INF 4130
September 20, 2016

- Today:
  - First hour:

    Ch. 23.5: Game trees and strategies for two-player games.
  - Second hour:

    Rune Djurhuus: About chess-playing programs.

    (His slides will be posted on the course web page.)
- Next week:
  - Petter Kristiansen: Implementation of priority queues
  - Torbjørn Rognes: On algorithms that are used in bio-informatics (e.g. searching in gene-sequences)

Ch. 23.5: Games, game trees and strategies

- We looked at «one player games» (= search) earlier, and their decision trees in Ch 23 (from start to 23.4).
  - This is search for a goal node that everybody agrees is «good».
- Then you can e.g. use A*-search. One can e.g. use it for:
  - Solve the 15-puzzle from a given position.
  - Find the shortest path between nodes in a graph (better than plain Dijksta)

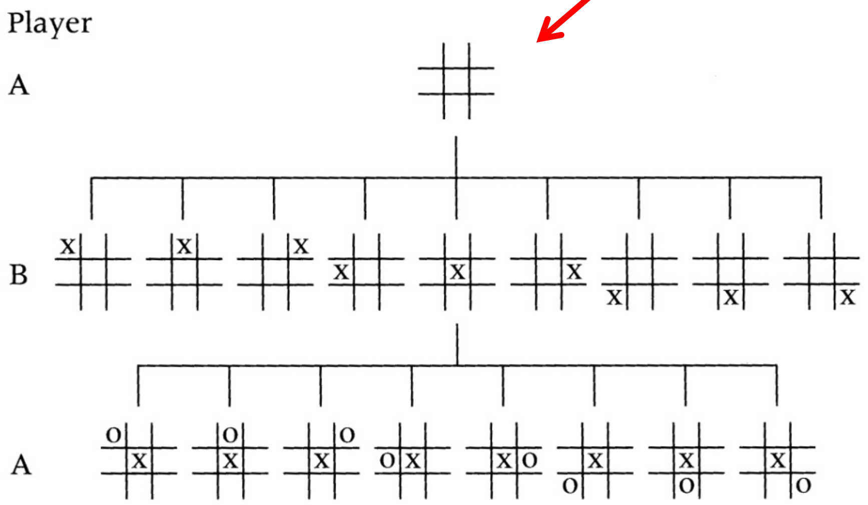
BUT:

- When two players are playing *against* each other, things get very different. What is *good* for one player is *bad* for the other.
  - The trees of possible plays are often enormous. For chess it is estimated to have 10^{100} nodes, and can therefore never (?) be searched exhaustively!
- We only look at *zero-sum games*:
  - The quality of a situation is represented by numbers.
  - The sum of A's evaluation and B's evaluation of a situation is always zero.
  - Then: What one player gains in a move is lost by the other.

Example: Tic-tac-toe and game trees

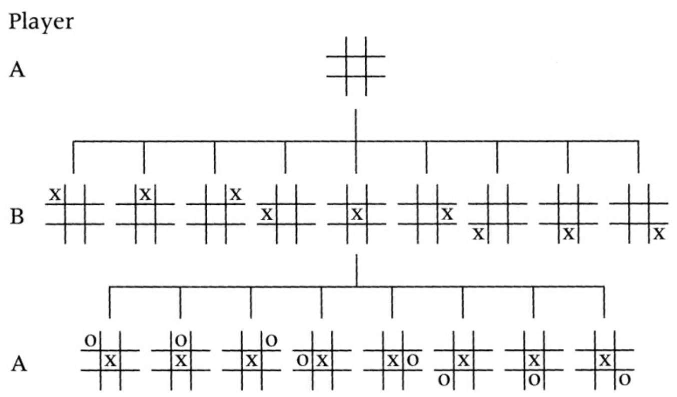
- The board has 3 x 3 squares.
- The game: Alternately do the moves:
  - *Player A* chooses an unused square and writes 'x' in it,
  - *Player B* does the same, but writes 'o'.
- When a player has three-in-a-row, he/she has won.

The start node of the *game tree* for «tic-tac-toe».



