Artificial Intelligence

Adversarial Search

Christopher Simpkins

Games

Competitive environments, in which two or more agents have conflicting goals, give rise to **adversarial search** problems.

Three approaches to dealing with multiagent environments:

1. With a large number of agents, consider aggregates of agents as an **economy**, e.g., supply-demand relationships, without predicting actions of any individual agents.
2. Consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic. But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't, we miss the idea that our adver- saries are actively trying to defeat us, whereas the rain supposedly has no such intention.
3. Explicitly model the adversarial agents with the techniques of adversarial game-tree search.

This lesson deals with the third approach.

From Game Theory to Adversarial AI

In game theory, games like Chess and Go are deterministic, two-player, turn-taking, perfect information, zero-sum games.

Game Theory	AI
Perfect information	Fully observable
Player	Agent
Move	Action
Position	State
Payoff/Score	Utility/Reward/Value

In AI we commonly study two-player zero sum games, in which the "score" for one agent is the negative of the other agent's, i.e., they add to zero.

Note: in Chess tournaments draws award each player scores of $\frac{1}{2}$, but in an AI chess playing algorithms you can use +1, -1, and 0. Alternatively, you can model chess as a "constant sum" game with scores of 1, 0, or $\frac{1}{2}$.

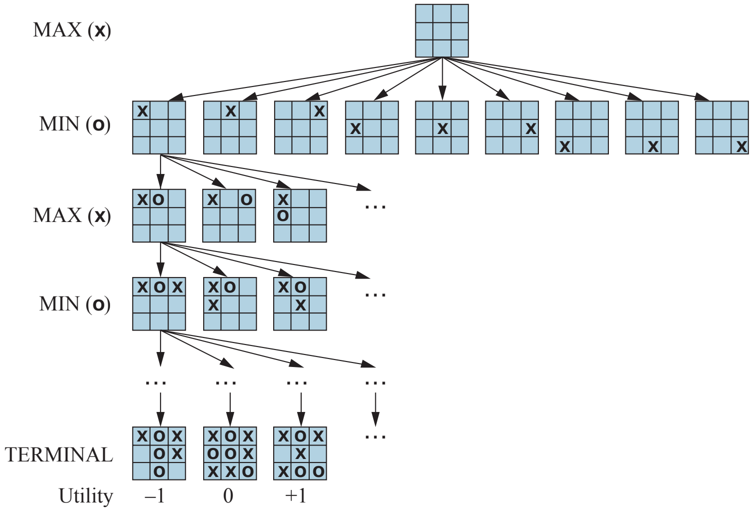
Two-Player Zero-Sum Games

Two players: MAX and MIN. MAX moves first.

A game can be formally defined with the following elements:

- S_0: The initial state, which specifies how the game is set up at the start.
- $ToMove(s)$: The player whose turn it is to move in state s, MAX or MIN.
- $Actions(s)$: The set of legal moves in state s.
- $Result(s, a)$: The transition model, which defines the state resulting from taking action a in state s.
- $IsTerminal(s)$: A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- $Utility(s, p)$: A utility function (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or $1/2$. Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

Tic-Tac-Toe Game Tree



Tic-tac-toe is relatively small – fewer than $9! = 362,880$ terminal nodes with only 5,478 distinct states.

Optimal Decisions in Games

- MAX searches for a sequence of actions leading to a win.
- MIN searches for a sequence of actions leading to a loss for MAX.
- So MAX's stratregy is a contingfency plan dependent on MIN's responses.
- We could use AND-OR search, but we'll study a more general approach: **minimax search**.

Minimax search generates a game tree with moves for both MAX and MIN

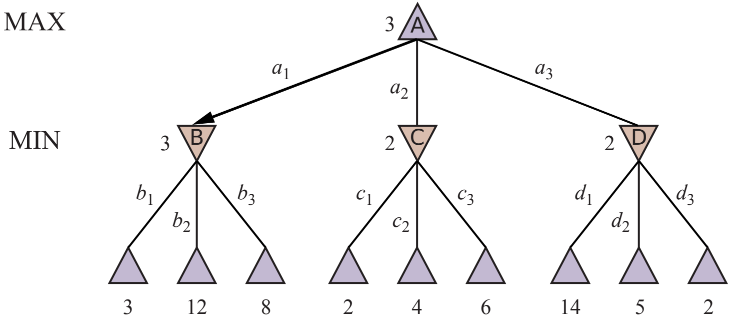
- At each MAX node, choose from actions $a \in Actions(s)$ with maximum value relative to MAX.
- At each MIN node, choose from actions $b \in Actions(s)$ with minimum value relative to MAX.
- One move by one player is called a **ply**. A ply for MAX plus the response ply for MIN constitute a game move.

