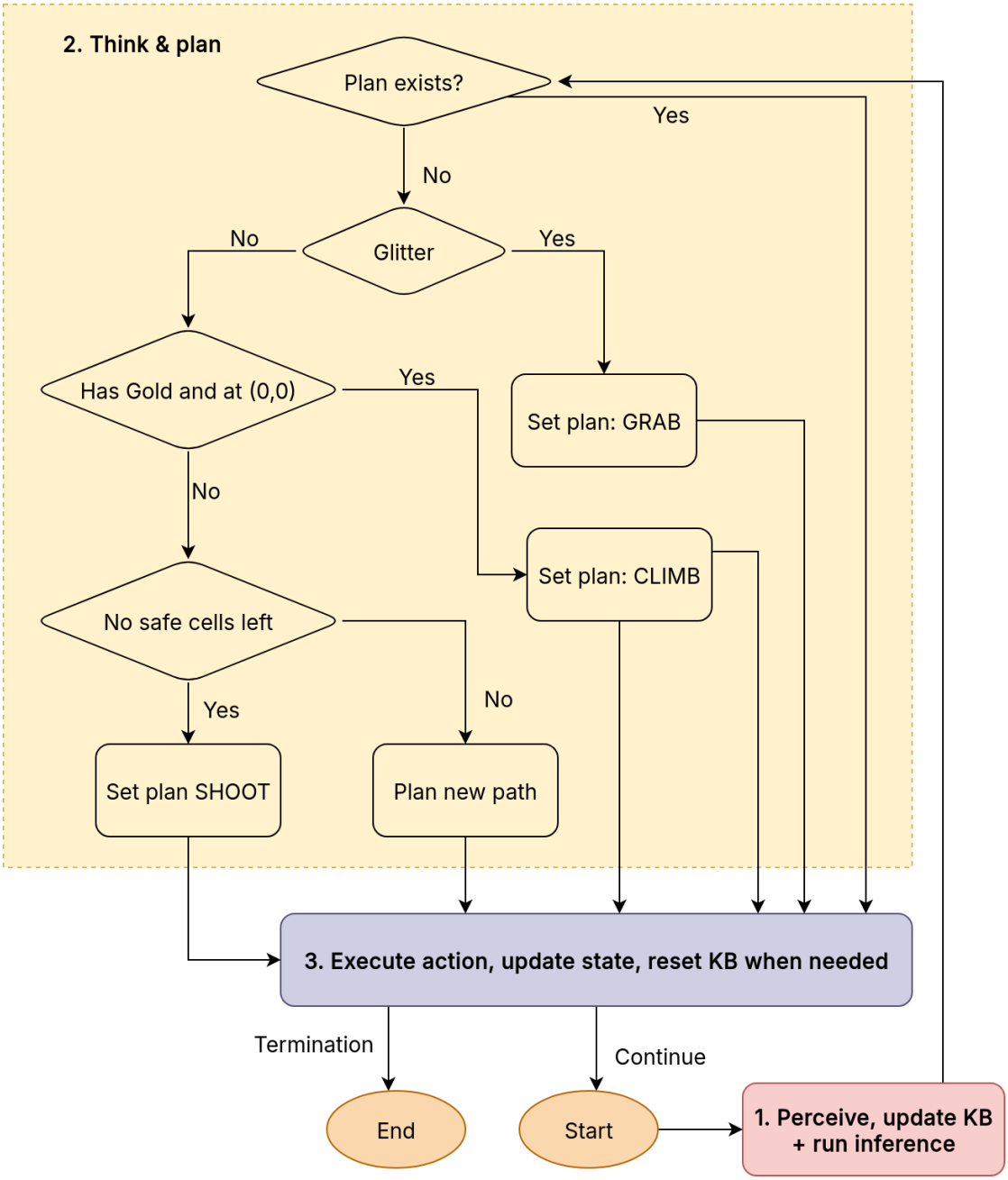

Figure 4.1: *The reasoning loop of the hybrid agent.*

5

Inference engine and Knowledge Base approach

The agent's ability to navigate the Wumpus World safely relies on its capacity to reason logically about its surroundings. This is achieved through a combination of a formal knowledge base (KB) that stores facts about the world and an inference engine that uses this KB to deduce new, unobserved information. This chapter details our choice of inference algorithm, the implementation of our knowledge base and inference engine, and our strategy for adapting this system to the advanced setting with a moving Wumpus.

5.1 Inference algorithm selection

A critical design decision was the selection of a suitable inference algorithm. The algorithm must be both *sound* (only deriving entailed sentences) and *complete* (able to derive any entailed sentence) to ensure the agent's conclusions are reliable. We considered several standard algorithms from propositional logic (Russell et al., 2021).

5.1.1 Algorithm considerations

- **Model checking:** This algorithm works by enumerating all possible models (all possible assignments of true/false to propositional symbols). To prove that the KB entails a query α, it checks that in every model where the KB is true, α is also true. While sound and complete, its time complexity is $O(2^n)$ for n symbols. For an 8x8 grid, the number of symbols for pits and Wumpuses is large ($2 \times 8 \times 8 = 128$), making this approach computationally infeasible.
- **Forward chaining:** This algorithm is highly efficient, often running in linear time. It works by repeatedly applying logical rules to the known facts in the KB to derive new facts, continuing until the query is proven or no more inferences can be made. However, its major limitation is that it is only complete for Horn clauses

(clauses with at most one positive literal). Our KB is more expressive; for example, the axiom for a breeze at (1,1), $B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$, translates to non-Horn clauses like $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$. Therefore, forward chaining is not suitable for our problem.

