COMP30030: Introduction to Artificial Intelligence

Neil Hurley

School of Computer Science
University College Dublin
neil.hurley@ucd.ie

October 4, 2018

Adversarial Search

- This is about trying to win when someone is actively trying to beat us.
- Games are an excellent example.
- It involves techniques similar to searching (or outright use of search techniques), but the adversary is another variable to take care of.
- This often considerably widens the search space.
- Opponents Mean Competition
  - Although it is possible to work out a sequence of actions to achieve a winning state, it is highly likely that a move of your opponent will prevent you carrying out that plan.
  - Can we make any assumptions about your opponent's expected actions

Example: Chess

Average branching factor of 35.

- Games running for 50 moves.
- That is 10^{154} nodes (10^{40} distinct ones).
- Impossible to search them all.
- However, we still have to act within a time limit.
- We need suboptimal strategies.

- It is still possible, in theory, to visualise the search tree.
- In this case, though, our player has only control on the moves on alternate levels of the tree. The other player moves in the other cases.
- Although we can't predict for sure how the adversary will play, we can still consider all its moves, and compute the most favourable.
- A common assumption is that the adversary will carry out the best move (for them).

MINIMAX value I

- A game ends with a certain utility (e.g. $+1$ for win, 0 for draw or -1 for loss).
  - Note that one player wants to win. His opponent wants him to lose (i.e. his opponent wants to win)
  - So one player – let's call him the **MAX** – wants to maximise the utility
  - His opponent – the **MIN** player – wants to minimise the utility.
- How good is any particular node, when it's **MAX**'s turn to play? i.e. how good is any particular move?

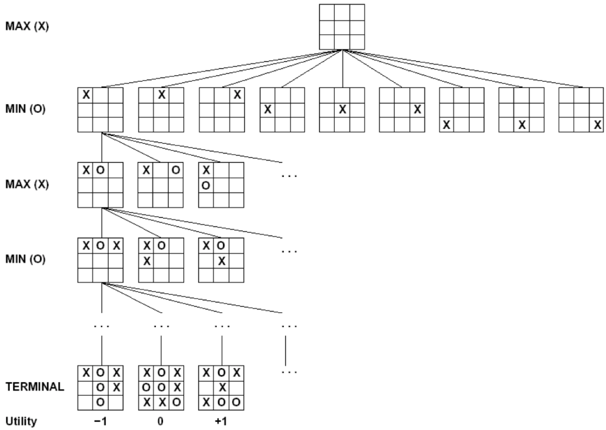
MINIMAX value II

- If **MAX** assumes that **MIN** will play the best possible moves from his point of view
  - he can follow a path of moves from the current node to the end of the game
  - each time it is his move, he can assume that he will play the move with maximum utility
  - each time it is the opponent's move, he can assume that the opponent will play the move with minimum utility
- Expanding the whole tree and working backwards from the bottom of the tree
  - At leaf nodes, the utility can be measured directly – depending on whether the node represents a win (+1), loss (-1) or draw (0).

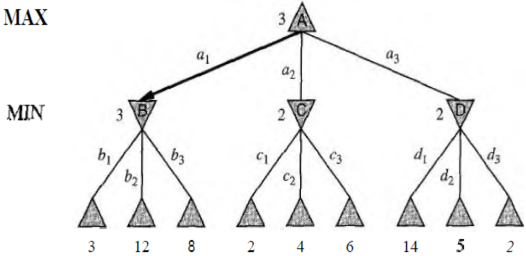
MINIMAX value III

- At the next level up
  - if its **MIN**'s move, the utilities of each state will be the minimum of those of the children below it.
  - if its **MAX**'s move, the utilities of each state will be the maximum of those of the children below it
- **MAX** is then playing the best move that can be made, assuming **MIN** is also doing the same – if **MIN** plays sub-optimally, **MAX** will do even better.

TicTacToe



Computing Minimax



Minimax

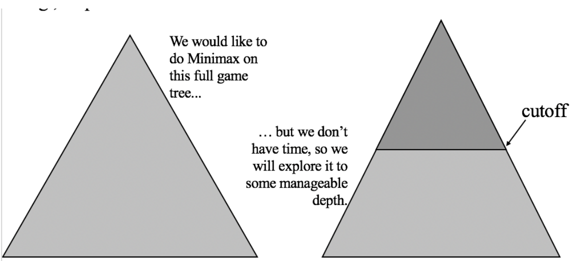
Very simple: recursively calculate the MINIMAX value of the possible choices at the current state, then pick the best.

- The recursion implicitly implements a depth first search.
- It is optimal.
- Unfortunately, its complexity is $o(b^d)$ where b is the branching factor and d the number of levels: impractical for any real game.

Imperfect Decisions and Lookahead

Static Evaluation Approach with MiniMax

- Cut off search according to some cut-off test.
- For example, limit the search to depth m
- Problem: payoffs are defined only at terminal states.
- Solution: Evaluate the pre-terminal leaf states using **heuristic** evaluation function rather than using the actual payoff function.



We would like to do Minimax on this full game tree...
… but we don't have time, so we will explore it to some manageable depth.
cutoff

Static Evaluation Heuristics

- Must evaluate possible board configurations (nodes) with incomplete information.
- Used to estimate the chances of winning from that node.

Important qualities:

- Must agree with the payoff function at the terminal states.
- Must not take long to compute.
- Should be accurate enough.

For example, one figures out how good a position is for the computer, and how good it is for the opponent, and subtracts the opponent's score from the computer's.
E.g., Chess: (Value of all white pieces) - (Value of all black pieces)

Example Evaluation Functions I

Tic Tac Toe

Assume Max is using "X"

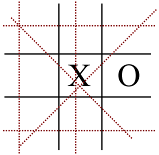
$e(n)$ =

if n is win for Max, $+\infty$

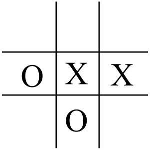
if n is win for Min, $-\infty$

else

(*possible* number of rows, columns and diagonals available to Max) - (*possible* number of rows, columns and diagonals available to Min)



$e(n) = 6 - 4 = 2$



$e(n) = 4 - 3 = 1$

Example Evaluation Functions II

Tic Tac Toe

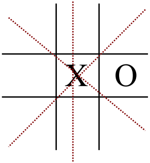
Assume Max is using "X"

$e(n)$ =

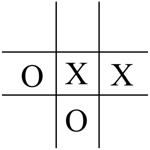
if n is win for Max, $+\infty$
if n is win for Min, $-\infty$

else

(*current* number of rows, columns and diagonals available to Max) - (*current* number of rows, columns and diagonals available to Min)



$e(n) = 3 - 1 = 2$



$e(n) = 3 - 2 = 1$

Example Evaluation Functions III

Tic Tac Toe

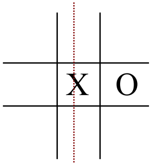
Again assume Max is using "X"

$e(n)$ =

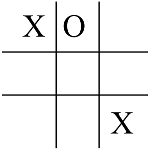
if n is win for Max, $+\infty$
if n is win for Min, $-\infty$

else

(Lowest number of moves for *Min* to win) - (Lowest number of moves for *Max* to win)



$e(n) = 2 - 2 = 0$



$e(n) = 2 - 1 = 1$

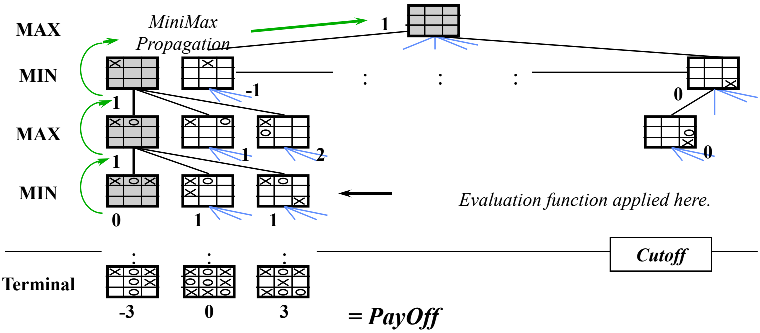
An Example

Tic-Tac-Toe

MaxWin = Number of moves for MAX to win

MinWin = Number of moves for MIN to win

$$\textbf{Eval} = \begin{cases} +3 & \textit{if Max wins} \\ -3 & \textit{if Min wins} \\ \text{MinWin-MaxWin} & \textit{Otherwise} \end{cases}$$



Heuristics in Chess

- Assume MAX is white
- Assume each piece has the following material value:

$$
\begin{aligned}
\text{pawn} &= 1 \\
\text{knight} &= 3 \\
\text{bishop} &= 3 \\
\text{rook} &= 5 \\
\text{queen} &= 9
\end{aligned}
$$

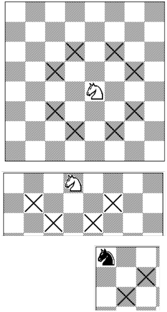
- Let w = sum of the value of the white pieces
- Let b = sum of the value of the black pieces

$$e(n) = \frac{w - b}{w + b} \quad e(n) \in [-1, 1]$$

Heuristics in Chess

- The previous evaluation function naively gave the same weight to a piece regardless of its position on the board...
- Let X_i be the number of squares the ith piece attacks.

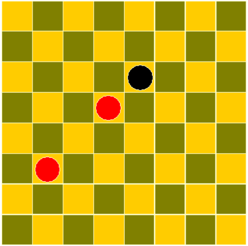
$$e(n) = X_1 \times \text{piecevalue}_1 + X_2 \times \text{piecevalue}_2 + \ldots$$



Problems with Cut-offs

- *non-quiescent states* (unstable states, where wild utility swings are going to occur: e.g. a player takes the other player's queen 1 move further)
- *horizon effects* (you stall something for a while, thereby moving it past max depth, but it will eventually occur anyway, just past the point you can see)

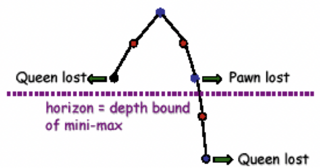
Non-quiescent sate for Draughts



- Heuristic function just counts the number of pieces available to each player – red player doing wel..
- However, it misses that in a following state, black player can capture both pieces by double jump

Horizon Effect

- A negative horizon effect
  - MAX may try to avoid a bad situation which is actually inevitable. For example, MAX tries to avoid losing the white queen and appears to be able to do so using a lookahead tree of depth 4, but a little deeper it becomes obvious that the queen is going to be lost.



- A positive horizon effect
  - MAX may not realise that something good is going to be achievable. For example, MAX would like to take MIN's queen and that can happen – but the restricted horizon prevents MAX from making the right choices to realise this possibility

Pruning

- Heuristics allow the space to be **pruned**, but whether we choose the best move depends on the quality of the heuristic.
- However, a basic strategy exists for pruning MINIMAX, which is still guaranteed to choose the optimum move – namely alpha-beta pruning.
- Note that while alpha-beta pruning can provide significant savings over a raw MINIMAX search, cut-offs and heuristic evaluation WILL be necessary for any realistic game with a large search space.