Minimax Decision

- Given a game tree, the optimal strategy can be determined by working out the **minimax value** of each state in the tree, which we write as $Minimax(s)$.
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game. The minimax value of a terminal state is just its utility.

$$Minimax(s) = \begin{cases} Utility(s, MAX) & \text{if } IsTerminal(s) \\ max_{a \in Actions(s)} Minimax(Result(s,a)) & \text{if } ToMove(s) = MAX \\ min_{a \in Actions(s)} Minimax(Result(s,a)) & \text{if } ToMove(s) = MIN \end{cases}$$

Minimax Game Tree



A two-ply game tree.

- △ nodes are "MAX nodes," – MAX's turn to move.
- ▽ nodes are "MIN nodes."
- Terminal nodes show the utility values for MAX.
- Other nodes are labeled with their minimax values.
- MAX's best move at the root is a_1, because it leads to state with highest minimax value.
- MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Algorithm

- Recurse through MIN and MAX plies of the game tree to leaf nodes.
- Back up minimax values through the tree.
- Choose branch with highest minimax value.

```
**function** MINIMAX-SEARCH($game$, $state$) **returns** $an\ action$
   player $\leftarrow$ $game$.TO-MOVE($state$)
   $value$, $move$ $\leftarrow$ MAX-VALUE($game$, $state$)
   **return** $move$

**function** MAX-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
   **if** $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
   $v$, $move$ $\leftarrow -\infty$
   **for each** $a$ **in** $game$.ACTIONS($state$) **do**
      $v2$, $a2$ $\leftarrow$ MIN-VALUE($game$, $game$.RESULT($state$, $a$))
      **if** $v2 > v$ **then**
         $v$, $move$ $\leftarrow$ $v2$, $a$
   **return** $v$, $move$

**function** MIN-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
   **if** $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
   $v$, $move$ $\leftarrow +\infty$
   **for each** $a$ **in** $game$.ACTIONS($state$) **do**
      $v2$, $a2$ $\leftarrow$ MAX-VALUE($game$, $game$.RESULT($state$, $a$))
      **if** $v2 < v$ **then**
         $v$, $move$ $\leftarrow$ $v2$, $a$
   **return** $v$, $move$
```

KENNESAW STATE UNIVERSITY

Analysis of Minimax

- Time complexity: $O(b^m)$
- Space complexity:
  - $O(bm)$ if generates all actions at once
  - $O(m)$ if generates actions one at a time

Chess has an average branching factor of 35 and an average game has a depth of 80.

- $O(b^m) = 35^{80} \approx 10^{123}$ states

Clearly, minimax won't work for Chess. We'll make two modifications to minimax that makes games like Chess tractable.

Alpha-Beta Pruning

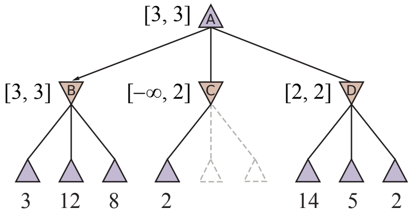
Number of game states is exponential in depth of tree, so we have to **prune** branches of the tree to make searching it tractable.

- $\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - Think: $\alpha =$ "at least."
- $\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.
  - Think: $\beta =$ "at most."

Alpha-beta pruning cuts off the exploration of nodes if those nodes would not change move selection.

Alpha-Beta Pruning in Action

Intervals show range of possible values at each node, $[\alpha, \beta]$.

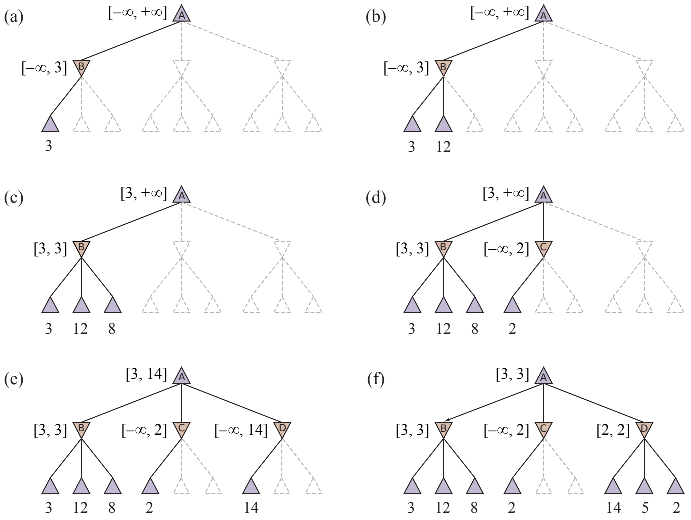


$$\begin{aligned} Minimax(root) &= max(min(3,12,8), min(2,x,y), min(14,5,2)) \\ &= max(3, min(2,x,y), 2) \\ &= max(3, z, 2) \text{ where } z = min(2,x,y) \leq 2 \\ &= 3. \end{aligned}$$