- **PL-Resolution:** This is a sound and refutation-complete inference rule. It works by converting all sentences to Conjunctive Normal Form (CNF) and proving that $(KB \wedge \neg\alpha)$ is unsatisfiable by repeatedly applying the resolution rule to derive a contradiction (an empty clause). While powerful, a naive implementation can be inefficient as the search for a proof can explore a vast number of clauses.
- **DPLL algorithm:** The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a sound and complete backtracking search algorithm for determining the satisfiability of a propositional logic formula in CNF (Davis et al., 1962). It is essentially a highly optimized form of model checking. By employing powerful heuristics such as *pure symbol elimination* and *unit clause propagation*, DPLL can prune enormous sections of the search space, making it far more efficient in practice than naive model checking or resolution (Biere, 2009).

Given these considerations, we selected the **DPLL algorithm** as the foundation for our inference engine. It offers the full expressive power required by our general propositional logic KB while providing practical efficiency, making it a robust choice for this Wumpus World problem.

5.2 Knowledge base implementation

Our `KnowledgeBase` class, implemented in `inference.py`, is designed to store propositional logic sentences in Conjunctive Normal Form (CNF). A sentence in CNF is a conjunction of clauses, where each clause is a disjunction of literals (Russell et al., 2021). We represent a literal as a tuple '(symbol, truth-value)', a clause as a 'frozenset' of literals, and the entire KB as a 'set' of these clauses.

The class provides two primary methods:

- `tell(clause)`: This method adds a new clause to the knowledge base. The clause is represented as a frozenset of literals, making it immutable and hashable for efficient storage in the master set.
- `ask(query)`: This method determines if the knowledge base entails the given query. It operates on the principle of *proof by refutation*. To check if $KB \models \alpha$, we check if the sentence $(KB \wedge \neg\alpha)$ is unsatisfiable. The `ask` method implements this by negating the query, converting it into a set of clauses, adding these to a temporary copy of the KB, and then calling our satisfiability solver.

The core of the satisfiability check is the `dpll_satisfiable` function, which implements the DPLL algorithm. If this function returns 'False' (indicating that $KB \wedge \neg\alpha$ is unsatisfiable), then the entailment holds, and the `ask` method returns 'True'.

5.3 Inference engine implementation

The `InferenceEngine` module acts as the bridge between the agent's high-level `MapKnowledge` and the formal `KnowledgeBase`. It is responsible for translating percepts into logical axioms and using the KB to deduce the status of unknown cells. This engine is invoked in every 'think' cycle of the `HybridAgent`.

5.3.1 Translating percepts into axioms

The engine first converts percept information into logical sentences that the KB can understand.

- The helper method `_pos_to_symbol(prefix, x, y)` creates a consistent string representation for a proposition, such as `P_1_2` for a pit at (1,2).
- The core translation is handled by `_add_biconditional(...)`. This method takes a percept (e.g., Breeze) at a given cell and its potential causes (e.g., Pits in adjacent cells) and generates the corresponding biconditional axiom. For example, for a breeze at (x, y), it formulates the rule $B_{x,y} \Leftrightarrow (P_{\text{neighbor1}} \vee P_{\text{neighbor2}} \vee \dots)$. This rule is then converted to CNF and added to the KB. It correctly handles both cases:
  - **Percept is true**: The KB is told $B_{x,y}$ and $(\neg B_{x,y} \vee P_{\text{neighbor1}} \vee P_{\text{neighbor2}} \vee \dots)$.
  - **Percept is false**: The KB is told $\neg B_{x,y}$, which implies that all neighbors are safe from that specific hazard. This is efficiently added as multiple clauses: $(B_{x,y} \vee \neg P_{\text{neighbor1}})$, $(B_{x,y} \vee \neg P_{\text{neighbor2}})$, etc.

5.3.2 The main inference loop

The `run_inference` method orchestrates the full reasoning process each turn.

1. **Add facts:** It iterates through all visited cells that have not yet been processed. For each, it asserts that the cell itself contains no pit or Wumpus (since the agent survived) and adds the axioms for its breeze and stench percepts using `_add_biconditional`.
2. **Query unknown cells:** It identifies the set of unvisited cells that are adjacent to the agent's current location as candidates for inference.
3. **Update knowledge:** For each candidate cell (x, y), it queries the KB to update the agent's map:
   - It asks if a pit is certain: $KB \models P_{x,y}$? If yes, `MapKnowledge` is updated.
   - It asks if a Wumpus is certain: $KB \models W_{x,y}$? If yes, `MapKnowledge` is updated.
   - It asks if the cell is provably safe: $(KB \models \neg P_{x,y}) \wedge (KB \models \neg W_{x,y})$? If both are true, the cell is marked as `SAFE`.

5.4 Handling of moving Wumpus in the advanced setting

The advanced setting, where the Wumpus moves every five agent actions, introduces a significant challenge: the agent's knowledge can become outdated and contradictory.

A previously recorded stench may no longer be valid, or a new stench may appear. Our strategy handles this dynamism with a periodic, targeted knowledge reset.

The core of the strategy is to maintain an incremental KB for efficiency between Wumpus movements, but to perform a full reset of Wumpus-related knowledge immediately after a move is expected to have occurred.

