

# Foundations of Artificial Intelligence

## 41. Board Games: Minimax Search and Evaluation Functions

Malte Helmert and Thomas Keller

University of Basel

May 11, 2020

# Board Games: Overview

## chapter overview:

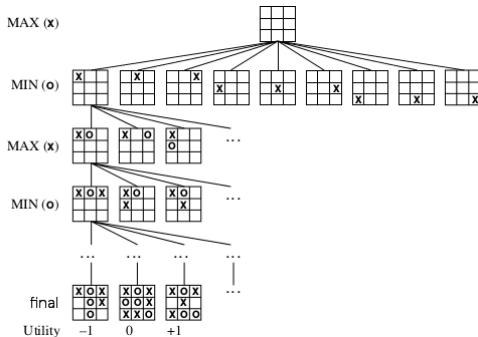
- 40. Introduction and State of the Art
- 41. Minimax Search and Evaluation Functions
- 42. Alpha-Beta Search
- 43. Monte-Carlo Tree Search: Introduction
- 44. Monte-Carlo Tree Search: Advanced Topics
- 45. AlphaGo and Outlook

# Minimax Search

# Terminology for Two-Player Games

- **Players** are traditionally called **MAX** and **MIN**.
- Our objective is to compute moves for MAX (MIN is the opponent).
- MAX tries to **maximize** its utility (given by the utility function  $u$ ) in the reached terminal position.
- MIN tries to **minimize**  $u$  (which in turn maximizes MINs utility).

# Example: Tic-Tac-Toe

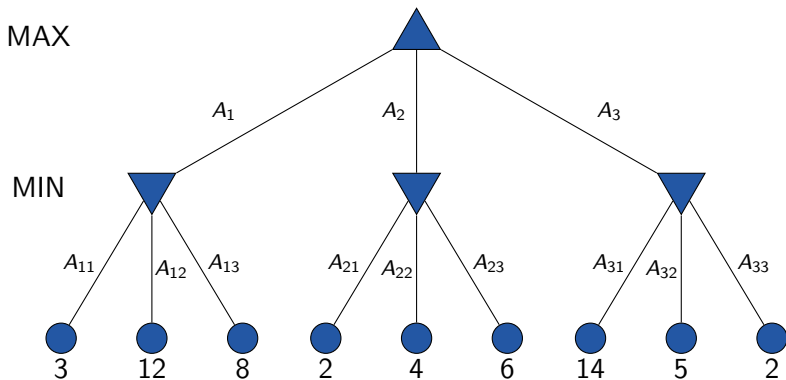


- **game tree** with player's turn (MAX/MIN) marked on the left
- last row: **terminal positions** with **utility**
- **size of game tree?**

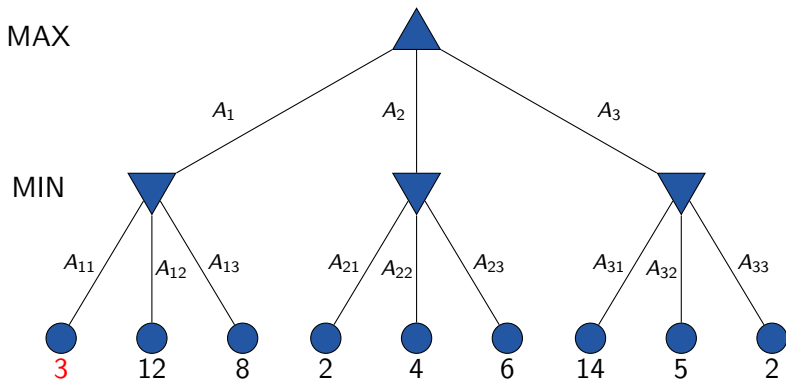
# Minimax: Computation

1. **depth-first search** through game tree
2. Apply utility function in terminal position.
3. Compute utility value of inner nodes from below to above through the tree:
  - MIN's turn: utility is **minimum** of utility values of children
  - MAX's turn: utility is **maximum** of utility values of children
4. move selection for MAX in root:  
choose a move that maximizes the computed utility value (**minimax decision**)

