Immediate Versus Delayed Rewards for the Game of Go

Reinforcement Learning

Chia-Man Hung, Dexiong Chen

Master MVA

January 23, 2017

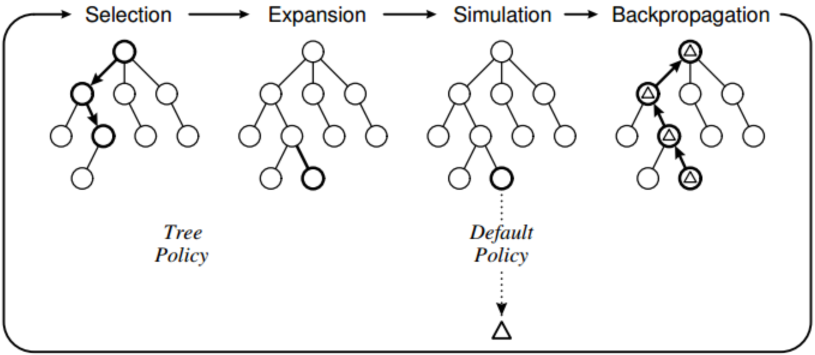
Summary

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

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

General Approach



General Approach

Algorithm 1: General MCTS approach.

<u>function</u> MCTSSearch (s_0)
create root node v_0 with state s_0
for *i* = *1*, ..., *itermax* **do**
  $v_l \leftarrow$ TreePolicy(v_0)
  $\Delta \leftarrow$ DefaultPolicy(s(v_l))
  BackPropagate(v_l, Δ)
end

UCT Algorithm

Upper Confidence Bound applied for Trees (UCT)
Tree policy:

$$v^* = \underset{v_c \in \text{child}(v)}{\arg\max} \frac{W(v_c)}{N(v_c)} + K\sqrt{\frac{\ln N(v)}{N(v_c)}} \tag{1}$$

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

$$I_4^W(p,q) = (4 - d_4(p,q))_+, I_4^B(p,q) = -(4 - d_4(p,q))_+, \quad (2)$$

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

$$\mathcal{I}_t(q) = \sum_{p \in W_t} I_4^W(p,q) + \sum_{p \in B_t} I_4^B(p,q), \quad (3)$$

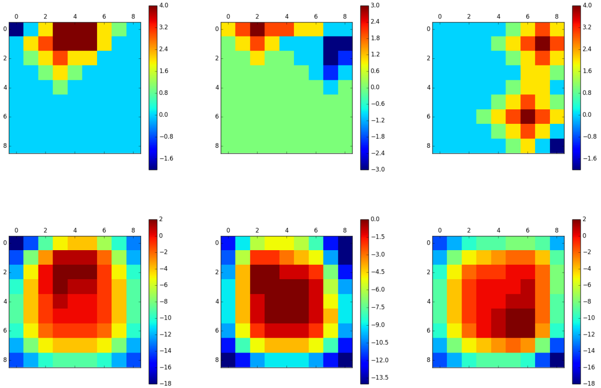
- Boundary: Empty, Adversarial

Problem Setting

- Reward function:
  The final reward functions for the τ^{th} play of player white (respectively black)

$$r_\tau^W(p) = \sum_{q \in G} (\mathcal{I}_{2\tau}^W(q) - \mathcal{I}_{2\tau-1}^W(q))_+ \mathbb{1}\{\mathcal{I}_{2\tau-1}^W(q) < 0 \leq \mathcal{I}_{2\tau}^W(q)\}$$
$$r_\tau^B(p) = \sum_{q \in G} (-\mathcal{I}_{2\tau+1}^B(q) + \mathcal{I}_{2\tau}^B(q))_+ \mathbb{1}\{\mathcal{I}_{2\tau}^B(q) > 0 \geq \mathcal{I}_{2\tau+1}^B(q)\} \tag{4}$$