Alpha-beta Pruning I

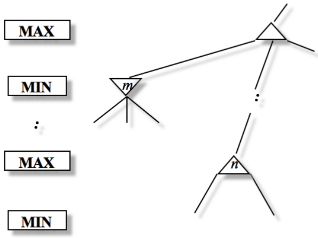
- The general principle is this: consider a node n somewhere in the tree, such that a Player has a choice of moving to that node.
- If the Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.
- So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Alpha-beta Pruning II

- Depth first search of game tree, keeping track of:
- *Alpha*: Highest value seen so far on maximizing level
- *Beta*: Lowest value seen so far on minimizing level

Pruning

- When Minimizing,
  - do not expand any more child nodes once a node has been seen whose evaluation is smaller than Alpha
- When Maximizing,
  - do not expand any child nodes once a node has been seen whose evaluation is greater than Beta



- If *m* is better than *n* for MAX, then *n* will never get into play because *m* will be chosen in preference

Alpha-beta Algorithm

```
MaxValue (Node,α,β)          Returns MiniMax value of Node

 If CutOff-Test(Node) then return Eval(Node)
 For each Child of Node do
   α:=Max(α,MinValue(Child,α,β))
   if α>=β then return β
 End For
Return α
```

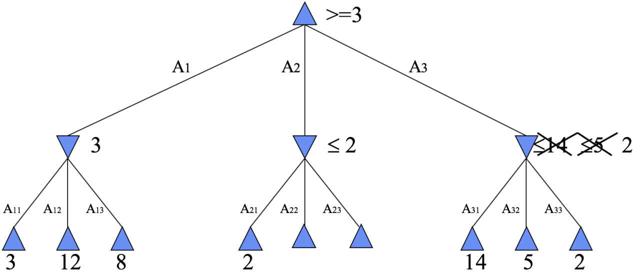
```
MinValue (Node,α,β)          Returns MiniMax value of Node

 If CutOff-Test(Node) then return Eval(Node)
 For each Child of Node do
   β:=Min(β,MaxValue(Child,α,β))
   if β>=α then return β
 End For
Return β
```

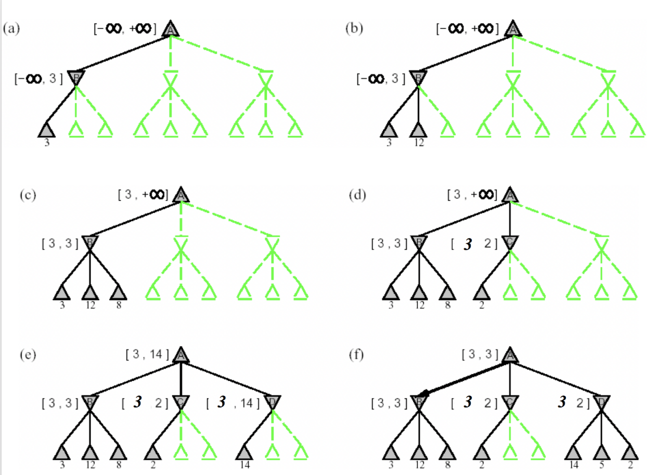
α represents MAX's best score along a particular path.

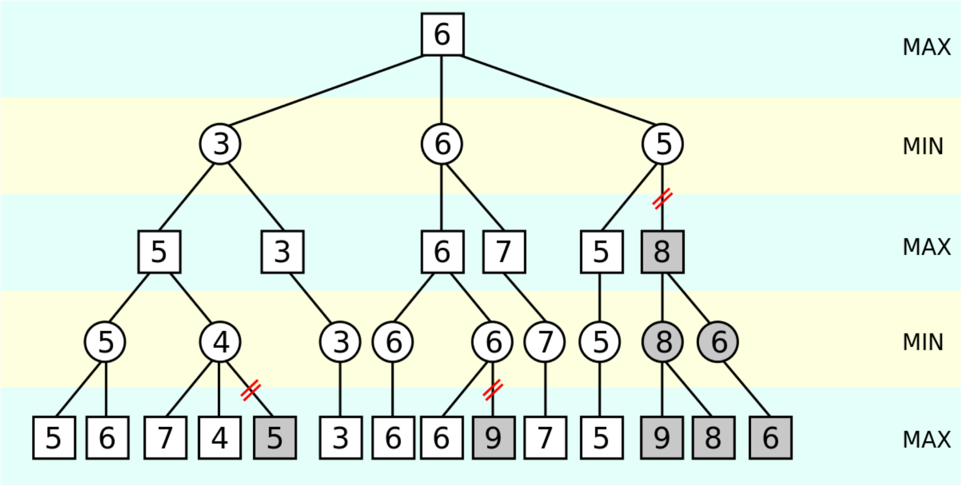
β represents the best score for MIN along this same path.

