Technical University of Cluj-Napoca
Department of Computer Science

Artificial Intelligence

Laboratory Activity

Name:
Stefania-Cristina MOZACU

Group:
30434

Email:
Mozacu.Cl.Stefania@student.utcluj.ro

Teaching Assistant:
Marius STOICA
Stoica.Ma.Marius@student.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search [1, 2]

1.1 Breadth-First Seach (BFS) [4]

Breadth-First Search (BFS) is a search strategy that explores the search tree level by level. It always expands the shallowest unexpanded node first.

1.1.1 Properties

- Uses a **queue (FIFO)** as the frontier.
- Guarantees finding the shortest path.
- Time complexity: $O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal.
- Space complexity: also $O(b^d)$, since all nodes at the current frontier must be stored.

1.1.2 Pseudocode

Algorithm 1: Breadth-First Search

```
1  frontier ← queue with (startState, emptyPath);
2  visited ← empty set;
3  while frontier not empty do
4  |  (state, path) ← frontier.pop();
5  |  if isGoal(state) then
6  |  |  return path;
7  |  end
8  |  if state not in visited then
9  |  |  add state to visited;
10 |  |  foreach (successor, action, stepCost) in getSuccessors(state) do
11 |  |  |  if successor not in visited then
12 |  |  |  |  frontier.push((successor, path + [action]));
13 |  |  |  end
14 |  |  end
15 |  end
16 end
```

1.1.3 Step-by-Step Execution

Grid: 4×4, Start: (0,3), Goal: (3,0), Walls: (1,2) and (2,1)
Coordinates: (column, row) where (0,0) is bottom-left

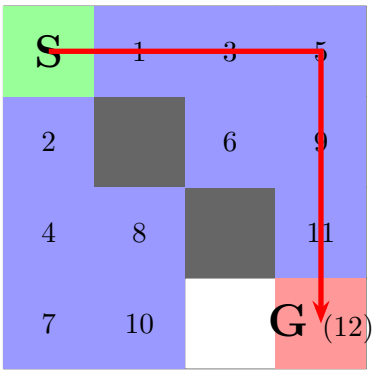

Figure 1.1: BFS exploration order (numbers) and shortest path from Start to Goal.

Queue and exploration order:

1. **Step 0-1:** Queue = [(0,3)] → Explore S(0,3), distance 0

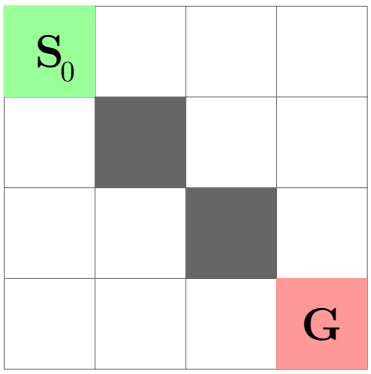

Figure 1.2: Step 0–1: Start expanded at (0,3).

2. **Step 2-3:** Queue = [(1,3), (0,2)] → Explore neighbors of S

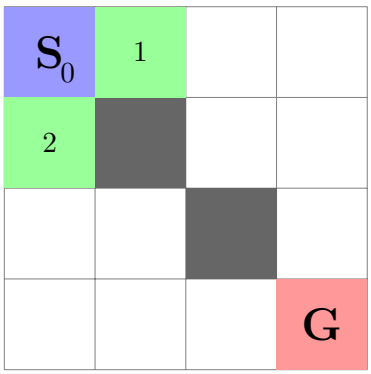

Figure 1.3: Step 2–3: Add Right(1,3), Down(0,2).

3. **Step 4-6:** Queue = [(2,3), (1,2), (0,1)] → Distance 2

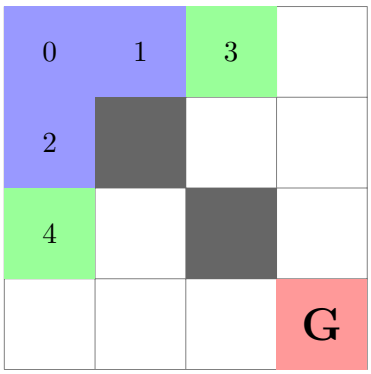

Figure 1.4: Step 4–6: Neighbors of (1,3) and (0,2).

4. **Step 7-10:** Queue = [(3,3), (2,2), (1,1), (0,0)] → Distance 3

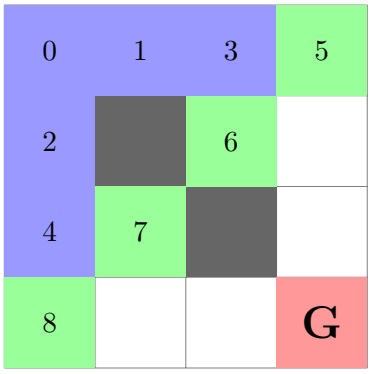

Figure 1.5: Step 7–10: Expanding wavefront, distance 3.

5. **Step 11-13:** Queue = [(3,2), (2,1), (1,0)] → Distance 4

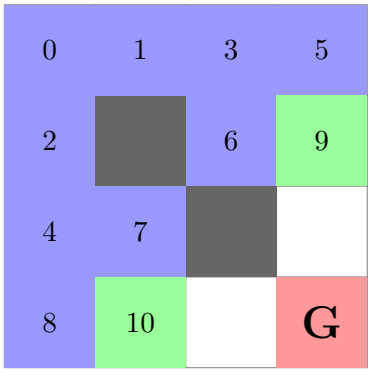

Figure 1.6: Step 11–13: Expanding at distance 4.

6. **Step 14-15:** Queue = [(3,1), (2,0)] → Distance 5

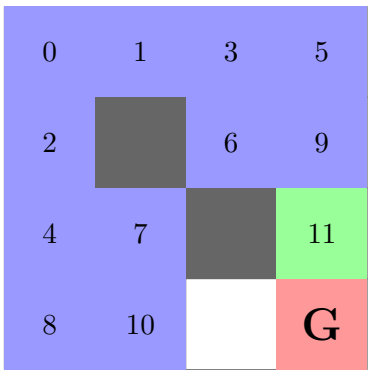

Figure 1.7: Step 14–15: Distance 5 reached, preparing goal.

7. **Step 16:** Queue = [(3,0)] → **GOAL FOUND!** Distance 6

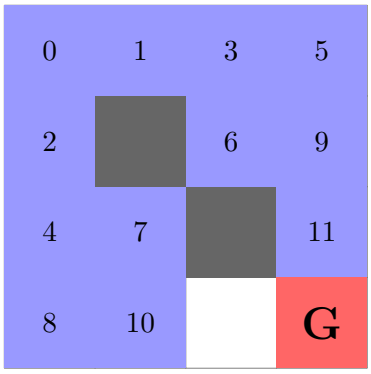

Figure 1.8: Step 16: Goal reached at (3,0). Distance = 6 moves.

1.2 Depth-First Search (DFS) [4]

Depth-First Search (DFS) is a search strategy that explores a path as deep as possible before backtracking. It uses a stack (LIFO) to store the frontier. Unlike BFS, it does not guarantee the shortest path, but it often reaches a solution faster in terms of node expansions.

1.2.1 Properties

- Uses a **stack (LIFO)** as the frontier.
- Does **not guarantee the shortest path**.
- Time complexity: $O(b^m)$, where m is the maximum depth of the search tree.
- Space complexity: $O(bm)$ (linear in the depth), more memory-efficient than BFS.

1.2.2 Pseudocode

Algorithm 2: Depth-First Search

```
frontier ← stack with (startState, emptyPath);
visited ← empty set;
while frontier not empty do
    (state, path) ← frontier.pop();
    if isGoal(state) then
        return path;
    end
    if state not in visited then
        add state to visited;
        foreach (successor, action, stepCost) in getSuccessors(state) do
            if successor not in visited then
                frontier.push((successor, path + [action]));
            end
        end
    end
end
```

1.2.3 Step-by-Step Execution

We use the same grid: 4×4, Start: (0,3), Goal: (3,0), Walls at (1,2) and (2,1). Coordinates: (column, row) with (0,0) bottom-left.

Stack and exploration order:

1. **Step 0:** Stack = [(0,3)] → Start node pushed.

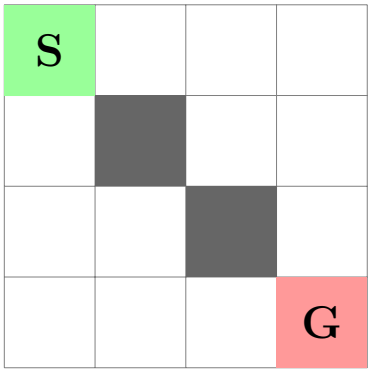

Figure 1.9: Step 0: Initial stack = [(0,3)].

2. **Step 1:** Pop (0,3). Push successors (0,2), (1,3). Stack = [(0,2), (1,3)].

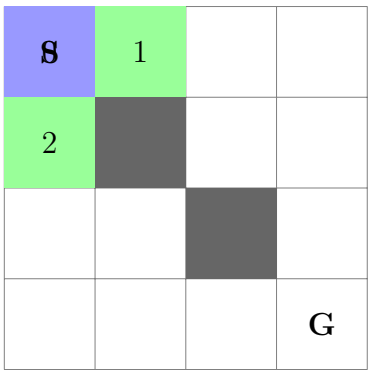

Figure 1.10: Step 1: Expand (0,3). Push (0,2), (1,3).

3. **Step 2:** Pop (1,3). Push successor (2,3). Stack = [(0,2), (2,3)].

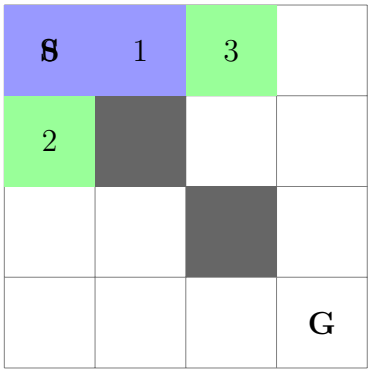

Figure 1.11: Step 2: Pop (1,3). Push (2,3).

4. **Step 3:** Pop (2,3). Push successors (3,3), (2,2). Stack = [(0,2), (3,3), (2,2)].

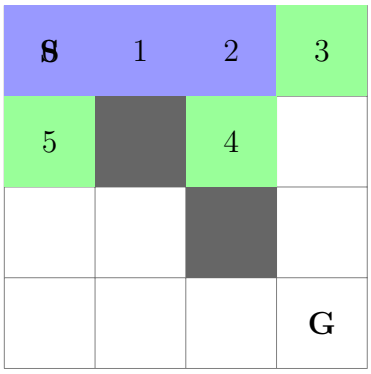

Figure 1.12: Step 3: Pop (2,3). Push (3,3), (2,2).

5. **Step 4:** Pop (2,2). Blocked at (2,1), no new successors. Stack = [(0,2), (3,3)].

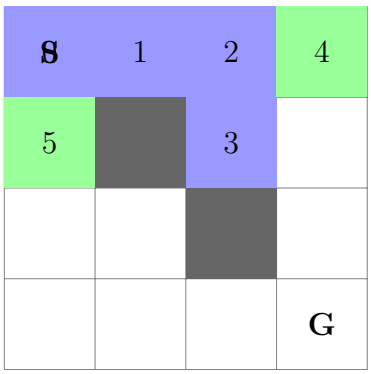

Figure 1.13: Step 4: Pop (2,2). Wall below, no successors.

6. **Step 5:** Pop (3,3). Push successor (3,2). Stack = [(0,2), (3,2)].

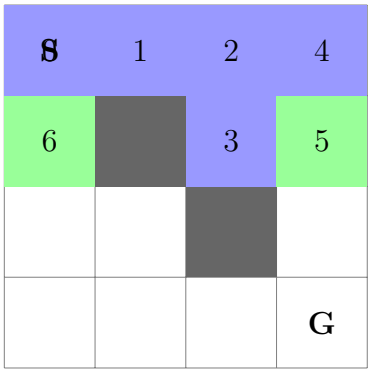

Figure 1.14: Step 5: Pop (3,3). Push (3,2).

