Univerzitet u Sarajevu
Elektrotehnički fakultet
Sarajevo

DSAI
DATA SCIENCE & ARTIFICIAL INTELLIGENCE

Game of The Amazons

Implementation Guide

Course: Artificial Intelligence

Professor: Izudin Džafić

Student: Emin Hadžiabdić

Academic Year: 2025/26

Student ID: 19960

Sarajevo, November 28th 2025

Contents

1 Main Goal of Project

The main goal is to build a complete Game of the Amazons application with a clean modular architecture, correct game rules, an AI opponent using minimax with alpha–beta pruning, and a polished GUI in natID with animations, sounds, multiple board sizes, difficulties, and languages.

2 Task 1: Core Board and Game State

- Define core data structures
    - Design a Board/Grid class using a 2D matrix to store empty tiles, queens, and arrows/blocked squares.
    - Define enums or constants for players (e.g., White, Black), piece types (queen, arrow), and a configuration for supported board sizes (6x6, 8x8, 10x10).
    - Implement helper methods like isInsideBoard, getTile, and setTile to safely access the matrix.
- Implement the GameState model
    - Create a GameState class that stores the board, current player, queen positions per player, arrow positions, move history, current difficulty, and selected board size.
    - Implement initialization for new games, including standard starting layouts for queens on 6x6, 8x8, and 10x10 boards.
    - Add functions to clone or copy GameState efficiently so that the AI can explore future states without corrupting the main game.

3 Task 2: Move Generation and Win Checking

- Implement line-of-sight ray casting
    - Create a generic rayCast function that, given a starting square and direction (dx, dy), walks step by step until a board edge, queen, or arrow is reached.
    - Use rayCast in 8 directions (horizontal, vertical, diagonal) to collect all reachable empty tiles from a queen's position.

- Generate legal queen and arrow moves
    - For each queen, generate all legal destination squares using the rayCast function and aggregate them into a list of queen moves for the current player.
    - Define a Move structure that contains: queen start position, queen destination, and arrow destination.
    - For each candidate queen move, simulate the queen at its new square and run rayCast again to generate all legal arrow targets, producing full (queen + arrow) moves.

- Validate and apply moves
    - Implement a function to validate a selected move: verify the chosen piece is the current player's queen, its destination is in the generated queen moves, and the arrow destination is in the generated arrow targets with unobstructed paths.
    - Implement applyMove(GameState, Move) that updates the board: move the queen, empty the starting tile, place an arrow tile, update queen lists and move history, and switch the current player.

- Implement win checker
    - After each move (or on request), generate all legal moves for the next player and check if the list is empty.
    - If there are no legal moves for the player to move, declare the opponent as the winner and mark the game state as finished to be used by the GUI and AI.

4 Task 3: Heuristic Evaluation and AI Search

- Implement heuristic evaluation components
    - Mobility: for each player, count the number of legal moves from the current GameState and define a mobility score as the difference between the players' move counts.
    - Territory / zone control: for each queen, run a flood-fill (BFS/DFS using queen-style reachability over empty tiles) and count reachable tiles; compute a territory score from the difference between players' controlled tiles.
    - Optional spatial influence: give bonuses for queens near the board center or controlling key zones and penalties for queens trapped near the edges with few moves.
- Combine heuristics into an evaluation function
    - Define an evaluate(GameState, player) function that combines mobility, territory, and spatial influence using chosen weights into a single numeric score where larger values mean a better position for the given player.
    - Ensure evaluation is efficient and side-effect-free so it can be safely used at minimax leaf nodes and possibly during move ordering.

- Implement minimax with alpha–beta pruning
    - Implement a recursive minimax(state, depth, alpha, beta, maximizingPlayer) that stops on depth limit or terminal positions found by the win checker and returns the evaluation score at leaf nodes.
    - At each node, generate all legal moves, apply each move to a copied state, recursively call minimax, update alpha/beta, and prune branches when alpha ≥ beta to improve performance.
- Add difficulty levels and engine interface
    - Map difficulty levels to search depth: Easy (1–2 plies), Medium (3–4), Hard (5+), with the exact depths chosen to be stable in performance.
    - Expose an engine API method like getBestMove(GameState, difficulty) that the GUI will call when it is AI's turn, returning the chosen Move.

5 Task 4: GUI Board, Interaction, and Animations

- Set up natID main view and layout
    - Create a MainView class that contains the AmazonsBoardCanvas (or similar) plus control elements such as buttons for "New Game", "Undo" (if implemented), and difficulty/board-size selection.
    - In the canvas's onDraw method, draw the grid for the current board size, tiles, queens, and arrows using the data from the GameState.
- Implement user interaction and highlighting
    - Handle mouse input in the GUI controller: first click selects a queen, second click selects a queen destination, third click selects the arrow destination, respecting the current phase of move selection.

- After selecting a queen, visually highlight all valid queen destination tiles; after selecting a destination, highlight all valid arrow target tiles, using data from the move generator.

- Add animations and sound effects
  - Implement smooth animations for moving a queen from its start to destination square and for the arrow “shot” to its target.
  - Add effects such as an impact/explosion animation when an arrow lands and play appropriate sound effects for queen moves, arrow shots, and victory/defeat.

- Integrate AI turns into the GUI
  - In human vs AI mode, detect when it is the AI player’s turn, call the engine’s getBestMove with the current difficulty and apply the returned move, optionally showing a small “AI thinking…” indicator.
  - After each move (human or AI), refresh the board, check for game end via the win checker, and if the game is over, display a result dialog with the winner.

6 Task 5: Options, Localization, and Final Polish

- Implement board size and difficulty options
  - Provide UI controls (menu or buttons) to choose board size: Small (6x6), Medium (8x8), Big (10x10); restarting the game with the selected size and resetting GameState.
  - Provide UI controls for difficulty: Easy, Medium, Hard, which map to the search depths configured in the AI engine.

- Implement language support (Bosnian and English)
    - Extract all user-facing text (labels, buttons, messages, dialogs) into language resource files or a simple dictionary.
    - Add a setting to toggle between Bosnian and English and reload or update GUI strings accordingly without changing the core logic.
- Testing and refinement
    - Create small test positions to verify ray casting, move generation, move validation, and win detection for correctness on all supported board sizes.
    - Playtest against the AI on each difficulty to ensure the AI always makes legal moves and exhibits stronger play at higher levels; adjust heuristic weights and search depth as needed.
    - Check the full GUI (animations, sounds, language switching, options) to ensure everything works smoothly together and the application feels polished and responsive.