Adversarial Search

Toolbox so far

- Uninformed search
  - BFS, DFS, uniform cost search
- Heuristic search
  - A*

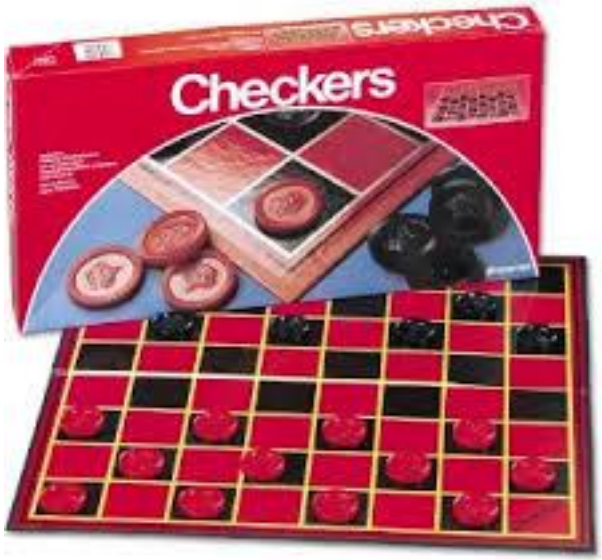
Common environmental factors: static, discrete, fully observable, deterministic actions.
Also: single agent, non-episodic.

There's More!

- Add a second agent, but not controlled by us.
- Assume this agent is our adversary.
- Environment (for now)
  - Still static
  - Still discrete
  - Still fully observable (for now)
  - Still deterministic (for now)

Games!

- Deterministic, turn-taking, two-player, zero-sum games of perfect information.



2007

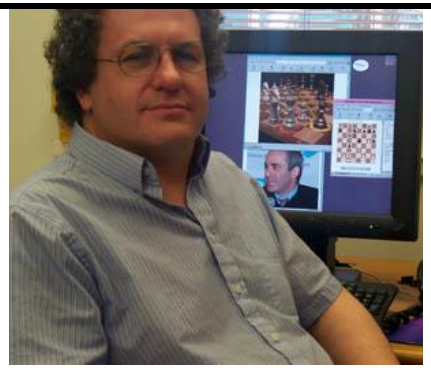
Checkers Is Solved

Jonathan Schaeffer,* Neil Burch, Yngvi Björnsson,† Akihiro Kishimoto,‡
Martin Müller, Robert Lake, Paul Lu, Steve Sutphen

The game of checkers has roughly 500 billion billion possible positions (5×10^{20}). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving

best known is the four-color theorem (*9*). This deceptively simple conjecture—that given an arbitrary map with countries, you need at most four different colors to guarantee that no two adjoining countries have the same color—has been extremely difficult to prove analytically. In 1976, a computational proof was demonstrated. Despite the convincing result, some mathematicians were skeptical, distrusting proofs that had not been verified using human-derived theorems. Although important components of the checkers

The game of checkers has roughly 500 billion billion possible positions (5×10^{20}). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving checkers, applying state-of-the-art artificial intelligence techniques to the proving process. This paper announces that checkers is now solved: Perfect play by both sides leads to a draw. This is the most challenging popular game to be solved to date, roughly one million times as complex as Connect Four. Artificial intelligence technology has been used to generate strong heuristic-based game-playing programs, such as Deep Blue for chess. Solving a game takes this to the next level by replacing the heuristics with perfection.



