CentraleSupélec | MScAI Master of Science in Artificial Intelligence Applied to society

ANNÉE: 2024-2025

AI CHALLENGE
-VAMPIRES VS WEREWOLVES-

AI FUNDAMENTALS

Realised by:

Manon LAGARDE
Maria KONTARATOU
Chaimae SADOUNE

Supervised by:

JEAN-PHILIPPE POLI

Table des matières

Introduction

1 Game Description

1.1 Bref Overview

The *Vampires vs. Werewolves* game is set in a mystical universe where two supernatural species-vampires and werewolves-engage in a nightly struggle for dominance. The game is played on a grid-based map, with humans populating specific cells. Each player controls one of the species, either vampires or werewolves, with the goal of converting humans and eliminating the opposing species.

The game is a turn-based strategy challenge, requiring careful planning, resource management, and strategic moves to outnumber and overpower the adversary. The dynamic nature of the game, influenced by random battles and competitive interactions, makes it an exciting test of artificial intelligence (AI) design.

1.2 Objective

The primary objective of the game is to establish dominance as either vampires or werewolves. This is achieved by :

— Converting humans into creatures of your species.
— Defeating enemy creatures in battles.
— Occupying as much of the grid as possible while minimizing the opponent's influence.

The AI agent developed for this project is tasked with making optimal moves to maximize its species' presence while adhering to the rules of movement, conversion, and combat. Success is determined by the ability to balance aggression, resource acquisition, and defense against the opposing species within the constraints of the game.

2 Game Rules

2.1 Universe Representation

The game takes place on an $n \times m$ grid, representing the game world. The grid contains the following entities :

— **Humans :** Located in specific cells, they can be converted into vampires or werewolves.

— **Vampires and Werewolves :** Start in their designated initial positions, occupying separate cells. All creatures of the same species initially share the same cell.
— **Empty Cells :** These can be traversed by either species.

Instead of using a sparse matrix to represent the game state, the implementation uses a more efficient representation that tracks the positions and counts of each entity (humans, vampires, and werewolves) using dictionaries. This approach reduces memory usage and computational overhead.

2.2 Actions

Each turn, a player must make at least one move. Players can :

— **Move creatures :**
  · Vampires and werewolves can move either all or part of their group from one cell to an adjacent cell (up to 8 possible directions).
  · Moves are restricted to adjacent cells unless at the border of the grid.
— **Interact with the contents of the destination cell :**
  · Convert humans.
  · Engage in battles with enemy creatures.
— **Split groups :** A group of creatures can split into subgroups to occupy different cells and optimize strategies.

Rules for Actions
— **Rule 1 :** At least one movement must occur per turn.
— **Rule 2 :** Only creatures of the player's species can be moved.
— **Rule 3 :** Moves depend on the number of creatures in the source cell ; adequate numbers must remain for valid actions.

2.3 Conversion and Battles

The interactions with the destination cell depend on its contents :

— **Converting Humans :**
  · To convert humans in a cell, the number of creatures moving into the cell must equal or exceed the number of humans.
  · Successfully converted humans become creatures of the player's species.
— **Battles with Opponents :** If the destination cell contains enemy creatures :
  · The attackers must outnumber the defenders by 1.5 times to kill them.
  · If the attackers are fewer than 1.5 times the defenders, a random battle occurs, with the outcome determined probabilistically.

Random Battle Probability :
— If the attacking creatures (E_1) are equal in number to the defenders (E_2), the probability of the attackers winning is 0.5.
— For $E_1 < E_2$, the probability P of attackers winning is :

$$P = \frac{E_1}{2 \times E_2}$$

— For $E_1 > E_2$, the probability P of attackers winning is :

$$P = \frac{E_1}{E_2} - 0.5$$

Survivors :
— Attackers who win have a probability P of surviving the battle.
— If humans are converted, each human has a probability P of surviving the conversion process.
— Defenders who lose have a probability $1 - P$ of surviving.

2.4 End of the Game

The game concludes under one of the following conditions :

— **Time Limit :** If the game reaches its maximum duration, the species with the larger population is declared the winner.
— **Species Elimination :** If one species is entirely eliminated, the remaining species wins.
— **Grid Control :** The game may end early if one species controls the majority of the grid.

