Immediate Versus Delayed Rewards for the Game of Go
Reinforcement Learning

Chia-Man Hung, Dexiong Chen

Master MVA

January 23, 2017
Introduction

- Go: a complex adversarial game
- Infeasibility of the basic MCTS algorithm
- Using a heuristic function can improve performance?
Summary

1. Monte Carlo Tree Search
   - General Approach
   - UCT Algorithm

2. Immediate Reward
   - Problem Setting
   - Variants

3. Implementation
   - Code Structure
   - Optimization

4. Experiments and Results

5. Conclusion
1. Monte Carlo Tree Search
   - General Approach
   - UCT Algorithm

2. Immediate Reward
   - Problem Setting
   - Variants

3. Implementation
   - Code Structure
   - Optimization

4. Experiments and Results

5. Conclusion
General Approach

Selection \rightarrow \text{Expansion} \rightarrow \text{Simulation} \rightarrow \text{Backpropagation}

\textit{Tree Policy}

\textit{Default Policy}
Algorithm 1: General MCTS approach.

1. function MCTSSearch \((s_0)\)
2. create root node \(v_0\) with state \(s_0\)
3. for \(i = 1, ..., \text{itermax}\) do
4. \(v_l \leftarrow \text{TreePolicy}(v_0)\)
5. \(\Delta \leftarrow \text{DefaultPolicy}(s(v_l))\)
6. BackPropagate\((v_l, \Delta)\)
7. end
UCT Algorithm

Upper Confidence Bound applied for Trees (UCT)

Tree policy:

\[ v^* = \arg \max_{v_c \in \text{child}(v)} \frac{W(v_c)}{N(v_c)} + K \sqrt{\frac{\ln N(v)}{N(v_c)}} \]  (1)

where \( v_c \) is a child of \( v \), \( W \) is the wins count, \( N \) is the visits count, and \( K \) is a exploration constant to tune.

Exploration vs. Exploitation
Summary

1 Monte Carlo Tree Search
   - General Approach
   - UCT Algorithm

2 Immediate Reward
   - Problem Setting
   - Variants

3 Implementation
   - Code Structure
   - Optimization

4 Experiments and Results

5 Conclusion
Problem Setting

- **Goal:** Control a territory

- **Influence function:**
  The influence function of a white stone (respectively black) at position \( p \) over \( q \)

  \[ I^W_4(p, q) = (4 - d_4(p, q))_+ , \quad I^B_4(p, q) = -(4 - d_4(p, q))_+ , \]  

  (2)

  The total influence of the stones on position \( q \) at step \( t \)

  \[ \mathcal{I}_t(q) = \sum_{p \in W_t} I^W_4(p, q) + \sum_{p \in B_t} I^B_4(p, q) , \]  

  (3)

- **Boundary:** Empty, Adversarial
problem setting

- Reward function:

\[
\begin{align*}
    r^W_\tau(p) &= \sum_{q \in G} (I^W_2(q) - I^W_{2\tau-1}(q)) + \mathbb{1}\{I^W_{2\tau-1}(q) < 0 \leq I^W_{2\tau}(q)\} \\
    r^B_\tau(p) &= \sum_{q \in G} (-I^B_{2\tau+1}(q) + I^B_{2\tau}(q)) + \mathbb{1}\{I^B_{2\tau}(q) > 0 \geq I^B_{2\tau+1}(q)\}
\end{align*}
\]  

(4)

The final reward functions for the \(\tau^{th}\) play of player white (respectively black)

\[
\begin{align*}
    r_{W,\tau}(p) &= r^W_\tau(p) + c^W_\tau \\
    r_{B,\tau}(p) &= r^B_\tau(p) + c^B_\tau.
\end{align*}
\]  

(5)
Illustration of white’s reward function

Left: white (1, 3), (0, 5), black (0, 0). Middle: white (0, 2), (0, 6), black (1, 7). Right: white (6, 6), (1, 7), black (8, 8).
Variants