UNIVERSITY OF ALBERTA
EDMONTON, ALBERTA, CANADA

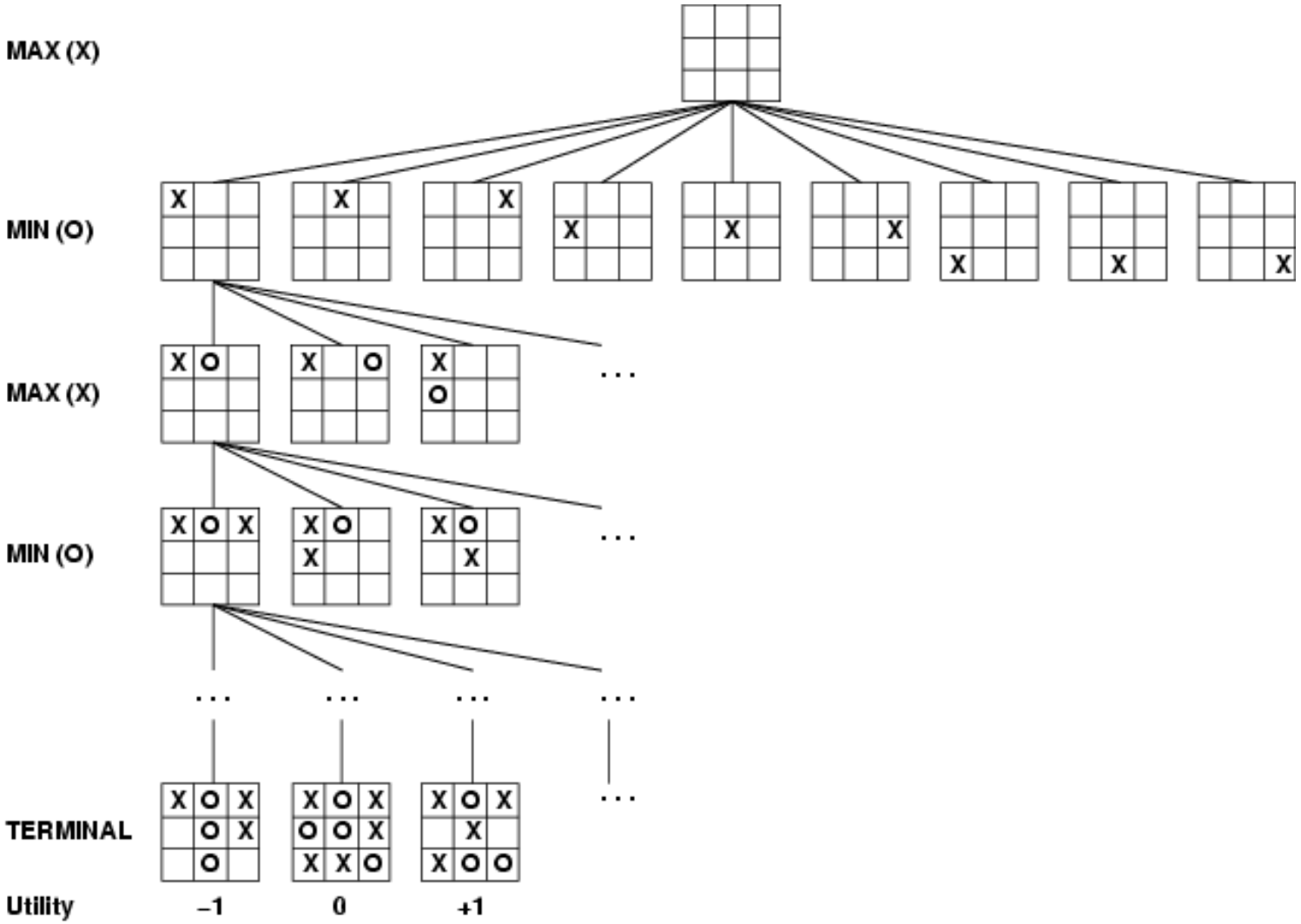
DEPARTMENT OF COMPUTING SCIENCE

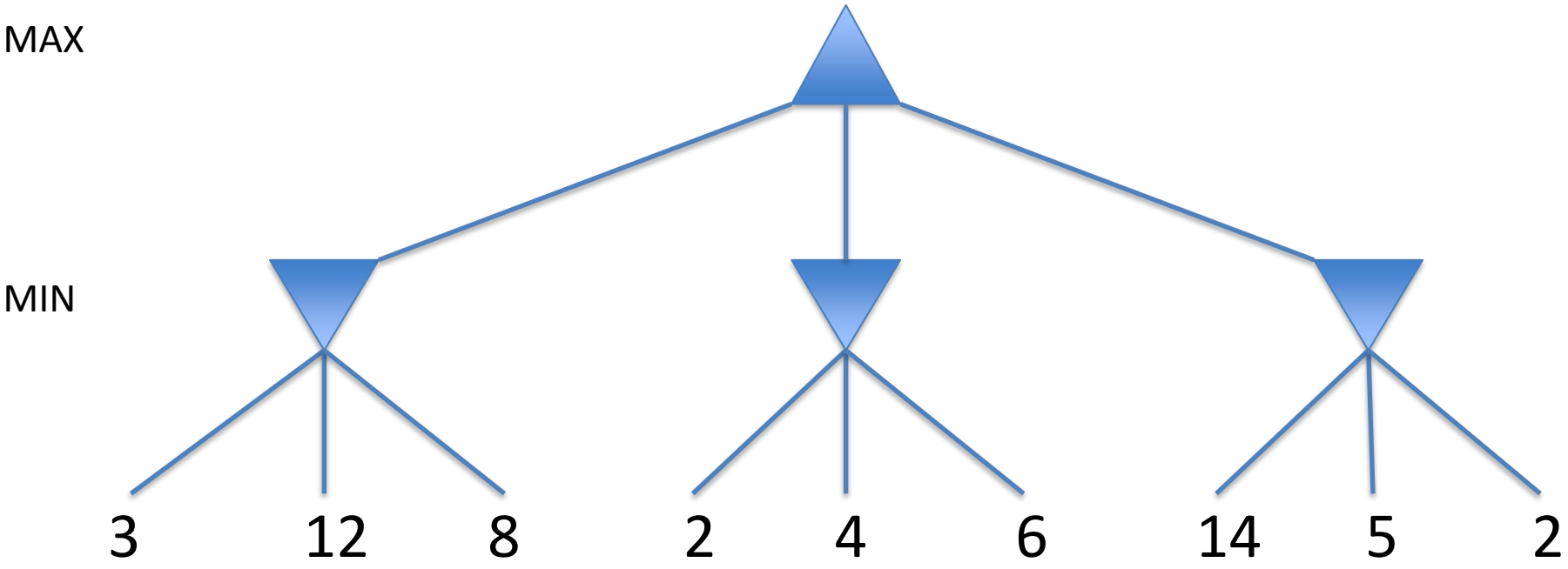
Adversarial search

- Still search!
  - But another agent will alternate actions with us.
- Main new concept:
  - Two players are called MAX and MIN.
  - Only works for zero-sum games.
    - Strictly competitive (no cooperation).
    - What is good for me is equally bad for my opponent (in regards to winning and losing).
  - Most “normal” 2-player games are zero-sum.

- Most all of our concepts from state-space search transfer here.
- S_0: initial state
- TO-MOVE(s): Defines who makes the next move at a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): Returns what state you go into (*transition model*)
- IS-TERMINAL(s): Returns true if s is a *terminal state*.
- UTILITY(s, p): Numeric value of a terminal state s for player p.

Game Tree




MAX
MIN
3
12
8
2
4
6
14
5
2

Minimax algorithm

- Select the best move for you, assuming your opponent is selecting the best move for themselves.
- Works like DFS.

