

Foundations of Artificial Intelligence

6. Board Games

Search Strategies for Games, Games with Chance, State of the Art

Wolfram Burgard, Bernhard Nebel, and Martin Riedmiller



Albert-Ludwigs-Universität Freiburg

May 31, 2011

Contents

- 1 Board Games
- 2 Minimax Search
- 3 Alpha-Beta Search
- 4 Games with an Element of Chance
- 5 State of the Art

Why Board Games?

Board games are one of the oldest branches of AI (Shannon and Turing 1950).

- Board games present a very abstract and pure form of competition between two opponents and clearly require a form of “intelligence”.
- The states of a game are easy to represent.
- The possible actions of the players are well-defined.

→ Realization of the game as a search problem

→ The world states are fully accessible

→ It is nonetheless a contingency problem, because the characteristics of the opponent are not known in advance.

Board games are not only difficult because they are **contingency problems**, but also because **the search trees can become astronomically large**.

Examples:

- **Chess**: On average 35 possible actions from every position; often, games have 50 moves per player, resulting in a search depth of 100:
→ $35^{100} \approx 10^{150}$ nodes in the search tree (with “only” 10^{40} legal chess positions).
- **Go**: On average 200 possible actions with ca. 300 moves
→ $200^{300} \approx 10^{700}$ nodes.

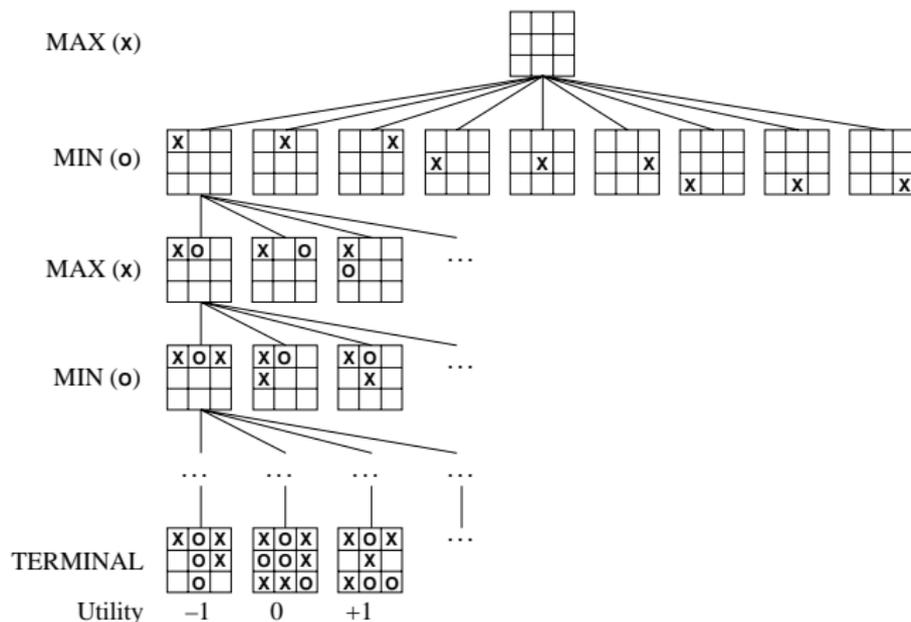
Good game programs have the properties that they

- delete irrelevant branches of the game tree,
- use good evaluation functions for in-between states, and
- look ahead as many moves as possible.

Terminology of Two-Person Board Games

- **Players** are MAX and MIN, where MAX begins.
- **Initial position** (e.g., board arrangement)
- **Operators** (= legal moves)
- **Termination test**, determines when the game is over. Terminal state = game over.
- **Strategy**. In contrast to regular searches, where a path from beginning to end is simply a solution, MAX must come up with a strategy to reach a terminal state *regardless of what* MIN *does* → correct reactions to all of MIN's moves.

