TFRP20 Artificial Intelligence (VT25)
Assignment 1: Adversarial search (Othello)

Magnus Gäfvert
magnus.gafvert@gmail.com

2025–02–09

Abstract

This report outlines the solution to the assignment on adversarial search for the Othello game. The solution is implemented in Python and provided alongside this report.

1 Introduction

The assignment is to implement an Othello game playing agent based on (at least) the classical min-max search with α,β pruning [3], with certain program requirents to pass [2]. The provided solution does not implement the optional competition API.

2 Method

The implementation toolchain is based on Python 3.9.21 and the Anaconda distribution [1] using Microsoft Visual Studio Code with free GitHub Copilot served by the preview Anthropic Claude 3.5 Sonnet LLM and Microsoft Python plugins. This report is typeset with LaTeXusing TexLive and and the LaTeX Workshop plugin by James Yu.

The program is structured in a generic module `search_minimax` wich implements classical min-max search with termination on timer expiration as an abstract base class. The Othello game mechanics with console computer and user player support is implemented in the module `othello` which uses the `search_minimax` module. A game can be started by calling python with the `othello` module as a script from the command line.

The modules are tested with the Python `unittest` module with test suites provided in the modules `test_search_minimax` and `test_othello`, respectively.

The complete solution is provided in the files:

```
search_minimax.py
test_search_minimax.py
othello.py
test_othello.py
```

2.1 The `search_minimax` module

The `search_minimax` module includes an abstract base class `Node` to be used for adding minmax search capability to any appropriate two-player zero-sum game. The inheriting class must implement the abstract methods:

```
def is_terminal(self):
    """
    Check if the current node is a terminal state.
    Implement game-specific logic here.
    """

def evaluate(self):
    """
    Evaluate the current game state from the perspective of the max
     player.
    Must return Utility game score at terminal state.
    For all non terminal states s, return a heuristic evaluation
    Eval(s) of the board such that
    Utility(loss) <= Eval(s) <= Utility(win) for all non terminal
    states s.
    Implement game-specific evaluation function here.
    """

def get_children(self):
    """
    Generate all possible moves from current state.
    Implement game-specific move generation here.
    """
```

The `Node` class provides implementation of classical min-max search with α, β pruning and timer interruption in the `minimax` method. A search is initiated by calling the convenience method `find_best_child` with arguments to limit search depth and optionally to indicate if player is maximizing (default) or minimizing and to limit search time. The search will return a tuple of best child the best evaluation score.

```
    def minimax(self, depth, alpha, beta, maximizing_player,
    timer_start = 0, timer_limit = float('inf')):
        """
        Minimax algorithm with alpha-beta pruning and optional
    timer interruption.

        Args:
            depth: Maximum depth to search
            alpha: Alpha value for pruning
            beta: Beta value for pruning
            maximizing_player: Boolean indicating if current player
     is maximizing
            timer_start, timer_limit: search interrupted when
    current time exceeds timer_start + timer_limit

        Returns:
            best_score: The best possible score from the current
    position
        """

    def find_best_child(self, depth, maximizing_player=True,
    timer_limit = float('inf')):
        """
        Convenience method to initiate search from current game
    state using
```

```
        Minimax algorithm with alpha-beta pruning and optional
    timer interruption.

        Args:
            depth: Maximum depth to search (must be > 0)
            alpha: Alpha value for pruning
            beta: Beta value for pruning
            maximizing_player: Boolean indicating if current player
     is maximizing
            timer_limit: search interrupted when time lapsed from
    calling this method exceeds timer_limit

        Returns:
            best_child: The best possible child found from the
    current node
                None if no children exists from current game state
            best_score: The best possible score found from the
    current node
                -inf for maximizing player or +inf for minimizing
    player if no children exists from current game state
        """
```

The test suite includes a small but non-trivial example from the internet[1] which verifies that nodes are pruned correctly.

2.2 The `othello` module

The `othello` module implements the Othello game mechanics and user interaction. The module uses the `search_minimax` module to provide a computer player. The module provides a console user interface to start a game and play against the computer using algebraic notation and a unicode string representation of the board state. The module provides a convenience method `play_othello` to start a game with a computer player, and which is automatically invoked when the module is called from the command line.