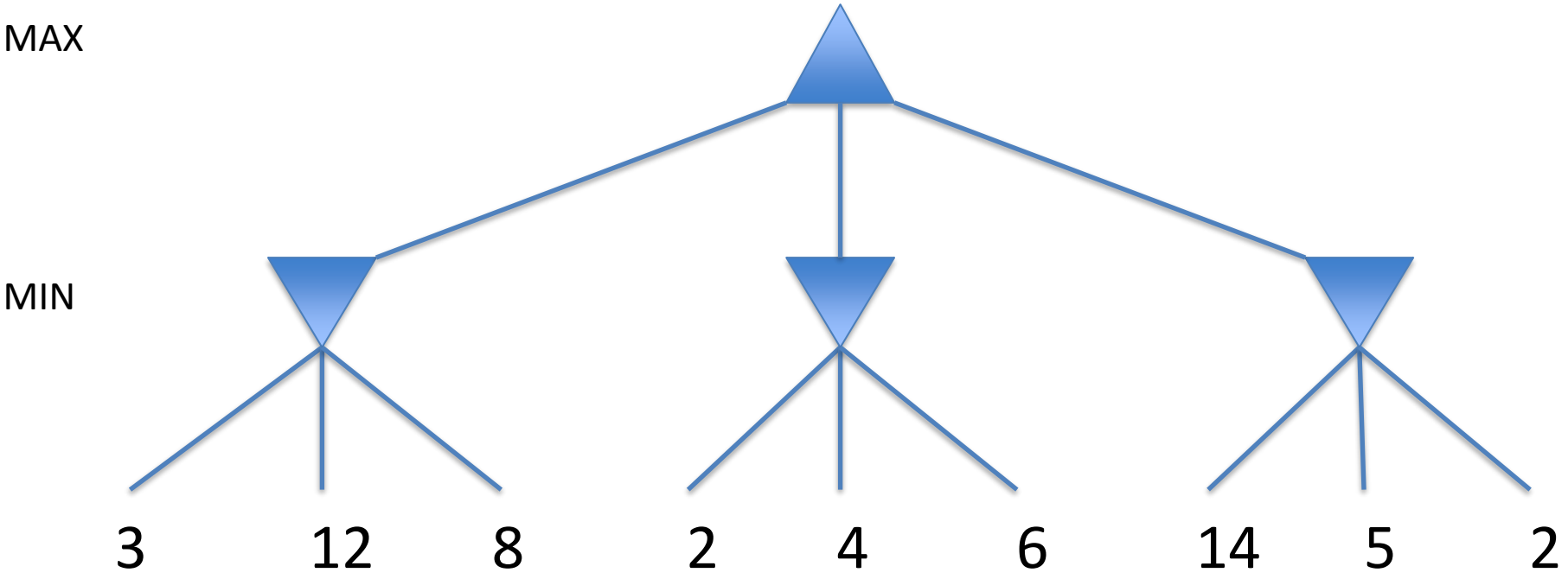
Minimax algorithm

(assuming it is MAX's turn)

minimax(s) =

utility(s, MAX)	if IS-TERMINAL(s)
$\max_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if TO-MOVE(s)=MAX
$\min_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if TO-MOVE(s)=MIN

result(s, a) means the new state generated by taking action *a* in state *s*.



minimax(s) =

utility(s, MAX)	if IS-TERMINAL(s)
$\max_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if TO-MOVE(s)=MAX
$\min_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if TO-MOVE(s)=MIN

Properties of minimax

- Complete?
  - Yes (assuming tree is finite)
- Optimal?
  - Yes (assuming opponent is also optimal)
- Time complexity: $O(b^m)$
- Space complexity: O(bm) (like DFS)
- But for chess, b ≈ 35, m ≈ 100, so this time is completely infeasible!

Real-World Minimax

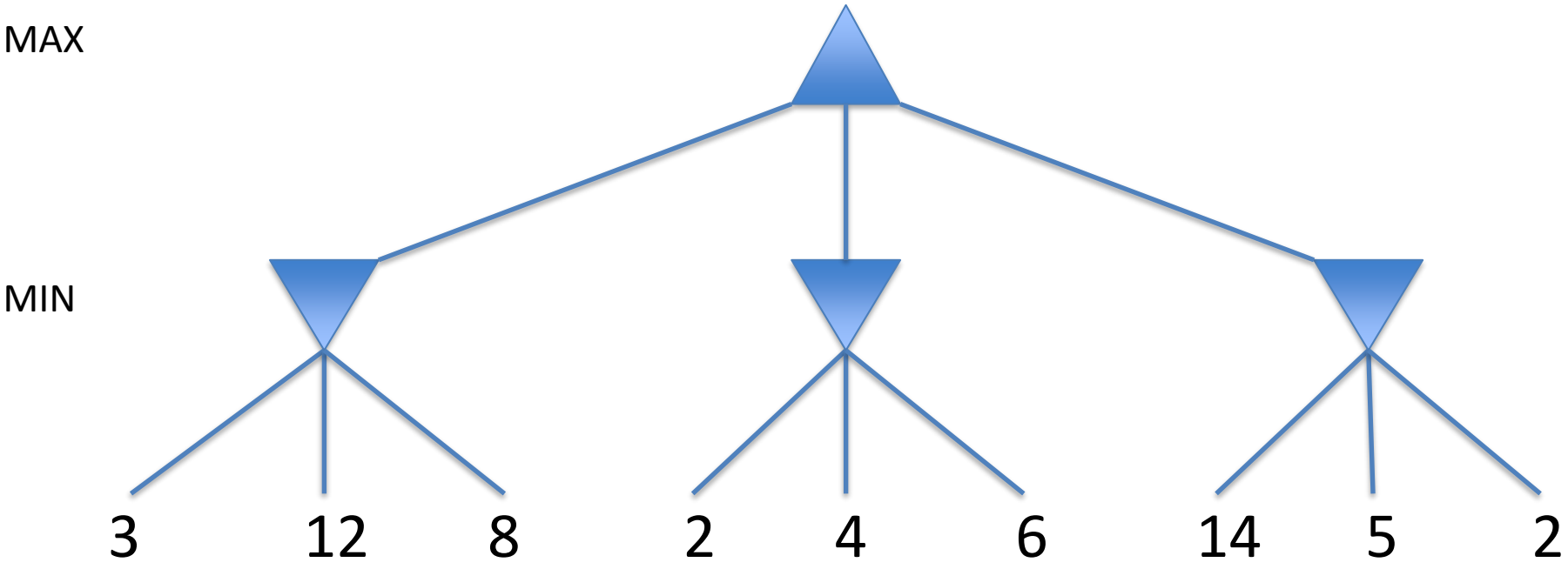
- The minimax algorithm given here only stores the utility values; "real-world" minimax should store utility values *and* the move that gives you the value.
- This is usually done by keeping an auxiliary data structure called a transposition table; this table also cuts down on search time.
  - Table stores, for every state, the minimax value and corresponding best move.