Tic-Tac-Toe Example



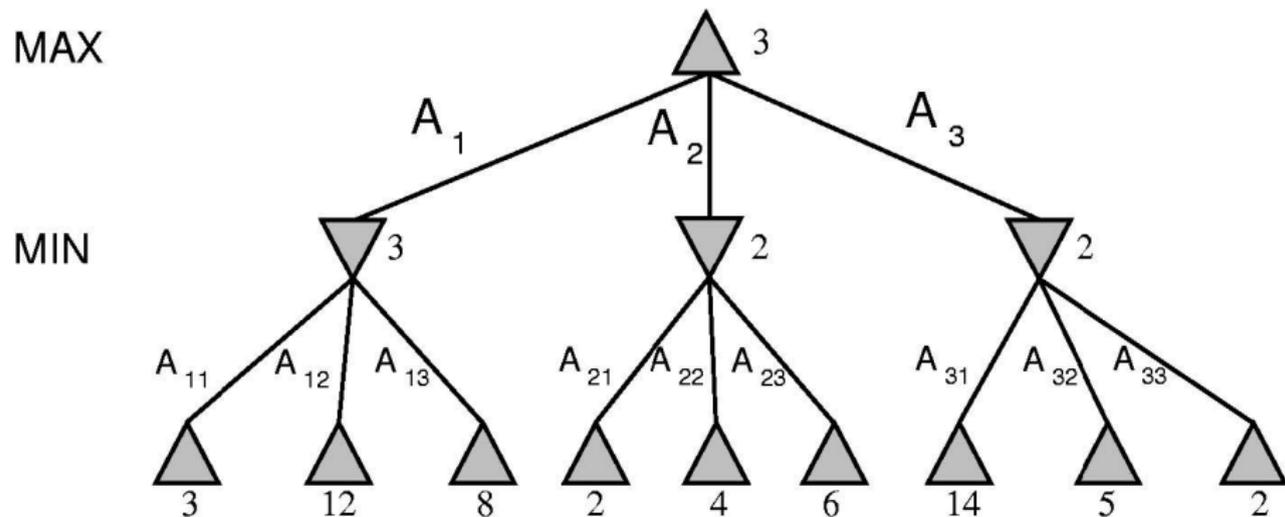
Every step of the [search tree](#), also called game tree, is given the player's name whose turn it is (MAX- and MIN-steps).

When it is possible, as it is here, to produce the full search tree (game tree), the [minimax algorithm](#) delivers an [optimal strategy](#) for MAX.

1. Generate the complete game tree using depth-first search.
2. Apply the utility function to each terminal state.
3. Beginning with the terminal states, determine the utility of the predecessor nodes as follows:
 - Node is a MIN-node
Value is the **minimum** of the successor nodes
 - Node is a MAX-node
Value is the **maximum** of the successor nodes
 - From the initial state (root of the game tree), MAX chooses the move that leads to the highest value (**minimax decision**).

Note: Minimax assumes that MIN plays perfectly. Every weakness (i.e., every mistake MIN makes) can only improve the result for MAX.

Minimax Example



Minimax Algorithm

Recursively calculates the best move from the initial state.

```
function MINIMAX-DECISION(state) returns an action  
return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(state, a))$ 
```

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

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

Note: Minimax only works when the game tree is not too deep. Otherwise, the minimax value must be approximated.

Evaluation Function

When the **search space is too large**, the game tree can be created to a **certain depth** only. The art is to **correctly evaluate the playing position of the leaves**.

Example of simple evaluation criteria in chess:

- Material value: pawn 1, knight/bishop 3, rook 5, queen 9
- Other: king safety, good pawn structure
- Rule of thumb: 3-point advantage = certain victory

The choice of evaluation function is decisive!

The value assigned to a state of play should reflect the chances of winning, i.e., the chance of winning with a 1-point advantage should be less than with a 3-point advantage.

