Samos Summer School on Combinatorics and Computer science

Combinatorial Game Theory

Part 2 : Partizan games and tree-search algorithms

UNIVERSITY OF THE AEGEAN

ERUA
European Reform University Alliance

Partizan Games

- A **partizan combinatorial game** (PCG) is a game where the two players have different sets of legal moves.
- **Example:** Partizan Subtraction Game
  - Player 1 can remove 1 or 2 tokens.
  - Player 2 can remove 1 or 3 tokens.
- **Grundy numbers** do not apply: they rely on the game being **impartial**, i.e., both players having the same moves.

Partizan Games

- A **partizan combinatorial game** (PCG) is a game where the two players have different sets of legal moves.
- **Example:** Partizan Subtraction Game
  - Player 1 can remove 1 or 2 tokens.
  - Player 2 can remove 1 or 3 tokens.
- **Grundy numbers** do not apply: they rely on the game being **impartial**, i.e., both players having the same moves.
- **Conway's Game Theory** provides a framework for analyzing partizan games by explicitly representing the moves available to each player:

$$G = \{G^L \mid G^R\}.$$

and associate to such a set of moves a **value**, that might not even be a number !

  - However, it works quite well for simple games but becomes unusable for more complicated games. We need **computers** !
  - **Reference:** J. H. Conway, *On Numbers and Games*.

Minimax algorithm.

The Minimax principle

- Assume that our PGC is a **zero-sum game**, that is the gain of one player is exactly the loss of the other.
- We will associate to each position of the game a **score**, that should indicate which player has the advantage.
- The **maximizing player** tries to maximize its score, while the **minimizing player** responds with the move that minimizes the opponent's score.

The Minimax principle

- Assume that our PGC is a **zero-sum game**, that is the gain of one player is exactly the loss of the other.
- We will associate to each position of the game a **score**, that should indicate which player has the advantage.
- The **maximizing player** tries to maximize its score, while the **minimizing player** responds with the move that minimizes the opponent's score.
- Hence, the maximizing player chooses the move with the **best worst-case outcome**:

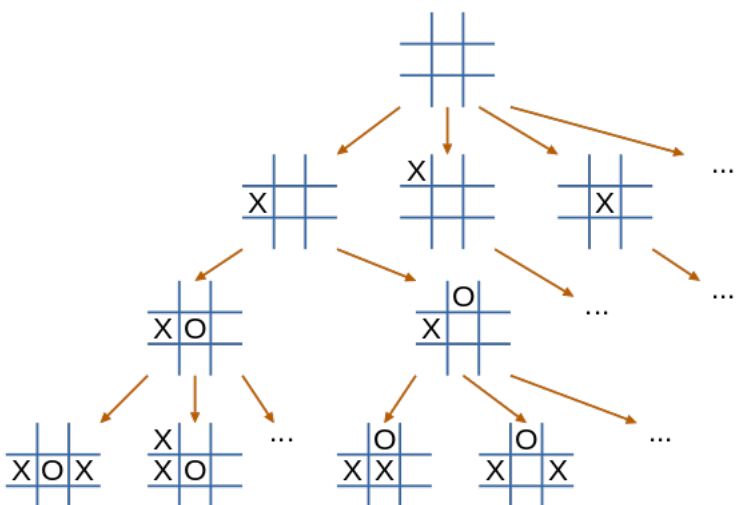
« max moves min opponent moves score »

- This principle can be implemented using a **tree search algorithm**: explore possible moves and recursively propagate the resulting scores back up the game tree.

The minimax algorithm

- The minimax algorithm is a **tree-search algorithm**. The idea is, from a given position of the game, to explore the tree of all possible next positions up to a certain point in order to determine the best move.

- **Example:** TicTacToe.



Historical overview on minimax

- AI based on minimax algorithm and its extensions have obtained a lot of powerful results for two-players games.
- **Checkers:**
  - In 1950, Arthur Samuel creates a program to play Checkers on IBM 701. This program plays exactly using a minimax, and is the base idea of the pruning optimization.
  - In 1952, a genetic element of self-training was added; two instances of the program were playing against one another with repetitions. The evolved program was beating intermediate players.
  - In 1962, the program won a public match against Robert Nealy, a blind checkers master but not a world-class master.

Historical overview on minimax

- **Chess:**
  - In 1950, Claude Shannon published a groundbreaking paper, which first put forth the idea of a function for evaluating the efficacy of a particular move and a "minimax" algorithm.
  - In the 80s, people working at IBM started to work on a computer program to play Chess, first called *Deep Thought*. The program is then renamed *Deep Blue* in 1993. In 1996, this computer loses against the world champion Gary Kasparov.
  - In 1997, *Deep Blue* finally manages to beat the world champion Garry Kasparov, 3.5-2.5.
  - The program is based on a **parallelized incremental implementation of an alpha-beta pruning**.

Problem: Spatial complexity

- For the game of TicTacToe, the number of reachable positions from the start is less than $3^9 \sim 20000$. Thus, exploring the whole game tree is still possible.
- Indeed, there is at most 9 following positions from a given position, and its box can have 3 states : X, O or empty. If we remove illegal positions: ~ 5000. If we consider rotations and symmetries: 765.

Problem: Spatial complexity

- For the game of TicTacToe, the number of reachable positions from the start is less than $3^9 \sim 20000$. Thus, exploring the whole game tree is still possible.
- Indeed, there is at most 9 following positions from a given position, and its box can have 3 states : X, O or empty. If we remove illegal positions: ~ 5000. If we consider rotations and symmetries: 765.
- However, for more complicated games, it is impossible to explore everything !

Game	Board size	State-space complexity	Game-tree complexity
Tic-tac-toe	9	10^3	10^5
Connect 4	42	10^{13}	10^{21}
English checkers	32	10^{18}	10^{31}
Hex	121	10^{57}	10^{98}
Chess	64	10^{47}	10^{120}
Connect 6	361	10^{70}	10^{140}
Backgammon	28	10^{20}	10^{144}
Go	361	10^{171}	10^{360}

- Therefore, we will only explore the tree up to a certain **depth**.

Evaluation of a position

- An important part of the algorithm is to assign to a position a **score**. This is done using an **evaluation function**, that is a function made to determine and quantify which player has the advantage.

- Ideally, we thus want a function f that takes a position $\mathtt{P}$ of the game, and returns a real number $f(\mathtt{P})$ such that:

$$\begin{cases} f(\mathtt{P}) > 0 & \text{if Player 1 is winning;} \\ f(\mathtt{P}) < 0 & \text{if Player 2 is winning.} \end{cases}$$

Evaluation of a position

- An important part of the algorithm is to assign to a position a **score**. This is done using an **evaluation function**, that is a function made to determine and quantify which player has the advantage.

- Ideally, we thus want a function f that takes a position $\mathtt{P}$ of the game, and returns a real number $f(\mathtt{P})$ such that:

$$\begin{cases} f(\mathtt{P}) > 0 & \text{if Player 1 is winning;} \\ f(\mathtt{P}) < 0 & \text{if Player 2 is winning.} \end{cases}$$

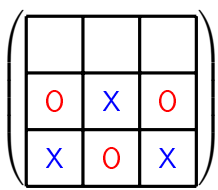
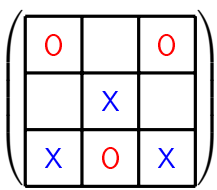
$$f\left(\begin{array}{|c|c|c|}\hline & & \\ \hline \mathtt{O} & \mathtt{X} & \mathtt{O} \\ \hline \mathtt{X} & \mathtt{O} & \mathtt{X} \\ \hline \end{array}\right) >> 0$$

Evaluation of a position

- An important part of the algorithm is to assign to a position a **score**. This is done using an **evaluation function**, that is a function made to determine and quantify which player has the advantage.

- Ideally, we thus want a function f that takes a position $\mathtt{P}$ of the game, and returns a real number $f(\mathtt{P})$ such that:

$$\begin{cases} f(\mathtt{P}) > 0 & \text{if Player 1 is winning;} \\ f(\mathtt{P}) < 0 & \text{if Player 2 is winning.} \end{cases}$$

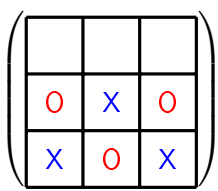
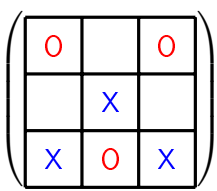
f  $>> 0$ $\qquad$ f  $=??$

Evaluation of a position

- An important part of the algorithm is to assign to a position a **score**. This is done using an **evaluation function**, that is a function made to determine and quantify which player has the advantage.

- Ideally, we thus want a function f that takes a position $\mathtt{P}$ of the game, and returns a real number $f(\mathtt{P})$ such that:

$$\begin{cases} f(\mathtt{P}) > 0 & \text{if Player 1 is winning;} \\ f(\mathtt{P}) < 0 & \text{if Player 2 is winning.} \end{cases}$$

f  $>> 0$ $\qquad$ f  $=??$

- **Remark:** In many cases, the evaluation function is built-up as a weighted sum of functions measuring the position of the game:

$$f(\mathtt{P}) = w_1 f_1(\mathtt{P}) + \cdots + w_k f_k(\mathtt{P})$$

and one can then try to learn which are the best coefficients.

Example: Chess evaluation

- For chess, there is an evaluation of a position P that is be quite good, given as follows:

$$\begin{aligned} f(P) = 200(\#wK - \#bK) + 9(\#wQ - \#bQ) + 5(\#wR - \#bR) \\ + 3(\#wB + \#wKn - \#bB - \#bKn) + (\#wP - \#bP) \\ - (\#wd - \#bd + \#ws - \#bs + \#wi - \#bi)/2 + (\#wm - \#bm)/10 \end{aligned}$$

where:

- $\#wd$ (resp. $\#bd$) is the number of doubled pawns for white (resp. black).
- $\#ws$ (resp. $\#bs$) is the number of stucked pawns for white (resp. black).
- $\#wi$ (resp. $\#bi$) is the number of isolated pawns for white (resp. black).
- $\#wm$ (resp. $\#bm$) is the number of legal moves for white (resp. black).

