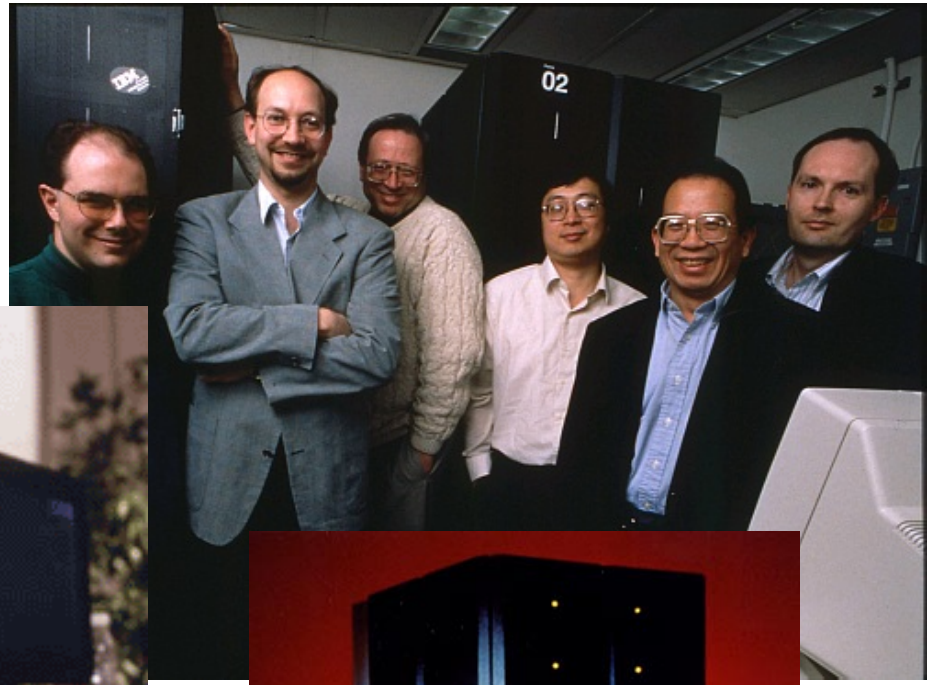
Adversarial Search

CMPSCI 383
September 29, 2011

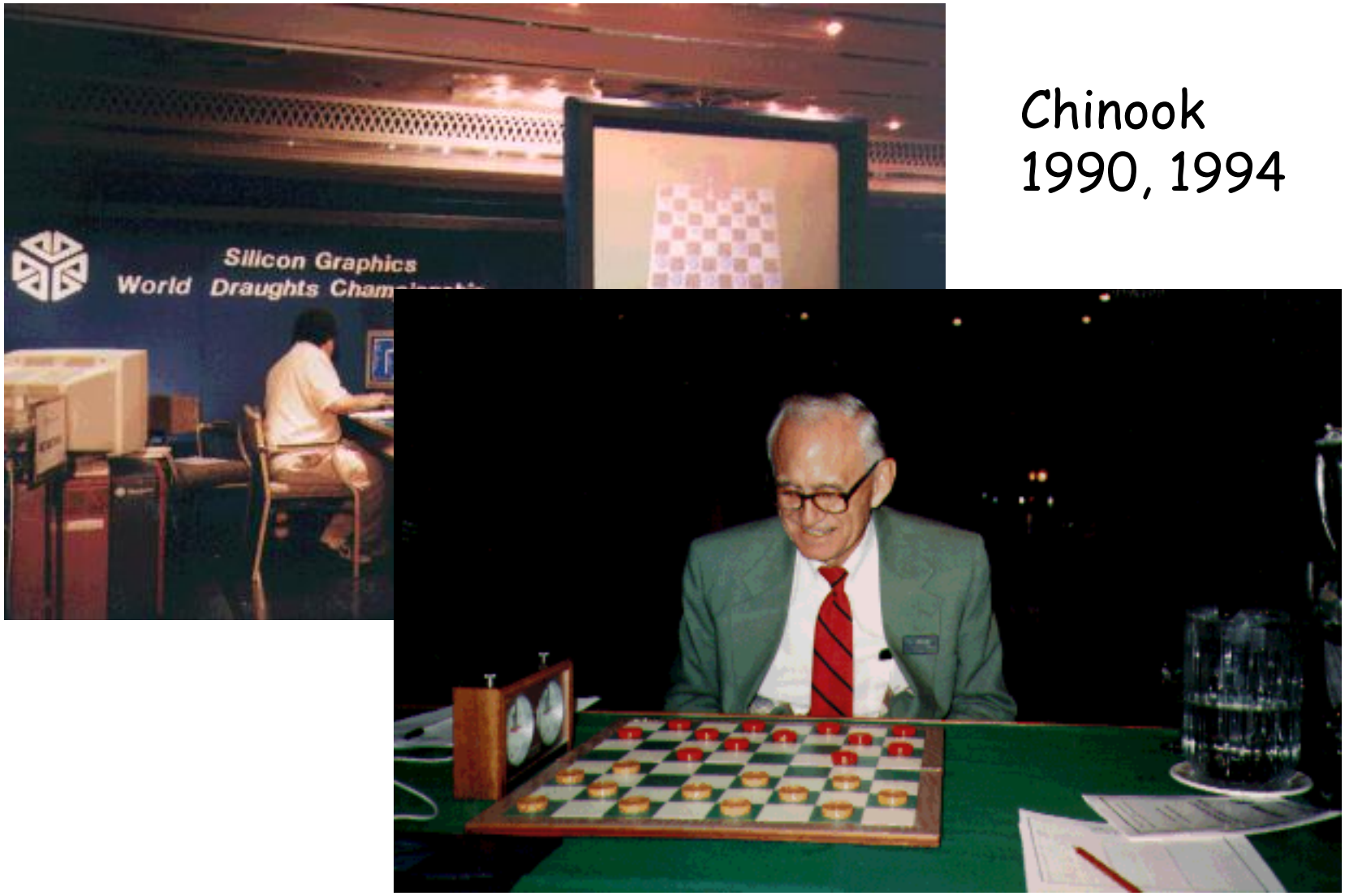
Why are games interesting to AI?

- Simple to represent and reason about
- Must consider the moves of an adversary
- Time constraints
- Russell & Norvig say:
  "Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible.
- Metareasoning: reasoning about reasoning

1997



Searched up to 30 billion positions per move; sometimes reaching a depth of 40 plies.