1. **Triggering the reset:** The `HybridAgent` maintains an `agent_action_count`. In its main `run` loop, after executing its action and before starting the next `think` cycle, it checks if `agent_action_count % 5 == 0`. If true, it calls `inference_engine.reset_kb()`.
2. **The reset mechanism:** The `InferenceEngine.reset_kb()` method orchestrates the purge of stale information:
   - It discards the entire logical knowledge base by setting `self.kb = None`.
   - It clears the set of `processed_cells`, ensuring that percepts from all previously visited cells will be re-evaluated and re-asserted into the new KB on the next inference run.
   - Most importantly, it calls `self.knowledge.reset_wumpus_knowledge()` to update the agent's high-level map representation.
3. **Updating the agent's map:** The `MapKnowledge.reset_wumpus_knowledge()` method performs a careful, partial reset of the agent's beliefs:
   - It resets the `stench` value of all visited cells to 'None', as this information is now unreliable.
   - It changes the status of any cell previously inferred to be a `WUMPUS` back to `UNKNOWN`.
   - It conservatively re-evaluates cells previously marked as `SAFE`. Any `SAFE` cell that is adjacent to an `UNKNOWN` cell is reverted to `UNKNOWN`, because a Wumpus could have moved into that neighboring unknown square.

By wiping and rebuilding its Wumpus-related knowledge every five steps, the agent ensures its KB remains consistent with the current world state, allowing it to reason correctly based on the freshest percepts available. Knowledge about static pits is preserved across resets, maintaining efficiency.

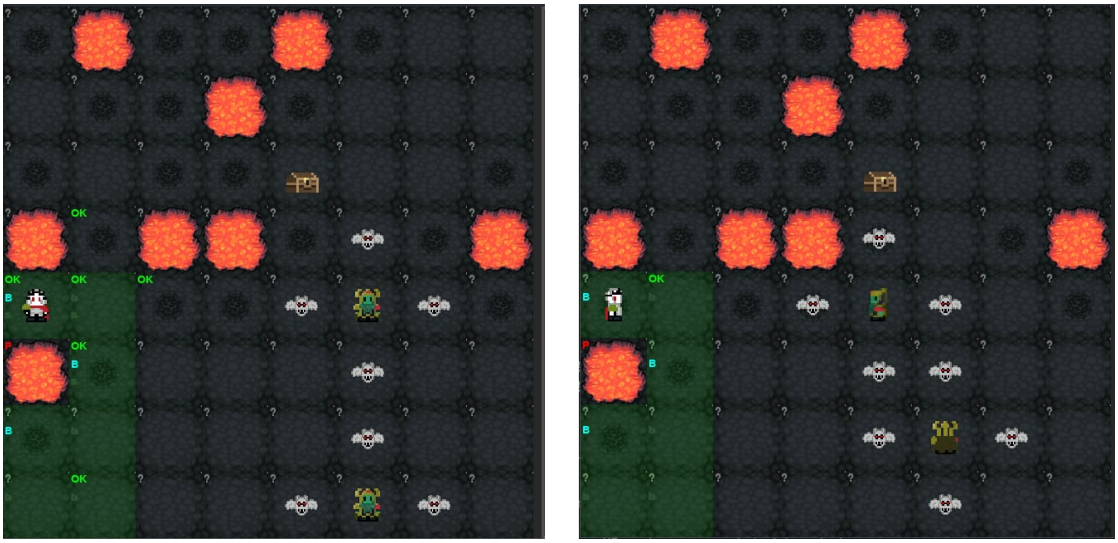

Figure 5.1: *An example of KB and agent's internal map reset. The agent and environment states after 4th action (left) and after 5th action (right). KB reset is triggered, wumpuses move; OK cell (1, 3) remains OK since it is adjacent to 4 OK cells after 4th action; other OK cells are reset to UNKNOWN; pits and breezes are preserved.*

6 Planning algorithm description

While the inference engine determines what the agent knows, the planning module decides what the agent does. It is responsible for generating sequences of actions to navigate the world efficiently and safely. The planner's primary role is to compute paths to given destinations, but it also encapsulates the logic for making intelligent, risk-averse decisions about where to explore next when no guaranteed safe path is obvious.

Our implementation, found in `planning.py`, is centered around the A* search algorithm, enhanced with a custom, risk-aware cost function.

6.1 A* Search for pathfinding

For navigating from its current position to a target destination, the agent employs the A* search algorithm. A* is an ideal choice for this task as it is both complete and optimal, guaranteeing it will find the shortest path, provided its heuristic function is admissible (Hart et al., 1968). It operates by minimizing an evaluation function, $f(n)$, for each node n in the search space.

$$f(n) = g(n) + h(n) \tag{6.1}$$

Here, $g(n)$ is the cost of the path from the start node to node n, and $h(n)$ is the heuristic estimate of the cost to get from node n to the goal.

6.1.1 Heuristic function ($h(n)$)

Our planner uses the Manhattan distance as its heuristic function, $h(n)$. This is calculated as the sum of the absolute differences of the x and y coordinates between the current cell and the goal cell. The Manhattan distance is a classic choice for grid-based

worlds because it is computationally trivial and, critically, it is *admissible*—it never overestimates the true cost of reaching the goal since the agent cannot move diagonally. This admissibility is essential for guaranteeing the optimality of A*.