The dominant species at the end of the game is declared the winner, emphasizing strategic planning and adaptability throughout the game.

3 Search Algorithm

The AI uses the **Minimax Algorithm with Alpha-Beta Pruning** to make optimal decisions during the game. This search algorithm evaluates all possible moves and their outcomes, considering both the AI's actions and the opponent's counteractions, to determine the best course of action.

3.1 Mini-Max Algorithm

The game is represented as a tree of states, where :

— **Maximizing Nodes** : Represent the AI's turn, aiming to maximize the evaluation score.

— **Minimizing Nodes** : Represent the opponent's turn, aiming to minimize the evaluation score for the AI.

The utility value of a state $V(s)$ is calculated as :

$$V(s) = \begin{cases} \max_{a \in A(s)} V(s') & \text{if AI's turn} \\ \min_{a \in A(s)} V(s') & \text{if Opponent's turn} \end{cases}$$

Where $A(s)$ represents the set of all possible actions from state s, and s' is the resulting state after applying action a.

The evaluation function, implemented in `evaluate_game_state` (from `heuristics.py`), assigns a score to each state based on metrics such as :

— **Number of creatures converted (resource gain).**
— **Opponent's strength reduction (attack power).**
— **Strategic positioning.**

3.2 Alpha-Beta Pruning

Alpha-Beta Pruning optimizes the Mini-Max Algorithm by reducing the number of nodes explored by skipping branches that cannot influence the final decision. This technique ensures faster decision-making while maintaining accuracy. We have two variables :

— **Alpha (α)** : The best score achievable for the maximizing player so far.
— **Beta (β)** : The best score achievable for the minimizing player so far.

Pruning occurs when :

— The value of a maximizing node exceeds β (further exploration is unnecessary as it won't affect the minimizing player's decision).
— The value of a minimizing node falls below α (further exploration is unnecessary as it won't affect the maximizing player's decision).

$$\text{If } \beta \leq \alpha, \text{ prune the branch.}$$

3.3 Depth Limitation

To manage computational complexity, the decision tree is explored up to a predefined depth. For this implementation, the depth is fixed at 3, an odd value chosen to ensure the algorithm evaluates the game state from the AI's perspective (the maximizing player) at the end of the search. By ending the search on the AI's turn, the evaluation reflects the AI's ability to optimize its position rather than being influenced by the opponent's move, which would occur with an

even depth. Terminal states, whether reached due to depth limitation or game-ending conditions, are evaluated using the heuristic function. Limiting depth helps ensure that decisions are computed within the 2-second time constraint.

This algorithm forms the backbone of the AI's decision-making, balancing strategic depth with computational constraints to make intelligent, real-time moves. The choice of a depth of 3 allows the AI to strike a balance between computational efficiency and strategic foresight, enabling it to anticipate the opponent's response while planning its counter-move effectively.

4 Evaluation Function

The evaluation function in this project assesses the desirability of game states by calculating a weighted score based on several dynamic metrics. These metrics reflect the AI's ability to dominate the game by converting humans, defeating opponents, and positioning itself strategically. The formula for the heuristic score is :

$$\begin{aligned}\text{Evaluation Score} = 10 \cdot (U_{\text{ai}} - U_{\text{op}}) + w_{\text{resource}} \cdot (\text{Resource Gain}_{\text{ai}} - \text{Resource Gain}_{\text{op}}) + \\ w_{\text{attack}} \cdot (\text{Attack Power}_{\text{ai}}) + \text{Strategic Positioning} + \text{Distance Penalty} + \text{Random Bias}\end{aligned}$$

4.1 Metrics for Game State Evaluation

4.1.1 Resource Gain

The Resource Gain metric evaluates the potential benefit of converting humans within a specific range around the AI's current position. This metric helps the AI prioritize areas where conversion is both feasible and efficient.

