Artificial Intelligence

Adversarial Search

Christopher Simpkins

Games

Economy

Pruning

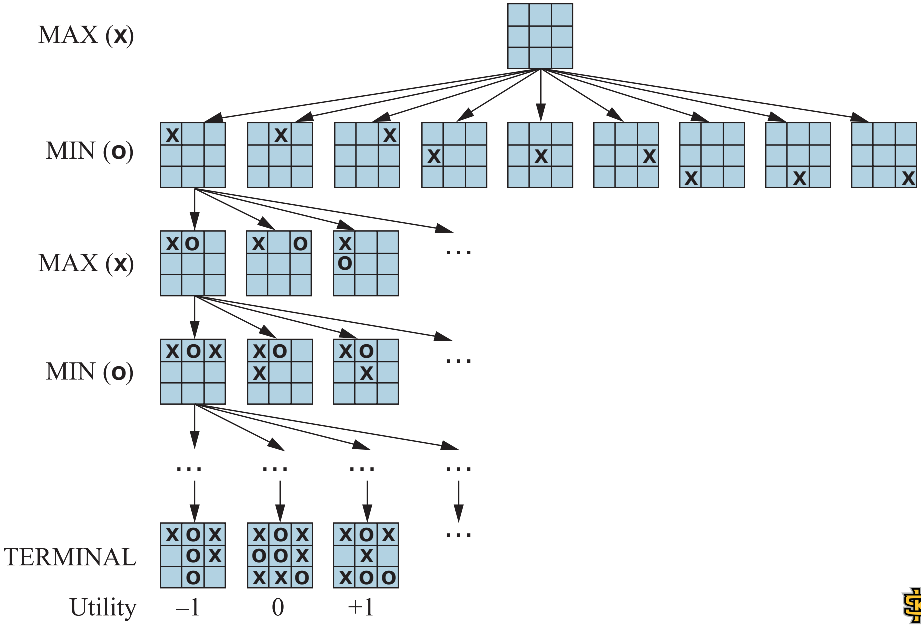
Evaluation function

Two-Player Zero-Sum Games

- 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.
- $Actions(s)$: The set of legal moves in state s.
- $Result(s, a)$: The transition model, which defines the state resulting from taking ac- Transition model tion a in state s.
- $IsTerminal(s)$: A terminal test, which is true when the game is over and false Terminal test otherwise. States where the game has ended are called terminal states. Terminal state
- $Utility(s, p)$: 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 1/2.2 Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