Evaluation Function - General

The preferred evaluation functions are weighted, linear functions:

$$w_1 f_1 + w_2 f_2 + \dots + w_n f_n$$

where the w 's are the weights, and the f 's are the features. [e.g., $w_1 = 3$, $f_1 =$ number of our own knights on the board]

The above linear sum makes a strong assumption: the contribution of each feature are independent. (not true: e.g. bishops in the endgame are more powerful, when there is more space)

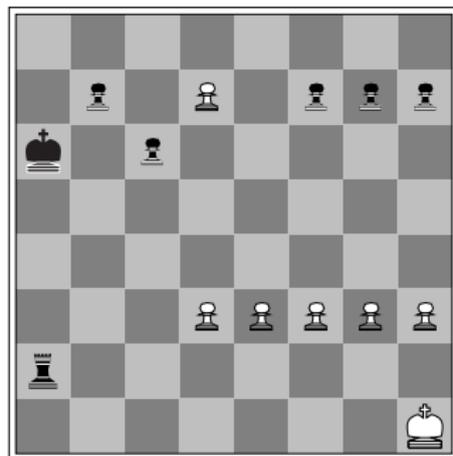
The weights can be learned. The features, however, are often designed by human intuition and understanding

When Should we Stop Growing the Tree?

Motivation: Return an answer within the allocated time.

- Fixed-depth search
- Better: iterative deepening search (stop, when time is over)
- but only stop and evaluate at 'quiescent' positions that won't cause large fluctuations in the evaluation function in the following moves. E.g. if one can capture a figure, then the position is not 'quiescent' because this might change the evaluation dramatically. Solution: Continue search at non quiescent positions, favorably by only allowing certain types of moves (e.g. capturing) to reduce search effort, until a quiescent position was reached.
- problem of limited depth search: horizon effect (see next slide)

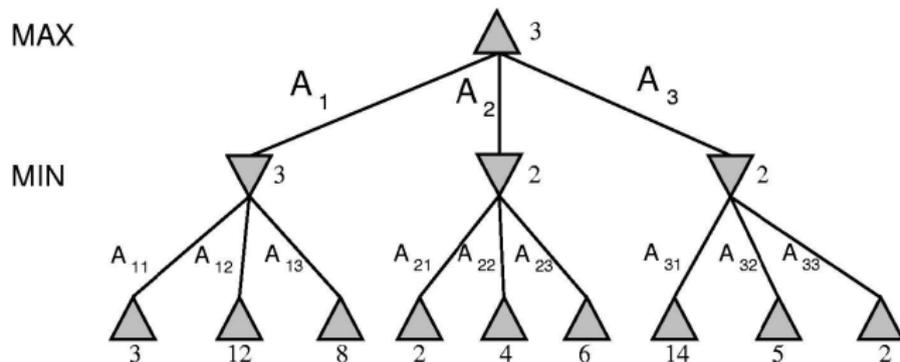
Horizon Problem



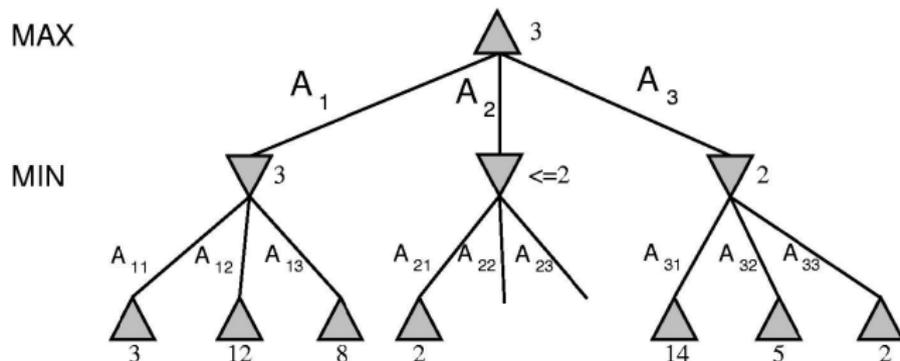
Black to move

- Black has a slight material advantage
- ... but will eventually lose (pawn becomes a queen)
- A fixed-depth search cannot detect this because it thinks it can avoid it (on the other side of the horizon - because black is concentrating on the check with the rook, to which white must react).

