CSE 537 Assignment 2 Report: Game Search

Remy Oukaour
SBU ID: 107122849
remy.oukaour@gmail.com

Jian Yang
SBU ID: 110168771
jian.yang.1@stonybrook.edu

Thursday, October 8, 2015

1 Introduction

This report describes our submission for assignment 2 in the CSE 537 course on artificial intelligence. Assignment 2 requires us to implement an AI player for the game of Connect Four that uses minimax search with our own position evaluation function, as well as minimax with alpha-beta pruning. We are also supposed to generalize the game to Connect K, and implement an alternate "longest-streak-to-win" mode with a corresponding evaluation function. In this report, we discuss the implementation details and performance of our solutions.

2 Implementation

We were given seven Python code files implementing the assignment framework. Of these, *basicplayer.py*, *connectfour.py*, and *lab3.py* are essential to the problem; *tree_searcher.py* is only needed for testing our search algorithms; and *tests.py*, *tester.py*, and *util.py* are left over from the original MIT assignment and can be deleted.

For the sake of structured code, we moved the definitions of all player callbacks and their search functions into *basicplayer.py*, leaving the core Connect Four code in *connectfour.py*, and with only the *run_game* function and main testing code in *lab3.py*. (Originally the player callbacks were scattered across all three files, and some of them were only needed by the MIT assignment. We deleted these to leave only the functions we needed.)

2.1 Minimax algorithm

Since Connect Four is a zero-sum game, we were able to implement a variant of minimax called negamax, where taking the minimum utility is replaced by taking the maximum negative utility.

```
minimax_nodesExpanded = 0

def minimax(board, depth, increment,
    eval_fn=basic_evaluate,
    get_next_moves_fn=get_all_next_moves,
    is_terminal_fn=is_terminal):
    """
    Do a minimax search on the specified board to the specified depth.
    Return the column that the search finds to add a token to.
    """
    node = minimax_helper(board, depth, increment,
        eval_fn, get_next_moves_fn, is_terminal_fn)
    return node.column
```

```
def minimax_helper(board, depth, increment,
    eval_fn, get_next_moves_fn, is_terminal_fn):
    """
    Do a recursive minimax search on the specified board
    to the specified depth.
    Return the node with the best score and the corresponding column move.
    """
    global minimax_nodesExpanded
    if increment:
        minimax_nodesExpanded += 1
    if depth <= 0 or is_terminal_fn(board):
        return Node(eval_fn(board))
    best_node = Node(-Infinity)
    for column, new_board in get_next_moves_fn(board):
        child_node = -minimax_helper(new_board, depth - 1, increment,
            eval_fn, get_next_moves_fn, is_terminal_fn)
        if child_node > best_node:
            best_node = Node(child_node.score, column)
    return best_node
```

2.1.1 Data structures

When searching the game tree, each node needs to store its utility score and the column to move to reach that node. We created a simple *Node* class that stores both of these pieces of data.

2.2 New evaluation function

The new evaluation function is based on the concept of a "possible win chunk": a consecutive series of four cells (or k for Connect K) which all either belong to a single player or are empty. In other words, a chunk is capable of turning into a winning chain for that player. A player's position evaluation is the sum of the scores of all their possible win chunks, minus the sum of the scores of all their opponent's chunks.

Each possible win chunk is scored based on the number of the player's pieces in that chunk, since intuitively having more pieces already in play makes it more likely for the player to complete the chunk and win the game. We use an exponential function to weight the score: a chunk with n pieces has the score 2^n.

