Artificial Intelligence

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Game trees

Goals for the lecture

- Introduce model for deterministic zero-sum games with perfect information and its solutions
- Algorithms for solving game trees

Two-player zero-sum game

Two-player game with deterministic actions, complete information and zero-sum payoffs (one player's reward is equal to other player's cost)

Examples:

– Tic-tac-toe
– Othello
– Checkers
– Chess
– Go
– ...

Non-examples: Backgammon, Poker, ...

Grand challenges

- Achieve super-human performance in game of Chess
  Solved 1996–1997: IBM's DeepBlue vs. Gary Kasparov

- Achieve super-human performance in game of Go
  Solved 2016: DeepMind's AlphaGo vs. Lee Sedol

Model: Game tree

The game is modeled as a game tree:

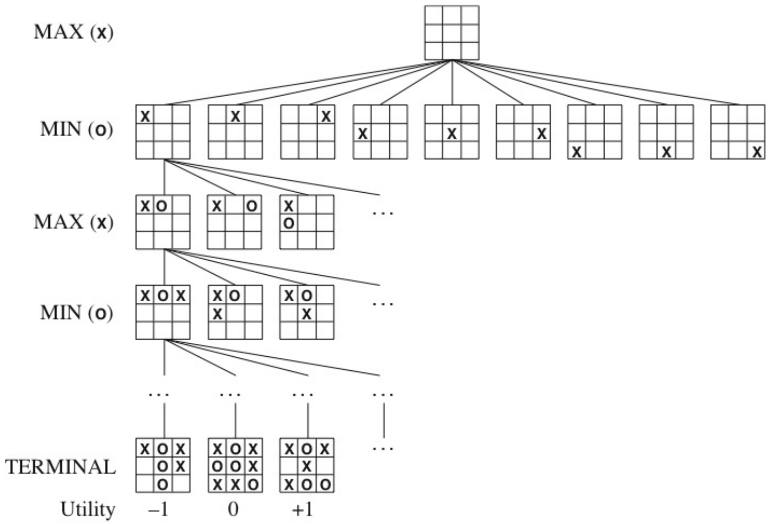
– Two types of nodes: **Max nodes** (associated to Max player) and **Min nodes** (associated to Min player)

– The tree is a **leveled tree** (also bipartite graph) rooted at Max node; the children of Max nodes are Min nodes; the children of Min nodes are Max nodes

– Each node represents a **complete configuration** of the game; the root is the initial configuration, while leaf nodes correspond to final configurations (end of the game)

– An edge between two nodes represent a **valid movement** of one player, the player incidient at the source of the edge

Example of game tree



[Image from `http://www.cs.tufts.edu/comp/131/classpages/tictactoe.jpg`]

Solutions

A solution for the initial player is a **strategy** that tells the player what to do for each possible movement of the other player

Graphically, a strategy for Max is a **subtree** T such that:

– the root belongs to T

– for each Max node n in T, T contains just one child of n

– for each Min node n in T, T contains all children of n

If Max uses the strategy described by T, as the game unfolds, one branch from root to a leaf node in T is followed

Such a branch depends on the movements of Min which are not controlled by Max

Winning strategies

A strategy T is a **winning strategy** if all leaf nodes in T correspond to configurations where Max wins

A strategy T is a **weakly winning strategy** if there is no leaf node in T that corresponds to a configuration where Max loses

Fundamental problem: determine for given game whether there is a winning or weakly winning strategy for Max

Examples:

– There is no winning strategy for Max in tic-tac-toe

– There is a weakly winning strategy for Max in tic-tac-toe

– We don't known whether there is a winning or weakly winning strategy for chess

Game value

Assign values to leaf nodes in a game tree:

– value of 1 to final configurations where Max wins

– value of 0 to final configurations where there is a tie

– value of -1 to final configurations where Max loses (i.e. Min wins)

Values are then propagated bottom-up towards the root:

– value of a **Max node** is **maximum value** of its children

– value of a **Min node** is **minimum value** of its children

Results for game trees with Max root:

- There is a winning strategy for Max iff value of root is 1
- There is a weakly winning strategy for Max iff value of root is 0

Minimax algorithm

The following **mutually recursive** DFS algorithms find the game value for a given tree (represented either implicitly or explicitly)

It assumes the game tree is finite

```
minimax(MinNode node)
    if node is terminal
        return value(node)
    score := ∞
    foreach child of node
        score := min(score, maximin(child))
    return score

maximin(MaxNode node)
    if node is terminal
        return value(node)
    score := -∞
    foreach child of node
        score := max(score, minimax(child))
    return score
```


Negamax algorithm

Observing that $\max\{a, b\} = -\min\{-a, -b\}$, Minimax can be expressed as the following algorithm known as Negamax:

```
negamax(Node node, int color)
    if node is terminal
        return color * value(node)
    alpha := -∞
    foreach child of node
        alpha := max(alpha, -negamax(child, -color))
    return alpha
```

- If called over Max node, value is `negamax(node, 1)`
- If called over Min node, value is $-$`negamax(node, -1)`

Obtaining best strategy from game tree with values

Consider a game tree where all nodes had been assigned with game values as described before

A best strategy T for Max is obtained as follows:

1. Start with a subtree T containing only the root node
2. Select leaf node n in T that is not final configuration of the game
3. If n is Max node, add best child to T (child with **max value**)
4. If n is Min node, add all its children to T
5. Repeat 2–4 until all leaves in T are final configurations of the game

Principal variation

The **principal variation** of a game is the **branch** that results when both players play in an **optimal** or **error-free** manner

There may be more than one principal variation since there may be more than one optimal play at some configuration

The node values along any principal variation are always equal to the game value (i.e. the value of the root)

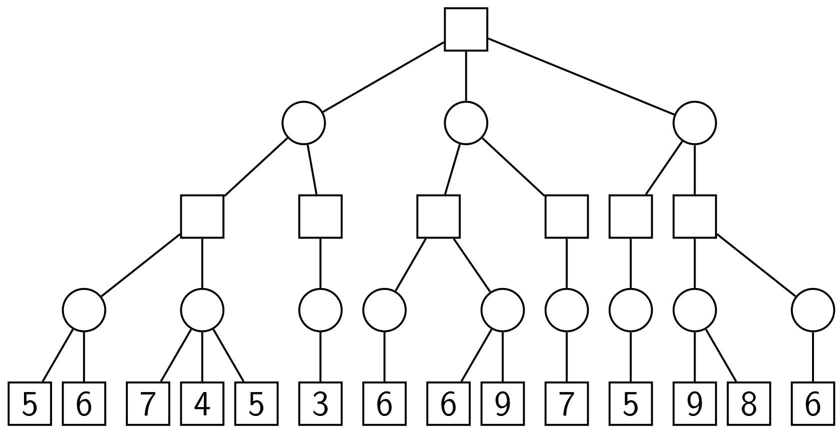
Pruned trees and heuristics

Except for trivial games, game trees are generally of exponential size (e.g. game tree for chess has about 10^{120} nodes while number of atoms in observable universe is about 10^{80})

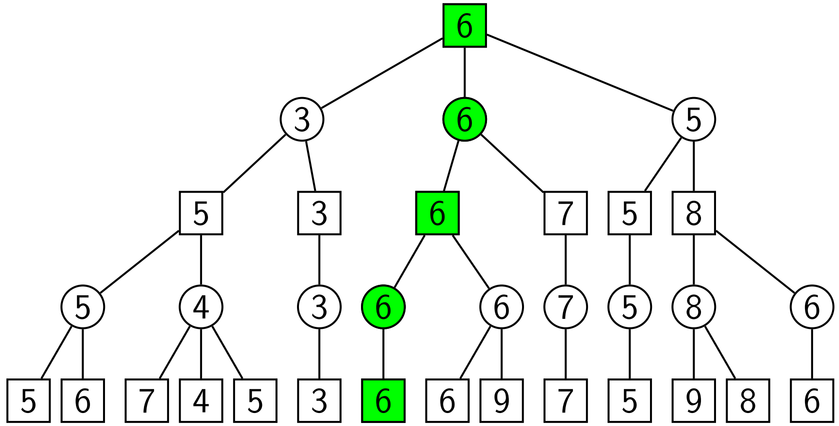
In such large games, it is impossible to compute the game value or best strategy