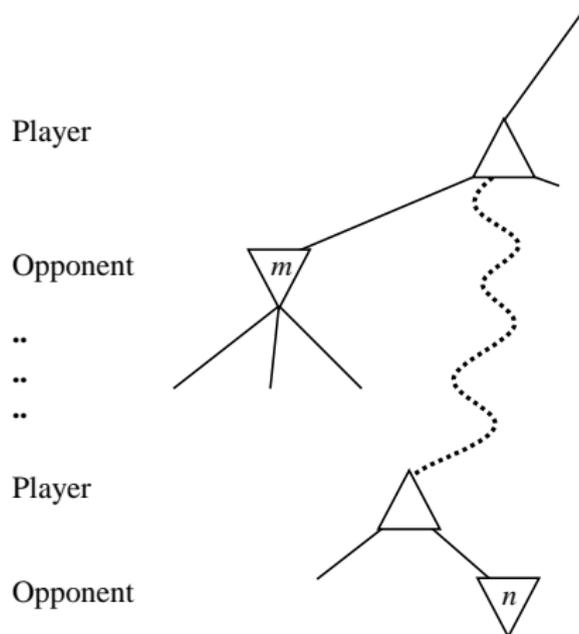
Alpha-Beta Pruning



Improvement possible? We do not need to consider all nodes.



Alpha-Beta Pruning: General



If $m > n$ we will never reach node n in the game.

Minimax algorithm with depth-first search

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.

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.

When Can we Prune?

The following applies:

α values of MAX nodes can never decrease

β values of MIN nodes can never increase

- (1) Prune below the MIN node whose β -bound is less than or equal to the α -bound of its MAX-predecessor node.
 - (2) Prune below the MAX node whose α -bound is greater than or equal to the β -bound of its MIN-predecessor node.
- Provides the same results as the complete minimax search to the same depth (because only irrelevant nodes are eliminated).

Alpha-Beta Search Algorithm

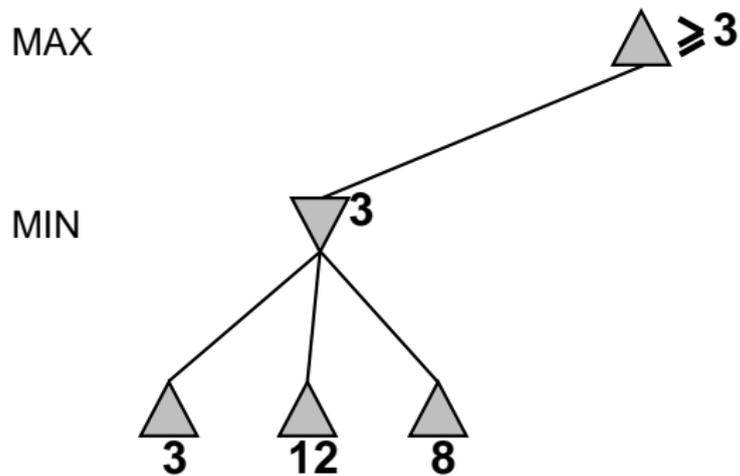
function ALPHA-BETA-SEARCH(*state*) **returns** an action
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$
 return the *action* in $\text{ACTIONS}(\text{state})$ with value v

function MAX-VALUE(*state*, α , β) **returns** a utility value
if $\text{TERMINAL-TEST}(\text{state})$ **then return** $\text{UTILITY}(\text{state})$
 $v \leftarrow -\infty$
 for each a **in** $\text{ACTIONS}(\text{state})$ **do**
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$
 if $v \geq \beta$ **then return** v
 $\alpha \leftarrow \text{MAX}(\alpha, v)$
 return v