- **Pruning**: Keep promising children
- **Min-Max principle**: Take into account the opponent’s move
  \[
  a^* = \max_{a \in A(s)} \min_{b \in A(s(a))} r(a, s) - r(b, s(a)) \tag{6}
  \]
- **Back-propagated value**: Immediate reward or the official game result (1 win, 0 draw, -1 lose)
Summary

1. Monte Carlo Tree Search
   - General Approach
   - UCT Algorithm

2. Immediate Reward
   - Problem Setting
   - Variants

3. Implementation
   - Code Structure
   - Optimization

4. Experiments and Results

5. Conclusion
Code Structure

- **game_node**
  - UCT_select_child
  - add_child
  - update

- **game_state**
  - get_immediate_reward
  - get_moves
  - do_move
  - get_result

- **uct**
  - UCT

- **strategy**
  - get_state
  - next_move

- **board**
  - set_boundary
  - get_immediate_reward

- **main**
  - play_game(strategy1, strategy2)

- **UCT_strategy**
  - random_strategy
def UCT(rootstate, itermax, verbose=False):
    """ Conduct a UCT search for itermax iterations starting from rootstate.
    Return the best move from the rootstate.
    """

    rootnode = game_node.GameNode(state=rootstate)

    for i in range(itermax):
        node = rootnode
        state = rootstate.clone()

        # Select
        while node.untried_moves == [] and node.child_nodes != []: # node is fully expanded and non-terminal
            node = node.UCT_select_child()
            state.do_move(node.move)

        # Expand
        if node.untried_moves != []: # if we can expand (i.e. state/node is non-terminal)
            m = random.choice(node.untried_moves)
            state.do_move(m)
            node = node.add_child(m, state) # add child and descend tree
# Rollout
# OpenAI Go board has its maximum limit of moves as 4096
# state.get_moves() always contains -1
while not(state.py_pachi_board.is_terminal) and state.nb_moves < 4096 and len(state.get_all_moves()) > 1:
    state.do_move(random.choice(state.get_all_moves()), update=False)

# Backpropagate
while node is not None: # backpropagate from the expanded node and work back to the root node
    node.update(state.get_result(node.player_just_moved)) # state is terminal.
    node = node.parent_node

return sorted(rootnode.child_nodes, key = lambda c: c.visits)[-1].move # return the move that was most visited
Code Structure

game_node
- UCT_select_child
- add_child
- update

game_state
- get_immediate_reward
- get_moves
- do_move
- get_result

board
- set_boundary
- get_immediate_reward

uct
- UCT

strategy
- get_state
- next_move

main

play_game(strategy1, strategy2)
In case of non-captures, the influence can be updated easily. This is done in `get_immediate_reward_aux` in `board.py`. 
Summary

1. Monte Carlo Tree Search
   - General Approach
   - UCT Algorithm

2. Immediate Reward
   - Problem Setting
   - Variants

3. Implementation
   - Code Structure
   - Optimization

4. Experiments and Results

5. Conclusion
Which boundary to use? Empty or adversarial?
Compared with the official game result on 1000 games and got similar performance.
⇒ We use the empty boundary in the following.
## Experiments and Results

The default UCT strategy is better than the random strategy.

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Random strategy</th>
<th>UCT strategy: 1000 iterations, without pruning, delayed reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Player B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wins A/B/draws</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2/97/1
Experiments and Results

The delayed reward is slightly better than the immediate reward.

<table>
<thead>
<tr>
<th>Scenario 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Player A</td>
<td>UCT strategy: 10 iterations, without pruning, <strong>delayed reward</strong></td>
</tr>
<tr>
<td>Player B</td>
<td>UCT strategy: 10 iterations, without pruning, <strong>immediate reward</strong></td>
</tr>
<tr>
<td>Wins A/B/draws</td>
<td>59/40/1</td>
</tr>
</tbody>
</table>
### Experiments and Results

Choosing the optimal action is better than without pruning.