Game trees are typically **pruned up to some depth** and the **leaves are annotated** with (heuristic) values that weigh in the merit of the nodes with respect to the Max player

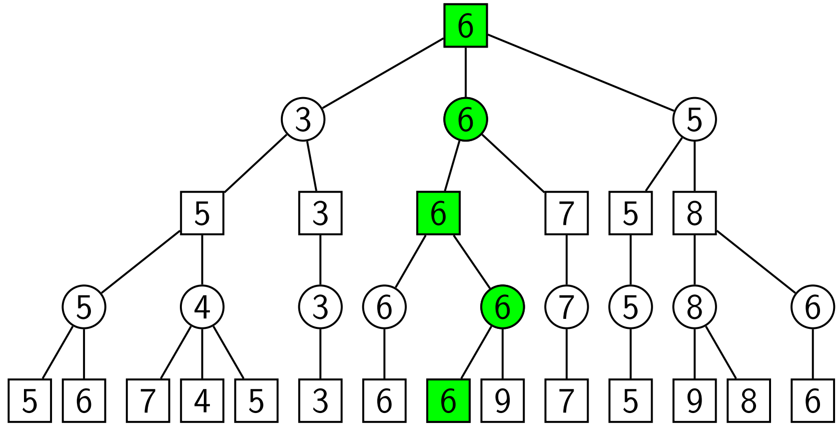
Example of pruned game tree



Example of pruned game tree



Example of pruned game tree



Material value in chess

[http://www.sumsar.net/images/posts/2015-06-10-big-data-and-chess/chess_piece_values.png]

Minimax algorithm

The following **mutually recursive** DFS algorithms find the game value for pruned tree (represented either implicitly or explicitly)

It assumes pruning is only done by depth; other criteria may be used

```
minimax(MinNode node, unsigned depth)
    if depth == 0 || node is terminal
        return h(node)
    score := ∞
    foreach child of node
        score := min(score, maximin(child, depth - 1))
    return score

maximin(MaxNode node, unsigned depth)
    if depth == 0 || node is terminal
        return h(node)
    score := -∞
    foreach child of node
        score := max(score, minimax(child, depth - 1))
    return score
```

Negamax algorithm

```
negamax(Node node, unsigned depth, int color)
    if depth == 0 || node is terminal
        return color * h(node)
    alpha := -∞
    foreach child of node
        alpha := max(alpha, -negamax(child, depth - 1, -color))
    return alpha
```

- If called over Max node, value is `negamax(node, depth, 1)`
- If called over Min node, value is $-$`negamax(node, depth, -1)`

Analysis of Minimax/Negamax

Assumptions:

– Average branching factor is b

– Search depth is d

Minimax/Negamax evaluates $O(b^d)$ nodes

Minimax with alpha-beta pruning

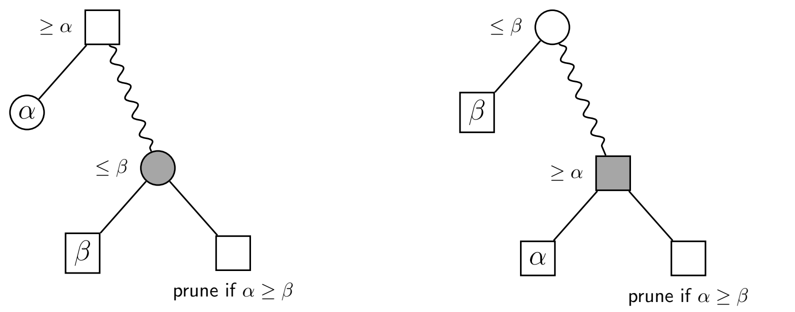
Attempt at decreasing number of nodes evaluated by Minimax