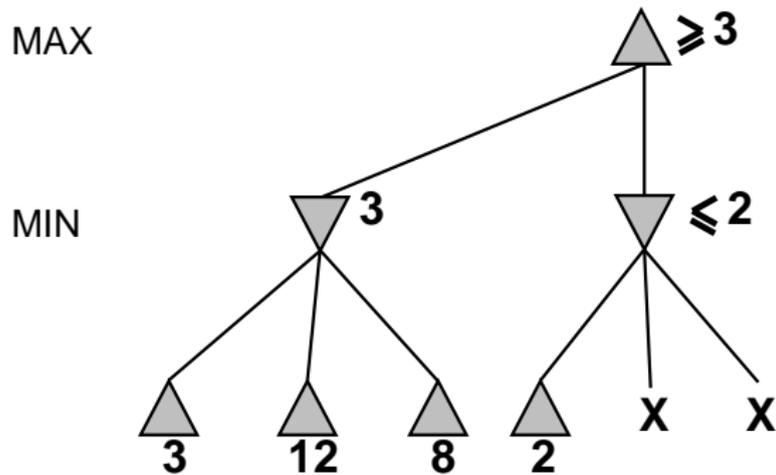
function MIN-VALUE(*state*, α , β) **returns** a utility value
if $\text{TERMINAL-TEST}(\text{state})$ **then return** $\text{UTILITY}(\text{state})$
 $v \leftarrow +\infty$
 for each a **in** $\text{ACTIONS}(\text{state})$ **do**
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$
 if $v \leq \alpha$ **then return** v
 $\beta \leftarrow \text{MIN}(\beta, v)$
 return v

