

Lab 2: Adversarial search

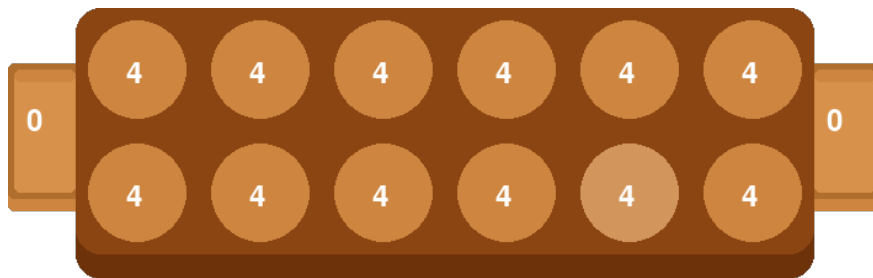
Adversarial search is a particular type of search, that applies to games with multiple players that are competing against each others. The main challenge lies in anticipating the opposing player's (or players') moves, with the assumption that they will try to thwart our progress to the best of their abilities.

This lab's objective is to use adversarial search algorithms to create a basic AI player for the two-player game of Kalah. You will implement in Python the MinMax algorithm, as well as an optimization that relies on Alpha-Beta pruning. By the end of the session, you should have a better understanding of the dynamics of adversarial search, and the limitations of the algorithms seen in class.

Section 1 presents the setting of the game we will work with, and **Section 2** the interface of the code you will have to complete. **Sections 3 and 4** contain the first questions and coding assignments.

1 The Game of Kalah

In this lab, we will play with the game of Kalah, which is a deterministic two-players game with perfect information. On their respective side of the board, each player has n pits, as well as one special pit on their right called “store” or “Kalah”. Initially, all pits (that are not stores) are filled with m seeds.



The standard game usually involves $n = 6$ pits and $m = 4$ seeds. With these dimensions, the game is usually hard to handle for the algorithm, which is why we will start by playing with smaller values of n and m , such as $n = m = 3$.

Rules of the game Players take turns moving. On your turn, you select one of your pits and distribute its seeds counterclockwise.

- **Extra turn:** If the last seed lands in your store, you get another turn.
- **Capture:** If the last seed lands in an empty pit on your side and the opposite pit is non-empty, you capture both the seed and the contents of the opposite pit and place them in your store.
- **Ending conditions:** The game ends when all pits on one player's side are empty. The remaining seeds on the opponent's side go into the store of the opponent. The player who gathered the most seeds in their store wins.

With the code provided, you can try the game against another human player, or against an AI that plays randomly.

2 Interface

You will implement the algorithms in file `ai.py`, where the skeleton is already provided. An example AI player is provided as class `Random`, which implements a player that randomly chooses an action among those available.

AI player Each AI player should extend class `AI`. It has to implement the static method `best_move`, which outputs the move chosen by the AI player. Its signature is as follows:

```
best_move(current_state: State, objective: Objective) -> int
```

where

- `current_state` is the state on which the AI has to make a decision;
- `objective` is either `Objective.MAX` or `Objective.MIN`, depending on whether the AI is player 0 (and has to maximize the score) or player 1 (and has to minimize the score).
- The return value, an integer, is the identifier of the pit that must be played.

Note that the method is static, but you can implement additional auxiliary methods if need be. More generally, only file `ai.py` should be modified by you (with a single exception, mentioned later).

Representation of a state Class `State` (in file `game.py`) represents a state of the game. It contains all the game logic that you need to implement the algorithm. Notably, you will need the following methods:

- `next_state(pit: int) -> State`: return the state that results of the given pit being played. The state the function is called on is untouched.
- `available_moves() -> List[int]`: return a list of integers, which are the pits that can be played by the current player. Note that a state where the game has ended has no available move, and the function returns the empty list.
- `check_victory() -> [None|int]`: return `None` if the game is not over, 0 (resp. 1) if player 0 (resp. player 1) won, or -1 if the game resulted in a tie.