7. **Step 6:** Pop (3,2). Push successor (3,1). Stack = [(0,2), (3,1)].

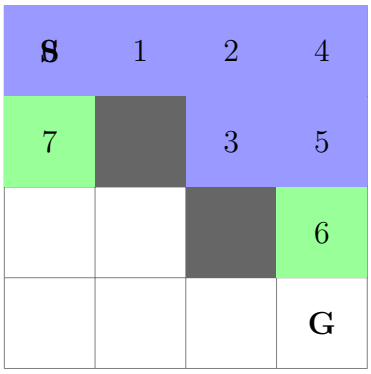

Figure 1.15: Step 6: Pop (3,2). Push (3,1).

8. **Step 7:** Pop (3,1). Push successor (3,0). Stack = [(0,2), (3,0)].

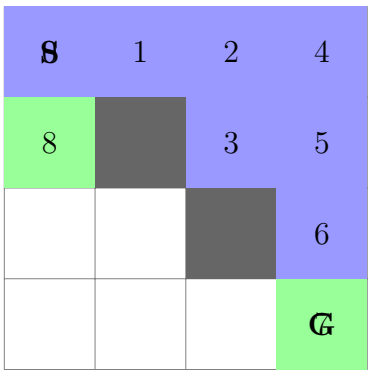

Figure 1.16: Step 7: Pop (3,1). Push (3,0).

9. **Step 8:** Pop (3,0). **GOAL FOUND!**

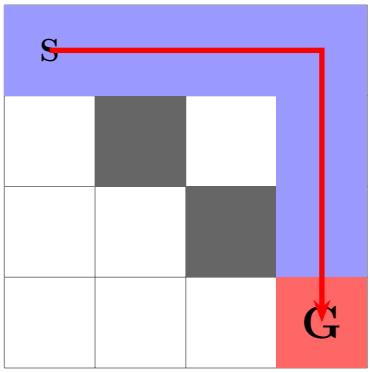

Figure 1.17: Step 8: Goal found at (3,0) via DFS path.

1.3 Uniform Cost Search (UCS)[6]

Uniform Cost Search (UCS) expands the node with the lowest path cost so far. It uses a **priority queue** (min-heap) instead of a simple queue, so the nodes are explored in order to increase the cumulative cost. If all edge costs are equal, UCS behaves identically to BFS.

1.3.1 Properties

- Uses a **priority queue** ordered by path cost $g(n)$.
- Guaranteed to find the least-cost solution (optimal).

1.3.2 Pseudocode

Algorithm 3: Uniform Cost Search

```
frontier ← priority queue with (startState, emptyPath), cost=0;
bestCost[startState] = 0;
while frontier not empty do
    (state, path), cost ← frontier.pop();
    if isGoal(state) then
        return path;
    end
    if cost ¿ bestCost[state] then
        continue;
    end
    foreach (successor, action, stepCost) in getSuccessors(state) do
        newCost = cost + stepCost;
        if successor not in bestCost or newCost ¡ bestCost[successor] then
            bestCost[successor] = newCost;
            frontier.push((successor, path + [action]), newCost);
        end
    end
end
```

1.3.3 Cost Function $g(n)$ in UCS

In Uniform Cost Search, each node n is associated with a **cumulative path cost**, denoted $g(n)$. This represents the sum of the step costs along the path from the start state to n. Formally:

$$g(n) = \sum_{i=0}^{k-1} cost(s_i, a_i, s_{i+1})$$

where $(s_0, s_1, \ldots, s_k = n)$ is the path and a_i are the actions taken.

Properties of $g(n)$:

- Always non-decreasing as the search progresses.
- If all step costs are equal (e.g., $cost = 1$), then $g(n)$ is equivalent to the depth of node n.
- UCS expands nodes in order of increasing $g(n)$, ensuring optimality.

Example on 4×4 Grid

Consider the same grid with Start $S = (0, 3)$ and Goal $G = (3, 0)$, walls at $(1, 2)$ and $(2, 1)$.

1. At $S = (0, 3)$, $g(S) = 0$.
2. From S to $(1, 3)$ or $(0, 2)$, $g = 1$ since each move costs 1.
3. From $(1, 3)$ to $(2, 3)$, $g = 2$. From $(0, 2)$ to $(0, 1)$, $g = 2$.
4. From $(2, 3)$ to $(3, 3)$, $g = 3$. From $(2, 3)$ to $(2, 2)$, $g = 3$. From $(0, 1)$ to $(0, 0)$, $g = 3$.
5. Continuing, the goal is reached at $(3, 0)$ with $g = 6$.

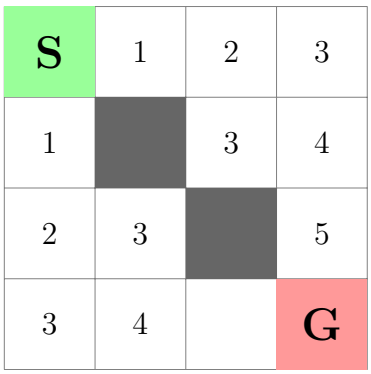

Figure 1.18: Cumulative path costs $g(n)$ across the grid during UCS exploration.

Observation: Since all moves cost 1, the cumulative costs grow exactly as BFS levels. If some moves had higher costs (e.g., moving right = 2, down = 1), UCS would deviate from BFS and always prefer the cheapest accumulated path.

1.3.4 Uniform Cost Search Example with Directional Costs

Grid and costs: Grid 4×4, Start $S = (0, 3)$, Goal $G = (3, 0)$, walls at $(1, 2)$ and $(2, 1)$. We define the step cost as:

- Moving **East** (to the right): cost 2
- Moving **West** (to the left): cost 2
- Moving **North** (up): cost 1
- Moving **South** (down): cost 1

Thus every edge has a direction-dependent cost. UCS always expands the state with the *lowest accumulated cost* $g(n)$.

Step Trace

1. Start $S = (0, 3)$, $g = 0$.
2. Neighbors: $(1, 3)$ with $g = 2$, $(0, 2)$ with $g = 1$.
3. UCS expands $(0, 2)$ first (cheapest). From there, $(0, 1)$ with $g = 2$.
4. Frontier now has $(0, 1)\, g = 2$ and $(1, 3)\, g = 2$. Tie $\rightarrow$ arbitrary pick.
5. Expanding downwards continues cheaply: $(0, 0)\, g = 3$.
6. But reaching goal via right moves is more expensive because each East adds +2.
7. Optimal UCS path is therefore:

$$S(0, 3) \rightarrow (0, 2) \rightarrow (0, 1) \rightarrow (0, 0) \rightarrow (1, 0) \rightarrow (2, 0) \rightarrow (3, 0)$$

with total cost:

$$g = 1 + 1 + 1 + 2 + 2 + 2 = \mathbf{9}.$$

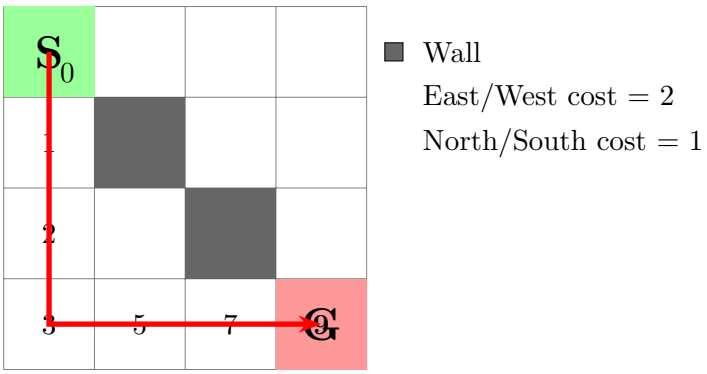

Figure 1.19: UCS expands by cumulative cost. The cheapest path goes straight down then right, total $g = 9$.

Comparison with BFS: In this grid, BFS would always expand nodes level by level, which means it would prefer the path going right across the top row:

$$S(0,3) \to (1,3) \to (2,3) \to (3,3) \to (3,2) \to (3,1) \to G(3,0).$$

This path has *6 moves* (shortest length), so BFS considers it optimal. However, since moving East has cost 2, UCS assigns the following cost:

$$g = 2 + 2 + 2 + 1 + 1 + 1 = 9.$$

In contrast, UCS discovers that going *down first* is cheaper:

$$S(0,3) \to (0,2) \to (0,1) \to (0,0) \to (1,0) \to (2,0) \to G(3,0),$$

with cost

$$g = 1 + 1 + 1 + 2 + 2 + 2 = 9.$$

Thus:

- BFS minimizes the *number of steps*.
- UCS minimizes the *total path cost*, considering edge weights.

Observation: BFS is a special case of UCS where all step costs are equal. When costs differ, UCS may explore longer routes (in steps) that are cheaper overall in terms of path cost.

1.4 A* Search[7, 3]

A* search expands nodes according to the function

$$f(n) = g(n) + h(n),$$

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

1.4.1 Properties

- Uses a **priority queue** ordered by $f(n) = g(n) + h(n)$.
- Guaranteed to find the optimal path if the heuristic is **admissible** (never overestimates) and **consistent**.
- Generalizes UCS: when $h(n) = 0$, A* is equivalent to UCS.

1.4.2 Pseudocode

Algorithm 4: A* Search

```
frontier ← priority queue with (startState, emptyPath), priority = h(start);
bestCost[start] = 0;
while frontier not empty do
    (state, path), cost ← frontier.pop();
    if isGoal(state) then
        return path;
    end
    if cost ¿ bestCost[state] then
        continue;
    end
    foreach (successor, action, stepCost) in getSuccessors(state) do
        newCost = cost + stepCost;
        if successor not in bestCost or newCost ¡ bestCost[successor] then
            bestCost[successor] = newCost;
            priority = newCost + h(successor);
            frontier.push((successor, path + [action]), priority);
        end
    end
end
```

1.4.3 Manhattan Distance Heuristic

The **Manhattan distance** (also called L_1 norm) is defined as:

$$h(x, y) = |x - x_G| + |y - y_G|,$$

where (x, y) is the current node and (x_G, y_G) is the goal position. It measures the number of steps required to reach the goal if only *horizontal and vertical* moves are allowed (as in a grid maze).

S 6	5	4	3
5	4	3	2
4	3	2	1
3	2	1	**G**

Figure 1.20: Manhattan heuristic values for each cell in the 4×4 grid, with Start at (0,3) and Goal at (3,0).

Properties of Manhattan heuristic:

- **Admissible**: it never overestimates, since the true cost can only be greater or equal (walls may force detours).

- **Consistent**: moving one step changes the heuristic by at most 1, matching the actual cost.
- **Efficient**: easy to compute with a few subtractions and absolute values.

Example: In the 4×4 grid, Start $S = (0, 3)$ and Goal $G = (3, 0)$:

$$h(S) = |0 - 3| + |3 - 0| = 6.$$

If Pacman moves Right to $(1, 3)$:

$$h(1, 3) = |1 - 3| + |3 - 0| = 5.$$

Each step reduces the heuristic by 1 until the goal is reached.

1.4.4 Comparison with UCS and BFS

- BFS finds the shortest path in terms of steps.
- UCS finds the least-cost path when edges have weights.
- A* improves UCS by directing the search toward the goal using $h(n)$.
- With Manhattan distance on a grid, A* typically expands far fewer nodes than UCS.

1.4.5 Step-by-Step Execution

Step 0: Start at $(0, 3)$ with $f = 6$. Frontier = { $(1, 3), f = 6$; $(0, 2), f = 6$ }.

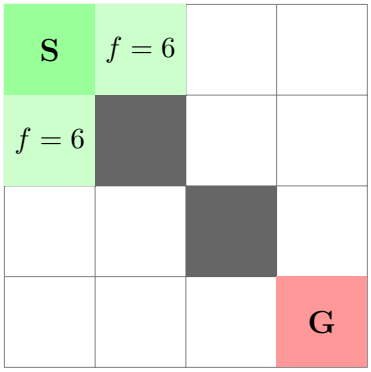

Figure 1.21: Step 0: Start expands, add two successors.

