LINFO1361: Artificial Intelligence
Assignment 2: Adversarial Search (Fenix)

Alice Burlats, Achille Morenville, Harold Kiossou, Amaury Fierens, Eric Piette
March 2025

Guidelines

- The report and your agent must be submitted on **Wednesday, 9th April 2025, 18h00**. Your contest agent implementation can to be submitted up to the **Friday, 18th April 2025, 18h00**.
- *No delay* will be tolerated.
- *Document* your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Copying code or answers from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated.
- Source code shall be submitted on the online *INGInious* system. Only programs submitted via this procedure will be graded. No program sent by email will be accepted.
- Respect carefully the *specifications* given for your program (arguments, input/output format, etc.) as the program testing system is *fully automated*.
- The answer to questions must be given by filling in the latex template provided. The final file must be submitted on *Moodle*. No report sent by email will be accepted.
- Nothing must be modified in the template except your answer that you insert in the *answer* environments as well as your names and your group number. The names are provided through the command *students* while the command *group* is used for the group number. The dimensions of *answer* fields *must not* be modified either. Any other changes to the file will *invalidate* your submission.

During the lectures, you have explored adversarial search algorithms–including MiniMax, Alpha-Beta, and Monte-Carlo Tree Search (MCTS). In this project, you will implement and improve these algorithms to create an AI agent capable of playing the game Fenix, a strategic board game. A detailed description of the game rules is provided below and you may watch this short video for a quick overview of the game.

1 Game Rules

Fenix is a strategic board game for two players, red and black, played on a 7×8 grid. Each player controls an army of pieces with the objective of capturing the opponent's king while ensuring that no new king can be created during the next turn.

Figure 1: Fenix Game Cover

1.1 Goal of the Game

The primary objective is to capture the opponent's king. Additionally, you must ensure that your opponent cannot resurrect their king on their next turn. If a player loses their king and cannot resurrect it, they lose the game.

1.2 Setup

The game begins with each player placing 21 pieces in opposite corners of the board, with the red player moves first. The game is divided into two phases: the setup phase and the classic game phase.

During the setup phase, which lasts for the first 10 turns, players take turns creating one king and three generals from their pieces. A soldier is a single piece, a general is a stack of two pieces, and a king is a stack of three pieces. To create a general or king, a player must place one of their soldiers on top of an orthogonally adjacent soldier (to create a general) or on top of a general (to create a king). After the setup phase, each player will have one king, three generals, and 12 soldiers. A valid setup is shown in Figure 2. The game then transitions to the classic phase, where players take turns moving their pieces and capturing the opponent's pieces.

1.3 Movements

Players can move one piece per turn—either a soldier, a general, or a king. Each type of piece moves differently. A soldier can move one square in an orthogonal direction (up, down, left, or right) and must land on an empty square. A general moves like a rook in chess: it can traverse any number of empty squares in an orthogonal direction, ending on an empty square. A king moves like the chess king: it can move one square in any direction (orthogonal or diagonal) and must also land on an empty square. Figure 3 provides a visual representation of the possible movements for each piece.

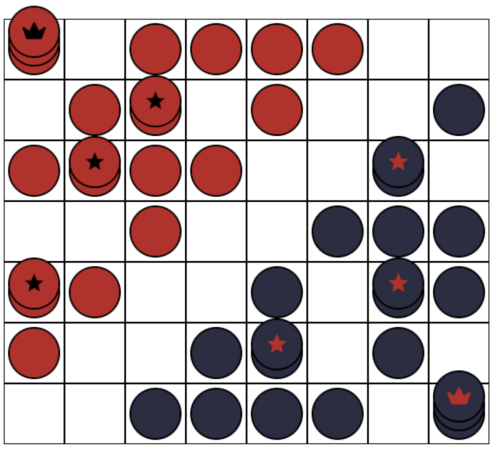

Figure 2: Example of a valid state after the setup phase. A stack with a star on top represents a general, and a stack with a crown on top represents a king.

1.4 Captures

Capturing the opponent's pieces is a key mechanic in Fenix. Captures are mandatory, and players must always choose the option that maximizes the number of opponent pieces captured. A soldier or king can capture an opponent's piece by jumping over it and landing on the empty square immediately beyond. Only one piece may be captured per jump. A general can also capture by jumping over an opponent's piece; however, it is allowed to jump over vacant spaces both before and after the opponent's piece, as long as the move remains in a straight line. Only one piece can be captured per jump.

If a piece can continue capturing after its initial jump, it must do so. Figure 4 illustrates an example of a chain of captures, where the red king captures multiple black pieces in a single turn by successive jumps. Note that pieces captured during a turn become obstacles and cannot be jumped over again. Moreover, if multiple capture options are available, the player must choose the one that results in the capture of the highest number of opponent pieces (a king counts as 3 pieces, a general as 2, and a soldier as 1).