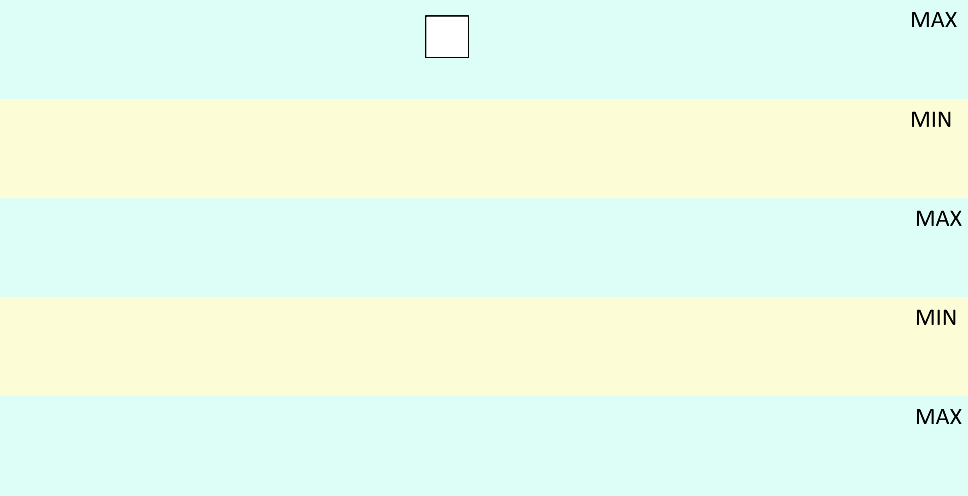
Alpha-beta Example I

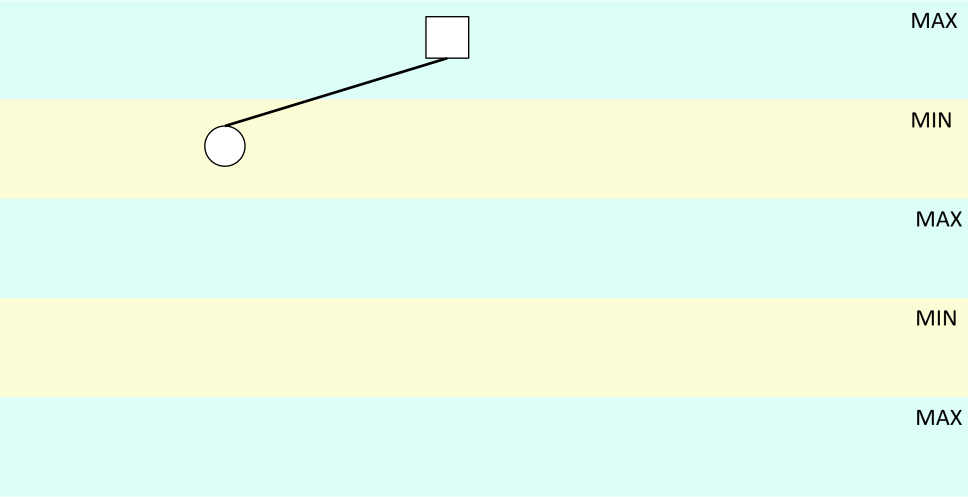


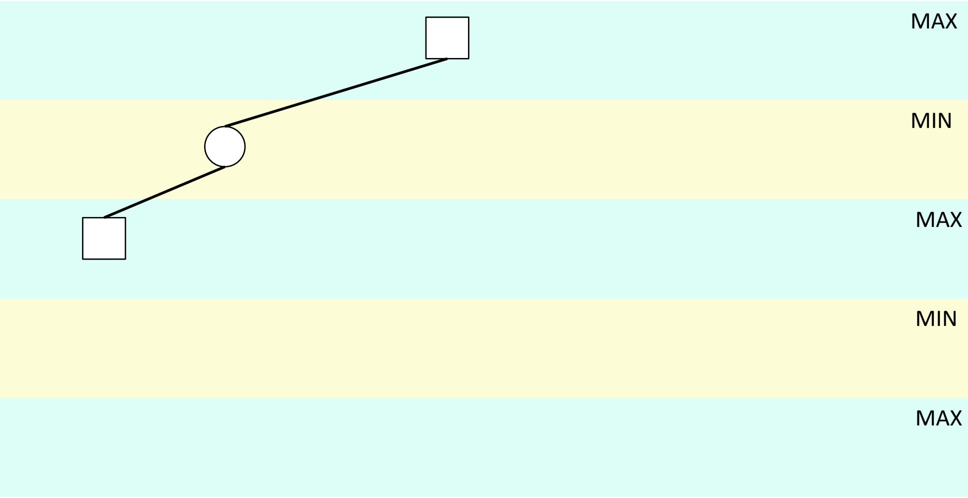
Alpha-beta Example II

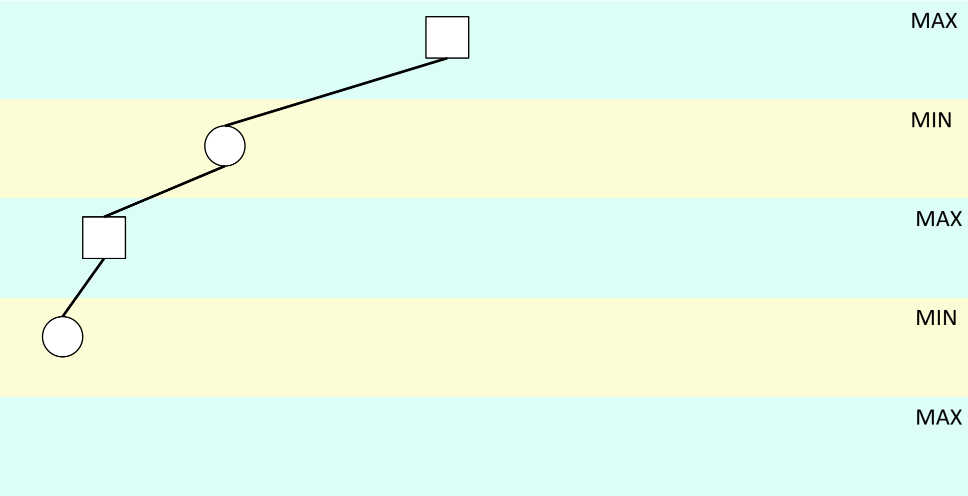


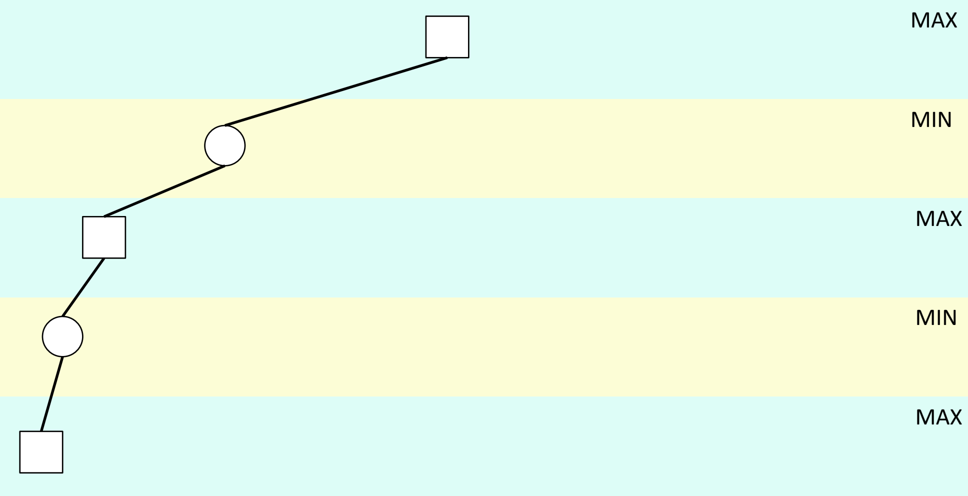

MAX
MIN
MAX
MIN
MAX

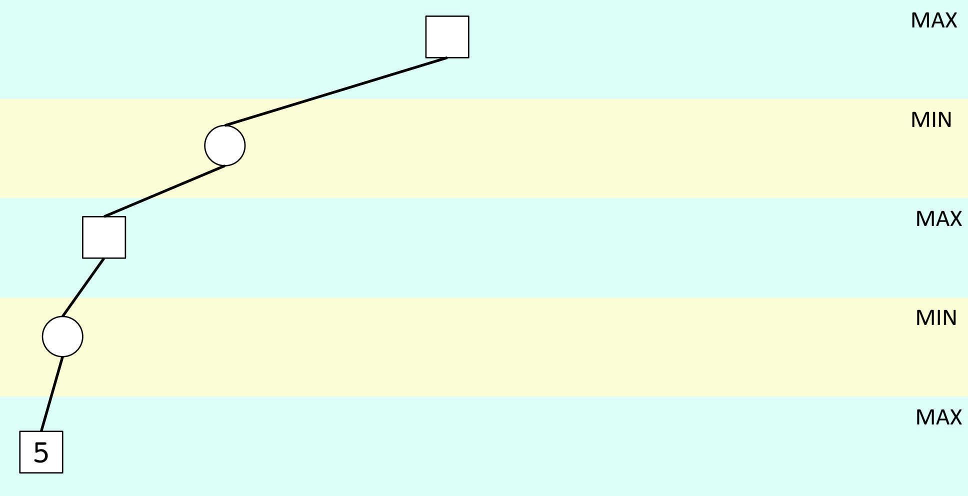

MAX
MIN
MAX
MIN
MAX

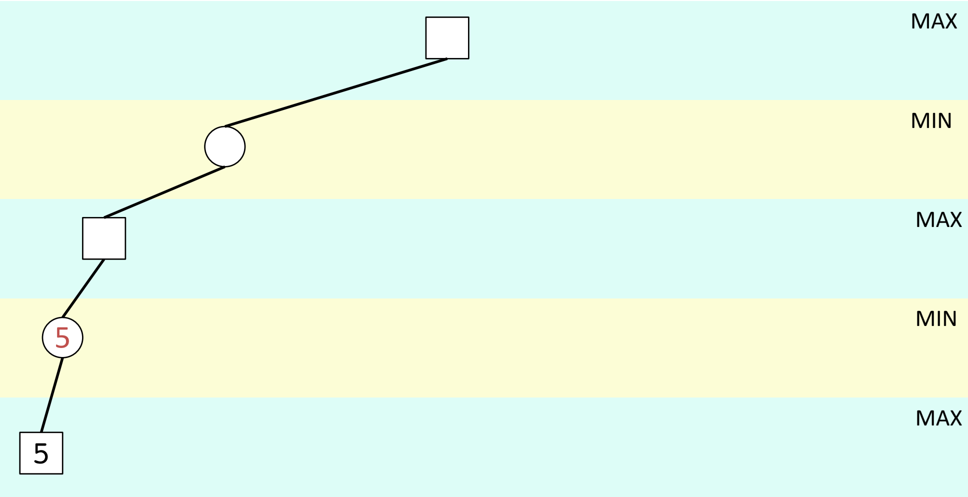

MAX
MIN
MAX
MIN
MAX

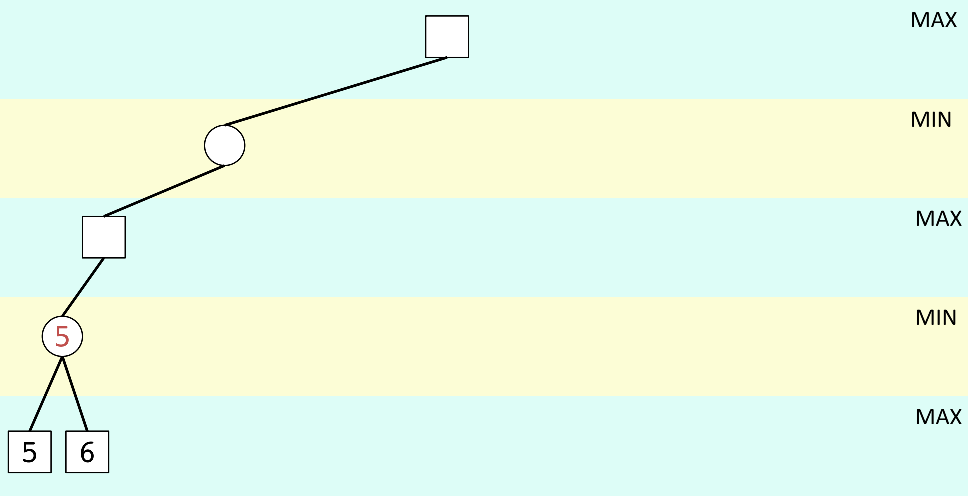

MAX
MIN
MAX
MIN
MAX


MAX
MIN
MAX
MIN
MAX

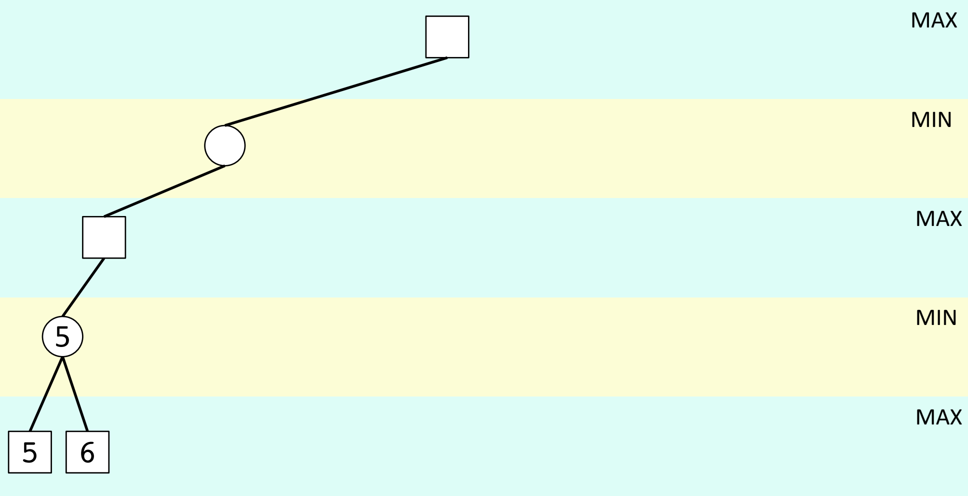

MAX
MIN
MAX
MIN
MAX

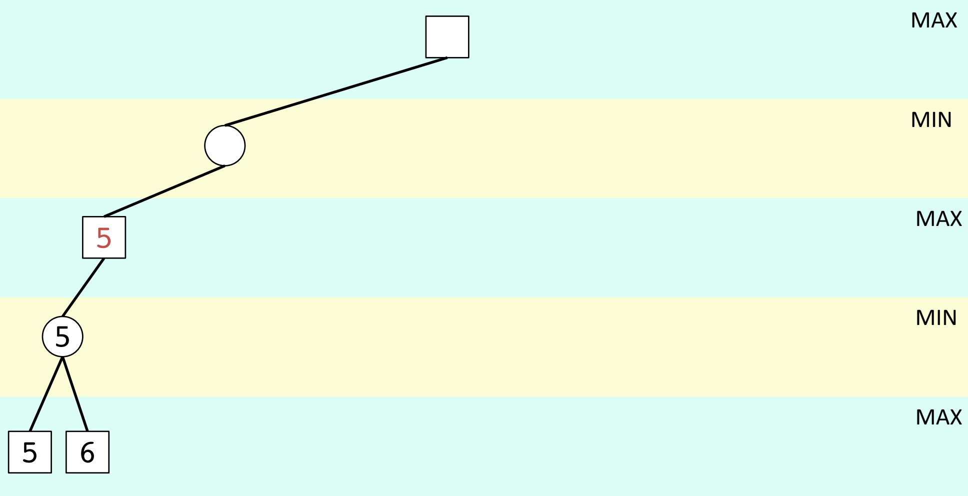

MAX
MIN
MAX
MIN
MAX
5

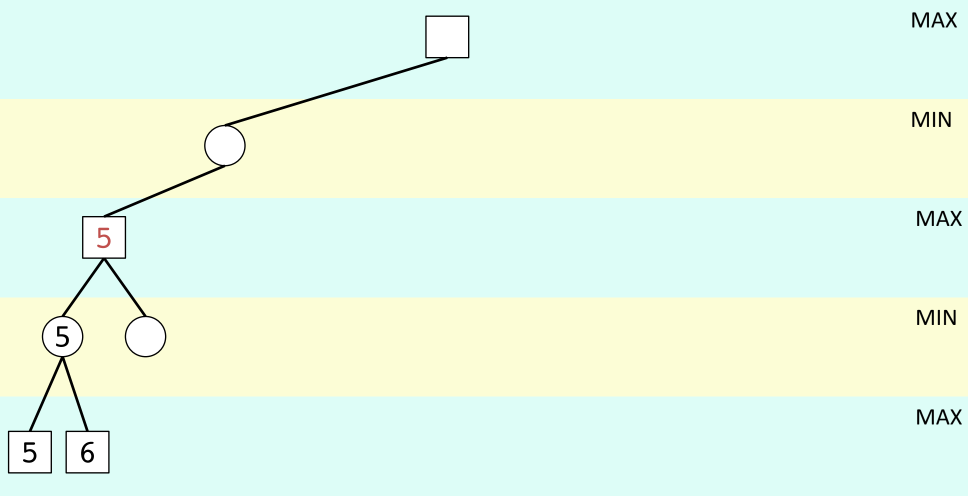

MAX
MIN
MAX
MIN
5
MAX
5



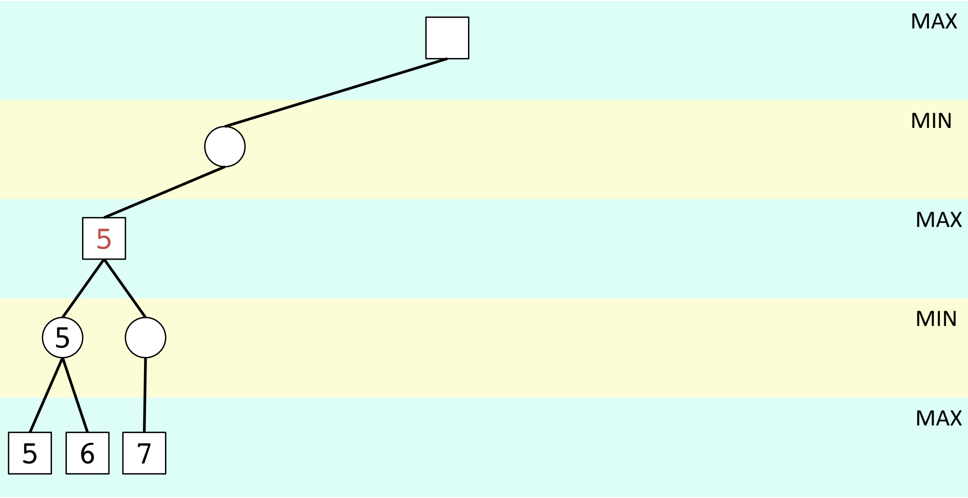