2.3 Alpha-beta pruning

Adding alpha-beta pruning to minimax involves changing only a few lines of code. Note that since we are using negamax, the algorithm is simplified compared to general minimax: we always check *alpha*, and alternation between *alpha* and *beta* is accomplished by recursing with (-*beta*, -*alpha*) as (*alpha*, *beta*).

```
alpha_beta_nodesExpanded = 0

def alpha_beta_search(board, depth, increment,
    eval_fn=new_evaluate,
    get_next_moves_fn=get_all_next_moves,
    is_terminal_fn=is_terminal):
    """
    Do a minimax search with alpha-beta pruning on the specified board
    to the specified depth.
    Return the column that the search finds to add a token to.
    """
```

```
    node = alpha_beta_helper(board, depth, increment, -Infinity, Infinity,
        eval_fn, get_next_moves_fn, is_terminal_fn)
    return node.column

def alpha_beta_helper(board, depth, increment, alpha, beta,
    eval_fn, get_next_moves_fn, is_terminal_fn):
    """
    Do a recursive minimax search with alpha-beta pruning
    on the specified board to the specified depth.
    Return the column that the search finds to add a token to.
    """
    global alpha_beta_nodesExpanded
    if increment:
        alpha_beta_nodesExpanded += 1
    if depth <= 0 or is_terminal_fn(board):
        return Node(eval_fn(board))
    best_node = Node(-Infinity)
    for column, new_board in get_next_moves_fn(board):
        child_node = -alpha_beta_helper(new_board, depth - 1, increment,
            -beta, -alpha, eval_fn, get_next_moves_fn, is_terminal_fn)
        if child_node > best_node:
            best_node = Node(child_node.score, column)
            alpha = max(alpha, best_node.score)
            if alpha >= beta:
                break
    return best_node
```

2.4 Generalizing the game

2.4.1 Connect K

We added a keyword argument *chain_length_goal* to to the constructor of *ConnectFourBoard*, with a default value of 4. We then modified the method *is_win_from_cell* to use this value instead of a hard-coded 4:

```
def _is_win_from_cell(self, row, col):
    """
    Return whether there is a winning set of four connected nodes
    containing the specified cell.
    """
    return (self._max_length_from_cell(row, col) >=
        self._chain_length_goal)
```

We also had to modify the methods *do_move* and *clone*, both of which return a new *ConnectFourBoard* derived from the old one, to copy the value of *self._chain_length_goal* to the child board.

2.4.2 Longest-streak-to-win

We added a keyword argument *longest_streak_to_win* to to the constructor of *ConnectFourBoard*, with a default value of False. We then modified the methods *is_win* and *is_tie* to use alternate definitions of winning and tying when *self._longest_streak_to_win* is True:

```
def is_win(self):
    """
    Return the ID of the player who has won this game.
    Return 0 if it has not yet been won.
```

```
    """
    if self._longest_streak_to_win:
        if self.num_tokens_on_board() < 20:
            return False
        current_streak = self.longest_chain(
            self.get_current_player_id())
        opponent_streak = self.longest_chain(
            self.get_opposite_player_id())
        if current_streak > opponent_streak:
            return self.get_current_player_id()
        if opponent_streak > current_streak:
            return self.get_opposite_player_id()
        return 0
    for i in xrange(self.board_height):
        for j in xrange(self.board_width):
            cell_player = self.get_cell(i, j)
            if cell_player and self._is_win_from_cell(i, j):
                return cell_player
    return 0
```

```
def is_tie(self):
    """
    Return whether the game has reached a stalemate, assuming
    that self.is_win() returns False.
    """
    if self._longest_streak_to_win:
        return self.num_tokens_on_board() == 20
    return 0 not in self._board_array[0]
```

We also had to modify the methods *do_move* and *clone*, both of which return a new *ConnectFourBoard* derived from the old one, to copy the value of *self._longest_streak_to_win* to the child board.

3 Results

3.1 Benchmarks

We tested our algorithms with three players: *basic_player*, which uses minimax with the original focused-evaluate function; *new_player*, which uses minimax with the new chunk-based function; and *alpha_beta_player*, which is like *new_player* but with alpha-beta pruning. All three players search to a depth of 4 plies.

We tested *new_player* and *alpha_beta_player* against *basic_player*. The results were:

New vs. Basic:
Execution Time: 21.734
Nodes Expanded: 19053

Alpha-Beta vs. Basic:
Execution Time: 10.766
Nodes Expanded: 4810

We observe that alpha-beta pruning expanded 25% as many nodes as unoptimized minimax search.

3.2 Connect K

Playing a game with different values of *chain_length_goal* behaves as expected. At 3, it is easier to win; at 5, it is harder. Winning at 2 is trivial, and at 6 or 7 is barely possible (with a cooperative opponent). At 1, the player to move first wins instantly, and at 8 or beyond, every game is a tie, since the board itself is only 6 cells by 7.

3.3 Longest-streak-to-win

Playing a game in longest-streak-to-win mode also behaves as expected. Each player gets to place 10 tokens, and then the player with the longest chain of tokens wins, or there is a tie if they have equally long chains.