Keep two bounds, α and β, on the maximum value for Max and minimum value for Min

Use bounds to detect when there is no need to continue exploration of child nodes

Example of alpha-beta cutoffs

Stop exploring node's children when it is **proved** that their value cannot influence the value of another node up in the tree



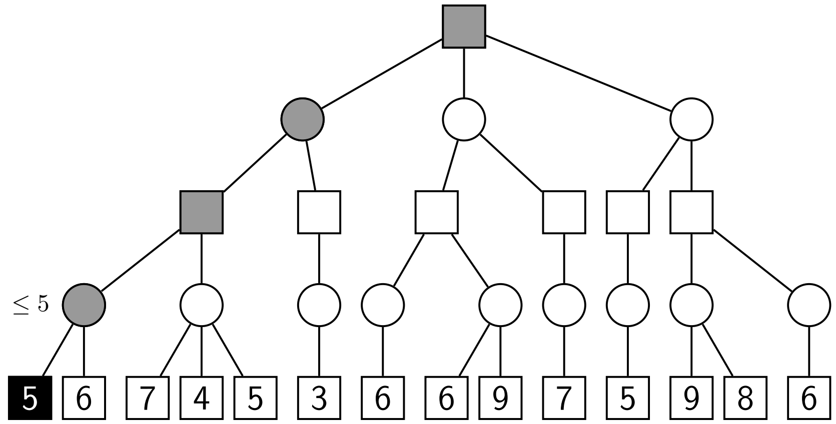
If Max/Min play optimally, game cannot reach gray nodes when $\alpha \geq \beta$

Minimax with alpha-beta pruning

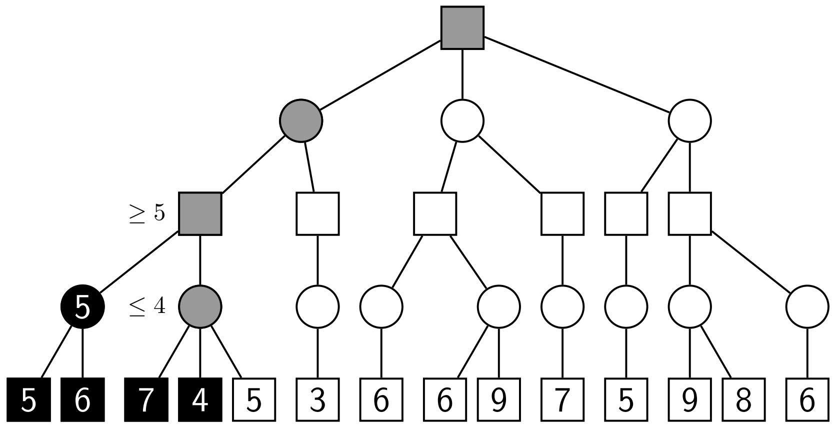
```
minimax-ab(Node node, unsigned depth, int alpha, int beta)
    if depth == 0 || node is terminal
        return h(node)

    if node is MaxNode
        foreach child of node
            value := minimax-ab(child, depth - 1, alpha, beta)
            alpha := max(alpha, value)
            if alpha >= beta then break              % beta cut-off
        return alpha

    else
        foreach child of node
            value := minimax-ab(child, depth - 1, alpha, beta)
            beta := min(beta, value)
            if alpha >= beta then break              % alpha cut-off
        return beta
```

