Assignment 3: Adversarial Search

Ole C. Eidheim Xavier F.C. Sánchez-Díaz

3rd October 2025

Deadline: 24.10.2025, 23:59 hrs

1 Overview

In this assignment, you will implement the **Minimax** algorithm for adversarial search. We recommend following the pseudocode provided in **Figure 6.3 (p. 196)** of the textbook (AIMA 4th Ed., Global edition) Afterwards, you will extend such implementation to use **alpha-beta pruning** to speed up the search. Pseudocode for alpha-beta pruning is provided in **Figure 6.7 (p. 200)** of the textbook.

You will run **Minimax** on the halving game and the bucket game, and **Minimax** and **alpha-beta pruning** on Tic-tac-toe. Especially Tic-tac-toe will require **alpha-beta pruning** in order to avoid spending several seconds to find the first move.

You need the required functions for the different games such as `to_move(state)`, `actions(state)` and `result(state, action)`, in order to be able to run the algorithms **Minimax** and **alpha-beta pruning** on the games. These functions are provided below, and you are welcome to use and adapt this code in your submission, but it is not mandatory.

2 The halving game

The halving game is a Nim game variant where the game starts with some number N, which can for example be the starting number of rocks on the ground. Players then take turns either decrementing N or replacing it with $\frac{N}{2}$ rounded down. The player that starts its turn with $N = 0$ wins.

The source code below contains a `Game` class where the halving game is defined for use with the **Minimax** algorithm, and a `while` loop in which the game is played through self-play where both players use the **Minimax** algorithm. You must implement the **Minimax** algorithm in the `minimax_search` function. The expected output of the code is shown in the comment at the end.

`halving_game.py` Download

```python
import math

State = tuple[int, int] # Tuple of player (whose turn it is),
                        # and the number to be decreased
Action = str  # Decrement (number <- number-1) or halve (number <- number / 2)
```

```
class Game:
    def __init__(self, N: int):
        self.N = N

    def initial_state(self) -> State:
        return 0, self.N

    def to_move(self, state: State) -> int:
        player, _ = state
        return player

    def actions(self, state: State) -> list[Action]:
        return ['--', '/2']

    def result(self, state: State, action: Action) -> State:
        _, number = state
        if action == '--':
            return (self.to_move(state) + 1) % 2, number - 1
        else:
            return (self.to_move(state) + 1) % 2, number // 2  # Floored division

    def is_terminal(self, state: State) -> bool:
        _, number = state
        return number == 0

    def utility(self, state: State, player: int) -> float:
        assert self.is_terminal(state)
        return 1 if self.to_move(state) == player else -1

    def print(self, state: State):
        _, number = state
        print(f'The number is {number} and ', end='')
        if self.is_terminal(state):
            if self.utility(state, 0) > 0:
                print(f'P1 won')
            else:
                print(f'P2 won')
        else:
            print(f'it is P{self.to_move(state)+1}\'s turn')

def minimax_search(game: Game, state: State) -> Action | None:
    # YOUR CODE HERE
    assert False, "Not implemented"

game = Game(5)

state = game.initial_state()
game.print(state)
while not game.is_terminal(state):
    player = game.to_move(state)
    action = minimax_search(game, state) # The player whose turn it is
                                         # is the MAX player
    print(f'P{player+1}\'s action: {action}')
    assert action is not None
    state = game.result(state, action)
    game.print(state)

# Expected output:
# The number is 5 and it is P1's turn
# P1's action: --
# The number is 4 and it is P2's turn
# P2's action: --
# The number is 3 and it is P1's turn
# P1's action: /2
# The number is 1 and it is P2's turn
# P2's action: --
# The number is 0 and P1 won
```

Note that if you would like, you can change the `while` loop such that you play against the **Minimax** algorithm yourself.

3 The bucket game

In the bucket game, there are three buckets: $A = \{-50, 50\}, B = \{3, 1\}, C = \{-5, 15\}$. The first player chooses bucket A, B or C, and the second player then chooses one of the numbers in the chosen bucket. The goal of the first player is to maximize the selected number, while the goal of the second player is to minimize this number.