MAX
MIN
MAX
MIN
MAX
5
5
6

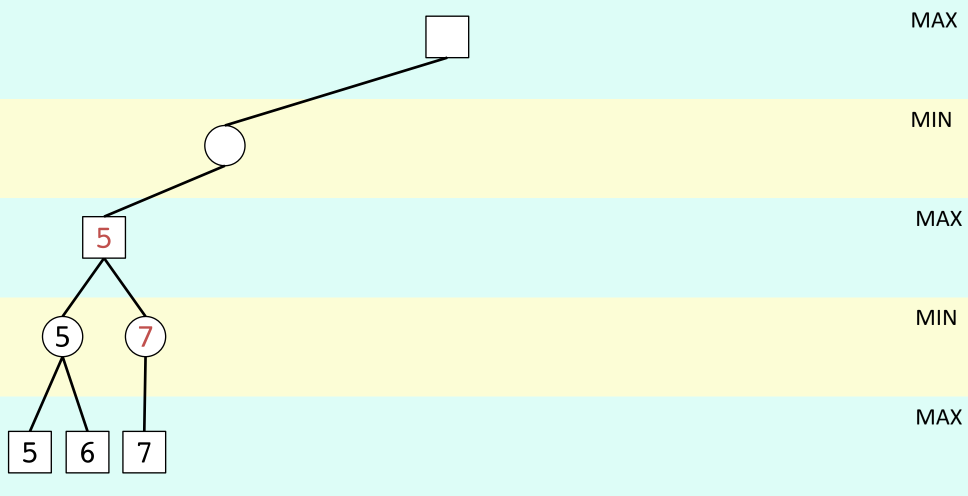

MAX
MIN
MAX
MIN
MAX
5
5
6

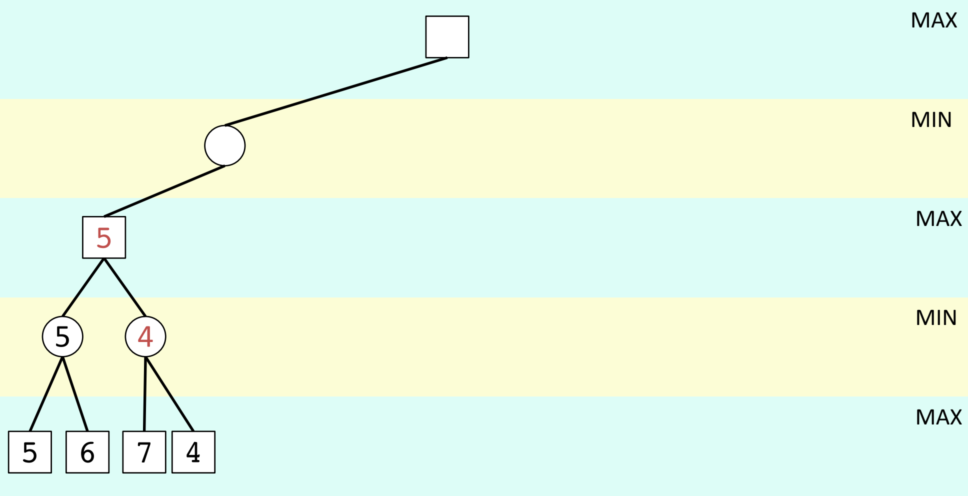

MAX
MIN
MAX
MIN
MAX
5
5
5
6

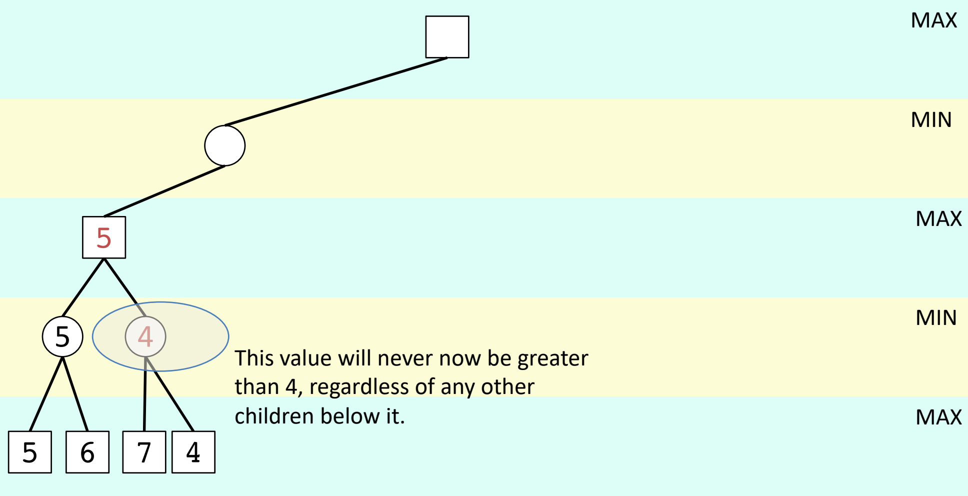


Each time a new branch is started, the child receives the current utility of its parent. For example, at this point, the child on this new branch receives its parent's utility value of 5

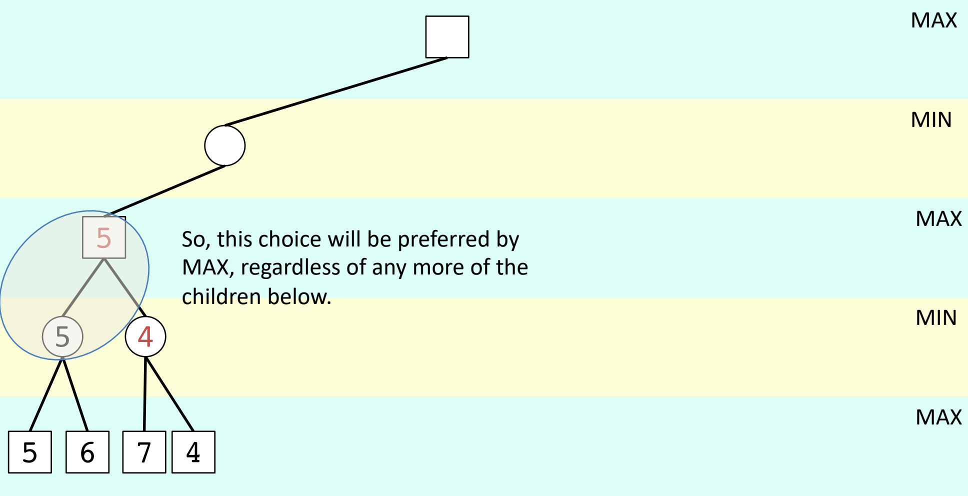

MAX
MIN
MAX
MIN
MAX
5
5
5
6
7


MAX
MIN
MAX
MIN
MAX
5
5
7
5
6
7

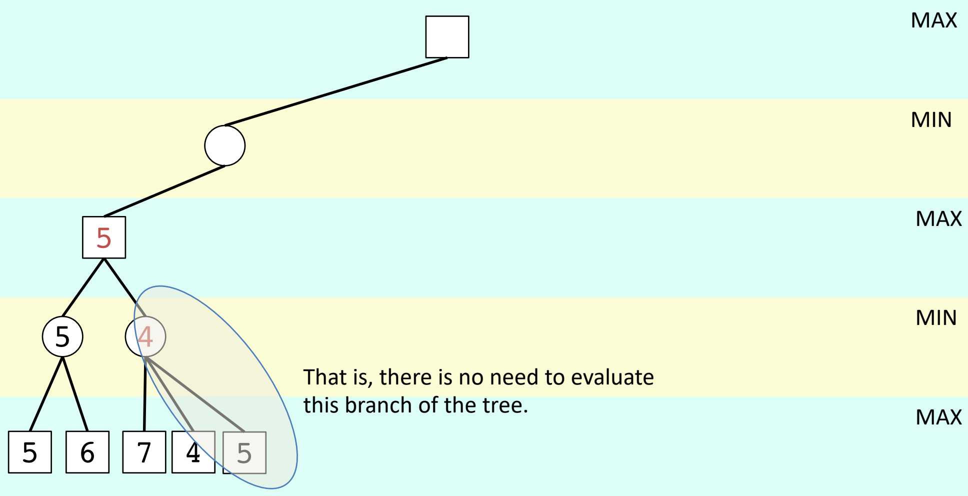

MAX
MIN
MAX
MIN
MAX
5
5
4
5
6
7
4



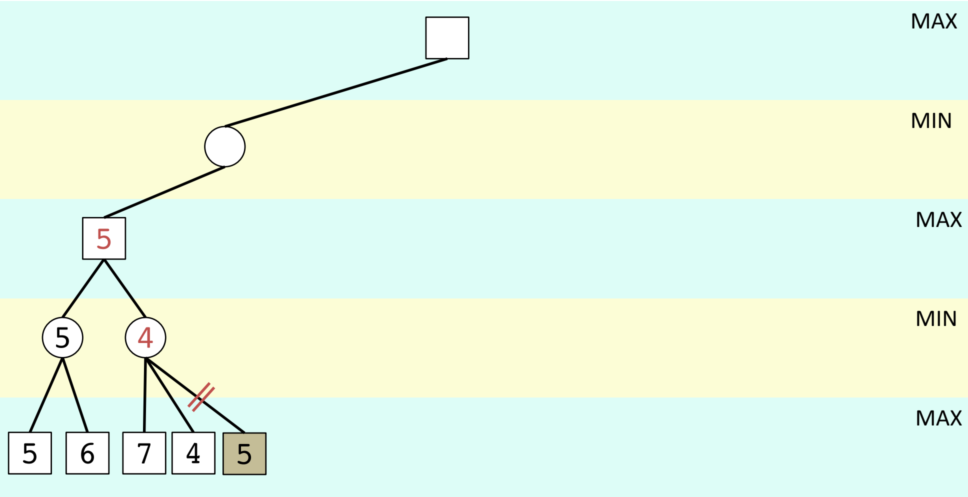
This value will never now be greater than 4, regardless of any other children below it.