Chinook
1990, 1994

Dr. Marion Tinsley

Checkers Is Solved

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

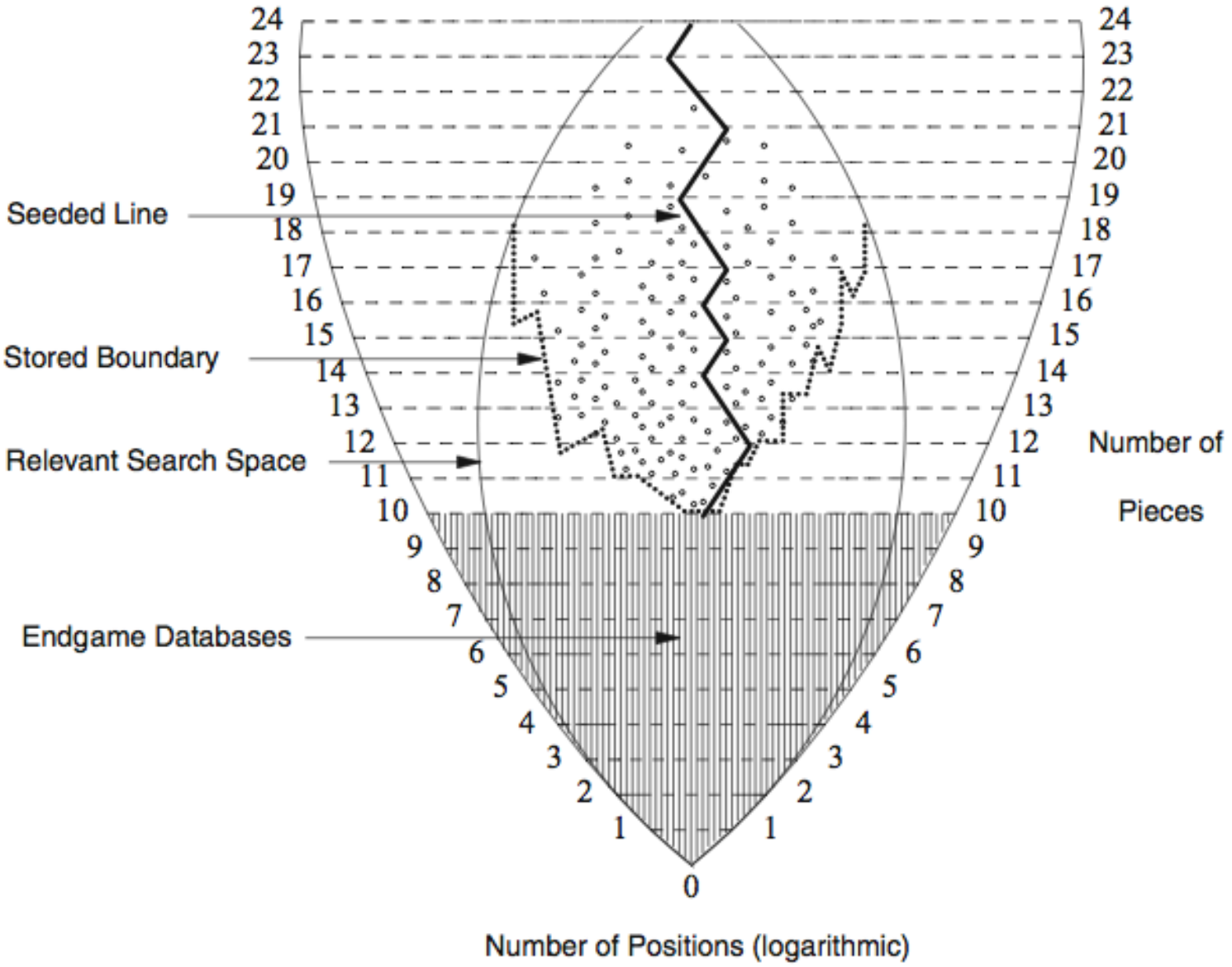
The game of checkers has roughly 500 billion billion possible positions (5×10^{20}). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving

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


Seeded Line
Stored Boundary
Relevant Search Space
Endgame Databases
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Number of Pieces
Number of Positions (logarithmic)

Pieces	Number of positions
1	120
2	6,972
3	261,224
4	7,092,774
5	148,688,232
6	2,503,611,964
7	34,779,531,480
8	406,309,208,481
9	4,048,627,642,976
10	34,778,882,769,216
Total 1–10	39,271,258,813,439
11	259,669,578,902,016
12	1,695,618,078,654,976
13	9,726,900,031,328,256
14	49,134,911,067,979,776
15	218,511,510,918,189,056
16	852,888,183,557,922,816
17	2,905,162,728,973,680,640
18	8,568,043,414,939,516,928
19	21,661,954,506,100,113,408
20	46,352,957,062,510,379,008
21	82,459,728,874,435,248,128
22	118,435,747,136,817,856,512
23	129,406,908,049,181,900,800
24	90,072,726,844,888,186,880
Total 1–24	500,995,484,682,338,672,639

Today's lecture

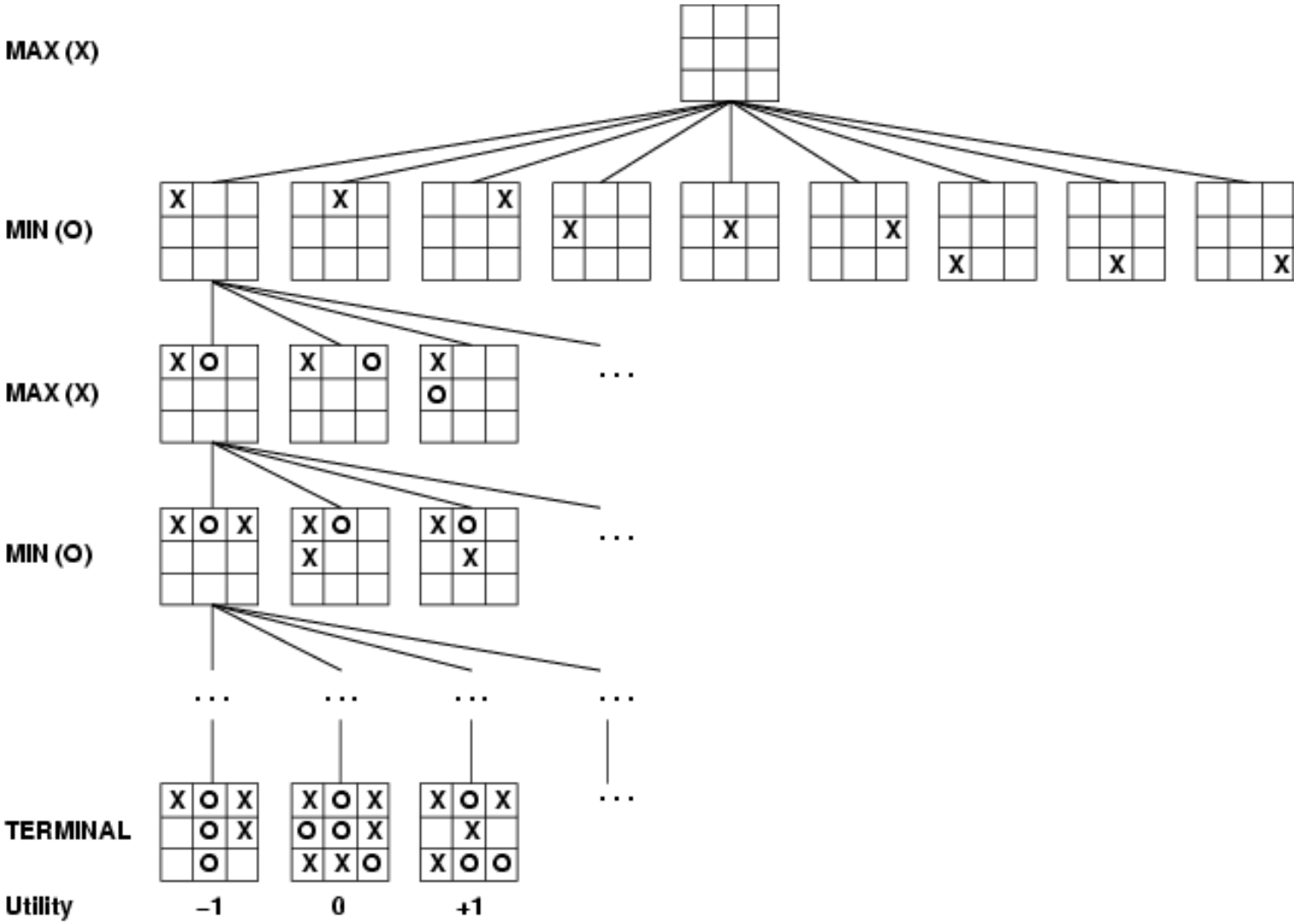
- Introduce search in adversarial environments
- Key concepts
  - Game tree
  - Min and Max players
  - Minimax value
- Methods for searching realistic game trees
  - Alpha-beta pruning
  - Approximate evaluation functions
  - Games with chance elements