1.5 Special Rules

If a general is captured, the player may use their next turn to create a new general by stacking a soldier on top of an adjacent soldier. This action counts as a full turn. If the player does not create a general on their next turn, they lose the opportunity to do so. Similarly, if the king is captured, the player may use their next turn to resurrect the king by stacking a soldier on top of an adjacent general. The king is dead? Long live the King! This action also counts as a full turn. If the player does not resurrect the king on their next turn, they lose the game.

Captures take precedence over other actions. If a player has the opportunity to capture an opponent's piece, they must do so, even if it prevents them from creating a general or resurrecting their king.

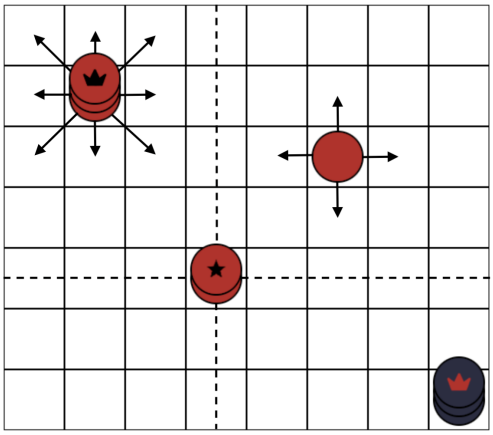

Figure 3: Possible movements for each type of piece.

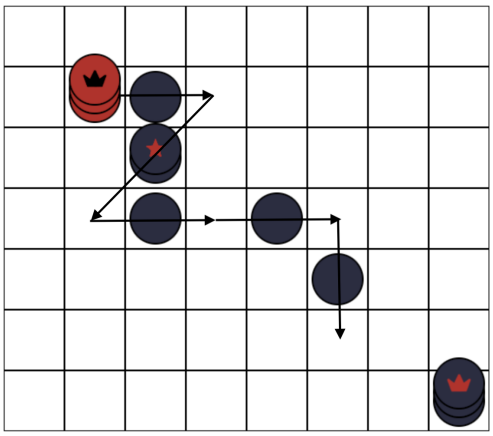

Figure 4: Example of a chain of captures. The red king can capture multiple black pieces in a single turn.

2 Provided Code

The provided codebase includes the implementation of the Fenix game, files for developing your AI agents, and testing them against each other.

Note: The provided codebase is a starting point for developing your AI agents. You are free to modify, extend, or refactor the code as needed to implement your agent's logic. However, the core game logic and interface should remain consistent with the provided structure to ensure compatibility with the testing and evaluation scripts. **All tests will be performed using the provided codebase**. Most notably, the action returned by your agent should be one of those listed by the state's `actions()` method provided in the `fenix.py` file. An invalid action will result in a loss for your agent.

Overview

Game Logic

The core game logic is implemented in the `fenix.py` file. The `FenixState` class encapsulates the game state and provides six key methods that follow the interface described in the reference book[1]:

- `__init__()`: Initializes the game state.
- `to_move()`: Returns the player who is to move in the current state.
- `actions()`: Returns a list of all legal actions available in the current state.
- `result(action)`: Applies a given action to the current state and returns the resulting state.
- `is_terminal()`: Checks whether the current state is a terminal state (i.e., the game has ended).
- `utility(player)`: Computes the utility of the current state for a specified player.

The game state representation is described in detail in the documentation of the `fenix.py` file.

Agents

The `agent.py` file defines the interface for implementing an AI agent.

To create your agent, extend the `Agent` class and implement the `act(state, remaining_time)` method. This method receives the current game state and the remaining time (in seconds) as inputs, and it should return the action to be taken by the agent. The returned action must be one of those listed by the state's `actions()` method. Note that if an agent exceeds the allotted time, it loses the game; thus, the `remaining_time` parameter is critical for managing your agent's time.

A basic random agent is provided in `random_agent.py`, which you can use as a starting point. This random agent selects moves uniformly at random from the list of legal actions, serving as a trivial baseline for testing and comparison.

Game Managers

The codebase includes two additional files to simplify testing and evaluation:

- `game_manager.py`: This file is used to run games between agents. It allows you to test your agents against each other or against the random agent, and it supports displaying game progress in the terminal for debugging and analysis.
- `visual_game_manager.py`: This file provides a graphical interface for visualizing and interacting with the game. You can use it to play against your agent, watch two agents compete, or simply observe the game state in a more interactive manner.

The documentation within these files provides all the necessary information for their use.

[1] Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach. pearson.

Note: Only the `game_manager.py` should be used for testing and evaluation. The `visual_game_manager.py` is provided for visualization purposes only.

Installations and Requirements

To run the codebase, ensure you have Python 3 installed. Install the required dependencies by running the following command in the project directory:

```
pip install -r requirements.txt
```

3 Your Agent (20 points)

In this section, you will outline what is expected for the project. The overall goal of the project is to go beyond the basic adversarial search algorithms covered in the course and develop a more sophisticated AI agent for the game of Fenix. You are free to use any adversarial search algorithm or combination of algorithms to create your agent. You may improve upon Alpha-Beta or Monte-Carlo Tree Search (MCTS) by enhancing their evaluation functions, search strategies, or other components. Alternatively, you can explore entirely different algorithms or techniques not covered in the course. The goal is to create an agent that outperforms the baseline implementations through creativity, experimentation, and optimization.

