Build an Adversarial Game Playing Agent

Project Report

Artificial Intelligence Nanodegree

In this project some experimentation is done with adversarial search techniques, to build an agent to play knights isolation. In this game two players are participating, and they can move tokens (chess' knight pieces) in L-shape.

For this project there is a (default) time limit of 150ms for each player to search for a move, in which the search is automatically cut-off after the time limit.

This is the report where the results of the implementation of the chosen experiment - developing an opening book - are presented and analyzed.

Experiment: Developing an opening book

The experiment chosen for this project was the usage of an opening book. The resulting book has a depth of 6 plies, resulting from a simulation of 1,000,000 games. Note that the version submitted to Udacity's auto-grader was a reduced one to be within the limits of the file's size (around 8MB), however the analysis in this document takes the full version (which size is around 44 MB).

Opening book generation

The process to collect statistics to build the opening book is described as follows:

- A round size is given, which is the number of games that are simulated, for this project a number of 1,000,000 was chosen.
- First a dictionary for counting purposes is created which it has a game state as a key, and the values are other dictionaries. This inner dictionaries have actions (to take in a particular state) as keys, and total rewards as values, a total reward is the number of won games minus the number of lost games from the player 1 perspective.
- A recursive function is implemented to explore down the depth plies and record the rewards for the actions on states at deeper levels. This function returns the negative of the reward, since in each call (one level deeper) the perspective is switched between player 1 and player 2.
- After the maximum deep to explore is reached, the rest of the match is simulated by performing random allowed actions for both players in each state of the game.

- The output of this process is a dictionary where the key is a state of the game, and the value is the found action with the highest reward. This is saved into the file `data.pickle`, making in this way the dictionary to be accessible by the custom player.

The following is the script in Python used to create the opening book, following the procedure previously described:

```
import random
import pickle
from collections import defaultdict, Counter
from isolation import Isolation

NUM_ROUNDS = 1000000
DEPTH = 6
DATA_FILE = 'data.pickle'

def build_table(num_rounds=NUM_ROUNDS):
    """
    Creates the table of opening moves,
    returns a dictionary with the game state as the key and
    the best found move as value.
    """

    # Creates a dictionary for counting purposes,
    # it has a game state as a key, and the values are other dictionaries.
    # This inner dictionaries have actions (taken in a particular state) as keys
    # and total reward (#won-matches - #lost-matches) as values
    book = defaultdict(Counter)

    for _ in range(num_rounds):
        state = Isolation() # Initial state (a blank board)
        build_tree(state, book)

    return {k: max(v, key=v.get) for k, v in book.items()}


def build_tree(state, book, depth=DEPTH):
    """
    Recursive function to explore the game states up to a given depth,
    and create the respective entries in the opening book.
    """
    # If the depth is 0 (i.e. the maximum depth to explore has been reached)
    # then performing a simulation for the rest of the game
    if depth <= 0 or state.terminal_test():
        return -simulate(state)

    # Taking a random action from the current state
    action = random.choice(state.actions())

    # Recursively calling this function to explore actions in deeper levels
    # reward would be 1 if the next player wins or -1 if it loses
    # (i.e. the current player wins)
    reward = build_tree(state.result(action), book, depth - 1)

    # Updating the total reward of the taken action,
    # the more wins the higher the total reward
    book[state][action] += reward
```

```
    # Returns the negative of the reward,
    # in order to adjust the result to the caller's perspective (i.e the other player's)
    return -reward


def simulate(state):
    """
    Performs a simulation of the rest of a game, by choosing random moves.
    """
    # The current player
    player_id = state.player()

    # Performing random moves till the game ends
    while not state.terminal_test():
        state = state.result(random.choice(state.actions()))

    # Returns 1 if the current player wins, or -1 if it loses
    return -1 if state.utility(player_id) < 0 else 1


# Creating the opening book (dictionary),
# and storing it into the specified pickle file
opening_book = build_table()
with open(DATA_FILE, 'wb') as f:
    pickle.dump(opening_book, f)
```

Suggested opening moves

After the generation of the opening book, the following visualization shows the recommended opening move for player 1, which was the one found as the most effective during the creation of the opening book:

```
OPENING MOVE FOR PLAYER 1

+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   | 1 |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - +
```

And the following visualization shows the most effective replay found for player 2, in response to the opening move taken by player 1:

```
BEST REPLY BY PLAYER 2

+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   | 1 |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   | 2 |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
|   |   |   |   |   |   |   |   |   |   |   |   |
+ - + - + - + - + - + - + - + - + - + - + - + - +
```