CSP terminology

- This data structure is defined by the initial game state and the legal moves for each player
- This is the value of a node for a given player, assuming that both players play optimally to the end of the game.
- This is a level of the search tree defined by a move by a single player

- *What is a* **Game tree**
- *What is the* **minimax value**
- *What is a* **Ply**

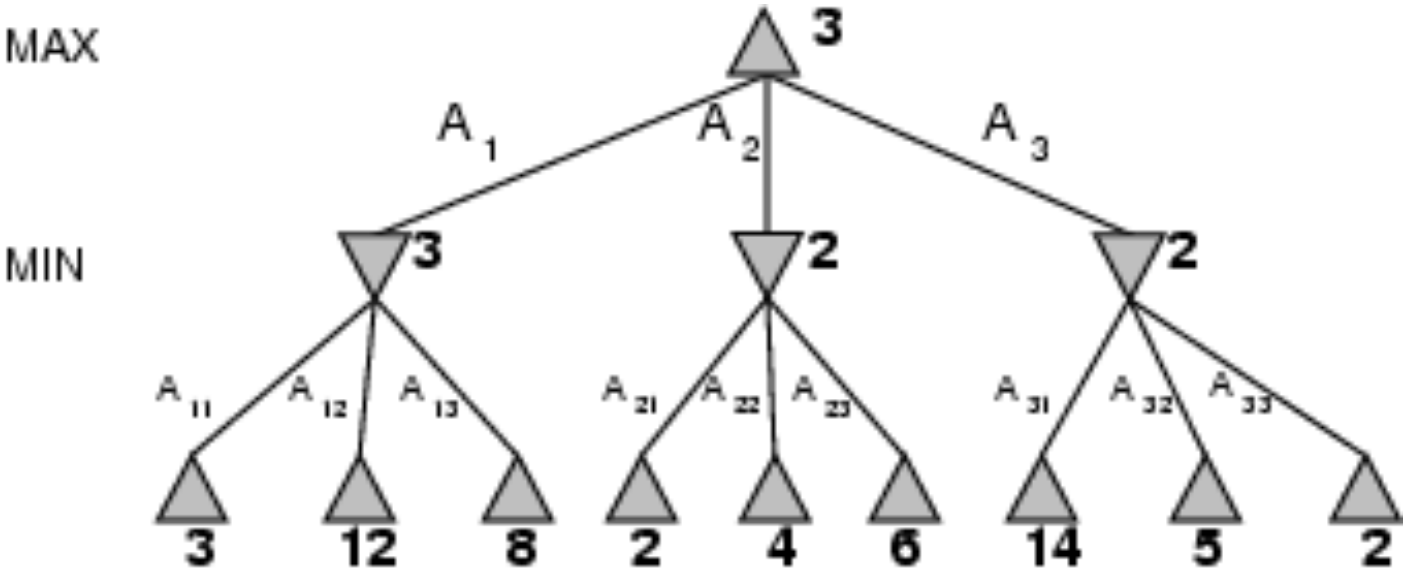
Game tree (2=player, deterministic, turns)



Minimax algorithm

- Perfect play for deterministic games

- Idea — select moves with highest minimax value. That is, select the best achievable payoff against best play by your opponent



Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

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

Properties of minimax

- Complete?
  Yes (if tree is finite)

- Optimal?
  Yes (against optimal opponent)

- Time complexity — $O(b^m)$

- Space complexity — $O(bm)$ (depth-first)

…but for chess, b ≈ 35, m ≈100

Exact solution is completely infeasible

Adversarial search terminology

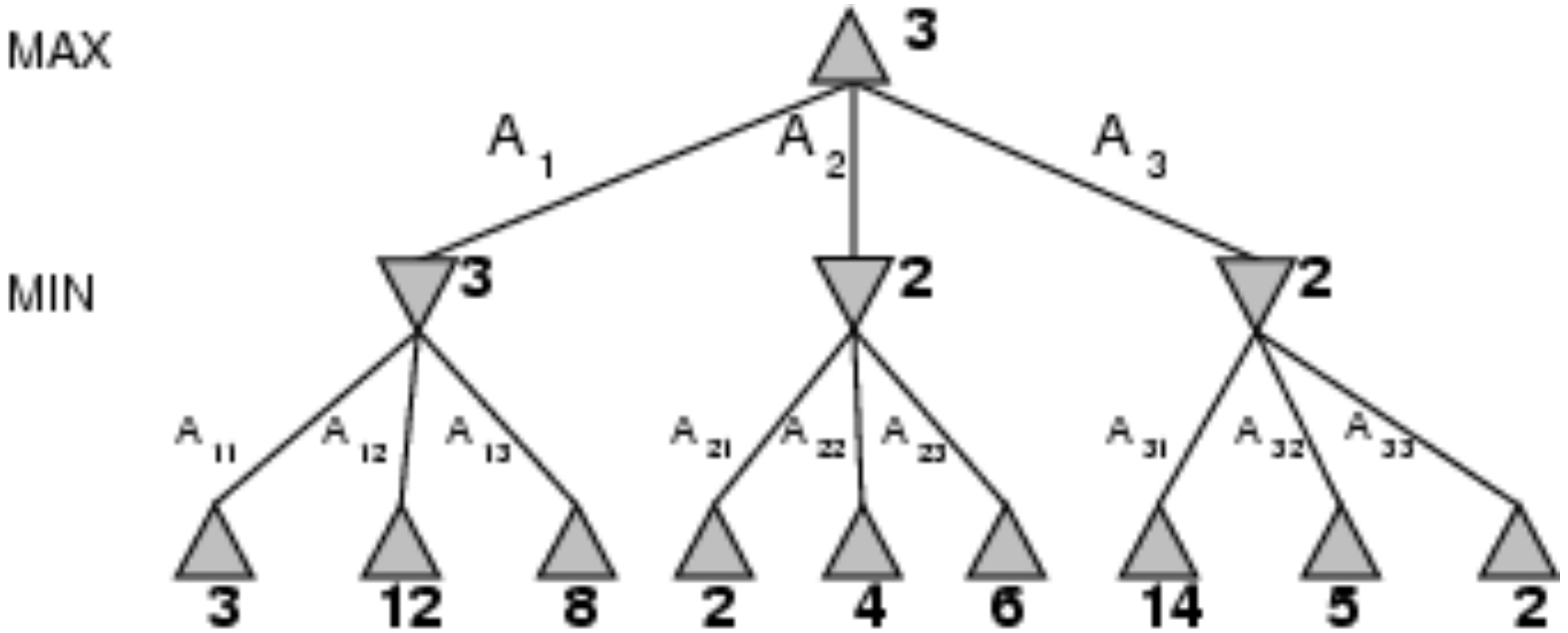
- This method can eliminate large portions of the game tree from consideration, thus speeding up search.
- This expression returns an *estimate* of the expected utility of the game for a given position

