UNIVERSITY OF LOUISVILLE J.B. SPEED SCHOOL OF ENGINEERING

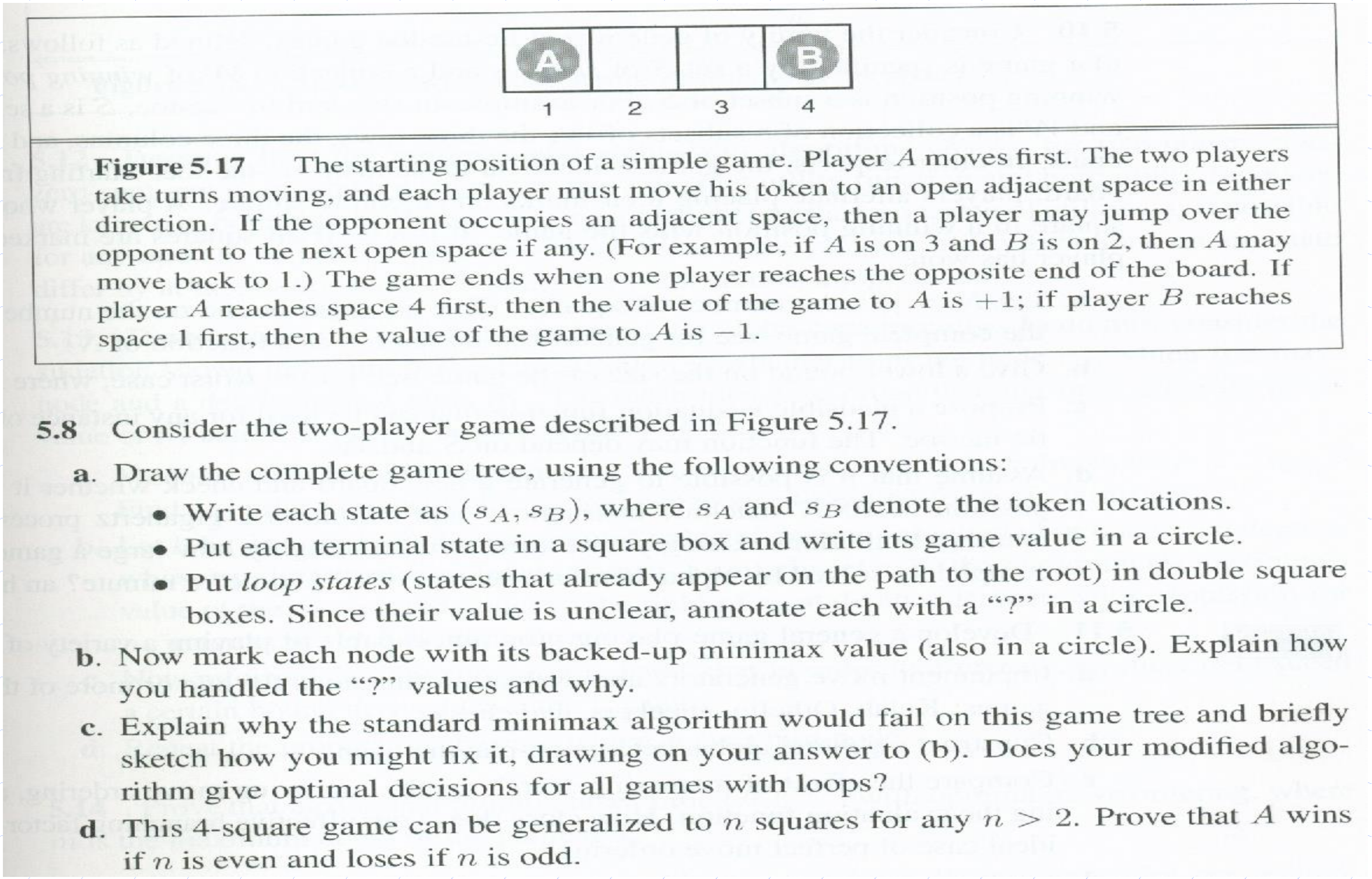
Computer Engineering and Computer Science Department

# CECS545-Artificial Intelligence

**Adversarial Search 2**

Dr. Roman Yampolskiy

# Exercises: 5.8



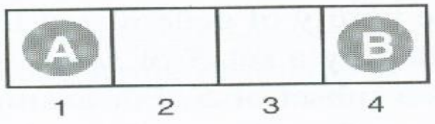


**Figure 5.17** The starting position of a simple game. Player $A$ moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if $A$ is on 3 and $B$ is on 2, then $A$ may move back to 1.) The game ends when one player reaches the opposite end of the board. If player $A$ reaches space 4 first, then the value of the game to $A$ is $+1$; if player $B$ reaches space 1 first, then the value of the game to $A$ is $-1$.

**5.8** Consider the two-player game described in Figure 5.17.

**a.** Draw the complete game tree, using the following conventions:

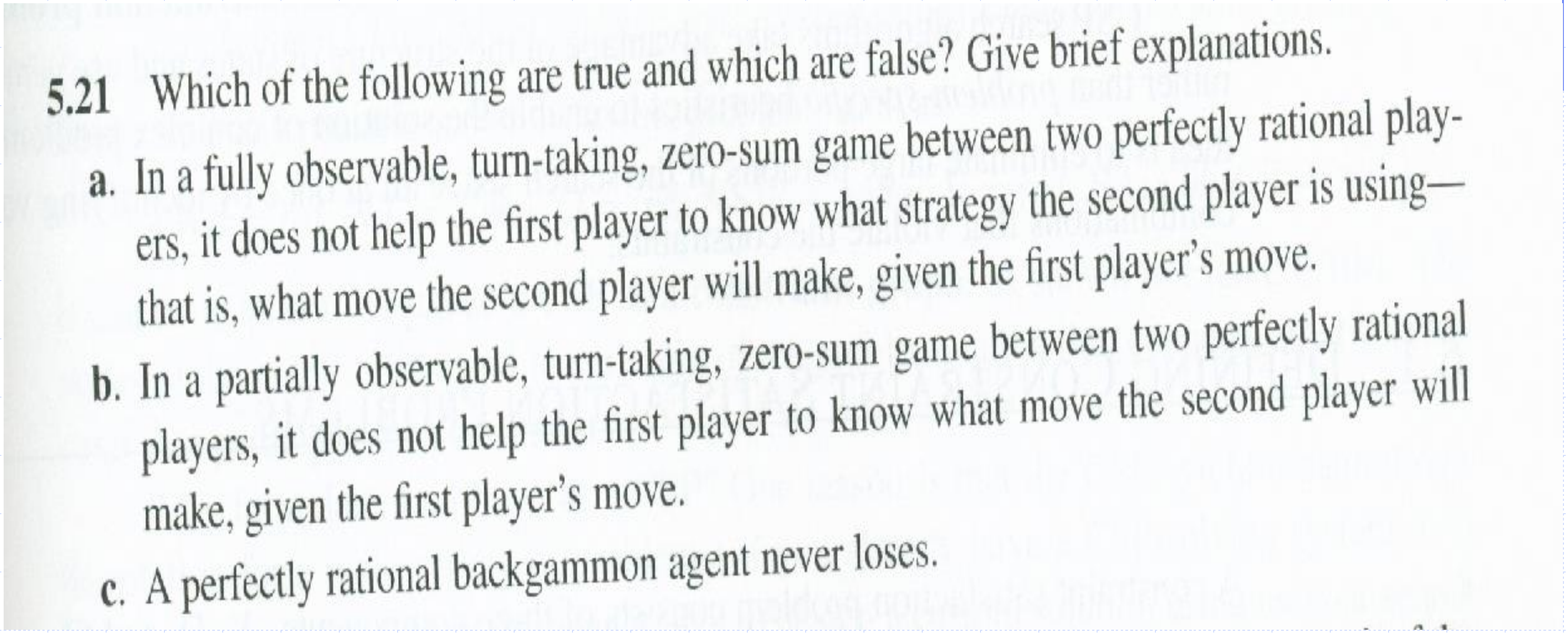
- Write each state as $(s_A, s_B)$, where $s_A$ and $s_B$ denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

**b.** Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

**c.** Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

**d.** This 4-square game can be generalized to $n$ squares for any $n > 2$. Prove that $A$ wins if $n$ is even and loses if $n$ is odd.

## 5.21



5.21 Which of the following are true and which are false? Give brief explanations.

a. In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.

b. In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.

c. A perfectly rational backgammon agent never loses.