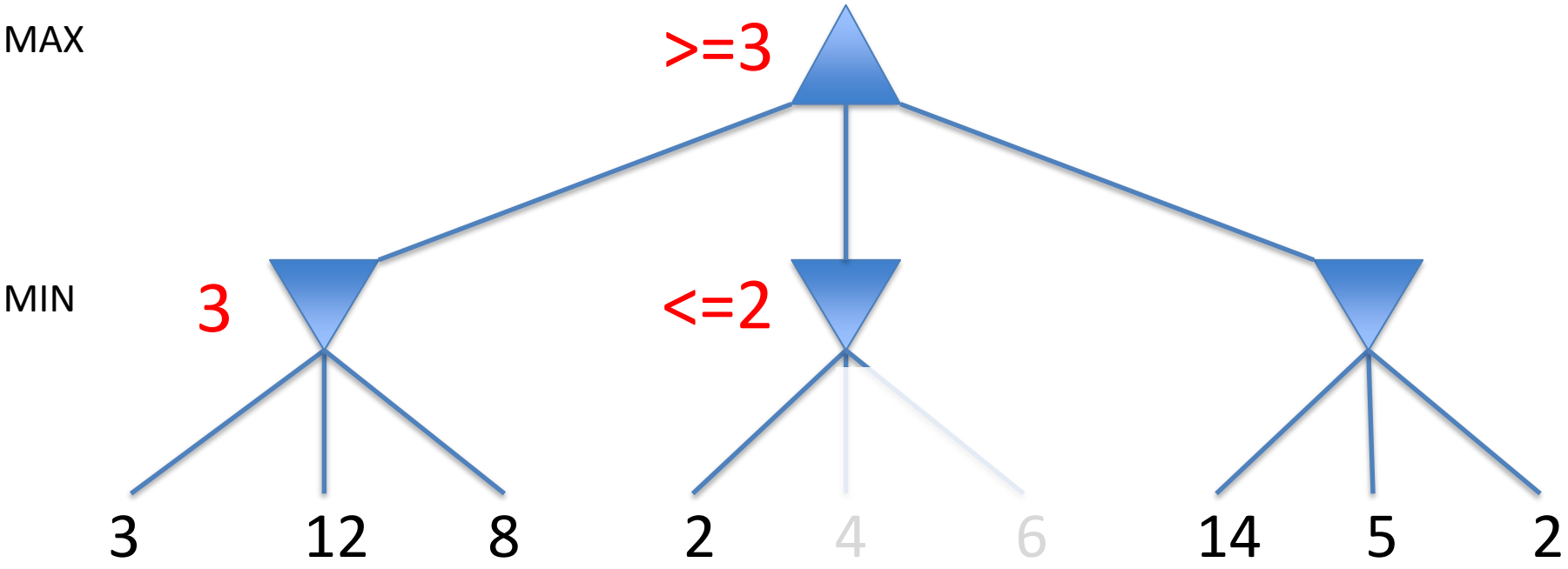
Nim

Nim

- How to represent a state?
- How to represent an action?

- Problem: minimax takes too long.
- Solution: improve algorithm to ignore parts of the tree that will definitely not be used (assuming both players play optimally).


MAX
MIN
3
12
8
2
4
6
14
5
2


MAX
>=3
MIN
3
<=2
3
12
8
2
4
6
14
5
2

- Idea: for each node, keep track of the range of possible values that minimax could produce for that node.

- If we ever find ourselves at a node that we know will never be selected during (optimal) game play, we can "prune" it (end the recursion on this part of the tree).

- Enhanced version of minimax is called ***minimax with alpha-beta pruning.***

Alpha-beta pruning

- Recall that minimax is a variant of depth-first search. During the algorithm, we will only consider nodes along the path from the root node to the current node.
- At each node in the search, we will maintain two variables:
  - alpha (α) = highest numeric value we've found so far on this path (best move for MAX)
  - beta (β) = lowest numeric value we've found so far on this path (best choice for MIN)

