Faculty of Technical Sciences, University of Novi Sad

Computing and Control Department

Chair of Automatic Control

Self-Adaptive & Learning Algorithms

Ultimate Tic-Tac-Toe

Students:

Filip Goldberger RA182/2022

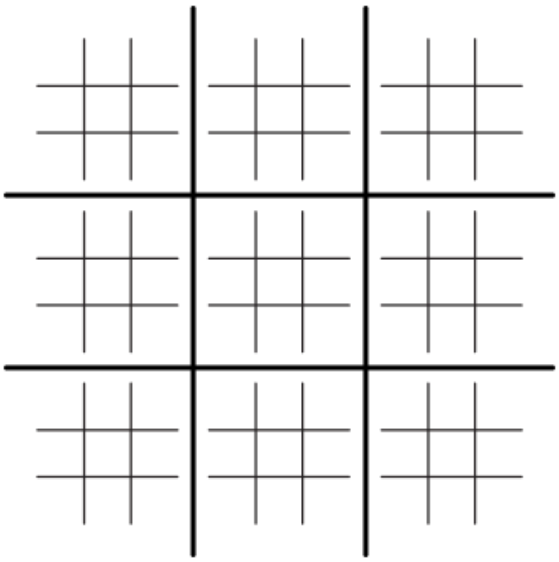
Minja Drakul RA58/2022

Srđan Slokar RA149/2022

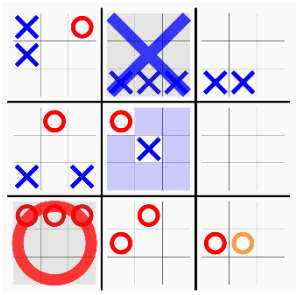
Nemanja Mitrović RA92/2022

Contents

Rules of the game



- **The global board:** The game is played on a massive 3x3 grid.
- **The local boards:** Each of the 9 squares on the Global Board contains a smaller, standard 3x3 Tic-Tac-Toe board.
- **Total cells:** There are **9 x 9 = 81** playable cells



The golden rule: The position of the move played on a *Local Board* determines the *Local Board* where the next player must play.

- ***Example****:* If Player X places a mark in the **top-right cell** of a Local Board, Player O **must** make their next move in the **top-right Local Board** of the Global grid.

Winning Conditions:

1. Winning a Local Board: To win a Local Board, a player must align 3 marks horizontally, vertically, or diagonally within that specific 3x3 grid.
   1.1. Once a Local Board is won, it is considered "closed." Visually, the entire 3x3 grid is marked with a large **X** or **O**
2. Winning **the global game**: To win the entire game, a player must win three **Local Boards** in a row (horizontally, vertically, or diagonally) on the Global Board.

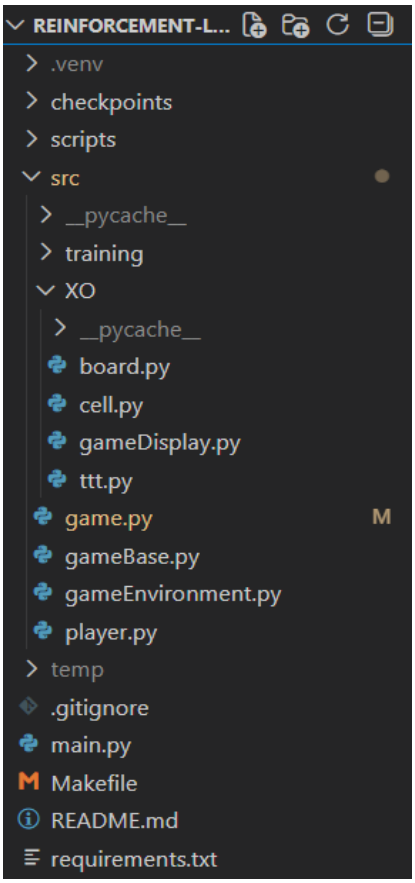
Special Rules & Edge Cases

When a player is sent to a Local Board that is already:

1. **Won:** The board has already been claimed by X or O.
2. **Full (Draw):** All 9 cells are filled, but neither player has won.

The current player may play their move on **any available cell** on any other non-terminated Local Board

Structure of project



XO folder:

- **board.py** implements the Ultimate Tic-Tac-Toe board state(global board), legal-move logic, play/clone/reset, and winner detection
- **cell.py** defines cell values, player constants, and small helpers for toggling or mapping players.
- **gameDisplay.py** handles Pygame rendering, input-to-board mapping, highlighting, and exposes methods for clicks and screen updates.
- **ttt.py** implemented the local board, check local win in 3x3 table.

- **game.py:** High-level game controller that coordinates the board, display, and players; runs the main loop, processes input events, and advances turns
- **gameBase.py:** Core game logic wrapper providing shared utilities and state (board initialization, current player tracking, win checks) used by higher-level classes.
- **gameEnvironment.py:** Environment API used for training/MCTS—encodes/decodes state, applies actions via step(), returns legal moves, terminal flag, and reward values.
- **player.py:** Player abstraction for human vs agent, holds player type (HUMAN/AGENT), symbol, and agent-related inference methods (model loading, move selection) when applicable.
- **checkpoints/** it stores saved weight files (model_{iteration}.pt)
- **main.py:** it initializes the game/UI or training flow, parses user commands (train/play/exit), creates temp runtime folders, and wires the game loop with player/agent logic and the Pygame display.

