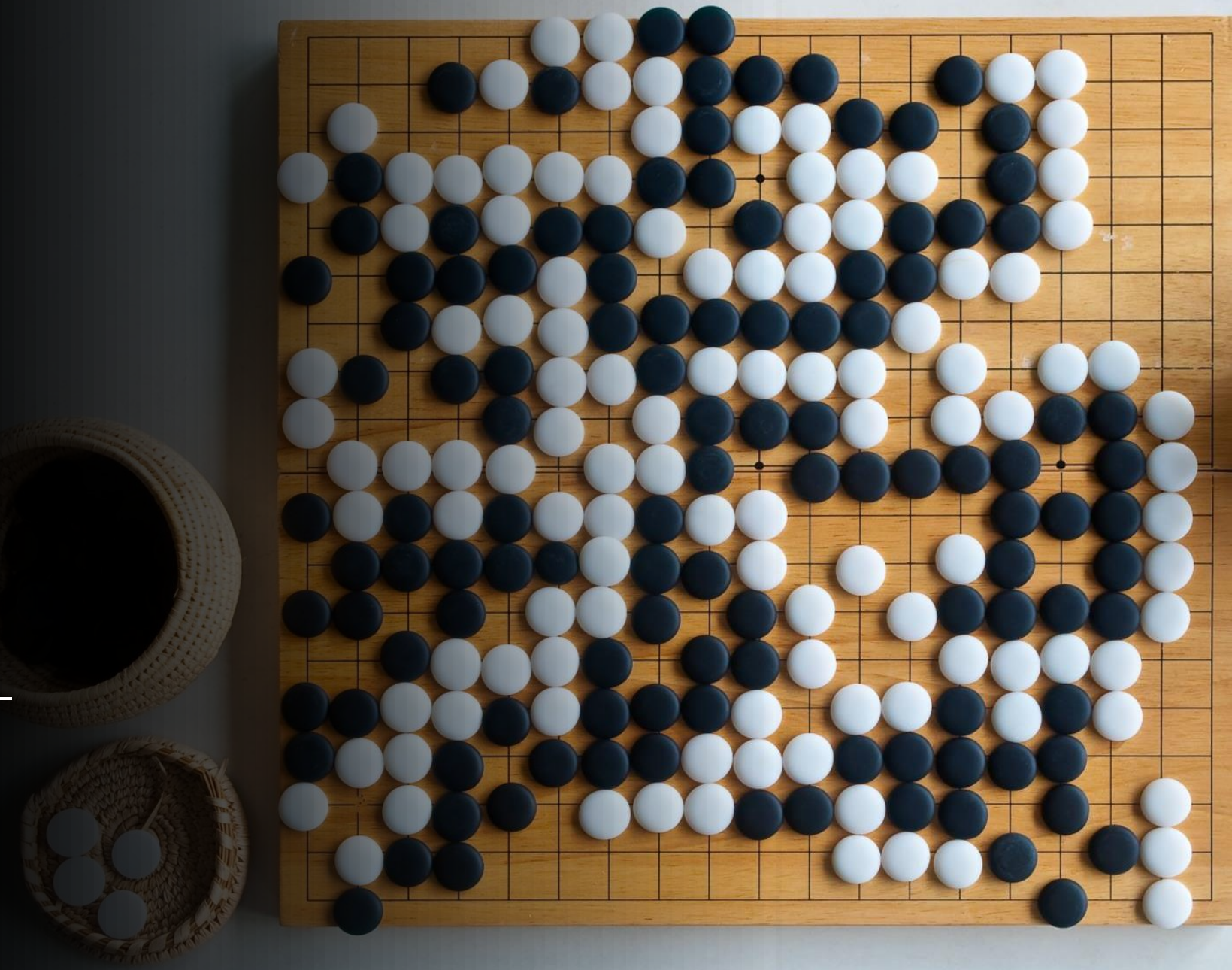
Adversarial Search Methods for Games – Jungle board game