Initial call for Max player is `minimax-ab(root, depth, -∞, +∞, 1)`

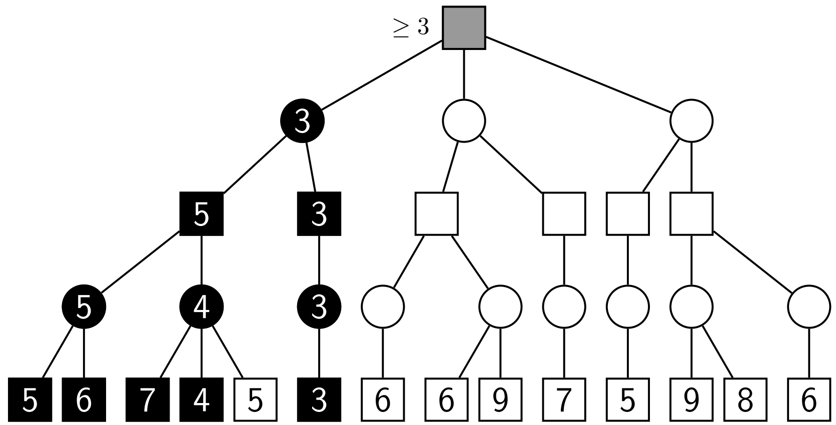
Example of minimax with alpha-beta pruning



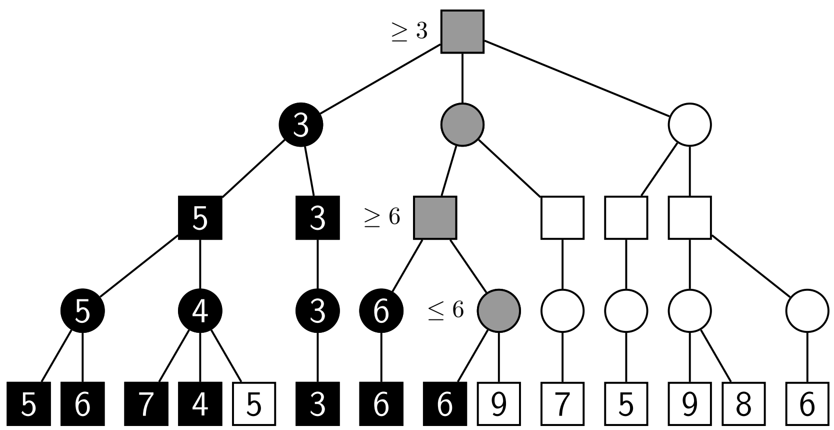
Example of minimax with alpha-beta pruning



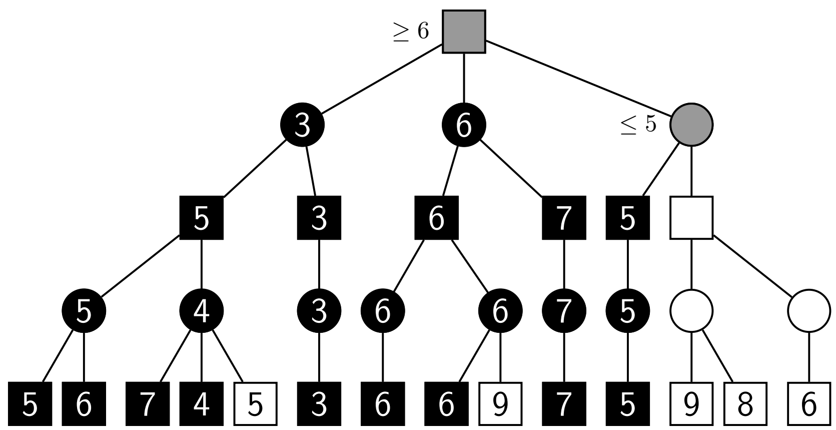
Example of minimax with alpha-beta pruning