Step 1: Expand $(1, 3)$. Add $(2, 3)$ with $f = 6$. Frontier = { $(0, 2), f = 6$; $(2, 3), f = 6$ }.

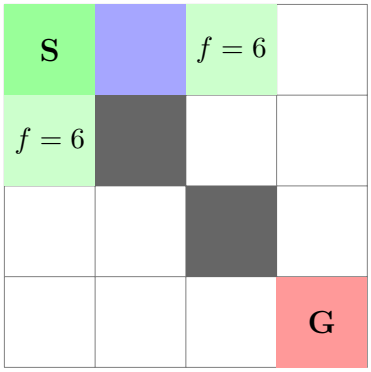

Figure 1.22: Step 1: Expand (1,3), push (2,3).

Step 2: Expand $(2, 3)$. Add $(3, 3)$ with $f = 6$. Frontier = { $(0, 2), f = 6$; $(3, 3), f = 6$ }.

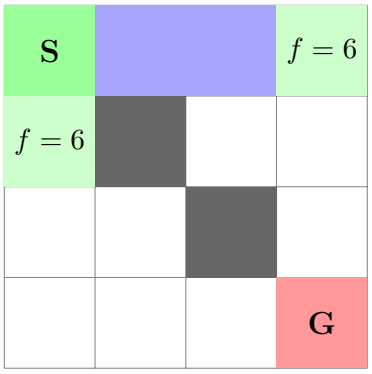

Figure 1.23: Step 2: Expand (2,3), push (3,3).

Step 3: Expand $(3, 3)$. Add $(3, 2)$ with $f = 6$. Frontier = { $(0, 2), f = 6$; $(3, 2), f = 6$ }.

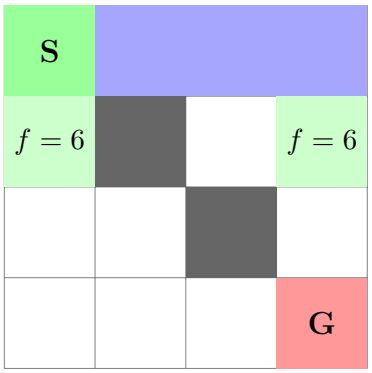

Figure 1.24: Step 3: Expand (3,3), push (3,2).

Step 4: Expand $(3, 2)$ then $(3, 1)$, finally reach $(3, 0)$ with $f = 6$. The goal is found in only a few expansions, guided straight down the right column.

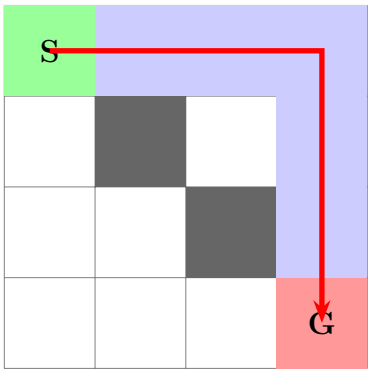

Figure 1.25: Step 4 (Final): Goal reached at (3,0). Path length = 6.

1.5 Minimax Search[5, 8]

Minimax is an **adversarial search algorithm** used in two-player (or multi-agent) games. Pacman is modeled as the **MAX player** (agent index 0), while each ghost acts as a **MIN player** (agent indices 1, 2, ...). The algorithm assumes that:

- Pacman chooses actions to *maximize* the evaluation score.
- Ghosts choose actions to *minimize* Pacman's outcome.
- Both sides play optimally at every step.

1.5.1 Pseudocode

Algorithm 5: Minimax with multiple agents (Pacman and ghosts)

```
Function MiniMax (state, agentIndex, depth):
    if depth = self.depth or state.isWin() or state.isLose() then
        return evaluationFunction(state)
    end
    nAgents ← state.getNumAgents()
    if agentIndex = 0                                          // Pacman = MAX
    then
        value ← −∞
        for a ∈ state.getLegalActions(0) do
            succ ← state.generateSuccessor(0, a)
            child ← MiniMax(succ, 1, depth)
            value ← max(value, child)
        end
        return value
    end
    else
                                                               // Ghost = MIN
        nextAgent ← agentIndex + 1
        nextDepth ← depth
        if nextAgent = nAgents then
            nextAgent ← 0; nextDepth ← depth + 1
        end
        value ← +∞
        for a ∈ state.getLegalActions(agentIndex) do
            succ ← state.generateSuccessor(agentIndex, a)
            child ← MiniMax(succ, nextAgent, nextDepth)
            value ← min(value, child)
        end
        return value
    end
```

1.5.2 How it Works in Pacman

Turn order: Pacman moves first, then each ghost in order, looping back to Pacman after the last ghost. The depth increases only when Pacman acts again.

Food: Eating food pellets increases the game score. The evaluation function rewards Pacman for reaching states with less food or closer food positions.

Ghosts: Ghosts act as minimizers:

- If ghosts are not scared, they try to move closer to Pacman to reduce Pacman's utility.
- If they are scared (after Pacman eats a power capsule), Pacman may seek them out for bonus points.

Terminal states: If Pacman wins (all food eaten or ghosts eaten) or loses (caught by a ghost), the recursion returns immediately with that state's evaluation.

1.5.3 Step-by-Step Example (Tiny Game Tree)

We illustrate with Pacman at the root choosing between two actions, and a single ghost minimizing afterwards.

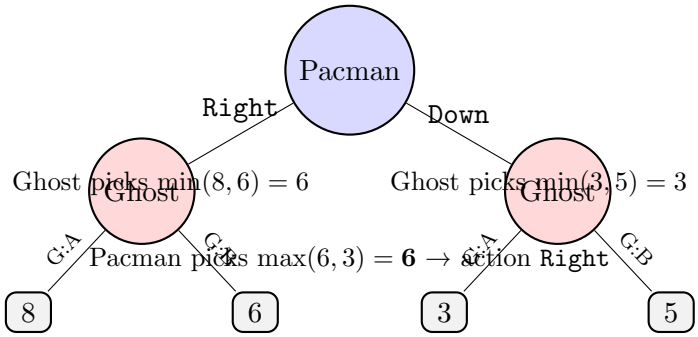

Figure 1.26: Minimax backup in a toy game: Pacman maximizes, Ghost minimizes.

1.5.4 Observations

- Pacman looks ahead at how ghosts might react and avoids paths that lead to poor scores.
- The algorithm guarantees optimal play if both Pacman and the ghosts act rationally.

Chapter 2

A2: Logics

2.1 Prover9 and Mace4

Prover9 and Mace4 are two automated reasoning tools designed for First-Order Logic (FOL), each serving a complementary purpose in logical analysis and verification.

2.1.1 Prover9: Automated Theorem Proving

Prover9 is an automated theorem prover for first-order and equational logic. Given a set of axioms (assumptions) and a goal formula, Prover9 attempts to determine whether the goal is a logical consequence of the axioms. In other words, it tries to construct a formal proof that the goal must be true in every model that satisfies the assumptions.

If Prover9 succeeds, the result means that the queried statement is logically implied by the rules of the system. If it fails, this does not necessarily mean that the statement is false, but rather that it cannot be derived from the given axioms.

In this project, Prover9 is used to verify deterministic logical implications in the Mafia game. For example, rules such as:

"If a cop investigates a mafia player, then the cop recognizes that player as mafia"

can be expressed as logical axioms and proven automatically using Prover9. This makes Prover9 suitable for checking whether certain conclusions necessarily follow from the game rules and known actions.

2.1.2 Mace4: Finite Model Finding

Mace4 is a finite model finder for first-order logic. Instead of proving that a statement must be true, Mace4 searches for a finite interpretation (model) in which all the given assumptions hold simultaneously.

If Mace4 finds a model, this demonstrates that the set of formulas is logically consistent: there exists at least one possible "world" where all the stated facts and rules are satisfied. If no model is found within the given bounds, the assumptions may be inconsistent, or a larger search space may be required.

In the context of this project, Mace4 is used to validate Mafia game scenarios by checking whether a consistent game state exists. For example, situations involving night actions such as kills and protections can be tested to see whether they can coexist without contradiction. Mace4 is also used to answer satisfiability-style queries, such as whether a player can still be alive at a certain time given all previous actions.

2.1.3 Why Both Tools Are Needed

Prover9 and Mace4 address different but complementary questions:

- Prover9 answers: *"Is this statement necessarily true given the rules?"*
- Mace4 answers: *"Is there at least one consistent model where all these facts are true?"*

By combining both tools, this project is able to both prove logical consequences of Mafia game rules and explore the consistency of dynamically evolving game states. This dual approach provides a more complete logical analysis than using either theorem proving or model finding alone.

2.2 The Puzzle Selected

The logical puzzle selected for this project is the well-known social deduction game *Mafia*. Although Mafia is typically played as a party game involving discussion, deception, and incomplete information, its underlying mechanics can be formalized as a logical system with clearly defined roles, actions, and rules.

At its core, the game consists of a finite set of players, each assigned a role that determines their abilities and objectives. The main roles considered in this project are:

- **Mafia**, whose objective is to eliminate other players;
- **Villagers**, who aim to eliminate all mafia members;
- **Doctor**, who can protect a player from being killed during the night;
- **Cop**, who can investigate a player to determine whether they belong to the mafia.

The game progresses in discrete rounds, each composed of a *night* phase and a *day* phase. During the night, special roles perform actions such as killing, protecting, or investigating. During the day, players vote to eliminate one suspected player. These phases introduce state changes that affect which players are alive and which roles remain active.

From a logical perspective, Mafia can be seen as a constraint satisfaction problem. Each round introduces new facts (e.g., actions taken by players) and the game rules impose constraints on how these facts influence the next state of the game. For example, a player who is attacked by the mafia does not die if they are protected by the doctor, and a cop who investigates a mafia member gains correct information about that player.

The puzzle lies in determining whether a given sequence of actions and events is logically consistent, and what conclusions can be drawn from it. Questions such as:

- Can a player still be alive after a certain night?
- Is it possible for the cop to recognize a specific player as mafia?
- Does a given game state imply a win condition for one of the teams?

can be naturally expressed and analyzed using first-order logic.

This makes Mafia a suitable and interesting choice for modeling with Prover9 and Mace4. The game combines static properties (roles of players) with dynamic behavior (actions over time), allowing both theorem proving and model finding to be applied in a meaningful way. By treating each game state as a logical theory, the Mafia game becomes a structured logical puzzle rather than an informal social game.

2.3 All Implemented Features

This project implements a complete environment for modeling, simulating, and logically analyzing the Mafia game using First-Order Logic (FOL), Prover9, and Mace4. The implemented features cover scenario configuration, logical encoding, automated reasoning, and interactive simulation.

2.3.1 Scenario Configuration and Logical Encoding

A central feature of the project is the ability to define Mafia game scenarios in a structured and flexible way. Each scenario specifies:

- a finite set of players;
- a role assigned to each player (mafia, doctor, cop, villager);
- the initial alive state of all players;
- actions performed during night phases (kills, protections, investigations);
- eliminations occurring during day phases as a result of voting.

From this configuration, the system automatically generates Prover9 and Mace4 input files in the `.in` format. The generated files contain:

- ground facts representing roles and actions;
- axioms encoding the rules of the game;
- optional goals or queries, depending on whether Prover9 or Mace4 is used.

This automated generation ensures that the logical representation is syntactically correct and consistent across scenarios, while also allowing the user to focus on the logical structure of the game rather than manual formula construction.

2.3.2 Graphical User Interface and Game Simulation

The project includes a graphical user interface (GUI) implemented in Python using the Tkinter library. The GUI serves as an interactive layer between the user and the logical reasoning tools.

Through the interface, the user can:

- assign roles to players and initialize the game state;
- simulate night actions by selecting targets for mafia kills, doctor protection, and cop investigations;
- simulate day voting rounds, recording votes and eliminating players when applicable;
- export the current game state to logical input files;
- trigger automated reasoning using Prover9 or Mace4.

