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:

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

ii 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/`

Minimax

```
function minimax(node, depth, maxPlayer, color)
    if depth = 0 ∨ node.terminal() then
        return heuristicFn(node, maxPlayer)
    if color = 1 then                                        ▷ maximizing player
        v ← −∞
        for all child of node do
            v ← max{v, minimax(child, depth - 1, maxPlayer, -color)}
        return v
    else                                                     ▷ minimizing player
        v ← ∞
        for all child of node do
            v ← min{v, minimax(child, depth - 1, maxPlayer, -color)}
        return v
```

Negamax

```
function negamax(node, depth, maxPlayer, color)
    if depth = 0 ∨ node.terminal() then
        return color × heuristicFn(node, maxPlayer)
    v ← −∞
    for all child of node do
        v ← max{v, −negamax(child, depth - 1, maxPlayer, -color)}
    return v
```

Alpha-Beta

function alphabeta(node, depth, α, β, maxPlayer, color)
    if depth $= 0 \vee$ *node.terminal*() **then**
        return heuristicFn(node, maxPlayer)
    if color $= 1$ **then** ▷ maximizing player
        v $\leftarrow -\infty$
        for all child of node **do**
            v $\leftarrow$ max$\{v$, alphabeta(child, depth - 1, α, β, maxPlayer, -color)$\}$
            $\alpha \leftarrow$ max$\{\alpha, v\}$
            if $\beta \leq \alpha$ **then break**
        return v
    else ▷ minimizing player
        v $\leftarrow \infty$
        for all child of node **do**
            v $\leftarrow$ min$\{v$, alphabeta(child, depth - 1, α, β, maxPlayer, -color)$\}$
            $\beta \leftarrow$ min$\{\beta, v\}$
            if $\beta \leq \alpha$ **then break**
        return v

Alpha-Beta Negamax

function alphabeta(node, depth, α, β, maxPlayer, color)
    if depth $= 0 \vee$ *node.terminal*() **then**
        return color $\times$ heuristicFn(node, maxPlayer)
    v $\leftarrow -\infty$
    for all child of node **do**
        v $\leftarrow$ max$\{v$, $-$alphabeta(child, depth - 1, $-\beta$, $-\alpha$, maxPlayer, -color)$\}$
        $\alpha \leftarrow \max\{\alpha, v\}$
        if $\beta \leq \alpha$ **then break**
    return v

Extending Alpha-Beta - Sorting Moves

function alphabeta(node, depth, α, β, maxPlayer, color)
    if depth $= 0 \vee$ *node.terminal*() **then**
        return color $\times$ heuristicFn(node, maxPlayer)
    v $\leftarrow -\infty$
    sort children by $-$color$\times$ heuristicFn(child, maxPlayer)
    for all child of node **do**
        v $\leftarrow$ max$\{v$, –alphabeta(child, depth - 1, $-\beta$, $-\alpha$, maxPlayer, -color)$\}$
        $\alpha \leftarrow$ max$\{\alpha, v\}$
        if $\beta \leq \alpha$ **then break**
    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 happend when there is a cut-off?

`http://people.csail.mit.edu/plaat/mtdf.html`

Extending Alpha-Beta - Transposition Table

Pseudocode

function alphabeta(node, depth, α, β, maxPlayer, color)
    $\alpha' \leftarrow \alpha$
    if (node, depth) $\in$ cache **then**
        $\alpha_c, \beta_c \leftarrow$ cache[node, depth]
        if $\beta_c \leq \alpha$ **then return** β_c
        if $\beta \leq \alpha_c$ **then return** α_c
        $\alpha \leftarrow \max\{\alpha, \alpha_c\}$
        $\beta \leftarrow \min\{\beta, \beta_c\}$
    else
        $\alpha_c \leftarrow -\infty, \beta_c \leftarrow \infty$
    if depth $= 0 \vee$ *node.terminal*() **then**
        return color $\times$ heuristicFn(node, maxPlayer)
    v $\leftarrow -\infty$
    sort children by $-$color$\times$ heuristicFn(child, maxPlayer)
    for all child of node **do**
        v $\leftarrow$ max$\{v$, –alphabeta(child, depth - 1, $-\beta$, $-\alpha$, maxPlayer, -color)$\}$
        $\alpha \leftarrow \max\{\alpha, v\}$
        if $\beta \leq \alpha$ **then break**
    if $v \leq \alpha'$ **then** cache[node, depth] $\leftarrow (\alpha_c, v)$
    else if $\alpha' < v < \beta$ **then** cache[node, depth] $\leftarrow (v, v)$
    else cache[node, depth] $\leftarrow (v, \beta_c)$
    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