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

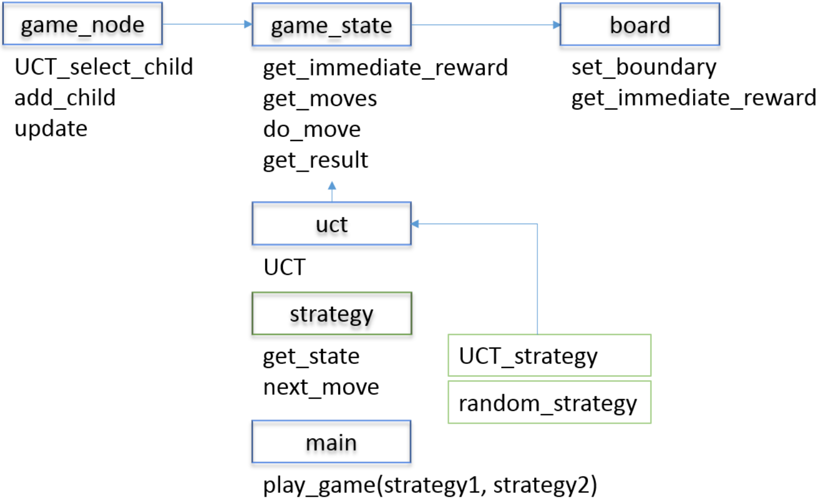
$$a^* = \max_{a \in A(s)} \min_{b \in A(s(a))} r(a, s) - r(b, s(a)) \tag{5}$$

- 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

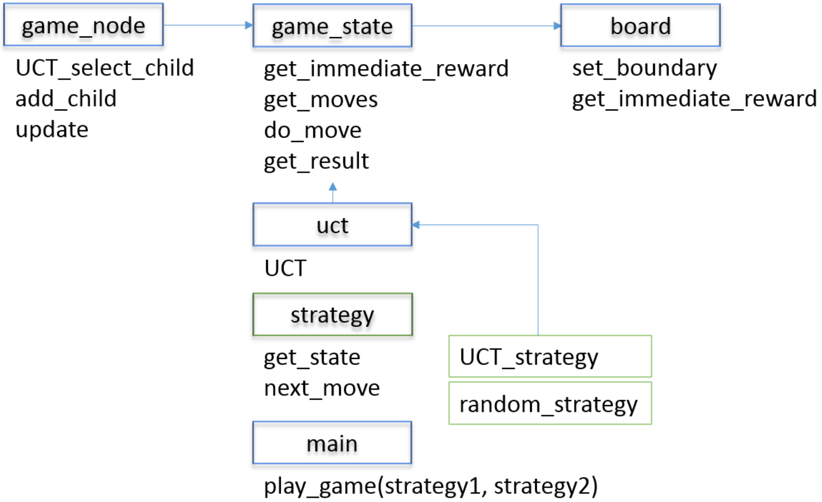
Code Structure



Implementation - game_node

```
class GameNode(object):
    """A node in the game tree. Note wins is always from the viewpoint of player_just_moved.
    """
    def __init__(self, move=None, parent=None, state=None):
        self.move = move # the move that got us to this node - "None" for the root node
        self.parent_node = parent # "None" for the root node
        self.child_nodes = []
        self.wins = 0
        self.visits = 0
        self.untried_moves = state.get_moves() # future child nodes
        self.player_just_moved = state.player_just_moved
```