The GUI enforces key game constraints to prevent invalid configurations, such as disallowing self-voting, preventing multiple votes by the same player during a single day, and eliminating a player only when there is a unique highest vote count. These constraints ensure that only valid game states are passed to the logical reasoning layer.

2.3.3 Time-Indexed Modeling of Game States

To represent the dynamic nature of the Mafia game, the project introduces explicit time indexing. Game states are associated with discrete time constants (e.g., `n0`, `n1`, `n2`), and transitions between states are expressed using a `next` relation.

Predicates such as `alive(Player, Time)` and `kill(Attacker, Victim, Time)` allow the encoding of how actions performed at one time step affect the state of the game at the next step. Transition axioms specify, for example, that a player killed during a night will not be alive in the subsequent round unless protected.

This approach transforms the Mafia game into a sequence of logical states connected by well-defined transition rules, enabling reasoning over multiple rounds using first-order logic.

2.3.4 Automated Theorem Proving with Prover9

Prover9 is integrated into the system to verify logical consequences of the encoded rules. It is primarily used for deterministic reasoning tasks, where conclusions must necessarily follow from the assumptions.

Typical uses of Prover9 in this project include:

- proving that a cop recognizes a player as mafia after a successful investigation;
- verifying that certain outcomes are logically implied by the rules and actions performed.

By encoding these properties as goals, Prover9 attempts to derive formal proofs. A successful proof confirms that the conclusion holds in all models satisfying the assumptions, providing strong guarantees about the correctness of the logical rules.

2.3.5 Finite Model Finding and Consistency Checking with Mace4

Mace4 is used as a complementary reasoning tool to Prover9. Instead of proving implications, Mace4 searches for finite models that satisfy the given assumptions.

This functionality is essential for:

- checking whether a given game scenario is logically consistent;
- determining whether certain game states are possible under the defined rules;
- answering satisfiability-style queries, such as whether a player can remain alive after a sequence of actions.

If Mace4 finds a model, this demonstrates the existence of at least one valid interpretation of the game state. If no model is found within the specified bounds, the scenario may be inconsistent or require a larger domain for analysis.

2.3.6 Automated Queries to the Logical Model

The project supports automated querying of logical models, particularly when using Mace4. Queries can be formulated to check whether specific predicates (such as `alive(Player, Time)`) are compatible with the current game state.

These queries allow the system to explore hypothetical outcomes and verify whether certain assumptions lead to contradictions. This feature highlights the practical use of model finding as a tool for exploring alternative game evolutions.

2.3.7 Win Condition Detection

Win conditions for both teams are encoded using logical rules:

- villagers win if no mafia players remain alive;
- mafia win if all remaining alive players are mafia.

These conditions can be checked using automated reasoning, providing a formal way to determine the outcome of a game state. While simplified, this approach demonstrates how high-level game objectives can be expressed and evaluated within first-order logic.

2.3.8 Heuristic Suggestions and Decision Support

In addition to strict logical reasoning, the system provides heuristic suggestions for player actions based on the current game state. These suggestions are not logically guaranteed but are informed by the available information and logical checks.

This combination of formal reasoning and heuristic guidance illustrates how logical tools can support decision-making in games with uncertainty and incomplete information.

2.3.9 Extensibility and Modularity

The project is designed to be modular and extensible. The separation between scenario generation, logical encoding, automated reasoning, and user interaction allows new roles, rules, or constraints to be added with minimal modifications.

This design encourages experimentation with alternative Mafia variants and demonstrates how first-order logic can be adapted to model increasingly complex systems.

2.4 Application Workflow

The developed system follows a clear application workflow that connects user interaction, logical constraint generation, automated reasoning, and result interpretation into a single coherent process. The workflow can be seen as a pipeline in which each stage produces structured output that is consumed by the next stage.

2.4.1 Scenario Definition via Desktop Application

The workflow starts at the level of the desktop application, implemented in Python with a graphical user interface. The user defines a Mafia game scenario by interacting with the GUI, which allows:

- selecting the number of players;
- assigning roles to each player;
- simulating night actions (kills, protections, investigations);
- simulating day actions such as voting and eliminations.

At this stage, the game state exists purely as structured data inside the application (e.g., Python objects and dictionaries). The GUI enforces basic game constraints to ensure that only valid scenarios can be constructed.

2.4.2 Automatic Generation of Constraint Files

Once a scenario is defined, the application automatically translates the internal representation of the game state into a formal logical specification. This is done by generating a constraint file in the Prover9/Mace4 `.in` format.

The generated file includes:

- ground facts representing roles and current states (e.g., who is alive at a given time);
- action facts describing events such as kills, protections, and investigations;
- logical axioms encoding the rules of the Mafia game;
- optional goals or queries, depending on the reasoning task.

This step effectively acts as a compiler from the application-level game description to a First-Order Logic theory.

2.4.3 Execution of Automated Reasoning Tools

After the constraint file is generated, the application invokes the appropriate reasoning tool:

- **Prover9** is executed when the task is to prove a logical consequence (e.g., whether a cop necessarily recognizes a mafia player);
- **Mace4** is executed when the task is to search for a finite model satisfying the constraints (e.g., checking consistency or satisfiability).

The tools are executed externally, and their standard output is captured programmatically by the application. This allows the reasoning process to remain fully automated from the user's perspective.

2.4.4 Capturing and Parsing the Output

Once Prover9 or Mace4 finishes execution, the application captures the textual output produced by the tool. This output contains either:

- a proof or proof status (in the case of Prover9), or
- a finite model description or a failure report (in the case of Mace4).

The application processes this output internally, extracting relevant information such as:

- whether a proof was found;
- whether a finite model exists;
- which predicates are true in the resulting model.

If parsing fails or the output format is unexpected, the raw output can still be displayed, ensuring transparency of the reasoning process.

2.4.5 Internal Interpretation of Results

After parsing, the results are interpreted at the application level. For example:

- a successful Prover9 proof confirms that a certain conclusion is logically implied by the rules;
- a model found by Mace4 confirms that a scenario is logically consistent;
- the absence of a model indicates potential inconsistency or conflicting constraints.

These interpretations are mapped back to game-level concepts, such as player survival, recognition of roles, or win conditions.

2.4.6 Feedback to the User

The final step of the workflow is presenting the results to the user through the GUI. The application provides feedback in an intuitive form, such as:

- confirmation messages (e.g., a player can still be alive);
- detected win conditions;
- warnings about inconsistent or contradictory scenarios;
- heuristic suggestions for possible actions.

This closes the loop between user interaction and logical reasoning, allowing the user to iteratively modify scenarios and immediately observe the logical consequences.

2.4.7 Summary of the Workflow

Overall, the application workflow can be summarized as follows:

1. The user defines a game scenario using the desktop application.
2. The application generates a First-Order Logic constraint file.
3. Prover9 or Mace4 is executed automatically.
4. The output is captured and processed internally.
5. Results are interpreted and presented back to the user.

This workflow demonstrates how automated reasoning tools can be effectively integrated into a practical application, transforming abstract logical formalisms into an interactive and usable system.

2.5 Example Scenarios and Results

This section presents several representative Mafia game scenarios together with their logical encodings and reasoning results. Each example illustrates how First-Order Logic formulas are generated, processed by Prover9 or Mace4, and interpreted by the application.

2.5.1 Cop Investigation Scenario

In this scenario, a cop investigates a player who is a mafia member. The purpose is to verify that the logical rules imply correct recognition.

Logical Encoding The following predicates and axioms are used:

$$\text{isCop}(c)$$

$$\text{isMafia}(a)$$

$$\text{investigate}(c, a)$$

The main rule of the game is encoded as:

$$\forall C \forall T \ (\text{isCop}(C) \land \text{isMafia}(T) \land \text{investigate}(C, T) \rightarrow \text{recognizes}(C, T))$$

Goal The goal submitted to Prover9 is:

$$\text{recognizes}(c, a)$$

Result Prover9 successfully proves the goal, showing that recognition is a logical consequence of the assumptions. This confirms that the investigation rule is correctly modeled.

2.5.2 Mafia Kill with Doctor Protection

This scenario models a night in which a mafia player attempts to kill a villager, but the doctor protects the same player.

Logical Encoding Initial facts:

$$\text{isMafia}(a), \quad \text{isDoctor}(b), \quad \text{isVillager}(d)$$

$$\text{alive}(d)$$

$$\text{kill}(a, d)$$

$$\text{protect}(b, d)$$

Game rules:

$$\forall X \forall Y \ (\text{kill}(X, Y) \land \neg\text{protected}(Y) \rightarrow \neg\text{alive}(Y))$$

$$\forall D \forall Y \ (\text{protect}(D, Y) \rightarrow \text{protected}(Y))$$

Goal The goal is to prove:

$$\neg\text{alive}(d)$$

Result Prover9 fails to prove the goal. This is the expected outcome, since the protection rule blocks the implication leading to death. The example demonstrates that the absence of a proof does not imply falsity, but rather non-derivability.

2.5.3 Consistency Check Using Mace4

The same kill-with-protection scenario is evaluated using Mace4, without specifying any goals.

Purpose Mace4 is used to determine whether the set of assumptions is logically consistent.

Result Mace4 finds a finite model satisfying all assumptions. This confirms that the rules governing killing and protection do not contradict each other and that such a scenario can exist in a valid interpretation.

2.5.4 Multi-Night Scenario with Time Indexing

To model the dynamic evolution of the game, time-indexed predicates are introduced.

Time Representation Discrete time constants are defined:

$$n_0, n_1, n_2$$

with transitions:

$$\text{next}(n_0, n_1), \quad \text{next}(n_1, n_2)$$

Alive status is indexed by time:

$$\text{alive}(p, n)$$

Transition Rule The effect of a kill across time is encoded as:

$$\forall X \forall Y \forall N \forall M \ (\text{kill}(X, Y, N) \land \neg\text{protected}(Y, N) \land \text{next}(N, M) \rightarrow \neg\text{alive}(Y, M))$$

Result Mace4 finds a finite model satisfying all constraints across multiple nights. This confirms that the sequence of actions and eliminations is logically coherent over time.

2.5.5 Application-Level Workflow

At the application level, the interaction between the GUI, logic generation, and automated reasoning can be summarized as follows:

```
function runScenario(gameState):
    constraints = generateLogic(gameState)
    writeToFile(constraints)

    if queryType == PROOF:
        output = runProver9(file)
        result = parseProof(output)
    else if queryType == MODEL:
        output = runMace4(file)
        result = parseModel(output)

    displayResult(result)
```

This pseudocode highlights how the application acts as an intermediary between the user-defined scenario and the logical reasoning tools.

2.5.6 Win Condition Evaluation

Win conditions are encoded using logical implications. For example, villagers win if no mafia players remain alive:

$$\forall N \ (\forall P \ (\text{isMafia}(P) \rightarrow \neg \text{alive}(P, N))) \rightarrow \text{villagersWin}(N)$$

These conditions are evaluated after applying night and day transitions, allowing automated detection of game outcomes.

2.6 Graphical User Interface and Interaction Flow

To support interactive scenario construction and logical analysis, the project includes a desktop graphical user interface (GUI). The GUI serves as the main entry point for the user and provides full control over scenario configuration, simulation, and automated reasoning.

2.6.1 Initial Configuration and File Generation

Figure 2.1 shows the initial state of the application. At this stage, the user specifies:

- the total number of players;
- the number of mafia, doctors, and cops;
- the number of nights to simulate.

Once the configuration is defined, the user can generate a Prover9/Mace4 constraint file directly from the GUI. The application automatically translates the configuration into a valid `.in` file containing logical facts and axioms.