- *What is* **Alpha-beta pruning**

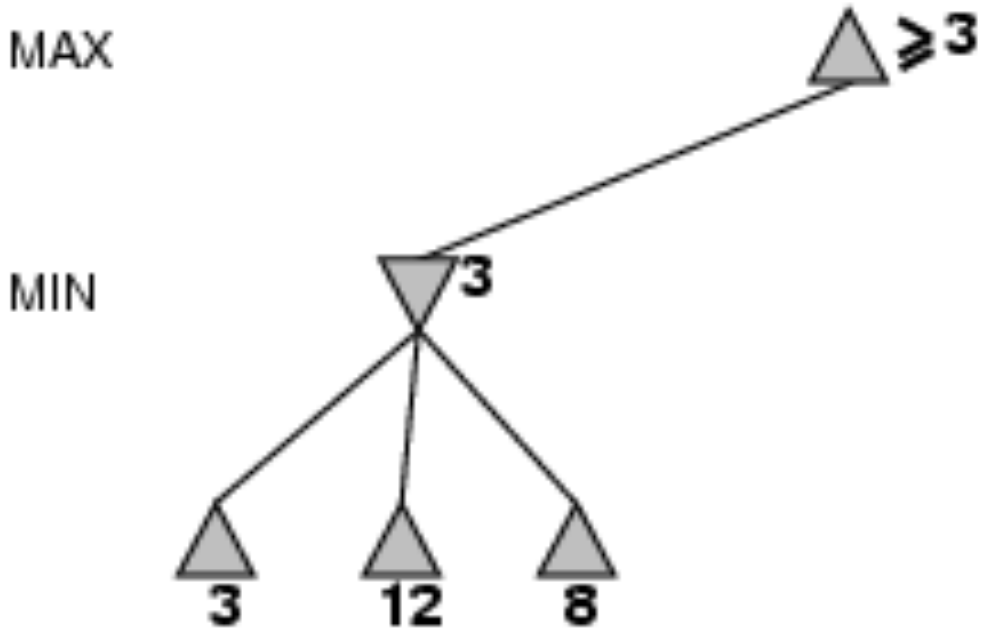
- *What is an* **Evaluation function**

How does pruning work?

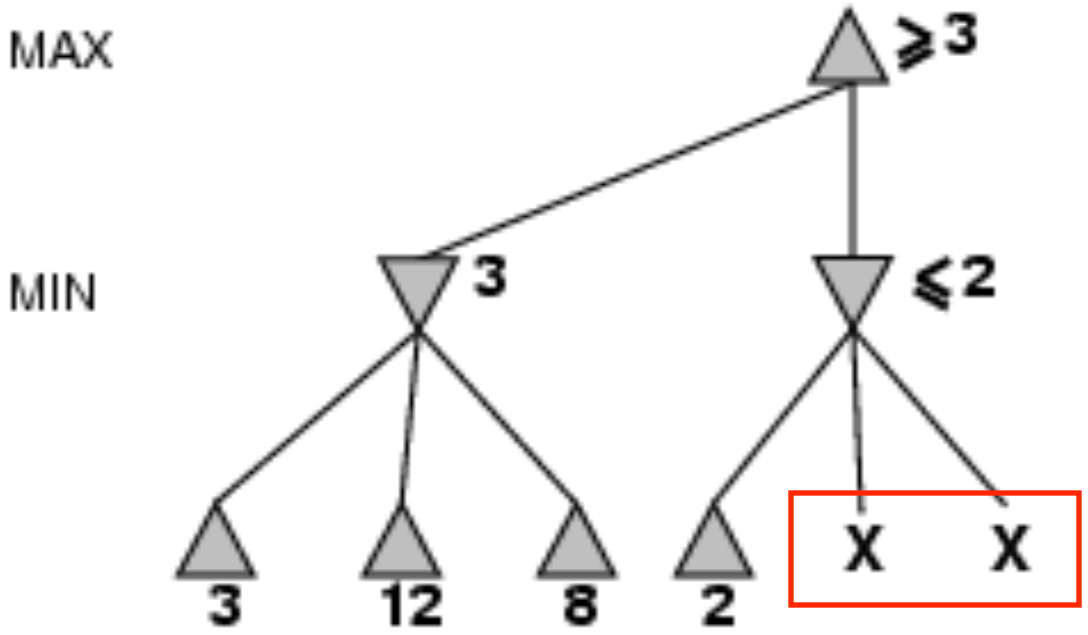
Using DFS, can we prune this tree?