The module provides the `Board` class which represent the board state in a `numpy` array (for convenience and efficiency in game logic functions vs using lists). Black and White player positions are encoded with -1 and 1, respectively, and vacant positions are 0.

2.2.1 Game logic functions

The `othello` module provides a number methods and convenience functions to support the Othello game mechanics. The methods[2] of the `Board` class and include: `valid_move`, `valid_move_dir`, `valid_moves`[3], `flip_dir`, `flip` which are explained by their respective names and docstrings. The method `make_move` updates the board state with a given move and hands the turn to the other player.

```
    def make_move(self, move, check = True):
        # updates the board with the move (if valid)
        # returns True if move is valid, False otherwise
        # move is a tuple (i,j) with i=row, j=column
        # move == None means pass
```

[1] URL in the code comments

[2] One might consider using class or static methods for some of these

[3] The `valid_moves` method randomly shuffles the list of moves to avoid predictive game play in certain situations

The `Board` class inherits from the `search_minimax.Node` class and implements the abstract methods `is_terminal`, `evaluate` and `get_children` to provide Othello game specific logic for the min-max search algorithm.

The `is_terminal` method will return `True` if the game is over, and `False` otherwise.

```
    def is_terminal(self):
        # returns True if the game is over, False otherwise
        # the game is over if at least one of the following
    conditions holds true:
        #   no player has a valid move
        #   the board is full
        #   the board has pieces of only one color
```

The `get_children` method is implemented as a generator function[4] and will identify all valid moves from the current position and then for each yield a new board state with the move applied. The method will yield a pass move (equal to current state) if no valid moves are available. The new board states are created by applying the move to a deep copy of current state[5].

```
    def get_children(self):
        # returns a list of all possible children of the current
    node
        # children are all possible moves for the current player
        # return pass move if no valid
```

The `evaluate` method is critical for the min-max search algorithm. There are many possible options to design the function, and the choice of evaluation function will have a significant impact on the performance of the computer player. In this implementation, the evaluation function is a a combination of coin parity, mobility and position static weights[6], adjusted based on the game phase (opening, mid, end). From the author's amateur intuition the proposed evaluation function will prioritize positions such as corners and edges in the opening phase, adding mobility in the mid game, and focus on coin parity in the end game.

```
    def evaluate(self):
        # For a terminal state, returns final score of the game
    from (from Black player perspective):
        #   (winner gets points for empty squares)
        # returns a heuristic evaluation of the board for the
    current state, as weighted sum of:
        #   coin parity: difference in number of pieces for black
    and white
        #   mobility: number of valid moves for black or white
        #   difference in position static weights, capturing e.g.
        #       value of corners
        #       value of pieces in the center
        #       value of number of pieces on the edges
        #   The heuristic evaluation is a weighted sum varying with
     the game phase (opening, mid, end)
```

For testing purposes, there is also a `find_random_move` method implemented which will randomly select a valid move from the current state.

[4] Even if the method likely is used in a `for` loop this will not have any practical improvement on memory footprint of the algorithm, since it is depth first.

[5] An alternative to deep copy would be to implement a roll-back of moves as the depth first algorithm propagates back.

[6] In this case picked from the arbitrary source [4]

3 Results and discussion

3.1 Playing a game

The `othello` module will initiate a console user interaction to start a game when called from the command line. User is prompted to select color for the human player (if any) as well as search depth and time limit for the computer player(s)[7]. The game is played by entering moves in algebraic notation (valid moves are shown). The played move is printed in the console together with the evaluation score of the current move and the score (if any) returned by the search algorithm. Game state is printed on the console after each move if there is a human player.[8]

```
> python othello.py
Do you want to play black ('b') or white ('w') or none ('n')? b
Enter search depth for computer player (default 3): 4
Enter permove time limit in seconds for computer player (default inf):
playing a game (black = user, white = computer)
  a b c d e f g h
1               1
2               2
3               3
4       O X     4
5       X O     5
6               6
7               7
8               8
  a b c d e f g h

Enter black move for turn 0 ['c4', 'e6', 'f5', 'd3']: f5
Turn 1: X plays f5 (value 2.0, score None)
  a b c d e f g h
1               1
2               2
3               3
4       O X     4
5       X X X   5
6               6
7               7
8               8
  a b c d e f g h

Turn 2: O plays f4 (value 0.0, score 2.0)
  a b c d e f g h
1               1
2               2
3               3
```