Formula :

$$\text{Efficiency}(i,j) = \frac{H(i,j)}{d((x,y),(i,j))}$$

Where :
- $H(i,j)$: Number of humans at position (i,j).
- $d((x,y),(i,j))$: Chebyshev distance between the AI's position (x,y) and target position (i,j) :

$$d((x,y),(i,j)) = \max(|x-i|, |y-j|)$$

4.1.2 Attack Potential

The Attack Potential metric evaluates the AI's ability to eliminate opponents (vampires or werewolves) within a specific range. This evaluation helps the AI prioritize targets based on

Algorithm 1 Resource Gain Calculation Algorithm

Require: Current position (x, y), number of units U, human positions H, range limit R
Ensure: Maximum resource gain G
$G \leftarrow 0$ *(Initialize maximum resource gain)*
for each position (i, j) in the range $(x - R, y - R)$ to $(x + R, y + R)$ **do**
  Compute $d \leftarrow \max(|x - i|, |y - j|)$ *(Chebyshev distance)*
  if $H(i, j) \leq U$ **then**
    (Enough units to convert humans) Compute $E(i, j) \leftarrow \frac{H(i,j)}{d}$ *(Efficiency calculation)*
    $G \leftarrow \max(G, E(i, j))$ *(Update maximum resource gain)*
Output : Return G

feasibility and strategic value.

Formula :

$$\text{Efficiency}(i, j) = \frac{E(i, j)}{d((x, y), (i, j))}$$

Where :
— $E(i, j)$: Number of enemies at position (i, j).
— $d((x, y), (i, j))$: Chebyshev distance between the AI's position (x, y) and enemy position (i, j) :

$$d((x, y), (i, j)) = \max(|x - i|, |y - j|)$$

Algorithm 2 Attack Potential Calculation Algorithm

Require: Current position (x, y), number of units U, enemy positions E, range limit R
Ensure: Maximum attack potential P
$P \leftarrow 0$ *(Initialize maximum potential)*
for each position (i, j) in the range $(x - R, y - R)$ to $(x + R, y + R)$ **do**
  Compute $d \leftarrow \max(|x - i|, |y - j|)$ *(Chebyshev distance)*
  if $E(i, j) < U$ **then**
    (Enough units to defeat the enemy group) Compute $A(i, j) \leftarrow \frac{E(i,j)}{d}$ *(Efficiency calculation)* $P \leftarrow \max(P, A(i, j))$ *(Update maximum attack potential)*
Output : Return P

4.1.3 Strategic Positioning

The Strategic Positioning metric evaluates how favorable the AI's position is relative to its opponent on the grid. This score encourages the AI to maintain a balance between proximity (being close enough to attack) and safety (avoiding overly close positions that may lead to vulnerability).

Formula :

$$S = \begin{cases} \frac{C}{d+1}, & \text{if } d \leq R \\ -P, & \text{if } d > R \end{cases}$$

Where :
— d : Chebyshev distance between AI and opponent.
— C : Bonus constant for proximity.
— P : Penalty for being outside the optimal range R.

Algorithm 3 Strategic Positioning Algorithm

Require: AI position (x_{ai}, y_{ai}), opponent position (x_{op}, y_{op}), optimal range R
Ensure: Strategic positioning score S
Calculate Distance :
Compute $d \leftarrow \max(|x_{ai} - x_{op}|, |y_{ai} - y_{op}|)$ *(Chebyshev distance)*
Compute Score :
if $d \leq R$ **then**
  $S \leftarrow \frac{C}{d+1}$ *(Score based on proximity)*
else
  $S \leftarrow -P$ *(Penalty for being out of range)*
Output : Return S

4.1.4 Unit Advantage

The Unit Advantage metric calculates the difference in the number of units (vampires or werewolves) between the AI and the opponent.

Formula :

$$\text{Advantage Score} = w_{\text{units}} \cdot (U_{\text{ai}} - U_{\text{op}})$$

Where :
— U_{ai} : Total units of the AI.
— U_{op} : Total units of the opponent.
— w_{units} : Weight for unit advantage (default : 10).

4.1.5 Dynamic Weighting

The game dynamically adjusts the importance of resource gain and attack power based on the game's progress :

— **Early Game (Progress Ratio** < 0.5**) :** Higher emphasis on attacking (w_{attack} is higher).