- Player A (always) starts
  - And we will here do *all our considerations* from A's point of view.
  - We use numbers for node quality.
  - High numbers are good for A and small numbers are good for B.

Number of nodes in a tic-tac-toe game tree



9 nodes

9*8 = 72 nodes

9*8*7 = 504 nodes

9*8*7*6 = 3024 nodes

9*8*7*6*5 = 15120 nodes

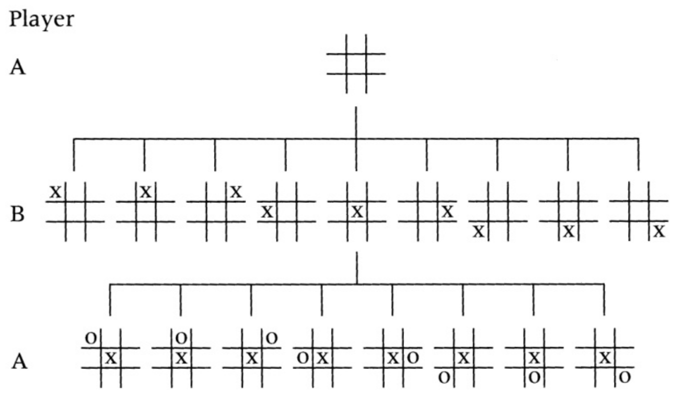
……

9*8*7*6*5*4*3*2*1 = 9! ("factorial") = 362 880 nodes

By searching depth-first in this tree, you never need to store more than 9 nodes, but it will take some time to go through all 362 880 nodes (and for "interesting" games there are usually a lot more!).

But if we represent each game position only once …

(also usable for «one player games»)



1 node	1 node
9 nodes	9 nodes
9*8 = 72 nodes	72 **different** nodes
9*8*7 = 504 nodes	252 **different** nodes
9*8*7*6 = 3024 nodes	756 **different** nodes
9*8*7*6*5 = 15120 nodes	1260 **different** nodes
9*8*7*6*5* 4 = 60480 nodes	1680 **different** nodes
……	……
9*8*7*6*5*4*3*2*1 = 362 880 nodes	126 **different** nodes = $\binom{9}{4}$

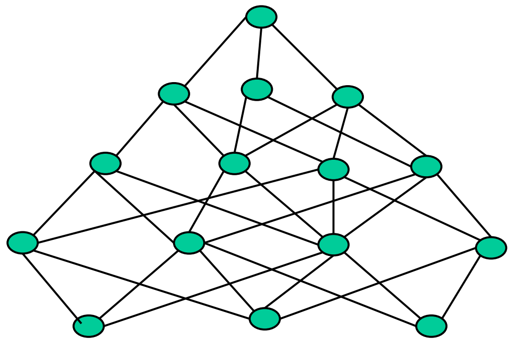
This usually requires a lot of memory!

BUT: In some games you can gain a lot by recognizing equal nodes, and not repeat the analysis for these

(see next slide).

But if we represent each game position only once …
(also usable for «one player games»)

1 node	1 node
9 nodes	9 nodes
9*8 = 72 nodes	72 **different** nodes
9*8*7 = 504 nodes	252 **different** nodes
9*8*7*6 = 3024 nodes	756 **different** nodes
9*8*7*6*5 = 15120 nodes	1260 **different** nodes
9*8*7*6*5* 4 = 60480 nodes	1680 **different** nodes
……	……
9*8*7*6*5*4*3*2*1 = 362 880 nodes	126 **different** nodes