In other words, the value of the root and hence the minimax decision are independent of the values of the leaves x and y, and therefore they can be pruned.

Alpha-Beta Calculation Stages

- (a) 1st child of B has value 3. So B, a MIN node, has value ≤ 3.
- (b) 2nd child of B has value 12; MIN would avoid, so B still ≤ 3.
- (c) 3rd child of B has value 8; that's all B's children, so value of B exactly 3. Now infer value of root ≥ 3 – MAX picks max child value.
- (d) 1st child of C has value 2. So C, a MIN node, value ≤ 2. But B is worth 3, so MAX won't choose C. No point looking at other children of C. **This is alpha–beta pruning**.
- (e) 1st child of D value 14, so D worth ≤ 14. 14 still > 3, so need to keep exploring D's children. Now also back up 14 to root, which is new "at most", or β, of root.
- (f) 2nd child of D worth 5, so need to keep exploring for a value < 3. 3rd child is worth 2, so now D is worth exactly 2. So MAX's move is to B, giving a value of 3.



Alpha-Beta Search Algorithm

- Updates the values of α and β as it goes along
- Prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α for MAX, or β for MIN.

```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
    return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← $-\infty$
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
        if v2 $>$ v then
            v, move ← v2, a
            $\alpha$ ← MAX($\alpha$, v)
        if v $\geq$ $\beta$ then return v, move
    return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← $+\infty$
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
        if v2 $<$ v then
            v, move ← v2, a
            $\beta$ ← MIN($\beta$, v)
        if v $\leq$ $\alpha$ then return v, move
    return v, move
```

Heuristic Alpha-Beta Tree Search

Alpha-beta search helps, but it is still highly dependent on move ordering. So we need more help in limiting search.

- We define a **heuristic evaluation function** that evaluates a static position.
- We replace the terminal test with a cutoff test and use the heuristic evaluation function to determine the node's value.

$$HMinimax(s,d) = \begin{cases} Eval(s, MAX) & \text{if } IsCutoff(s,d) \\ max_{a \in Actions(s)} HMinimax(Result(s,a), d+1) & \text{if } ToMove(s) = MAX \\ min_{a \in Actions(s)} HMinimax(Result(s,a), d+1) & \text{if } ToMove(s) = MIN \end{cases}$$

Static Evaluation Functions

A heuristic evaluation function $Eval(s, p)$ returns an estimate of the expected utility of state s to player p, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal.

- For terminal states, $Eval(s, p) = Utility(s, p)$.
- For nonterminal states, the evaluation must be somewhere between a loss and a win: $Utility(loss, p) \leq Eval(s, p) \leq Utility(win, p)$.

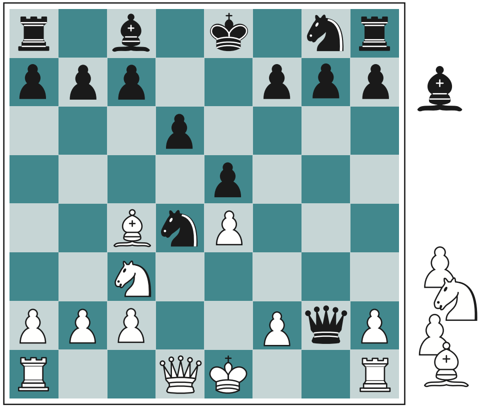
A popular form of hueristic evaluation function is a weighted linear evaluation function:

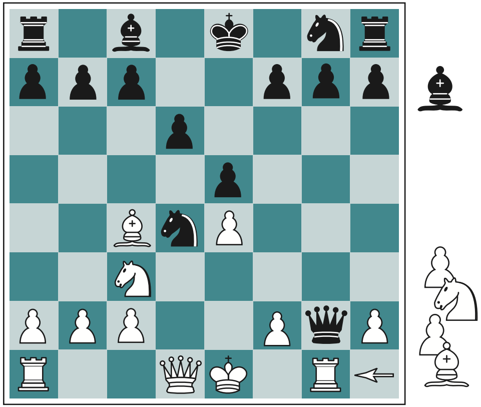
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s),$$

- Each f_i is a feature of the position such as "number of white bishops."
- Each w_i is a weight saying how important that feature is.