Algorithm 4 Advantage in Unit Count Algorithm

Require: AI units U_{ai}, opponent units U_{op}, weight w_{units}
Ensure: Advantage score
1: **Compute Unit Difference :**
2: Unit Difference $\leftarrow U_{ai} - U_{op}$
3: **Scale Unit Difference :**
4: Advantage Score $\leftarrow w_{units} \cdot$ Unit Difference
5: **Output :** Return Advantage Score

— **Late Game (Progress Ratio ≥ 0.5) :** Higher emphasis on converting humans (w_{resource} is higher).

Formula :

$$\text{Progress Ratio} = \frac{U_{\text{ai}} + U_{\text{op}}}{U_{\text{ai}} + U_{\text{op}} + H_{\text{total}}}$$

H_{total} : Total number of humans on the grid.

Weight Adjustment :

$$w_{\text{attack}} = \begin{cases} \text{High}, & \text{if Progress Ratio} < 0.5 \\ \text{Low}, & \text{otherwise} \end{cases}$$

$$w_{\text{resource}} = \begin{cases} \text{Low}, & \text{if Progress Ratio} < 0.5 \\ \text{High}, & \text{otherwise} \end{cases}$$

Algorithm 5 Dynamic Weighting Algorithm

Require: AI units U_{ai}, opponent units U_{op}, total humans H_{total}
Ensure: Weights w_{attack} and $w_{resource}$
1: **Compute Progress Ratio :**
2: Progress Ratio $\leftarrow \frac{U_{ai}+U_{op}}{U_{ai}+U_{op}+H_{total}}$
3: **Adjust Weights :**
4: **if** Progress Ratio < 0.5 **then**
5: $w_{attack} \leftarrow$ High *(Prioritize attacking)*
6: $w_{resource} \leftarrow$ Low
7: **else**
8: $w_{attack} \leftarrow$ Low
9: $w_{resource} \leftarrow$ High *(Prioritize converting humans)*
10: **Output :** Return w_{attack}, $w_{resource}$

4.1.6 Random Bias

A small random bias is added to the evaluation score to :

- Break ties between equally valued game states.
- Introduce slight variability, making the AI less predictable.

Formula :

$$\text{Random Bias} = \text{Uniform}(-0.1, 0.1)$$

Algorithm 6 Random Bias Algorithm

Ensure: Random bias value
1: **Generate Random Value :**
2: Random Bias $\leftarrow$ Uniform$(-0.1, 0.1)$ *(Random value between* -0.1 *and* 0.1*)*
3: **Output :** Return Random Bias

4.2 Subgroup Division Strategy

The **Subgroup Division Strategy** ensures that the AI can optimize its actions by splitting its units into smaller groups when beneficial. This approach enables the AI to perform multiple objectives simultaneously, such as engaging enemies and converting humans elsewhere.

However, this strategy was not implemented in the current version of the AI and remains a theoretical proposal. It has been included in this report as a potential enhancement for future iterations, as its implementation could improve the AI's ability to manage resources efficiently and adapt to dynamic game scenarios.

4.2.1 Conditions for Subgroup Division

— **Human Proximity :** Groups are divided to maximize their ability to reach and convert humans within range.

— **Size Constraints :** A subgroup must be large enough to handle nearby threats or convert humans effectively.

$$N_k \geq T_h + T_e$$

Where :
— N_k : Size of subgroup S_k.
— T_h : Minimum units required to convert nearby humans.
— T_e : Minimum units required to defend against nearby enemies.

— **Enemy Proximity :** The division accounts for enemy positions to ensure subgroups remain defensible. Subgroups with higher threats are assigned more units.

4.2.2 Division Process

1. **Analyzing the Spatial Distribution :** The AI examines the positions of humans and enemies on the grid, identifying clusters of humans or enemies within a specific range.
2. **Determining Subgroups :** Units are split into subgroups based on their ability to :
   — Convert nearby humans.
   — Engage nearby enemies.
3. **Prioritization :** Each subgroup is assigned a primary objective, such as converting humans or defending against threats.
4. **Merging Subgroups :** If the game state requires coordinated action (e.g., facing a large enemy group), subgroups may be merged back into a single group.