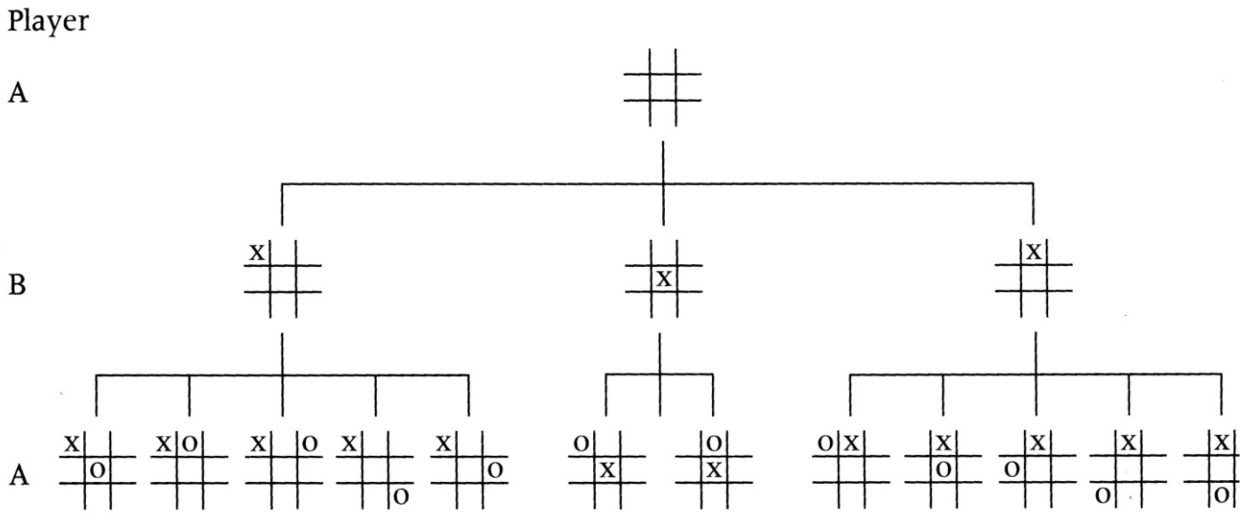
Sketch of a collapsed tree (a DAG)

This usually requires a lot of memory!

BUT: In some games you can gain a lot by reconizing equal nodes, and not repeat the analysis for these (this is somewhat like dynamic programming). In the above simple game we never need more than 1680 nodes.

Representing symmetric solutions by one node
(also usable for «one player games»)

- One can also gain a lot by looking at symmetries:
  - Represent positions that are symmetries of each other only once .
  - Tic-tac-toe: Symmetric solutions will always be at the same depth, but this is *not* generally the case!
- Using this will often reduce the memory needs even further!
  - But in e.g. chess there are few symmetries to utilize.



Zero-sum games

- Seen from A, high values are good, and low ones are bad
- For B the opposite is true
- We will (for the time being) look at the values as seen form A!

A "strategy" for A means:

"A rule telling A what to do in all possible "A-situations".

Aim: We will look for a strategy (if one exists) so that A is sure to win

- "Fully analyzable games" means: The full tree can be analyzed
  - Then there are three possibilities for each A-situation S:

1. A has a strategy from S, so that it will win whatever B does, and chooses its move from S according to that (score: +1 for A)
2. Whatever A does from S, B has a winning strategy from the new situation (score: -1 for A).
3. If A and B both play perfectly, it will end in a tie (score: 0 for A)
   - This can occur only for some games.
   - The game tic-tac-toe ends in a tie if both players play perfectly.

