Part 4: Adversarial Search: Planning Against an Opponent

Univ.-Prof. Dr. Gerhard Widmer
Department of Computational Perception
Johannes Kepler University Linz

gerhard.widmer@jku.at
http://www.cp.jku.at/people/widmer

Overview

Game Playing in AI

Perfect Play

- minimax search
- α-β pruning

Resource limits and heuristic evaluation

Briefly: Games involving chance

AI Game Playing: State of the Art

Games as a Research Ground for AI

Games …

… have challenged the intellect of humans for thousands of years
… are abstract problems – good subject of study
… are easy to formalise (states well defined, small number of actions)
… are easy to reason about (direct outcomes of actions are precisely defined)
… but are hard to solve
(huge search spaces –
e.g., chess: typical game trees have something like 35^{100} or 10^{154} nodes …)

Algorithmic approaches to game playing have been studied even before the invention of the computer; some basic *ideas* developed very early:

- computer considers possible lines of play (Babbage, 1846)
- minimax-type algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- ideas related to finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- first chess program – on paper (Turing, 1951)
- machine learning to improve evaluation accuracy (Samuel, 1952-57)
- idea of pruning to allow deeper search (McCarthy, 1956)

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble

In this class:

- deterministic (and chance, briefly)
- perfect information
- two-player
- zero-sum ("what is good for one player is bad for the other")

games

First Step: Perfect Play in Deterministic 2-Player Games

Scenario:

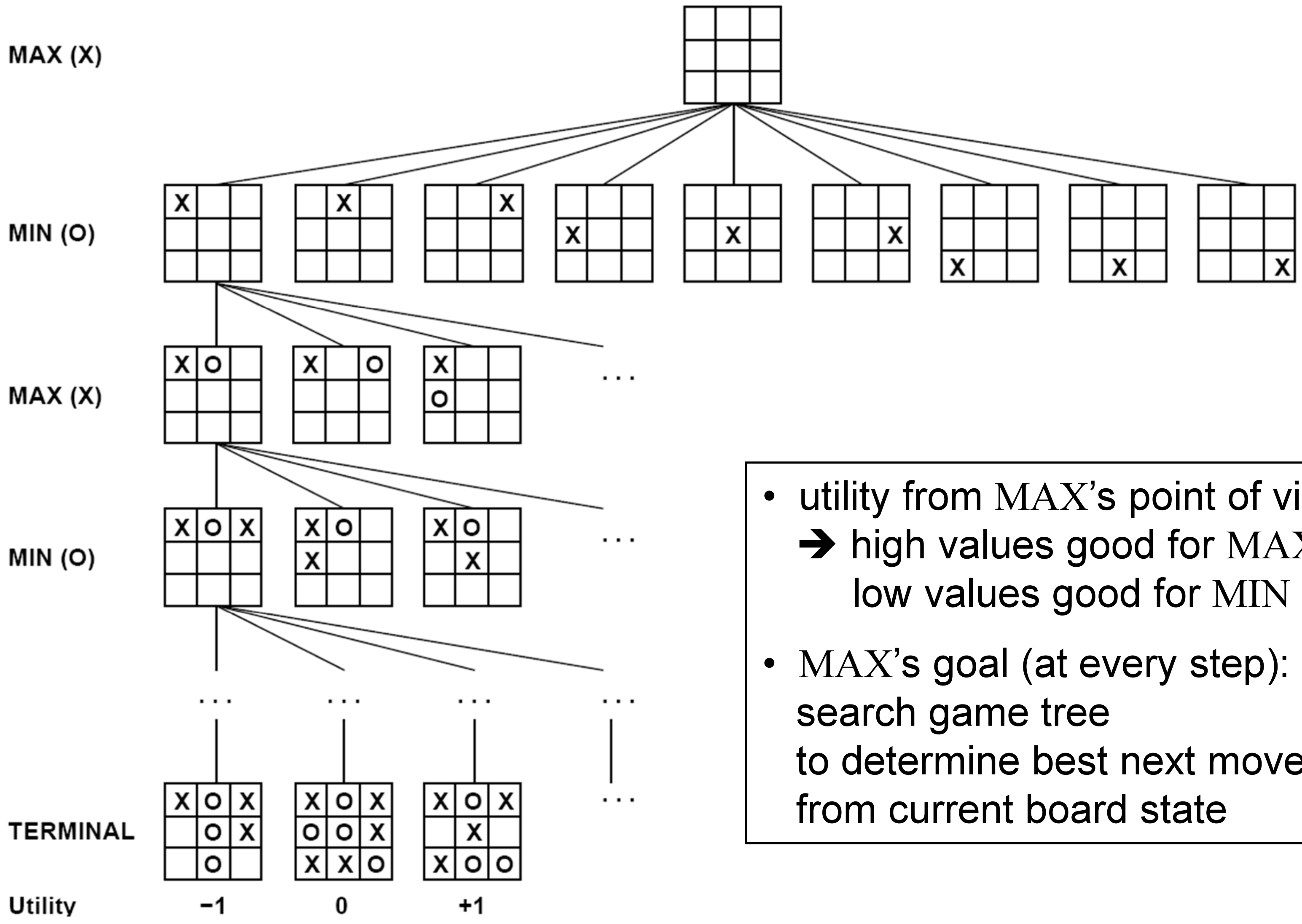
- 2 players (called MAX and MIN)
- MAX moves first
- then they take turns making moves until the game is over

Game Playing as a Search Problem:

- **Initial state:** initial board position and player to move (MAX)
- **Successor function:** returns a list of <move, newstate> pairs (legal moves and corresponding resulting states) for given state
- **Terminal (goal) test:** determines when the game is over; states where the game is over are called **terminal states**
- **Utility function** (also called objective function or payoff function): gives a numerical value to the terminal states (e.g., in chess: +1 for win, -1 for loss, 0 for draw)

➔ Initial state + successor function define a **game tree**

Game Tree for Tic-Tac-Toe



- utility from MAX's point of view
  ➔ high values good for MAX,
  low values good for MIN
- MAX's goal (at every step):
  search game tree
  to determine best next move
  from current board state

Optimal Strategies

- In a "normal" search problem, optimal solution would be one sequence of moves leading to a goal state (i.e., terminal state that is a win)
- but in 2-player game, the opponent (MIN) must be considered

➔ MAX needs a **strategy** that computes MAX's move
- in the initial state
- in all the states that might result from MIN's response to the first move
- in all the states that might result from MIN's response to all possible responses by MAX …

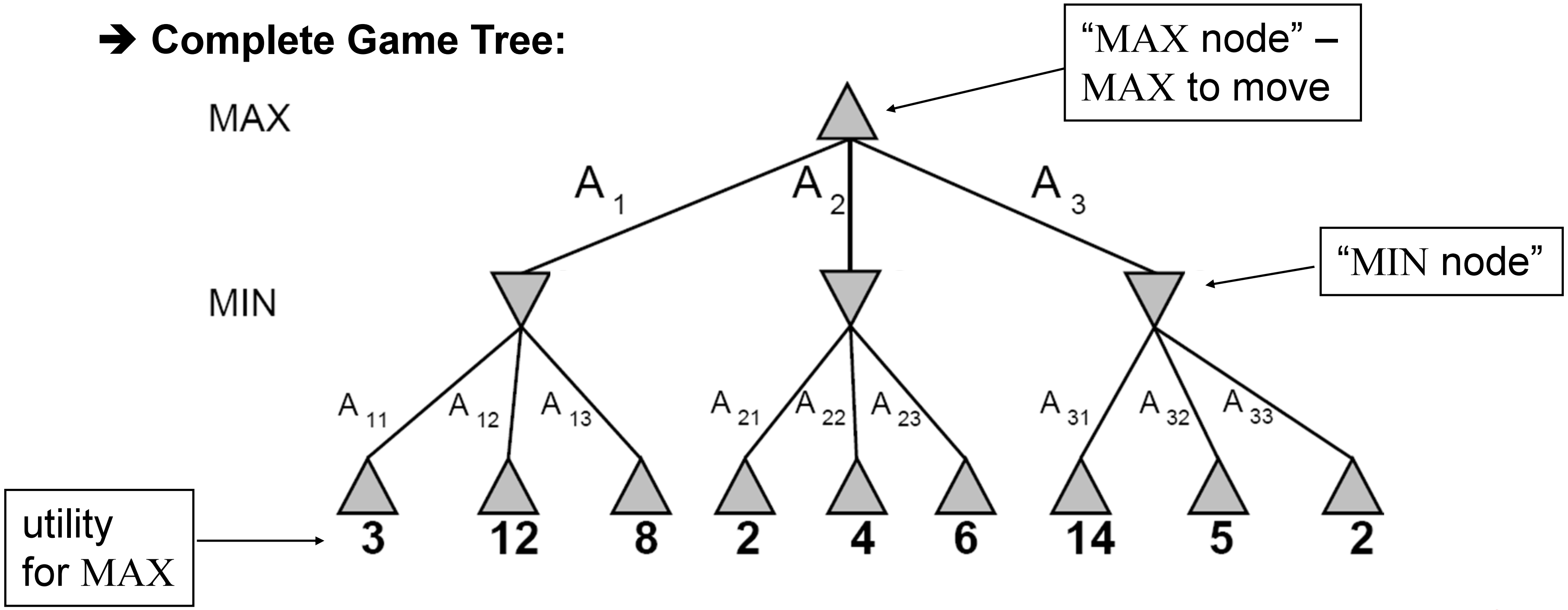
Definition:
An **optimal strategy** is a strategy that leads to the best outcome that is possible if one plays against a perfectly playing (infallible) opponent