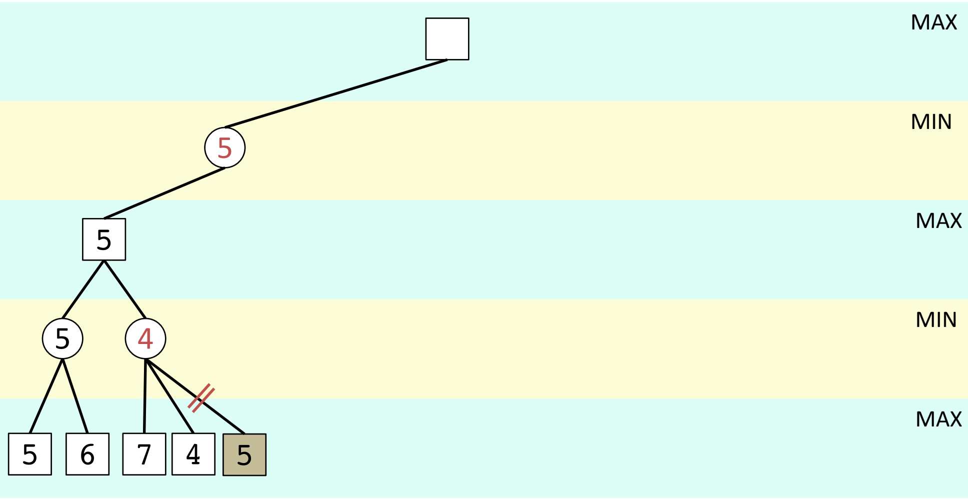


It is possible for the node to know that should no longer compute any more children because it was given its parent’s current utility value i.e. Here, the circular node with current utility of 4, checks its parent’s current utility (5) and knows it can stop expanding since 4<5

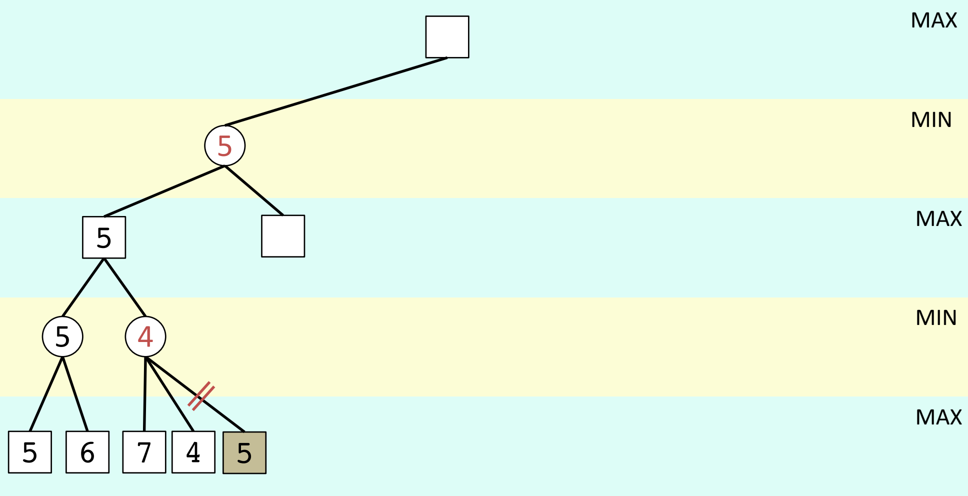


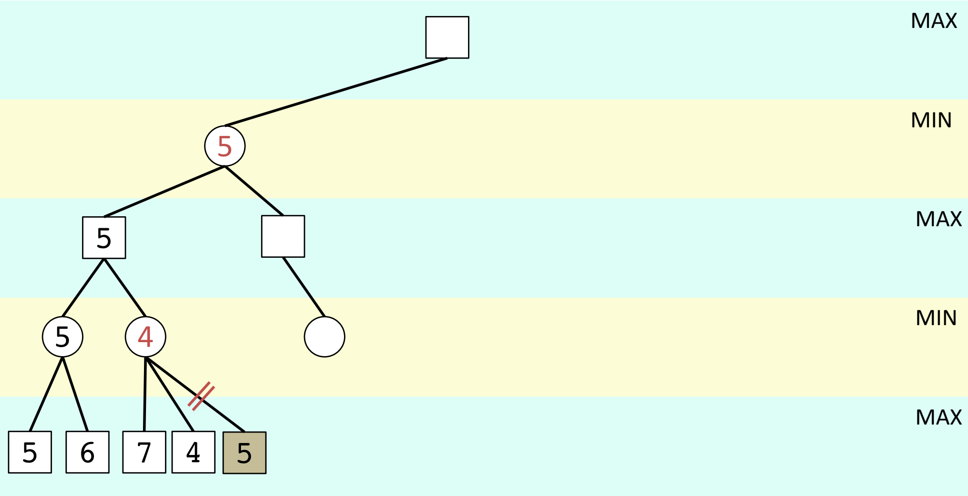
That is, there is no need to evaluate this branch of the tree.

