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

Reading list

read this list before / during reading the rest of the slides

- http://itcs.shufe.edu.cn/download/Part%201.1%20-%20zero-sum%20games,%20proof%20of%20minimax.pdf
- http://people.csail.mit.edu/plaat/mtdf.html
- https://courses.cs.washington.edu/courses/cse599i/18wi/resources/lecture3/lecture3.pdf

Minimax Strategy

- a strategy chooses an action for each state – is a function from $\mathcal{S}$ to $\mathcal{A}(s)$.
- there exists a Nash equilibrium in zero-sum games – called minimax equilibrium
- the existence of optimal value (minimax equilibrium) is guaranteed by the minimax theorem
- finding optimal strategy can be framed as a linear program

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 $-V$

Minimax Strategy

...continued

- from the convexity of the minimax equilibrium we know, that to find the our optimal strategy it is enough to assume the other player is choosing the best action possible and vice versa
- we will calculate the optimal value of the game by recursively taking maximizing (minimizing) the returned value across all valid actions for maximizing (minimizing) player
- the recursion forms a **game tree** where each node represents a state of the game (e.g. in a chess game, it would be the positions of all pieces) and each edge is a valid action from that state (e.g. moving a pawn from a2 to a3)
- this idea is demonstrated in the minimax algorithm on the next slide

Minimax

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

Negamax

The same algorithm can be rewritten as follows:

```
1: function negamax(node, depth, color)
2:     if depth = 0 ∨ node.terminal() then
3:         return color × heuristicFn(node, maximizing_player)
4:     v ← −∞
5:     for all child of node do
6:         v ← max{v, −negamax(child, depth - 1, -color)}
7:     return v
```

Two Player Zero-Sum Games

`http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/`

- each player will not even consider taking an action for which it would not get as big reward as from an already explored action
- this idea is the base of the Alpha-Beta algorithm
- please play with Alpha-Beta simulator on the provided webpage to gain a deeper understanding how the algorithm works.

Alpha-Beta

```
1:  function alphabeta(node, depth, α, β, color)
2:      if depth = 0 ∨ node.terminal() then
3:          return heuristicFn(node, maximizing_player)
4:      if color = 1 then                                  ▷ maximizing player
5:          v ← −∞
6:          for all child of node do
7:              v ← max{v, alphabeta(child, depth - 1, α, β, -color)}
8:              α ← max{α, v}
9:              if β ≤ α then break
10:         return v
11:     else                                               ▷ minimizing player
12:         v ← ∞
13:         for all child of node do
14:             v ← min{v, alphabeta(child, depth - 1, α, β, -color)}
15:             β ← min{β, v}
16:             if β ≤ α then break
17:         return v
```

Alpha-Beta Negamax

The algorithm can be rewritten as follows:

```
1: function alphabeta(node, depth, α, β, color)
2:     if depth = 0 ∨ node.terminal() then
3:         return color × heuristicFn(node, maximizing_player)
4:     v ← −∞
5:     for all child of node do
6:         v ← max{v, –alphabeta(child, depth - 1, −β, −α, -color)}
7:         α ← max{α, v}
8:         if β ≤ α then break
9:     return v
```