6.1.2 Risk-aware cost function ($g(n)$)

A key feature of our planner is its risk-averse nature, which is implemented directly within the path cost function, $g(n)$, through the `_get_action_cost` method. The cost of an action is not uniform; it depends on the agent's knowledge of the destination cell.

- **Turns:** Turning left or right has a standard cost of 1.
- **Moving to a SAFE cell:** Moving forward into a cell known to be `SAFE` also has a standard cost of 1.
- **Moving to a DANGEROUS cell:** Attempting to plan a path through a cell known to contain a `WUMPUS` or a `PIT` is assigned an infinite cost, effectively making such paths impassable and removing them from consideration.
- **Moving to an UNKNOWN cell:** This is where the risk assessment becomes explicit. Moving into an `UNKNOWN` cell incurs a base cost of 1 plus a significant penalty proportional to the estimated risk of that cell. The cost is calculated as: $1.0 + \texttt{RISK\_PENALTY} \times \text{risk\_estimate}$. The large `RISK_PENALTY` constant makes the A* algorithm heavily favor paths through known safe cells, even if they are much longer. It will only plan a path through an unknown cell if it is the only viable option.

6.1.3 Estimating cell risk

The risk estimate for an unknown cell is calculated by the `_estimate_cell_risk` method. It provides a simple heuristic for how dangerous a cell might be.

The risk score is computed by inspecting the percepts in adjacent, visited cells: for each neighboring cell that has a `stench`, the risk score increases by 0.5, and for each that has a `breeze`, it also increases by 0.5. This score is then normalized. This mechanism allows the planner to differentiate between a completely isolated unknown cell (low risk) and an unknown cell surrounded by stenches (high risk).

6.2 Exploration strategy: finding the next target

Pathfinding with A* is only useful if the agent has a destination in mind. The agent's exploration logic, found in the `_plan_exploration` method of the `HybridAgent`, determines this destination. When the agent does not have an immediate goal (like grabbing gold or climbing out), it follows a clear hierarchy to choose its next target.

1. **Explore safest option first:** The agent first scans its knowledge map for any cell that is both marked `SAFE` and has not yet been visited. If any exist, it targets the

one with the shortest Manhattan distance and plans a path to it. This ensures the agent always prioritizes guaranteed-safe exploration.
2. **Plan to move for shooting:** If no unvisited safe cells are left, but the agent knows the location of a Wumpus, it will plan to move to a safe cell that is aligned with the Wumpus for a future shot using the `_plan_align_wumpus()` method.
3. **Gamble with the least risky unknown cell:** Only when no safe exploration or tactical alignment options are available does the agent decide to take a calculated risk. It invokes the `planner.find_least_risky_unknown()` method to select the most promising unknown cell to visit.

The `find_least_risky_unknown` function itself has a tie-breaking procedure. It selects a target based on the following prioritized criteria:

1. **Lowest estimated risk:** It prefers the cell with the lowest risk score, as calculated by `_estimate_cell_risk`.
2. **Preference for unvisited:** If multiple cells share the same lowest risk, it prefers an `UNKNOWN` cell that has not been visited before over one that has (i.e., an `UNKNOWN` cell that was previously `SAFE` but was reset).
3. **Shortest distance:** If a tie still remains, it chooses the cell that is closest to the agent's current position, ensuring efficient exploration of nearby areas first.

This multi-layered planning and target-selection strategy creates an agent that is cautious yet also capable of taking calculated risks and shooting when necessary.

7 Experiment results

To quantitatively evaluate the performance of our hybrid agent, we conducted a series of experiments across a range of procedurally generated environments. The agent's performance was measured against a baseline random agent that selects actions uniformly from the set of legal moves. This chapter presents the results of these experiments and provides a detailed analysis of the findings.

7.1 Experiment setup

The evaluation was performed over a total of 250 unique Wumpus World environments. These environments were generated based on 5 distinct configurations, with 50 maps per configuration, shown in Table 7.1, to test the agents' adaptability to varying levels of difficulty. The key metrics recorded for each trial were:

- **Success rate:** The percentage of trials where the agent successfully retrieved the gold and climbed out of the cave.
- **Average score:** The mean score achieved across all trials, factoring in rewards for success and penalties for actions and death.
- **Decision efficiency:** The average number of actions (steps) taken per trial. A lower number for a successful run indicates higher efficiency.
- **Computation time:** The average time taken for the agent to complete a trial.

Table 7.1: *Map configurations used for generating test environments.*

Grid Size	Wumpus Count	Pit Probability	Moving Wumpus?
8 x 8	1	0.10	No
8 x 8	2	0.20	No
8 x 8	2	0.10	Yes
10 x 10	2	0.10	No
10 x 10	1	0.05	Yes

7.2 Hardware specifications

We use the hardware specifications specified in Table 7.2 to conduct our experiments.

Table 7.2: *Hardware specifications*

Component	Specification
Model	ASUS ROG Strix G16 G614JV-N4455W
Processor	Intel® Core™ i7-13650HX (up to 4.90 GHz, 24MB Cache)
Memory	16GB DDR5-4800 SO-DIMM (2 slots, expandable to 32GB)
Storage	512GB PCIe 4.0 NVMe M.2 SSD
Graphics	NVIDIA® GeForce RTX 4060 8GB GDDR6
Operating system	Windows 11

7.3 Performance analysis

7.3.1 Overall performance

First, we consider the overall performance of both agents across all 250 trials. The summary provides a high-level overview of the hybrid agent's effectiveness compared to the baseline.