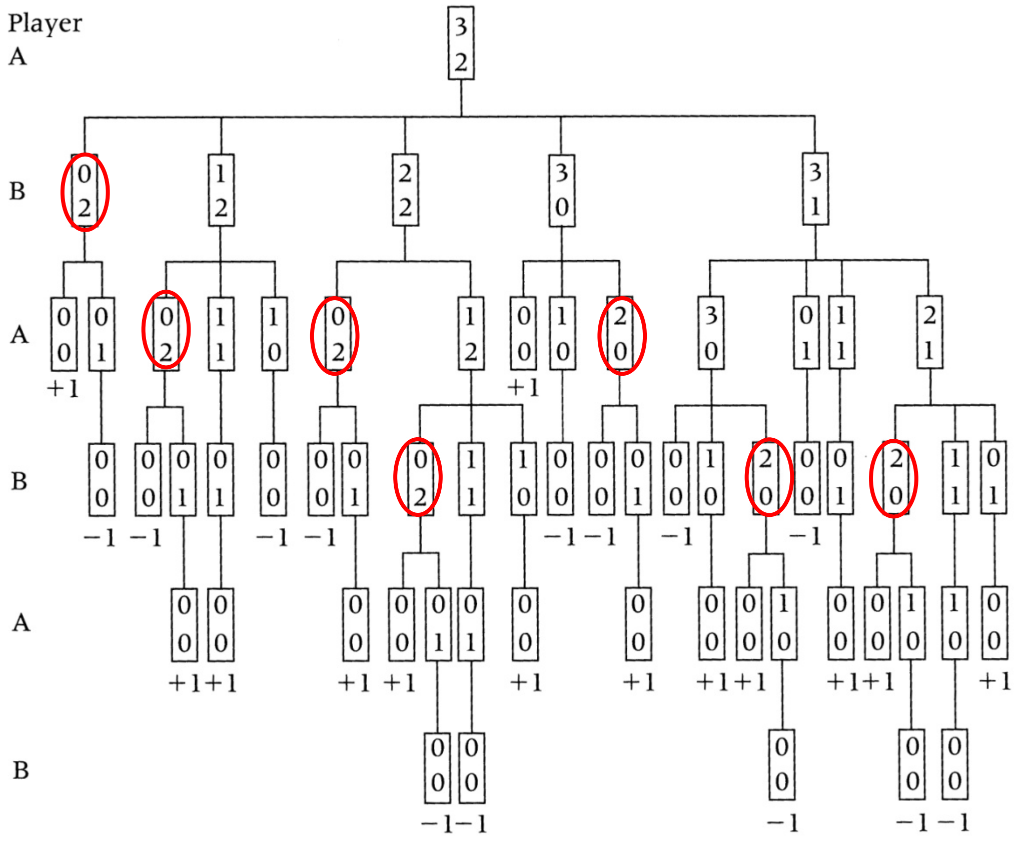
Another example: The game **Nim**

The game **Nim**:

- We start with two (or more) piles of sticks.
- Number of sticks: *m* and *n*.
- One player can take any number of sticks from one pile, but have to take at least 1.
- The player taking the last stick has lost.

- Nim will never end in a tie.
- With $m = 3$ and $n = 2$, the full game tree (utilizing some symmetries) is shown to the right.
- The value seen from A is indicated for the final situations (leaf nodes).
- Next problem: What is the value of the rest of the nodes?