Alpha-beta pruning

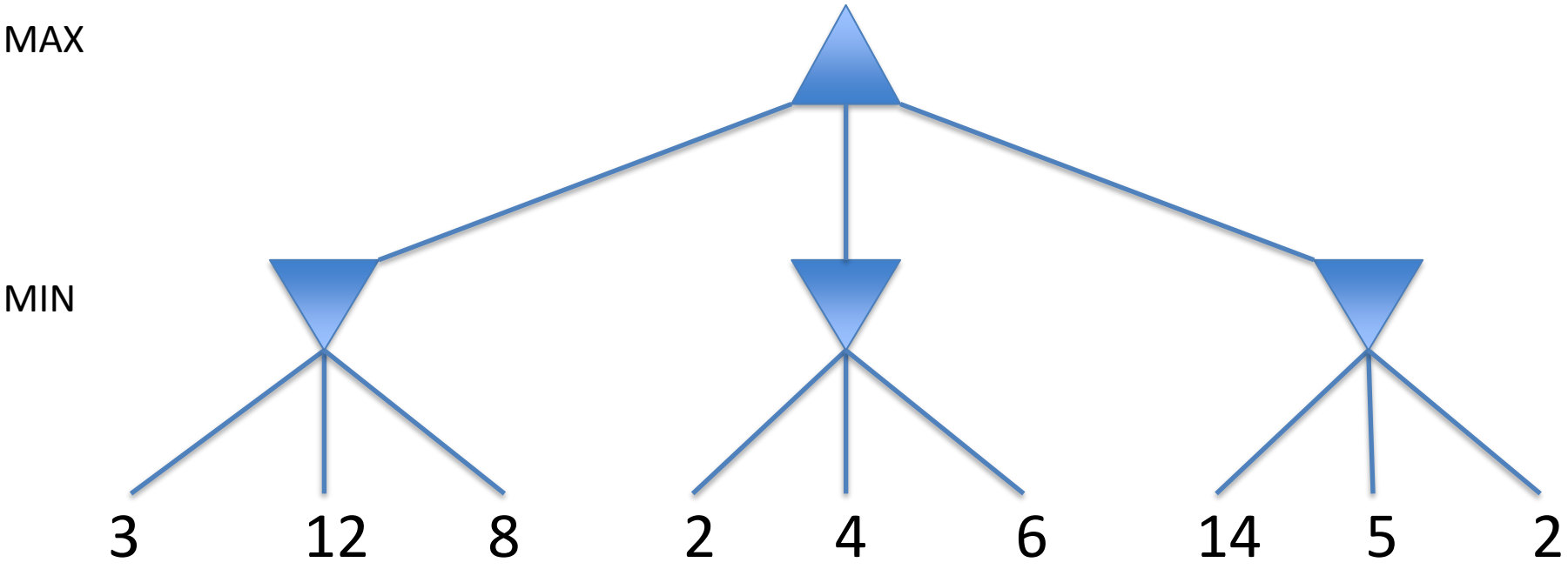
- Alpha and beta are inherited from parent nodes as we recursively descend the tree.

- If at a MAX node, we see a child node that has a value >= than beta, **short-cut**.
- If at a MIN node, we see a child node that has a value <= than alpha, **short-cut**.

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

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move
    return v, move
```


MAX
MIN
3
12
8
2
4
6
14
5
2

Alpha-beta code

- For programming, use code in the book.
- For offline use, use this idea:

alpha-beta(node):

inherit alpha & beta from parents

let v be each child value in turn:

Do either the red or the blue for each state (not both).

If at a MAX node:

if v >= beta, then short-circuit and return v
else if v > alpha, then alpha = v (and continue)

If at a MIN node:

if v <= alpha, then short-circuit and return v
else if v < beta, then beta = v (and continue)

if MAX, return alpha; if MIN, return beta (to parent)

- The results of alpha-beta depend on the order in which moves are considered among the children of a node.
- If possible, consider better moves first!

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

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move
    return v, move
```

Real-world use of alpha-beta

- (Regular) minimax is normally run as a preprocessing step to find the optimal move from every possible situation.
- Minimax with alpha-beta can be run as a preprocessing step, but might have to re-run during play if a non-optimal move is chosen.
- Save states somewhere so if we re-encounter them, we don't have to recalculate everything.

Real-world use of alpha-beta

- States get repeated in the game tree because of *transpositions*.
- When you discover a best move in minimax or alpha-beta, save it in a lookup table (probably a hash table).
  - Called a *transposition table*.

Real-world use of alpha-beta

- In the real-world, alpha-beta does not "pre-generate" the game tree.
  - The whole point of alpha-beta is to not have to generate all the nodes.
- The DFS part of minimax/alpha-beta is what generates the tree.

Summary so far

- Minimax: Find the best move for each player, assuming the other player plays perfectly.
  - Based on DFS; searches the whole game tree.
  - Usually used as a preprocessing step (too slow for real time).
- Alpha-beta: Always gives same result as minimax, but prunes sub-optimal branches.
  - Can be used to preprocess game tree, but sub-optimal moves will necessitate rerunning.
  - Can be used in real time, but often still too slow.

Improving on alpha-beta

- Alpha-beta still must search down to terminal nodes sometimes.
  - (and minimax has to search to terminal nodes all the time!)
- Improvement idea: can we get away with only looking a few moves ahead?

Heuristic minimax algorithm

REGULAR MINIMAX

minimax(s) =

utility(s, MAX)	if is-terminal(s)
$\max_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if to-move(s)=MAX
$\min_{a \text{ in actions(s)}}$ minimax(result(s, *a*))	if to-move(s)=MIN