If you find it challenging to decide where to start, here are some sources of inspiration:

- For the Alpha-Beta algorithm, you could try to improve the evaluation function, the search strategy, or the pruning techniques. There is a long list of possible improvements that are cited in the reference book[2] in the Chapters 6.2.3 to 6.3.4 and in the slides of the course. As an example, you could try to implement a transposition table, a quiescence search, or a killer move heuristic (those are just examples, do not limit yourself to those). You could also try to design a good evaluation function for the game of Fenix.
- For the MCTS algorithm, there is a long list of improvements that are described in the scientific literature. Especially in this survey paper[3]. As a starting point, you could try to improve the simulation policy, the tree policy, or the selection strategy.
- You could also try to use neural networks to improve the evaluation function of your agent. This is a very popular approach in the field of AI and could be very interesting to try.
- ...
- And many more possibilities! This list is not exhaustive, and you are free to try anything you want.

[2] Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach. pearson.

[3] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 4(1), 1-43.

3.1 Agent Description (5 points)

As a first step, provide a detailed explanation of the design of your agent. The important thing here is to describe what you have implemented and why you have chosen this particular approach. Explain all the tricks that you have coded and why you think they are good for the game of Fenix.

Question

Describe in detail the enhancements or new techniques you implemented in your agent. Focus on:

- The specific improvements you made to the baseline algorithms.
- Describe any new algorithms or techniques you experimented with for Fenix.
- Discuss any approaches you tried that did not work as expected.
- Provide a clear explanation of why your design is well suited to Fenix.

Important: Do not explain the basic Alpha-Beta or MCTS algorithms, as these were covered in detail during the course. Focus solely on your modifications, enhancements, or new techniques and how they differ from the baseline implementations. **I repeat, do not explain the basics of Alpha-Beta or MCTS. No points will be given for that.**

3.2 Experimental Validation (7 points)

To support your design choices and evaluate your agent's performance, you must conduct a series of experiments. Your experimental analysis should provide clear, quantitative evidence of the improvements made to your agent.

Question

Design and conduct experiments to validate the performance of your agent. Your evaluation should include:

- A small description of the experimental setup, including the number of games played, the opponents faced, and any other relevant details you consider important.
- A detailed table (or tables, or graphs) that tracks the performance evolution of the various versions of your agent against a chosen baseline. This table should include relevant metrics such as win rates, or other performance indicators. The choice of the baseline is up to you, but it should be justified.
- A comparative analysis of your final agent against a random agent, a baseline Alpha-Beta agent, a baseline MCTS agent, and any other agents you consider relevant.
- A comprehensive discussion of your experimental results, including relevant metrics such as win rates, average game length, or other performance indicators. Explain how these results support your design decisions and highlight any insights or trends observed.

3.3 Implementation (3 points)

Submit

Submit your agent on Inginious via this task before the **Wednesday, 9th April 2025, 18h00**. Automated testing will evaluate the correctness of your agent across various scenarios to ensure it functions as expected. At a minimum, your agent should be able to win a game against a random agent and succeed in other specific scenarios.

Important:

- You have to submit only **1** file for your agent. This is non-negotiable. However, you may put everything needed inside this file. For your final submission, your names should be written as comments in the code to help identify your group.
- Your agent should extend the `Agent` class provided in the codebase and implement the `act(state, remaining_time)` method and should be able to interact with the interface provided in the codebase.

3.4 Contest (5 points)

Now, it is time for blood and glory.

Submit

Once you have implemented your agent with all desired features, submit it on Inginious on this task before the **Friday, 18th April 2025, 18h00**. We will play a simple match with a random agent to ensure that your code runs without any problem.

Important:

- You have to submit only **1** file for your agent. This is non-negotiable. However, you may put everything needed inside this file. For your final submission, your names should be written as comments in the code to help identify your group.
- Your agent should extend the `Agent` class provided in the codebase and implement the `act(state, remaining_time)` method and should be able to interact with the interface provided in the codebase.

An extended deadline is provided for the contest agent if you wish to continue improving your agent after the project deadline. **This second deadline is not mandatory**. However, if your contest agent outperforms the agent submitted in the previous task, you will receive a bonus of 1 point.

The final ranking is determined by a rating system. At the beginning, all agents have the same rating. In each round, a matchmaking system pairs agents with similar ratings, and these ratings are updated based on match results. Once the ratings have stabilized, the leaderboard is divided into groups, agents in higher-performing groups will receive more points.

Each match lasts 5 minutes, and similar to chess, if a game state is repeated three times, or if 50 turns pass without a capture, the match is declared a draw. In addition, for each pair,

2 matches will be played so that each agent can play as red and black.

4 Deliverables

Below is a summary of the project deliverables:

Submit

- A report in PDF format, based on the provided LaTeX template, that includes your agent description and experimental validation. This report should be submitted on *Moodle*.
- Your agent implementation (`agent.py`) submitted on Inginious.
- Your contest agent implementation (`agent.py`) submitted on Inginious.