Example: Chess evaluation

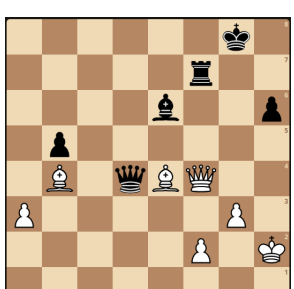
- For chess, there is an evaluation of a position P that is be quite good, given as follows:

$$\begin{aligned} f(P) = {} & 200(\#wK - \#bK) + 9(\#wQ - \#bQ) + 5(\#wR - \#bR) \\ & + 3(\#wB + \#wKn - \#bB - \#bKn) + (\#wP - \#bP) \\ & - (\#wd - \#bd + \#ws - \#bs + \#wi - \#bi)/2 + (\#wm - \#bm)/10 \end{aligned}$$

where:

  - $\#wd$ (resp. $\#bd$) is the number of doubled pawns for white (resp. black).
  - $\#ws$ (resp. $\#bs$) is the number of stucked pawns for white (resp. black).
  - $\#wi$ (resp. $\#bi$) is the number of isolated pawns for white (resp. black).
  - $\#wm$ (resp. $\#bm$) is the number of legal moves for white (resp. black).

- **Example:**



$$\begin{aligned} & 200(1-1) + 9(1-1) + 5(0-1) \\ & + 3(2+0-1-0) + (3-2) \\ & - (0-2+0-1+1-1)/2 \\ & + (37-46)/10 \simeq -1.4 \end{aligned}$$

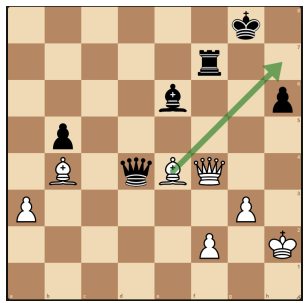
Problem: horizon effect

- Ideally, the closer we get to the ending positions of the game, the better the evaluation function is. However, since we generally do not explore the tree until the leafs, one can never be sure it is completely faithful.

Problem: horizon effect

- Ideally, the closer we get to the ending positions of the game, the better the evaluation function is. However, since we generally do not explore the tree until the leafs, one can never be sure it is completely faithful.

- **Horizon effect:** In the previous example, if W moves his remaining bishop in H7:

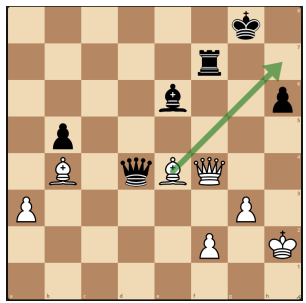


he will take advantage in the next round capturing the black queen.

Problem: horizon effect

- Ideally, the closer we get to the ending positions of the game, the better the evaluation function is. However, since we generally do not explore the tree until the leafs, one can never be sure it is completely faithful.

- **Horizon effect:** In the previous example, if W moves his remaining bishop in H7:



he will take advantage in the next round capturing the black queen.

- The idea is then to find the best compromise between the chosen depth for exploring the tree, and how good the evaluation function is.

Minimax algorithm: the idea

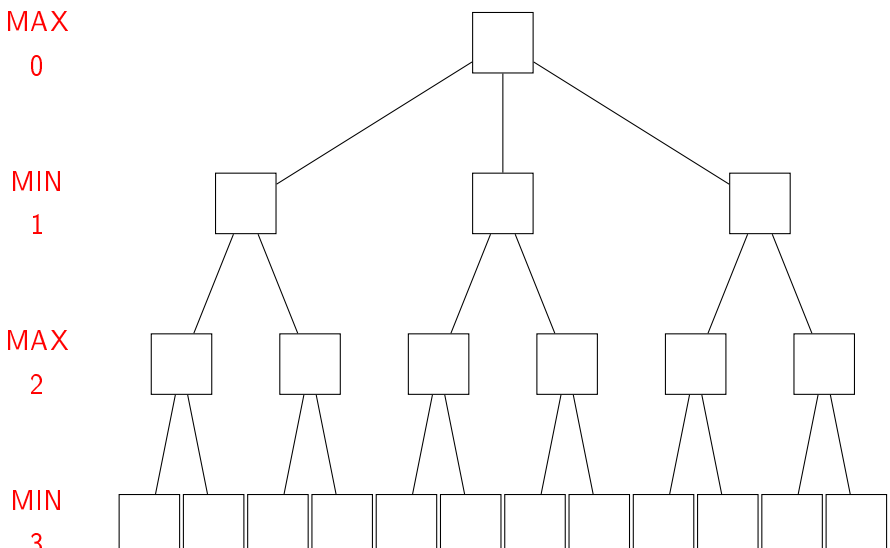
- The minimax algorithm is designed to chose the best reachable position for a given player at the chosen depth, and to play the sequence of moves to get in that position.

Minimax algorithm: the idea

- The minimax algorithm is designed to chose the best reachable position for a given player at the chosen depth, and to play the sequence of moves to get in that position.
- **Assumption:** Two-player games, Player 1 = MAX, Player 2 = MIN, both playing with the same strategy and both wanting to win.
- MAX will always tro to maximize its evaluation; while MIN will always try to minimize it.
- The idea is to explore the tree using a **Depth First Search (DFS)** : explore all the position until the leafs, evaluate them, and backpropagate the evaluations up to the top using the above strategy.

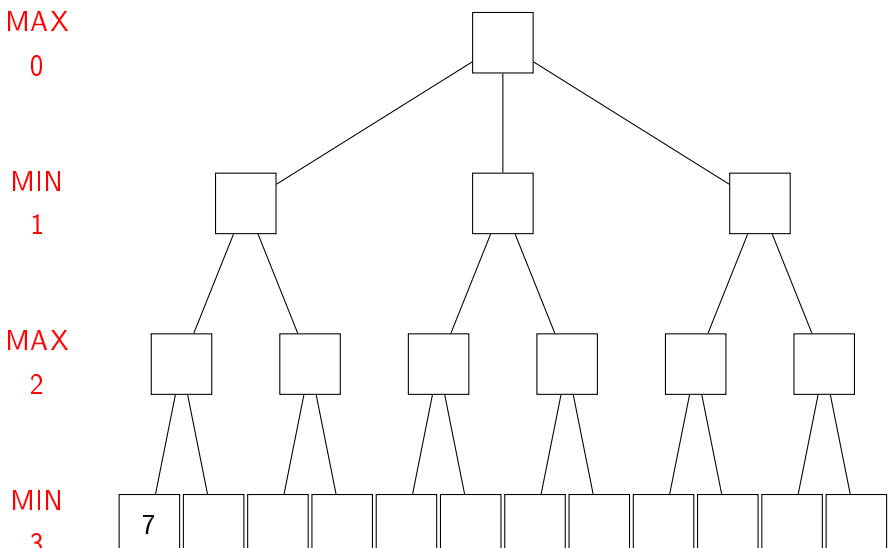
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



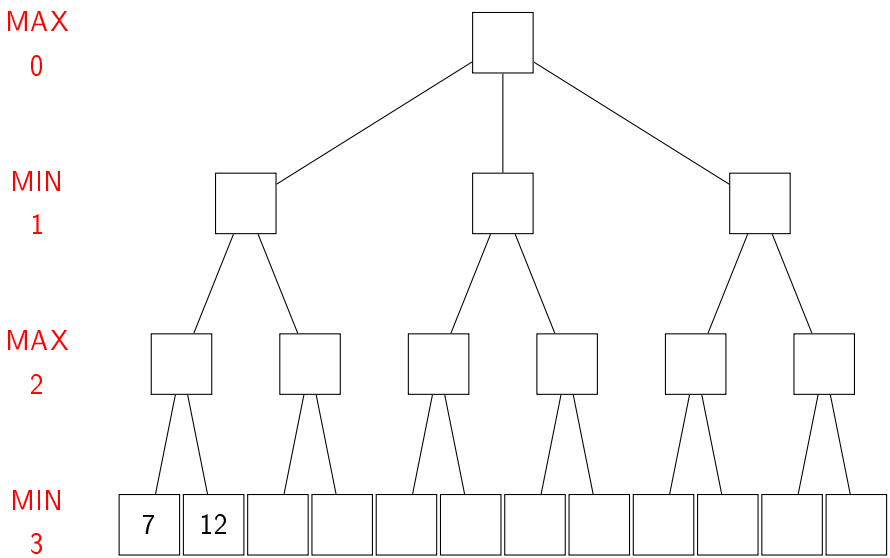
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



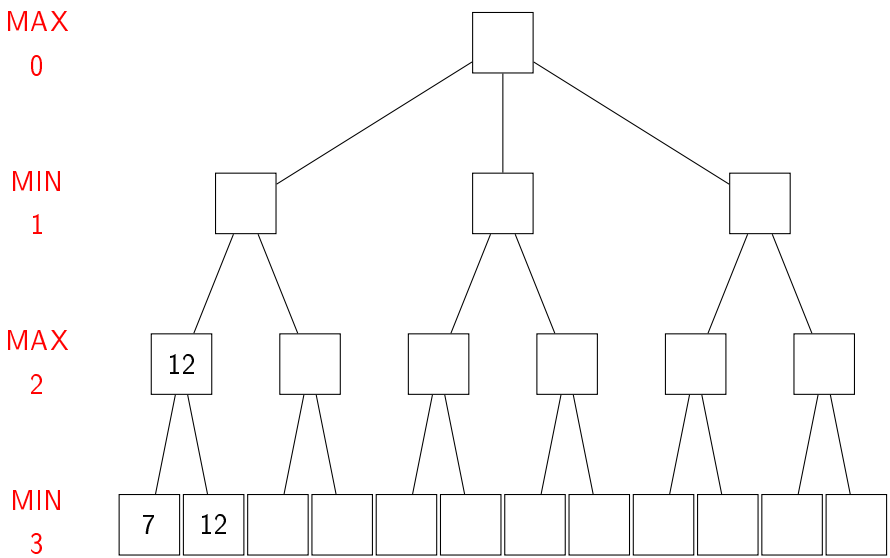
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