HEURISTIC MINIMAX

h-minimax(s, d) =

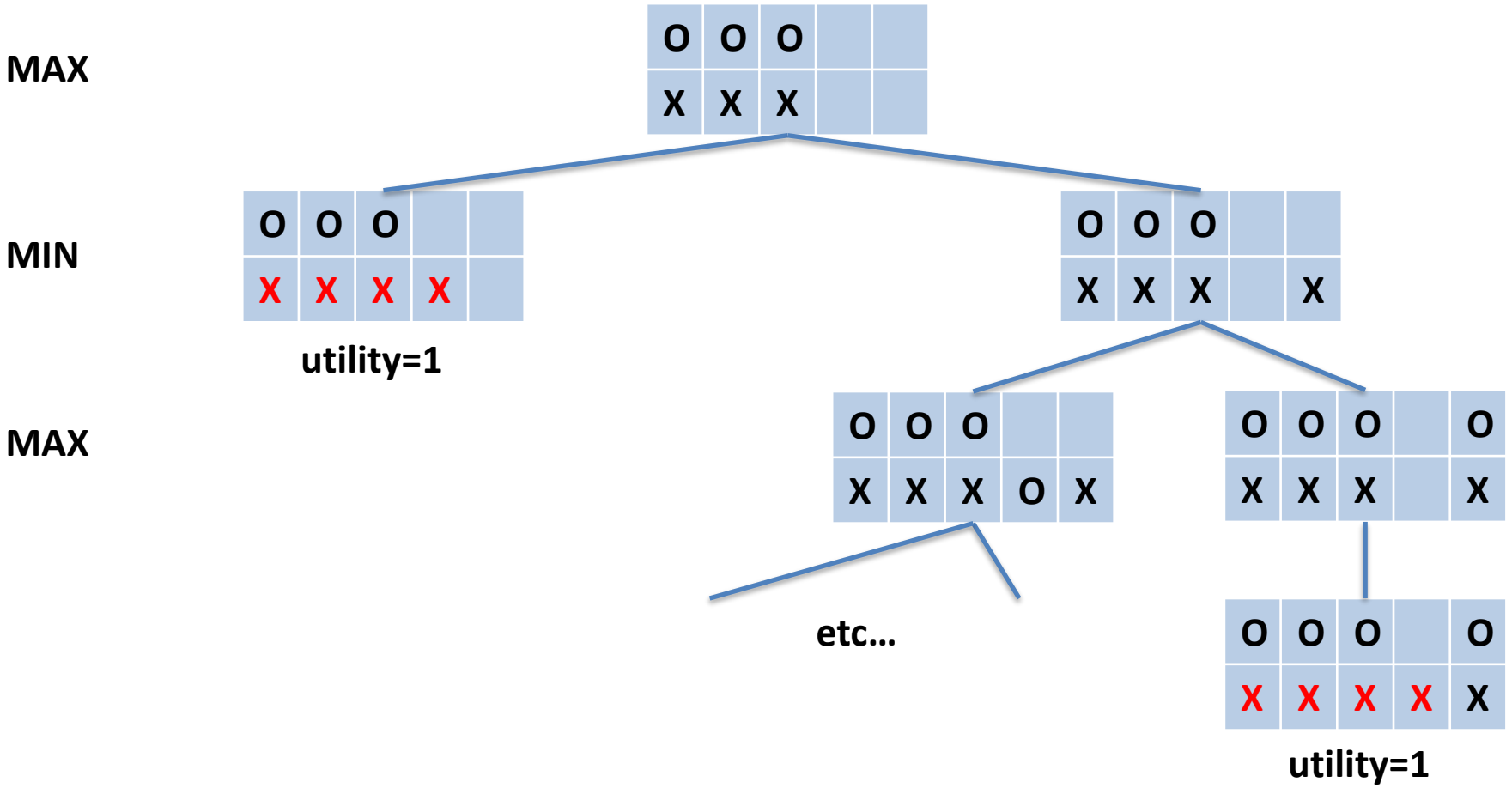
eval(s, MAX)	if is-cutoff(s, d)
$\max_{a \text{ in actions(s)}}$ h-minimax(result(s, *a*), d+1)	if to-move(s)=MAX
$\min_{a \text{ in actions(s)}}$ h-minimax(result(s, *a*), d+1)	if to-move(s)=MIN

result(s, a) means the new state generated
by taking action *a* in state *s*.
is-cutoff(s, d) is a boolean test that determines whether
we should stop the search and evaluate our position.

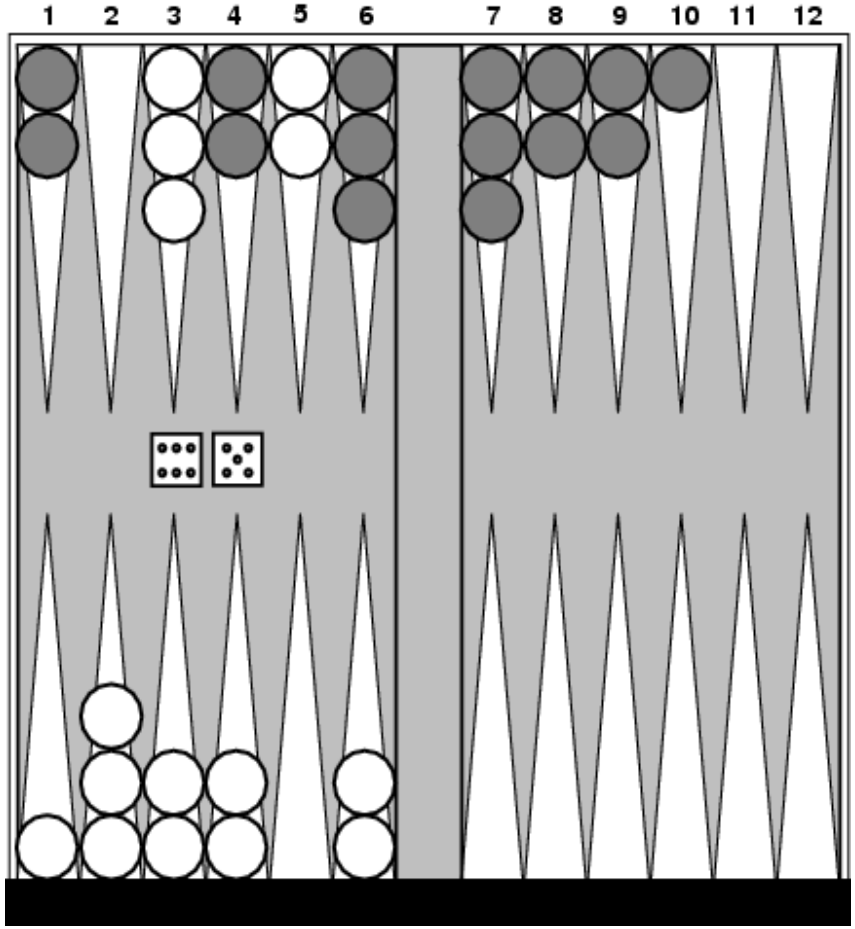
How to create a good evaluation function?