[7] In the current implementation both computer players will use the same setting, although extending to different settings is trivial

[8] The default pretty unicode characters representing black and white coins are not supported by LaTeXso 'X' and 'O' are used instead in the report.

```
4       O O O     4
5       X X X     5
6                 6
7                 7
8                 8
  a b c d e f g h

Enter black move for turn 2 ['g3', 'c3', 'f3', 'd3', 'e3']:
```

The game will end when no player has a valid move, and the winner is announced. The example below shows the final 6 moves of a computer vs computer game with depth 4.

```
Turn 55: X plays h4 (value 21, score 12)
Turn 56: O plays h5 (value 14, score 23)
Turn 57: X plays h7 (value 23, score 41)
Turn 58: O plays g8 (value 20, score 41)
Turn 59: X plays g7 (value 23, score 41)
Turn 60: O plays h8 (value 41, score 41)
  a b c d e f g h
1 X O X X X X X 1
2 X O O O O X O X 2
3 X O X X O O X X 3
4 X X O O X X O X 4
5 X O X O O X O X 5
6 X O X X O O X X 6
7 X X O X X X O X 7
8 X X X X X X O O 8
  a b c d e f g h

Black wins (black:41, white:23, empty:0)
```

3.2 Observations

The computer will beat the author already at search depth 2 or 3... The implementation is mostly tested on computer vs computer games with search depths up 5 without time limit. At search depth 6 there are moves which takes longer than the patience of the author.

Several variants of the evaluation function was tested but no systematic benchmarking has been performed.

Pass moves occur more frequently in the end game and seems less common for higher search depths. With an evaluation functions that ignores mobility the pass moves semmed more frequent.

With two computer players with search depths differing by 1, then the player with lower depth will always find the same best move as the player with higher depth in the previous move, which indicates correctness of the algorithm (with no time limit active).

With time limit it is possible to play larger depths, but only a few branches will be explored (depth first) and not necessarily good ones. With the shuffling of valid moves in the implementation the result becomes unpredictable such that

also two computer players with the same settings will play differently, but they still beat the author in all tested cases.

The introduced shuffling of the valid moves does not affect the result of the algorithm with a depth limit but may vary the performance since the pruning depends on the ordering of the children. If time limit is introduced, then the shuffling will affect the result, since the single or few explored branches will be random. There was an idea to try sorting the children of a node based on some evaluation, but it was not tested. Again, without time limit the only possible effect would be on performance if the sorting would prioritize branches such that the pruning works more.

One can see that the evaluation score from the algorithm correctly equals the utility (final score) for the N last moves with $N =$ `depth`.

3.3 Implementation efficiency

Litte effort has been spent on optimizing the implementation vs memory or compute resources. Since the algorithm is depth first the memory footprint of the recursion is small even if there is little consideration in data representation vs memory. There are likely a significant unnecessary computational overhead in the game logic functions which could be optimized by caching results etc.

4 Conclusions and future work

This is a lot of fun and a true rabbit hole with many ideas to test out, including of course experimenting with different types of evaluation functions. It would be interesting to apply optimization methods (Nelder-Mead or similar) on selected parameters in the evaluation function (weights etc) and run computer-vs-computer battles to find optimal sets. Another interesting extension would be the addition of Monte-Carlo search and explore mixed strategies. Some refactoring and cleanup of the code would be beneficial, e.g. to better support different player settings, and the implementation of the competition API would be a nice addition.

References

[1] *Anaconda Software Distribution*. Version Ver. 2.6.4. 2024. URL: `https://docs.anaconda.com/`.

[2] *Assignment 1: Adversarial search*. URL: `https://canvas.education.lu.se/courses/33984/assignments/225768`.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. 4 global. Pearson, 2021.

[4] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. *An Analysis of Heuristics in Othello*. URL: `https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/miniproject1_vaishu_muthu/Paper/Final_Paper.pdf`.