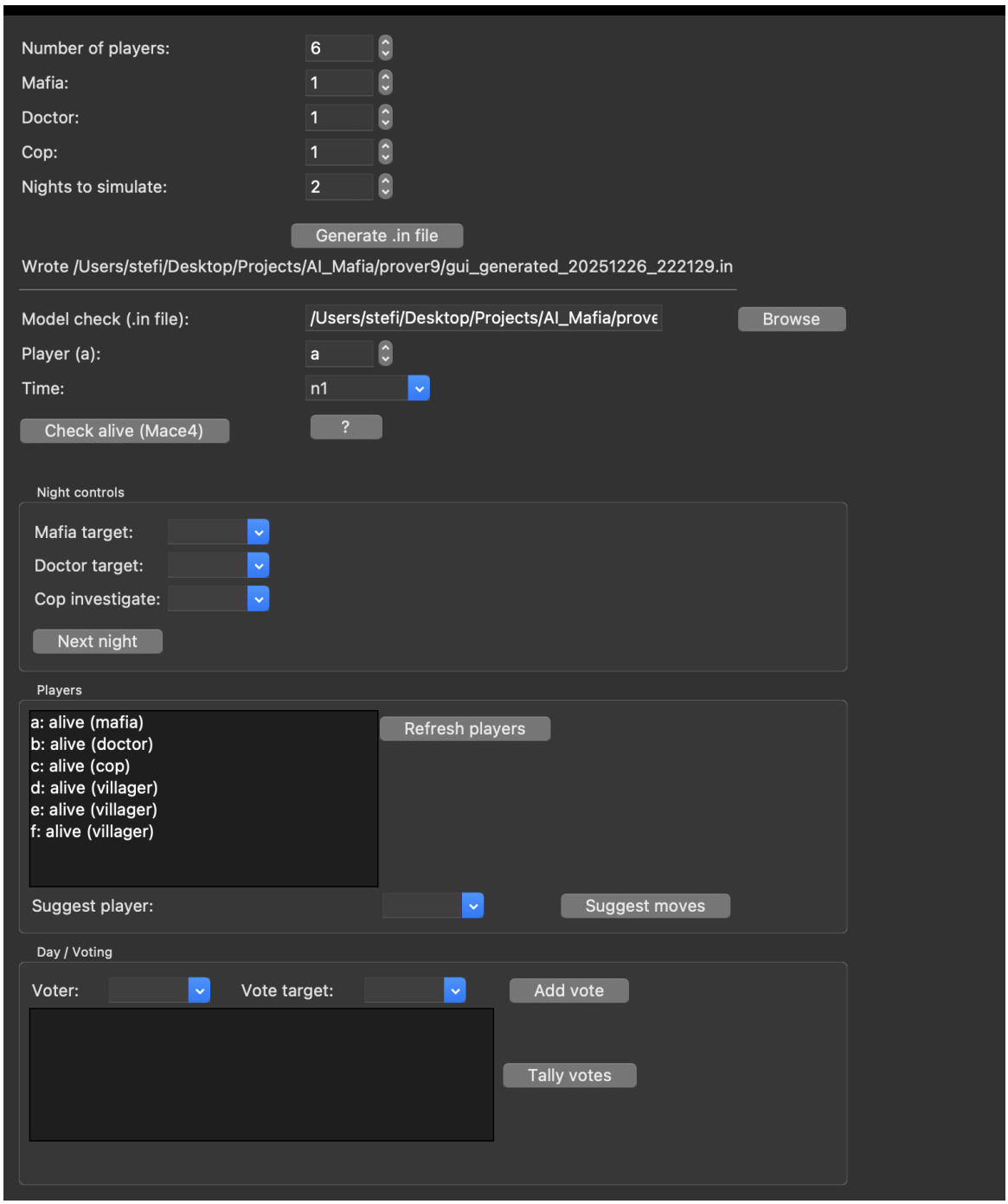

Figure 2.1: Initial GUI state for scenario configuration.

Figure 2.2 confirms the successful generation of the constraint file, which can then be used for automated reasoning.

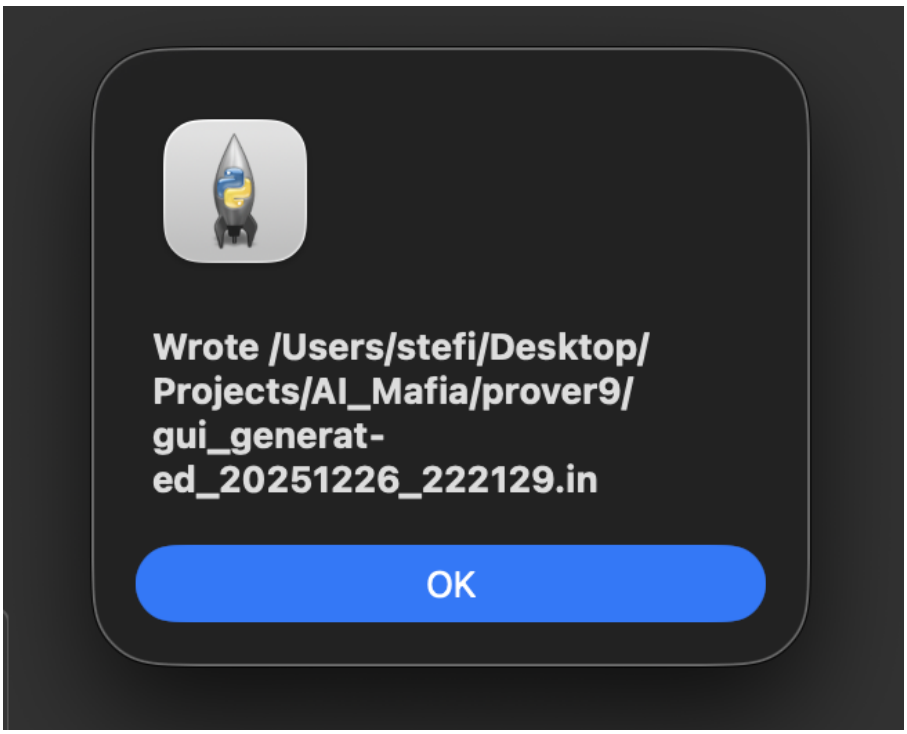

Figure 2.2: Confirmation of generated Prover9/Mace4 constraint file.

2.6.2 Night Phase Simulation

During the night phase, the GUI allows the user to select actions corresponding to special roles:

- the mafia chooses a kill target;
- the doctor selects a player to protect;
- the cop chooses a player to investigate.

After executing the night phase, the application updates the internal game state and records the actions as logical facts indexed by time. Figure 2.3 illustrates the result of a night phase where a villager is killed and the cop successfully investigates a mafia member.

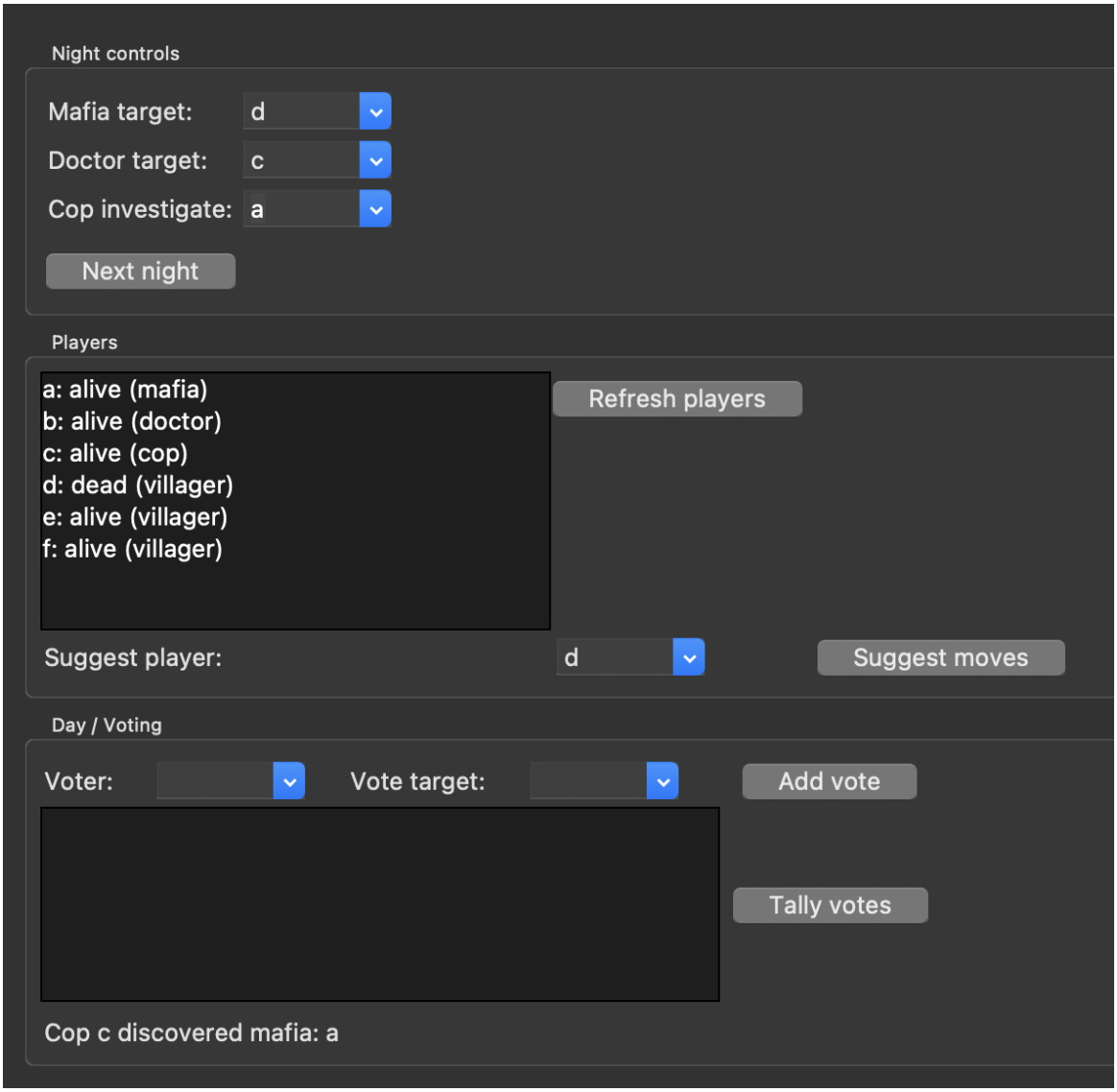

Figure 2.3: Game state after night actions: kill, protection, and investigation.

2.6.3 Day Phase and Voting

During the day phase, players cast votes to eliminate a suspected mafia member. The GUI enforces voting rules, such as:

- each player may vote only once;
- self-voting is disallowed;
- a player is eliminated only if there is a unique highest vote count.

Figure 2.4 shows the voting interface, where votes are collected and displayed.

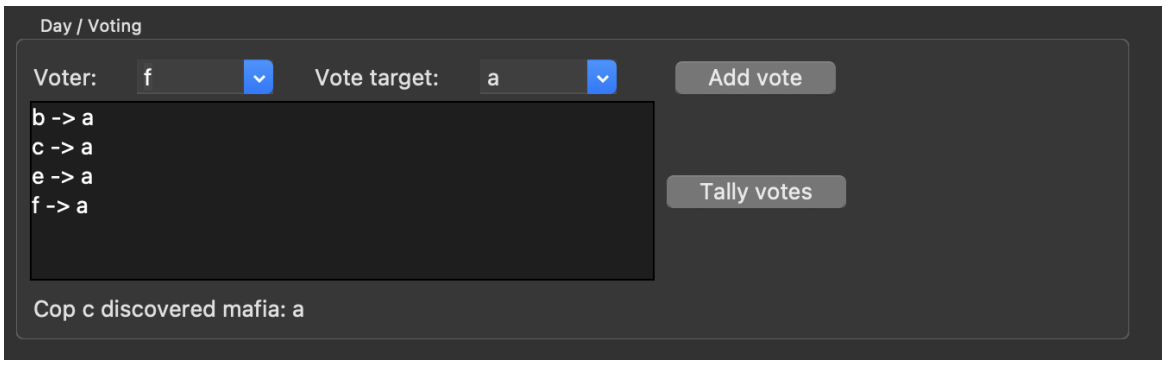

Figure 2.4: Day phase voting interface.

After tallying the votes, the eliminated player is removed from the game state, as shown in Figure 2.5.

