

Two Player Zero-Sum Games

Jonáš Kulhánek
`jonas.kulhanek@live.com`

MFF, Charles University

February 2020

Outline

- ▶ minimax strategy
- ▶ alpha-beta
- ▶ transposition table
- ▶ principal variation search
- ▶ Monte Carlo tree search
- ▶ optimality of UCB

Minimax Strategy

- ▶ same as Nash equilibrium in zero-sum games
- ▶ the existence of optimal value is guaranteed by the minimax theorem

Theorem

For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that:

- Given player 2's strategy, the best payoff possible for player 1 is V
- Given player 1's strategy, the best payoff possible for player 2 is $-V$

Two Player Zero-Sum Games

[http://inst.eecs.berkeley.edu/~cs61b/fa14/
ta-materials/apps/ab_tree_practice/](http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/)

Minimax

```
1: function minimax(node, depth, color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return heuristicFn(node, maximizing_player)
4:   if color = 1 then                                ▷ maximizing player
5:      $v \leftarrow -\infty$ 
6:     for all child of node do
7:        $v \leftarrow \max\{v, \text{minimax}(\text{child}, \text{depth} - 1, \text{-color})\}$ 
8:     return v
9:   else                                                ▷ minimizing player
10:     $v \leftarrow \infty$ 
11:    for all child of node do
12:       $v \leftarrow \min\{v, \text{minimax}(\text{child}, \text{depth} - 1, \text{-color})\}$ 
13:    return v
```

Negamax

```
1: function negamax(node, depth, color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return color  $\times$  heuristicFn(node, maximizing_player)
4:    $v \leftarrow -\infty$ 
5:   for all child of node do
6:      $v \leftarrow \max\{v, -\text{negamax}(\text{child}, \text{depth} - 1, -\text{color})\}$ 
7:   return v
```

Alpha-Beta

```
1: function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return heuristicFn(node, maximizing_player)
4:   if color = 1 then ▷ maximizing player
5:      $v \leftarrow -\infty$ 
6:     for all child of node do
7:        $v \leftarrow \max\{v, \text{alphabeta}(\text{child}, \text{depth} - 1, \alpha, \beta, \text{-color})\}$ 
8:        $\alpha \leftarrow \max\{\alpha, v\}$ 
9:       if  $\beta \leq \alpha$  then break
10:    return v
11:  else ▷ minimizing player
12:     $v \leftarrow \infty$ 
13:    for all child of node do
14:       $v \leftarrow \min\{v, \text{alphabeta}(\text{child}, \text{depth} - 1, \alpha, \beta, \text{-color})\}$ 
15:       $\beta \leftarrow \min\{\beta, v\}$ 
16:      if  $\beta \leq \alpha$  then break
17:    return v
```

Alpha-Beta Negamax

```
1: function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return color  $\times$  heuristicFn(node, maximizing_player)
4:    $v \leftarrow -\infty$ 
5:   for all child of node do
6:      $v \leftarrow \max\{v, -\text{alphabeta}(\text{child}, \text{depth} - 1, -\beta, -\alpha, \text{-color})\}$ 
7:      $\alpha \leftarrow \max\{\alpha, v\}$ 
8:     if  $\beta \leq \alpha$  then break
9:   return v
```


Extending Alpha-Beta - Sorting Moves

```
1: function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return color  $\times$  heuristicFn(node, maximizing_player)
4:    $v \leftarrow -\infty$ 
5:   sort children by  $-\text{color} \times \text{heuristicFn}(\text{child}, \text{maximizing\_player})$ 
6:   for all child of node do
7:      $v \leftarrow \max\{v, -\text{alphabeta}(\text{child}, \text{depth} - 1, -\beta, -\alpha, -\text{color})\}$ 
8:      $\alpha \leftarrow \max\{\alpha, v\}$ 
9:     if  $\beta \leq \alpha$  then break
10:  return  $v$ 
```

Extending Alpha-Beta - Transposition Table

- ▶ we can cache the alpha and beta bounds
- ▶ what needs to be the cache key?
- ▶ when are we allowed to cache a value?
- ▶ what happen when there is a cut-off?