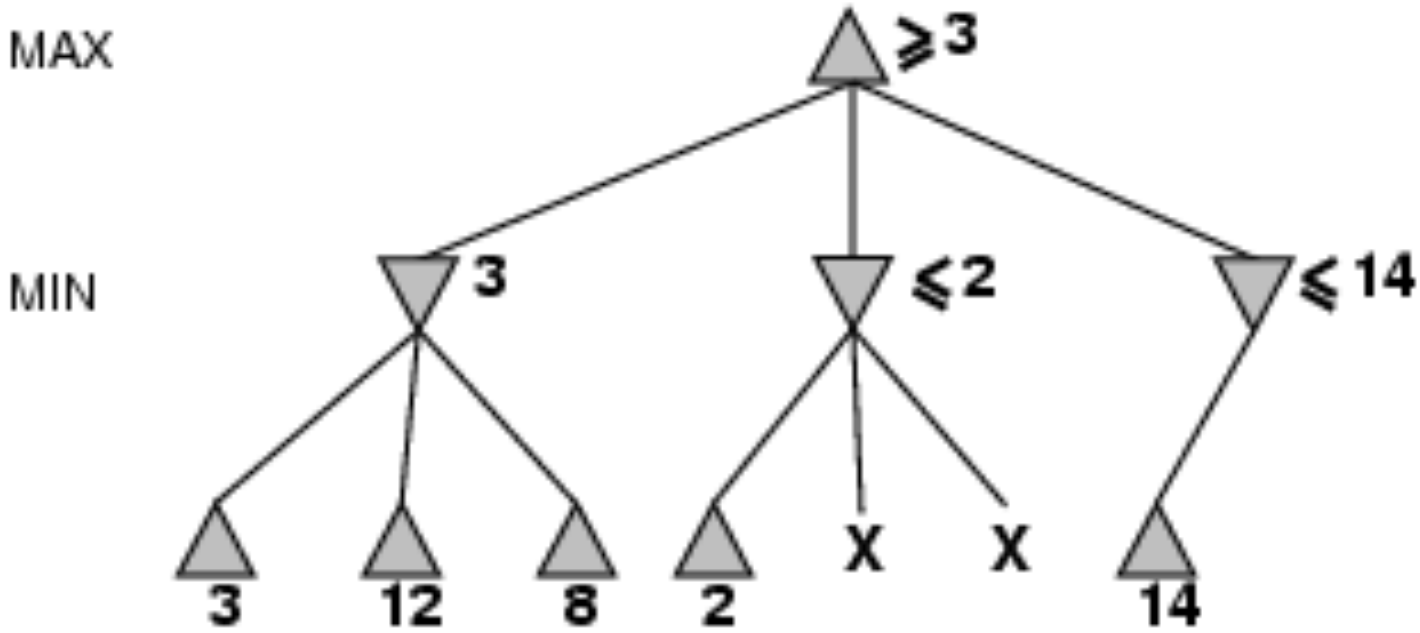
Example: α-β pruning



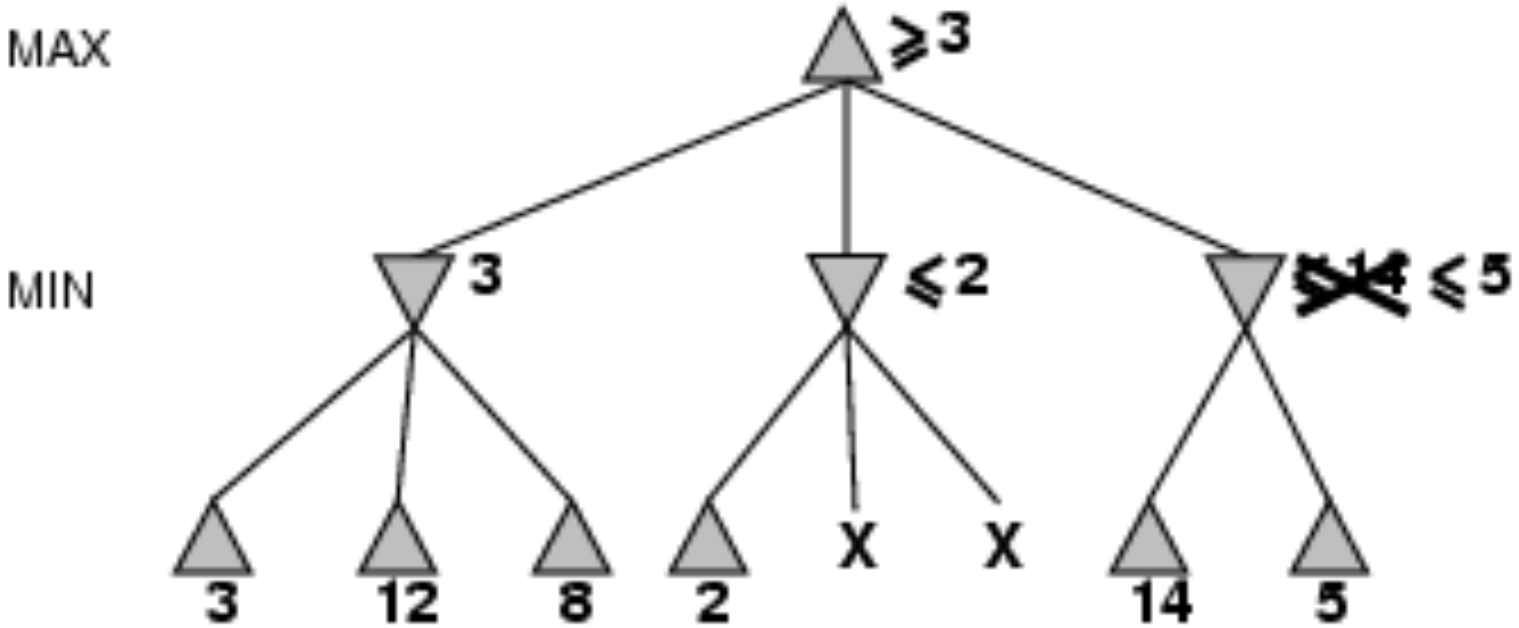
Example: α-β pruning



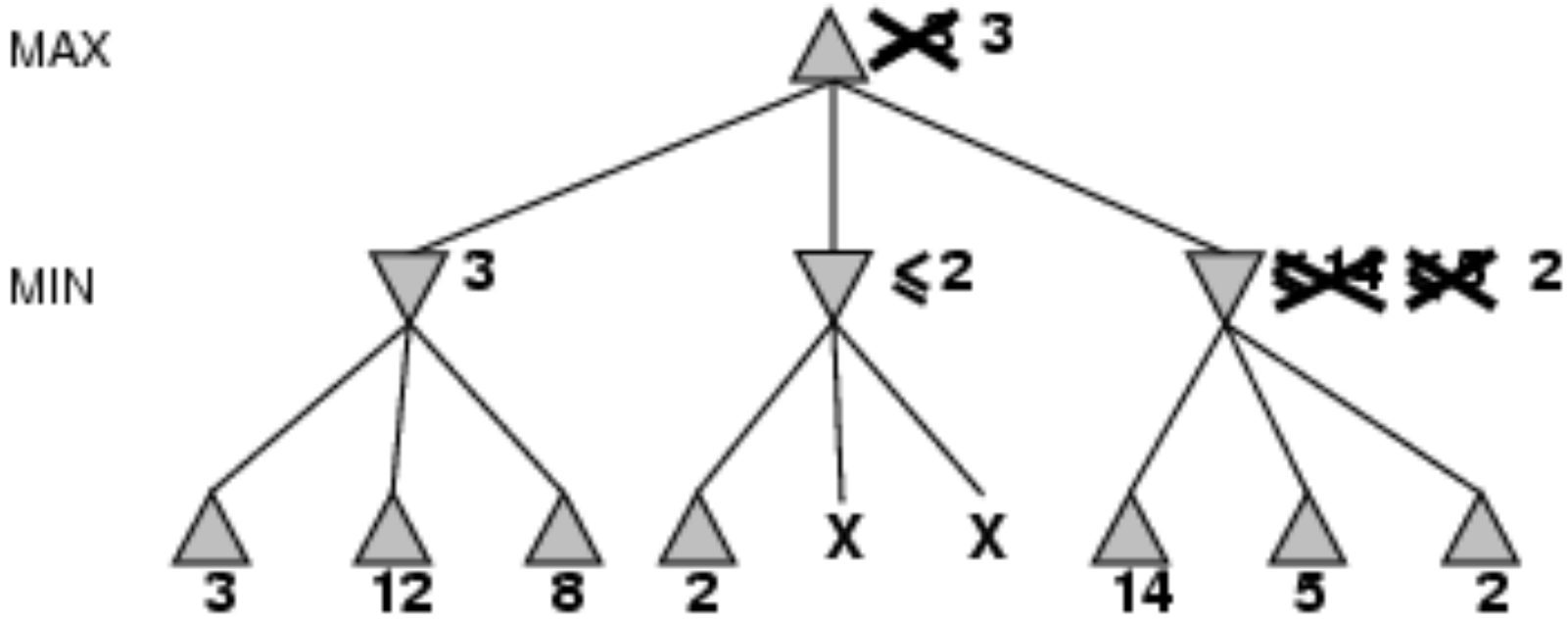
Example: α-β pruning



Example: α-β pruning



Example: α-β pruning

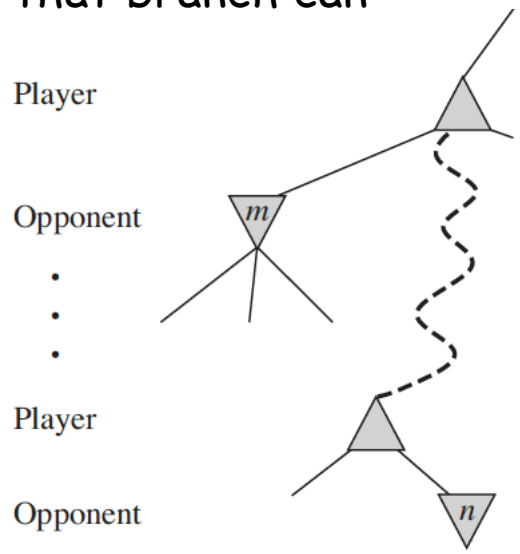


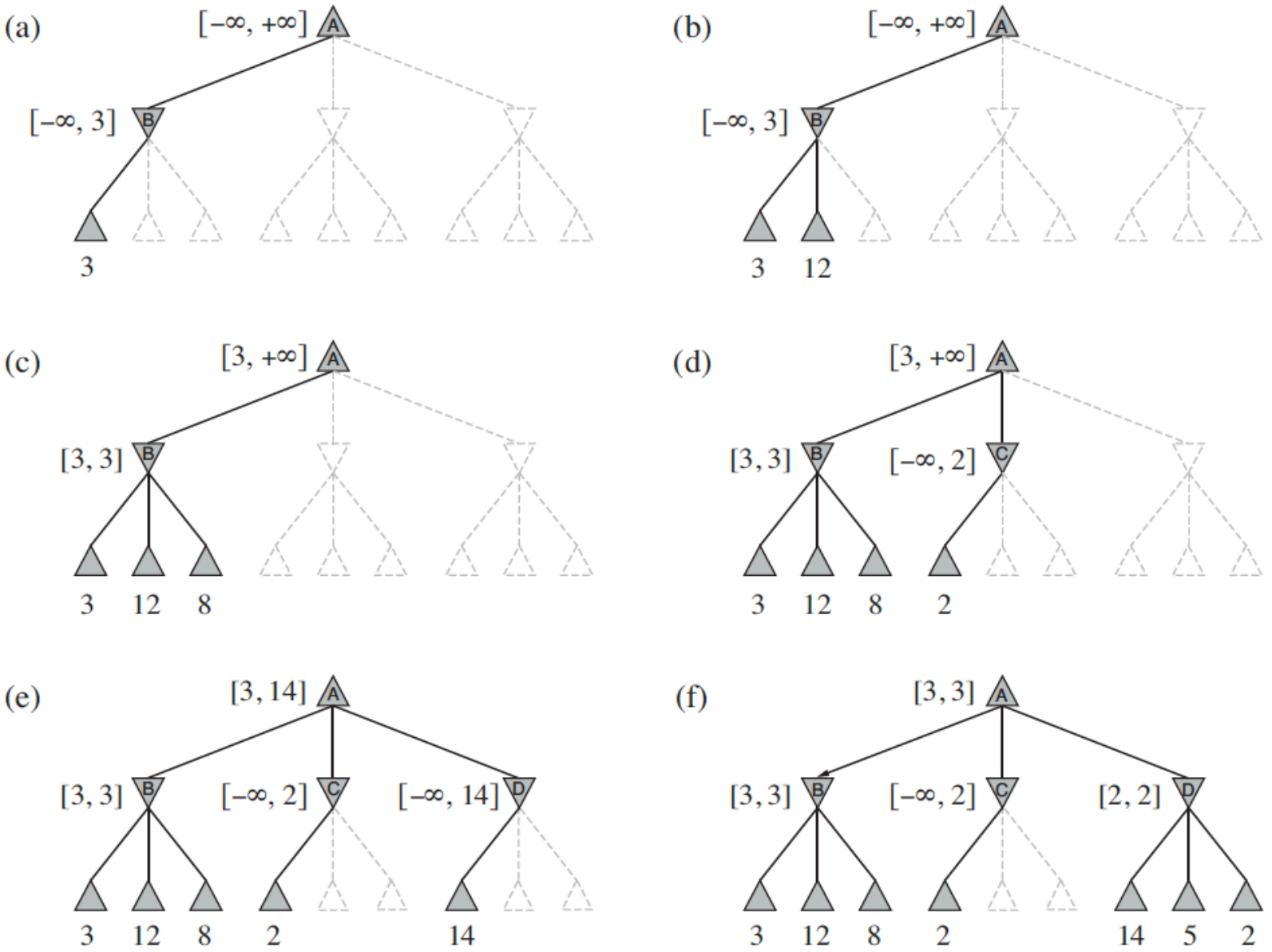
Why is it called α-β?

- α is the value of the best (highest-value) choice found so far at any choice point along the path for *MAX*
  - If *v* is worse than α, *MAX* will avoid it, so that branch can be pruned.

- β is the value of the best (lowest-value) choice found so far at any choice point along the path for *MIN*
  - If *v* is worse than β , *MIN* will avoid it, so that branch can be pruned.