## Games vs. search problems

"Unpredictable" opponent ⇒ solution is a strategy
specifying a move for every possible opponent reply

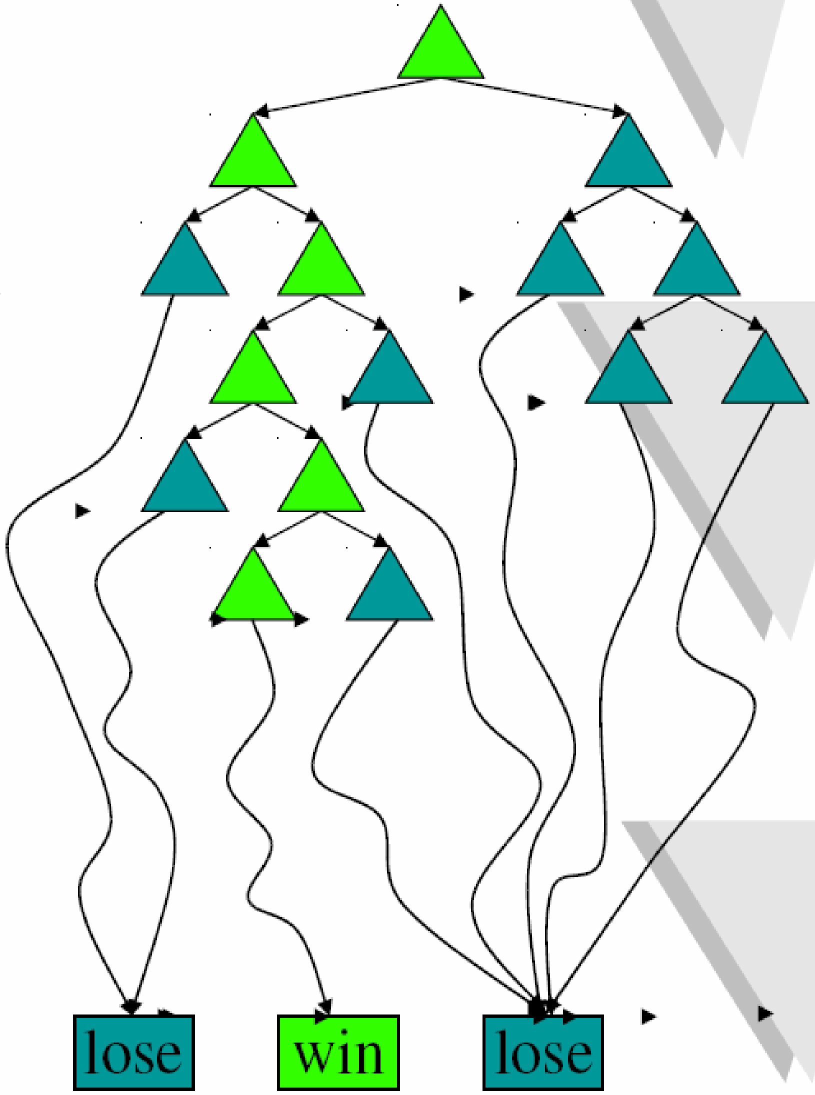
Time limits ⇒ unlikely to find goal, must approximate

Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

# Deterministic Single-Player?

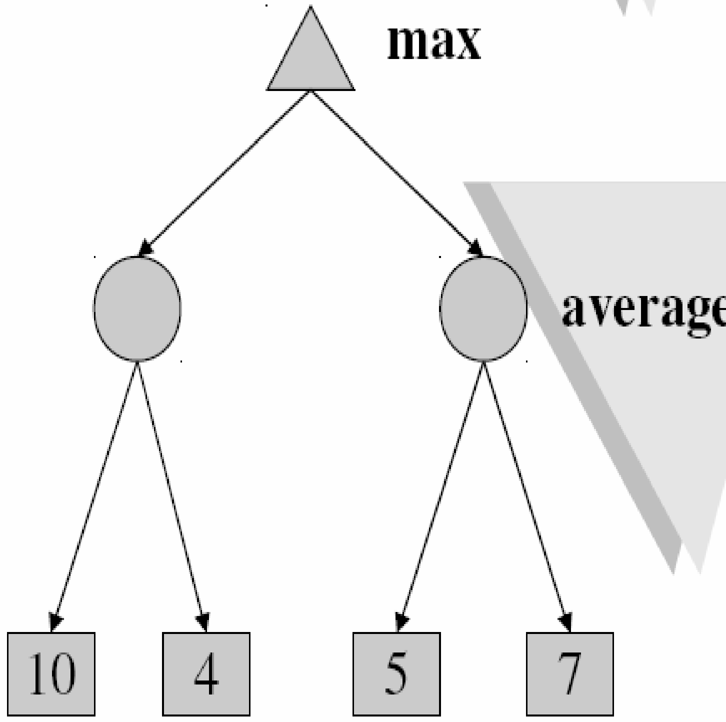
- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik's cube
- … it's just search!
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node



# Stochastic Single-Player

- What if we don't know what the result of an action will be? E.g.,
  - In solitaire, shuffle is unknown
  - In minesweeper, mine locations
  - In pacman, ghosts!

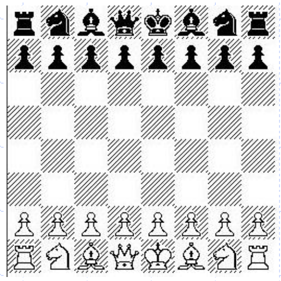
- Can do **expectimax search**
  - Chance nodes, like actions except the environment controls the action chosen
  - Calculate utility for each node
  - Max nodes as in search
  - Chance nodes take average (expectation) of value of children



# Two Player Games

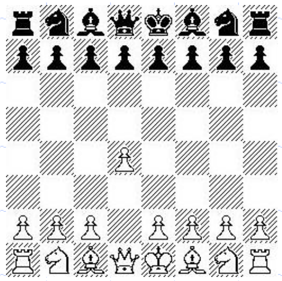
- Competitive rather than cooperative
  - One player loses, one player wins
- Zero sum game
  - One player wins what the other one loses
- Getting an agent to play a game
  - Boils down to how it plays each move
  - Express this as a search problem
    - Cannot backtrack once a move has been made (episodic)