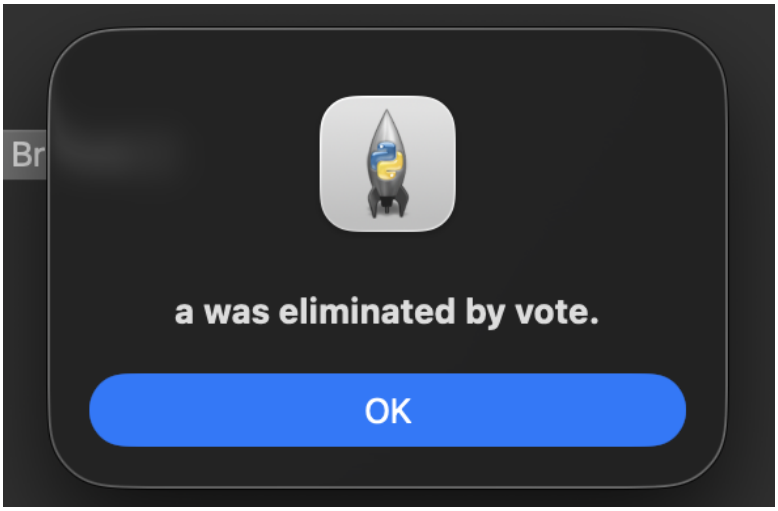

Figure 2.5: Player elimination after vote tallying.

2.6.4 Win Condition Detection

After each phase, the application checks whether a win condition has been reached. Villagers win if all mafia players have been eliminated, while mafia win if they reach numerical dominance.

Figure 2.6 shows a scenario where all mafia players have been eliminated, and the villagers are declared winners.

Figure 2.6: Detected win condition: villagers win.

2.6.5 Role-Based Suggestions and Logical Guidance

An important feature of the GUI is the *Suggest Moves* functionality. Based on the current game state, the application provides role-specific guidance informed by logical consistency checks and heuristics.

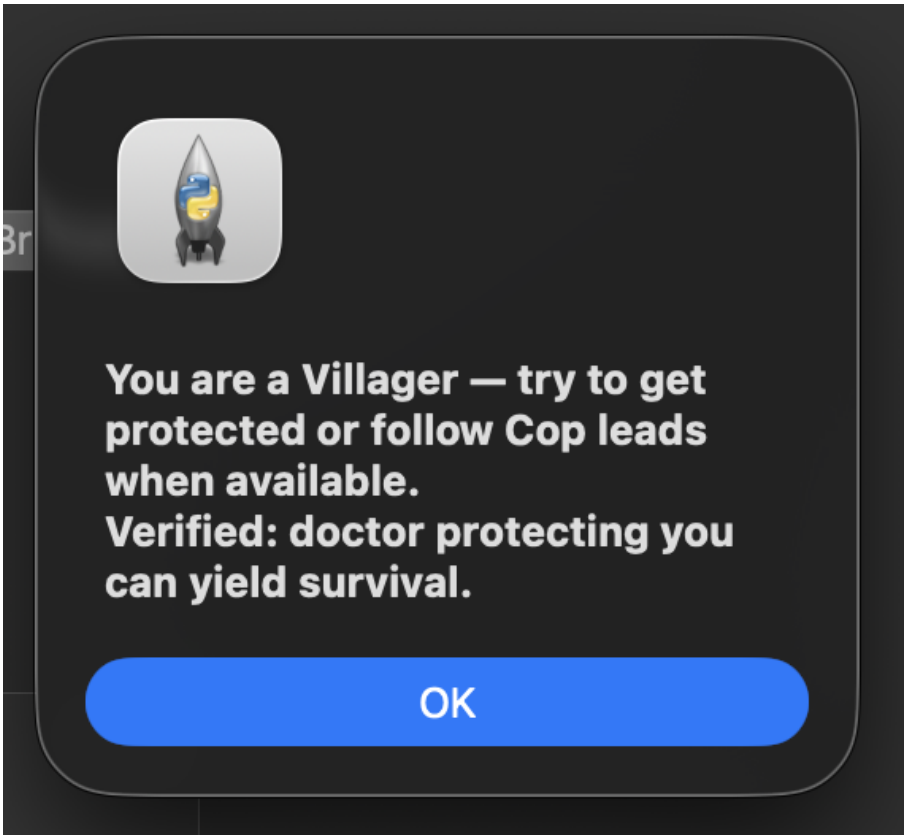

Figure 2.7: Suggested moves for villagers.

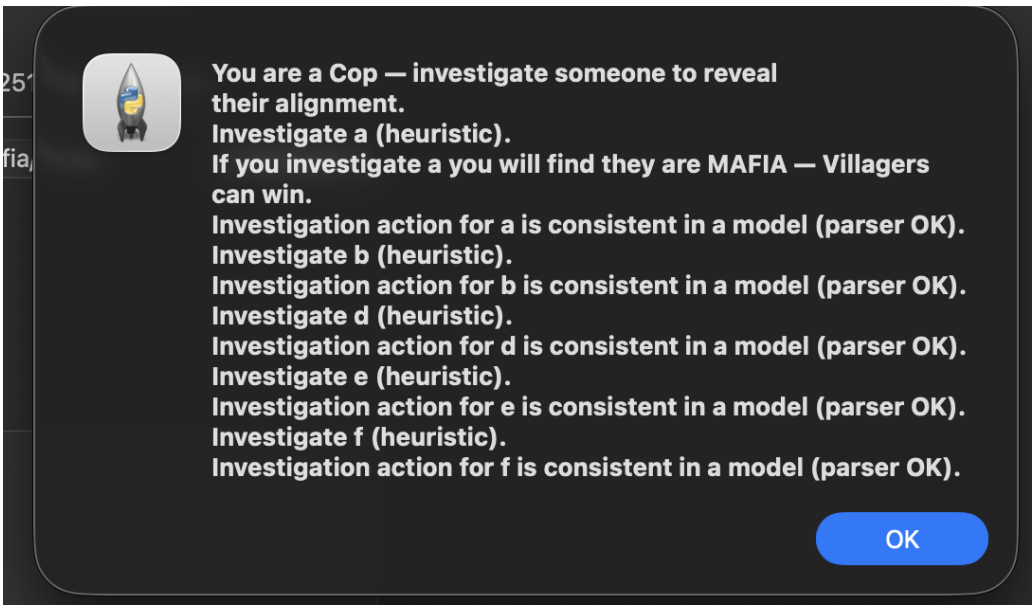

Figure 2.8: Suggested investigation actions for the cop.

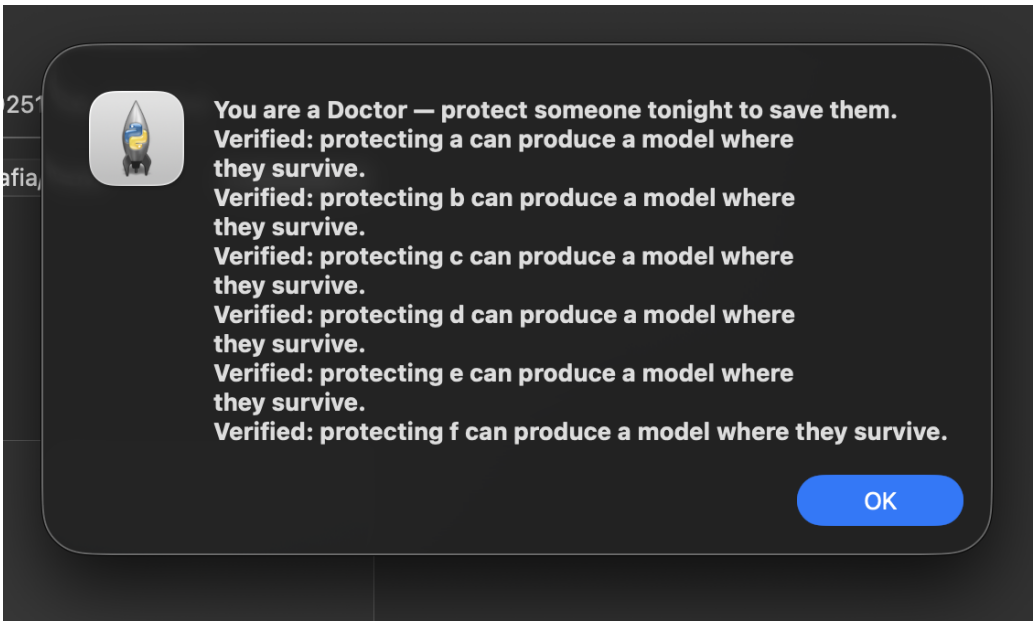

Figure 2.9: Suggested protection actions for the doctor.

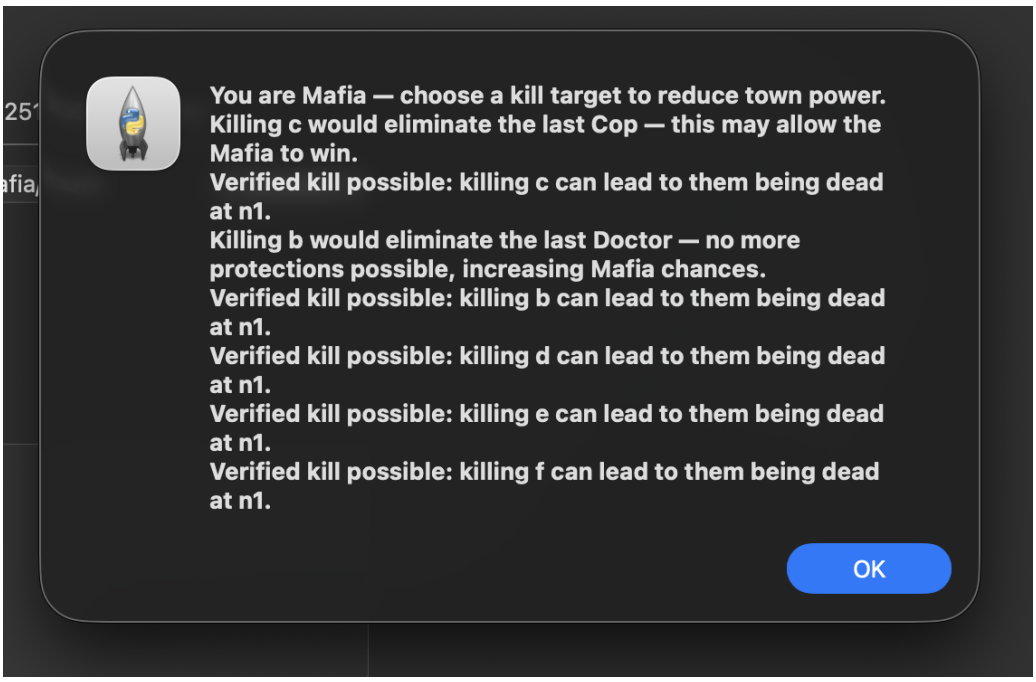

Figure 2.10: Suggested kill actions for the mafia.

2.7 UML Diagrams

To better illustrate the structure and behavior of the application, Unified Modeling Language (UML) diagrams are used. These diagrams provide a high-level view of the system workflow and interactions between components, complementing the logical and implementation details presented earlier.