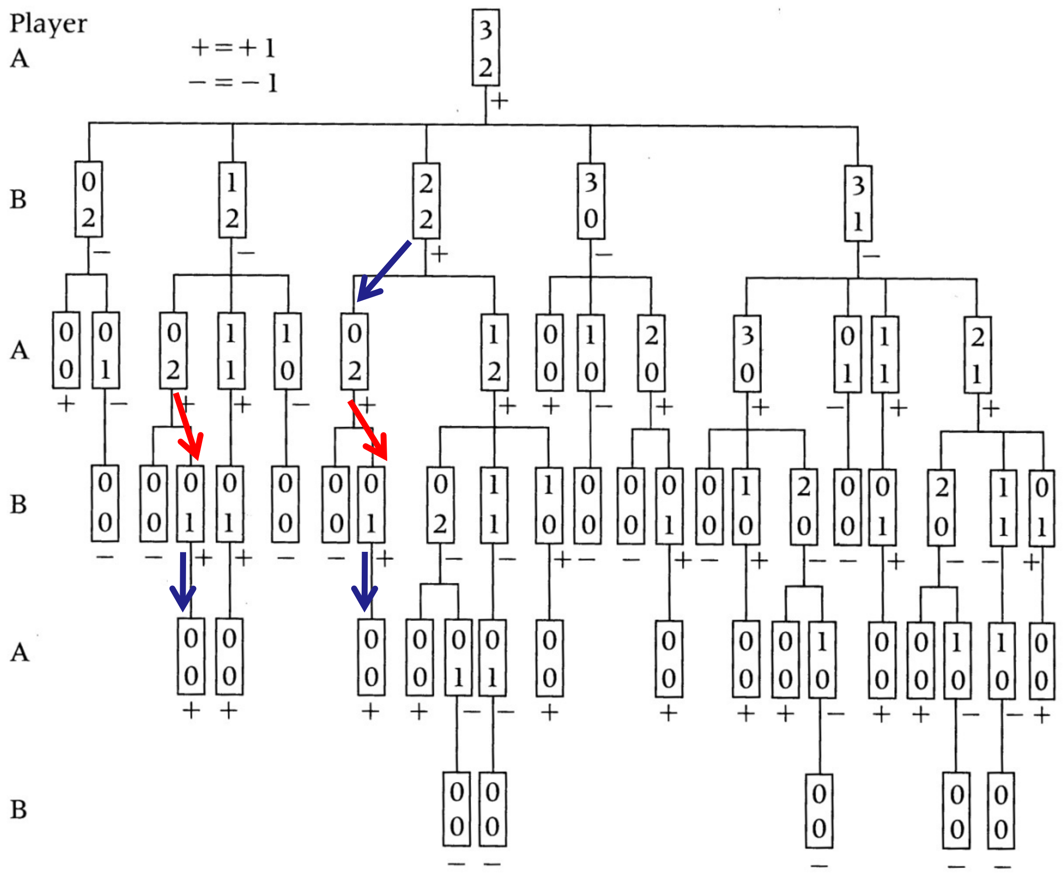
Here *m=3* and *n=2*



NB: We could reduce the number of separate nodes further by recognizing equivalent nodes (see **red circles** above)

How can we find a strategy so that A wins?

Or prove that no such strategy exists.



Strategy for A: If possible, move to a node with value +1. Otherwise make a random move.

Strategy for B: If possible, move to a node with value -1. Otherwise make a random move.

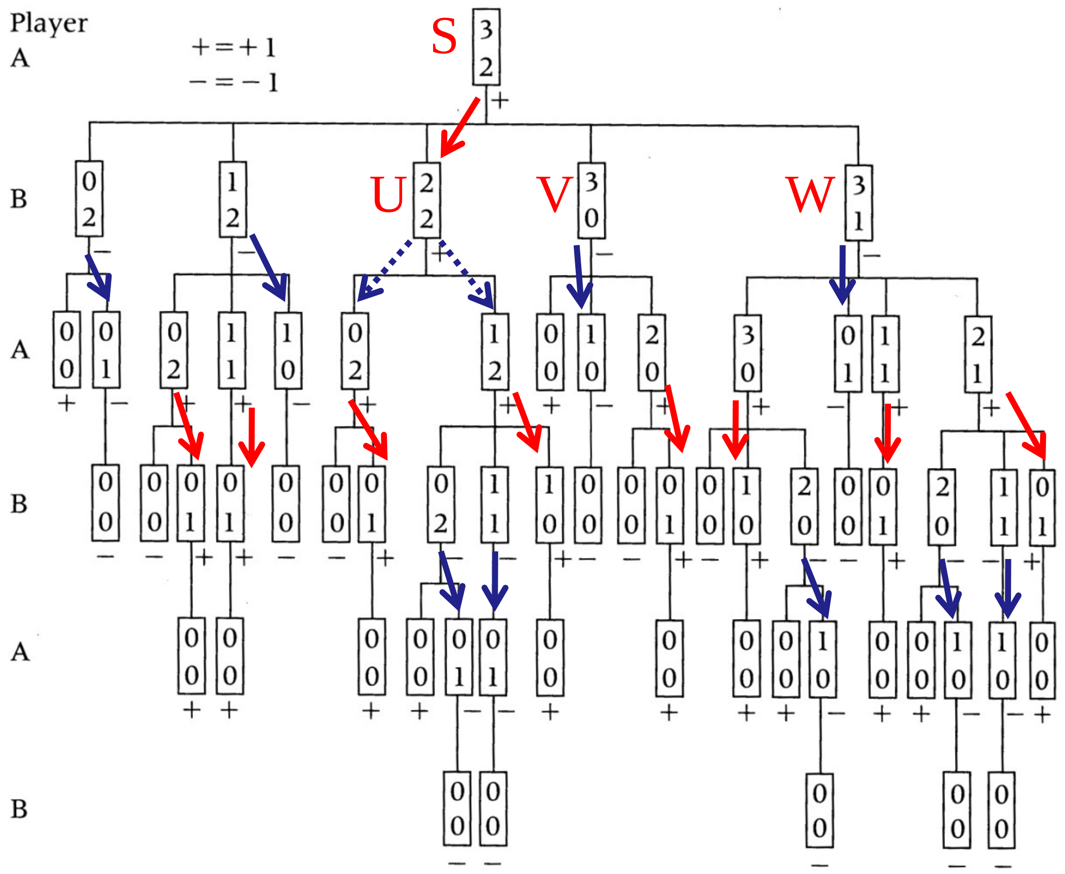
- A wants to find an optimal move.
- We must assume that also B will do optimal moves seen from its point of view.
- Since the values are as seen from A, B will move to the subnode with *smallest* value,