# Basis of Game Playing: Search for best move every time



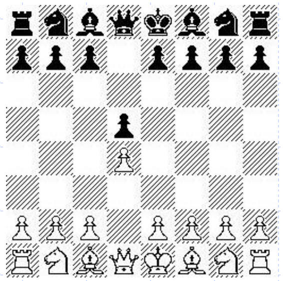
Initial Board State

Search for $\xrightarrow{\hspace{2cm}}$ Move 1



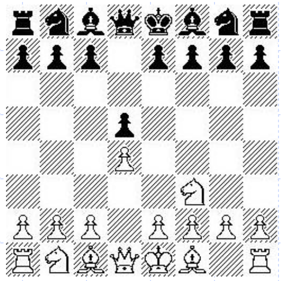
Board State 2

Opponent $\xrightarrow{\hspace{2cm}}$ Moves



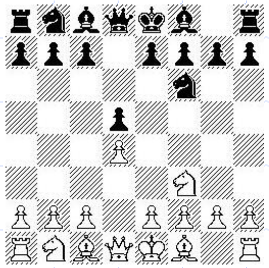
Board State 3

Search for $\xrightarrow{\hspace{2cm}}$ Move 3



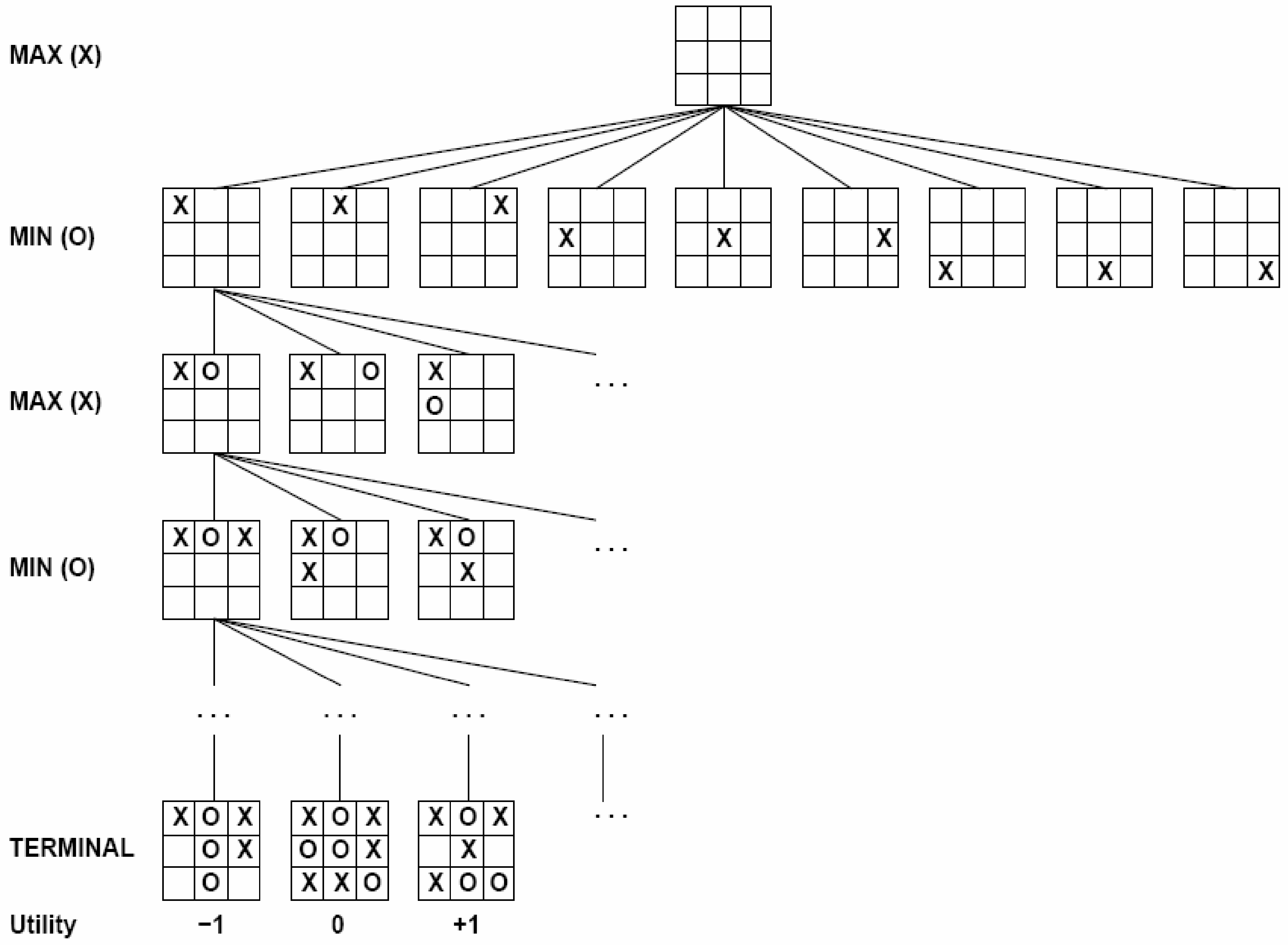
Board State 4

Opponent $\xrightarrow{\hspace{2cm}}$ Moves



Board State 5

## Game tree (2-player, deterministic, turns)



# Lookahead Search

- If I played this move
  - Then they might play that move
    - Then I could do that move
      - And they would probably do that move
  - Or they might play that move
    - Then I could do that move
      - And they would play that move
    - Or I could play that move
      - And they would do that move
- If I played this move…

## Lookahead Search (best moves)

- If I played this move
  - Then their best move would be
    - Then my best move would be
      - Then their best move would be
  - Or another good move for them is…
    - Then my best move would be
      - Etc.

## Minimax Search

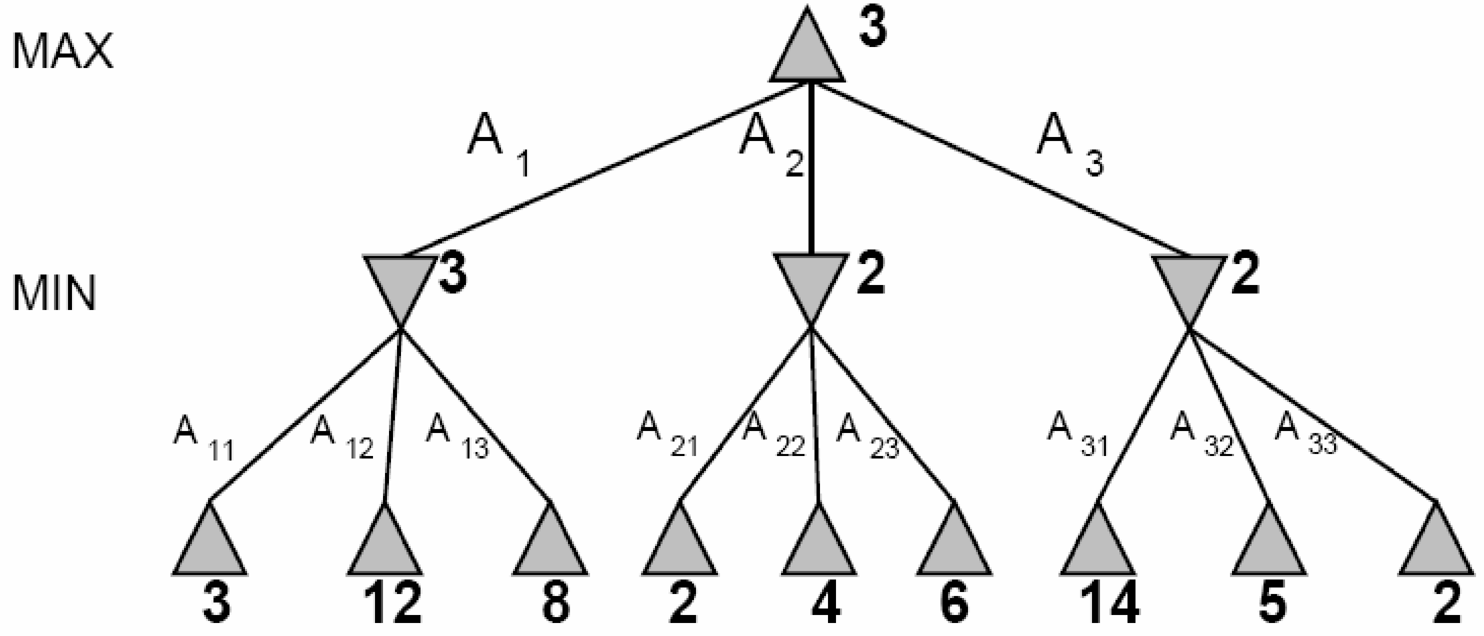
- Like children sharing a cake
- Underlying assumption
  - Opponent acts rationally
- Each player moves in such a way as to
  - Maximise their final winnings, minimise their losses
  - i.e.,  play the best move at the time
- Method:
  - Calculate the guaranteed final scores for each move
    - Assuming the opponent will try to minimise that score move that maximises this guaranteed score
  - Choose

# Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



# Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MIN(v, MAX-VALUE(s))
    return v
```

## Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

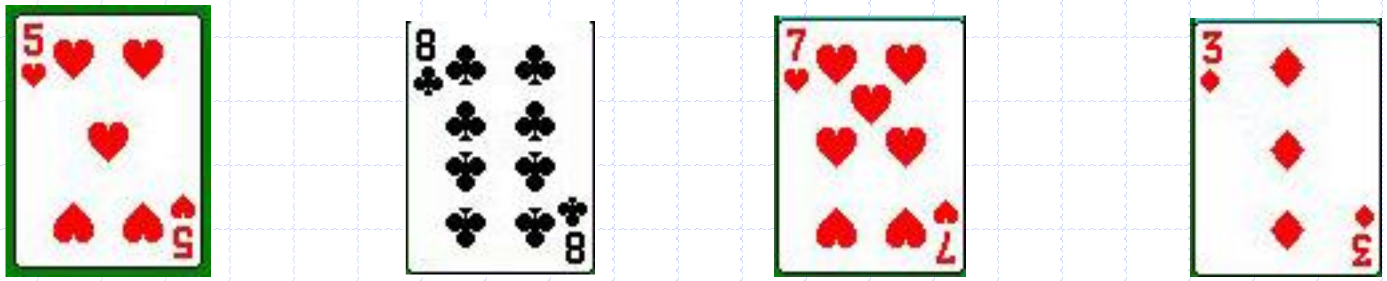
Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

But do we need to explore every path?

# Example Trivial Game

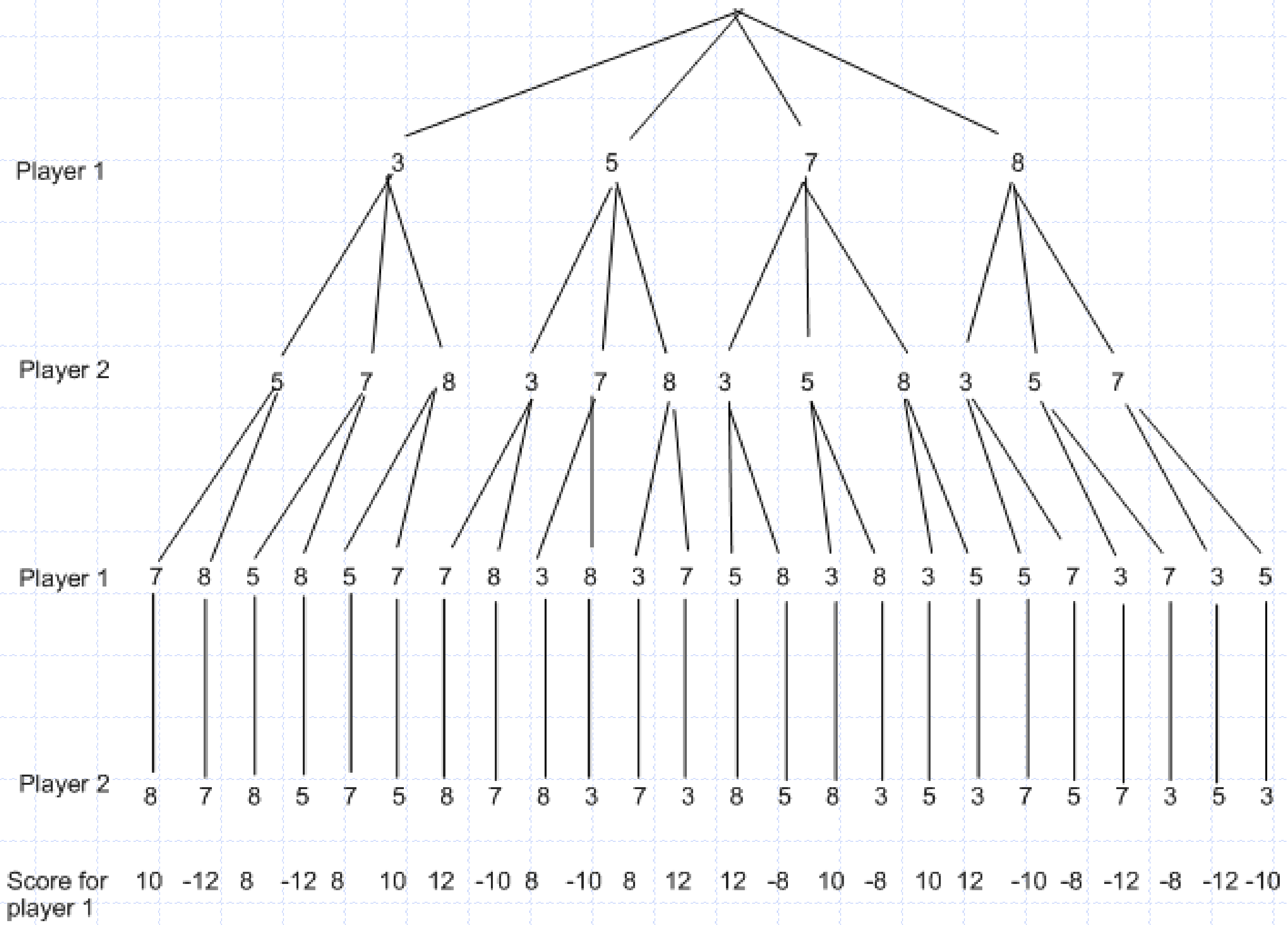
- Deal four playing cards out, face up
- Player 1 chooses one, player 2 chooses one
  - Player 1 chooses another, player 2 chooses another
- And the winner is….
  - Add the cards up
  - The player with the highest <u>even</u> number
    - Scores that amount (in pounds sterling from opponent)



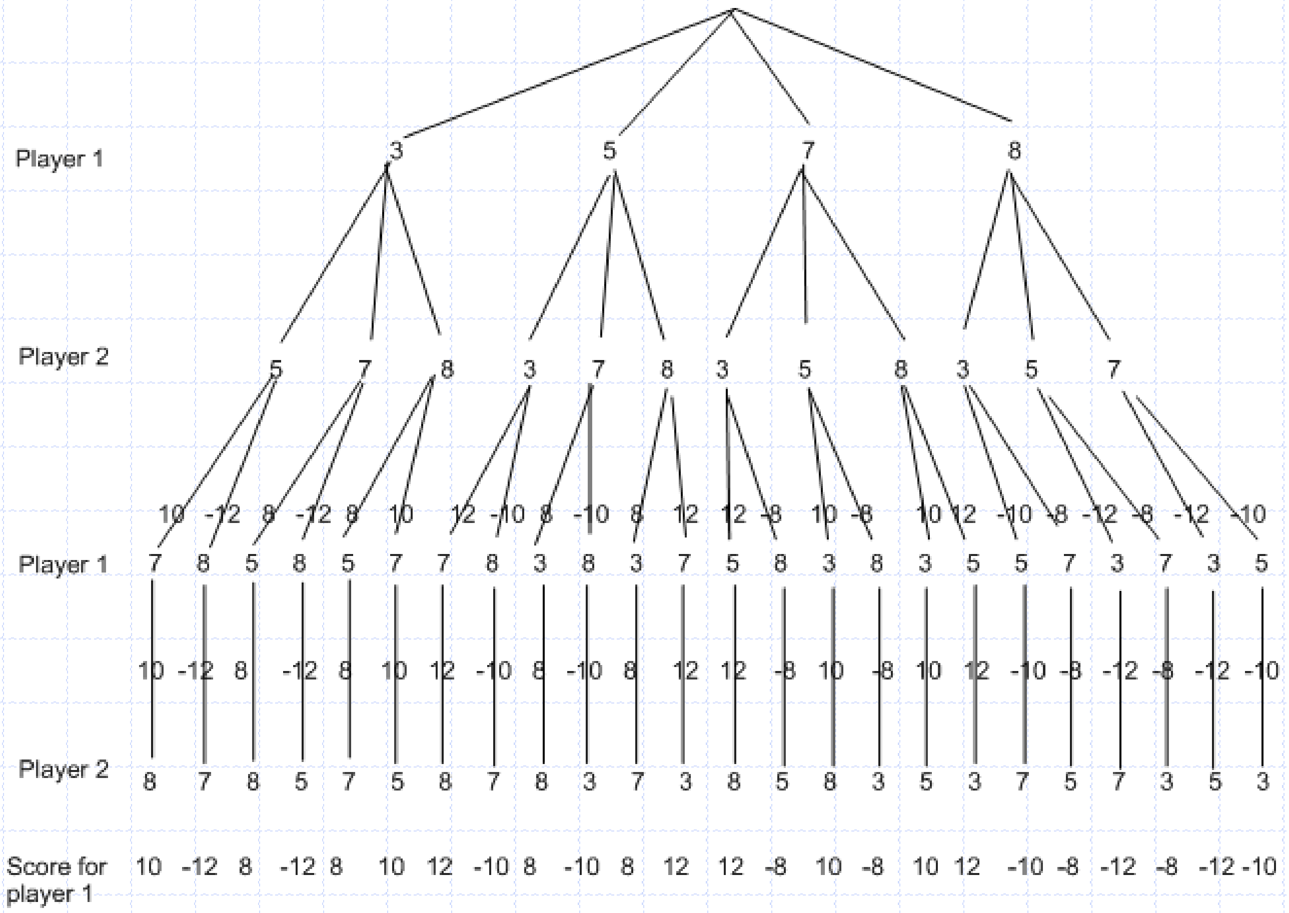
## For Trivial Games

- Draw the entire search space
- Put the scores associated with each final board state at the ends of the paths
- Move the scores from the ends of the paths to the starts of the paths
  - Whenever there is a choice use minimax assumption
  - This guarantees the scores you can get
- Choose the path with the best score at the top
  - Take the first move on this path as the next move

# Entire Search Space

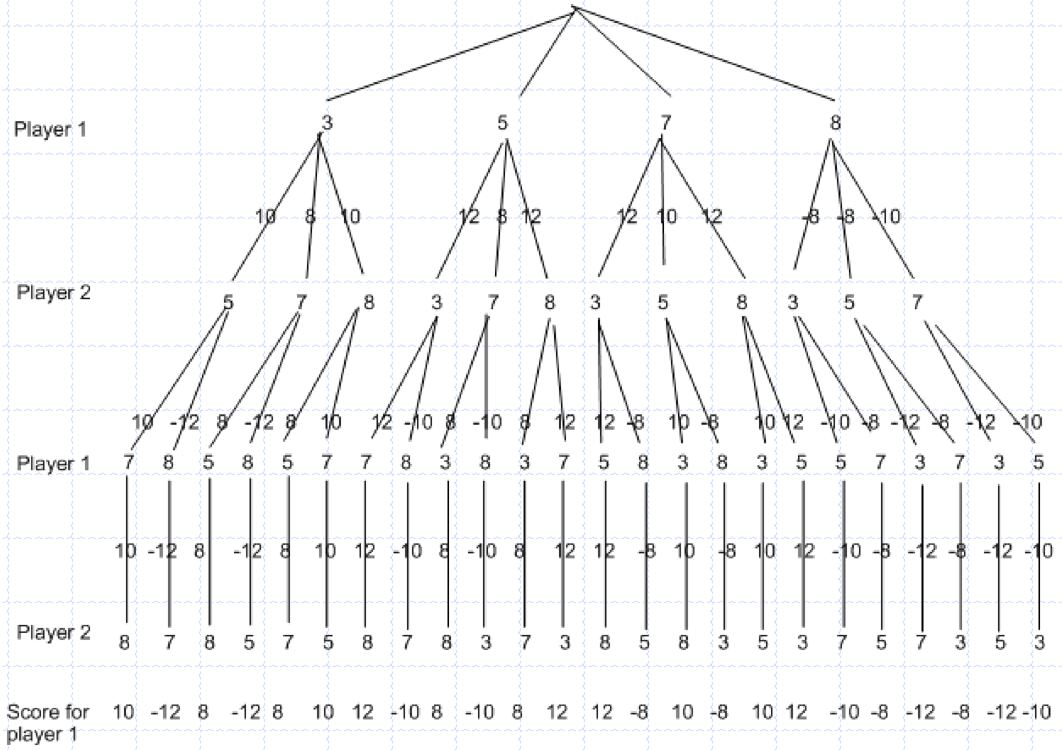


# Moving the scores from the bottom to the top

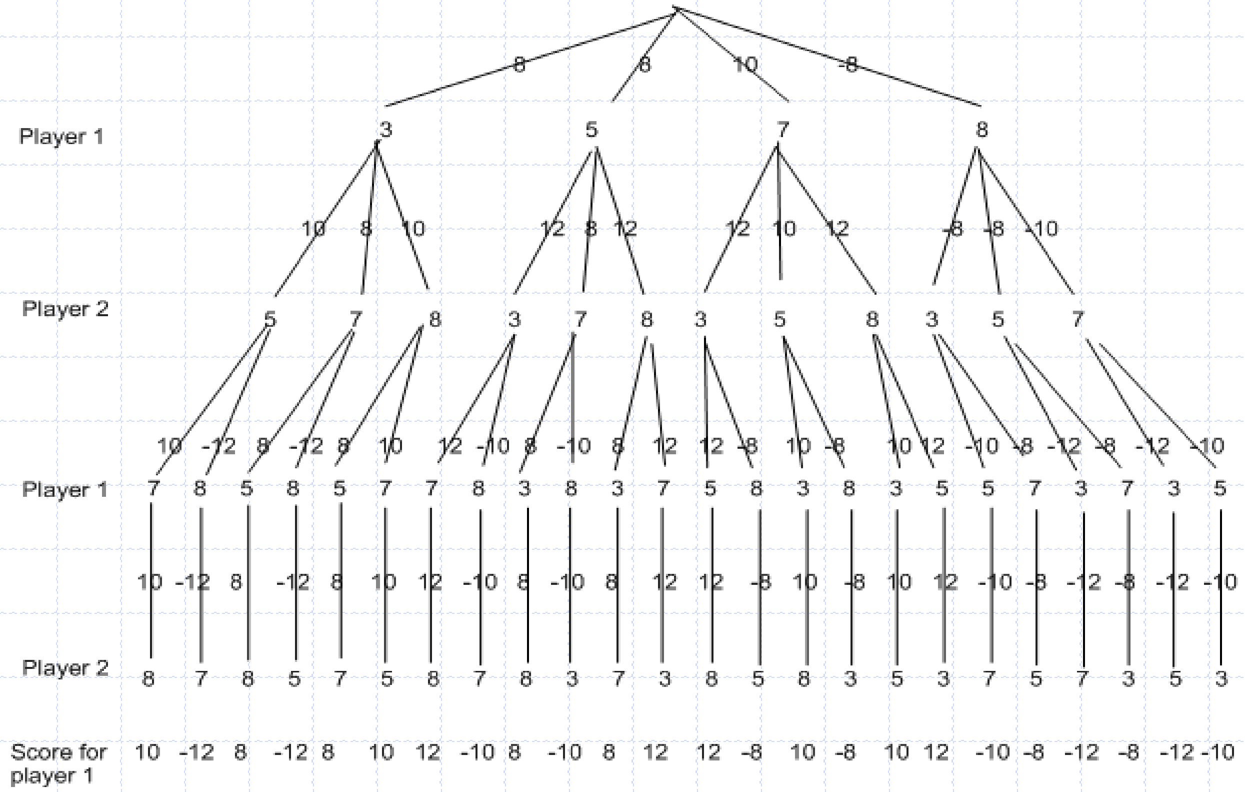


# Moving a score when there's a choice

- Use minimax assumption
  - Rational choice for the player <u>below</u> the number you're moving



# Choosing the best move



## For Real Games

- Search space is too large
  - So we cannot draw (search) the entire space
- For example: chess has branching factor of ~35
  - Suppose our agent searches 1000 board states per second
  - And has a time limit of 150 seconds
    - So can search 150,000 positions per move
  - This is only three or four ply look ahead