Table 7.3: *Overall agent performance summary across 250 trials.*

Metric	Hybrid Agent	Random Agent
Success rate (%)	55.60	2.00
Average score	70.83	-278.87
Average steps	51.43	24.60
Average time (ms)	10664.62	0.33

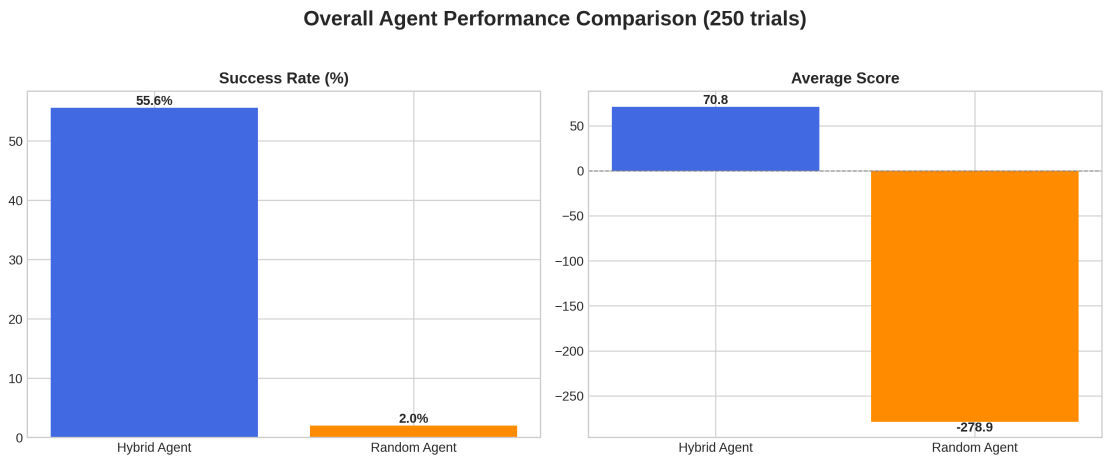

Figure 7.1: *Overall comparison of success rate and average score for the hybrid and random agents.*

7.3.2 Performance by map characteristics

To understand how each agent responds to different challenges, we analyzed performance by isolating each environmental parameter: the number of wumpuses, the probability of pits, and the presence of a moving wumpus.

Impact of wumpus count

Increasing the number of wumpuses from one to two introduces more uncertainty and potential threats.

Table 7.4: *Performance variation based on the number of wumpuses.*

NumWumpus	Hybrid Agent		Random Agent	
	Success (%)	Avg Score	Success (%)	Avg Score
1	70.0	344.79	2.0	-256.23
2	46.0	-111.81	2.0	-293.96

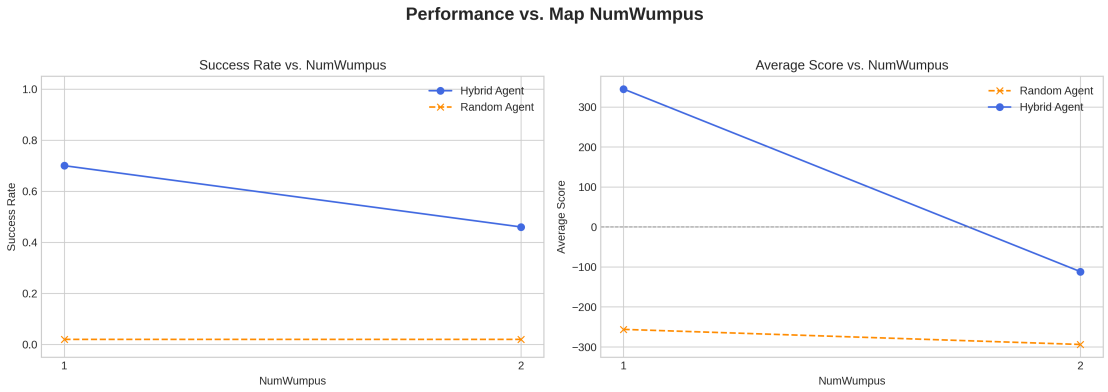

Figure 7.2: *Effect of increasing wumpus count on agent performance.*

Impact of pit probability

A higher pit probability increases the density of static hazards, making safe navigation more difficult.

Table 7.5: *Performance variation based on pit probability.*

PitProb	Hybrid Agent		Random Agent	
	Success (%)	Avg Score	Success (%)	Avg Score
0.05	62.0	186.74	0.0	-239.56
0.10	58.0	110.82	2.0	-288.65
0.20	42.0	-165.04	4.0	-288.82

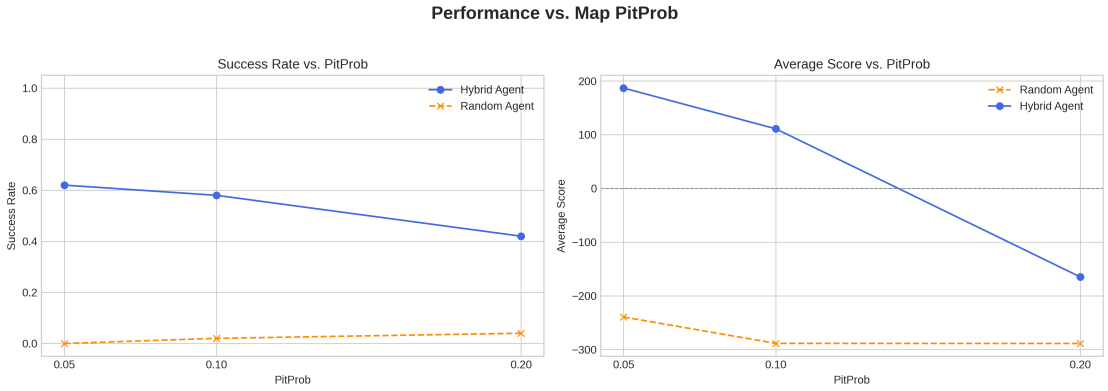

Figure 7.3: *Effect of increasing pit probability on agent performance.*

Impact of moving wumpus

The moving wumpus setting introduces a dynamic environment where the agent's knowledge can become outdated, representing a significant increase in complexity.