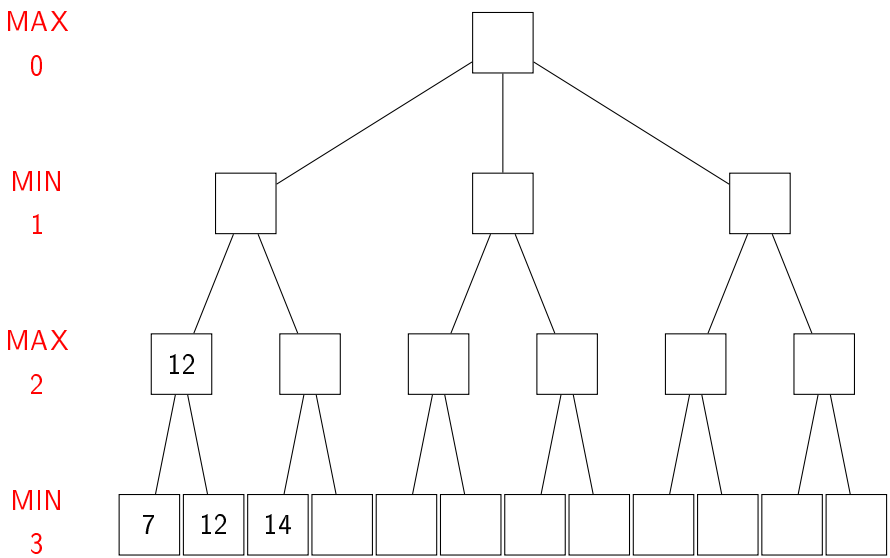
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



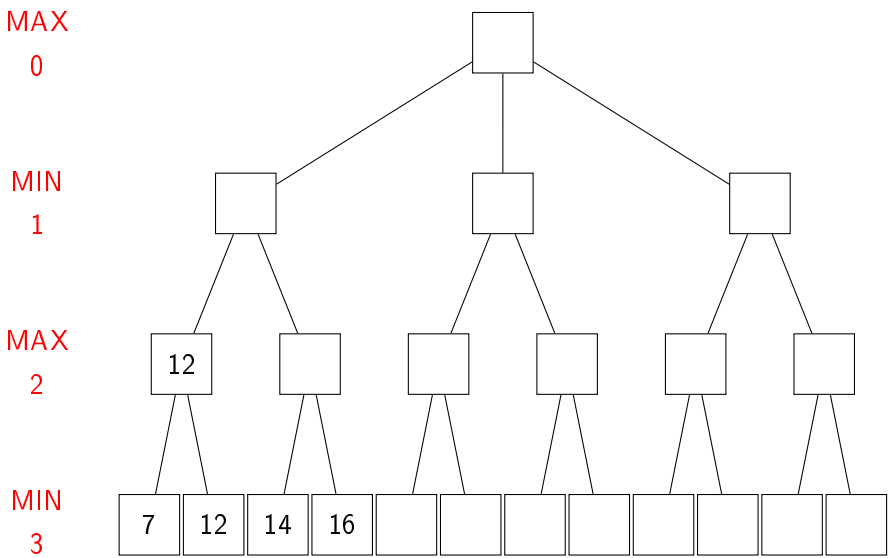
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



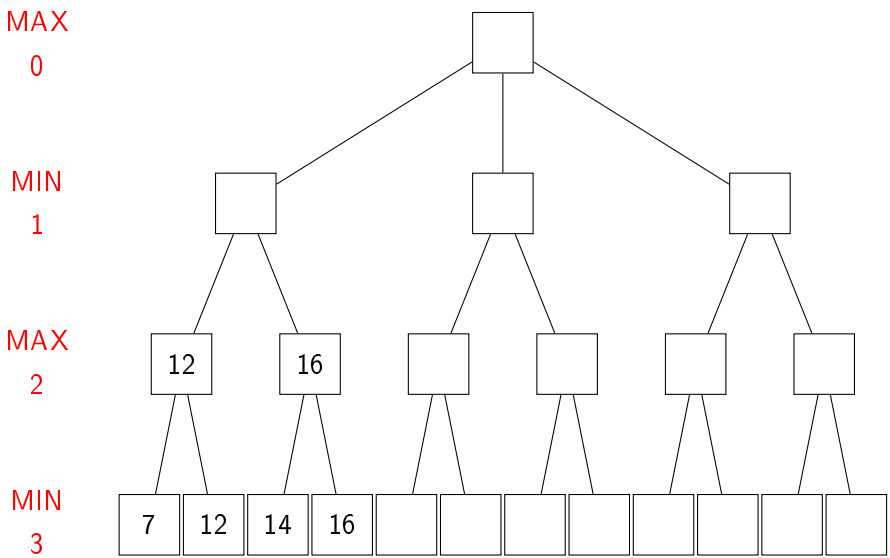
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



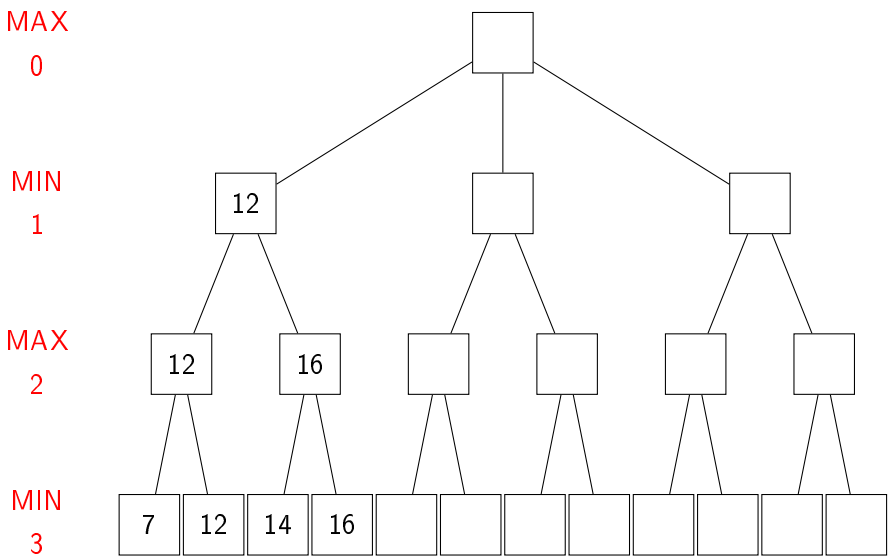
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



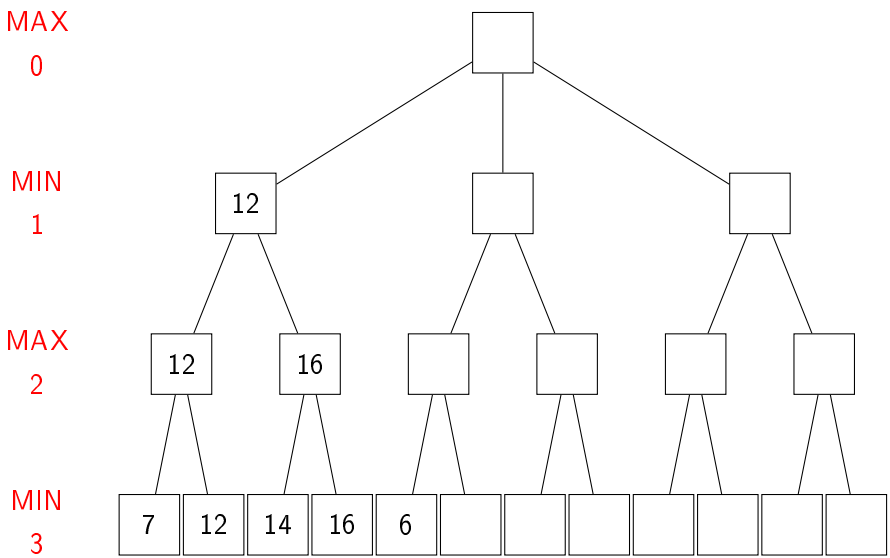
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



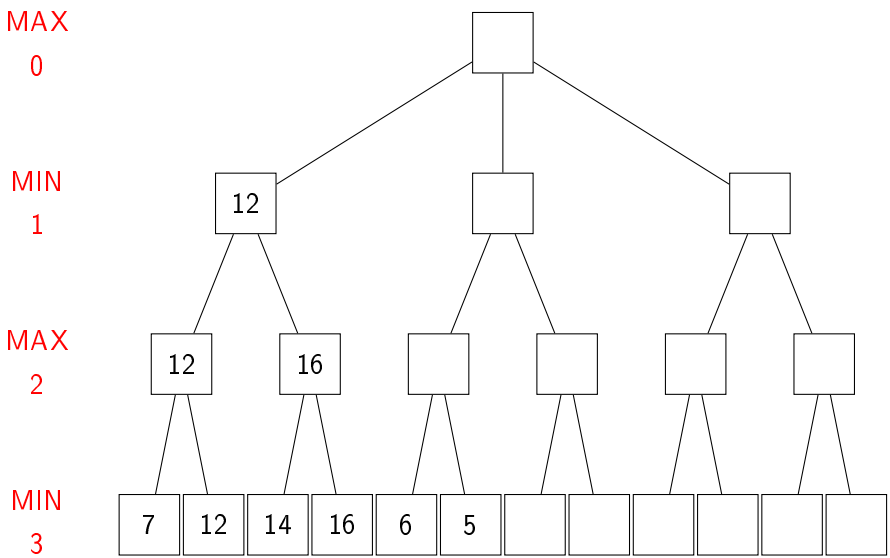
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



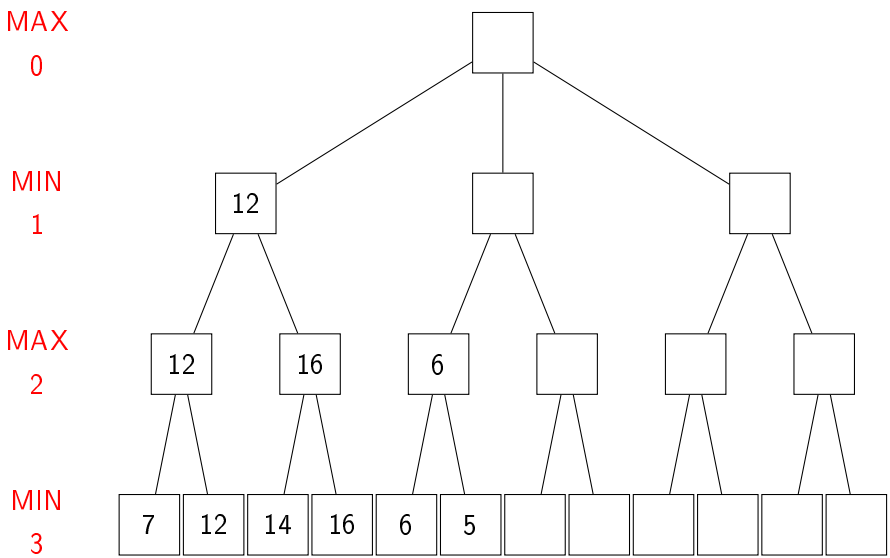
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