<table>
<thead>
<tr>
<th>Scenario 3</th>
<th>Player A</th>
<th>Player B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UCT strategy: 100 iterations, without pruning, delayed reward</td>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0$, delayed reward</td>
</tr>
<tr>
<td>Wins A/B/draws</td>
<td></td>
<td>0/100/0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario 4</th>
<th>Player A</th>
<th>Player B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UCT strategy: 100 iterations, without pruning, immediate reward</td>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0$, immediate reward</td>
</tr>
<tr>
<td>Wins A/B/draws</td>
<td></td>
<td>0/100/0</td>
</tr>
</tbody>
</table>
Experiments and Results

Considering the min-max principle really boosts the performance.

<table>
<thead>
<tr>
<th>Scenario 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player A</td>
</tr>
<tr>
<td>Player B</td>
</tr>
<tr>
<td>Wins A/B/draws</td>
</tr>
</tbody>
</table>
### Experiments and Results

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Player A</th>
<th>Player B</th>
<th>Wins A/B/draws</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario 6</strong></td>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0$, delayed reward</td>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0.5$, delayed reward</td>
<td>75/25/0</td>
</tr>
<tr>
<td><strong>Scenario 7</strong></td>
<td>Player A</td>
<td>Player B</td>
<td>Wins A/B/draws</td>
</tr>
<tr>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0$, delayed reward</td>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0.5$ delayed reward</td>
<td>55/45/0</td>
<td></td>
</tr>
<tr>
<td><strong>Scenario 8</strong></td>
<td>Player A</td>
<td>Player B</td>
<td>Wins A/B/draws</td>
</tr>
<tr>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0$, the delayed reward</td>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0.25$, delayed reward</td>
<td>64/36/0</td>
<td></td>
</tr>
<tr>
<td><strong>Scenario 9</strong></td>
<td>Player A</td>
<td>Player B</td>
<td>Wins A/B/draws</td>
</tr>
<tr>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0$, delayed reward</td>
<td>UCT strategy: 100 iterations, with pruning, $\epsilon=0.125$, delayed reward</td>
<td>49/51/0</td>
<td></td>
</tr>
<tr>
<td><strong>Scenario 10</strong></td>
<td>Player A</td>
<td>Player B</td>
<td>Wins A/B/draws</td>
</tr>
<tr>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0$, delayed reward</td>
<td>UCT strategy: 10 iterations, with pruning, $\epsilon=0.125$, delayed reward</td>
<td>63/37/0</td>
<td></td>
</tr>
</tbody>
</table>
Experiments and Results

A (UCT epsilon = 0) vs B (UCT epsilon = 0 ~ 1)

B's win rate

B's epsilon

10 iterations
100 iterations
Summary

1. Monte Carlo Tree Search
   • General Approach
   • UCT Algorithm

2. Immediate Reward
   • Problem Setting
   • Variants

3. Implementation
   • Code Structure
   • Optimization

4. Experiments and Results

5. Conclusion
Difficulties

- Simulate the game of Go in the OpenAI Gym.
- From understanding MCTS to actually implementing it. Data structure.
- Experiments are time-consuming (python vs. C++), especially when the min-max principle is considered. (Impossible when min-max level > 2). Ideally, we’d like to have more iterations, otherwise hard to draw conclusion.
Conclusion

- Benefits of the immediate reward: Pruning and the min-max principle boost the performance in general.
- Drawbacks of the immediate reward: The optimal action might be eliminated by pruning. Very slow. Even slower with the min-max principle.
- More iterations will be needed as $\epsilon$ grows.
- The choice of a reasonable boundary does not have much influence on the performance.
- Future work: The number of iterations fixed $\rightarrow$ Time budget fixed. (The min-max level can be studied under a fixed time budget.) Optimization with parallel computing. Try other variants combined with the immediate reward.