In the following:
show how to find an optimal strategy
(even though too expensive to compute in reasonably complex games)

A Trivial "Game"

- Possible moves for MAX: A_1, A_2, A_3
- Possible moves for MIN: A_{11}, A_{12}, etc.
- Game ends after one move each by MAX and MIN
  (tree is "one move deep"; each of the half-moves is called a "ply")

➔ **Complete Game Tree:**



What is the best utility value that MAX can achieve with certainty?

➔ max(min(3,12,8),min(2,4,6),min(14,5,2)) = max(3,2,2) = 3

The Minimax Value

Given a complete game tree, an optimal strategy can be deduced from the minimax values of the tree nodes

Definition:
The **minimax value** of a node n is the utility (for MAX) of being in n (i.e., the maximum utility MAX can be guaranteed to gain from n onwards), assuming that both players play optimally from n to the end of the game:

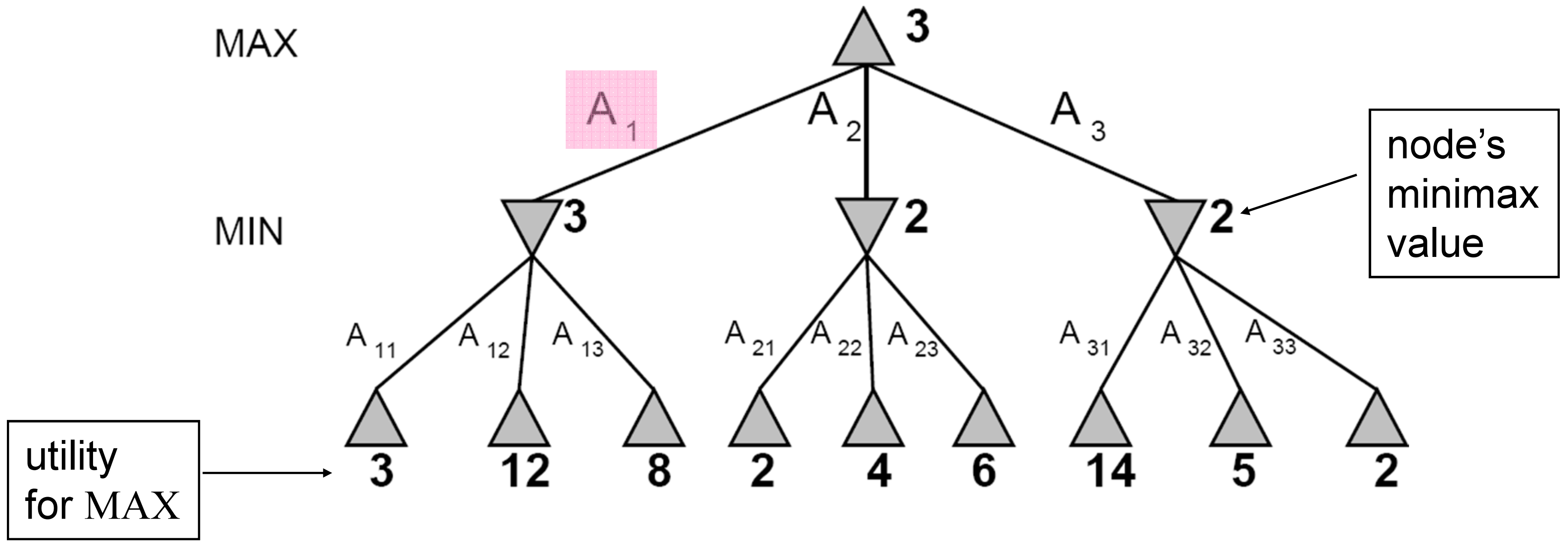
$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \end{cases}$$

➔ MAX will prefer to move to a state with maximum value
➔ MIN will prefer to move to a state with minimum value

The Minimax Decision

If MIN plays rationally, it will choose the action with minimum MINIMAX-VALUE

➔ Optimal decision for MAX:
choose action with highest MINIMAX-VALUE (“minimax decision”)
➔ leads to optimal play by MAX



➔ max(min(3,12,8),min(2,4,6),min(14,5,2)) = max(3,2,2) = 3

The Minimax Algorithm

➔ *recursive depth-first traversal of the game tree*

… *because MAX starts*

```
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
```

minimax.pl → ttt_minimax(x)

Properties of Minimax

Completeness:

Minimax is complete, if game tree is finite (chess has special rules for this)

Optimality:

Yes, if opponent also plays optimally
(otherwise there might be better strategies)

Time Complexity (in terms of nodes generated):

$O(b^m)$

Space Complexity:

$O(bm)$ (depth-first exploration of search tree)

For chess, $\boldsymbol{b} \approx 35$, $\boldsymbol{m} \approx 100$ for „reasonable“ games ➔ exact solution impossible
But: do we really need to explore every path in order to play optimally?

$\boldsymbol{b}$ ... branching factor (max. number of successors of any node)
$\boldsymbol{m}$ ... maximum length of any path in the state space (may be infinite)

α-β Pruning

Question:

Can we compute the ***correct*** minimax decision (i.e., the same decision that the minimax algorithm would compute) without looking at every node in the game tree?

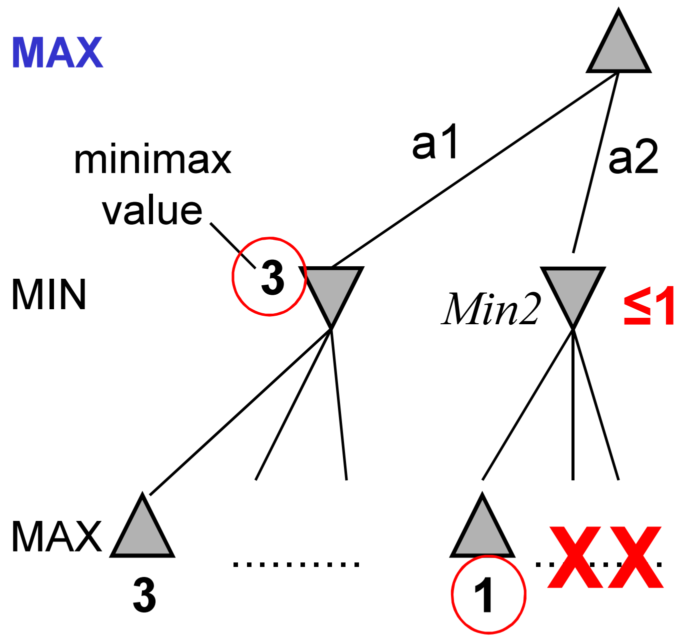
„Pruning":

Eliminating parts of a tree without examining them in detail (literally: „pruning" means cutting branches off a (real) tree)