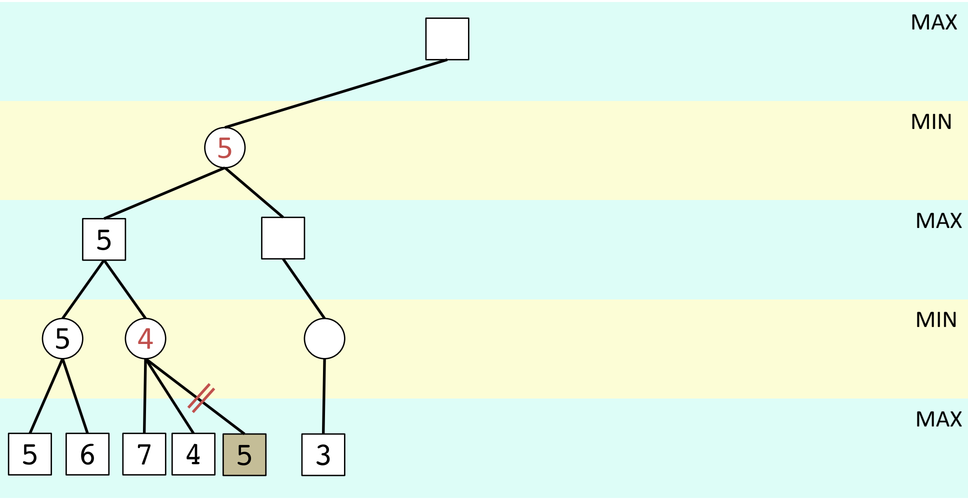

MAX
MIN
MAX
MIN
MAX
5
5
4
5
6
7
4
5

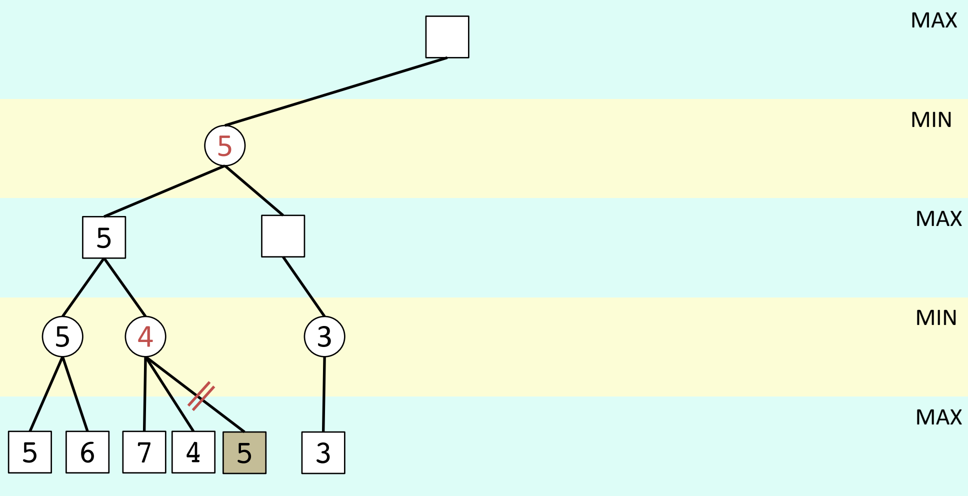


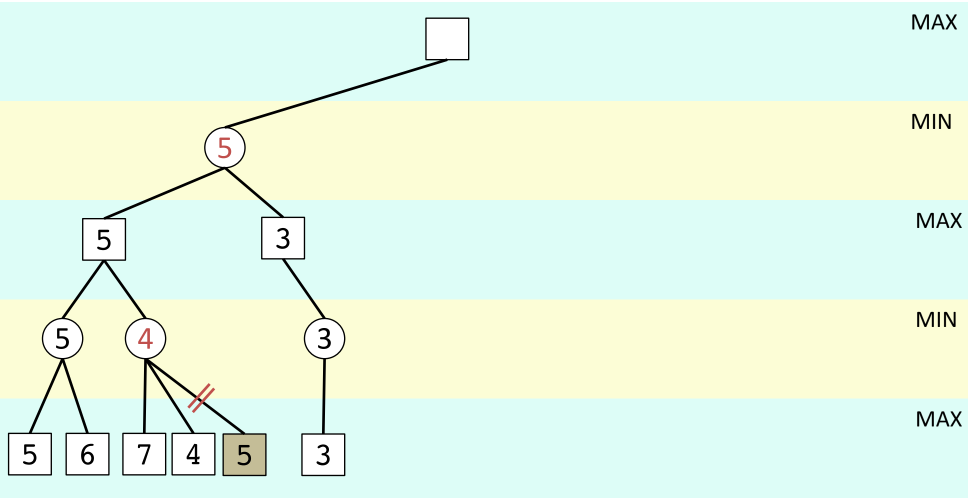
MAX
MIN
MAX
MIN
MAX
5
5
5
4
5
6
7
4
5


MAX
MIN
MAX
MIN
MAX
5
5
5
4
5
6
7
4
5

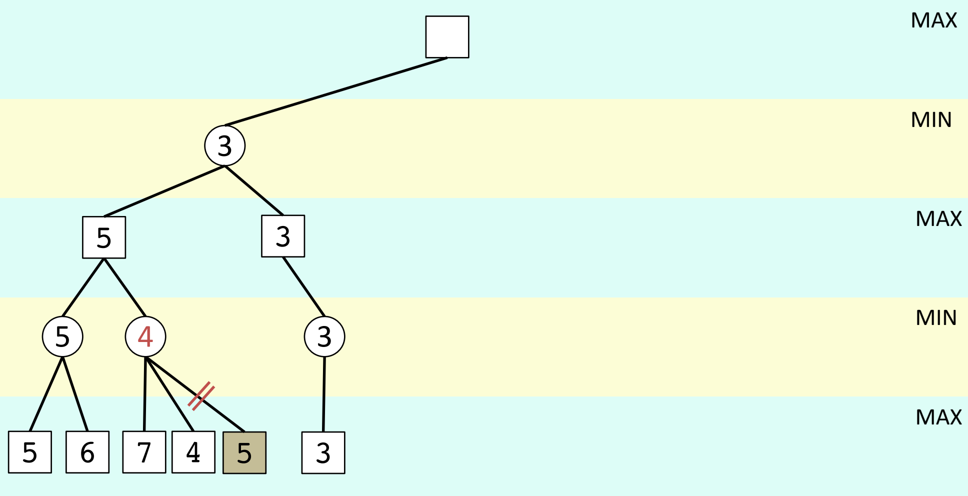

MAX
MIN
MAX
MIN
MAX
5
5
5
4
5
6
7
4
5

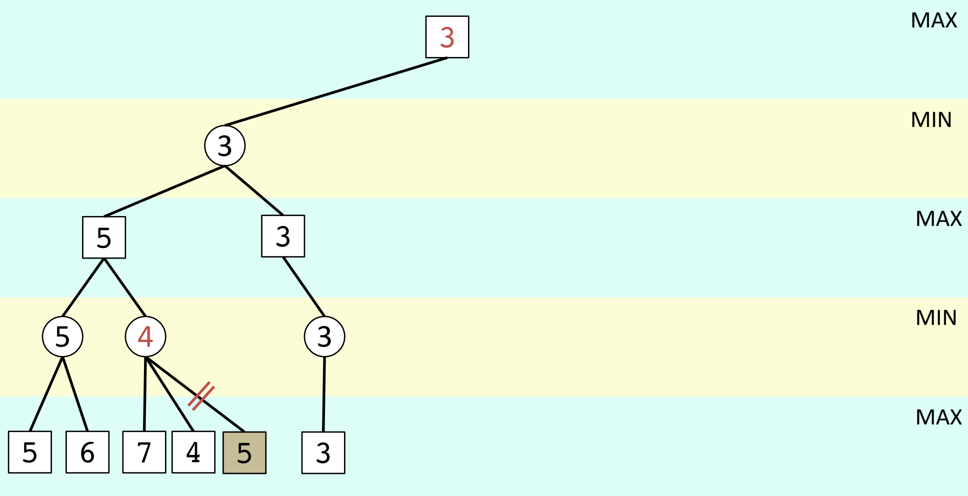

MAX
MIN
MAX
MIN
MAX
5
5
5
4
5
6
7
4
5
3

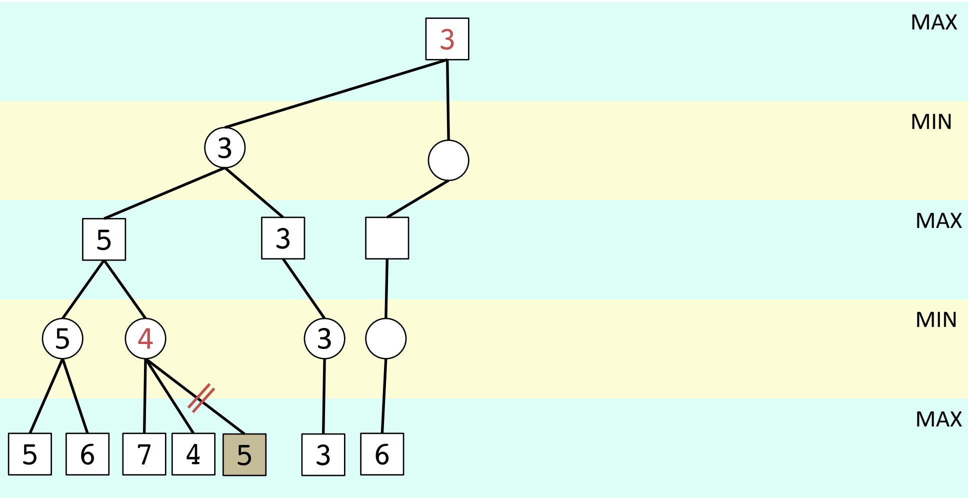

MAX
MIN
MAX
MIN
MAX
5
5
5
4
3
5
6
7
4
5
3

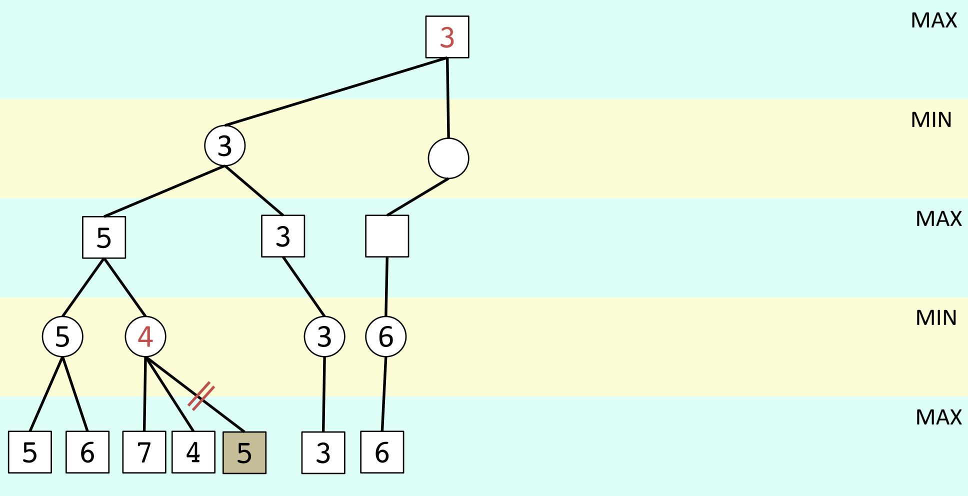


MAX
MIN
MAX
MIN
MAX
5
5
3
5
4
3
5
6
7
4
5
3

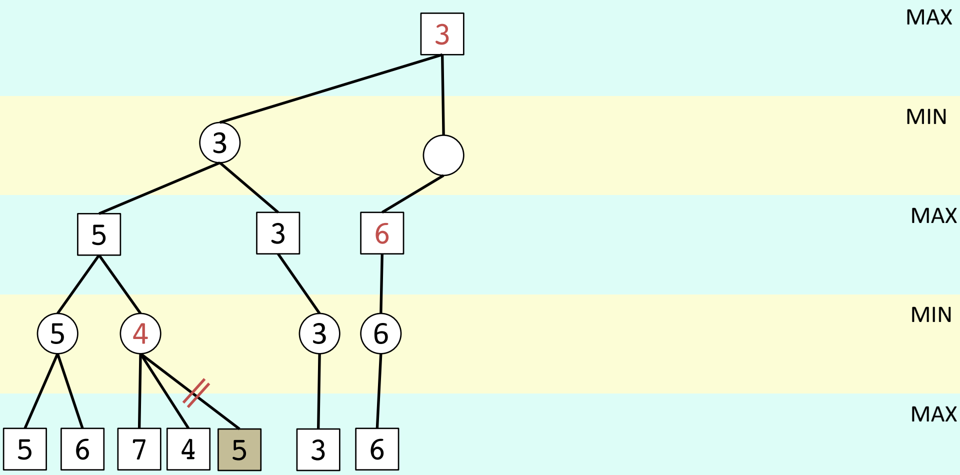

MAX
MIN
MAX
MIN
MAX
3
5
3
5
4
3
5
6
7
4
5
3


MAX
MIN
MAX
MIN
MAX
3
3
5
3
5
4
3
5
6
7
4
5
3

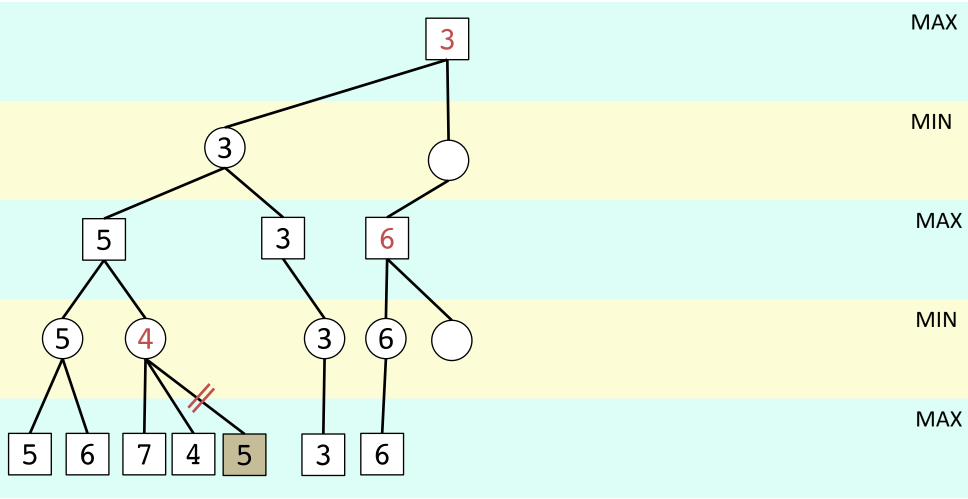

MAX
MIN
MAX
MIN
MAX
3
3
5
3
5
4
3
5
6
7
4
5
3
6

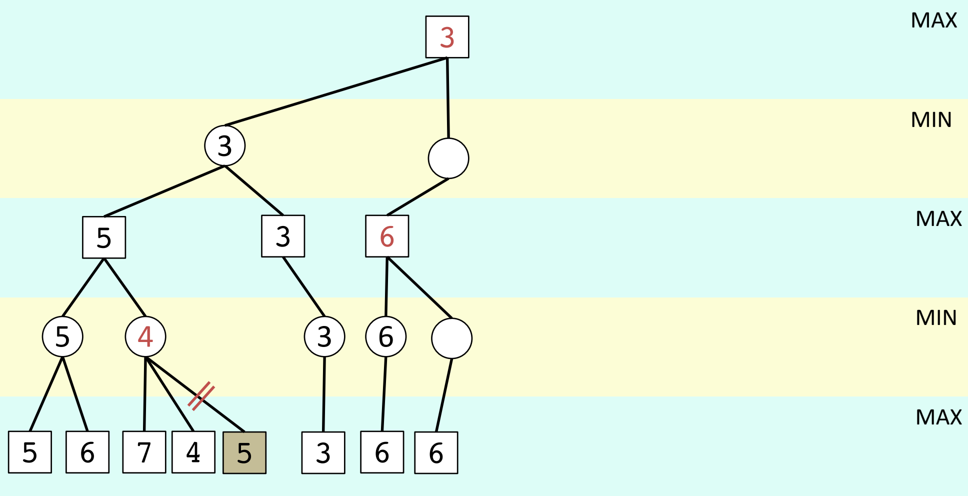


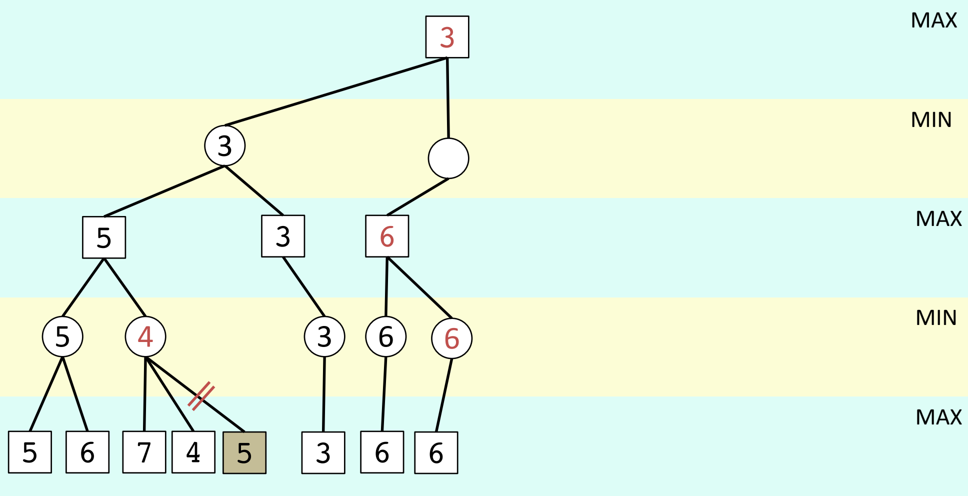
MAX
MIN
MAX
MIN
MAX
3
3
5
3
5
4
3
6
5
6
7
4
5
3
6