Extending Alpha-Beta - Sorting Moves

It is beneficial for the algorithm to explore the best (worst) possible action for maximizing (minimizing) player **first** as it will set tighter bounds on alpha and beta.

```
1: function alphabeta(node, depth, α, β, color)
2:     if depth = 0 ∨ node.terminal() then
3:         return color × heuristicFn(node, maximizing_player)
4:     v ← −∞
5:     sort children by −color× heuristicFn(child, maximizing_player)
6:     for all child of node do
7:         v ← max{v, –alphabeta(child, depth - 1, −β, −α, -color)}
8:         α ← max{α, v}
9:         if β ≤ α 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 happend 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, α, β, color)
2:     α′ ← α
3:     if (node, depth) ∈ cache then
4:         α_c, β_c ← cache[node, depth]
5:         if β_c ≤ α then return β_c
6:         if β ≤ α_c then return α_c
7:         α ← max{α, α_c}
8:         β ← min{β, β_c}
9:     else
10:        α_c ← −∞, β_c ← ∞
11:    if depth = 0 ∨ node.terminal() then
12:        return color × heuristicFn(node, maximizing_player)
13:    v ← −∞
14:    sort children by −color× heuristicFn(child, maximizing_player)
15:    for all child of node do
16:        v ← max{v, –alphabeta(child, depth - 1, −β, −α, -color)}
17:        α ← max{α, v}
18:        if β ≤ α then break
19:    if v ≤ α′ then cache[node, depth] ← (α_c, v)
20:    else if α′ < v < β then cache[node, depth] ← (v, v)
21:    else cache[node, depth] ← (v, β_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

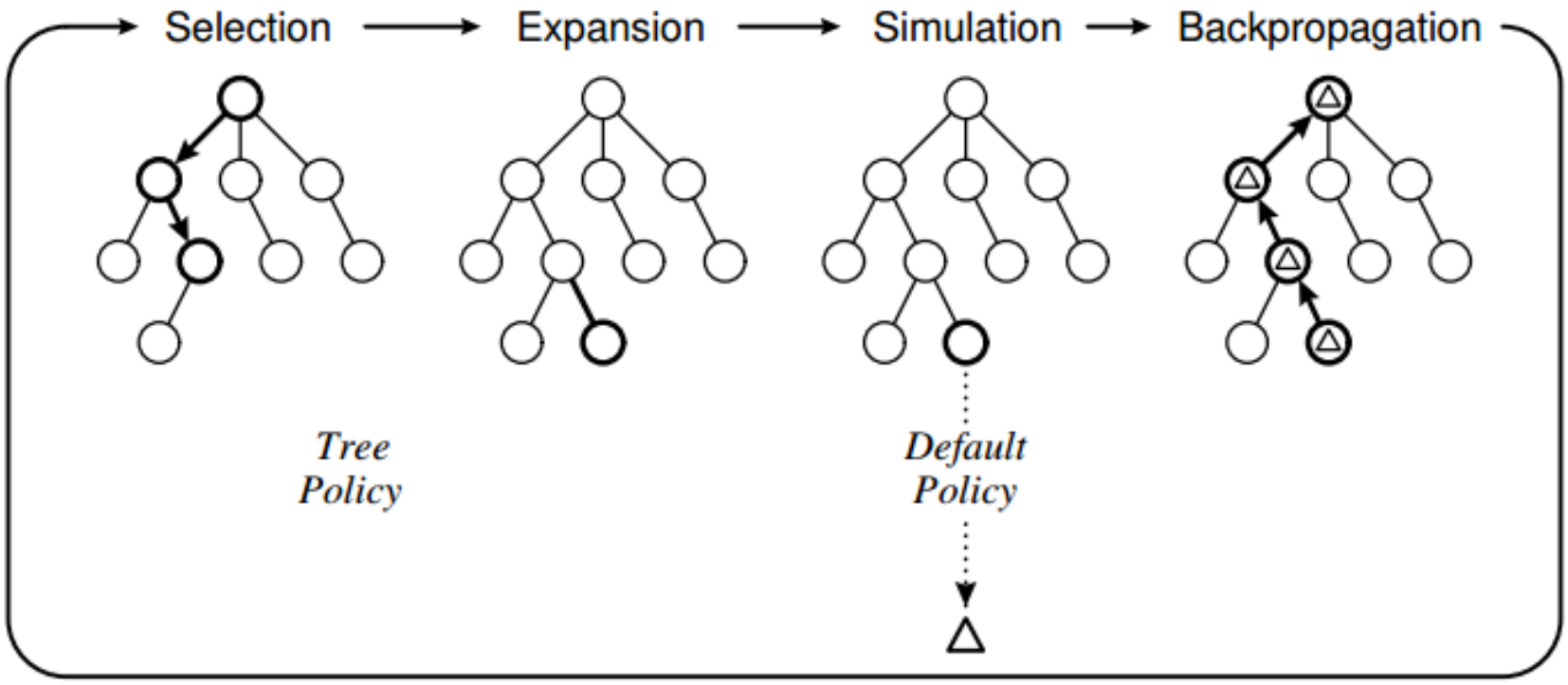
```
1:  function alphabeta(node, depth, α, β, color)
2:      if depth = 0 ∨ node.terminal() then
3:          return color × heuristicFn(node, maximizing_player)
4:      sort children by −color× heuristicFn(child, maximizing_player)
5:      for all child of node do
6:          if child is first child then
7:              v ← –pvs(child, depth - 1, −β, −α, -color)
8:          else
9:              v ← –pvs(child, depth - 1, −α − 1, −α, -color)
10:             if α < v < β then
11:                 v ← –pvs(child, depth - 1, −β, −v, -color)
12:         α ← max{α, v}
13:         if β ≤ α then break
14:     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

```
1: function mcts(game)
2:     root ← new Node
3:     loop                ▷ until a fixed number of iterations is reached
4:         node ← root
5:         scratchGame ← clone game
6:         searchPath ← new Stack
7:         append node to searchPath
8:         while node is fully expanded do
9:             action, node ← uctSelect(node)
10:            apply action in scratchGame
11:            append node to searchPath
12:        node, searchPath ← expand(searchPath, node,
       scratchGame)
13:        value ← 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 ← new Node
5:     child.player ← game.player
6:     child.terminal ← 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 ← searchPath.top().player
3:     while searchPath is not empty do
4:         child ← pop top element from searchPath
5:         child.visitCount ← child.visitCount +1
6:         if node.player = player then
7:             child.valueSum ← child.valueSum + value
8:         else
9:             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 by setting

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

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`

Evaluation

https://forms.gle/dBjAst7dqdPwAhBT8