2.7.1 Activity Diagram

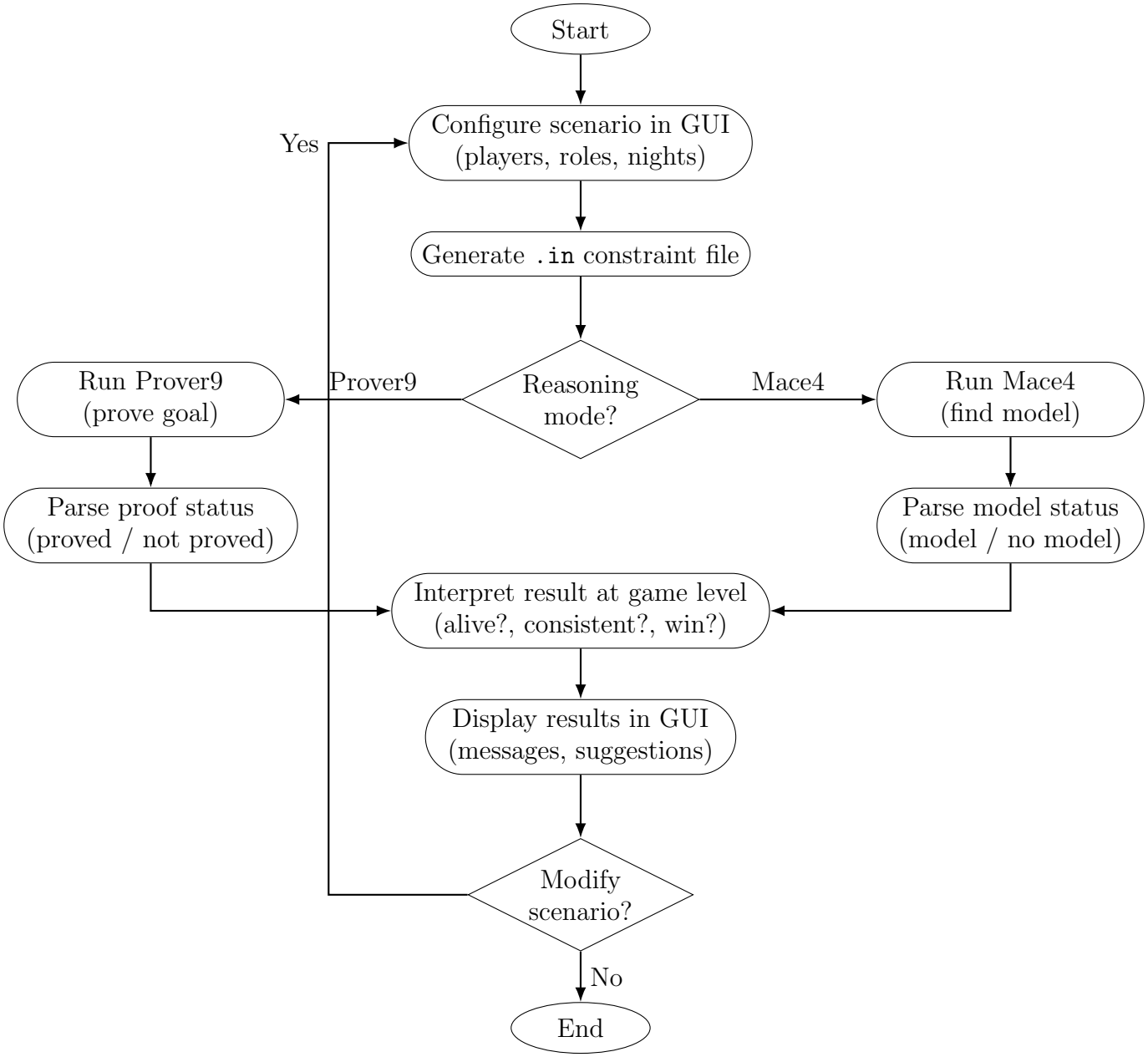

Figure 2.11: Activity diagram: end-to-end application workflow (GUI → constraint generation → Prover9/Mace4 → parsing → feedback).

2.7.2 Sequence Diagram

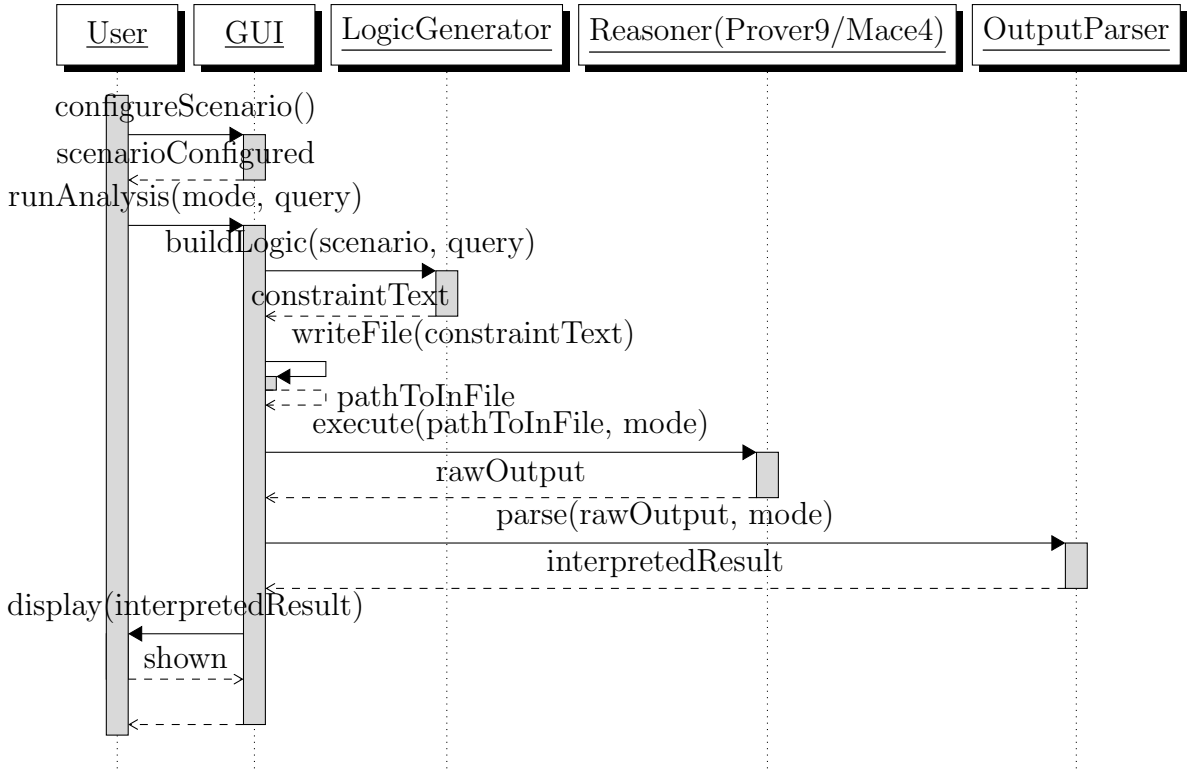

Figure 2.12: Sequence diagram: typical reasoning request from GUI to Prover9/Mace4 and back.

2.7.3 Class Diagram

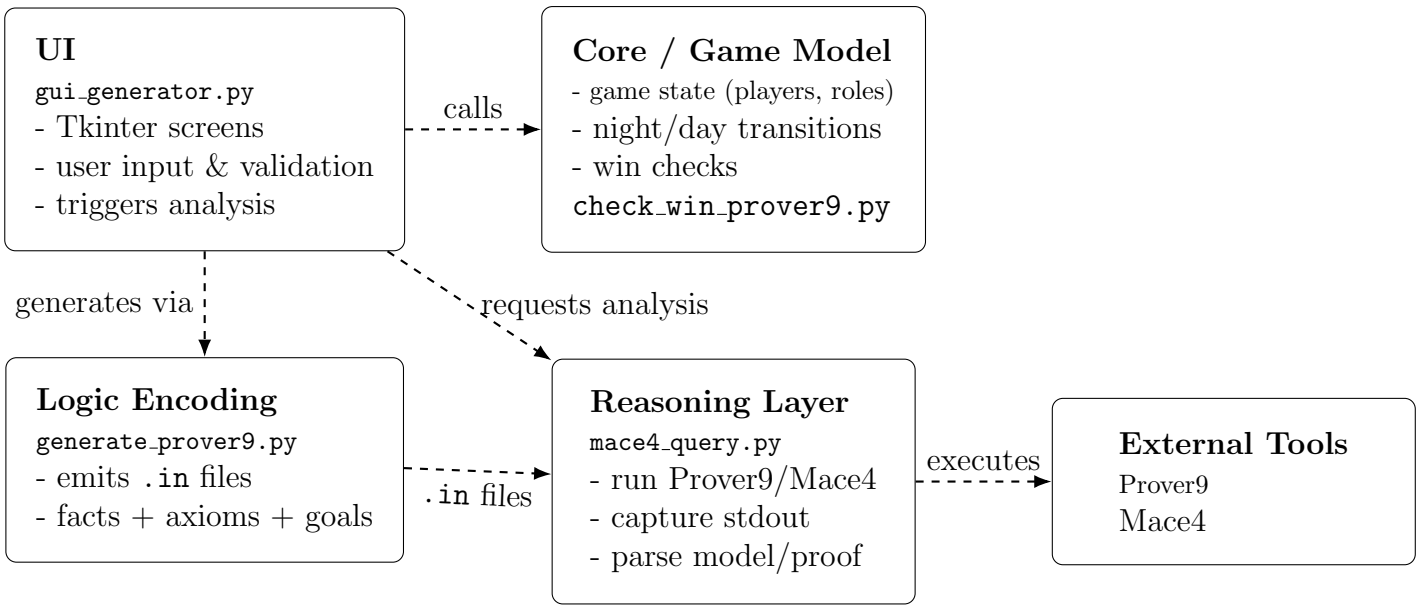

Figure 2.13: Package diagram: major modules and dependencies of the application.

2.7.4 Use Case Diagram

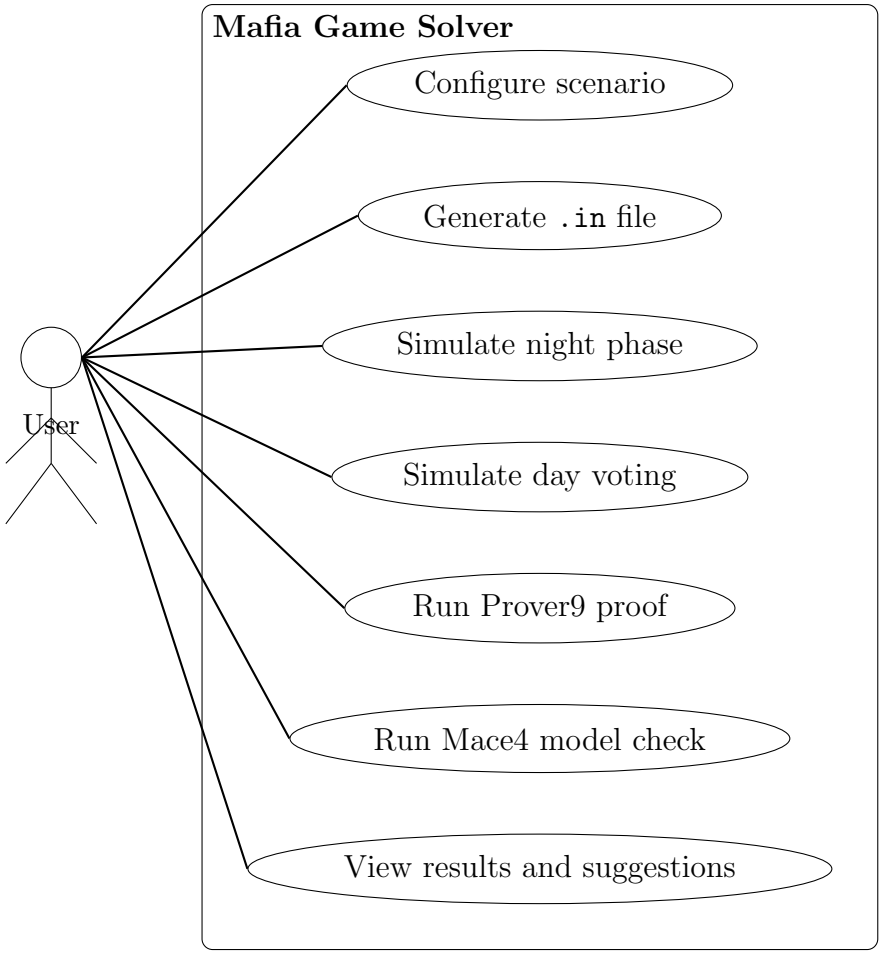

Figure 2.14: Use case diagram showing the main interactions between the user and the Mafia solver application.