Table 7.6: *Performance in static vs. dynamic (moving wumpus) environments.*

Environment	Hybrid Agent		Random Agent	
	Success (%)	Avg Score	Success (%)	Avg Score
Static	64.0	236.47	2.7	-286.17
Dynamic	43.0	-177.62	1.0	-267.92

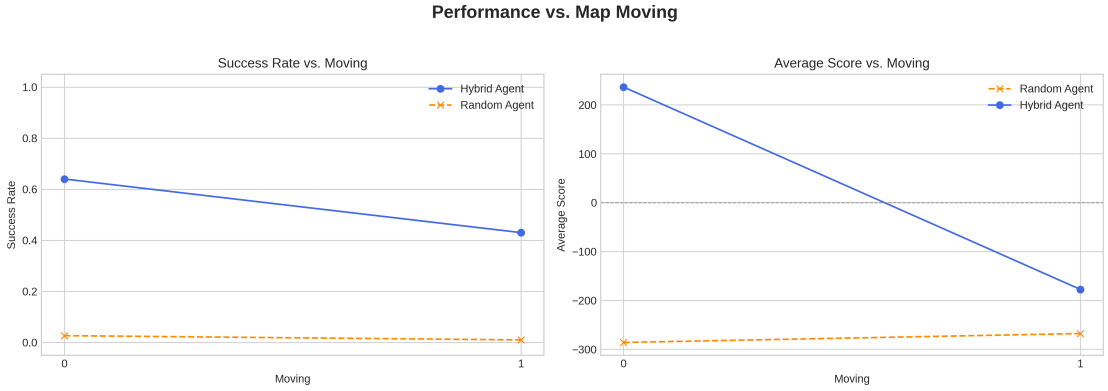

Figure 7.4: *Effect of a dynamic environment on agent performance.*

7.3.3 Distribution of outcomes

Beyond averages, the distribution of scores and steps reveals the consistency of each agent. Box plots are used to visualize the median, quartiles, and range of outcomes.

Table 7.7: *Descriptive statistics for final score and steps taken (for successful runs).*

Statistic	Final Score (All Trials)		Steps (Successful Trials)	
	Hybrid	Random	Hybrid	Random
Count	250	250	139	5
Mean	70.83	-278.87	69.88	2.00
Std Dev	979.04	502.64	53.08	0.00
Min	-1231.00	-1316.00	2.00	2.00
25% (Q1)	-1013.00	-1001.00	27.50	2.00
50% (Median)	878.00	-11.00	56.00	2.00
75% (Q3)	960.00	0.00	99.50	2.00
Max	1010.00	1010.00	277.00	2.00

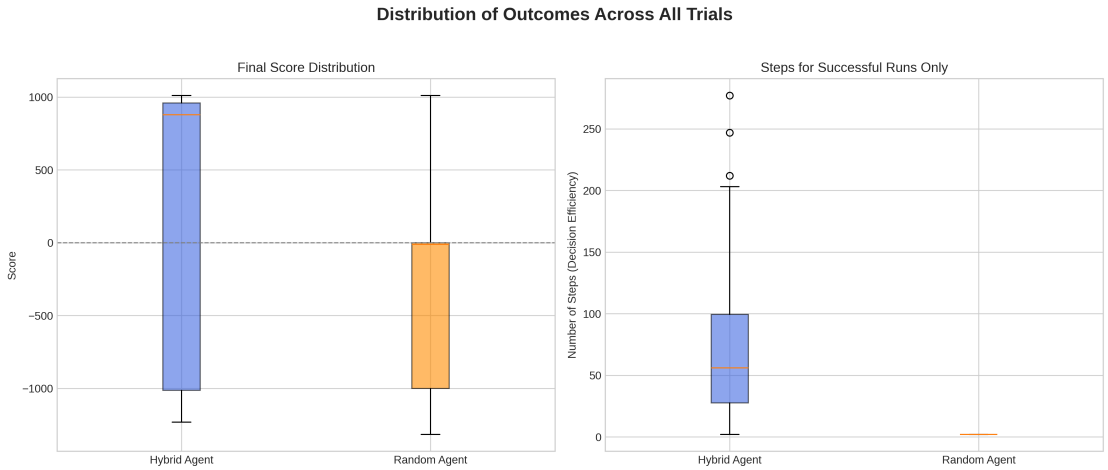

Figure 7.5: *Box plot comparing the distribution of final scores (left) and steps for successful runs (right) for both agents.*

7.3.4 Score vs. steps correlation

Finally, we analyze the relationship between the number of steps an agent takes and its final score. This helps clarify the connection between effort and outcome for each agent's strategy.