Initial call with $\text{MAX-VALUE}(\text{initial-state}, -\infty, +\infty)$

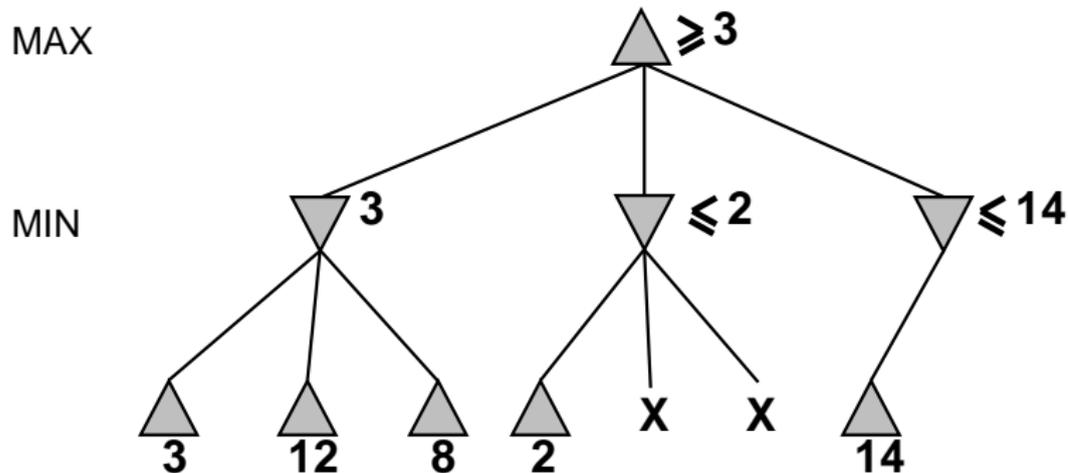
Alpha-Beta Pruning Example



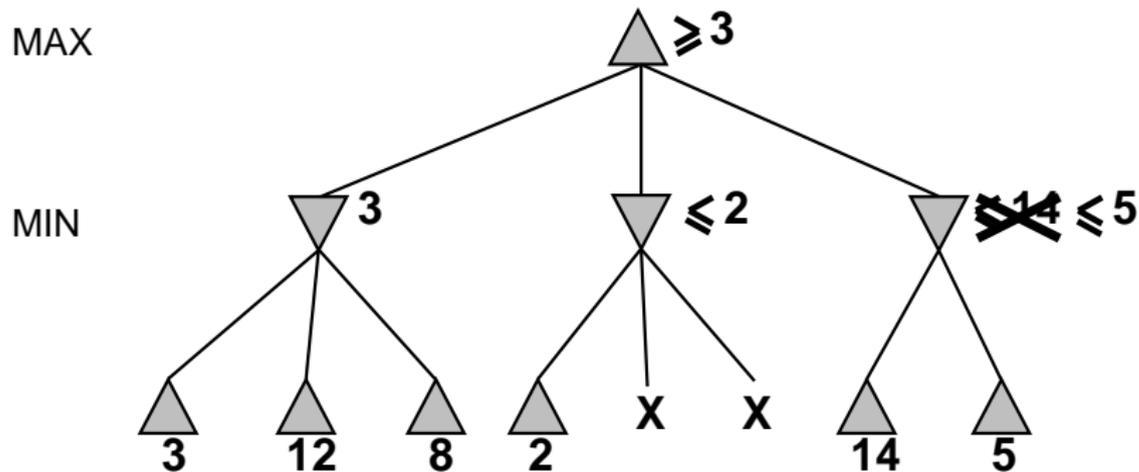
Alpha-Beta Pruning Example



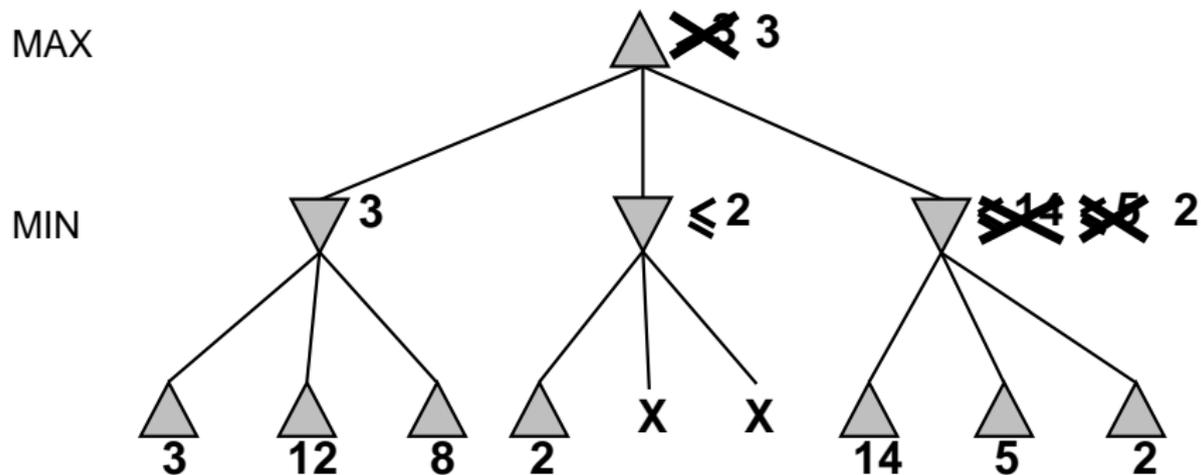
Alpha-Beta Pruning Example



Alpha-Beta Pruning Example



Alpha-Beta Pruning Example

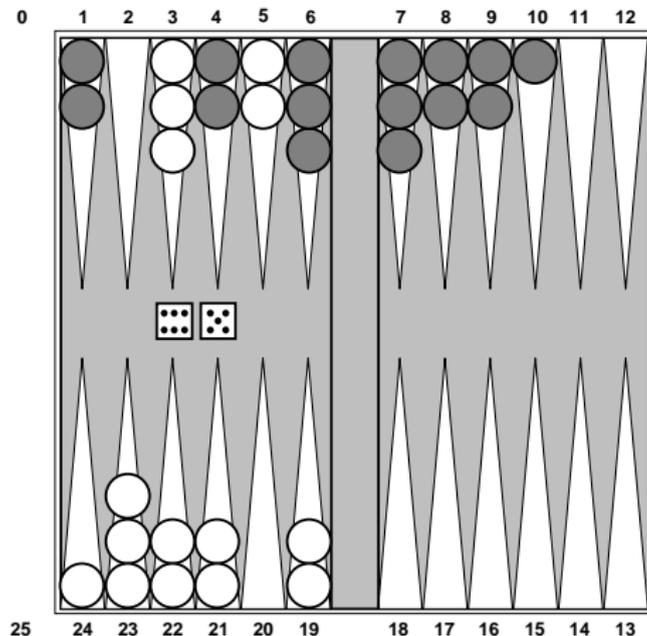


Efficiency Gain

- The alpha-beta search **cuts the largest amount off the tree** when we examine the **best move first**.
- In the **best case** (always the best move first), the search expenditure is reduced to $O(b^{d/2}) \Rightarrow$ we can search twice as deep in the same amount of time.
- In the average case (randomly distributed moves), the search expenditure is reduced to $O((b/\log b)^d)$.
- For moderate b ($b < 100$), we roughly have $O(b^{3d/4})$.
- However, best move typically is not known. **Practical case**: A simple ordering heuristic brings the performance close to the best case \Rightarrow In chess, we can thus reach a depth of 6-7 moves.