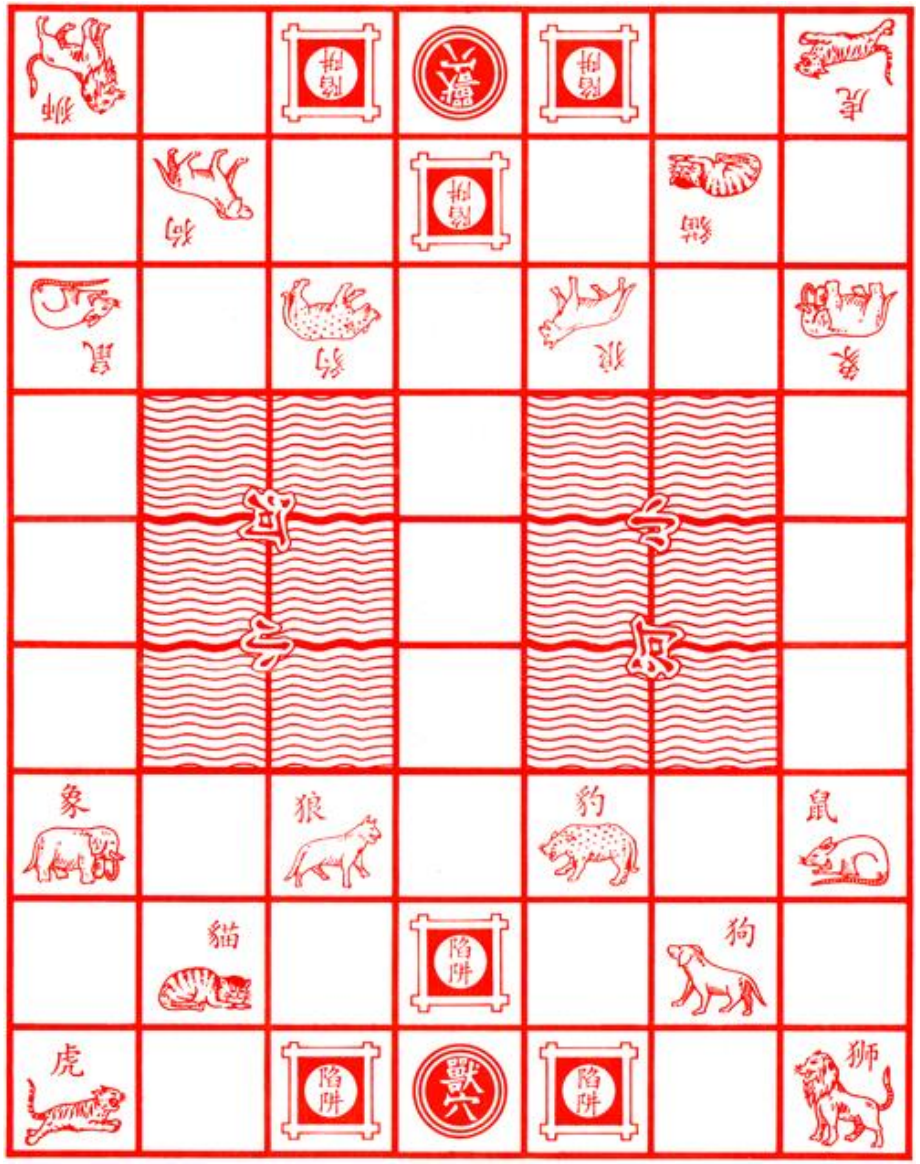
Claudio Savelli, Jakub Głatki



Jungle

Jungle is an old Indian game, also known as Jungle Chess (however, due to many differences it is not included in chess category).

The Jungle gameboard represents a jungle terrain with dens, traps "set" around dens, and rivers. Each player controls eight game pieces representing different animals of various rank. Stronger-ranked animals can capture ("eat") animals of weaker or equal rank. The player who is first to maneuver any one of their pieces into the opponent's den wins the game. An alternative way to win is to capture all the opponent's pieces.



Game implementation

1) Formulation search problem
2) Implementation of game for human players
3) Implementation of MinMax algorithm with alpha-beta cuts with basic evaluation function
4) Implementation of more advanced evaluation function
5) Testing evaluation functions playing against each other, with various MinMax depths

State representation

- Board [7x9]
- 8 animals for each player, with different power and set of moves
- Players
- Depth level for player
- Difficulty level for player

STATE RAPPRESENTATION:

THE BOARD:

It is played on 7 x 9 board which contains water fields, player lairs and traps. There are several special squares and areas of the Jungle board:

- Each player has one **den** (獸穴)
- Three **traps** (陷阱)
- Two **water areas or rivers** (小河) are in the center of the board

THE PIECES:

1. **ELEPHANT** (象): IS THE STRONGEST ANIMAL AND CAN CAPTURE ALL OTHER ANIMALS EXCEPT A MOUSE
2. **LION** (獅) AND **TIGER** (虎): CAN MAKE (AS AN ADDITION TO ITS NORMAL MOVES) JUMPS OVER A WATER. THERE IS ONE EXCEPTION - IT IS NOT POSSIBLE TO JUMP IF A MOUSE (PLAYER'S OR OPPONENT'S) IS BLOCKING THE JUMP PATH.
3. **Leopard** (豹), **Wolf** (狼), **Dog** (狗) and **Cat** (貓): They have no extra moves or abilities.
4. **Mouse** (鼠): the most interesting piece in this game.