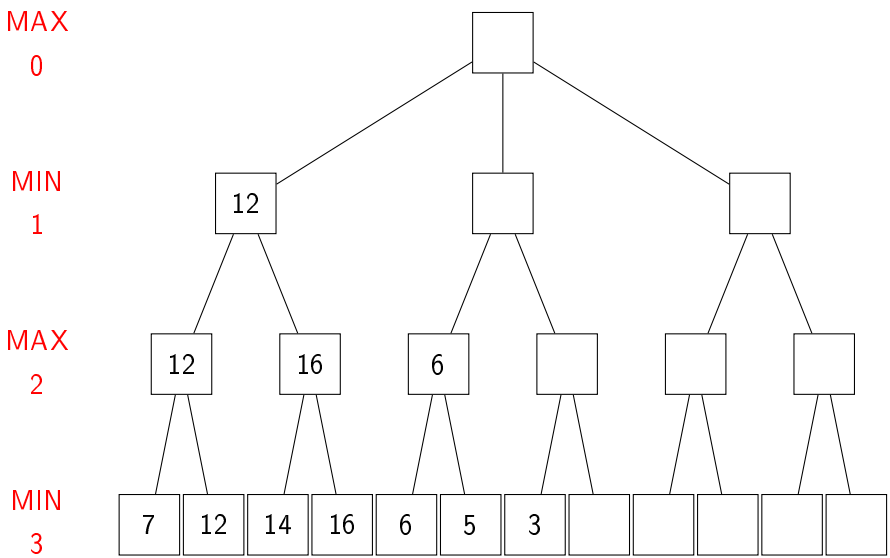
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



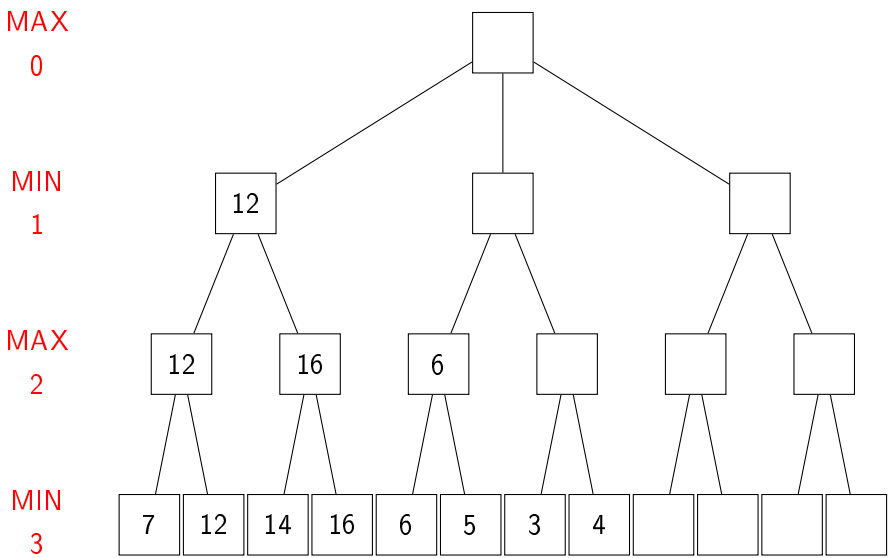
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



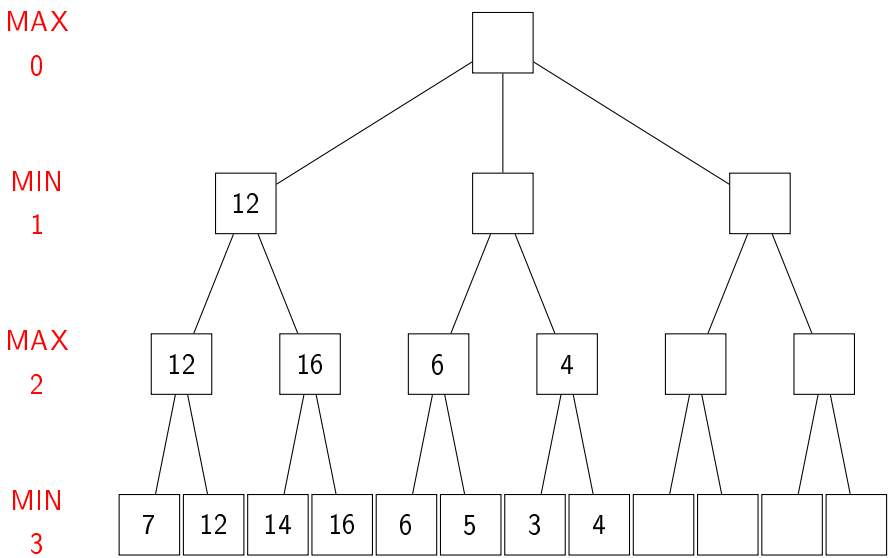
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



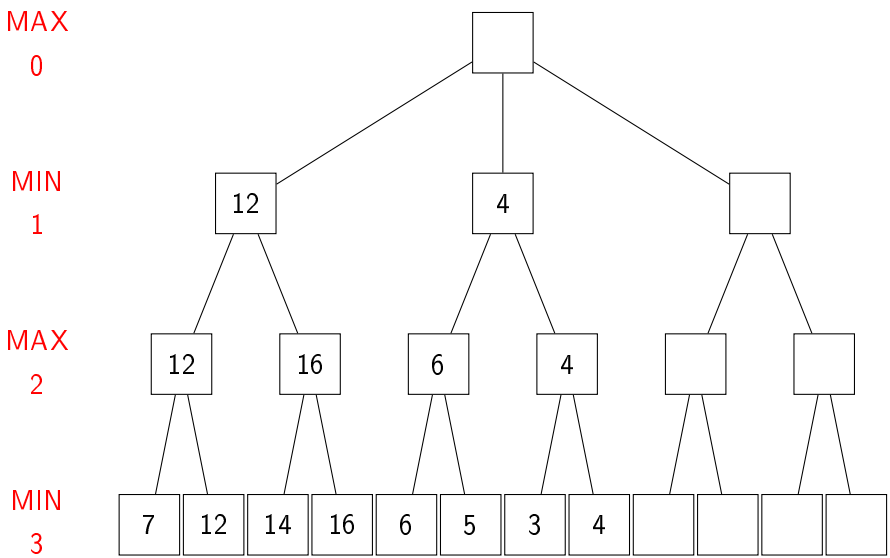
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



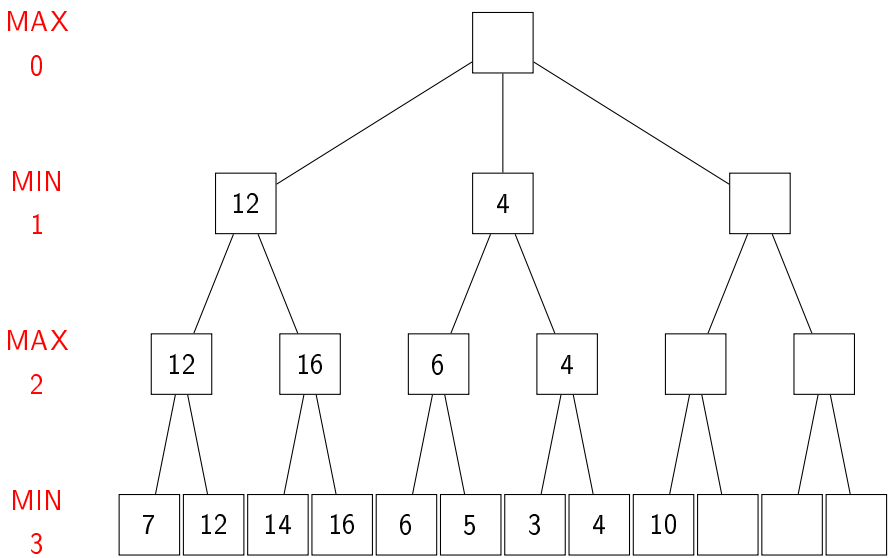
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



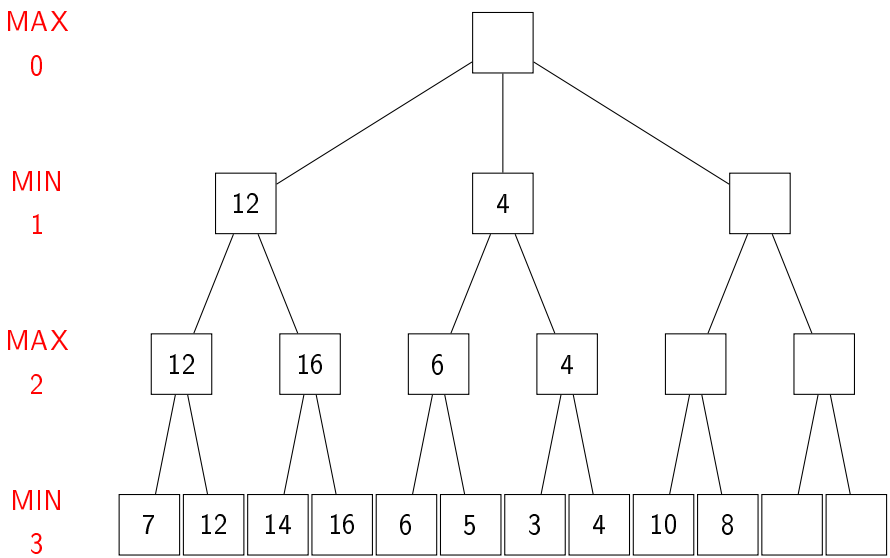
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