    - Because $35^3 = 42,875$ and $35^4 = 1,500,625$
  - Average humans can look ahead six-eight ply

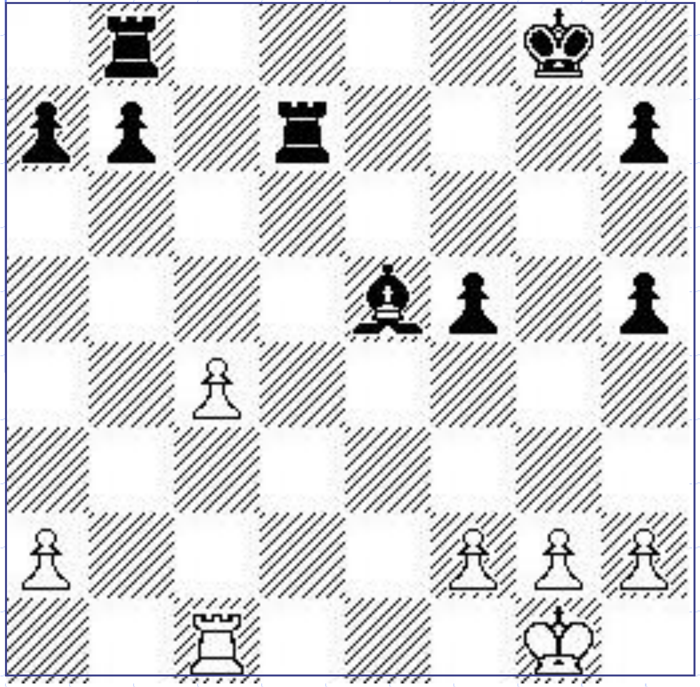
## Cutoff Search

- Must use a heuristic search
- Use an evaluation function
  - Estimate the guaranteed score from a board state
- Draw search space to a certain depth
  - Depth chosen to limit the time taken
- Put the estimated values at the end of paths
- Propagate them to the top as before
- Question:
  - Is this a uniform path cost, greedy or A* search?

## Evaluation Functions

- Must be able to differentiate between
  - Good and bad board states
  - Exact values not important
  - Ideally, the function would return the true score
    - For goal states
- Example in chess
  - **Weighted linear function**
  - Weights:
    - Pawn=1, knight=bishop=3, rook=5, queen=9

# Example Chess Score



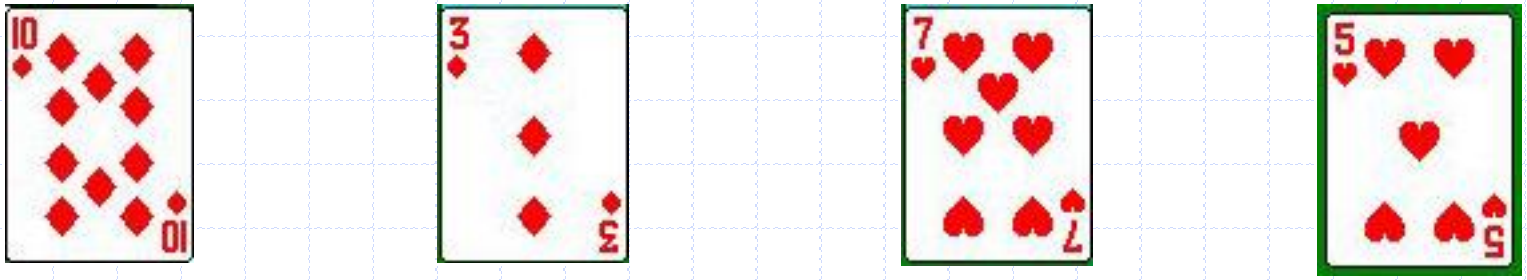
- Black has:
  - 5 pawns, 1 bishop, 2 rooks
- Score = 1*(5)+3*(1)+5*(2)
  = 5+3+10 = 18

White has:
  - 5 pawns, 1 rook
- Score = 1*(5)+5*(1)
  = 5 + 5 = 10

Overall scores for this board state:
black = 18-10 = 8
white = 10-18 = -8

# Evaluation Function for our Game

- Evaluation after the first move
  - Count zero if it’s odd, take the number if its even



- Evaluation function here would choose 10
  - But this would be disastrous for the player

## Problems with Evaluation Functions

- Horizon problem
  - Agent cannot see far enough into search space
    - Potentially disastrous board position after seemingly good one
- Possible solution
  - Reduce the number of initial moves to look at
    - Allows you to look further into the search space
- Non-quiescent search
  - Exhibits big swings in the evaluation function
  - E.g., when taking pieces in chess
  - Solution: advance search past non-quiescent part

# The $\alpha$–$\beta$ algorithm