4.2.3 Evaluation for Subgroups

— Each subgroup is evaluated individually using the evaluation function :

$$\text{Score}(S_k) = 10 \cdot (U_{\text{ai}} - U_{\text{op}}) + w_{\text{resource}} \cdot (\text{Resource Gain}_{\text{ai}} - \text{Resource Gain}_{\text{op}}) + \dots$$

— **Alpha-Beta Pruning :** To minimize computational overhead, Alpha-Beta pruning is applied to focus on promising moves for each subgroup.

4.2.4 Algorithm for Subgroup Division

Algorithm 7 Subgroup Division Strategy

Require: Units N, human positions H, enemy positions E, range R
Ensure: Subgroups $S_1, S_2, \dots, S_k$

Analyze Spatial Distribution :
Identify clusters of humans and enemies within range R.
Evaluate Division Conditions :
if $N \geq T_h + T_e$ **and** $H_{\text{near}}(S_k) > 0$ **then**
    Divide units into subgroups $S_1, S_2, \dots, S_k$.
    for each subgroup S_k **do**
        Assign S_k to handle nearby humans or enemies.
else
    Keep units as a single group.
Evaluate Subgroups :
for each subgroup S_k **do**
    Compute $\text{Score}(S_k)$ using the evaluation function.
Output : Subgroups $S_1, S_2, \dots, S_k$ with assigned objectives.

5 Implementation

5.1 Architecture and Modular Design

The AI system for Vampires vs. Werewolves is implemented using a modular design that separates core functionalities into distinct components. This design ensures scalability, ease of maintenance, and reusability. The main components of the architecture include :

5.1.1 Game State Representation :

The `GameMap` class manages the current game state, tracking the positions and counts of humans, vampires, and werewolves. To ensure efficiency and clarity, the following attributes and structures are used :

Key Attributes :

- `humans_state`, `vamps_state`, `wolves_state` **:** Dictionaries that track the positions and counts of humans, vampires, and wolves on the grid. Each key represents a coordinate (x, y), and the associated value represents the count of entities at that position. For example :

$$\texttt{humans\_state} = \{(0,0) : 5, (2,3) : 8\}$$

  indicates 5 humans at position $(0, 0)$ and 8 humans at $(2, 3)$.
- `grid_size` **:** A NumPy array that stores the dimensions of the grid, defined as $[height, width]$.
- `is_vampire` **:** A Boolean flag indicating whether the player's species is vampires (`True`) or werewolves (`False`), determined by the player's initial position.
- `map_state` **:** A structured representation of all populated cells, formatted as a list of tuples :

$$\texttt{map\_state} = [(x, y, num\_humans, num\_vampires, num\_wolves)]$$

  For example :

$$\texttt{map\_state} = [(0,0,5,0,0), (1,1,0,3,0), (2,2,0,0,4)]$$

  This structure allows efficient updates and provides a clear overview of the game state.

Key Features :

— **Efficient Data Structures :** Positions and counts are stored in dictionaries for quick access and updates.

— **Real-Time Representation :** The game map supports updates from the server, ensuring real-time synchronization with game events.

Core Methods :

- `find_strongest_ally_position`: Identifies the position and count of the player's strongest group of allies.

- `calculate_possible_moves`: Generates valid moves based on the current game state and the number of creatures in the source cell.
- `resolve_move`: Simulates the result of a move, updating the game state by resolving interactions with humans or opponents at the target position.

5.1.2 Tree-Based Search for Decision-Making :

The `Tree` and `Node` classes enable the AI to construct and traverse a game tree, representing all potential game states up to a specified depth. Each node corresponds to a possible game state, while edges represent moves.

— **Tree Construction :** The `create_tree` method builds a decision tree starting from the current game state up to a specified depth. It alternates between maximizing layers (AI's turn) and minimizing layers (opponent's turn), simulating both players' strategies.
— **Node Management :** Each `Node` stores information about its game state, child nodes, parent node, and associated utility values.

5.1.3 Alpha-Beta Pruning :

The `AlphaBeta` class enhances the Mini-Max search by implementing Alpha-Beta pruning, an optimization that reduces the number of nodes explored in the game tree.

Core Steps :
— **Maximization and Minimization :** Alternates between maximizing the AI's utility and minimizing the opponent's utility.
— **Pruning Ineffective Branches :** Skips branches where further exploration cannot affect the decision outcome.
— **Move Selection :** The `alpha_beta_search` method identifies the optimal move by traversing the pruned game tree.