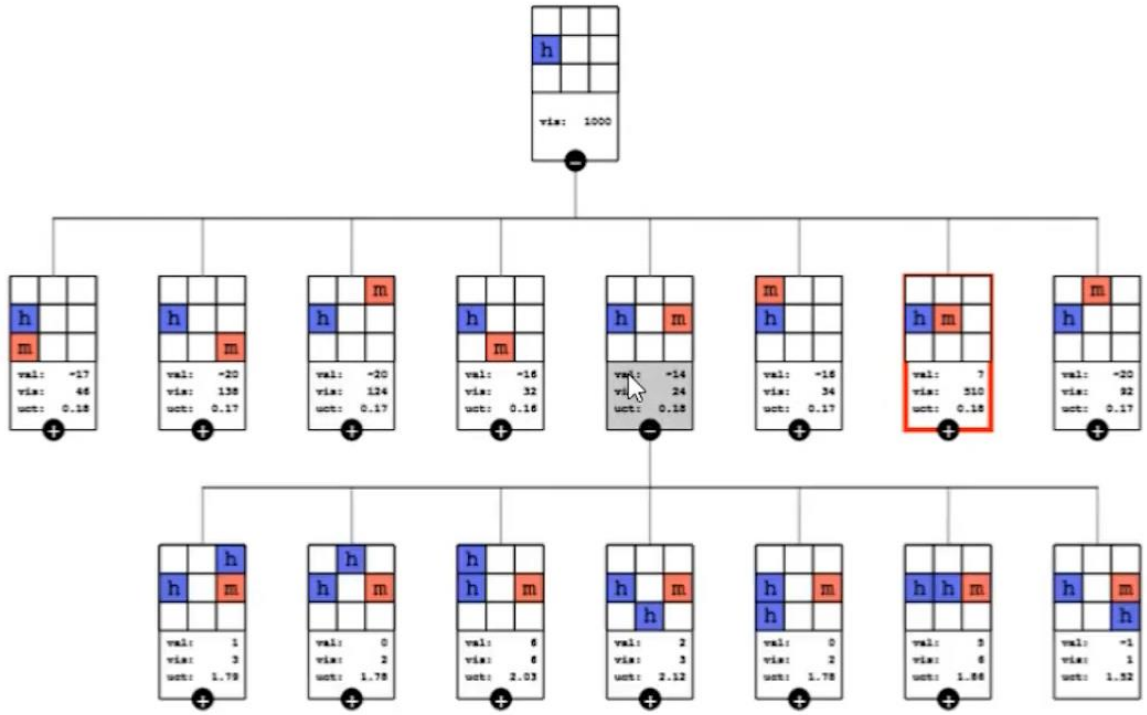
Used algorithms

Monte Carlo Tree Search (MCTS)

How it works: It builds a search tree by simulating random playouts from the current state.

The 4 Phases:

1. **Selection:** Traversing the existing tree to find the most promising node
2. **Expansion:** Adding potential next moves to the tree
3. **Simulation:** Playing out the game to the end (or evaluating it) to see who wins
4. **Backpropagation:** Updating the win/loss statistics up the tree



Picture 1: Example of MCTS on classic 3x3 board

The AlphaZero Approach

We utilize the AlphaZero methodology to supercharge standard MCTS. Instead of using random simulations to evaluate a board state, we use a Deep Neural Network

The Neural Network: It takes the current board state as input and outputs two things:

1. **Policy (*p*):** Probabilities suggesting which move is likely the best.
2. **Value (*v*):** An estimation of the current player's chance of winning from this position

Proximal Policy Optimization (PPO)
PPO is the optimization algorithm used to train our Neural Network. It is a Policy Gradient method known for its stability and efficiency.

Reinforcement learning concept

MDP environment

- implemented by *GameEnvironment* . It stores board, current_player, next_board_pos and exposes reset(), step(action), legal_actions(), is_terminal(), clone() so MCTS and training treat the game as an episodic MDP.
- **States**: a full board snapshot + current player + next_board_pos; states are encoded for the network by encode_state(...) in src/encoding.py into a 4×9×9 tensor (X plane, O plane, next-board mask, player-turn plane)
- **Actions:** represented as flat indices 0–80 (index_to_action / action_to_index in src/training/encoding.py) corresponding to the 9×9 grid; legal_action_mask(...) provides the set A(s) used by MCTS and action selection
- **Rewards:** sparse terminal rewards provided by env.step(): **+1 for win**, **−1 for loss**, **0 for draw/intermediate**; these z values are used as value targets and backed up through the MCTS tree
- **MCTS:** implemented in src/mcts.py. Each simulation uses the model (src/model.py) to get policy priors and a value; nodes store prior, visit_count, and value_sum; selection uses PUCT (prior-weighted exploration), expansion uses NN policy, simulation uses NN value, and backup updates visit counts/value sums.