Alpha-beta pruning:

Specific method for computing the correct minimax decision, while ignoring subtrees that cannot possibly influence the final decision

α-β Pruning



Basic idea:

- assume **MAX** has already computed the minimax value for action a1 (3)
- **MAX** is now considering an alternative action a2 and looks at possible reactions of MIN to a2 (i.e., the MAX nodes below the MIN node *Min2*)
- assume one of these turns out to have a minimax of 1

  ➔ this is less than the utility **MAX** can guarantee itself so far (3 for action a1)

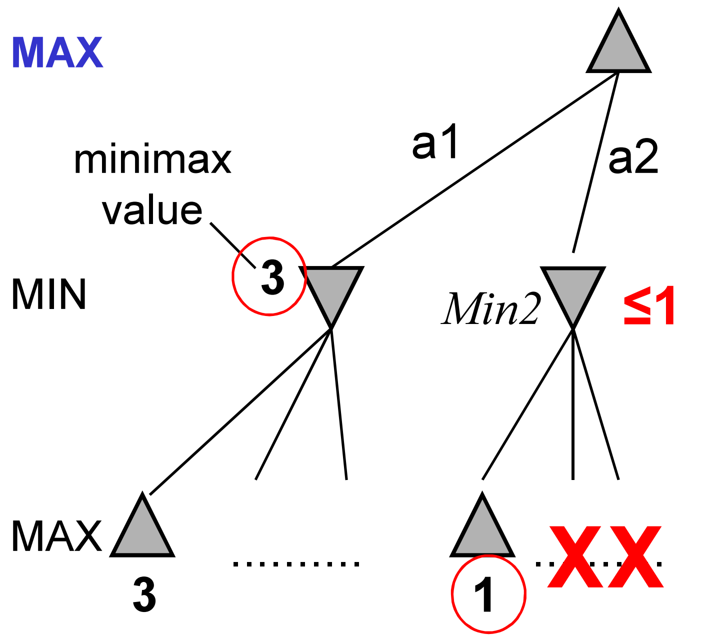
➔ there is no point looking at the other successors of *Min2* because MIN will either choose this one or one that is even worse (i.e., lower) for MAX

➔ no need to think about a2 any further; **MAX** will not choose a2 anyway

---

- This is what is formalised in function MAX-VALUE on the following algorithm slide
- MIN thinks analogously when considering alternatives below a MAX node ...

α-β Pruning



Realisation:

propagate two parameters along the expansion of a path, and update them when backing up: $[\alpha, \beta]$

α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

Pruning:

- Whenever a minimax value below a MIN node is below the current α
  ➔ ignore remaining nodes (subtrees) below the MIN node
  (because MAX will not choose this MIN node)
- Whenever a minimax value below a MAX node is higher than the current β
  ➔ ignore remaining nodes (subtrees) below the MAX node
  (because MIN will not choose this MAX node)

α-β Pruning: The Algorithm

-∞,+∞

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

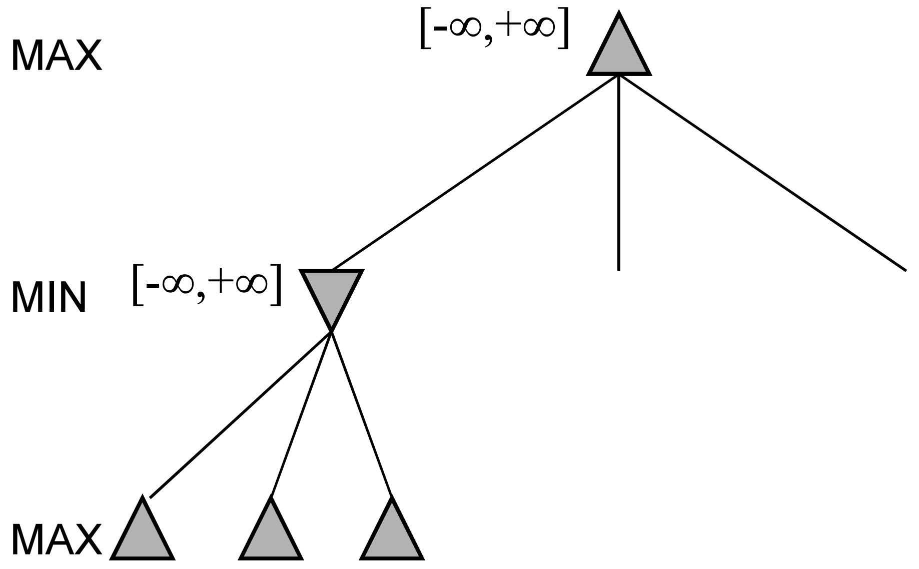
```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    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
```

pruning – leave loop – ignore the remaining successor nodes

keep track of best value for MAX found so far ...

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

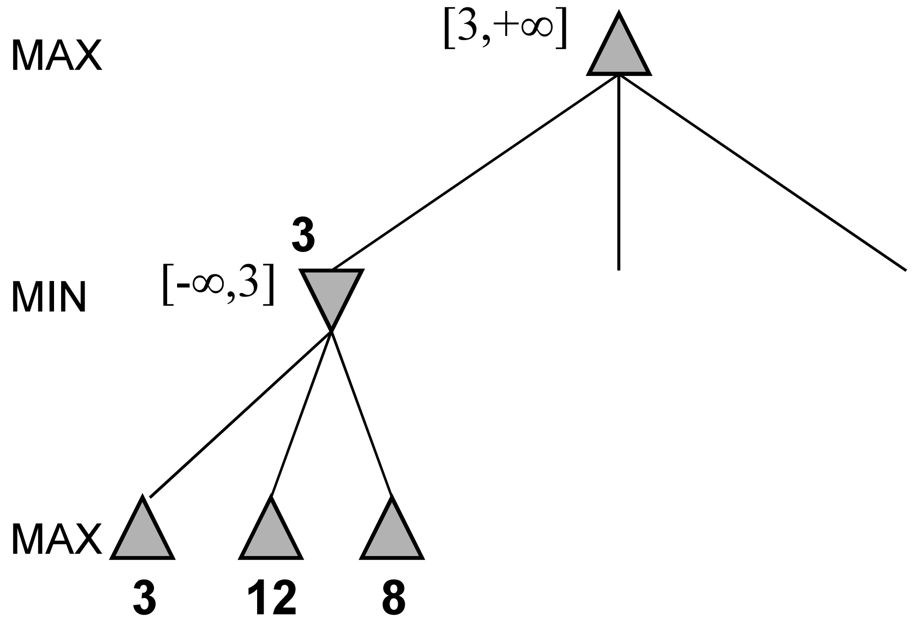
α-β Pruning: Example 1



```
 | ?-  ab_minimax(ra,X,V).
MAX: Expanding State rb (alpha = -inf, beta = +inf)
MIN: Expanding State 3 (alpha = -inf, beta = +inf)
     State 3 = terminal. Utility: 3
MIN: Expanding State 12 (alpha = -inf, beta = 3)
     State 12 = terminal. Utility: 12
MIN: Expanding State 8 (alpha = -inf, beta = 3)
     State 8 = terminal. Utility: 8
MIN: => choose move 3 (eval: 3)
MAX: Expanding State rc (alpha = 3, beta = +inf)
MIN: Expanding State 2 (alpha = 3, beta = +inf)
     State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: Expanding State rd (alpha = 3, beta = +inf)
MIN: Expanding State 14 (alpha = 3, beta = +inf)
     State 14 = terminal. Utility: 14
MIN: Expanding State 5 (alpha = 3, beta = 14)
     State 5 = terminal. Utility: 5
MIN: Expanding State 2 (alpha = 3, beta = 5)
     State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: => choose move rb (eval: 3)
V = 3,
X = rb ?
yes
```