```
function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

---

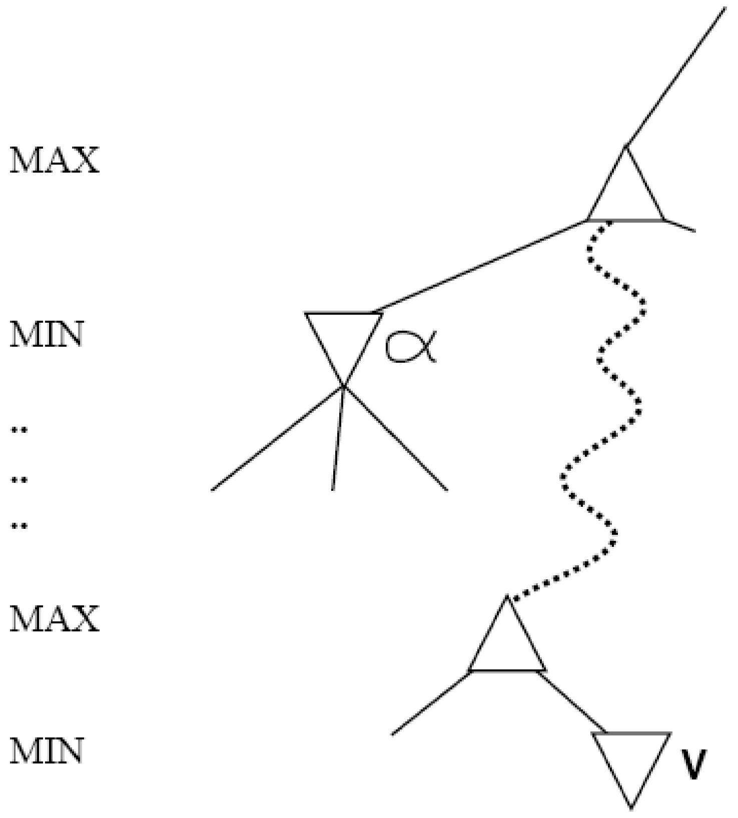
```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

---