KENNESAW STATE UNIVERSITY

Tic-Tac-Toe Game Tree



Optimal Decisions in Games

Minimax search

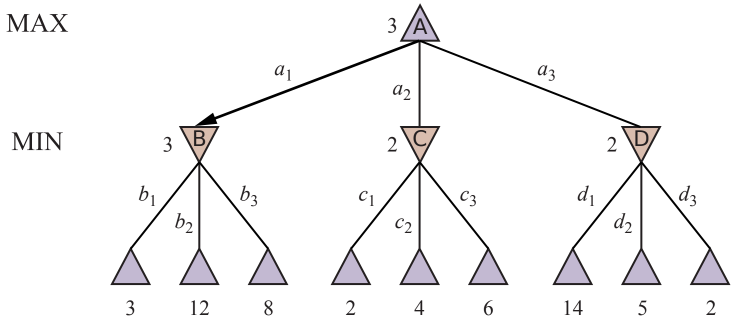
Ply

Minimax value

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

Minimax Decision

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 ← game.TO-MOVE(state)
    value, move ← 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 ← $-\infty$
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if $v2 > v$ then
            v, move ← 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 ← $+\infty$
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if $v2 < v$ then
            v, move ← v2, a
    return v, move
```

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.

- $O(b^m) = 35^80 \sim 10^123$ states

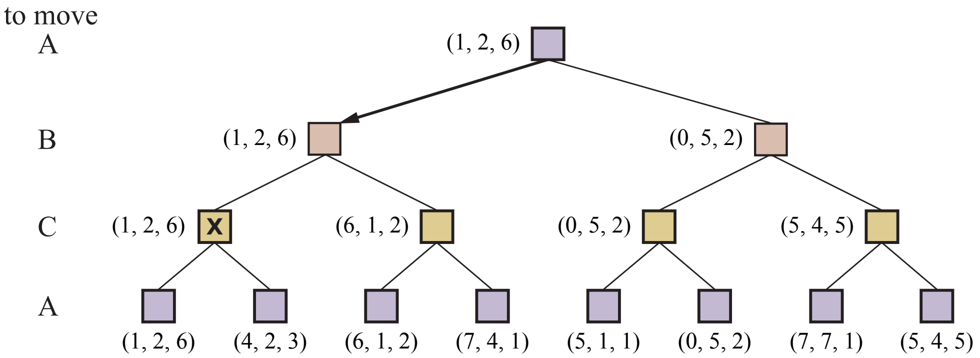
Clearly, minimax won't work for Chess. After we briefly consider multiplayer games we'll learn a modification to minimax that makes games like Chess tractable.

Multiplayer Games

Instead of single value, vector of values

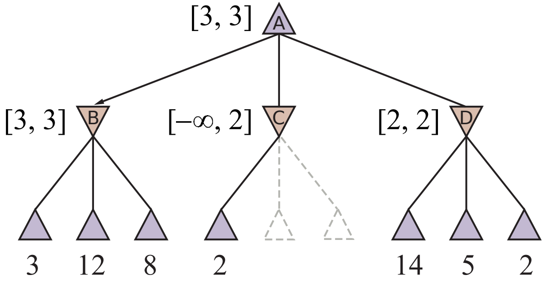
alliances

Three Player Game Tree



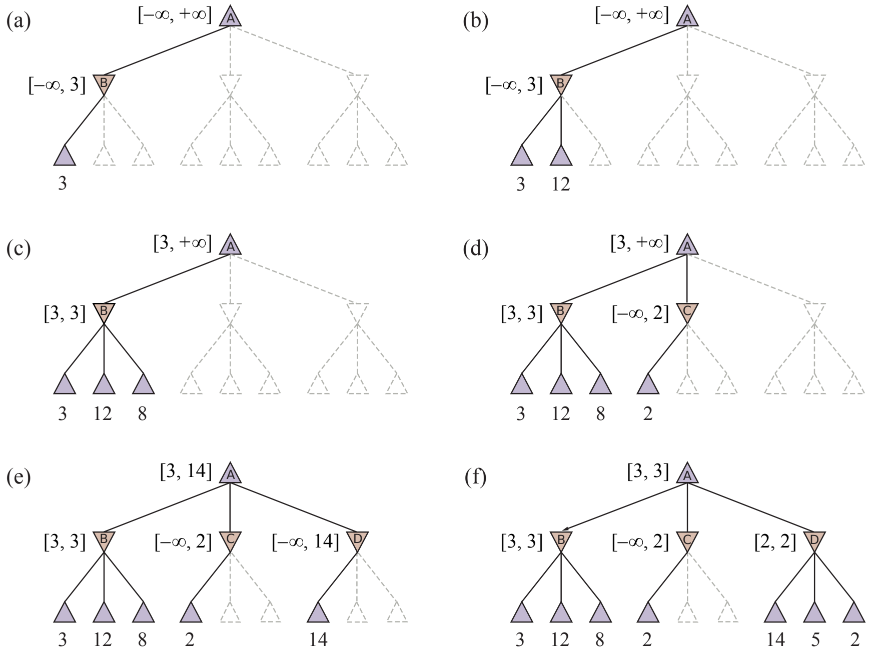
The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha-Beta Pruning

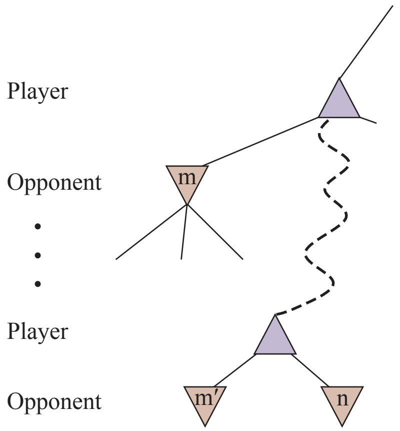


$$\begin{aligned} Minimax(root) &= max(min(3,12,8), min(2,x,y), min(14,5,2)) \\ &= max(3, min(2,x,y), 2) \\ &= max(3,z,2) \text{ where } z = min(2,x,y) \leq 2 \\ &= 3. \end{aligned}$$

Alpha-Beta Calculation Stages



Alpha Beta



If m or m' is better than n for Player, we will never get to n in play.

- α = 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: α = "at least."
- β = 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: β = "at most."

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
  $v \leftarrow -\infty$
  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
      $\alpha \leftarrow$ 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
  $v \leftarrow +\infty$
  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
      $\beta \leftarrow$ MIN($\beta$, v)
    if $v \leq \alpha$ then return v, move
  return v, move
```