KENNESAW STATE UNIVERSITY

Chess Configuration Examples


(a) White to move


(b) White to move

- (a) Black has an advantage of a knight and two pawns, which should be enough to win the game.
- (b) White will capture the queen, giving it an advantage that should be strong enough to win.

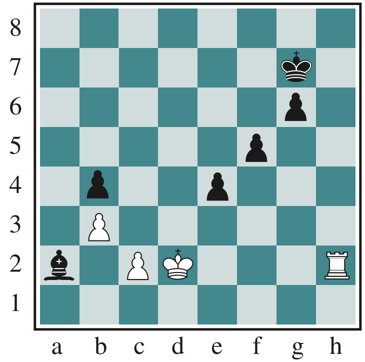
Cutting Off Search

In alpha-beta search algorithm, keep track of depth so you can cut off at a particular depth and replace lines with $IsTerminal(state)$ with

$$\textbf{if } game.IsCutoff(state, depth) \textbf{ then return } game.Eval(state, player), null$$

Best to apply cutoff quiescent positions, that is, positions that don't contain a killer move that would radically change the value of the position. Adding this to the $IsCutoff$ function is called **quiescence search**.

Horizon Effect



With Black to play, if Black only searches to a depth of 4-ply, it will see that

- moving the bishop either leads to its capture on b3, or to attack by the white rook moving to h1, but
- playing ... e3+; Kxe3, f4+; Kxf4 sacrifices two pawns but "saves" the bishop.

This is an example of the horizon effect – failing to see that these moves do not, in fact, save the bishop and ultimately result in a worse posiotion for black.

Monte Carlo Tree Search (MCTS)

Two major weaknesses of heuristic alpha-beta tree search:

- Can't handle high branching factors. Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
- Can't always define a good static evaluation function. E.g., in Go material value is not a strong indicator and most positions are in flux until the endgame.

MCTS:

Instead of searching to a given depth and applying a heurstic evaluation function to the resulting positions, we

- simulate complete games (from a given position) to terminal positions, and
- back-up the win/loss scores up the tree.

Playout policy: bias move selection towards good ones. Two options:

- Pure MCTS: do N simulations, track which move leads to highest win percentage.
- Selection policy: focus computation on important parts of game tree by balancing exploration and exploitation.

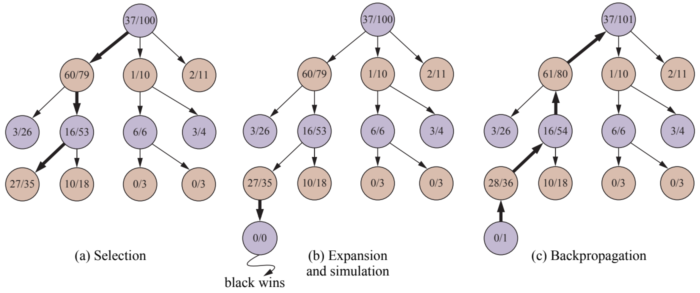
Selection-Based MCTS Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

- Selection: starting at root, choose a move leading to a successor node, and repeat that process, moving down the tree to a leaf.
- Expansion: grow the search tree by generating a new child of the selected node.
- Simulation: perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree. In the next slide, the simulation results in a win for black.
- Back-propagation: use the result of the simulation to update all search tree nodes going up to the root.

MCTS Iteration

White just played (light purple nodes) – black to play.



(a) Selection (b) Expansion and simulation (c) Backpropagation

- (a) Select moves, all the way down the tree, ending at the leaf node marked 27/35 (27 wins for black out of 35 playouts).
- (b) Expand the selected node and do a simulation (playout), which ends in a win for black.
- (c) The results of the simulation are back-propagated up the tree.
  - Since black won the playout, black nodes are incremented in both number of wins and number of playouts, so $27/35 \rightarrow 28/36$, $60/79 \rightarrow 61/80$.
  - White lost, so white nodes incremented in number of playouts, so $16/53 \rightarrow 16/54$, root $37/100 \rightarrow 37/101$.