```
function MIN-VALUE(state, α, β) returns a utility value
    same as MAX-VALUE but with roles of α, β reversed
```

# Why is it called $\alpha$–$\beta$?



$\alpha$ is the best value (to MAX) found so far off the current path

If $V$ is worse than $\alpha$, MAX will avoid it $\Rightarrow$ prune that branch

Define $\beta$ similarly for MIN

## Properties of $\alpha$–$\beta$

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, $35^{50}$ is still impossible!

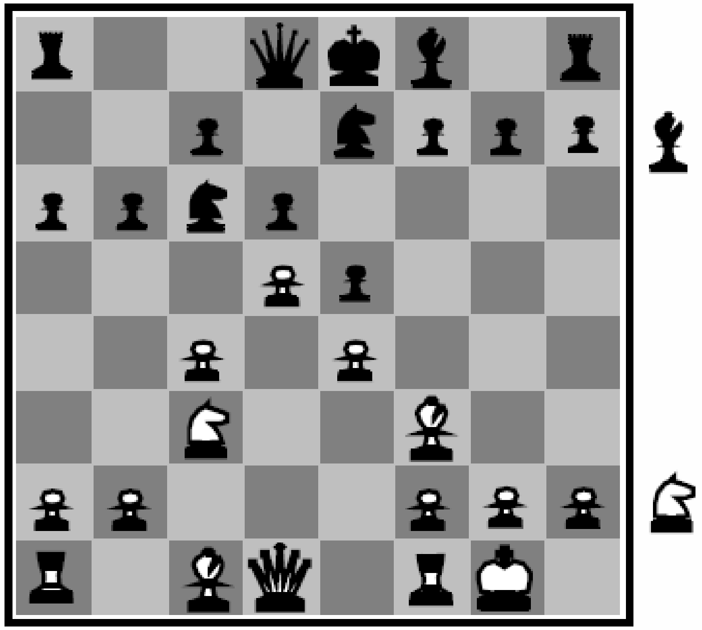
## Resource limits

Standard approach:

- Use CUTOFF-TEST instead of TERMINAL-TEST
  e.g., depth limit (perhaps add quiescence search)
- Use EVAL instead of UTILITY
  i.e., evaluation function that estimates desirability of position

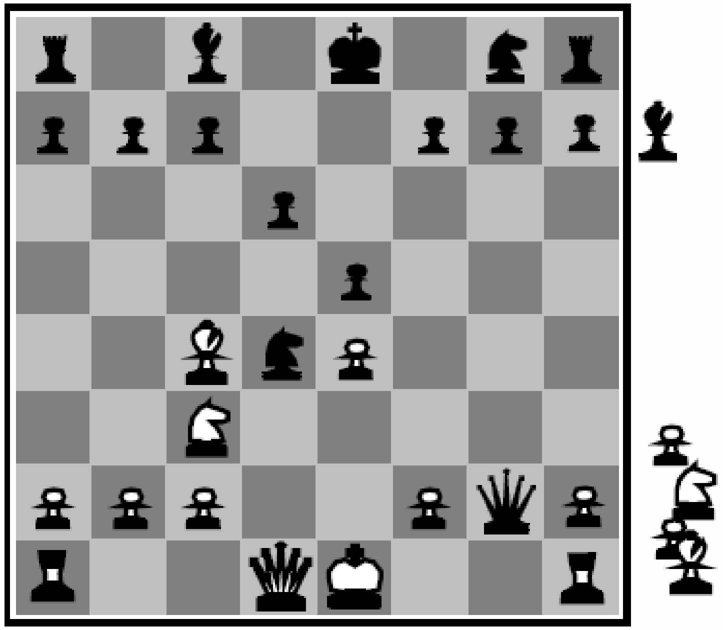
Suppose we have $100$ seconds, explore $10^4$ nodes/second
$\Rightarrow$ $10^6$ nodes per move $\approx 35^{8/2}$
$\Rightarrow$ $\alpha$–$\beta$ reaches depth 8 $\Rightarrow$ pretty good chess program

# Evaluation functions



Black to move

White slightly better



White to move

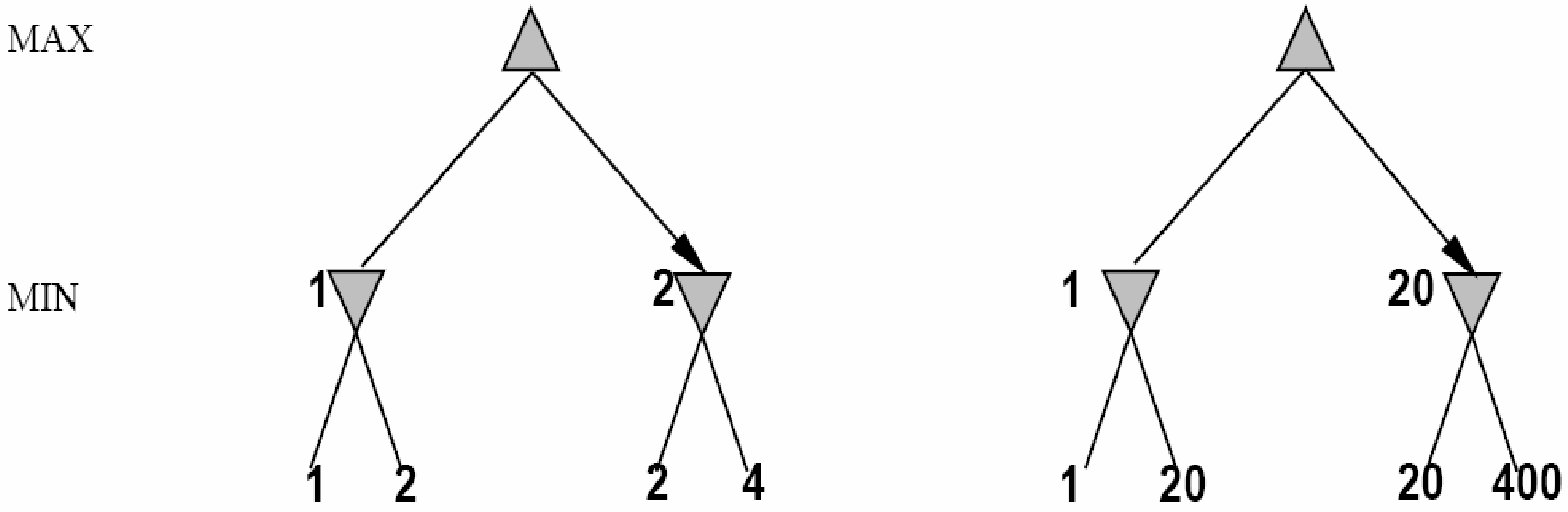
Black winning

For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s)$ = (number of white queens) – (number of black queens), etc.

# Digression: Exact values don't matter



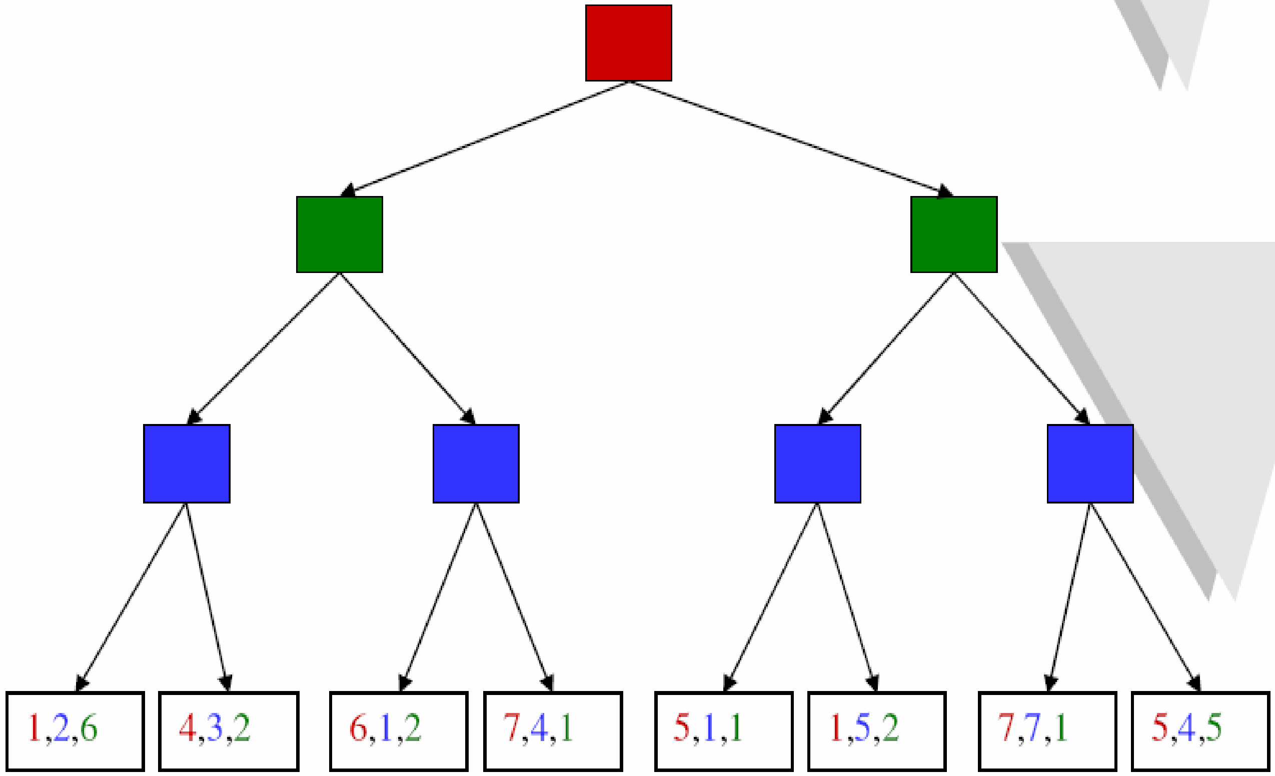
Behaviour is preserved under any monotonic transformation of EVAL

Only the order matters:

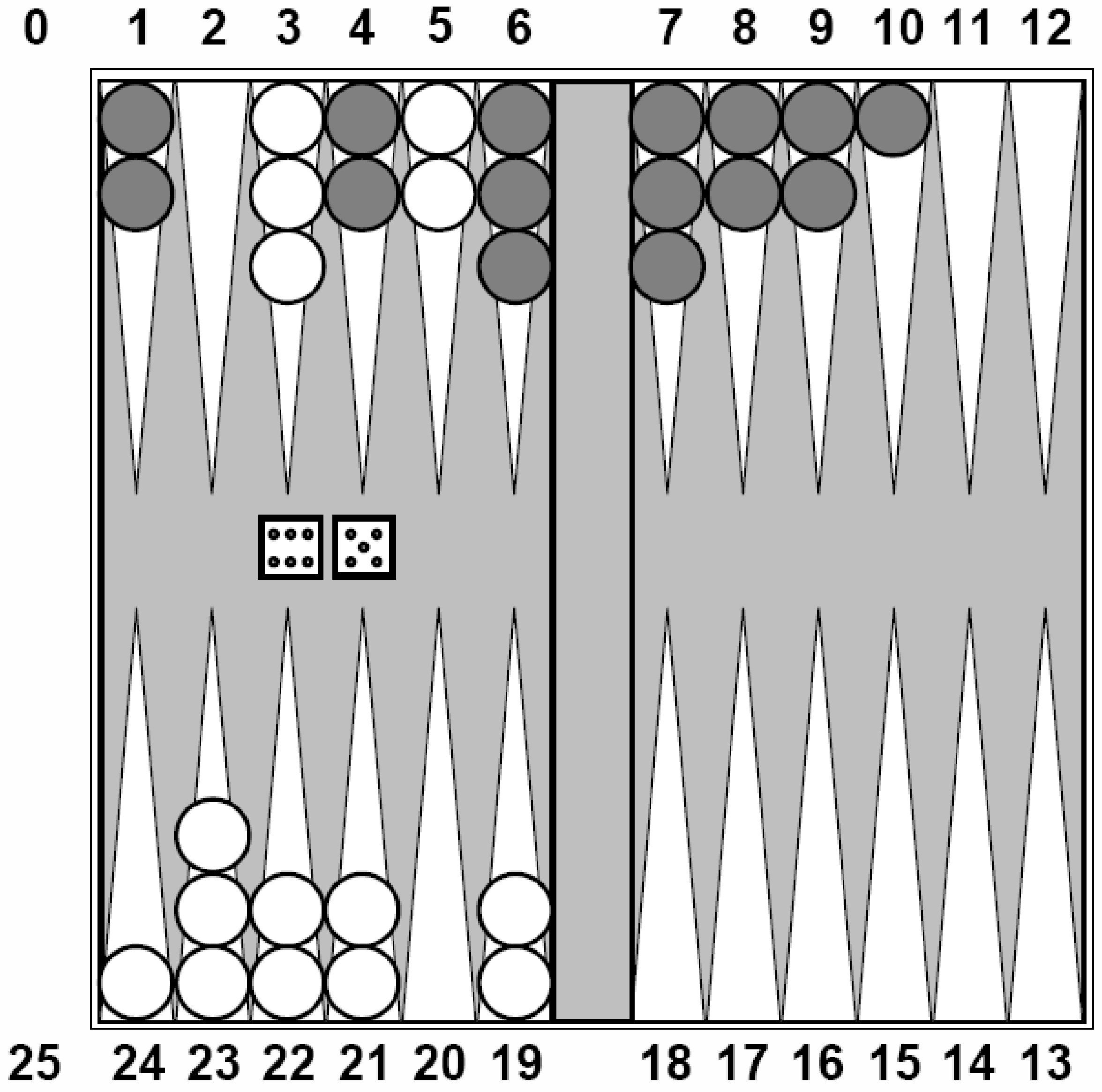
payoff in deterministic games acts as an ordinal utility function

# Non-Zero-Sum Games

- Similar to minimax:
  - Utilities are now tuples
  - Each player maximizes their own entry at each node
  - Propagate (or back up) nodes from children



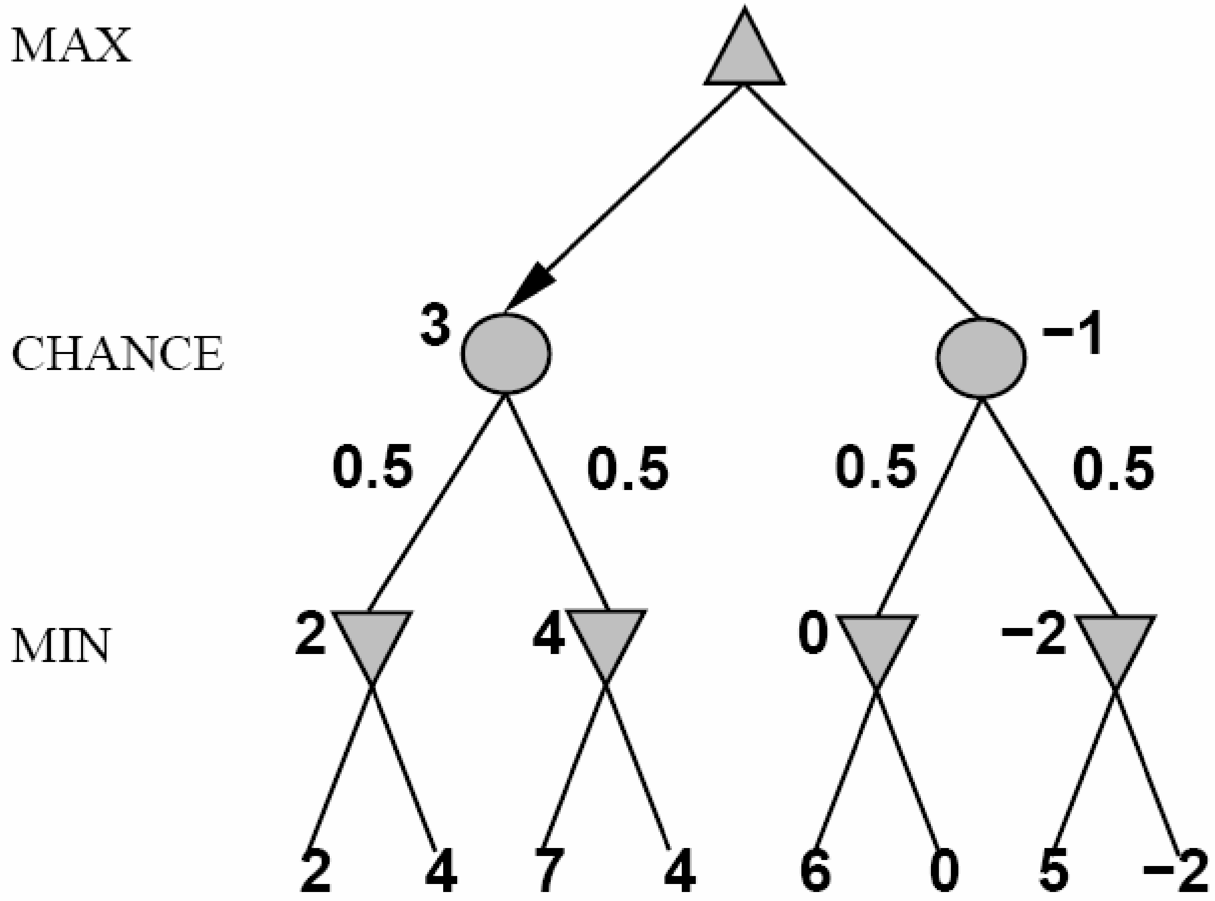
# Nondeterministic games: backgammon



# Nondeterministic games in general

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



## Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