Initial state

- Board with animals on their starting positions
- Player 1 to move
- Chosen game mode (player vs player, player vs computer and computer vs computer)
- Information about the player’s difficulty and depth level in case of them being bots

Objective state

- Information if one of the players won:

1. *def testFinalGame(self, p1: Player, p2: Player, board: Board, calledFromGameController: bool):* returns True or False in case of win of one of the players and notify who has win the game
2. *def noPossibleMoveForPlayer(self, player: Player, board: Board):* returns True in case the player has no valid moves. Unlike chess, if a player cannot make any legal moves, he has lost the game

Operators:

1. *def calculateMove(self, animal: Animal, board: Board, direction: str):* after choosing the animal to move and where to move it ('u', 'd', 'l', 'r'), this function returns the movement that the animal will perform on the chessboard.
2. *def killAnimal(self, board: Board, endingx: int, endingy: int):* if there is another animal in the landing square of an animal, this function kills the animal that is in that function.
3. *def moveAnimal(self, animal: Animal, board: Board, endingx: int, endingy: int):* Change the position of the moving animal on the board.

Operators

- *isValidStartingPoint(self, player: Player, board: Board, x: int, y: int):* Checks if the chosen starting point has an animal belonging to a player who makes move

- *isValidEndingPoint(self, animal: Animal, board: Board, endingX: int, endingY: int, startingX: int, startingY: int): Checks if the move is valid, by checking if the chosen field is next to the starting point (or across the water in case of tiger and lion), is valid for the animal which makes the move, and does not have an animal with a power stronger of the moving animal (except it stays in the trap)*

Game Implementation

The computer moves are implemented using a min-max algorithm, with alpha-beta cuts. This game is formulated like a search problem with an opponent.

ADDITIONAL PRUNING:

- *if state.currentPlayer.lastMoves.isARecentMove (action) == False and minimaxLastMoves.isARecentMove(action) == False:*

In the algorithm we have inserted a second pruning taking inspiration from some similar functions implemented in the game of chess. With this second pruning, the algorithm saves the last moves that the player who has to move has made and the moves that have been analyzed within the call of the mini-max function, in order to avoid generating branches that simply mirror variations of branches already generated.

Game Implementation

MOVE GENERATOR:

- *def listOfPossibleMoves(self, player: Player, board: Board):*

This function generates from a state of the board (arrangement of the pieces, player who has to move), all the possible movements that can be made by the given player. Used in the mini-max algorithm

EVALUATION FUNCTION:

- *def evaluationFunction (self, state: State, difficulty: int):*

Given a state of the board, this function returns, through **human** references, the current value of the player who has to make the move. Being a zero-sum game, a move with a positive value for one player becomes a move with a negative value for the opponent.

Evaluation functions

- Easy – opponents dojo is a win and each animal has it’s power
- Medium – each field on the board has its value, which is small in comparison to animals’ power
- Hard – each animal has its own value for being on a specific field on the board (E.g. the lion and tiger are stronger near water), and the value of all animals is given more strength to emphasize the importance of capturing an opponent's piece

Complexity of the game

The complexity in the Jungle game (just like the complexity of a normal chess game) depends on two dimensions:

- Decision complexity: The difficulty to choose the correct movement. In chess the average number of moves that a player can make are ≈35, in jungle, at most, 8*4=32.
- Space complexity: Number of possible positions in the search space. In chess we have a number of possible positions between 10^{43} and 10^{50}. In Jungle the number of possible positions is, approximately:

$$N_{moves} = D_{(51,16)} + D_{(51,15)} * D_{(12,1)} + D_{(51,14)} * D_{(12,2)} =$$

$$= \frac{51!}{(51-16)!} + \left(\frac{51!}{(51-15)!} * \left(\frac{12!}{(12-1)!}\right)\right) + \left(\frac{51!}{(51-14)!} * \left(\frac{12!}{(12-2)!}\right)\right) \simeq 10^{26}$$