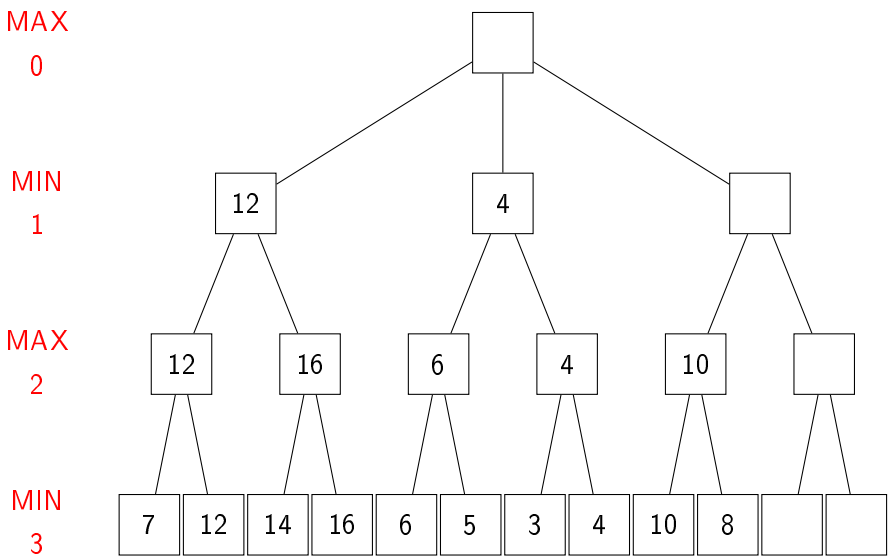
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



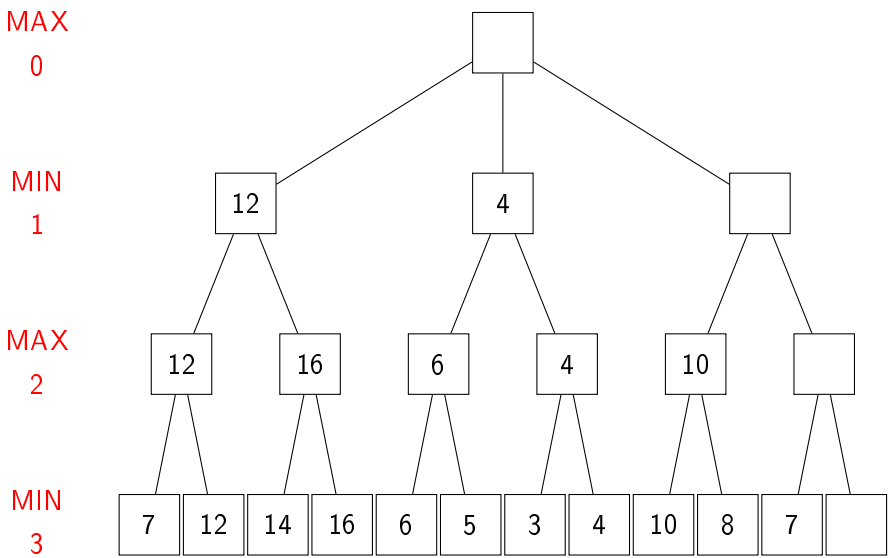
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



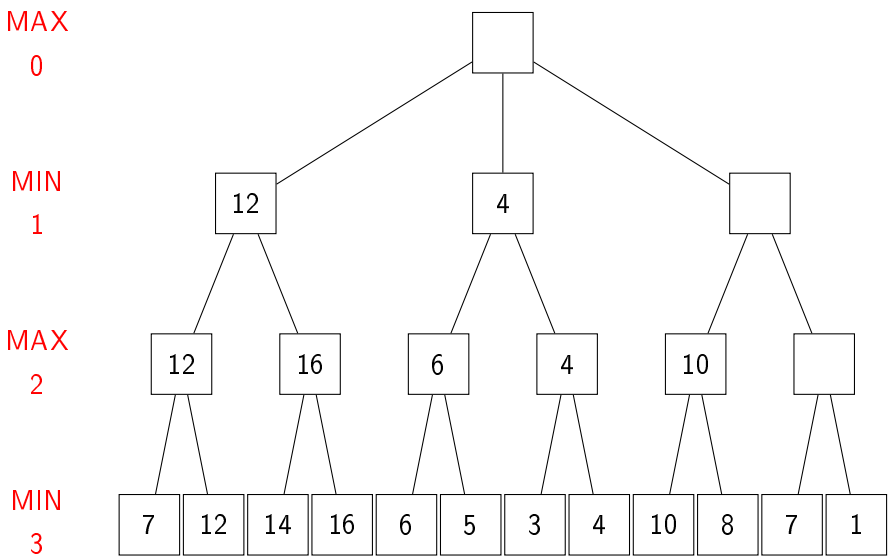
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



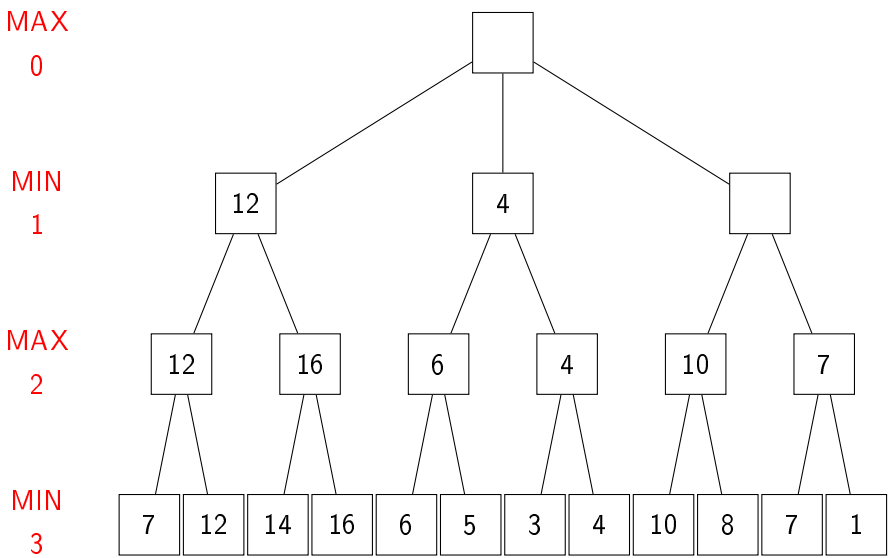
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



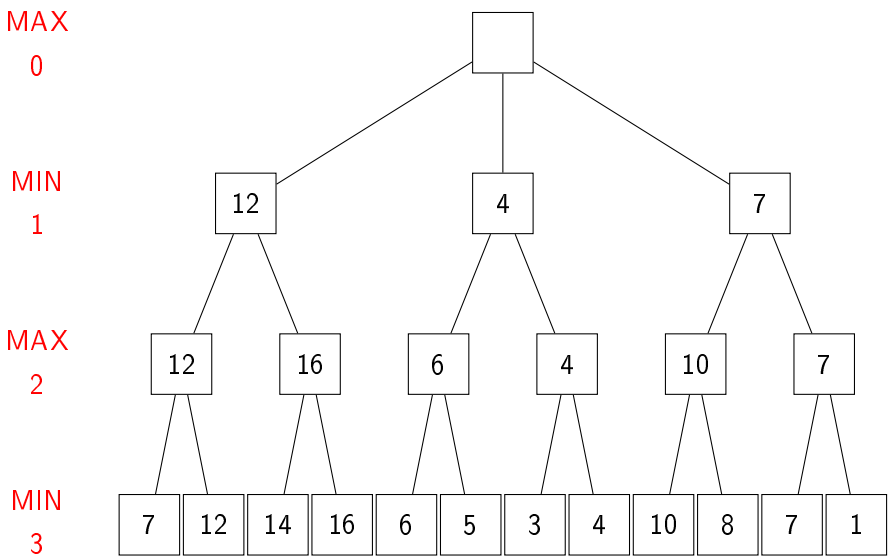
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



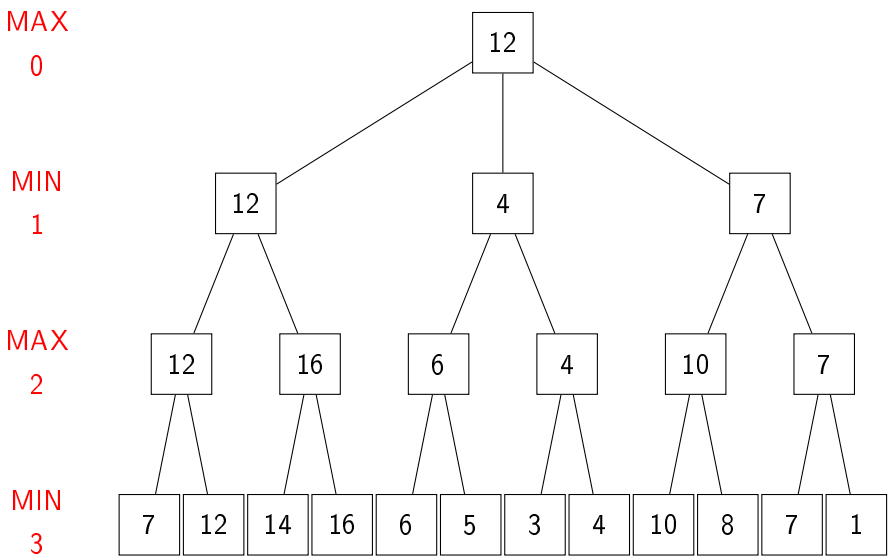
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



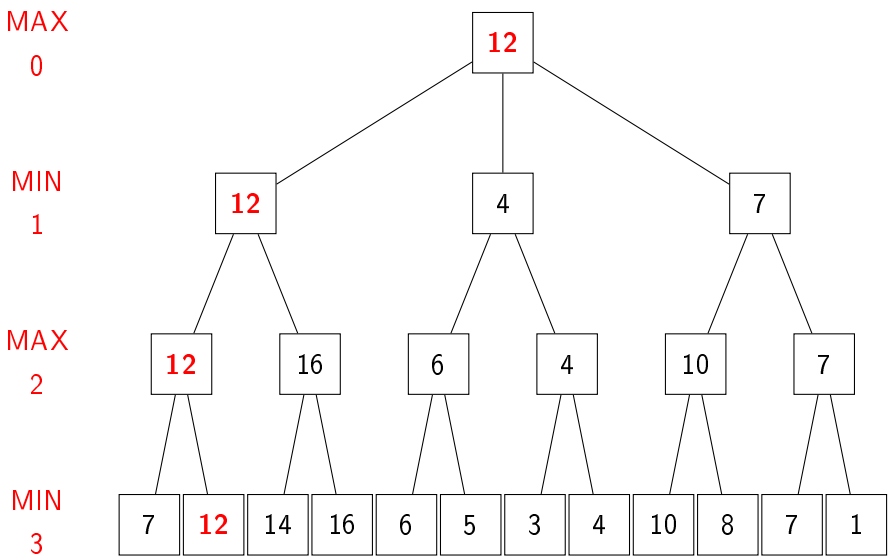
An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



An example of the minimax strategy

- **Example:** Assume the depth, is 3, and it is MAX's turn to play.