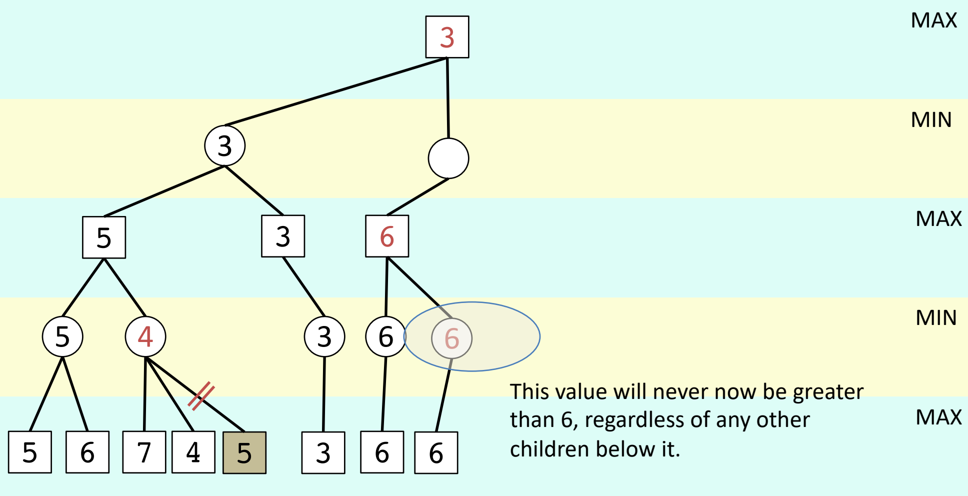


MAX
MIN
MAX
MIN
MAX
3
3
5
3
6
5
4
3
6
5
6
7
4
5
3
6

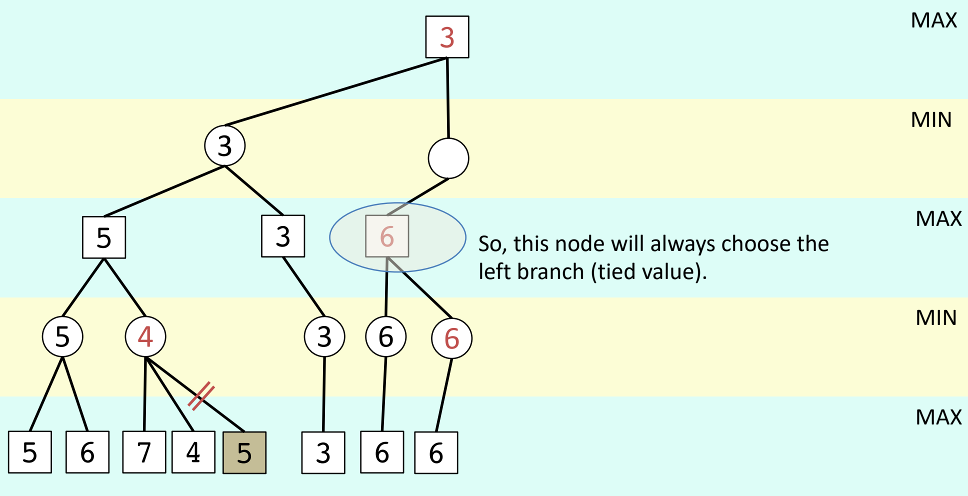

MAX
MIN
MAX
MIN
MAX
3
3
5
3
6
5
4
3
6
5
6
7
4
5
3
6

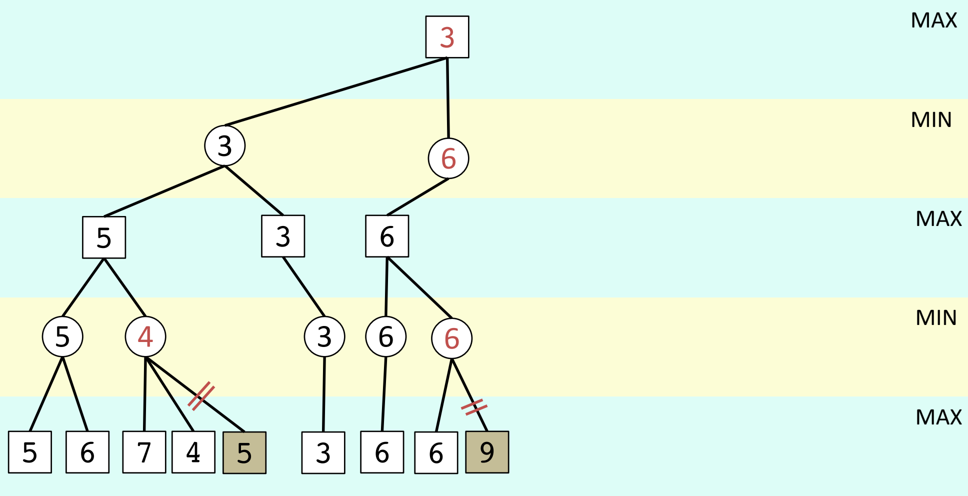

MAX
MIN
MAX
MIN
MAX
3
3
5
3
6
5
4
3
6
5
6
7
4
5
3
6
6

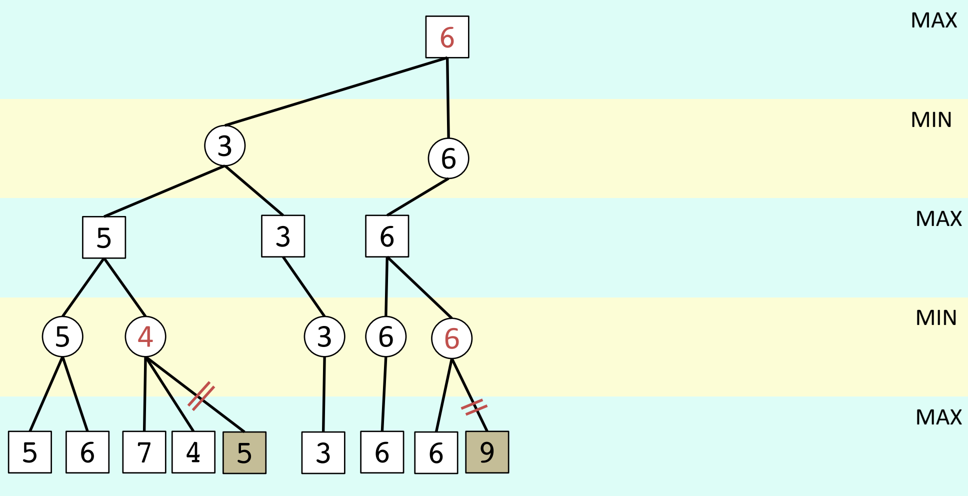

MAX
MIN
MAX
MIN
MAX
3
3
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6

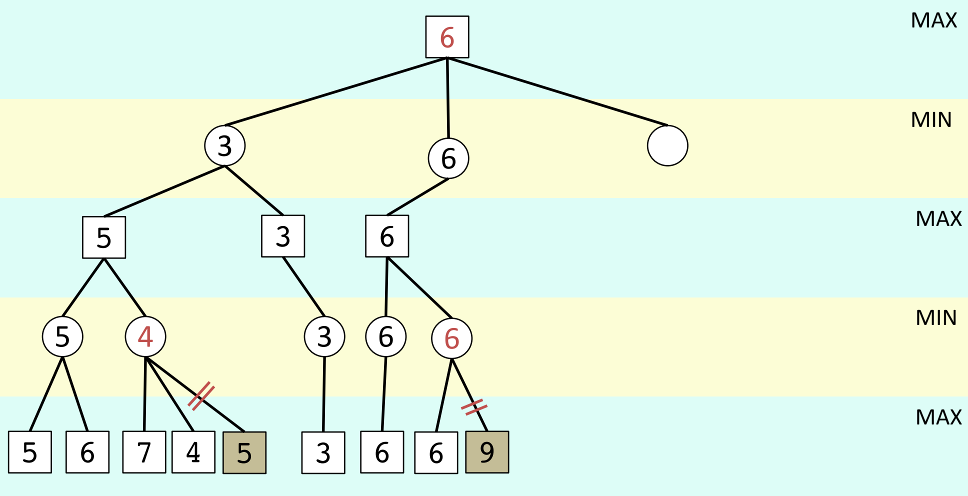


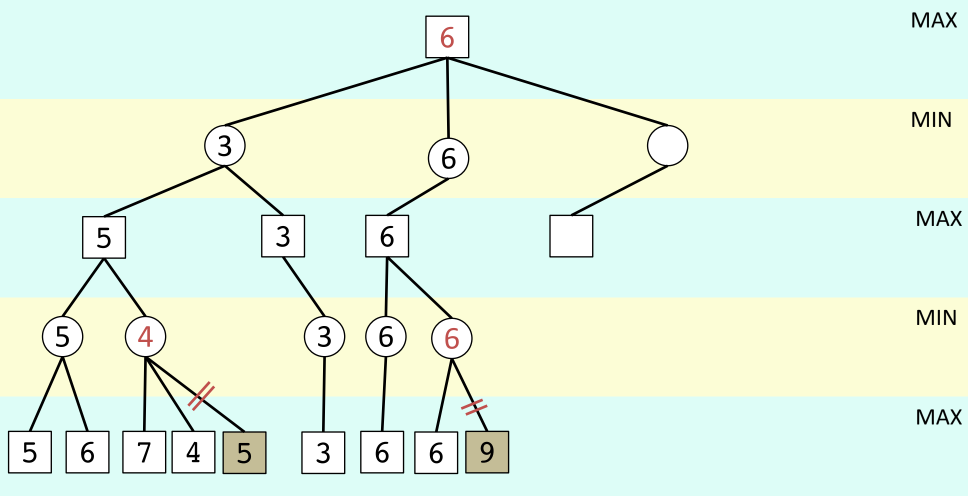
This value will never now be greater than 6, regardless of any other children below it.

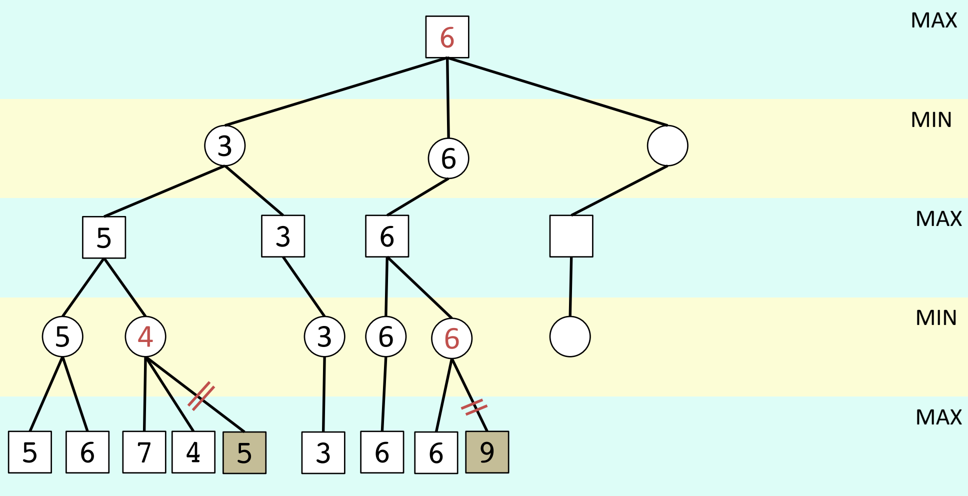

MAX
MIN
MAX
MIN
MAX
3
3
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
So, this node will always choose the left branch (tied value).