α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

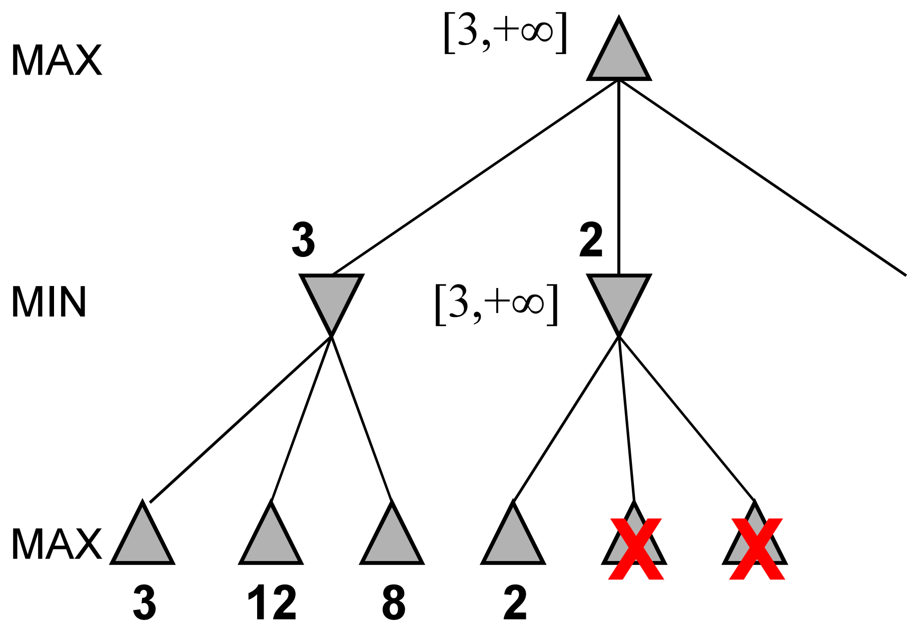
α-β Pruning: Example 1



```
  | ?-  ab_minimax(ra,X,V).
MAX: Expanding State rb (alpha = -inf, beta = +inf)
MIN: Expanding State 3 (alpha = -inf, beta = +inf)
       State 3 = terminal. Utility: 3
MIN: Expanding State 12 (alpha = -inf, beta = 3)
       State 12 = terminal. Utility: 12
MIN: Expanding State 8 (alpha = -inf, beta = 3)
       State 8 = terminal. Utility: 8
MIN: => choose move 3 (eval: 3)
MAX: Expanding State rc (alpha = 3, beta = +inf)
MIN: Expanding State 2 (alpha = 3, beta = +inf)
       State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: Expanding State rd (alpha = 3, beta = +inf)
MIN: Expanding State 14 (alpha = 3, beta = +inf)
       State 14 = terminal. Utility: 14
MIN: Expanding State 5 (alpha = 3, beta = 14)
       State 5 = terminal. Utility: 5
MIN: Expanding State 2 (alpha = 3, beta = 5)
       State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: => choose move rb (eval: 3)
V = 3,
X = rb ?
yes
```

α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

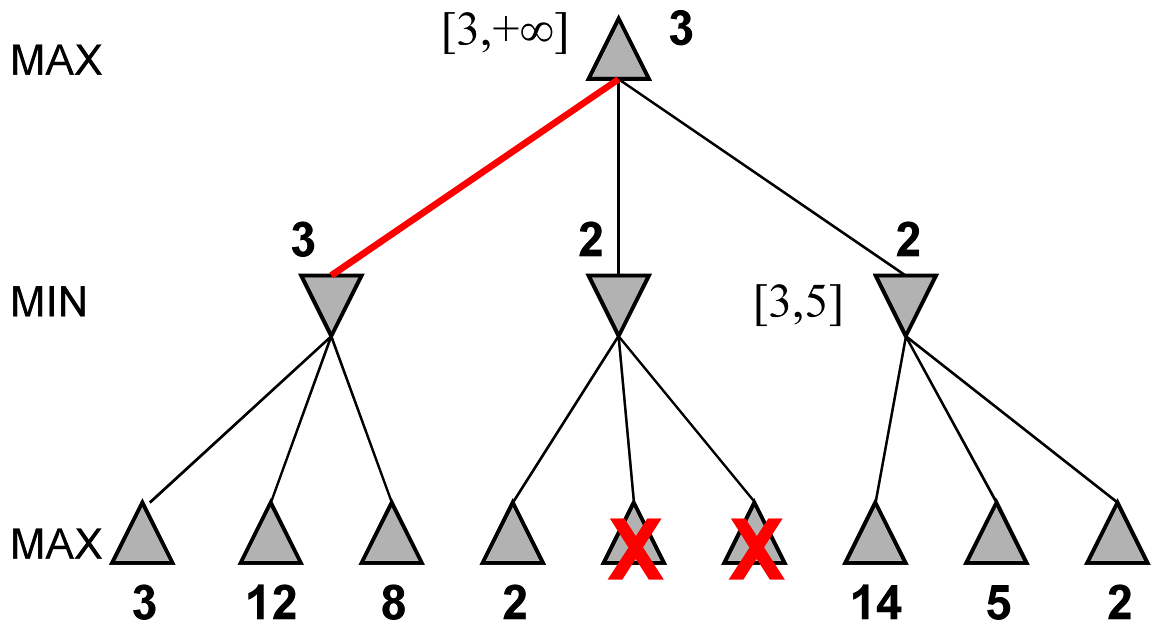
α-β Pruning: Example 1



```
 | ?-  ab_minimax(ra,X,V).
MAX: Expanding State rb (alpha = -inf, beta = +inf)
MIN: Expanding State 3 (alpha = -inf, beta = +inf)
      State 3 = terminal. Utility: 3
MIN: Expanding State 12 (alpha = -inf, beta = 3)
      State 12 = terminal. Utility: 12
MIN: Expanding State 8 (alpha = -inf, beta = 3)
      State 8 = terminal. Utility: 8
MIN: => choose move 3 (eval: 3)
MAX: Expanding State rc (alpha = 3, beta = +inf)
MIN: Expanding State 2 (alpha = 3, beta = +inf)
      State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: Expanding State rd (alpha = 3, beta = +inf)
MIN: Expanding State 14 (alpha = 3, beta = +inf)
      State 14 = terminal. Utility: 14
MIN: Expanding State 5 (alpha = 3, beta = 14)
      State 5 = terminal. Utility: 5
MIN: Expanding State 2 (alpha = 3, beta = 5)
      State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: => choose move rb (eval: 3)
V = 3,
X = rb ?
yes
```