The minimax algorithm

- **Pseudo-code** for the minimax algorithm:

Data: A depth d, a position Pos of the game, a player p

Result: The best evaluation possible for player p.

Function MinMax(d,*Pos*,p):

```
if Pos is an ending position or d = 0 then
    return evaluation(Pos)
end
else
    Consider Pos_1,Pos_2,...,Pos_k all the legal position from Pos via a move of p.
    if p is MAX then
        return max_{1≤i≤k} MinMax(d − 1,Pos_i,MIN)
    end
    if p is MIN then
        return min_{1≤i≤k} MinMax(d − 1,Pos_i,MAX)
    end
end
```

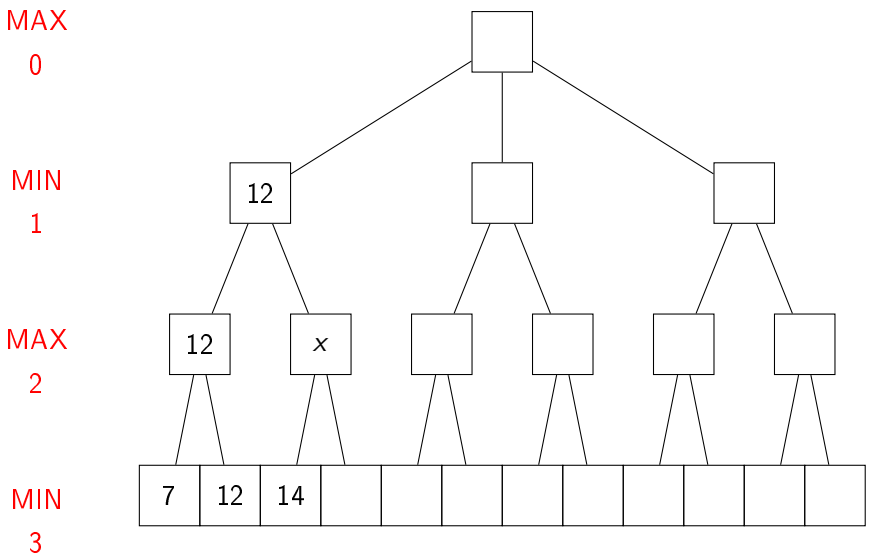
Drawbacks and improvements

- **Main drawbacks:**
  - With a depth d and an average number of b moves possible at each position, the space complexity is $O(b^d)$, that is huge ! Playing at high depths can be very costful.
  - Horizon effects are really hard to predict: $\texttt{MinMax}(n, \texttt{Pos}_n, p)$ can highlight an advantageous position while a position $\texttt{Pos}_{n+1}$ is really bad.

- **Improvement:** Pruning the game tree !
  - The idea is to try to reduce the number of positions one needs to explore in the game tree, so that we can go to higher depth !
  - This is done by an algorithm called the α-β pruning algorithm. It is based on minimax with two ways to reduce the browse of the game tree : α-cuts and β-cuts.

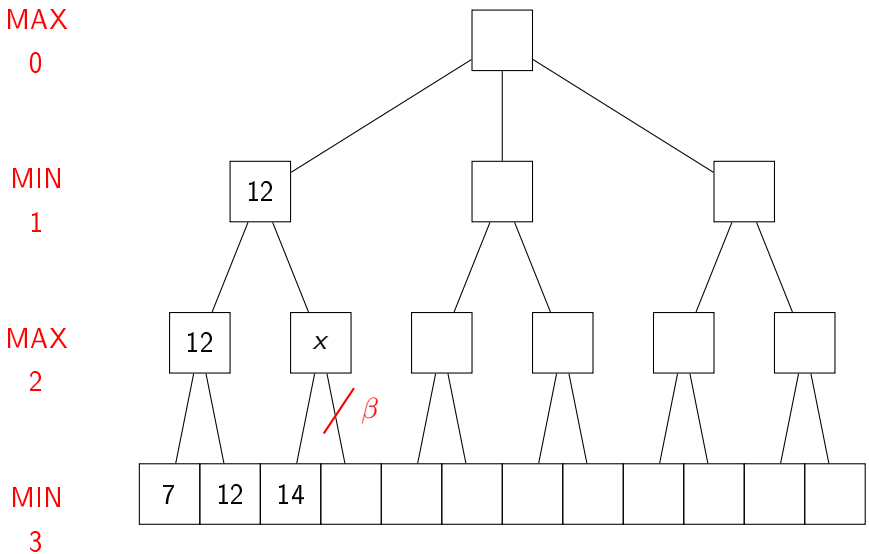
Alpha-beta pruning

- Illustration of the α-cuts and β-cuts: suppose we are in the situation



Alpha-beta pruning

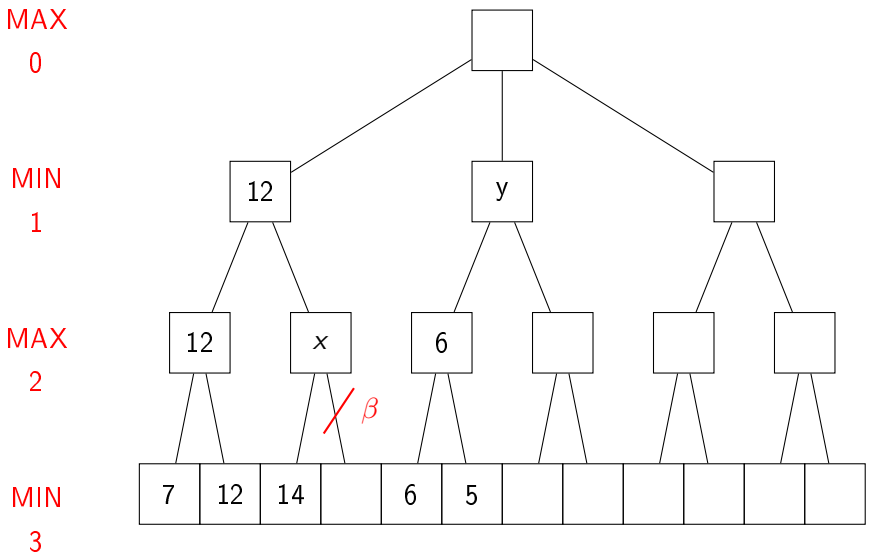
- Illustration of the α-cuts and β-cuts: suppose we are in the situation



- At depth 1, it is MAX turn to play, he will try to maximize its evaluation so $x \geq 14$. At depth 2, MIN will then have to choose between 12 and $x \geq 14$, so it will choose 12. No matter what happens on the right of 14, it will never be selected !

Alpha-beta pruning

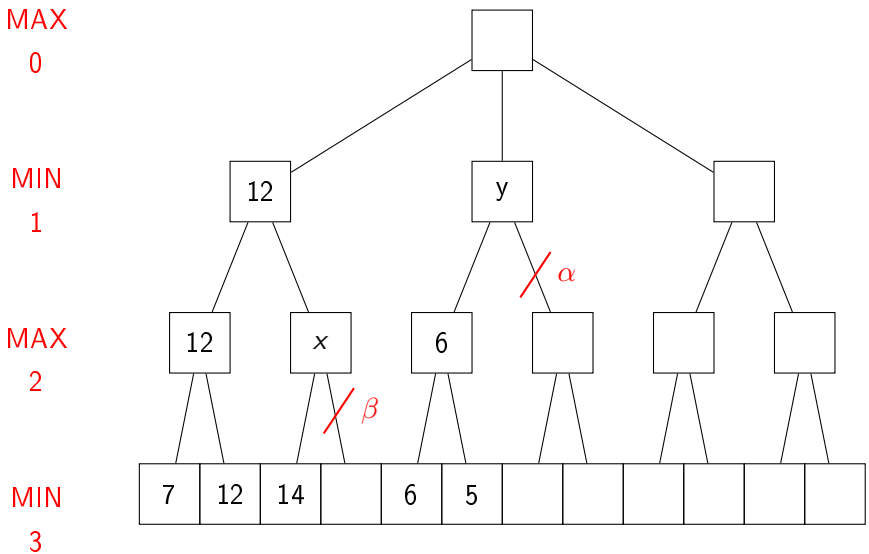
- Illustration of the α-cuts and β-cuts: suppose we are in the situation



- At depth 1, it is MAX turn to play, he will try to maximize its evaluation so $x \geq 14$. At depth 2, MIN will then have to choose between 12 and $x \geq 14$, so it will choose 12. No matter what happens on the right of 14, it will never be selected !

Alpha-beta pruning

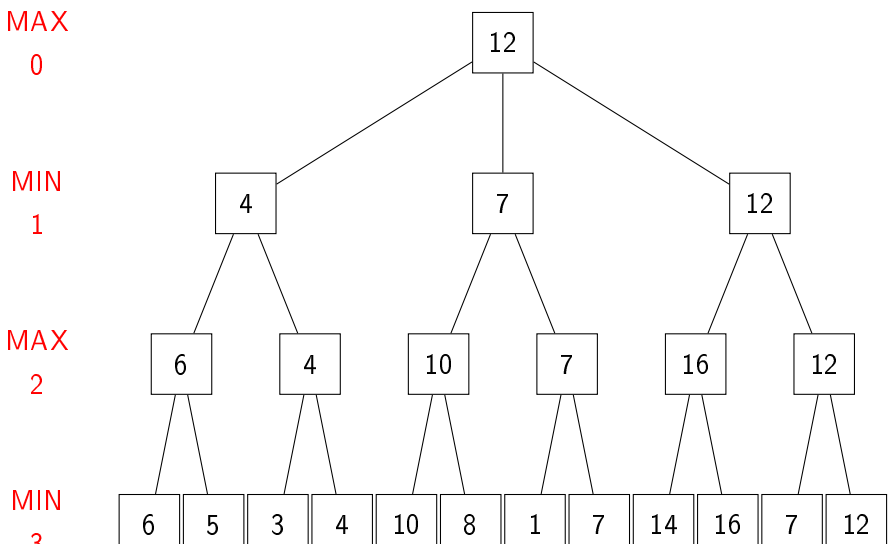
- Illustration of the α-cuts and β-cuts: suppose we are in the situation



- At depth 2, it is MIN turn to play, he will try to minimize its evaluation so $y \leq 6$. At depth 3, MAX will then have to chose between 12 and $y \leq 6$, so it will choose 12. No matter what happens on the right of y, it will never be selected !