- `copy()` → State: return a copy of the state.

In addition, you also have access to the following attribute, whose logic you will implement later:

- `score`: a property that returns the score of the state.

Settings In file `settings.py`, you can change a few parameters of the game, including the dimensions of the board. In file `main.py`, you can change the algorithms behind each AI player.

3 MinMax

3.1 First implementation

Recall that MinMax simulates all possible developments of the game in a depth-first search fashion. It then selects the move with the best outcome, assuming that the opponent plays optimally.

Tasks

- Implement MinMax using a recursive function. The function should return the utility of each available move, so that in function `best_move`, you can implement the logic to choose the move with the best utility for the current player.
- Add a counter for the number of expanded states to decide on a single move, and output its value in the command line.

Questions

1. Suppose that we allow 5 seconds for the AI player to choose its next move.
 - (a) What are the maximum dimensions of the board (number of pits and seeds) that allow this?
 - (b) How many nodes are then expanded?
2. For a small game, let the algorithm you implemented play against itself.
 - (a) Which player won?
 - (b) What does it mean? Do you think that, as player 1, you could beat MinMax as player 0?

3.2 Depth-bounded MinMax

As seen in the previous section, the basic MinMax algorithm can not scale up to the size of the original board. This is due to the complete exploration of the search tree that the algorithm performs. However, good moves can still be found by cutting the search earlier on each branch, up to a certain depth.

In practice, this is done by treating states at a fixed depth as terminal, and computing their heuristic value, or score. In the course's slides, this technique is detailed in the "Heuristic Alpha-Beta Search" section, but can readily be applied for MinMax as well.

Tasks

- Implement a function computing the score of each state, in class State, in file game.py. The more positive (resp. negative) the score is, the more the state should be favorable to player 0 (resp. player 1).
- Modify your implementation so that each branch is explored up to a constant depth. For now, we will set that maximum depth to 8.

Questions

1. What score function did you choose? Justify the intuition behind it.
2. Just like before, suppose that we allow 5 seconds for the AI player to choose its next move.
 - (a) What are the maximum dimensions of the board (number of pits and seeds) that allow this?
 - (b) How many nodes are then expanded?
3. Find some dimensions of the board such that, on your machine, the original (non-depth-bounded) algorithm takes 30 to 40 seconds to compute the first few moves.
 - (a) At which (minimum) value should the cutoff be set so that depth-bounded MinMax achieves comparable results to MinMax?
 - (b) How many nodes do the algorithms then expand, respectively? How much faster is the depth-bounded algorithm?
4. Suppose that we set the cutoff to depth 1. How is that search then called?

4 Alpha-Beta Pruning

With each player having (at most) n pits to choose from, the search tree has a branching factor of about n . Cutting a branch early can then result in the pruning of a significant amount of nodes.

Alpha-Beta pruning is a technique that extends the MinMax algorithm, and that prunes branches whose exploration can not influence the final decision anymore, given the information gained by the previous exploration of other branches.

The key idea is to maintain, during the exploration of a branch, two values, Alpha and Beta, which are the best values that the Max and Min players can guarantee on that branch, respectively. When exploring a branch, if at any time, we have that $\text{Alpha} \geq \text{Beta}$, then the exploration of the branch can be cut short.

Tasks

- Starting from the code that you wrote previously, implement MinMax with Alpha-Beta pruning in a separate class.
- Add a counter that outputs the number of branches that have been cut during search.

Questions

1. When the value for the depth cutoff is the same for both algorithms, how do the outputs of Alpha-Beta compare to the ones of MinMax? Justify.
2. Find some dimensions of the board such that, on your machine, depth-bounded MinMax takes 30 to 40 seconds to compute the first few moves.
 - (a) How long does Alpha-Beta take to perform the same moves?
 - (b) How many nodes are expanded?

Bonus questions

- How could the difficulty/strength of the AI should be tuned?
- Is it a good idea to simply limit the search time, and cut the search of the unbounded algorithm when it runs out of time?