α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

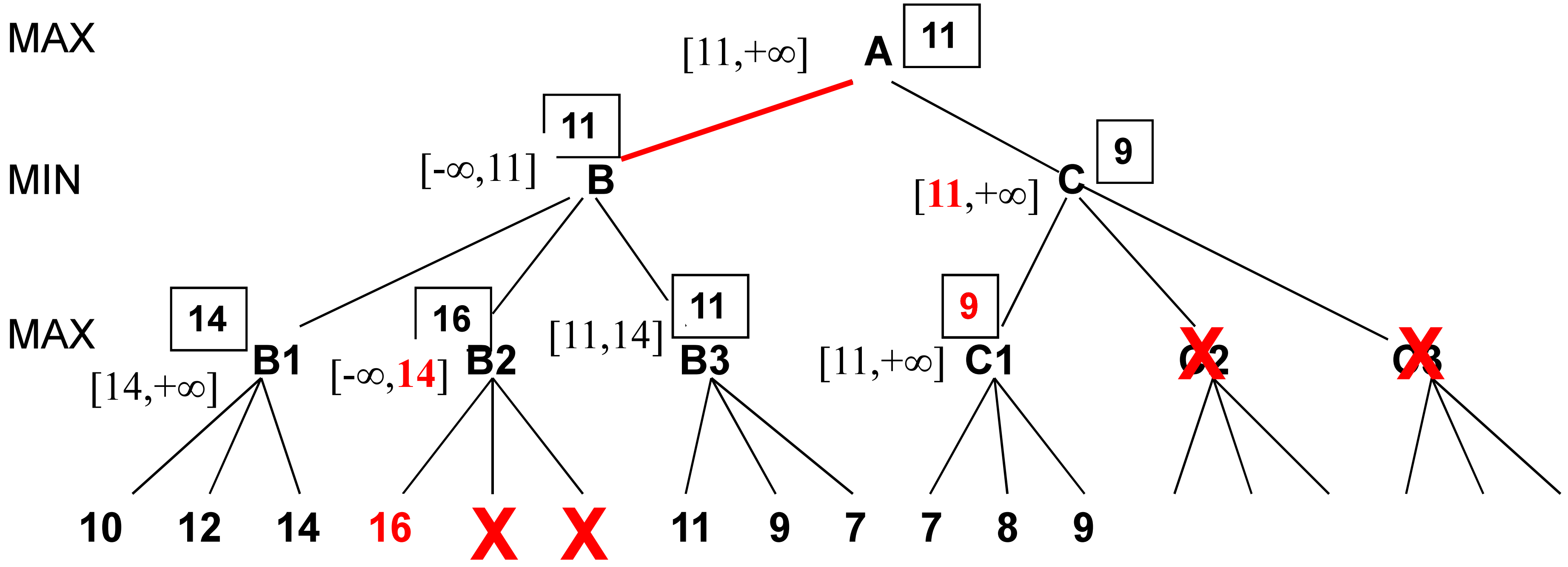
α-β Pruning: Example 1



```
 | ?-  ab_minimax(ra,X,V).
MAX: Expanding State rb (alpha = -inf, beta = +inf)
MIN: Expanding State 3 (alpha = -inf, beta = +inf)
      State 3 = terminal. Utility: 3
MIN: Expanding State 12 (alpha = -inf, beta = 3)
      State 12 = terminal. Utility: 12
MIN: Expanding State 8 (alpha = -inf, beta = 3)
      State 8 = terminal. Utility: 8
MIN: => choose move 3 (eval: 3)
MAX: Expanding State rc (alpha = 3, beta = +inf)
MIN: Expanding State 2 (alpha = 3, beta = +inf)
      State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: Expanding State rd (alpha = 3, beta = +inf)
MIN: Expanding State 14 (alpha = 3, beta = +inf)
      State 14 = terminal. Utility: 14
MIN: Expanding State 5 (alpha = 3, beta = 14)
      State 5 = terminal. Utility: 5
MIN: Expanding State 2 (alpha = 3, beta = 5)
      State 2 = terminal. Utility: 2
MIN: PRUNING AFTER State 2 -- Eval (2) <= Alpha (3)
MIN: => choose move 2 (eval: 2)
MAX: => choose move rb (eval: 3)
V = 3,
X = rb ?
yes
```

α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

α-β Pruning: Example 2



α ... best (highest) value found so far for MAX
β ... best (lowest) value found so far for MIN

Properties of α-β

- Pruning does not affect the final result
- Good ordering of moves improves effectiveness in pruning

  ➔ Exercise: according to what criterion should moves be ordered (ideally)?
- With *„perfect ordering“,* time complexity would be $O(b^{m/2})$

  ➔ doubles solvable depth!

... unfortunately, 30^{50} is still unsolvable ...

(*Exercise:* write a program that plays tic-tac-toe against a human opponent, using alpha-beta pruning)

minimax.pl → ttt_ab(x)

Realistic Case: Searching with Resource Limits

Problem with minimax / alpha-beta:

- minimax generates entire game tree
- alpha-beta prunes part of the game tree, but still has to search all the way to terminal states
  ➔ exponential time complexity
- in real game, moves must be made in limited time (e.g., a few minutes)!

Two important changes to the search strategy:

1. Impose **depth limit**
   ➔ use CUTOFF-TEST instead of TERMINAL-TEST
   (in the simplest case: cut off search when depth limit is reached; more complex strategies are possible (e.g., „quiescence search“))
2. Apply **heuristic evaluation function** EVAL (to non-terminal states) instead of well-defined UTILITY

Heuristic Evaluation Functions

Desired properties of evaluation function $\mathrm{EVAL}(x)$ **:**

- Should return an estimate of the expected utility of the game from a certain position (state) S
  ➔ estimate of the „value“ (for MAX) of position S
- Clearly: if S is a terminal state ➔ $\mathrm{EVAL}(S) = \mathrm{UTILITY}(S)$
- If S is non-terminal state: $\mathrm{EVAL}(S)$ should somehow be correlated with chances of winning the game
- Important: $\mathrm{EVAL}(x)$ must be efficiently computable from the state itself, without lookahead !
  (otherwise: could just call $\mathrm{MINIMAX}(S)$ as (perfect) evaluation function ...)

Feature-based Evaluation Functions

A simple example: Weighted linear function $\mathrm{EVAL}(s)$

- Compute various **features (properties)** $f_i(S)$ of a state S
  - simple features (e.g., the number of pawns possessed by each side)
  - or more complex ones (e.g., „pawn structure“, „king safety“)
- Define evaluation function as linear weighted combination of these features:

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

In good chess programs, non-linear functions are used
to capture, e.g., the fact that features are not independent in their value (e.g., a pair of bishops is worth more than twice the value of a single bishop; or a bishop is worth more in the endgame than in the beginning)

Cutting Off Search

Simple modification to ALPHA-BETA-SEARCH:

- keep track of search depth
- replace lines

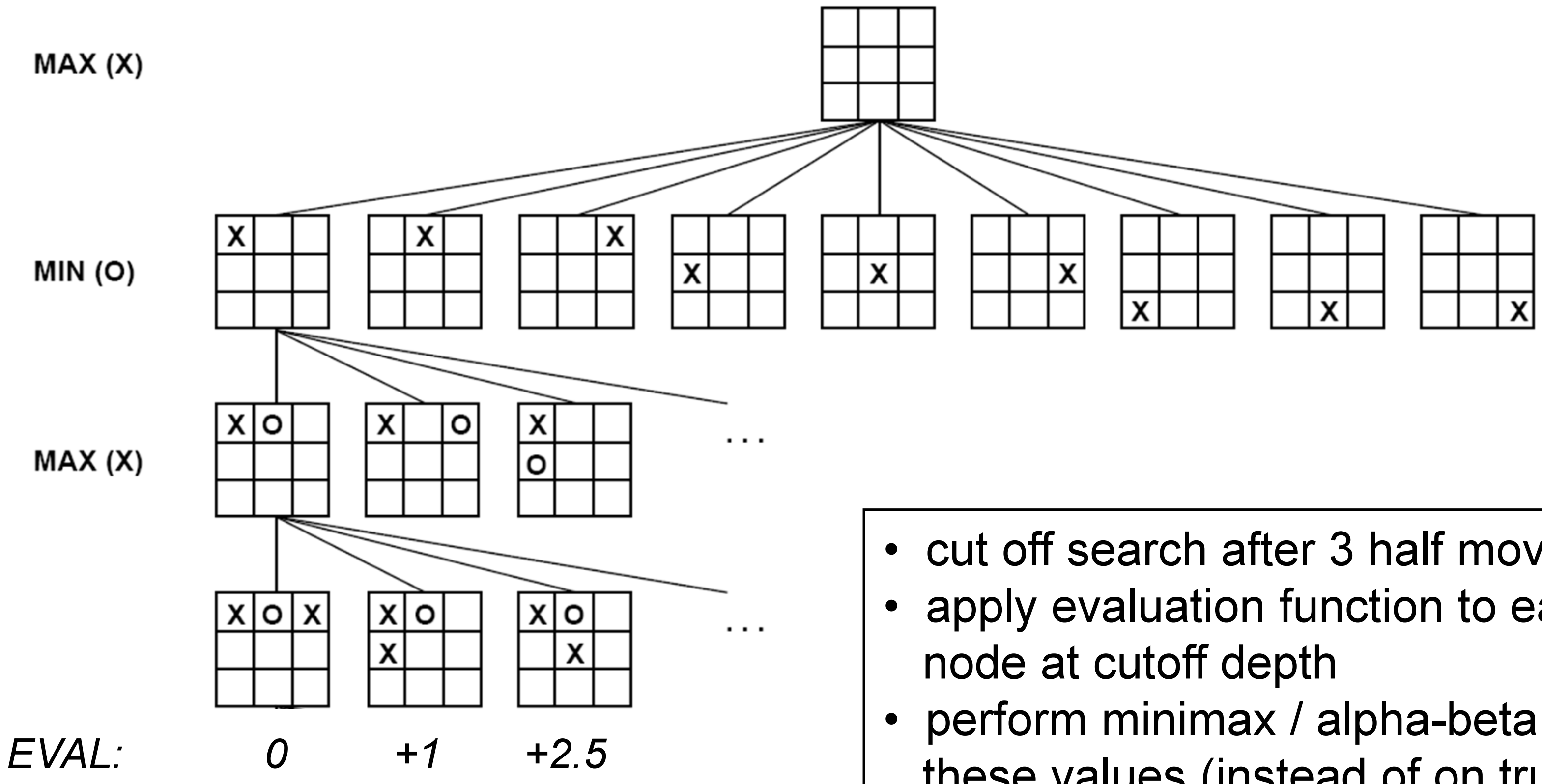
  if TERMINAL-TEST(*state*) then return UTILITY(*state*)

  with

  if **CUTOFF-TEST(*state,depth*)** then return **EVAL(*state*)**
- simplest strategy: set fixed depth limit