If *m* is better than *n*, we will never get to n




(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

The α - β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

-----------------------------------------------------------

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
```

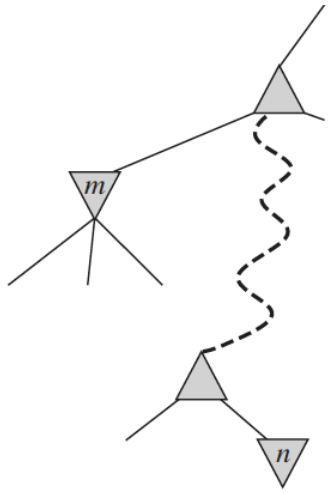
The α - β algorithm

```
function MIN-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 ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Comments on α-β pruning

- Pruning produces results that are exactly equivalent to complete (unpruned) search
- Entire subtrees can be pruned.
- Ordering
  - Node *ordering* can improve effectiveness
  - Perfect ordering gives time complexity $O(b^{m/2})$
  - Branching factor goes from *b* to *sqrt(b)*
  - Thus, alpha-beta pruning can search twice as far as ordinary minimax in equal time
- Repeated states are possible
  - Can avoid recomputing their value by using a hash table of previous states, called a *transposition table*



…but search is still intractable. What now?

- Stop the search before you reach terminal states (using a cutoff-test)
- Evaluate nodes using an *evaluation function* with properties such that it is…
  - Able to order terminal states in the same way as the true utility function
  - Efficient to calculate
  - Strongly correlated with the actual probability of winning
- Sounds difficult. How can we create such an evaluation function?

Cutting off

MinimaxCutoff is identical to *MinimaxValue* except

1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

4-ply lookahead is a hopeless chess player!

- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov

Evaluation functions

- Typically calculate *features* — simple characteristics of the game state that are correlated with the probability of winning
- The evaluation function combine feature values to produce a score
- Typically, evaluation functions are a weighted linear function

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

Example features

What would be some useful features for chess?

- Relative number of
  - Bishops
  - Knights
  - Rooks
  - Pawns
  - Total number of pieces
- Has queen?
- Castled?
- In check?
- Distance of furthest pawn from start
- Relative freedom (relative total number of possible moves)
- etc.

Evaluation functions

- Evaluation functions in the form of linear equations make an assumption about features. What is it?

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

Feature independence

- Is this assumption accurate?

Example features

What would be some useful features for chess?

- Relative number of
  - Bishops
  - Knights
  - Rooks
  - Pawns
  - Total number of pieces
- Has queen?
- Castled?
- In check?
- Distance of furthest pawn from start
- Relative freedom (relative total number of possible moves)
- etc.

Evaluation functions

- Evaluation functions in the form of linear equations make an assumption about features. What is it?

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

Feature independence

- Is this assumption accurate? No
- Does violating this assumption matter?

Often, No. As long as the ordering of function values is accurate (not necessarily the raw values), the results will be the same

How could you *learn* a good evaluation function?

Arthur Samuel 1901-1990

~1970 at Stanford AI Laboratory

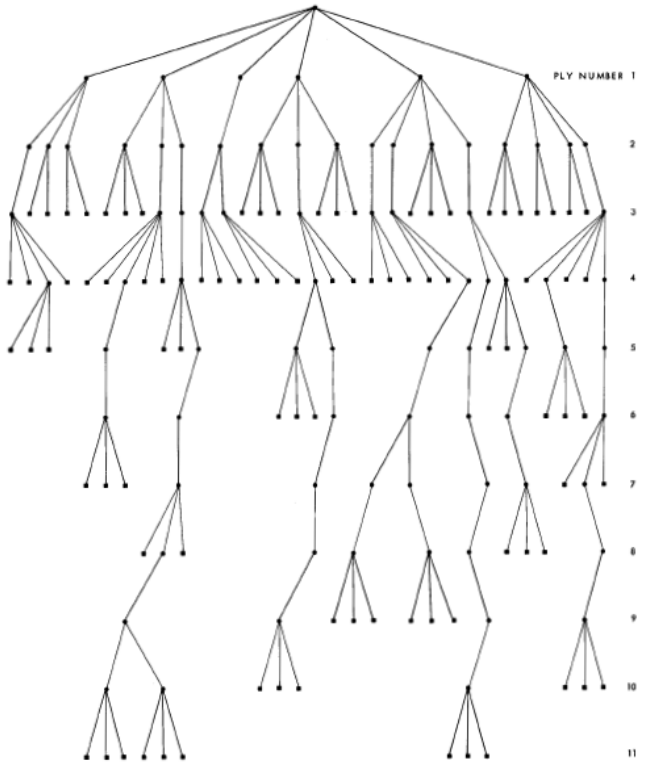
IBM Poughkeepsie Laboratory

Worked on machine learning
for the game of checkers
from 1949 through the 1960s

Some Studies in Machine Learning Using the Game of Checkers

Abstract: Two machine-learning procedures have been investigated in some detail using the game of checkers. Enough work has been done to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program. Furthermore, it can learn to do this in a remarkably short period of time (8 or 10 hours of machine-playing time) when given only the rules of the game, a sense of direction, and a redundant and incomplete list of parameters which are thought to have something to do with the game, but whose correct signs and relative weights are unknown and unspecified. The principles of machine learning verified by these experiments are, of course, applicable to many other situations.

IBM Journal July 1959



Why Samuel chose checkers

- Checkers instead of chess so focus could be on learning
- "Checkers contains all the characteristics of an intellectual activity in which heuristic procedures and learning processes can play a major role and in which these processes can be evaluated."
  - Not deterministic
  - Can't explore every path (~10^{40} choices of moves)
  - A definite goal
  - Definite rules that are known: leave learning the rules until later
  - Need background knowledge against which learning performance can be compared
  - Familiar to lots of people so it is understandable
  - Provides a convincing demonstration for those who don't believe machines can learn; playing against humans "adds spice."
- Many complications of real life are absent

Computational Challenges?

- Large search space
- Uncertainty
- Delayed reward
- Representation
- Time constraints ("situated")

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

- Othello: human champions refuse to compete against computers, who are too good.

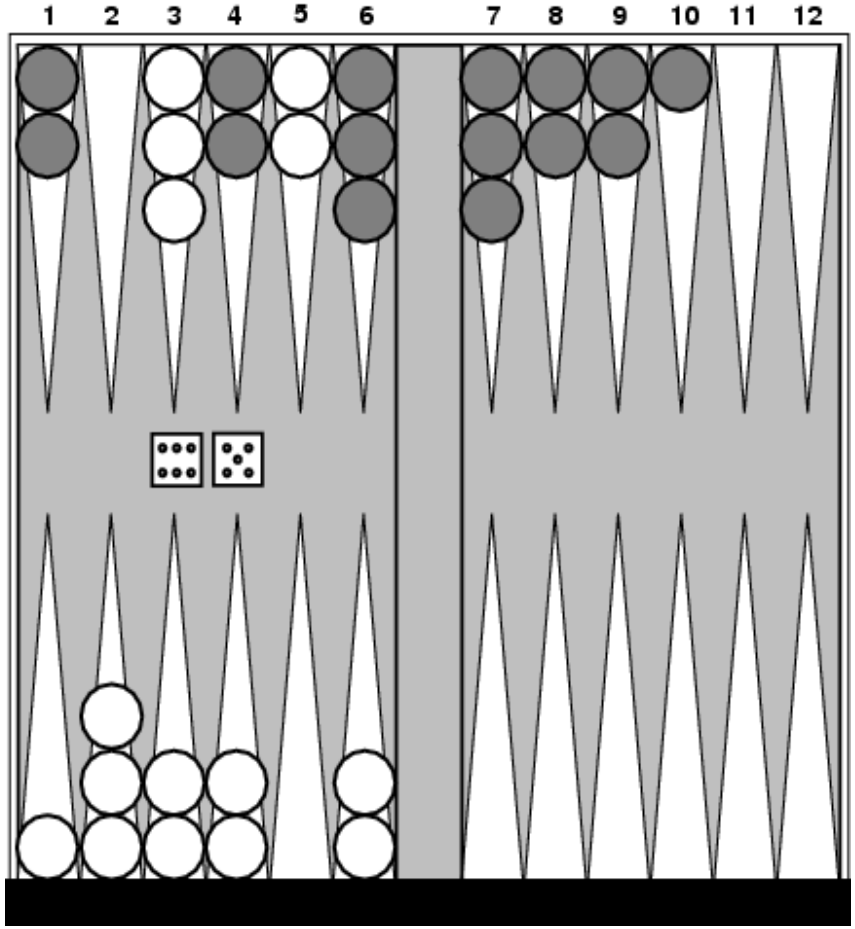
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