Good ordering for chess? try captures first, then threats, then forward moves, then backward moves

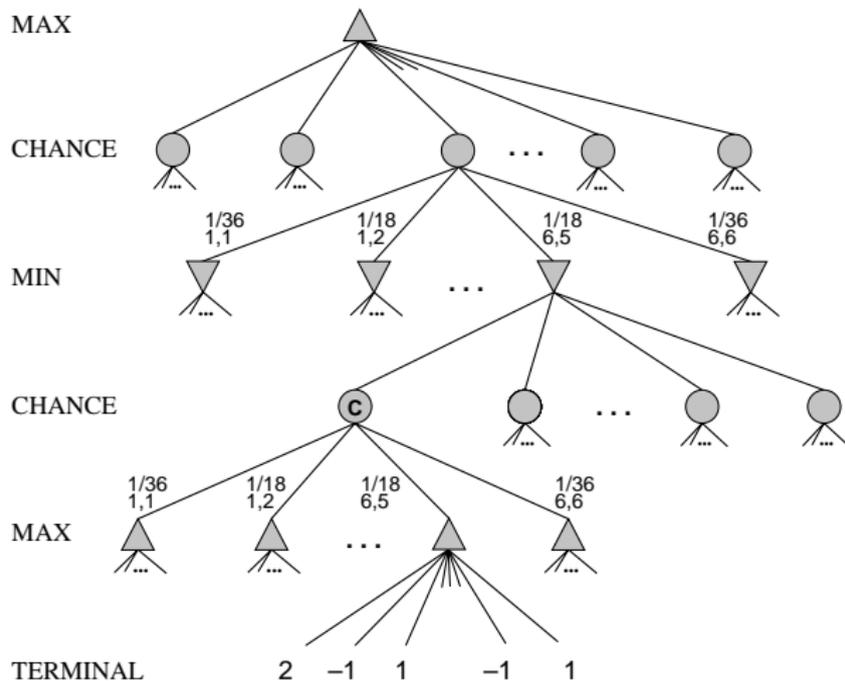
Games that Include an Element of Chance



White has just rolled 6-5 and has 4 legal moves.

Game Tree for Backgammon

In addition to MIN- and MAX nodes, we need **chance nodes** (for the dice).



Calculation of the Expected Value

Utility function for chance nodes C over MAX:

d_i : possible dice rolls

$P(d_i)$: probability of obtaining that roll

$S(C, d_i)$: attainable positions from C with roll d_i

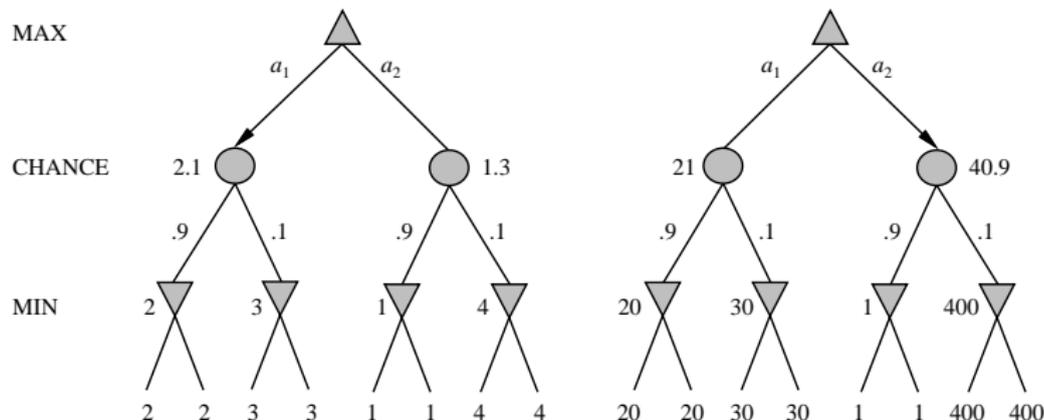
UTILITY(s): Evaluation of s

$$\text{EXPECTIMAX}(C) = \sum_i P(d_i) \max_{s \in S(C, d_i)} (\text{UTILITY}(s))$$