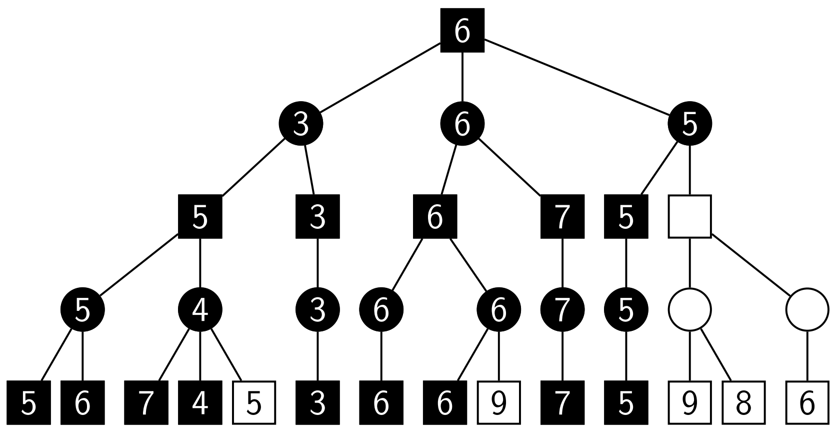
Example of minimax with alpha-beta pruning



Example of minimax with alpha-beta pruning



Example of minimax with alpha-beta pruning



Analysis of alpha-beta pruning

Assumptions:

– Average branching factor of b

– Search depth is d

Complexity depends on ordering of child nodes:

– **In worst case**, $O(b^d)$ evaluations are needed

– **In best case**, $O(b^{d/2}) = O((b^{1/2})^d)$ evaluations are needed, meaning that with the same computation power, alpha-beta pruning may go twice deeper than Minimax

– If **“random values”** at leaves, $O(b^{3d/4})$ evaluations are needed in average

Negamax with alpha-beta pruning

```
negamax-ab(Node node, unsigned depth, int alpha, int beta, int color)
    if depth == 0 || node is terminal
        h := h(node)
        return color * h

    score := -∞
    foreach child of node
        val := -negamax-ab(child, depth-1, -beta, -alpha, -color)
        score := max(score, val)
        alpha := max(alpha, val)
        if alpha >= beta then break                         % cut-off
    return score
```

Initial call for Max player is `negamax-ab(root, depth, -∞, +∞, 1)`

Motivation for scout algorithm

Can we improve on alpha-beta pruning?

Consider this situation:

– Searching branch for child n' of Max node n

– Already know that value of n is ≥ 10

If we could prove that the subtree rooted at n' cannot yield value better than 10, there is **no reason to search below** n'

Pearl's Scout algorithm

High-level idea:

- When about to search child n' of Max node n with $value(n) \geq \alpha$:

  TEST whether it is possible for n' to have value $> \alpha$. If true, search below n' to determine its value. If false, skip (prune) n'

- When about to search child n' of Min node n with $value(n) \leq \alpha$:

  TEST whether it is possible for n' to have value $< \alpha$. If true, search below n' to determine its value. If false, skip (prune) n'

Scout algorithm

```
scout(Node node, unsigned depth)
    if depth == 0 || node is terminal
        return h(node)

    score := 0
    foreach child of node
        if child is first child
            score := scout(child, depth - 1)
        else
            if node is Max && TEST(child, score, >)
                score := scout(child, depth - 1)
            if node is Min && !TEST(child, score, >=)
                score := scout(child, depth - 1)
    return score
```

Testing value of node

```
TEST(Node node, unsigned depth, int score, Condition >)
    if depth == 0 || node is terminal
        return h(node) > score ? true : false

    foreach child of node
        if node is Max && TEST(child, depth - 1, score, >)
            return true
        if node is Min && !TEST(child, depth - 1, score, >)
            return false

    return node is Max ? false : true
```

Algorithm can be adapted to change condition to $\geq$, $<$ and $\leq$

Scout algorithm: Discussion

- TEST may evaluate less nodes than alpha-beta pruning
- Scout may visit a node that is pruned by alpha-beta pruning
- For TEST to return true at subtree T, it needs to evaluate at least:
  - one child for each Max node in T
  - all children for each Min node in T