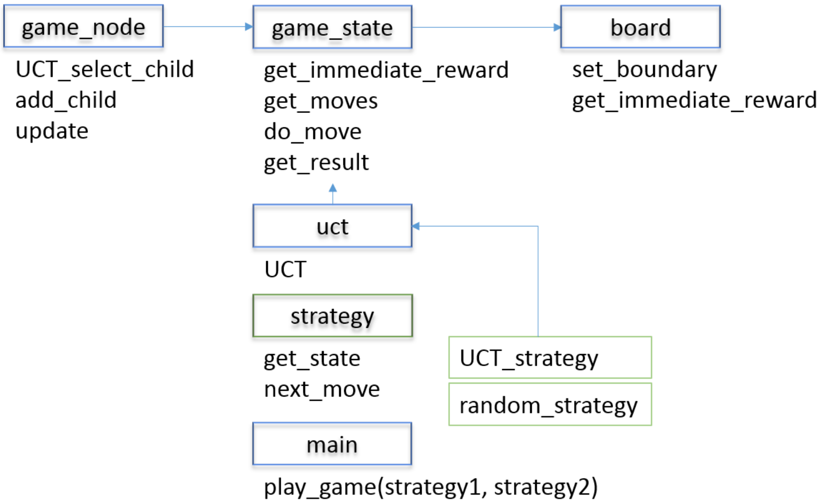
Code Structure



Implementation - game_state

```
class GameState(object):
    """A state of the game board, needed in Monte Carlo Tree Search.
       By convention, the players are numbered 1 (Black, X) and 2 (White, O).
    """
    def __init__(self, prune=False, zero_sum=False, epsilon=0., minmax=False, minmax_p=2, immediate=False):
        self.py_pachi_board = env.state.board.clone()
        self.player_just_moved = CONST.WHITE()
        self.nbmoves = 0
        self.prune = prune
        self.accumulated_reward = [0.0, 0.0]
        self.zero_sum = zero_sum
        self.epsilon = epsilon
        self.minmax = minmax
        self.minmax_p = minmax_p
        self.IW = None
        self.IB = None
        self.immediate = immediate
```

Code Structure



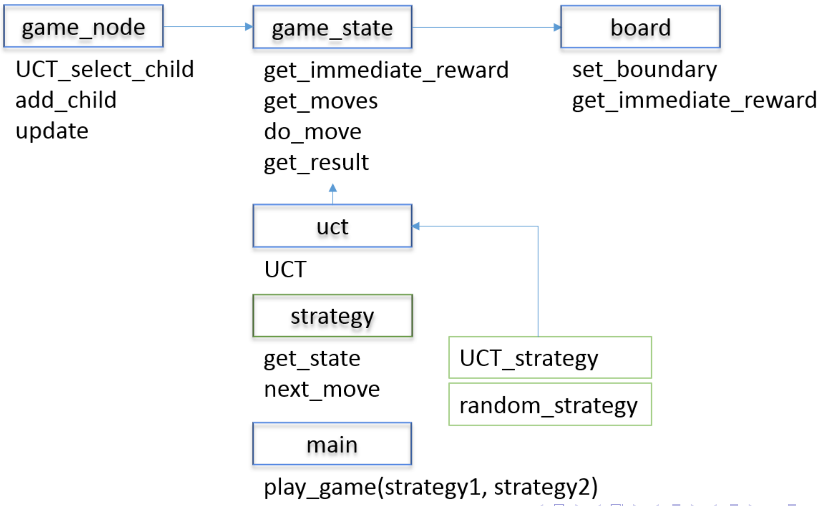
Implementation - UCT

```
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
```

Implementation - UCT

```
    # 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.nbmoves < 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



Optimization

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

Experiments and Results

Which boundary to use? Empty or adversarial?
Compared with the official game result on 1000 games and got similar performance.
$\Rightarrow$ We use the empty boundary in the following.

Experiments and Results

Scenario 1	
Player A	Random strategy
Player B	UCT strategy: 1000 iterations, without pruning, delayed reward
Wins A/B/draws	2/97/1

The default UCT strategy is better than the random strategy.

Experiments and Results

Scenario 2	
Player A	UCT strategy: 10 iterations, without pruning, delayed reward
Player B	UCT strategy: 10 iterations, without pruning, immediate reward
Wins A/B/draws	59/40/1

The delayed reward is slightly better than the immediate reward.

Experiments and Results

Scenario 3	
Player A	UCT strategy: 100 iterations, without pruning, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Wins A/B/draws	0/100/0
Scenario 4	
Player A	UCT strategy: 100 iterations, without pruning, immediate reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0, immediate reward
Wins A/B/draws	0/100/0

Choosing the optimal action is better than without pruning.

Experiments and Results

Scenario 5	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0 and min-max, delayed reward
Wins A/B/draws	19/80/1

Considering the min-max principle really boosts the performance.

Experiments and Results

Scenario 6	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.5, delayed reward
Wins A/B/draws	75/25/0
Scenario 7	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0.5 delayed reward
Wins A/B/draws	55/45/0
Scenario 8	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, the delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.25, delayed reward
Wins A/B/draws	64/36/0
Scenario 9	
Player A	UCT strategy: 100 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 100 iterations, with pruning, ϵ=0.125, delayed reward
Wins A/B/draws	49/51/0
Scenario 10	
Player A	UCT strategy: 10 iterations, with pruning, ϵ=0, delayed reward
Player B	UCT strategy: 10 iterations, with pruning, ϵ=0.125, delayed reward
Wins A/B/draws	63/37/0

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, 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 based on the pruning and the min-max principle.
- More iterations will be needed as ϵ grows.
- 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.