KENNESAW STATE UNIVERSITY

Upper Confidence Bound MCTS Selection Policy

$$UCB1(n) = \overbrace{\frac{U(n)}{N(n)}}^{\text{Exploitation Term}} + C \times \overbrace{\sqrt{\frac{\log N(PARENT(n))}{N(n)}}}^{\text{Exploration Term}}$$

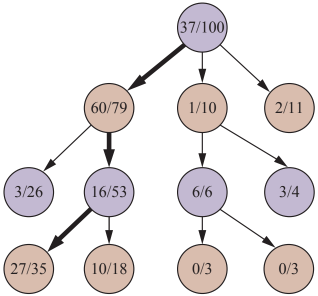
- $U(n)$: total utility of all playouts that went through node n,
- $N(n)$: number of playouts through node n, $PARENT(n)$ is parent node of n in tree,
- C: constant that balances exploitation and exploration.
  - Theoretically, $C = \sqrt{2}$ is best.
  - In practice, programmers try different values.
  - AlphaZero adds move probability term, calculated by neural network trained on past self-play.

Remember that MCTS returns the node with the highest number of playouts, not the node with the highest average utility. Why?

- A node with 65/100 wins better than one with 2/3 wins, due to lower uncertainty.
- UCB1 ensures that node with most playouts is almost always node with highest win percentage – selection process increasingly favors win percentage as number of playouts increases.

KENNESAW STATE UNIVERSITY

UCB1 In Action – Exploitation



With $C = 1.4$,

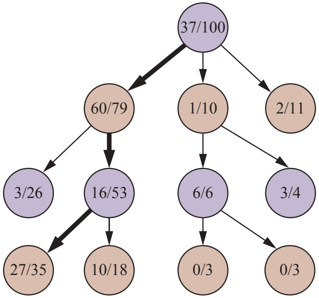
- the 60/79 node:

$$\begin{aligned} UCB1(n) &= \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}} \\ &= \frac{60}{79} + 1.4 \times \sqrt{\frac{\log 100}{79}} \\ &= 1.1 \checkmark \end{aligned}$$

- and the 2/11 node:

$$\begin{aligned} UCB1(n) &= \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}} \\ &= \frac{2}{11} + 1.4 \times \sqrt{\frac{\log 100}{11}} \\ &= 1.09 \end{aligned}$$

UCB1 In Action – Exploration



With $C = 1.5$,

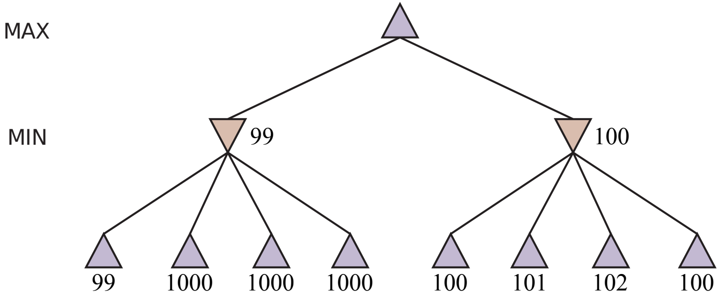
- the 60/79 node

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$
$$= \frac{60}{79} + 1.5 \times \sqrt{\frac{\log 100}{79}}$$
$$= 1.12$$

- and the 2/11 node:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$
$$= \frac{2}{11} + 1.5 \times \sqrt{\frac{\log 100}{11}}$$
$$= 1.15\checkmark$$

Errors in Heuristic Minimax Search



- If evaluation function is 100% correct, right branch is correct.
- If evaluation function has random error with $\sigma = 5$, left is better 71% of the time because one of the four right-hand leaves will dip below 99.
- If evaluation function has random error with $\sigma = 2$, left is better 58% of the time.

Closing Thoughts

The game playing algorithms we considered here have limitations that we will address in furture lessons.

- Vulnerability to errors in the heuristic function. (See Previous slide.)
- Sometimes there is a single clearly best move, but Alpha-beta and MCTS waste computation by calculating values of many legal moves.
  - Better to employ metareasoning about the utility of node expansion – only expand nodes likely to lead to a better move.
- Alpha-beta and MCTS reason about individual moves. Humans reason abstractly, selectively forming plausible plans to acheive (intermediate) goals, e.g., trapping the opponent's queen.
  - We will learn about such approaches when we study **planning**.
- Alpha-beta (and MCTS when depth-limited) rely on human-crafted evaluation functions.
  - Programs like AlphaZero[1] need only the rules of the game to learn how to play at superhuman levels.

[1] https://arxiv.org/pdf/1712.01815