What are the big ideas for today?

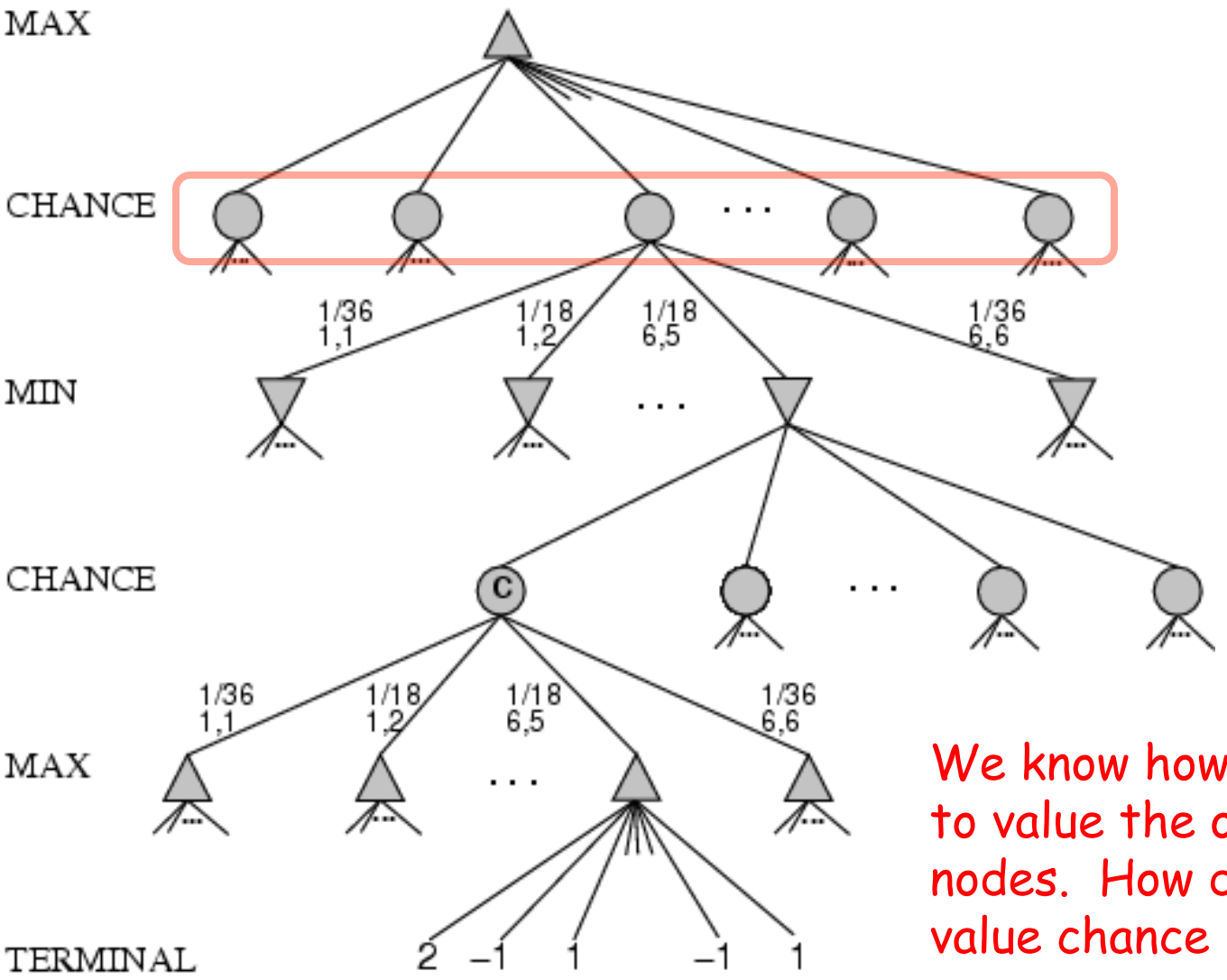
Next Class

- Stochastic and Partially Observable Games
- Secs. 5.5–5.8

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

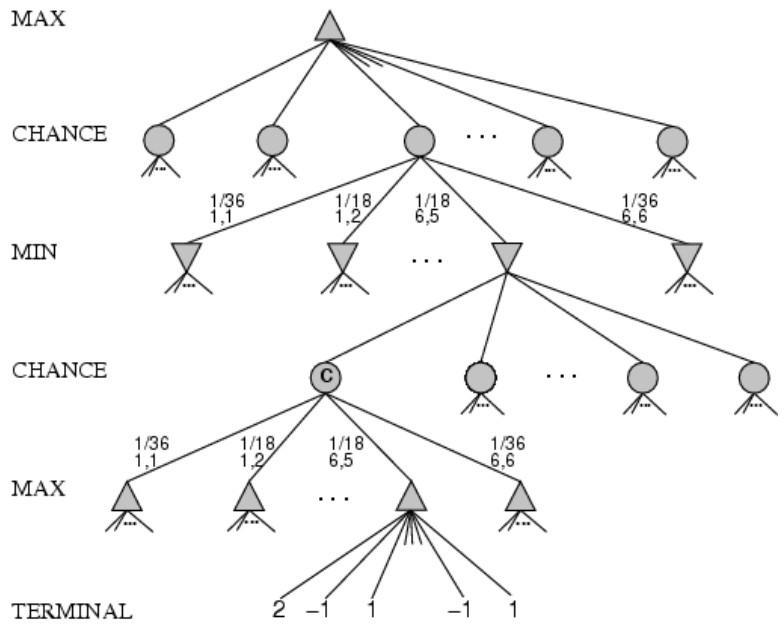
- The sum of the probability of each possible outcome multiplied by its value:

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

- Are there pathological cases where this statistic could do something strange?
  - Extreme values ("outliers")
  - Functions that are a non-linear transformation of the probability of winning

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

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