The source code for the bucket game is shown below. You can use the same **Minimax** function that you implemented for the halving game, as well as the `while` loop for self-play.

bucket_game.py Download

```python
State = tuple[int, list[str | int]]  # Tuple of player (whose turn it is),
                                     # and the buckets (as str)
                                     # or the number in a bucket
Action = str | int  # Bucket choice (as str) or choice of number


class Game:
    def initial_state(self) -> State:
        return 0, ['A', 'B', 'C']

    def to_move(self, state: State) -> int:
        player, _ = state
        return player

    def actions(self, state: State) -> list[Action]:
        _, actions = state
        return actions

    def result(self, state: State, action: Action) -> State:
        if action == 'A':
            return (self.to_move(state) + 1) % 2, [-50, 50]
        elif action == 'B':
            return (self.to_move(state) + 1) % 2, [3, 1]
        elif action == 'C':
            return (self.to_move(state) + 1) % 2, [-5, 15]
        assert type(action) is int
        return (self.to_move(state) + 1) % 2, [action]

    def is_terminal(self, state: State) -> bool:
        _, actions = state
        return len(actions) == 1

    def utility(self, state: State, player: int) -> float:
        assert self.is_terminal(state)
        _, actions = state
        assert type(actions[0]) is int
        return actions[0] if player == self.to_move(state) else -actions[0]

    def print(self, state):
        print(f'The state is {state} and ', end='')
        if self.is_terminal(state):
            print(f'P1\'s utility is {self.utility(state, 0)}')
        else:
            print(f'it is P{self.to_move(state)+1}\'s turn')
```

4 Tic-tac-toe

The source code for the Tic-tac-toe game is shown below. You can use the same **Minimax** function that you implemented for the halving game, as well as the `while` loop for self-play. You must also implement **alpha-beta pruning** and try this algorithm instead of **Minimax**.

tic_tac_toe.py Download

```python
from copy import deepcopy

State = tuple[int, list[list[int | None]]]  # Tuple of player (whose turn it is),
                                            # and board
Action = tuple[int, int]  # Where to place the player's piece

class Game:
    def initial_state(self) -> State:
        return (0, [[None, None, None], [None, None, None], [None, None, None]])

    def to_move(self, state: State) -> int:
        player_index, _ = state
        return player_index

    def actions(self, state: State) -> list[Action]:
        _, board = state
        actions = []
        for row in range(3):
            for col in range(3):
                if board[row][col] is None:
                    actions.append((row, col))
        return actions

    def result(self, state: State, action: Action) -> State:
        _, board = state
        row, col = action
        next_board = deepcopy(board)
        next_board[row][col] = self.to_move(state)
        return (self.to_move(state) + 1) % 2, next_board

    def is_winner(self, state: State, player: int) -> bool:
        _, board = state
        for row in range(3):
            if all(board[row][col] == player for col in range(3)):
                return True
        for col in range(3):
            if all(board[row][col] == player for row in range(3)):
                return True
        if all(board[i][i] == player for i in range(3)):
            return True
        return all(board[i][2 - i] == player for i in range(3))

    def is_terminal(self, state: State) -> bool:
        _, board = state
        if self.is_winner(state, (self.to_move(state) + 1) % 2):
            return True
        return all(board[row][col] is not None for row in range(3) for col in range(3))

    def utility(self, state, player):
        assert self.is_terminal(state)
        if self.is_winner(state, player):
            return 1
        if self.is_winner(state, (player + 1) % 2):
            return -1
        return 0

    def print(self, state: State):
        _, board = state
        print()
        for row in range(3):
            cells = [
                ' ' if board[row][col] is None else 'x' if board[row][col] == 0 else 'o'
                for col in range(3)
            ]
            print(f' {cells[0]} | {cells[1]} | {cells[2]}')
            if row < 2:
                print('---+---+---')
        print()
        if self.is_terminal(state):
            if self.utility(state, 0) > 0:
                print(f'P1 won')
            elif self.utility(state, 1) > 0:
                print(f'P2 won')
            else:
                print('The game is a draw')
        else:
            print(f'It is P{self.to_move(state)+1}\'s turn to move')
```

4.1 Non-mandatory

In this game:

x	o	o
x		

it is x's turn, and **Minimax** may play middle square instead of bottom left. Why? Can you change the `Game` class for Tic-tac-toe such that **Minimax** instead plays to win right away?

This part of the assignment is non-mandatory, while the rest of the assignment in this document is mandatory.

5 Deliverables

1. Source files containing the game classes, the algorithms **Minimax** and **alpha-beta pruning**, and the code for actually playing the games using the algorithms with outputs showing for example game states and actions.
2. A **single** PDF report containing:
   (a) Text outputs showing the actual playing of the different games with the **Minimax** algorithm for both players. The output should include the state of the game at every turn, the action performed and by whom, and who won the game at the end or if the game ended in a draw.
      i. If you did the non-mandatory part for Tic-tac-toe, you can include these outputs here as well

(b) The runtime of finding the first move for Tic-tac-toe with **Minimax** compared to **alpha-beta pruning**.

Recommendations

Make sure that your code is well documented. **Deliver the code and the PDF report as two separate files. If your code project consists of many files, the project should be archived in a ZIP file.**