<http://people.csail.mit.edu/plaat/mtdf.html>

Extending Alpha-Beta - Transposition Table

Pseudocode

```
1: function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , color)
2:    $\alpha' \leftarrow \alpha$ 
3:   if (node, depth)  $\in$  cache then
4:      $\alpha_c, \beta_c \leftarrow$  cache[node, depth]
5:     if  $\beta_c \leq \alpha$  then return  $\beta_c$ 
6:     if  $\beta \leq \alpha_c$  then return  $\alpha_c$ 
7:      $\alpha \leftarrow \max\{\alpha, \alpha_c\}$ 
8:      $\beta \leftarrow \min\{\beta, \beta_c\}$ 
9:   else
10:     $\alpha_c \leftarrow -\infty, \beta_c \leftarrow \infty$ 
11:    if depth = 0  $\vee$  node.terminal() then
12:      return color  $\times$  heuristicFn(node, maximizing_player)
13:     $v \leftarrow -\infty$ 
14:    sort children by  $-\text{color} \times$  heuristicFn(child, maximizing_player)
15:    for all child of node do
16:       $v \leftarrow \max\{v, -\text{alphabeta}(\text{child}, \text{depth} - 1, -\beta, -\alpha, -\text{color})\}$ 
17:       $\alpha \leftarrow \max\{\alpha, v\}$ 
18:      if  $\beta \leq \alpha$  then break
19:    if  $v \leq \alpha'$  then cache[node, depth]  $\leftarrow (\alpha_c, v)$ 
20:    else if  $\alpha' < v < \beta$  then cache[node, depth]  $\leftarrow (v, v)$ 
21:    else cache[node, depth]  $\leftarrow (v, \beta_c)$ 
22:    return  $v$ 
```

Principal Variation Search

- ▶ we have good order heuristic function
- ▶ after evaluating the first action, the algorithm checks whether the remaining actions are worse
- ▶ the test is performed via null-window search

Principal Variation Search

Pseudocode

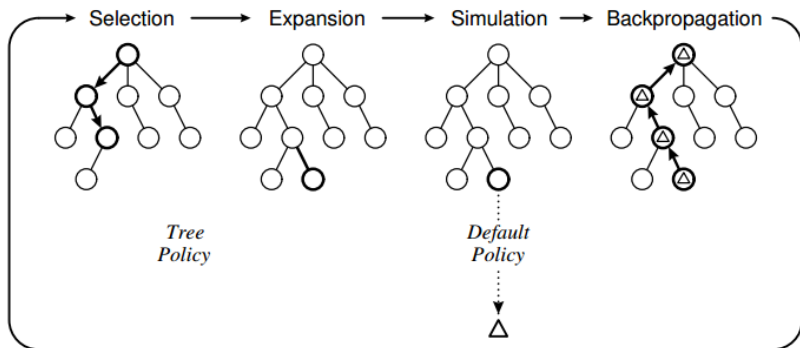
```
1: function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , color)
2:   if depth = 0  $\vee$  node.terminal() then
3:     return color  $\times$  heuristicFn(node, maximizing_player)
4:   sort children by  $-\text{color} \times \text{heuristicFn}(\text{child}, \text{maximizing\_player})$ 
5:   for all child of node do
6:     if child is first child then
7:        $v \leftarrow -\text{pvs}(\text{child}, \text{depth} - 1, -\beta, -\alpha, -\text{color})$ 
8:     else
9:        $v \leftarrow -\text{pvs}(\text{child}, \text{depth} - 1, -\alpha - 1, -\alpha, -\text{color})$ 
10:      if  $\alpha < v < \beta$  then
11:         $v \leftarrow -\text{pvs}(\text{child}, \text{depth} - 1, -\beta, -v, -\text{color})$ 
12:       $\alpha \leftarrow \max\{\alpha, v\}$ 
13:      if  $\beta \leq \alpha$  then break
14:   return  $\alpha$ 
```

Alpha-Beta Variants in Practice

- ▶ we often use PVS with good heuristic function
- ▶ transposition table is necessary in almost any game
- ▶ do we need to use the search depth in our caching key in transposition table?
- ▶ fast game rules implementation is necessary – using bit operations
- ▶ iterative deepening is used for consecutive moves
- ▶ what if we do not have any domain knowledge?