KENNESAW STATE UNIVERSITY

Move Ordering

Section 6.2.4

Heuristic Alpha-Beta Tree Search

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

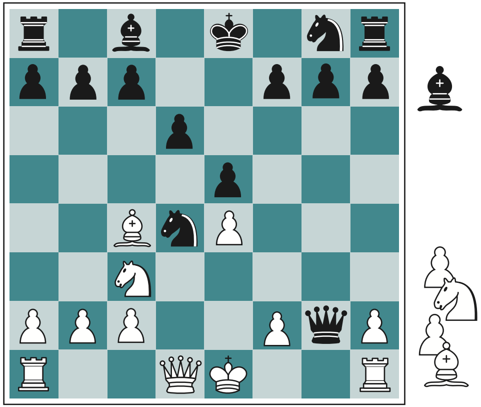
Static Evaluation Functions

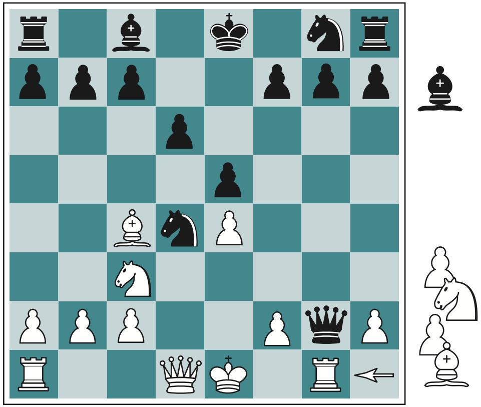
Weighted linear evaluation functions:

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

- 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.

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

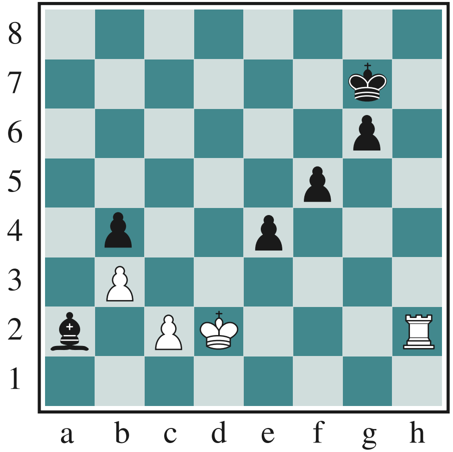
In alpha-beta search algorithm, replace lines with $IsTerminal(state)$ with

```
**if** $game.IsCutoff(state, depth)$ **then return** $game.Eval(state, player), null$
```

quiescent positions

quiescence

Horizon Effect



Monte Carlo Tree Search (MCTS)

Two major weaknesses of heuristic alpha-beta tree search:

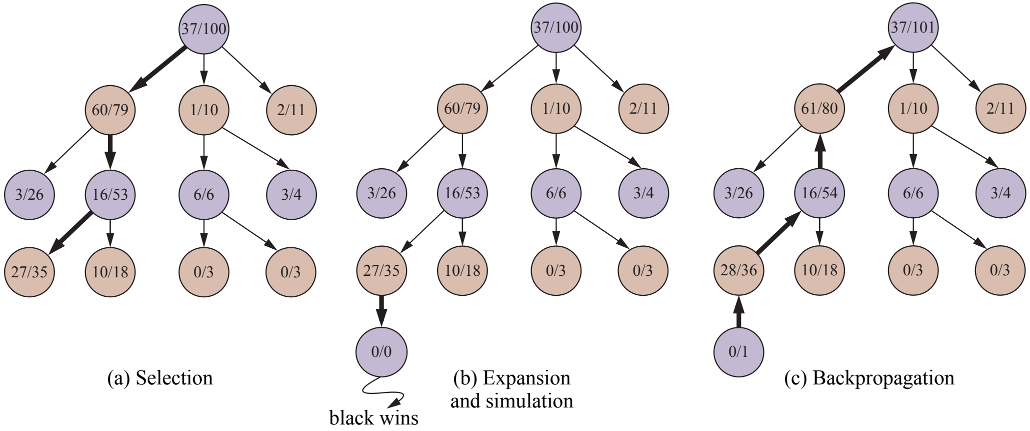
- Can't handle high branching factors. Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
- Can't always define a good static evaluation function. E.g., in Go material value is not a strong indicator and most positions are in flux until the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

. . .