Alpha-beta pruning

-  The order in which the positions are stored in the tree is important for the pruning ! For instance, the following configuration is the same as before but nodes are stored in a different order at each level, and no pruning is possible.



Alpha-beta algorithm

- **Pseudo-code** for the alpha-beta algorithm:

Data: A depth d, a position Pos of the game, a player p, two values α and β.
Result: The best evaluation possible for player p.

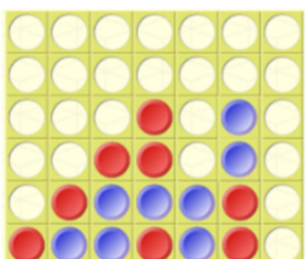
Function AlphaBeta(d,*Pos*,p,α,β):
- $\alpha = -\infty$, $\beta = +\infty$.
- **if** *Pos is an ending position or* $d = 0$ **then return** evaluation(Pos) ;
- **else**
  - Consider $\text{Pos}_1, \ldots, \text{Pos}_k$ all the legal positions from Pos via p-move.
  - **if** p *is MAX* **then**
    - $e \leftarrow$ AlphaBeta($d-1$,*Pos*$_1$,*MIN*,α,β)
    - **if** $e \geq \beta$ **then return** β ;
    - **else**
      - $\alpha \leftarrow \max(\alpha, e)$
      - Recalculate e for all other positions $\text{Pos}_2, \ldots, \text{Pos}_k$.
  - **if** p *is MIN* **then**
    - $e \leftarrow$ AlphaBeta($d-1$,*Pos*$_1$,*MAX*,α,β)
    - **if** $e \leq \alpha$ **then return** α ;
    - **else**
      - $\beta \leftarrow \min(\beta, e)$
      - Recalculate e for all other positions $\text{Pos}_2, \ldots, \text{Pos}_k$.

Heuristics for Connect 4

- Finding an optimal evaluation for Connect 4 is not so easy !
  - See Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game by Kang, Wang & Hu, 2019.

Heuristics for Connect 4

- Finding an optimal evaluation for Connect 4 is not so easy !
  - See Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game by Kang, Wang & Hu, 2019.
- **Heuristic 1.** Use static weights on the Connect 4 grid.

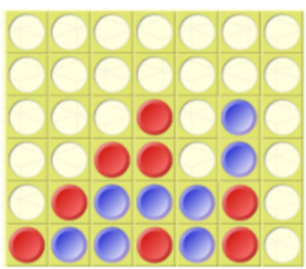


$$M := \begin{matrix} 3 & 4 & 5 & 7 & 5 & 4 & 3 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 3 & 4 & 5 & 7 & 5 & 4 & 3 \end{matrix}$$

$$\text{eval}(\text{Pos}) = \sum_{\text{P1 Tokens in } (i,j)} M_{i,j} - \sum_{\text{P2 Tokens in } (i,j)} M_{i,j}$$

Heuristics for Connect 4

- Finding an optimal evaluation for Connect 4 is not so easy !
  - See Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game by Kang, Wang & Hu, 2019.
- **Heuristic 1.** Use static weights on the Connect 4 grid.



$$M := \begin{matrix} 3 & 4 & 5 & 7 & 5 & 4 & 3 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 5 & 8 & 11 & 13 & 11 & 8 & 5 \\ 4 & 6 & 8 & 10 & 8 & 6 & 4 \\ 3 & 4 & 5 & 7 & 5 & 4 & 3 \end{matrix}$$

$$\text{eval}(\text{Pos}) = \sum_{\text{P1 Tokens in } (i,j)} M_{i,j} - \sum_{\text{P2 Tokens in } (i,j)} M_{i,j}$$

- On the above grid, the eval gives:

$$(3+6+11+13+13+7+4+6) - (4+5+6+8+10+7+8+8) = 7$$

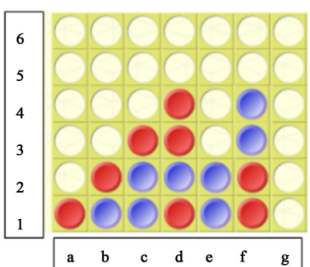
which gives a (small) advantage to Red, who won !

Heuristics for Connect 4

- **Problem:** Heuristic 1 doest take into account any configuration of aligned tokens, which is a key point in Connect 4.

Heuristics for Connect 4

- **Problem:** Heuristic 1 doest take into account any configuration of aligned tokens, which is a key point in Connect 4.
- **Heuristic 2.** Look for different features on the board, and attribute them different values.



Feature	Score
4 tokens connected H,V or D, e.g a1-b2-c3-d4.	$+\infty$
3 tokens connected H,V or D, e.g c2-d2-e2 or c1-d2-f4.	900000 or 50000 (∗)
2 tokens connected H,V or D, e.g b1-c1.	40000, 30000, 20000, 10000 (∗)
A token is not connected to another.	200 , 120, 70, or 40 (∗)

- The scores indicated with (∗) depend on the game configuration, e.g for Feature 2:
  - 900000 if a move can only be made on one of the immediately adjacent columns, or a same token can be found a square away from two connected tokens.
  - 50000 if a move can be made on either immediately adjacent columns.

MCTS algorithm.

Behind Monte-Carlo Tree Search

- Monte-Carlo Tree Search (MCTS) is, as Minimax, another tree search AI algorithm for two player zero-sum games.
- **Key idea:** instead of a hand-crafted heuristic to estimate the value of a game state, let's just repeatedly randomly simulate a game from that state to see if it looks promising !

Behind Monte-Carlo Tree Search

- Monte-Carlo Tree Search (MCTS) is, as Minimax, another tree search AI algorithm for two player zero-sum games.
- **Key idea:** instead of a hand-crafted heuristic to estimate the value of a game state, let's just repeatedly randomly simulate a game from that state to see if it looks promising !
- **A bit of history:**
  - The Monte Carlo method, which uses random sampling for deterministic problems difficult to solve dates back to the 1940s.
  - In 1987, Bruce Abramson combined minimax search with an expected-outcome model based on random game playouts to the end, instead of the usual static evaluation function.
  - Model "is shown to be precise, accurate, easily estimable, efficiently calculable, and domain-independent", tested on Tic-tac-toe, Othello and Chess.
  - In 2006, Rémi Coulom described the application of the Monte Carlo method to game-tree search and coined the name Monte Carlo tree search. L. Kocsis and Cs. Szepesvári developed the **UCT algorithm**, and S. Gelly et al. implemented UCT in their program *MoGo*.

Results in game theory

- In 2008, *MoGo* achieved dan (master) level in 99 Go, and the Fuego program began to win against strong amateur players in 9×9 Go.

Results in game theory

- In 2008, *MoGo* achieved dan (master) level in 99 Go, and the Fuego program began to win against strong amateur players in 9×9 Go.
- In January 2012, the Zen program won 3:1 in a Go match on a 1919 board with an amateur 2 dan player.

Results in game theory

- In 2008, *MoGo* achieved dan (master) level in 99 Go, and the Fuego program began to win against strong amateur players in 9×9 Go.
- In January 2012, the Zen program won 3:1 in a Go match on a 1919 board with an amateur 2 dan player.
- Google Deepmind developed the program *AlphaGo*. In October 2015, it became the first Computer Go program to beat a professional human Go player without handicaps on a full-sized 19×19 board.

Results in game theory

- In 2008, *MoGo* achieved dan (master) level in 99 Go, and the Fuego program began to win against strong amateur players in 9×9 Go.
- In January 2012, the Zen program won 3:1 in a Go match on a 1919 board with an amateur 2 dan player.
- Google Deepmind developed the program *AlphaGo*. In October 2015, it became the first Computer Go program to beat a professional human Go player without handicaps on a full-sized 19×19 board.
- In March 2016, AlphaGo was awarded an honorary 9-dan (master) level in 19×19 Go for defeating Lee Sedol in a five-game match with a final score of four games to one.

Results in game theory

- In 2008, *MoGo* achieved dan (master) level in 99 Go, and the Fuego program began to win against strong amateur players in 9×9 Go.
- In January 2012, the Zen program won 3:1 in a Go match on a 1919 board with an amateur 2 dan player.
- Google Deepmind developed the program *AlphaGo*. In October 2015, it became the first Computer Go program to beat a professional human Go player without handicaps on a full-sized 19×19 board.
- In March 2016, AlphaGo was awarded an honorary 9-dan (master) level in 19×19 Go for defeating Lee Sedol in a five-game match with a final score of four games to one.
- AlphaGo is a significant improvement over previous Go programs. It uses Monte Carlo tree search with artificial neuron networks to learn on a data basis of games between humans.

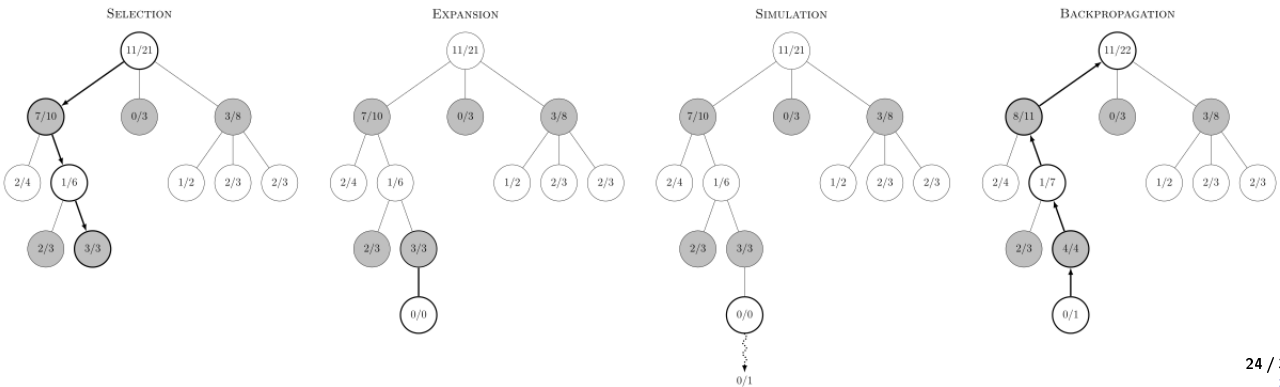
Principle of Monte-Carlo Tree Search

- The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on random sampling (called *roll-outs*) of the search space.