# Minimax: Example

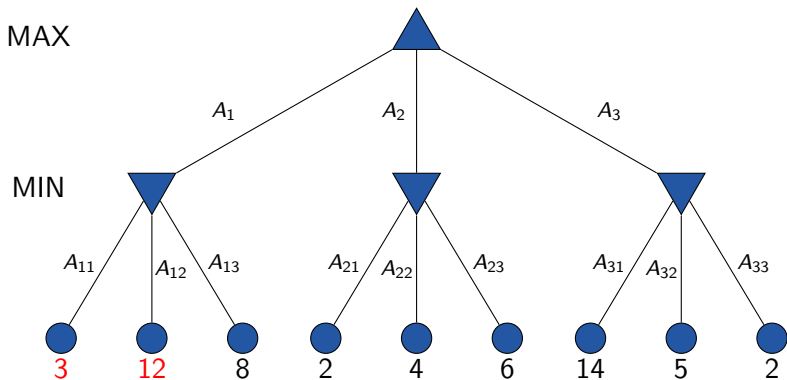


# Minimax: Example

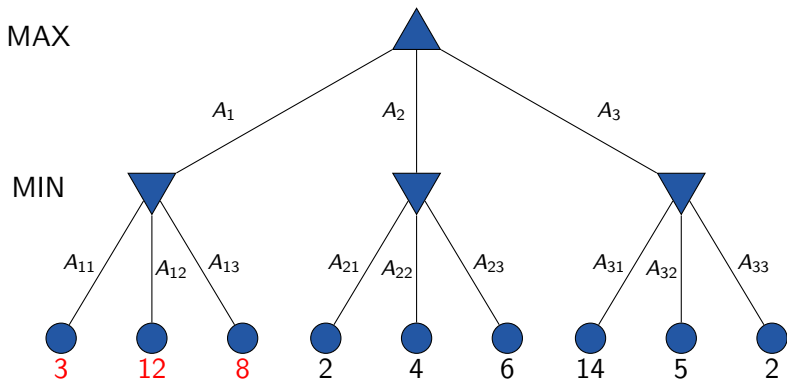




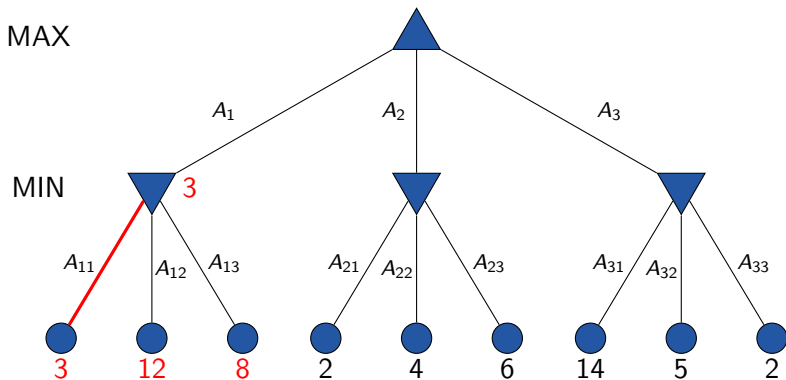
# Minimax: Example



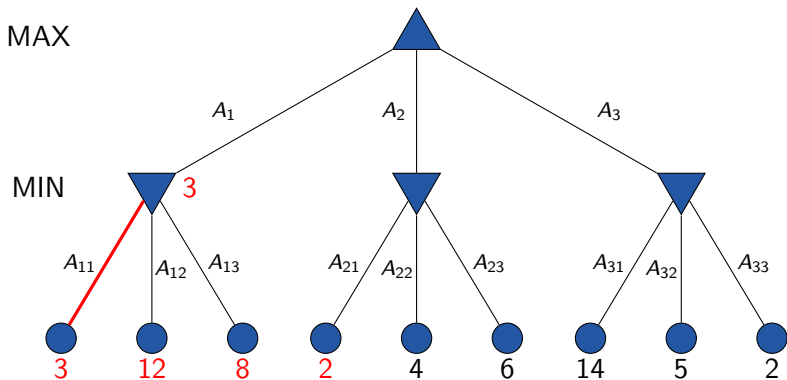
# Minimax: Example



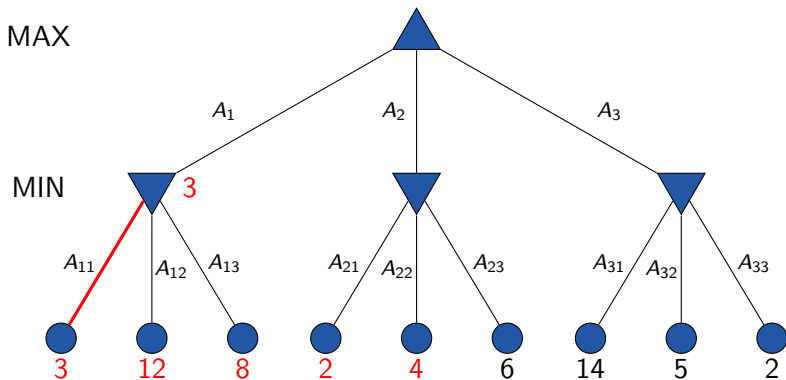
# Minimax: Example



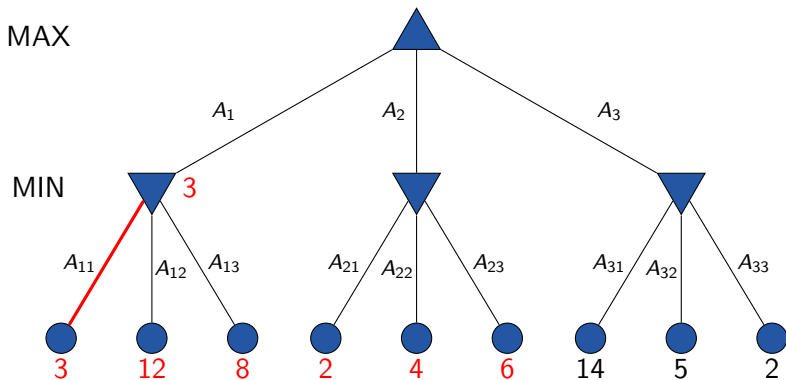
# Minimax: Example



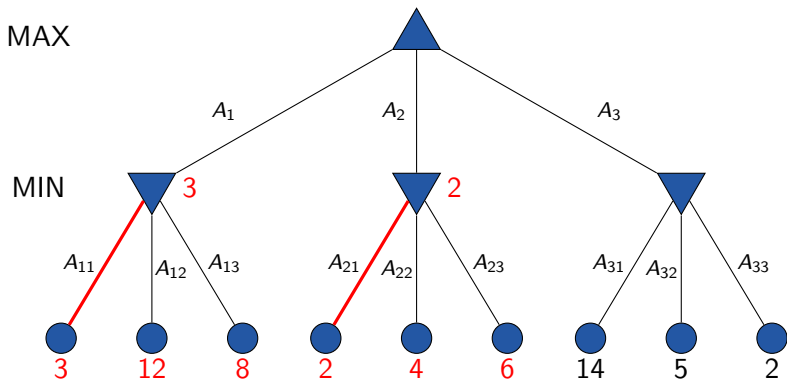
# Minimax: Example



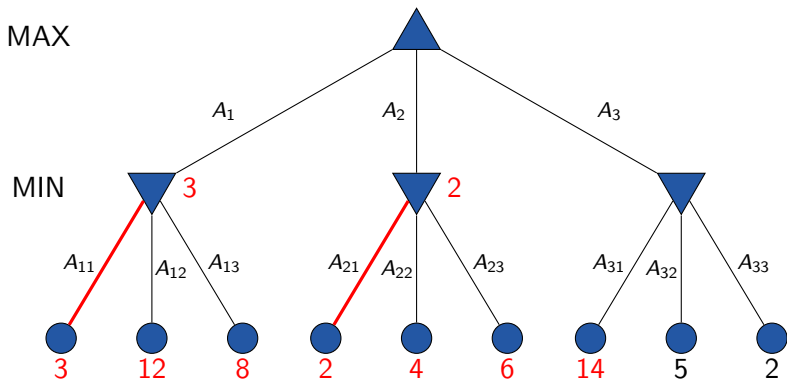
# Minimax: Example



## Minimax: Example

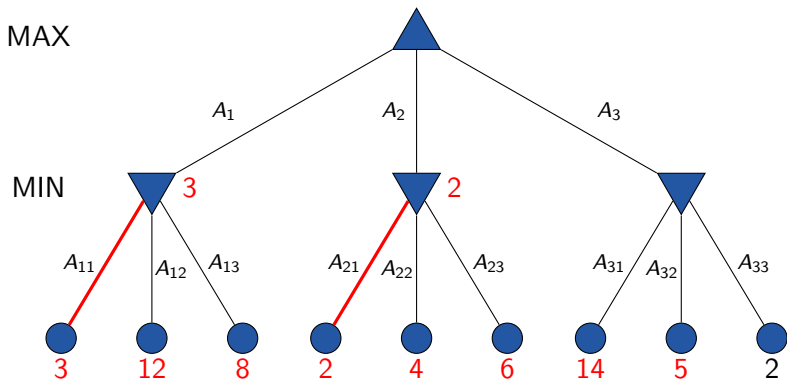


# Minimax: Example

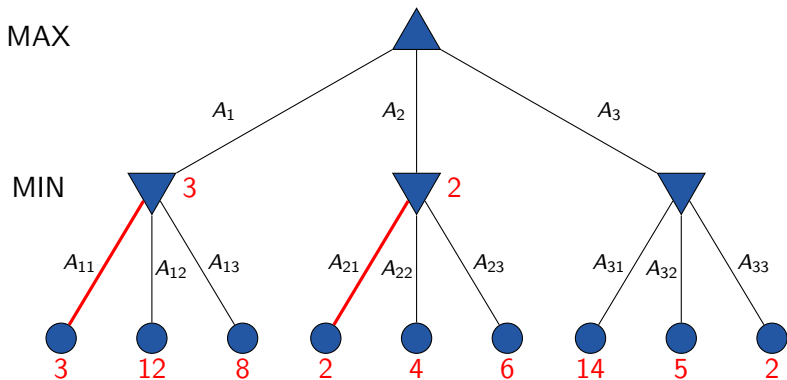




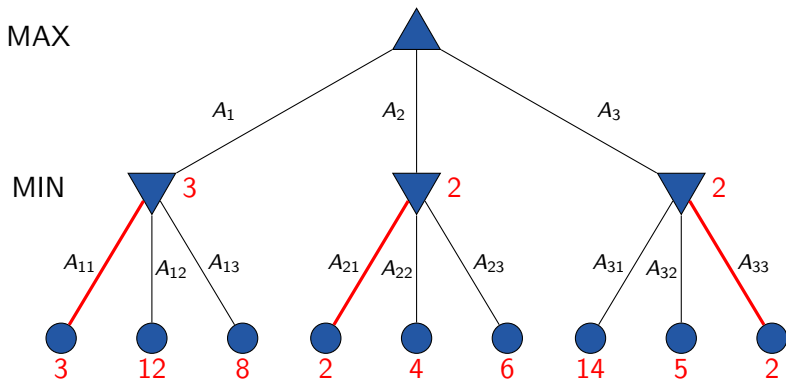
# Minimax: Example



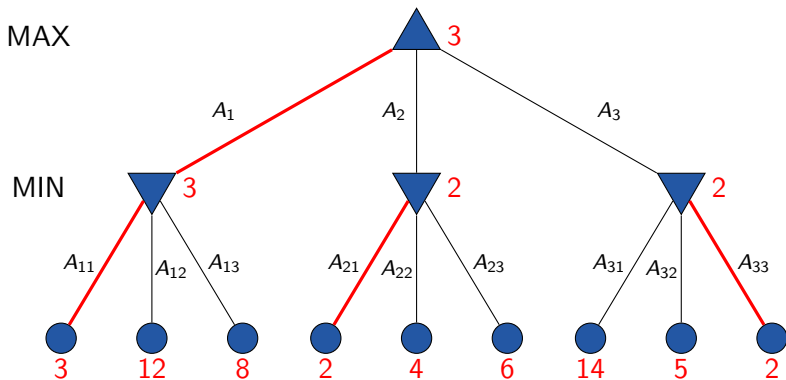
# Minimax: Example



# Minimax: Example



## Minimax: Example



# Minimax: Discussion

- **Minimax** is the simplest (decent) search algorithm for games
- Yields optimal strategy\* (in the game theoretic sense, i.e., under the assumption that the opponent plays perfectly), but is too time consuming for complex games.
- We obtain **at least** the utility value computed for the root, no matter how the opponent plays.