Monte Carlo Tree Search (MCTS)

- ▶ can be used when no heuristic is available
- ▶ can be used in stochastic games
- ▶ AlphaZero is based on it
- ▶ gradually increases the precision of its policy, can be used online in time-restricted domains



MCTS

```
1: function mcts(game)
2:   root  $\leftarrow$  new Node
3:   loop  $\triangleright$  until a fixed number of iterations is reached
4:     node  $\leftarrow$  root
5:     scratchGame  $\leftarrow$  clone game
6:     searchPath  $\leftarrow$  new Stack
7:     append node to searchPath
8:     while node is fully expanded do
9:       action, node  $\leftarrow$  uctSelect(node)
10:      apply action in scratchGame
11:      append node to searchPath
12:      node, searchPath  $\leftarrow$  expand(searchPath, node,
    scratchGame)
13:      value  $\leftarrow$  simulate(node, scratchGame)
14:      backpropagate(searchPath, value)
15:   return root
```


MCTS

helper functions

- ▶ **expand**(searchPath, node, game) selects randomly unexplored action and applies it (if the node is not terminal). Then, it appends the new child node to its parent and adds it to the search path.
- ▶ **simulate**(node, game) plays one game until the end, following random policy. It returns the game score relative to node.player!
- ▶ **backpropagate**(searchPath, value) updates all nodes on the search path from the leaf to the root, increasing the visit count by 1 and adding +value to nodes with the same player property as the leaf node and -1 to all nodes with different player property than the leaf node.

MCTS

simulate

```
1: function simulate(node, game)
2:   while game is not terminal do
3:     apply random action in game
4:   return terminal value of the game w.r.t. node.player
```

MCTS

expand

```
1: function expand(searchPath, node, game)
2:   if node is terminal then return node, searchPath
3:   apply random unexplored action in game
4:   child  $\leftarrow$  new Node
5:   child.player  $\leftarrow$  game.player
6:   child.terminal  $\leftarrow$  game.terminal
7:   append child to node
8:   put child to searchPath
9:   return child, searchPath
```

MCTS

backpropagate

```
1: function backpropagate(searchPath, value)
2:   player  $\leftarrow$  searchPath.top().player
3:   while searchPath is not empty do
4:     child  $\leftarrow$  pop top element from searchPath
5:     child.visitCount  $\leftarrow$  child.visitCount + 1
6:     if node.player = player then
7:       child.valueSum  $\leftarrow$  child.valueSum + value
8:     else
9:       child.valueSum  $\leftarrow$  child.valueSum - value
```

MCTS

uct select

- ▶ we select action with maximal UCT score
- ▶ the UCT score is defined as follows:

$$\text{UCT} = \text{prior} + c \sqrt{\frac{\log(N_p + 1)}{N + 1}}, \quad (1)$$

where $\text{prior} = -\frac{\text{child.valueSum}}{\text{child.visitCount}}$ if visit count is greater than 0 and 0 otherwise. N is child.visitCount and N_p is the visit count of the parent node, c is a constant.

- ▶ different formulas can be used with the same properties, in the homework assignment, you should use this one
- ▶ the motivation behind UCB follows from Chernoff-Hoeffding's inequality
- ▶ it can be shown that regret of UCB is asymptotically optimal, see Lai and Robbins (1985), Asymptotically Efficient Adaptive Allocation Rules.

UCB

- ▶ UCB explores enough to assure asymptotic optimality
- ▶ multi-armed bandit problem, $Q_t(a)$ is sample mean value in time t of action a and $q_*(a)$ is the true mean value of a
- ▶ assuming random variables X_i bounded by $[0, 1]$ and $\bar{X} = \sum_{i=1}^n X_i$, Chernoff-Hoeffding's inequality states that

$$P\{\bar{X} - \mathbb{E}[\bar{X}] \geq \delta\} \leq e^{-2n\delta^2} \quad (2)$$

- ▶ our goal is to choose δ such that for every action,

$$P\{Q_t(a) - q_*(a) \geq \delta\} \leq \left(\frac{1}{t}\right)^\alpha \quad (3)$$

- ▶ we can achieve the required inequality by setting

$$\delta \geq \sqrt{\frac{\alpha \ln t}{2N_t(a)}} \quad (4)$$

UCB

- ▶ it can be shown that regret of UCB is asymptotically optimal, see Lai and Robbins (1985), Asymptotically Efficient Adaptive Allocation Rules.
- ▶ <https://courses.cs.washington.edu/courses/cse599i/18wi/resources/lecture3/lecture3.pdf>