Principle of Monte-Carlo Tree Search

- The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on random sampling (called *roll-outs*) of the search space.

- Each round of Monte-Carlo tree search consists of 4 distinct steps:
  - **Selection:** start from root, and select successive child nodes until a leaf node L is reached.
  - **Expansion:** unless L ends the game for either player, create one or more child nodes and choose one of them: C.
  - **Simulation:** Complete one random *roll-out* from node C.
  - **Backpropagation:** Use the result of the *roll-out* to update victory rate information in the nodes on the path from C to R.



MCTS : Selection

- From a root node R, the **selection** step aims at choosing a leaf node L, that is a node that has still unexplored children. **How to make the best possible choice ?**
- The selection step must balance **exploitation**, which favors moves that have performed well so far, and **exploration**, which gives opportunities to moves that have been simulated less often.

MCTS : Selection

- From a root node R, the **selection** step aims at choosing a leaf node L, that is a node that has still unexplored children. **How to make the best possible choice ?**
- The selection step must balance **exploitation**, which favors moves that have performed well so far, and **exploration**, which gives opportunities to moves that have been simulated less often.
- **Theorem [Kocsis, Szepesvári]: Upper Confidence Bound applied to trees (UCT)**

The best choice in each node of the game tree is the move for which the expression

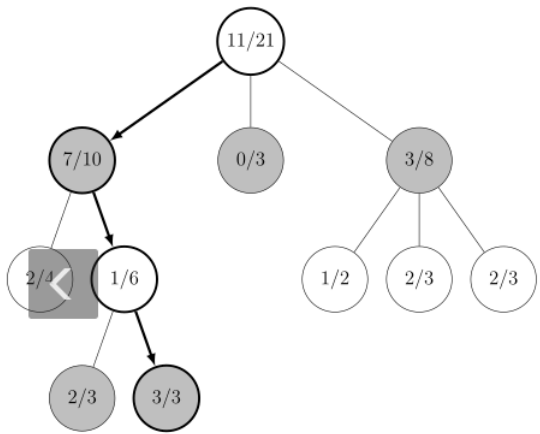
$$\frac{w_i}{n_i} + c\sqrt{\frac{\log(N_i)}{n_i}}$$

has the highest value, where:

- w_i stands for the number of wins for the node considered after the i-th move for the current player,
- n_i is the number of simulations for the node considered after the i-th move,
- N_i stands for the total number of times the parent node has been visited,
- c is the **exploration parameter**, often times equal to $\sqrt{2}$, but in practice is chosen empirically.

MCTS : Selection

- On this example:



- Start from root node R:

Child Node	Left	Middle	Right
UCT value	$\frac{7}{10} + \sqrt{2}\sqrt{\frac{\log(21)}{10}}$ $\simeq$ **1.480**	$\frac{0}{3} + \sqrt{2}\sqrt{\frac{\log(21)}{3}}$ $\simeq 1.425$	$\frac{3}{8} + \sqrt{2}\sqrt{\frac{\log(21)}{8}}$ $\simeq 1.247$

- So with UCT, we select the leftmost child. One can then repeat this process until a leaf is reached

MCTS : Selection

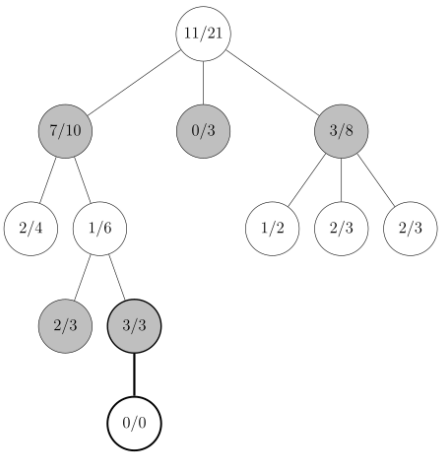
Data: A position Pos of the game.
Result: The leaf chosen by UCT.

Function `Select(`*Pos*`)`:

- **if** *Pos is a terminal position* **then return** Pos;
- $N \leftarrow$ number of visits of Pos;
- **if** $N = 0$ **then return** Pos;
- $\mathcal{M} \leftarrow$ {legal moves from Pos};
- (max, best) $\leftarrow (-1, \varnothing)$;
- **for** *each* $m \in \mathcal{M}$ **do**
  - $\text{Pos}_m \leftarrow$ position obtained from Pos by playing m ;
  - $n_m \leftarrow$ number of visits of Pos_m;
  - **if** $n_m = 0$ **then return** Pos;
  - new_eval $\leftarrow$ UCT(Pos,m);
  - **if** *new_eval* $>$ *max* **then** (max, best) $\leftarrow$ (new_eval, m);
- **end**
- Pos' $\leftarrow$ position obtained from best;
- **return** Select (Pos')

MCTS : Expansion

- **Expansion** arises when we reach an unvisited children. We then add it to the tree with 0 wins and 0 visits.

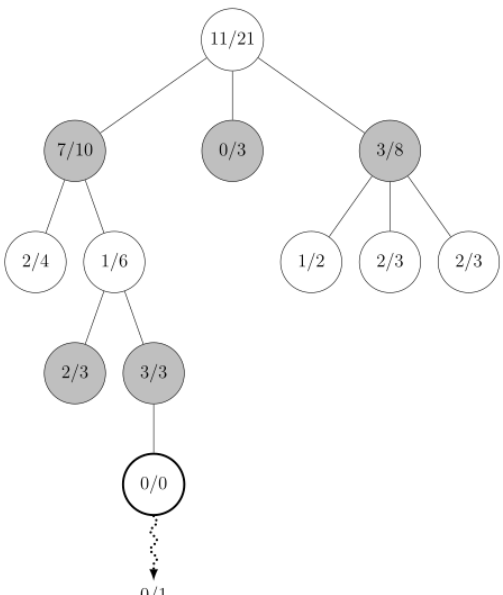


Data: A position Pos of the game.
Result: The game tree with a new leaf Pos'.

Function Expand(*Pos*):
  if *Pos is a terminal position* **then return** Pos;
  $N \leftarrow$ number of visits of Pos;
  if $N = 0$ **then** Make all possible moves from Pos;
  $\mathcal{M} \leftarrow$ {legal moves from Pos};
  for *each* $m \in \mathcal{M}$ **do**
    $\text{Pos}_m \leftarrow$ position obtained from Pos by playing m ;
    $n_m \leftarrow$ number of visits of Pos_m;
    if $n_m = 0$ **then**
      Pos' $\leftarrow$ position obtained from Pos by playing m ;
      return Pos'
    end
  end

MCTS : Simulation

- **Simulation** consists in estimating the value of the new node using a *roll-out*, that is a random simulation of the game until the end is reached.

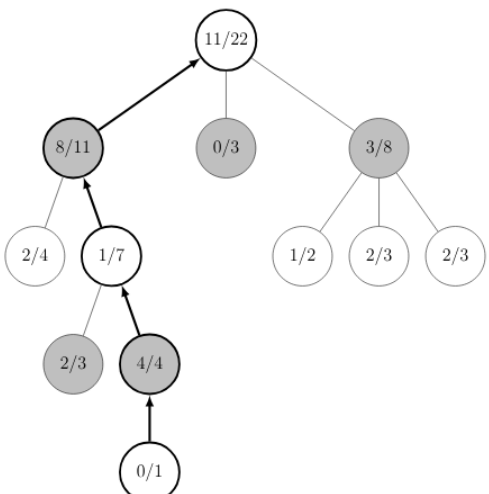


Data: A position Pos of the game.
Result: A value in $\{0, 1\}$ (*roll-out* is winning or not).

Function `Simul(`*Pos*`)`:
  while *Pos is not a terminal position* **do**
    $\mathcal{M} \leftarrow$ {legal moves from Pos};
    $m \leftarrow$ random choice in $\mathcal{M}$;
    Pos $\leftarrow$ position obtained from Pos by playing m ;
  end
  if *Pos is winning* **then**
    return 1
  end
  else
    return 0
  end

MCTS : BackPropagation

- Once a score has been determined for the new node, **backpropagation** consists in updating all the number of wins and visits of the tree.

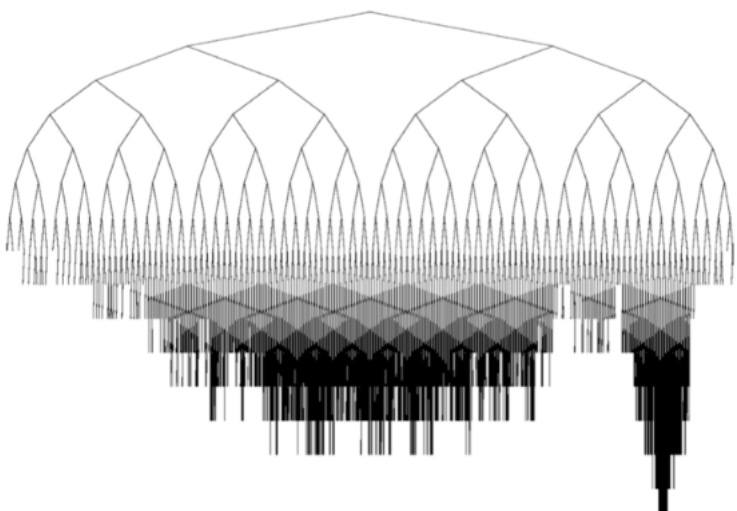


Data: A position Pos of the game with its *roll-out* score.
Result: An update of the decision tree.

Function BackP(*Pos*):
  if *P is the root node (i.e. has no parent)* **then return**;
  if *score* = 1 **then**
    Add 1 in the number of wins and the number of visits of Pos.
  end
  else
    Add 1 in the number of visits of Pos.
  end
  return BackP(*Parent(Pos)*,*score*)

Remarks

- MCTS has the advantage that it works well without expert knowledge of the game.
- MCTS is anytime: the more MCTS steps we do, the more accurate it is and is likely to select the best moves.
- MCTS is easy to parallelize: indeed, one can do several rollouts for the same node on parallel CPUs to get a better estimate.



from *A survey of Monte Carlo Tree Search Methods* by Browne, et al. 2012.