- In case the opponent plays perfectly, we obtain **exactly** that value.

(\* ) for games where no cycles occur; otherwise things get more complicated (because the tree will have infinite size in this case).

# Minimax: Pseudo-Code

```
function minimax(p)
```

```
if p is terminal position:
```

```
    return  $\langle u(p), \text{none} \rangle$ 
```

```
best_move := none
```

```
if player(p) = MAX:
```

```
    v :=  $-\infty$ 
```

```
else:
```

```
    v :=  $\infty$ 
```

```
for each  $\langle \text{move}, p' \rangle \in \text{succ}(p)$ :
```

```
     $\langle v', \text{best\_move}' \rangle := \text{minimax}(p')$ 
```

```
    if (player(p) = MAX and  $v' > v$ ) or
```

```
        (player(p) = MIN and  $v' < v$ ):
```

```
        v := v'
```

```
        best_move := move
```

```
return  $\langle v, \text{best\_move} \rangle$ 
```

# Minimax

What if the size of the game tree is too big for minimax?  
↪ approximation by **evaluation function**

# Evaluation Functions



# Evaluation Functions

- **problem**: game tree too big
- **idea**: search only up to certain depth
- depth reached: **estimate** the utility according to **heuristic criteria** (as if terminal position had been reached)

## Example (evaluation function in chess)

- **material**: pawn 1, knight 3, bishop 3, rook 5, queen 9  
positive sign for pieces of MAX, negative sign for MIN
- **pawn structure, mobility, ...**

rule of thumb: advantage of 3 points  $\rightsquigarrow$  clear winning position

## Accurate evaluation functions are crucial!

- High values should relate to high “winning chances” in order to make the overall approach work.
- At the same time, the evaluation should be efficiently computable in order to be able to search deeply.

# Linear Evaluation Functions

Usually **weighted linear functions** are applied:

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

where  $w_i$  are **weights**, and  $f_i$  are **features**.

- assumes that feature contributions are mutually **independent** (usually wrong but acceptable assumption)
- allows for efficient **incremental computation** if most features are unaffected by most moves
- Weights can be learned automatically.
- Features are (usually) provided by human experts.

# How Deep Shall We Search?

- **objective:** search as deeply as possible within a given time
- **problem:** search time difficult to predict
- **solution: iterative deepening**
  - sequence of searches of increasing depth
  - time expires: return result of previously finished search

# How Deep Shall We Search?

- **objective:** search as deeply as possible within a given time
- **problem:** search time difficult to predict
- **solution: iterative deepening**
  - sequence of searches of increasing depth
  - time expires: return result of previously finished search
- **refinement:** search depth not uniform, but deeper in “turbulent” positions (i.e., with strong fluctuations of the evaluation function)  $\rightsquigarrow$  **quiescence search**
  - **example chess:** deepen the search if exchange of pieces has started, but not yet finished

# Summary

# Summary

- **Minimax** is a tree search algorithm that plays perfectly (in the game-theoretic sense), but its complexity is  $O(b^d)$  (branching factor  $b$ , search depth  $d$ ).
- In practice, the search depth must be limited  
     $\rightsquigarrow$  apply **evaluation functions**  
    (usually linear combinations of features).