  ➔ CUTOFF-TEST returns true when depth limit is reached

Cutting Off Search in Tic-Tac-Toe



- cut off search after 3 half moves
- apply evaluation function to each node at cutoff depth
- perform minimax / alpha-beta on these values (instead of on true utility)

(*Exercise:* try to think of a good evaluation function for tic-tac-toe)

Cutting Off Search

Simple modification to ALPHA-BETA-SEARCH:

- keep track of search depth
- replace lines

  if TERMINAL-TEST(*state*) then return UTILITY(*state*)

  with

  if **CUTOFF-TEST(*state*,*depth*)** then return **EVAL(*state*)**
- simplest strategy: set fixed depth limit

  ➔ CUTOFF-TEST returns true when depth limit is exceeded

In reality (e.g., in good chess programs):

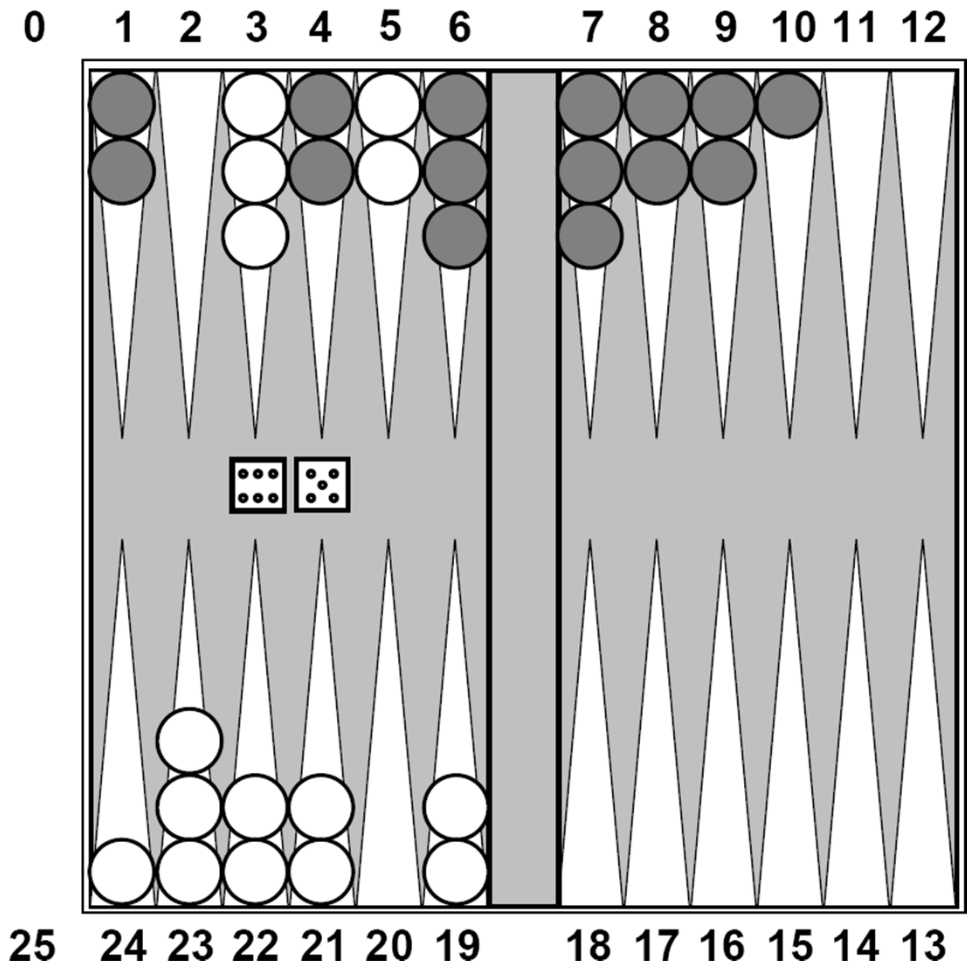
- perform **selective deepening** of search tree
- don´t cut off search in positions that might be critical (e.g., queen in danger)

➔ „quiescence search“ (expand positions further that are not „quiescent“, i.e., that are highly unstable)

➔ still other, more complex strategies possible ...

Non-deterministic Games

An example: Backgammon



Non-deterministic Games

Problem with non-deterministic games:

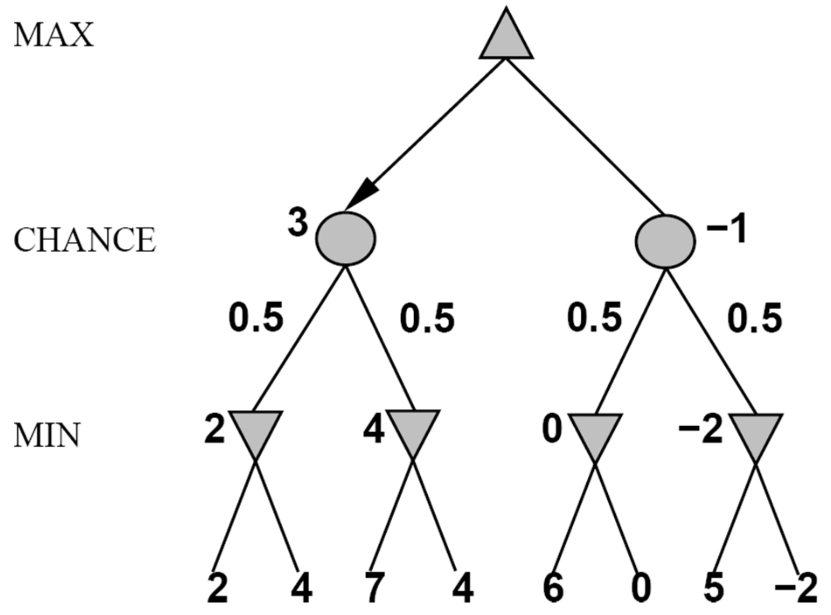
- Chance (probabilistic randomness) introduced by, e.g., dice or card shuffling
- Not known what the available moves will be at any point
- Impossible to construct a standard minimax game tree

Solution:

- Model chance explicitly
- Include **„chance nodes“** in game tree (in addition to MIN and MAX nodes)
- Instead of computing true minimax value, compute
  expected minimax value
  by computing average minimax values over all ways the chance node (e.g., dice roll) could turn out, weighted by the respective probability

EXPECTIMINIMAX

Game tree with chance nodes:
Simplified example with coin-flipping



➔ EXPECTIMINIMAX **Algorithm:**

```
...
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)
              weighted
...
```

State of the Art in Game Playing

- **Checkers (1):**
  Chinook ended 40-year reign of human world champion Marion Tinsley in 1994.
  Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443.748.401.247 positions.

A Bit of Trivia: Marion Tinsley (1927 – 1995)

Marion Tinsley is the greatest checkers player in the history of the game.
He was recognized as the World Champion in 1954 and since that time lost only 9 games. Dr. Tinsley was awarded the title of World Champion Emeritus in honour of his decisive domination of and contribution to the game.