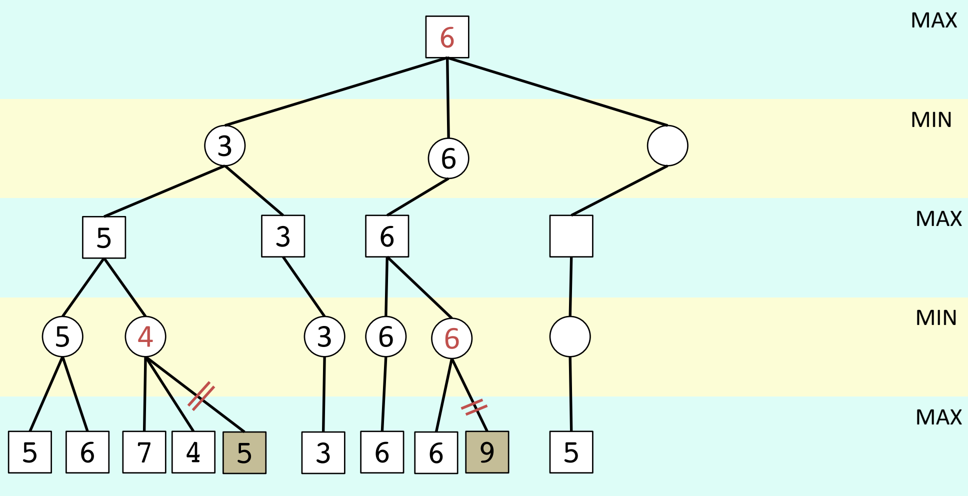

MAX
MIN
MAX
MIN
MAX
3
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9

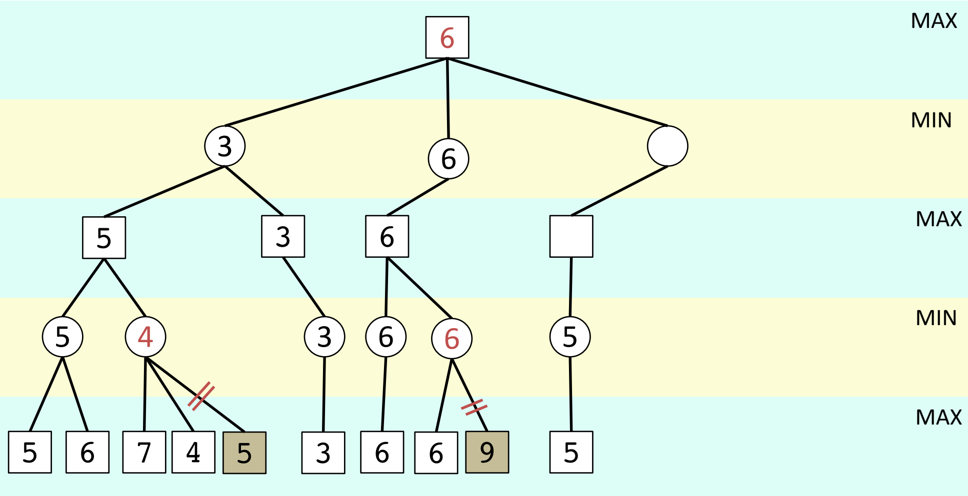

MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9

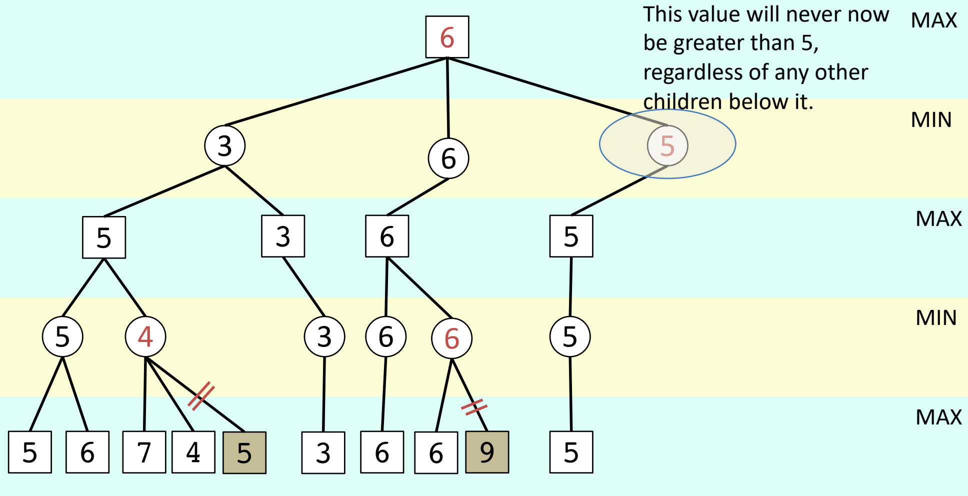

MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9


MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9


MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9

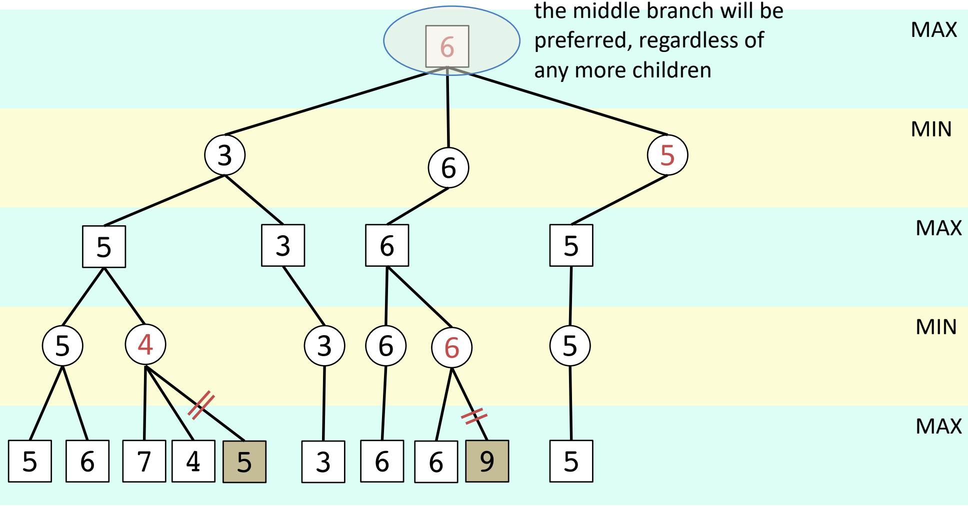

MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
6
7
4
5
3
6
6
9
5

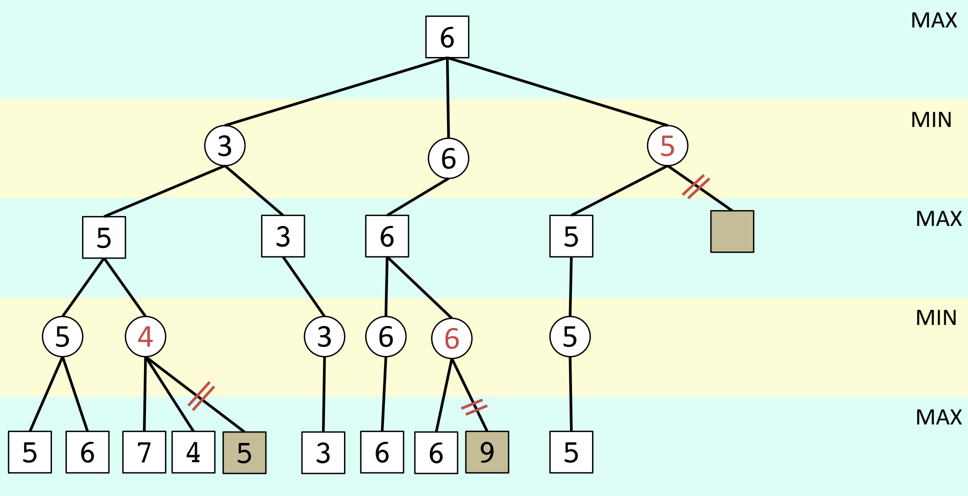

MAX
MIN
MAX
MIN
MAX
6
3
6
5
3
6
5
4
3
6
6
5
5
6
7
4
5
3
6
6
9
5

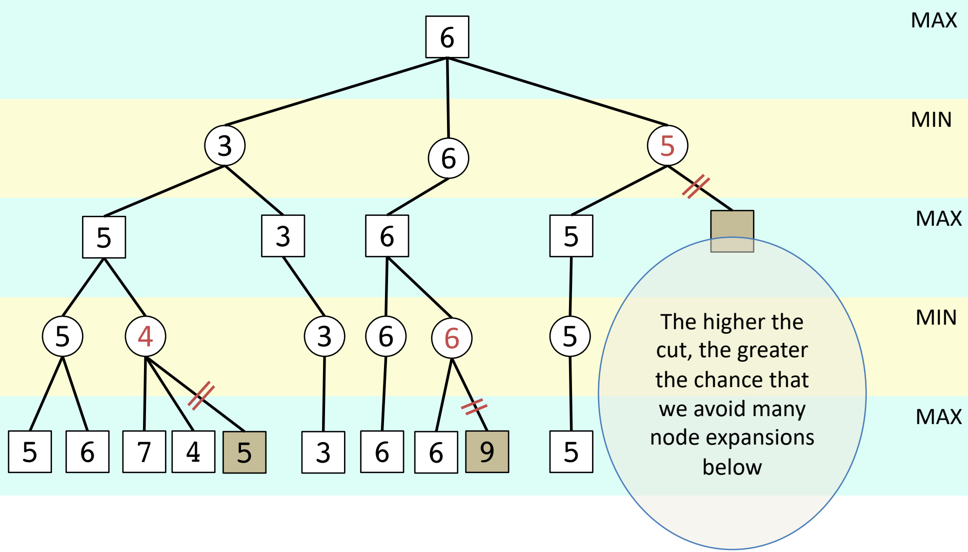


This value will never now be greater than 5, regardless of any other children below it.
MAX
MIN
MAX
MIN
MAX
6
3
6
5
5
3
6
5
5
4
3
6
6
5
5
6
7
4
5
3
6
6
9
5

So, we now know that the middle branch will be preferred, regardless of any more children




MAX
MIN
MAX
MIN
MAX
6
3
6
5
5
3
6
5
5
4
3
6
6
5
5
6
7
4
5
3
6
6
9
5


MAX
MIN
MAX
MIN
MAX
6
3
6
5
5
3
6
5
5
4
3
6
6
5
5
6
7
4
5
3
6
6
9
5
The higher the cut, the greater the chance that we avoid many node expansions below

```
function alphabeta(node, depth, α, β, maximizingPlayer)
     if depth = 0 or node is a terminal node
         return the heuristic value of node
     if maximizingPlayer
         v := -∞
         for each child of node
             v := max(v, alphabeta(child, depth – 1, α, β, FALSE))
             α := max(α, v)
             if β ≤ α
                 break (* β cut-off *)
         return v
     else
         v := ∞
         for each child of node
             v := min(v, alphabeta(child, depth – 1, α, β, TRUE))
             β := min(β, v)
             if β ≤ α
                 break (* α cut-off *)
         return v
```