Table 7.8: *Correlation between steps and final score.*

Agent	Score vs. Steps Correlation
Hybrid Agent	0.3655
Random Agent	-0.4800

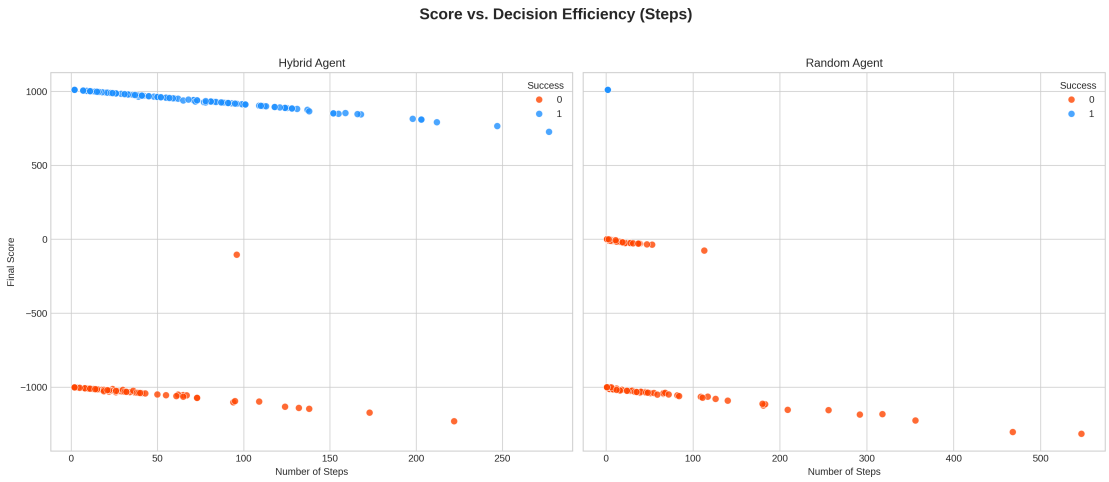

Figure 7.6: *Relationship between the number of steps taken and the final score for successful (blue) and unsuccessful (red) trials.*

7.4 Discussion and insights

The experimental results clearly show that the knowledge-based hybrid agent performed much better than the random agent. The data shows that the hybrid agent is more effective and also helps us understand how it makes decisions and handles difficult situations.

7.4.1 Hybrid agent effectiveness and reliability

As shown in Table 7.3 and Figure 7.1, the hybrid agent had a success rate of 55.6%, which is far better than the random agent's 2.0%. This resulted in a positive average score of 70.83, while the random agent's score was -278.87.

Looking at how the scores are spread out in Figure 7.5 tells us even more. The hybrid agent's median score is 878, which means it won in over half of all trials. Its failures lead to big score penalties (as shown by the 25th percentile of -1013), but the high median shows that failing is not what usually happens. In contrast, the random agent's median score is -11, and its 75th percentile is 0.0. This means that in at least 75% of all trials, the random agent failed, which shows its wins are just lucky chances on very easy maps.

7.4.2 Handling increasing difficulty

The line charts in Figures 7.2, 7.3, and 7.4 show how the agents perform as the maps get harder. The hybrid agent's performance drops in a predictable way: its success rate and average score go down as more pits and wumpuses are added, but it still performs well. For instance, its success rate falls from 70% with one wumpus to a solid 46% with two.

Most importantly, the agent maintains a 43% success rate even when the wumpus is moving. This shows that its strategy of resetting its knowledge about the wumpus

location every five steps is effective. This stops the agent from getting stuck because of conflicting or old information. The random agent's performance is already very poor and does not really change much as the maps get harder.

7.4.3 Efficiency and problem-solving capability

The box plot in Figure 7.5 (right) looks at how many steps were taken in successful runs. The hybrid agent's successful runs took a median of 56 steps, but sometimes took as many as 277 steps on harder maps. This shows the agent can actually solve problems, since it can find its way through difficult maps that require many steps.

On the other hand, the random agent's 5 wins were all completed in exactly 2 steps. This shows it did not solve any real problems; it only won when a few lucky moves were all that was needed. The average step count for the random agent in Table 7.3 can be misleading, as the low average is simply because it usually dies very quickly.

The link between the number of steps and the final score in Figure 7.6 is also interesting. For the hybrid agent, there is a positive relationship. This might seem strange, because more steps should lower the score. However, it is because winning (even after many steps) results in a high positive score, while failing results in a large negative score. The big difference between winning and losing makes it clear that taking more steps to win is better than failing early. For the random agent, the relationship is negative, which is simpler: the more it moves, the more likely it is to die and get a lower score.

Bibliography

Biere, A. (2009). *Handbook of Satisfiability*. Frontiers in artificial intelligence and applications. IOS Press. ISBN: 9781586039295. URL: https://books.google.com.vn/books?id=YVSM3sxhBhcC.

Davis, Martin, George Logemann, and Donald Loveland (July 1962). "A machine program for theorem-proving". In: *Commun. ACM* 5.7, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

Google LLC (2025). *Google Sheets*. Online spreadsheet application for collaborative project management. URL: https://sheets.google.com.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

José Areia (Dec. 2023). *Polytechnic University of Leiria: LaTeX Thesis Template*. URL: https://github.com/joseareia/ipleiria-thesis (visited on 2025-06-18).

Minh Nguyen Tran Duy (2025). *Project 2. Wumpus World Agent*. Introduction to Artificial Intelligence CS14003. Ho Chi Minh City, Vietnam.