  If T has regular branching and uniform depth, the number of evaluated nodes is at least $O(b^{d/2})$
- Similar for TEST to return false at subtree T
- A node may be visited more than once: one due to TEST and another due to Scout
- Scout shows great improvement for deep games with small branching factor, but may be bad for games with large branching factor

Alpha-beta pruning with null windows

In a (fail-soft) alpha-beta search with window $[\alpha, \beta]$:

- returned value v lies in $[\alpha, \beta]$ means the value of the node is v
- **Failed high** means the search returns a value $v > \beta$ (value is $> \beta$)
- **Failed low** means the search returns a value $v < \alpha$ (value is $< \alpha$)

A **null or zero window** is a window $[\alpha, \beta]$ with $\beta = \alpha + 1$

Result of alpha-beta with a null window $[m, m+1]$ can be:

- Failed-high or $m+1$ meaning that the node value is $\geq m+1$

  Equivalent to TEST$(\text{node}, m, >)$ is true
- Failed-low or m meaning that the node value is $\leq m$

  Equivalent to TEST$(\text{node}, m, >)$ is false

Negascout = alpha-beta pruning + scout

Pruning done by alpha-beta doesn't dominate pruning done by scout, and vice versa

It makes sense to combine two types of pruning into a single algorithm

Additionally, alpha-beta with null windows can be used to implement the TEST in scout

Negamax with alpha-beta pruning combined with scout is **Negascout**

Negascout

```
negascout(Node node, unsigned depth, int alpha, int beta, int color)

  if depth == 0 || node is terminal
    h := heuristic(node)
    return color * h

  foreach child of node
    if child is first child
      score := -negascout(child, depth - 1, -beta, -alpha, -color)
    else
      score := -negascout(child, depth - 1, -alpha - 1, -alpha, -color)

      if alpha < score < beta
        score := -negascout(child, depth - 1, -beta, -score, -color)
        alpha := max(alpha, score)
        if alpha >= beta then break

    return alpha
```


Transposition tables

Game trees contain many **duplicate nodes** as different sequences of movements can lead to same game configuration

A transposition table can be used to store values for nodes in order to avoid searching below duplicate nodes

Transposition tables have **limited capacity** in order to make very efficient implementations (i.e. minimally affect node generation rate)

Good strategies to decide which nodes to store in table are needed:

– store nodes at "shallow" levels of the tree

– randomized insertion in transposition table: each time a node is encountered, throw a coin to decide whether node is stored or not. If table is full, replace node with "less use"

Other algorithms

- Conspiracy-number search (CNS) (McAllester 1985, 1988)
- Proof-number search (PNS) (Allis et al. 1994)
- Monte-carlo tree search (MCTS) (Coulom 2006)
- AlphaGo (Silver et al. 2016) and AlphaZero (Silver et al. 2017)

DeepMind's AlphaZero (Silver et al. 2017)

- Starting with only rules of game, AlphaZero achieves **super-human performance in Go** after 72h of self-play training
- Starting with only rules of game, AlphaZero achieves **super-human performance in Chess** after 4h of self-play training
- AlphaZero has two main components:
  – Single **deep neural network** to evaluate game positions that is trained from games of self play using **reinforcement learning**
  – MCTS algorithm that uses the neural network as evaluator function and that it is used for self play
  – Two components are clearly separated but operate tightly coupled during training

Summary

- Model for deterministic zero-sum games with perfect information
- Solutions, best strategies, game value and principal variation
- Necessity to prune game tree and node evaluation functions
- Algorithms that compute game value: minimax, minimax with alpha-beta pruning, scout, and negascout
- **Very important:** deciding search depth of each branch and good evaluation functions
- Recent algorithms have achieved super-human performance in games that were considered grand challenges in AI