```
. . .
if state is a MAX node then
    return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
    return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
. . .
```

## Nondeterministic games in practice

Dice rolls increase $b$: 21 possible rolls with 2 dice
Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

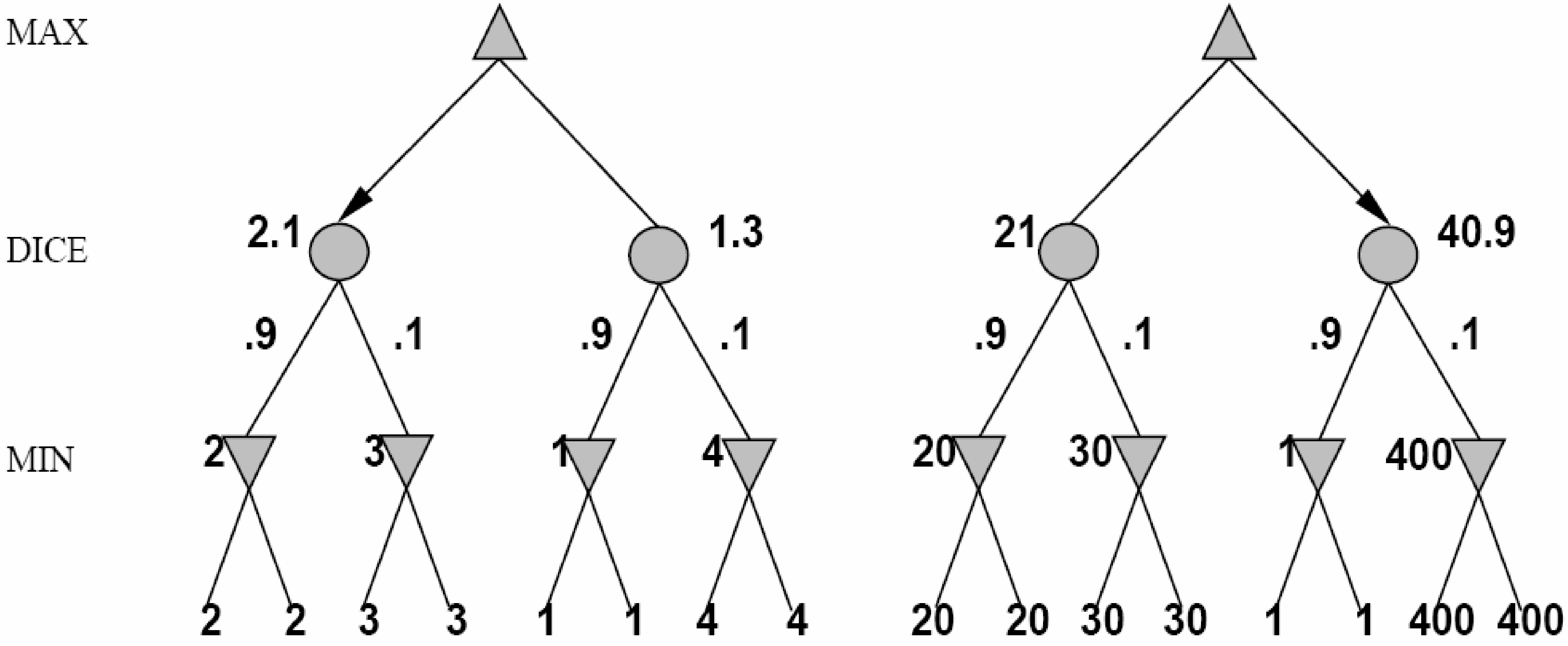
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
$\Rightarrow$ value of lookahead is diminished

$\alpha$–$\beta$ pruning is much less effective

TDGammon uses depth-2 search + very good Eval
$\approx$ world-champion level

# Digression: Exact values DO matter



Behaviour is preserved only by positive linear transformation of EVAL

Hence EVAL should be proportional to the expected payoff

# Games of imperfect information

E.g., card games, where opponent's initial cards are unknown

Typically we can calculate a probability for each possible deal

Seems just like having one big dice roll at the beginning of the game*

Idea: compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals*

Special case: if an action is optimal for all deals, it's optimal.*

GIB, current best bridge program, approximates this idea by
1) generating 100 deals consistent with bidding information
2) picking the action that wins most tricks on average

## Proper analysis

* Intuition that the value of an action is the average of its values in all actual states is **WRONG**

With partial observability, value of an action depends on the information state or belief state the agent is in

Can generate and search a tree of information states

Leads to rational behaviors such as
- ◇ Acting to obtain information
- ◇ Signalling to one's partner
- ◇ Acting randomly to minimize information disclosure

## The End!