EXPECTIMIN likewise

Problems

- Order-preserving transformations on evaluation values change the best move:



- Search costs increase:** Instead of $O(b^d)$, we get $O((b \times n)^d)$, where n is the number of possible dice outcomes.

→ In Backgammon ($n = 21$, $b = 20$, can be 4000) the maximum for d is 2.

- Recently **card games** such as bridge and poker have been addressed as well
- One approach: simulate play with open cards and then average over all possible plays (or make a Monte Carlo simulation) using minimax (perhaps modified)
- Pick the move with the best expected result (usually all moves will lead to a loss, but some give better results)
 - **Averaging over clairvoyancy**
- Although “incorrect”, appears to give reasonable results

Checkers, draughts (by international rules): A program called *CHINOOK* is the official world champion in man-computer competition (acknowledges by ACF and EDA) and the highest-rated player:

| | |
|----------------|--------------------|
| CHINOOK: 2712 | Ron King: 2632 |
| Asa Long: 2631 | Don Lafferty: 2625 |

Backgammon: The *BKG* program defeated the official world champion in 1980. A newer program TD-Gammon is among the top 3 players.

Othello: Very good, even on normal computers. In 1997, the *Logistello* program defeated the human world champion.

Go: The best programs (Zen, Mogo, CrazyStone) using Monte Carlo techniques (UCT) are rated as good as strong amateurs (1kyu/1dan) on the Internet Go servers. However, it's usually easy to adapt to the weaknesses of these programs.

Chess as “Drosophila” of AI research.

- A limited number of rules produces an unlimited number of courses of play. In a game of 40 moves, there are 1.5×10^{128} possible courses of play.
- Victory comes through logic, intuition, creativity, and previous knowledge.
- Only special chess intelligence, no “general knowledge”

Chess (2)

In 1997, world chess master G. Kasparow was beaten by a computer in a match of 6 games.

Deep Blue (IBM Thomas J. Watson Research Center)

- Special hardware (32 processors with 8 chips, 2 Mi. calculations per second)
- Heuristic search
- Case-based reasoning and learning techniques
 - 1996 Knowledge based on 600,000 chess games
 - 1997 Knowledge based on 2 million chess games
 - Training through grand masters
- Duel between the “machine-like human Kasparow vs. the human machine Deep Blue.”

Nowadays, ordinary PC hardware is enough ...

| Name | Strength (ELO) |
|--------------------|-----------------------|
| Rybka 2.3.1 | 2962 |
| G. Kasperow | 2828 |
| V. Anand | 2758 |
| A. Karpow | 2710 |
| Deep Blue | 2680 |

But note that the machine ELO points are not strictly comparable to human ELO points ...

The Reasons for Success . . .

- Alpha-Beta-Search
- . . . with dynamic decision-making for uncertain positions
- Good (but usually simple) evaluation functions
- Large databases of opening moves
- Very large game termination databases (for checkers, all 10-piece situations)
- For Go, Monte-Carlo techniques proved to be successful!
- And very fast and parallel processors!

Summary

- A **game** can be defined by the **initial state**, the **operators** (legal moves), a **terminal test** and a **utility function** (outcome of the game).
- In two-player board games, the **minimax algorithm** can determine the best move by enumerating the entire game tree.
- The **alpha-beta algorithm** produces the same result but is more efficient because it prunes away irrelevant branches.
- Usually, it is not feasible to construct the complete game tree, so the utility of some states must be determined by an **evaluation function**.
- **Games of chance** can be handled by an **extension of the alpha-beta algorithm**.