Interesting Situations


1


2

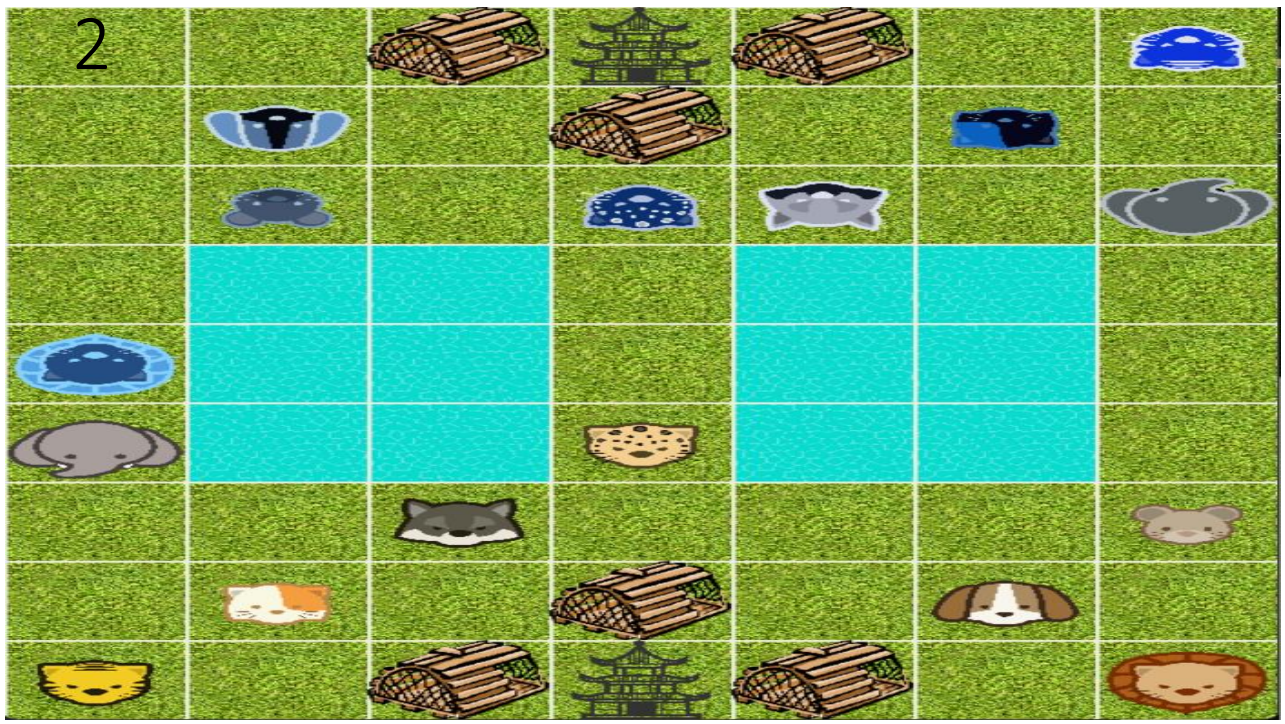

3

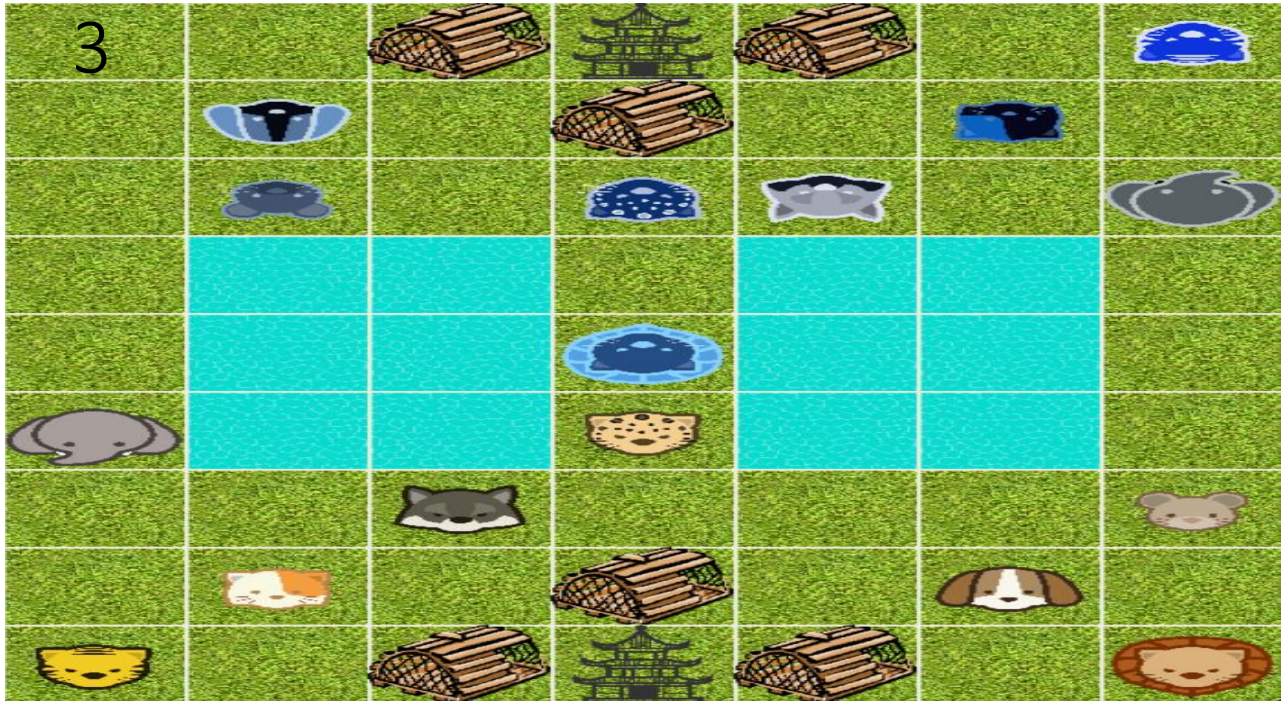

4

No danger for lion

Elephant would capture lion after its capture


1


2


3

Testing game

- *On 100 played games

Kind of match (difficulty-depth)	%win Player 1/ %win Player 2	AvgTime per round Player1/player2	Average n. of turns in the match
(1-1) vs (1-1)	44%/56%	0.03/0.04	69
(1-1) vs (3-3)	0%/100%	0,09/2,56	26
(1-3) vs (2-1)	37%/63%	2,02/0.05	128
(1-3) vs (3-1)	10%/90%	2.13/0.06	30
(2-3) vs (3-1)	41%/59%	1,78/0.07	89
(3-1) vs (3-3)	12%/88%	0,17/2,65	34
(3-2) vs. (3-2)	43%/57%	0,88/0,91	42
(3-3) vs. (3-3)	52%/48%	2,24/2,15	60

What's next?

- **Negamax**: Semplification of MinMax by using the following property: $\max(a, b) = -\min(-a, -b)$, so, instead of having the conditonnal value compute in minmax, we can use a single line that does the same in Negamax: *value := max(value, −negamax(child, depth − 1, −player))*
- **Endgame databases**: Just like in chess, we can build and save a database within the program that saves all possible moves when there are few pieces on the board, to avoid going too deep into the search tree. In this way, as soon as there are only n pieces on the board we can access the database to know all the possible best continuations to bring the game to an end.

What’s next?

Resolution of the "human factor“:

- **Genetic algorithm**: We can change the values within the value matrices of the evaluation function through a genetic algorithm, thus having the possibility of generating a series of better offspring than the starting point.
- **Monte Carlo Tree Search (MCTS)**: Instead of using an evaluation function generates a set of possible continuations (states), and, analyzing the latter and what are the final results that come from them (victory or defeat), evaluates which is the approach better to the problem by analyzing what winning solutions and losing solutions have in common. With this approach, it is possible to solve non-deterministic games too in fact

Thanks for the attention!

Bibliography:

- Artificial Intelligence, a Modern Approach – Stuart Russel
- Python implementatin of algorithms from «Artificial Intelligence, a Modern Approach» (https://github.com/aimacode/aima-python)
- Minimax and Monte Carlo Tree Search – Philipp Muens (https://philippmuens.com/minimax-and-mcts)
- Solving Jungle Checkers – Bas van Boven (https://theses.liacs.nl/pdf/2013-2014BasvanBoven.pdf)
- Discovering and Searching Loosely Coupled Subproblems in Dou Shou Qi – Joseph Burnett