Russell, S.J. and P. Norvig (2021). *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson. ISBN: 9781292401133. URL: https://books.google.com.vn/books?id=B4xczgEACAAJ.

A

Demo video YouTube URLs and other materials

YouTube video URLs

Our demonstration video is available on YouTube at:

https://www.youtube.com/watch?v=jhLWgMBamOg

Drive folder for other materials

Other materials (such as raw CSV metrics, chart images, etc.) can be found at: Drive link.

Report template and stock images for GUI

This LaTeX report template is provided free-of-charge by José Areia (2023). Images used to design the GUI:

- Agent and Wumpus sprites: Wraith Skelleton King from dota 2 pixel style.
- Tileset (background cell, breeze, pit): LPC Tile Atlas.
- Arrows: Rotating Arrow Projectile.
- Bat sprite (for stench): Simple Grey Bat.
- Buttons: Pixel video game graphic elements ui buttons arrows menu icons in 8bit retro style.

B

AI Acknowledgement

We acknowledge the use of AI-powered tools to assist in aspects of this project. These tools were primarily utilized to refine the academic tone and improve the linguistic accuracy of this report, including aspects of grammar, punctuation, and technical vocabulary. We emphasize that all core concepts, algorithm design, experimental methodology, and analytical insights presented in this work are the product of our own understanding and effort. The AI tools served as supplementary aids to enhance the quality and presentation of our work, while the intellectual contributions and problem-solving approaches remain our own.

AI usage

LaTeX content and formatting

- **Claude Opus 4**, Anthropic, claude.ai, accessed: 10h 2025-08-01. Prompted: *"Format this propositional logic rule in LaTeX: Bx,y if and only if (Pneighbor1 or Pneighbor2)"*, this helped us correctly format the biconditional axioms in our knowledge base formulation chapter. We verified the generated equations and integrated them into our self-authored text.
- **ChatGPT-4o**, OpenAI, chatgpt.com, accessed: 19h 2025-08-04. Prompted: *"format this data into LaTeX table, follow this example code"*, this assisted us in generating the multi-column tables for our experiment results chapter. We verified the table structure and syntax; the data presented in the table was generated from our own experiments.
- **GPT-5**, OpenAI, chatgpt.com, accessed: 17h 2025-08-08. Prompted: *"Review this paragraph about our A* risk cost function for clarity and academic tone"*, this assisted in refining the language in our planning chapter to be more formal and precise. We reviewed the suggestions and incorporated them to improve readability.
- **Claude Opus 4**, Anthropic, claude.ai, accessed: 21h 2025-08-11. Prompted: *"Fix the LaTeX syntax, getting 'missing $' errors with symbols P_1_2"*, this helped us de-

bug our report by correctly escaping special characters and placing logical symbols in math mode. We verified the corrections and the document's ability to compile.

Data processing and charts

- **ChatGPT-4o**, OpenAI, chatgpt.com, accessed: 14h 2025-08-12. Prompted: *"write a python script with matplotlib to generate line charts and box plots comparing two agents from this csv file"*, this provided the visualization script. We customized the plotting styling, and added logic to save the charts as image files for the report.
- **Gemini 2.5 Pro**, Google, gemini.google.com, accessed: 23h 2025-08-12. Prompted: *"Based on these box plot results, what insights can you draw about the performance of the hybrid vs random agent?"*, this provided an initial comment on our charts, which we used as a starting point for writing our analysis and discussion section. The final insights were written and reviewed by our group.
- **Claude Sonnet 4**, Anthropic, claude.ai, accessed: 12h 2025-08-10. Prompted: *"write a python script to read map configurations from a json, create multiple environments for each config, run both a hybrid and random agent, and save the results (score, steps, time, success) to a CSV file"*, this provided the main structure for our `test_map.py` script. We implemented the agent execution logic and customized the final data output format.

Codebase

- **o4-mini**, OpenAI, chatgpt.com, accessed: 11h 2025-08-01. Prompted: *"Suggest a python project structure for a wumpus world agent with modules for environment, knowledge, inference, and planning modules"*, this gave us an initial high-level structure for our project. We adapted and modified this structure to fit our specific class designs and implementation details.
- **Gemini 2.5 Pro**, Google, gemini.google.com, accessed: 15h 2025-08-03. Prompted: *"explain and analyze the heuristics in the DPLL algorithm and provide an example python snippet, following AIMA textbook"*, this helped clarify our understanding of one of the core heuristics in our inference engine. The final implementation was written and integrated by our group based on our understanding of the algorithm.
- **GitHub Copilot, Gemini 2.5 Pro**, Gemini, Visual Studio Code, accessed: 19h 2025-08-10. Prompted: *"Check my logic when resetting the KB for a moving wumpus, it seems wrong, by reviewing this python function `reset_wumpus_knowledge` and suggest potential issues"*, this helped us debug and refine the logic for handling the dynamic environment by pointing out bugs with reset. We tested the suggestions and fixed our code.
- **ChatGPT-4o**, OpenAI, chatgpt.com, accessed: 7h 2025-08-03. Prompted: *"I need to build a Pygame GUI for my Wumpus World project, suggest a code structure to handle drawing the grid and displaying the agent's state and knowledge"*, this provided us

with a high-level architectural suggestion, such as using a main controller class and separate view classes. The actual Pygame implementation, event handling, and visualization logic later on were developed by us.