The code to display these visualizations is shown below:

```
import pickle
import queue
from isolation import Isolation, DebugState

# Loading the opening book form a pickle file
with open("data.pickle", "rb") as f:
    book = pickle.load(f)

# Creating the initial state (a blank board)
initial_statate = Isolation()

# Displaying the initial move taken by the first player
first_move_state = initial_statate.result(book[initial_statate])
print('OPENING MOVE FOR PLAYER 1')
print(DebugState().from_state(first_move_state))

# Displaying the reply taken by the second player
response_move_state = first_move_state.result(book[first_move_state])
print('BEST REPLY BY PLAYER 2')
print(DebugState().from_state(response_move_state))
```

Experimentation

The following are the variants to the search method used to measure the improvement on using an opening book. Basically they are a combination of any of these two kinds of search:

1) based on just performing random moves,
2) based on iterative deepening with Minimax and Alpha-Beta pruning.

Combined with the usage of the opening book in the first 6 levels, random opening moves or just using the opening moves given by the search algorithm.

#	Base Search	Search Technique	Purpose	Code to perform the search
1	Random choice of moves	Random opening moves	Benchmark	`self.queue.put(random.choice(state.actions()))`

2	Random choice of moves	Opening moves from book	Evaluation of Opening Book	`# Choosing a random move just in case of timeout`
				`self.queue.put(random.choice(state.actions()))`
				
				`# Choosing a move in the Opening Book,`
				`# if depth is 6 or less`
				`if state.ply_count < 6:`
				`  self.iterative_deepening(state)`

3	Iterative deepening with Minimax and Alpha-Beta pruning	Random opening moves	Benchmark	`# Choosing a random move in case of timeout`
				`self.queue.put(random.choice(state.actions()))`
				
				`# If this is not an opening move`
				`# using iterative deepening`
				`if state.ply_count >= 6:`
				`  self.iterative_deepening(state)`

4	Iterative deepening with Minimax and Alpha-Beta pruning	Opening moves given by iterative deepening with Minimax and alpha-beta pruning	Benchmark	`# Choosing a random move in case of timeout`
				`self.queue.put(random.choice(state.actions()))`
				
				`# Using iterative deepening`
				`# for searching the best move`
				`self.iterative_deepening(state)`

5	Iterative deepening with Minimax and Alpha-Beta pruning	Opening moves from book	Evaluation of Opening Book	`# Choosing a random move in case of timeout`
				`self.queue.put(random.choice(state.actions()))`
				
				`# If depth is 6 or less using an Opening Book,`
				`# otherwise using Iterative Deepening with Minimax and Alpha-Beta Pruning`
				`if state.ply_count < 6 and state in self.data:`
				`  self.queue.put(self.data[state])`
				`else:`
				`  self.iterative_deepening(state)`

The results of the experiment are reported in the following table, in which the games were executed using the following commands (note that the `fair_matches` flag is not in use):

- 10 games: `python run_match.py -r 5`
- 20 games: `python run_match.py -r 10`
- 100 games: `python run_match.py -r 50`
- 200 games: `python run_match.py -r 100`

#	Base Search	Search Technique	Percentage of Wins			
			10 Games	20 Games	100 Games	200 Games
1	Random choice of moves	Random opening moves	0.0	5.0	4.0	6.0
2	Random choice of moves	Opening moves from book	0.0	5.0	6.0	4.0
3	Iterative deepening with Minimax and Alpha-Beta pruning	Random opening moves	70.0	75.0	70.0	74.0
4	Iterative deepening with Minimax and Alpha-Beta pruning	Opening moves given by iterative deepening with Minimax and alpha-beta pruning	90.0	70.0	64.0	62.5
5	Iterative deepening with Minimax and Alpha-Beta pruning	Opening moves from book	90.0	75.0	79.0	77.5

Conclusions

In the search based on random moves it doesn't seem to have a consistent advantage of any method, and no clear tendency can be identified, it looks like the outcome is actually more dependent on the randomness of the late moves.

The cases where search algorithm is based on iterative deepening with Minimax and Alpha-Beta pruning are far more interesting. The first result to observe is that executing random opening moves is actually more efficient than using the ones given by the base algorithm, with 74% of won games given by choosing random opening moves, in comparison to 62.5% using the ones given by minimax-alpha-beta-pruning, in the run with 200 games.

The second result is that the usage of an opening book actually presents a significant improvement to the base search algorithm based on iterative deepening with Minimax and

Alpha-Beta pruning, and it is more effective that choosing random opening moves. With the usage of the created opening book (with 6 levels) a percentage of 77.5% of won games (of 200) was reached, in comparison with the base algorithm 62.5%, which represents an improvement of around 15%. The improvement against just using random opening moves (which reached 74% of won games) was of 3.5%.