5.1.4 Heuristic Evaluation :

The heuristic function, implemented in `evaluate_game_state` (from `heuristics.py`), calculates the utility of a game state based on weighted metrics, ensuring the AI aligns short-term actions with long-term objectives.

Key Metrics :

— **Advantage Units :** Favors game states where the AI outnumbers its opponent.
— **Resource Gain :** Measures the potential for converting nearby humans.
— **Attack Power :** Measures the AI's ability to eliminate nearby enemies.
— **Strategic Positioning :** Rewards advantageous placements relative to opponents.
— **Distance Penalty :** Penalizes states where allied groups are dispersed.

— **Dynamic Weighting :** Adjusts coefficients based on game progression (e.g., prioritizing resource gain early and aggression later).

5.1.5 Server Communication :

The `ClientSocket` and `ServerInterface` classes manage real-time communication between the AI and the game server via a TCP connection. These components ensure synchronization, accurate game state updates, and move transmission.

ClientSocket :
— Handles sending and receiving commands to/from the server.
— Processes Commands :

1. **Initialization :**
   · `SET`: Grid dimensions.
   · `HUM`: Human positions.
   · `HME`: Initial position of the player's species.
   · `MAP`: Full initial game state.
2. **Updates :** `UPD` provides ongoing changes to the board.
3. **Move Transmission :** `MOV` sends the AI's move to the server (source, target, number of units).
4. **End Game :** `END` and `BYE` finalize the session.

ServerInterface :
— Parses server messages and updates the internal game state.
— Integrates with `ClientSocket` for real-time synchronization and move execution.

5.2 Execution Flow / Decision-Making Process

The AI's decision-making process follows a structured execution flow, ensuring efficient adaptation to dynamic game conditions. This process aligns with the modular architecture to maintain seamless integration between components.

5.2.1 Execution Flow :

1. **Initialization :**
   · The server provides the initial game state using commands like `SET`, `HUM`, `HME`, and `MAP`.
   · The `ServerInterface` and `GameMap` classes parse this data to create an accurate representation of the game board.
2. **Game State Updates :**
   · Ongoing updates (`UPD`) are processed to reflect changes in the positions and counts of humans, vampires, and werewolves.

3. **Tree Construction :**
   · A decision tree is constructed with the current game state as the root node.
   · The `Tree` and `Node` classes build the tree up to a specified depth, alternating between maximizing and minimizing layers.
   · Moves are generated using `calculate_possible_moves` in `map.py`, ensuring they adhere to game rules.
4. **Heuristic Evaluation :**
   · Each node in the decision tree is evaluated using the `evaluate_game_state` function.
   · Metrics such as resource gain, attack power, and strategic positioning are combined into a weighted score.
5. **Alpha-Beta Pruning :**
   · The `alpha_beta_search` method identifies the optimal move by pruning branches that cannot influence the outcome, ensuring computational efficiency.
6. **Move Transmission :**
   · The optimal move is sent to the server using the `send_mov` command, specifying source and target positions and the number of units to move.
7. **Random Battles Handling :**
   · If a move triggers a battle, the AI uses probabilistic outcomes (handled in `battle_humans` and `battle_enemy`) to update the game state.
8. **Iterative Improvement :**
   · After each move, the AI recalculates its strategy based on the updated game state, repeating the process dynamically.

6 Testing and Results

6.1 Testing Scenarios

For testing, we focused on validating the correctness of the AI's moves and its ability to interact with the game server. Full and comprehensive strategy testing will take place during the competition against other groups.

Verified Capabilities :

— **Connection to Server :** The AI reliably communicates with the server, processes updates, and submits moves.
— **Valid Moves :** All generated moves comply with game rules and reflect the game's dynamic state.