Exploration/Exploitation Tradoff in MCTS

- Selection: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf. Figure 6.10(a) shows a search tree with the root representing a state where white has just moved, and white has won 37 out of the 100 playouts done so far. The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best if we gain more information about it. Selection continues on to the leaf node marked 27/35.
- Expansion: We grow the search tree by generating a new child of the selected node; Figure 6.10(b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)
- Simulation: We 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 figure, the simulation results in a win for black.
- Back-propagation: We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/36 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54, and the root 37/100 becomes 37/101.

MCTS Iteration



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

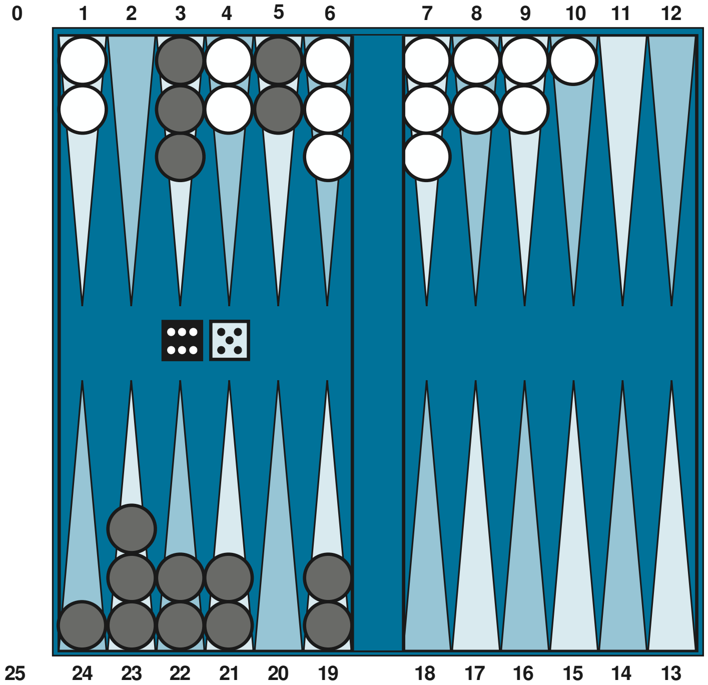
MCTS Algorithm

```
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
```