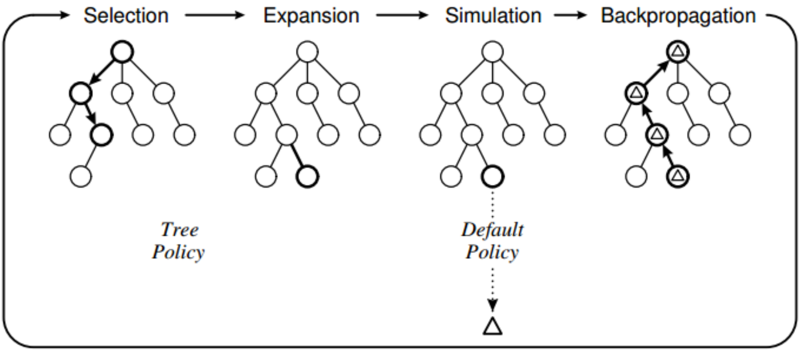
```
function alphabeta(node, depth, α, β, maxPlayer, color)
    if depth = 0 ∨ node.terminal() then
        return color × heuristicFn(node, maxPlayer)
    sort children by −color× heuristicFn(child, maxPlayer)
    for all child of node do
        if child is first child then
            v ← −pvs(child, depth - 1, −β, −α, maxPlayer, -color)
        else
            v ← −pvs(child, depth - 1, −α − 1, −α, maxPlayer, -color)
            if α < v < β then
                v ← −pvs(child, depth - 1, −β, −v, maxPlayer, -color)
        α ← max{α, v}
        if β ≤ α then break
    return α
```

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

```
function mcts(game)
    root ← new Node
    loop                ▷ until a fixed number of iterations is reached
        node ← root
        scratchGame ← clone game
        searchPath ← new Stack
        append node to searchPath
        while node is fully expanded do
            action, node ← uctSelect(node)
            apply action in scratchGame
            append node to searchPath
        node, searchPath ← expand(searchPath, node, scratchGame)
        value ← simulate(node, scratchGame)
        backpropagate(searchPath, value)
    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

function simulate(node, game)
    while game is not terminal **do**
        apply random action in game
    return terminal value of the game w.r.t. node.player

MCTS

expand

function expand(searchPath, node, game)
    if node is terminal **then return** node, searchPath
    apply random unexplored action in game
    child $\leftarrow$ **new** Node
    child.player $\leftarrow$ game.player
    child.terminal $\leftarrow$ game.terminal
    append child to node
    put child to searchPath
    return child, searchPath

MCTS

backpropagate

```
function backpropagate(searchPath, value)
    player ← searchPath.top().player
    while searchPath is not empty do
        child ← pop top element from searchPath
        child.visitCount ← child.visitCount +1
        if node.player = player then
            child.valueSum ← child.valueSum + value
        else
            child.valueSum ← child.valueSum − value
```

MCTS

uct select

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

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

  where $\mathrm{prior} = -\frac{child.valueSum}{child.visitCount}$ 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

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

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

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

- we can achieve the required inequality (with $\alpha = 2$) by setting

$$\delta \geq \sqrt{(\ln t)/N_t(a)} \tag{4}$$

- it can be shown that regret of UCB is asymptotically optimal, see Lai and Robbins (1985), Asymptotically Efficient Adaptive Allocation Rules.