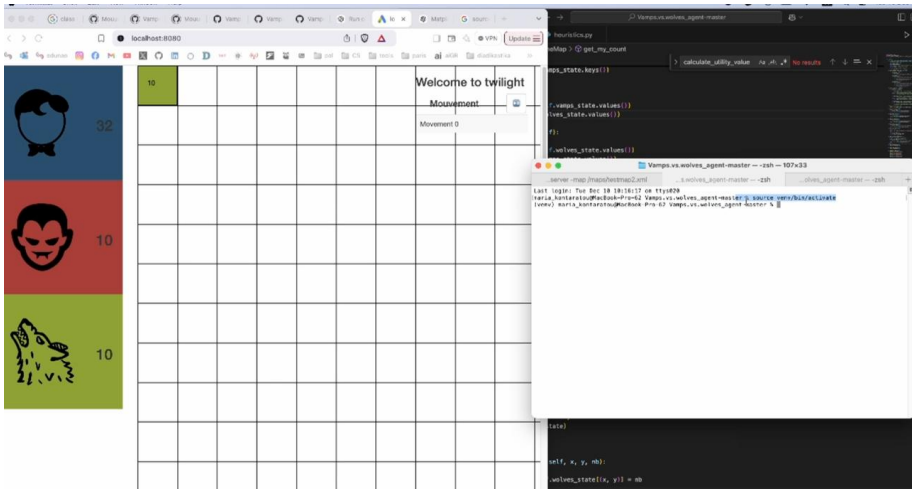

FIGURE 2 – Game interface and server connection demonstration

6.2 Observations

6.2.1 Strengths :

— **Adaptability :** The dynamic heuristic function allowed the AI to excel in varied game configurations, adjusting priorities effectively based on game state.
— **Efficiency :** Alpha-Beta pruning ensured that moves were computed within the **2-second time limit**, even for deeper game trees.
— **Real-Time Synchronization :** The modular design of the GameMap class provided robust and accurate updates, ensuring that decisions reflected the current game state.

6.2.2 Weaknesses :

— **Tree Depth Limitations :** Limited search depth caused occasional oversights of long-term opportunities or threats that could emerge from multi-step strategies.
— **Probabilistic Battles :** Probabilistic combat outcomes introduced variability that the AI could not fully predict or incorporate into its planning.
— **Late-Game Challenges :** As the grid becomes sparse, balancing resource acquisition, combat efficiency, and positioning exposed weaknesses in the heuristic's ability to adapt to endgame scenarios.

7 Limitations and Potential Next Steps

7.0.1 Limitations :

The AI implementation follows game rules well and makes effective decisions, but it still has some limitations :

Splitting Groups Not Implemented :

— Due to time constraints and complexity, the feature to split groups dynamically into subgroups for parallel actions was not implemented.

Computational Complexity :

— The search space grows exponentially with larger grids and deeper trees.

Simplified Heuristics :

— Current metrics lack the ability to fully account for multi-step strategies.

Randomness in Combat :

— Probabilistic battle outcomes introduce unpredictability that the AI cannot fully account for in its decision-making process.

7.0.2 Next Steps :

To further improve the AI's robustness, we recommend the following enhancements :

— **Reinforcement Learning :** Train the AI to learn advanced strategies via self-play, allowing it to improve decision-making through experience.
— **Monte Carlo Tree Search (MCTS) :** Incorporate stochastic search methods to handle uncertainty and improve decision-making in probabilistic scenarios.
— **Scalability Improvements :** Optimize the system for larger grids and deeper search trees to enhance performance in more complex configurations.

8 Conclusion

The Vampires vs. Werewolves project presented the implementation of an AI system designed for a dynamic, adversarial environment. Our AI uses a modular design, efficient game state representation, and advanced search algorithms like Minimax with Alpha-Beta Pruning to follow game rules and make quick decisions. The heuristic adapts to different game phases, balancing resource collection, attack strategies, and positioning. The AI shows strengths in adaptability, efficiency, and real-time updates from the server. However, it has limitations, such as shallow search depth, lack of group-splitting ability, and difficulties with late-game scenarios. Full performance testing during the competition will provide valuable insights into its effectiveness against other implementations. Future improvements include better scalability, adding Monte Carlo Tree Search (MCTS), and using reinforcement learning to boost strategy and flexibility. These updates will help the AI handle complex scenarios and expand its potential in competitive strategy games. This project shows the challenges of using AI in competitive games and lays the groundwork for improving strategy-based AI systems.