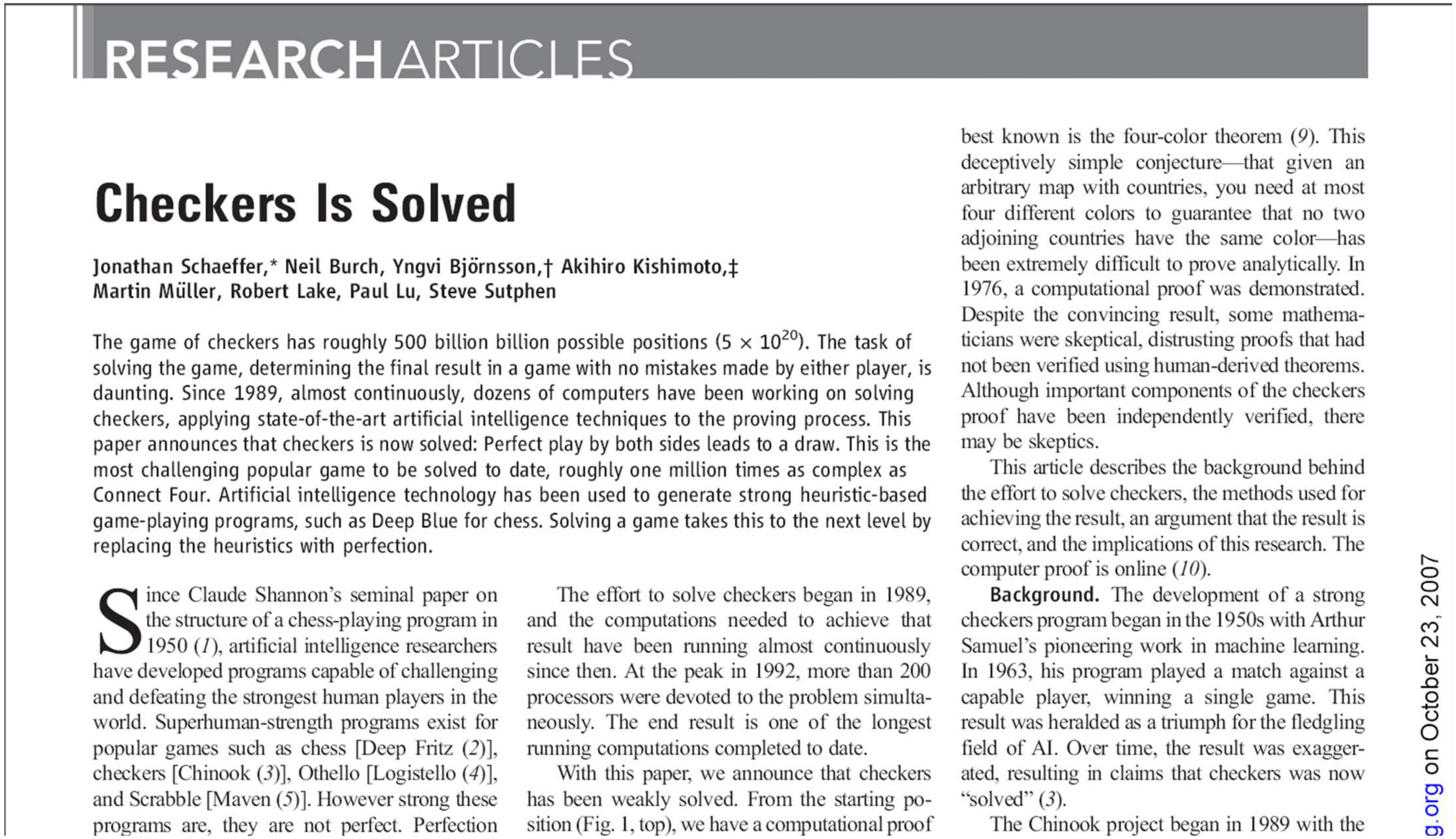
(http://www.cs.ualberta.ca/~chinook/people/Tinsley.php)

State of the Art in Game Playing

- **Checkers (1):**
  Chinook ended 40-year reign of human world champion Marion Tinsley in 1994.
  Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443.748.401.247 positions.

- **Checkers (2):**
  In Sept. 2007, AI Researchers from the Chinook Team (Univ. of Alberta, Canada) announced that the game of Checkers is now **(weakly) solved**:
  It was proven that given perfect play by both players, the game ends in a draw.
  However, no perfect strategy for playing the game optimally from any given board position is known.

Jonathan Schaeffer, *et al.*
Science **317**, 1518 (Sept. 2007);
DOI: 10.1126/science.1144079



RESEARCH ARTICLES

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 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.

Since Claude Shannon's seminal paper on the structure of a chess-playing program in 1950 (*1*), artificial intelligence researchers have developed programs capable of challenging and defeating the strongest human players in the world. Superhuman-strength programs exist for popular games such as chess [Deep Fritz (*2*)], checkers [Chinook (*3*)], Othello [Logistello (*4*)], and Scrabble [Maven (*5*)]. However strong these programs are, they are not perfect. Perfection

The effort to solve checkers began in 1989, and the computations needed to achieve that result have been running almost continuously since then. At the peak in 1992, more than 200 processors were devoted to the problem simultaneously. The end result is one of the longest running computations completed to date.

With this paper, we announce that checkers has been weakly solved. From the starting position (Fig. 1, top), we have a computational proof

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 proof have been independently verified, there may be skeptics.

This article describes the background behind the effort to solve checkers, the methods used for achieving the result, an argument that the result is correct, and the implications of this research. The computer proof is online (*10*).

Background. The development of a strong checkers program began in the 1950s with Arthur Samuel's pioneering work in machine learning. In 1963, his program played a match against a capable player, winning a single game. This result was heralded as a triumph for the fledgling field of AI. Over time, the result was exaggerated, resulting in claims that checkers was now "solved" (*3*).

The Chinook project began in 1989 with the

Jonathan Schaeffer, *et al.*
Science **317**, 1518 (Sept. 2007);
DOI: 10.1126/science.1144079

Table 1. The number of positions in the game of checkers. For example, the possible positions for one piece include 32 squares for the Black king, 32 squares for the White king, 28 squares for a Black checker, and 28 squares for a White checker, for a total of 120 positions.