- Trying to judge the probability of winning from a given state.
- Typically use features: simple characteristics of the game that correlate well with the probability of winning.

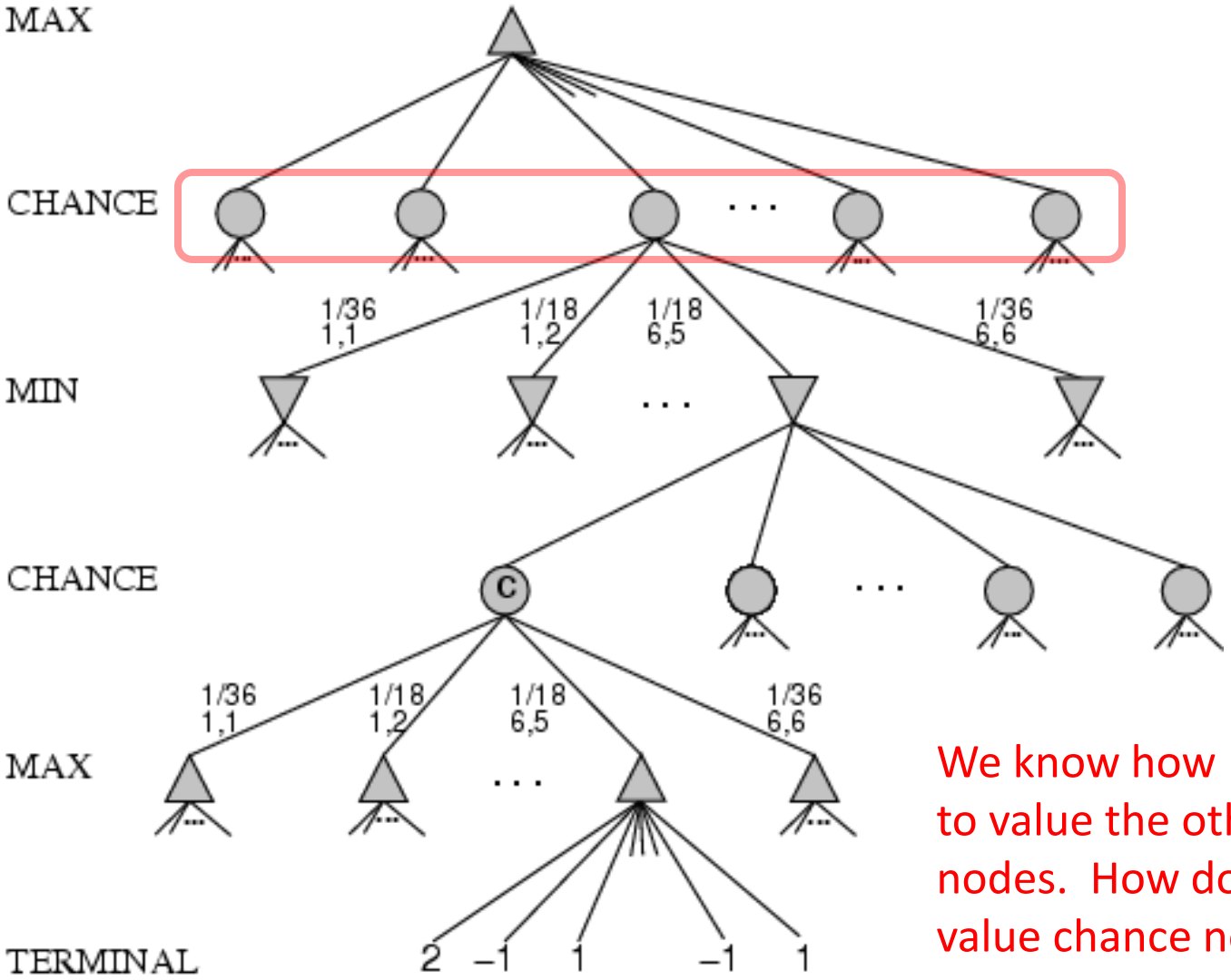
One last point



What if a game has a “chance element”?



What if a game has a "chance element"?



We know how to value the other nodes. How do we value chance nodes?