MCTS Selectoin Policy

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

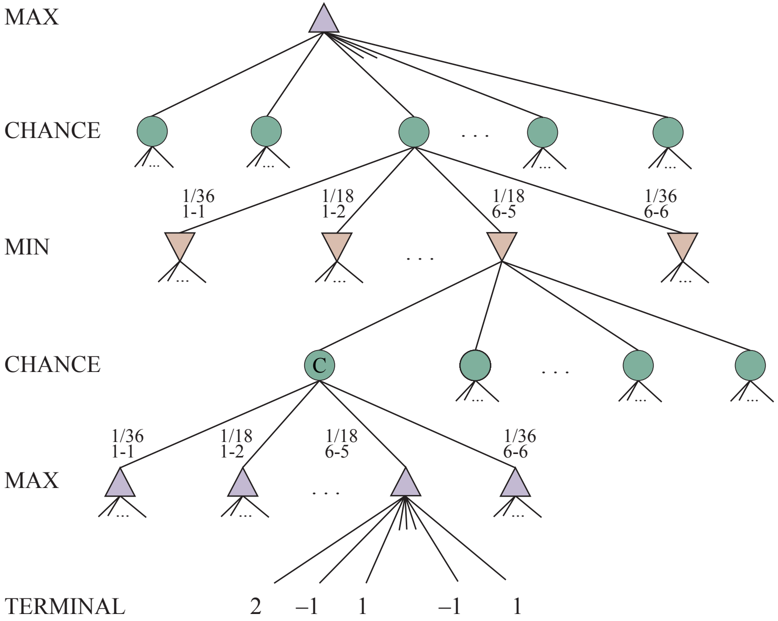
Stochastic Games



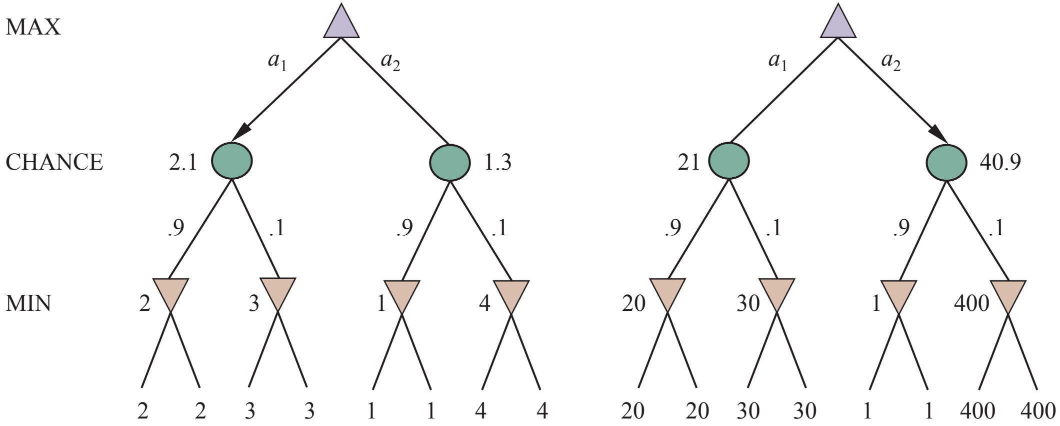
Expectiminimax

$$ExpectiMinimax(s) = \begin{cases} Utility(s, MAX) & \text{if } IsTerminal(s) \\ max_a ExpectiMinimax(Result(s,a)) & \text{if } ToMove(s) = MAX \\ min_a ExpectiMinimax(Result(s,a)) & \text{if } ToMove(s) = MIN \\ \sum_r Pr(r) ExpectiMinimax(Result(s,a)) & \text{if } ToMove(s) = CHANCE \end{cases}$$

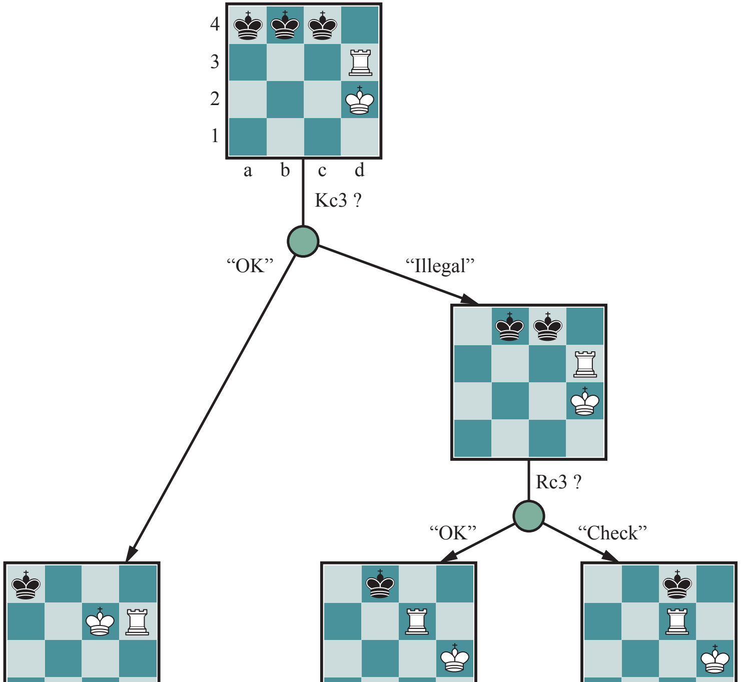
Backgammon Game Tree



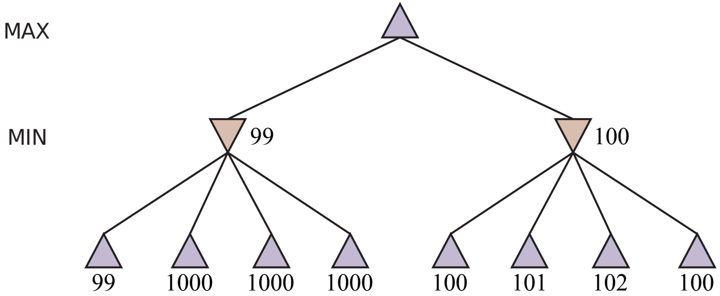
Evaluation FUnctions for Games of Chance



Partially Observable Games



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