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

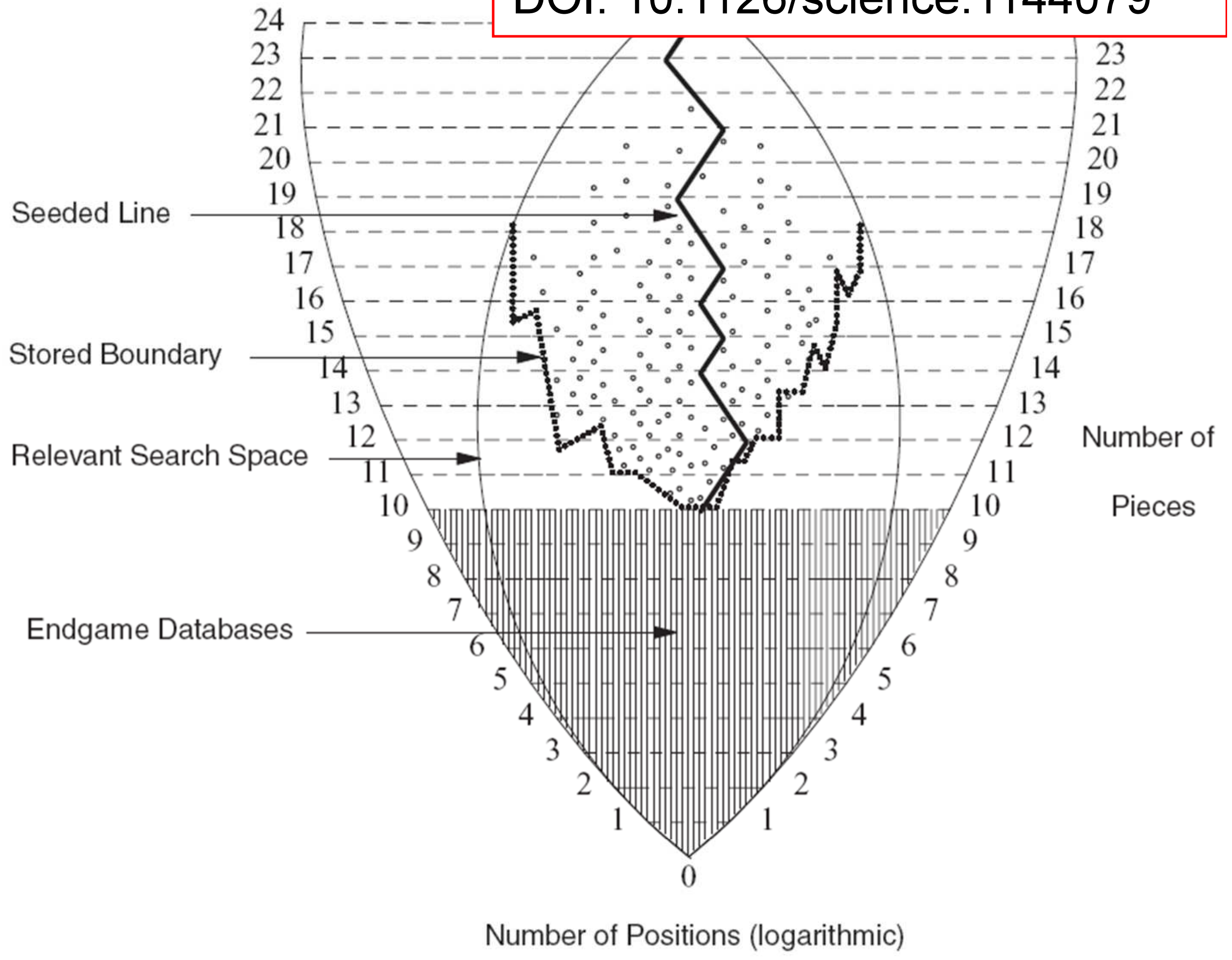


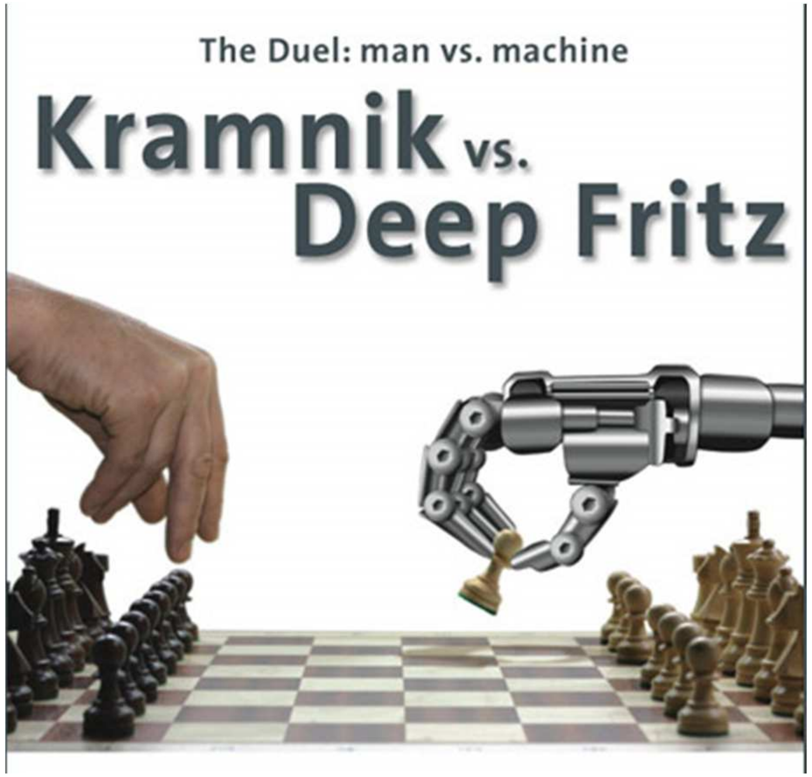
Fig. 2. Forward and backward search. The number of pieces on the board are plotted (vertically) versus the logarithm of the number of positions (Table 1). The shaded area shows the endgame database part of the proof—i.e., all positions with ≤10 pieces. The inner oval area shows that only a portion of the search space is relevant to the proof. Positions may be irrelevant because they are unreachable or are not required for the proof. The small open circles indicate positions with more than 10 pieces for which a value has been proven by a solver. The dotted line shows the boundary between the top of the proof tree that the manager sees (and stores on disk) and the parts that are computed by the solvers (and are not saved in order to reduce disk storage needs). The solid seeded line shows a "best" sequence of moves.

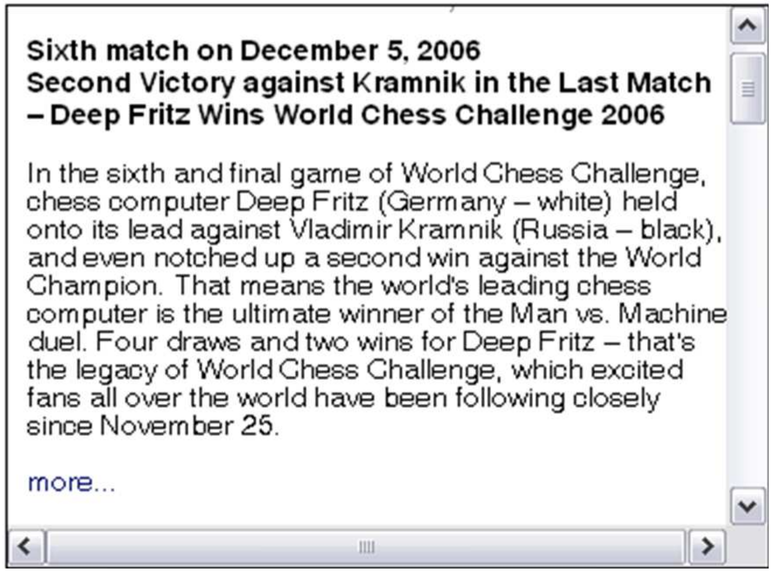
State of the Art in Game Playing

- **Chess (1):**
  1997: For the first time, a chess computer beats the world champion:
  Deep Blue defeated human world champion Garry Kasparov in a six-game match.
  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.

- **Chess (2):**
  Computers now routinely beat human grandmasters.
  2006: Deep Fritz beats the (then) current world champion Vladimir Kramnik in a six-game match (4:2). The program runs on standard hardware with an Intel Core 2 Duo processor and computes only around 8 million positions / sec, to a depth of around 17-18 ply.
  The power is in the improved heuristics (and databases)

World Chess Challenge 2006: Machine vs. World Champion Vladimir Kramnik





Sixth match on December 5, 2006
Second Victory against Kramnik in the Last Match – Deep Fritz Wins World Chess Challenge 2006

In the sixth and final game of World Chess Challenge, chess computer Deep Fritz (Germany – white) held onto its lead against Vladimir Kramnik (Russia – black), and even notched up a second win against the World Champion. That means the world's leading chess computer is the ultimate winner of the Man vs. Machine duel. Four draws and two wins for Deep Fritz – that's the legacy of World Chess Challenge, which excited fans all over the world have been following closely since November 25.

more...

http://de.wikipedia.org/wiki/Fritz_(Schach)

State of the Art in Game Playing

- **Othello:** human champions refuse to compete against computers, which are too good.
- **Go:** human champions refuse to compete against computers, which are **too bad**. In Go, b > 300, so most programs use pattern knowledge bases to suggest plausible moves.
- **Backgammon:** TD-Gammon achieves world-class level around 1995; searches only to depth 2, but has a **very good evaluation function (learned by itself!)**
  ➔ *will come back to this when we talk about machine learning in this class*

Summary

- **Games** are challenging problems that drive AI research
- Game can be defined as a **search problem** (initial state, legal actions, terminal test, utility function)
- Search problem defines a **game tree** (with layers according to the 2 players)
- In two-player zero-sum games with **perfect information**, the **minimax algorithm** selects optimal moves via depth-first search in the game tree
- **Alpha-beta pruning** computes the same moves as minimax, but eliminates provably irrelevant subtrees
- Usually: complete enumeration of game tree is impossible
  ➔ search must be **cut off**, and a **heuristic evaluation function** computes estimated utility of non-terminal states
- **Games of chance** can be modeled by including chance nodes in game tree
  ➔ **expectiminimax algorithm**