Expected value

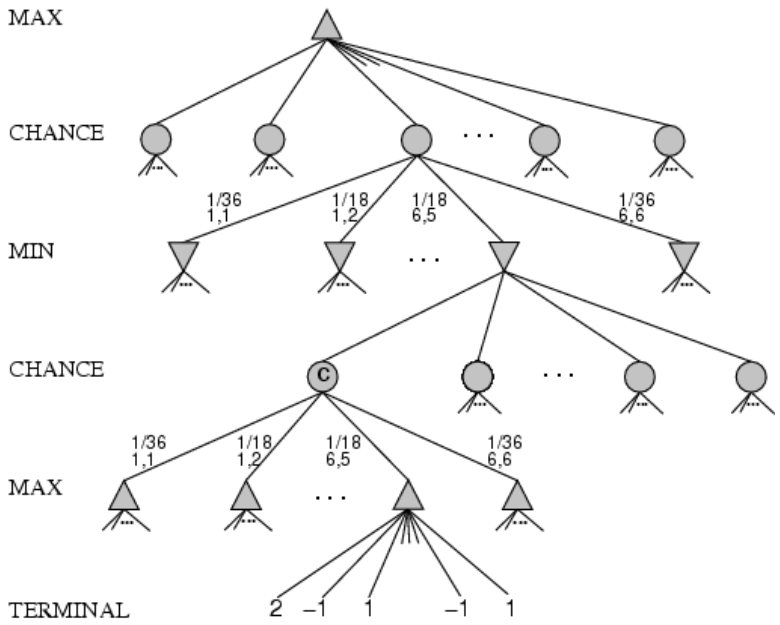
- The sum of the probability of each possible outcome multiplied by its value:

$$E(X) = \sum_i p_i x_i$$

- x_i is a possible value of (random variable) X.
- p_i is the probability of x_i happening.

Expected minimax value

- Now *three* different cases to evaluate, rather than just two.
  - MAX
  - MIN
  - CHANCE



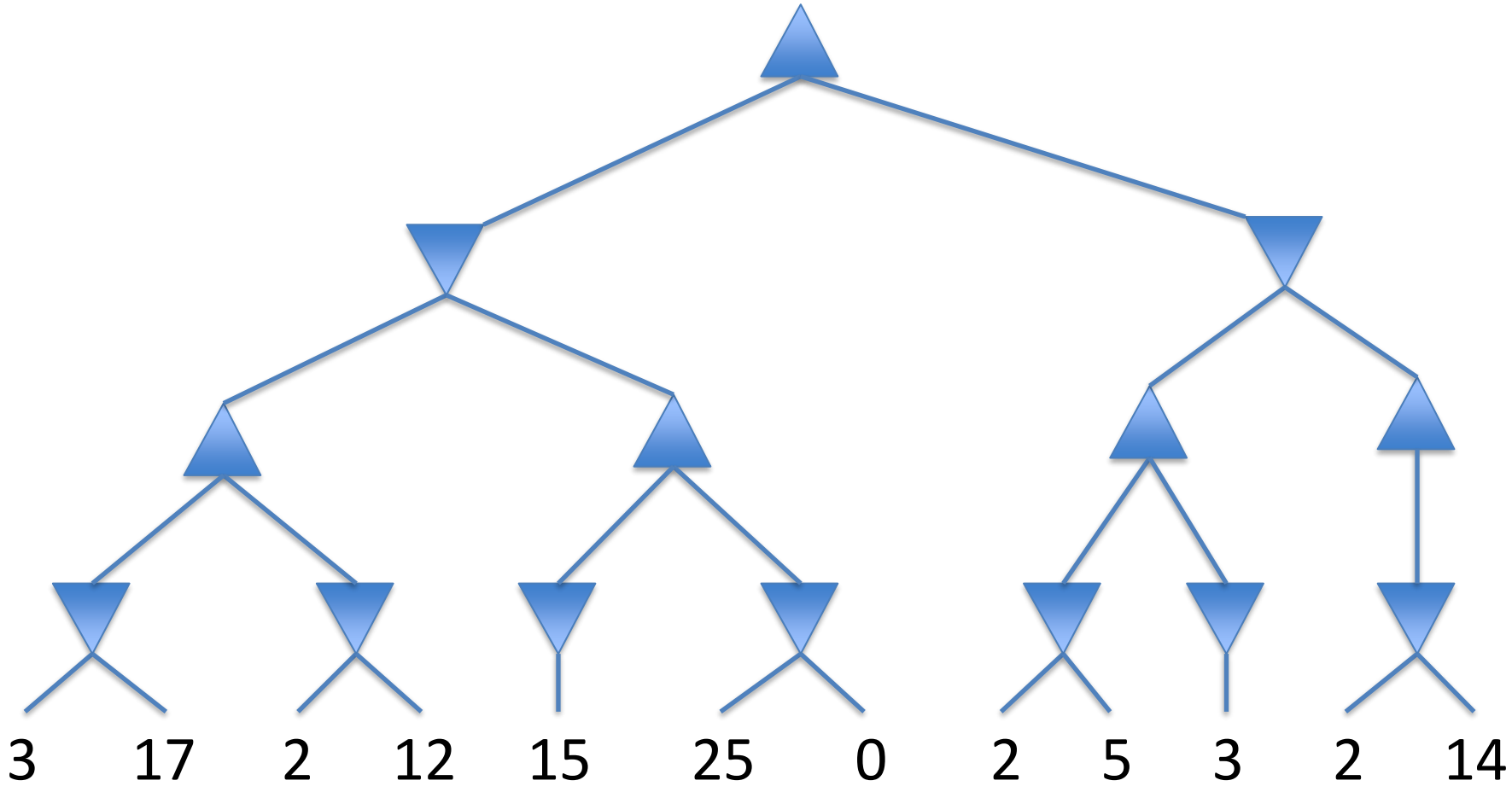
EXPECTED-MINIMAX-VALUE(n) =

UTILITY(n), If terminal node

$\max_{s \in successors(n)}$ MINIMAX-VALUE(s), If MAX node

$\min_{s \in successors(n)}$ MINIMAX-VALUE(s), If MIN node

$\sum_{s \in successors(n)} P(s)$ • EXPECTEDMINIMAX(s), If CHANCE node


3
17
2
12
15
25
0
2
5
3
2
14