2.8 Commands Used to Run Prover9 and Mace4

In this project, Prover9 and Mace4 are not executed manually by the user. Instead, they are invoked programmatically by the desktop application after the logical constraint files are automatically generated by the GUI.

2.8.1 Automatic Generation of Constraint Files

When the user configures a Mafia game scenario using the graphical interface, the application internally constructs a First-Order Logic representation of the current game state. This representation is then written to a Prover9/Mace4 input file (`.in`) using relative paths.

For example, a generated file may be stored in the project directory as:

```
prover9/generated_scenario.in
```

This file contains all required assumptions, axioms, and optional goals derived from the GUI state.

2.8.2 Executing Prover9 from the Application

If the selected task is logical proof (e.g., proving that a cop recognizes a mafia player), the application invokes Prover9 automatically using a system call.

The path to the input file is relative to the project directory, ensuring portability across different systems. The output produced by Prover9 is captured directly by the application and processed internally.

2.8.3 Executing Mace4 from the Application

For consistency checks and model finding, the application invokes Mace4 in a similar way.

Mace4 attempts to find a finite model that satisfies all constraints. The result (model found or no model found) is returned as text output, which is then parsed by the application.

2.8.4 Internal Execution Logic (Pseudocode)

The interaction between the GUI, file generation, and automated reasoning tools can be summarized as follows:

```
function runReasoning(gameState, mode):
    inFile = generateInFile(gameState)

    if mode == PROVER9:
        output = execute("prover9 -f " + inFile)
    else if mode == MACE4:
        output = execute("mace4 -f " + inFile)

    result = parseOutput(output)
    return result
```

This approach hides the complexity of command-line interaction from the user while preserving the full power of automated reasoning.

2.8.5 Running the Graphical User Interface

The application is launched by executing the Python script that implements the graphical user interface. This script initializes the game state, displays the configuration interface, and orchestrates all subsequent steps of the workflow.

The GUI is started using a relative path as follows:

```
python3 scripts/gui_generator.py
```

Once the GUI is running, the user can configure scenarios, simulate game phases, and trigger automated reasoning without interacting directly with Prover9 or Mace4 from the command line.

2.8.6 Project Repository

The complete source code of the application, including the graphical user interface, logic generators, example Prover9 and Mace4 input files, and documentation, is publicly available in a Git repository. The repository can be accessed at:

`https://github.com/stefi19/Mafia-Game-Solver`

Chapter 3

A3: Planning

Bibliography

[1] UC Berkeley. Project 1: Search — cs188. URL: https://inst.eecs.berkeley.edu/~cs188/fa25/projects/proj1/#welcome-to-pacman.

[2] UC Berkeley. Project 2: Multi-agent search — cs188. URL: https://inst.eecs.berkeley.edu/~cs188/fa25/projects/proj2/.

[3] GeeksforGeeks. A* search algorithm. URL: https://www.geeksforgeeks.org/dsa/a-search-algorithm/.

[4] GeeksforGeeks. Difference between bfs and dfs. URL: https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/.

[5] GeeksforGeeks. Mini-max algorithm in artificial intelligence. URL: https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/.

[6] GeeksforGeeks. Uniform cost search (ucs) in ai. URL: https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/.

[7] Wikipedia contributors. A* search algorithm. URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

[8] Wikipedia contributors. Minimax. URL: https://en.wikipedia.org/wiki/Minimax.

Appendix A

Your original code

A.1 BFS

Listing A.1: Breadth-First Search implementation

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    from util import Queue
    # Each element in the queue: (state, path_to_state)
    queue = Queue()
    queue.push((problem.getStartState(), []))
    # A set to record visited states and prevent revisiting
    visited = set()
    while not queue.isEmpty():
        state, path = queue.pop()  # Get the earliest inserted unexplored
            state
        # If we've reached the goal, return the path of actions
        if problem.isGoalState(state):
            return path
        if state not in visited:
            visited.add(state)
            # Expand the current state by generating its successors
            for successor, action, stepCost in problem.getSuccessors(state):
                if successor not in visited:
                    # Add successor to queue with updated path
                    queue.push((successor, path + [action]))
    # If no solution is found, return empty path
    return []
```

A.2 DFS

Listing A.2: Depth-First Search implementation

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.
    """
    "*** YOUR CODE HERE ***"
    from util import Stack
```

```
    # Each element on the stack: (state, path)
    # path is the sequence of actions taken to reach this state
    stack = Stack()
    stack.push((problem.getStartState(), []))
    # A set to keep track of already visited states
    visited = set()
    # DFS loop: keep exploring until stack is empty
    while not stack.isEmpty():
        state, path = stack.pop()  # Get the most recent unexplored state
        # Check if we've reached the goal state
        if problem.isGoalState(state):
            return path  # Found a solution, so we return the path
        # If this state hasnt been visited yet, expand it
        if state not in visited:
            visited.add(state)
            # For each possible move from this state
            for successor, action, stepCost in problem.getSuccessors(state):
                # Only consider unvisited states
                if successor not in visited:
                    # Push successor onto the stack with the updated path +
                        [action]
                    stack.push((successor, path + [action]))
    # If we exhaust the stack without finding a goal, return empty path
    return []
```

A.3 UCS

Listing A.3: Uniform Cost Search implementation

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    from util import PriorityQueue
    # Each element in PQ: (state, path_to_state, cumulative_cost)
    frontier = PriorityQueue()
    frontier.push((problem.getStartState(), [], 0), 0)
    visited = set()
    while not frontier.isEmpty():
        state, path, costSoFar = frontier.pop()
        # If this state has already been expanded, skip it
        if state in visited:
            continue
        visited.add(state)
        # Goal check
        if problem.isGoalState(state):
            return path
        # Expand successors
        for successor, action, stepCost in problem.getSuccessors(state):
            newCost = costSoFar + stepCost
            frontier.push((successor, path + [action], newCost), newCost)
    return []
```

A.4 A*

Listing A.4: A* Algorithm implementation

```python
def manhattanHeuristic(state, problem=None):
    """
    Heuristic based on Manhattan distance between the current state
    and the goal state.
    """
    x1, y1 = state
    x2, y2 = problem.goal
    return abs(x1 - x2) + abs(y1 - y2)

def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[
    Directions]:
    """Search the node that has the lowest combined cost and heuristic first
        ."""
    "*** YOUR CODE HERE ***"
    from util import PriorityQueue
    # starting point
    start_state = problem.getStartState()
    # keeps track of (state, moves_so_far, cost_so_far)
    frontier = PriorityQueue()
    frontier.push((start_state, [], 0), heuristic(start_state, problem))
    # remember the cheapest cost  w e ve  seen to each state
    seen_costs = {start_state: 0}
    while not frontier.isEmpty():
        current_state, moves, cost = frontier.pop()
        # if goal reached, return the moves that got us here
        if problem.isGoalState(current_state):
            return moves
        # if we already found a cheaper path to this state, skip it
        if cost > seen_costs.get(current_state, float("inf")):
            continue
        for next_state, direction, step_cost in problem.getSuccessors(
            current_state):
            new_cost = cost + step_cost
            # only consider this path if it's better than what we had before
            if new_cost < seen_costs.get(next_state, float("inf")):
                seen_costs[next_state] = new_cost
                priority = new_cost + heuristic(next_state, problem)
                frontier.push((next_state, moves + [direction], new_cost),
                    priority)
    # if no path found
    return []
```

A.5 Minimax

Listing A.5: Minimax Algorithm implementation

```python
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Minimax agent: Pacman tries to maximize the score,
    ghosts try to minimize it.
    """

    def getAction(self, gameState: GameState):
        """
        Return the best action for Pacman using minimax search.
        """
        # recursive helper function
        def minimax(state, agentIndex, depth):
```

```
            # if game is over (win/lose) or we reached max depth, then stop
            if depth == self.depth or state.isWin() or state.isLose():
                return self.evaluationFunction(state)
            # Pacman is agent 0, MAX player
            if agentIndex == 0:
                bestScore = float("-inf")
                for move in state.getLegalActions(agentIndex):
                    nextState = state.generateSuccessor(agentIndex, move)
                    score = minimax(nextState, 1, depth)  # go to first
                        ghost
                    bestScore = max(bestScore, score)
                return bestScore
            # Ghosts are agents 1..N-1, MIN players
            else:
                nextAgent = agentIndex + 1
                nextDepth = depth
                # if we finished with last ghost, go back to Pacman and
                    increase depth
                if nextAgent == state.getNumAgents():
                    nextAgent = 0
                    nextDepth += 1
                bestScore = float("inf")
                for move in state.getLegalActions(agentIndex):
                    nextState = state.generateSuccessor(agentIndex, move)
                    score = minimax(nextState, nextAgent, nextDepth)
                    bestScore = min(bestScore, score)
                return bestScore
        # Pacmans actual decision: choose the action that gives the
            highest minimax value
        bestAction = None
        bestScore = float("-inf")
        for move in gameState.getLegalActions(0):
            nextState = gameState.generateSuccessor(0, move)
            score = minimax(nextState, 1, 0)  # start recursion with ghost 1
                at depth 0
            if score > bestScore:
                bestScore = score
                bestAction = move
        return bestAction
```

A.6 Graphical User Interface Entry Point

Listing A.6: Launching the Mafia Game GUI

```
import tkinter as tk

def main():
    root = tk.Tk()
    root.title("Mafia Game Solver")
    app = MafiaGUI(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

A.7 Automatic Generation of Prover9 and Mace4 Files

Listing A.7: Generating Prover9/Mace4 constraint files from game state

```
def generate_in_file(game_state, filename):
    with open(filename, "w") as f:
        f.write("formulas(assumptions).\n")

        # roles
        for player, role in game_state.roles.items():
            f.write(f"is{role}({player}).\n")

        # alive players at initial time
        for player in game_state.alive_players:
            f.write(f"alive({player}, n0).\n")

        # night actions
        for killer, target, time in game_state.kills:
            f.write(f"kill({killer},{target},{time}).\n")

        for doctor, target, time in game_state.protections:
            f.write(f"protect({doctor},{target},{time}).\n")

        for cop, target, time in game_state.investigations:
            f.write(f"investigate({cop},{target},{time}).\n")

        f.write("end.\n")
```

A.8 Running Prover9 from the Application

Listing A.8: Executing Prover9 using a relative path

```
import subprocess

def run_prover9(input_file):
    command = ["prover9", "-f", input_file]
    result = subprocess.run(
        command,
        capture_output=True,
        text=True
    )
    return result.stdout
```

A.9 Running Mace4 from the Application

Listing A.9: Executing Mace4 to search for a finite model

```
def run_mace4(input_file):
    command = ["mace4", "-f", input_file]
    result = subprocess.run(
        command,
        capture_output=True,
        text=True
    )
    return result.stdout
```

A.10 Parsing Mace4 Output

Listing A.10: Checking whether Mace4 found a model

```
def model_exists(mace4_output):
    return "MODEL FOUND" in mace4_output
```

A.11 Win Condition Detection

Listing A.11: Detecting win conditions from the game state

```
def check_win(game_state):
    mafia_alive = [
        p for p in game_state.alive_players
        if game_state.roles[p] == "Mafia"
    ]

    if not mafia_alive:
        return "Villagers win"

    villagers_alive = len(game_state.alive_players) - len(mafia_alive)
    if len(mafia_alive) >= villagers_alive:
        return "Mafia win"

    return "Game continues"
```

A.12 Suggested Moves Logic

Listing A.12: Role-based heuristic suggestions

```
def suggest_moves(role, alive_players):
    if role == "Cop":
        return ["Investigate a player with unknown role"]

    if role == "Doctor":
        return ["Protect a player likely to be targeted"]

    if role == "Mafia":
        return ["Eliminate a non-mafia player"]

    return []
```

A.13 End-to-End Reasoning Workflow

Listing A.13: Overall reasoning workflow

```
def run_reasoning(game_state, mode):
    in_file = generate_in_file(game_state, "prover9/generated.in")

    if mode == "PROVER9":
        output = run_prover9(in_file)
    else:
        output = run_mace4(in_file)
```

```
8
9       return output
```