Installation & Usage

Minimal runtime packages:

- python>=3.9
- numpy
- pygame
- torch
- MinGW-w64 (for makefile)

Usage:

🛠️ Installation & Usage

1. **Run the game** (automatically creates `.venv` and installs requirements):

```
make
```

Playing the game:

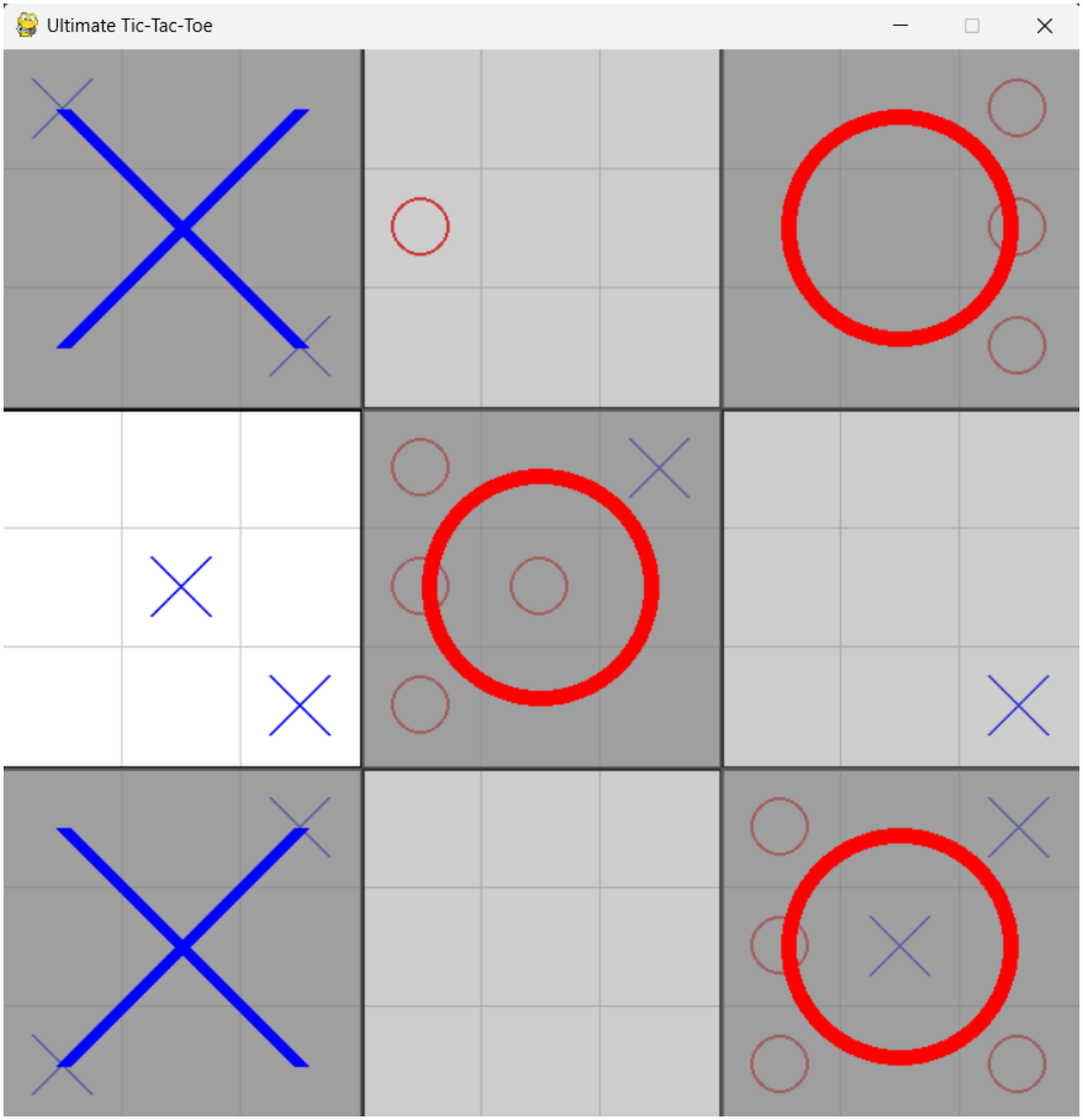
```
.venv\Scripts\python.exe main.py
pygame 2.6.1 (SDL 2.28.4, Python 3.12.6)
Hello from the pygame community. https://www.pygame.org/contribute.html

=== Ultimate Tic-Tac-Toe ===
Enter command (train/play/exit):
```

Playing mode:

```
=== Ultimate Tic-Tac-Toe ===
Enter command (train/play/exit): play

--- Play Menu ---
1. Against another Player
2. Against an Agent
3. Back to Main Menu
Select an option (1-3): 2
```



End of game:

```
--- Play Menu ---
1. Against another Player
2. Against an Agent
3. Back to Main Menu
Select an option (1-3): 2
🤖 Agent match started...
Winner of the game is player:  CellValues.X
```