Min-Max Strategy:

- To compute the value of a node, we have to know the values of all the subnodes.
- This can be done by a depth first search, computing node values during the withdrawal (**postfix**).

The Min-Max-Algorithm in action

With simple alpha-beta cutoff (pruning)



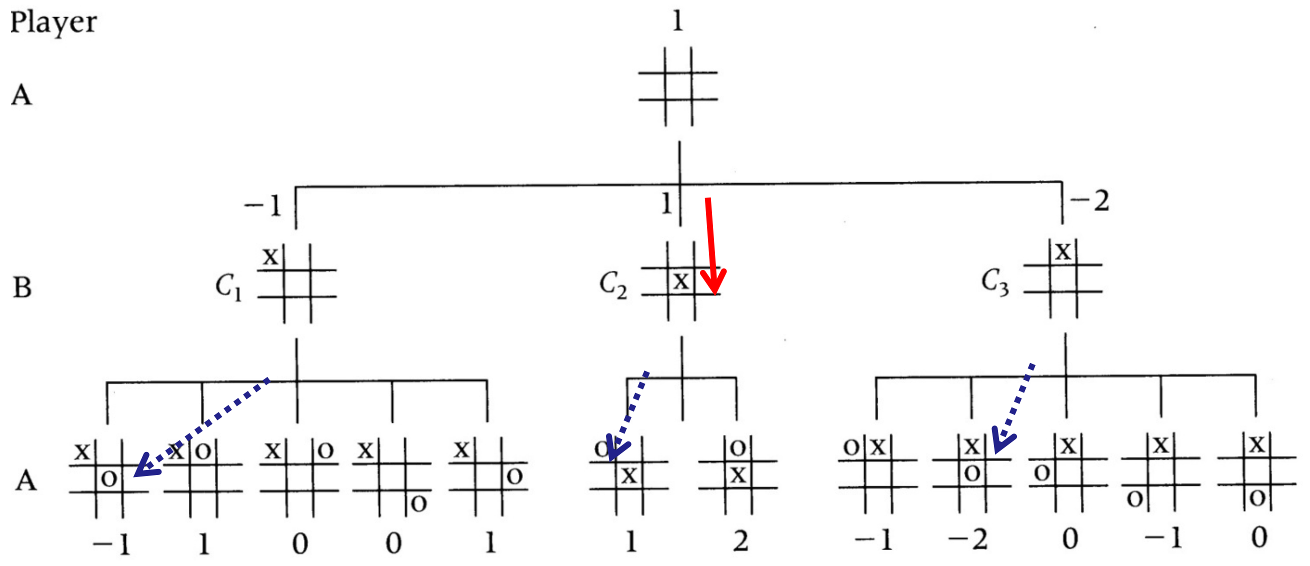
- **Red arrows:** Good move from winning situations for A
- **Blue arrows:** Good move from winning situations for B

- Previous slide: This is done by a deph first traversal of the game tree, computing values on withdrawal (that is postfix)
- The result of this is given in the figure to the left as + and -.

Possible optimalization:

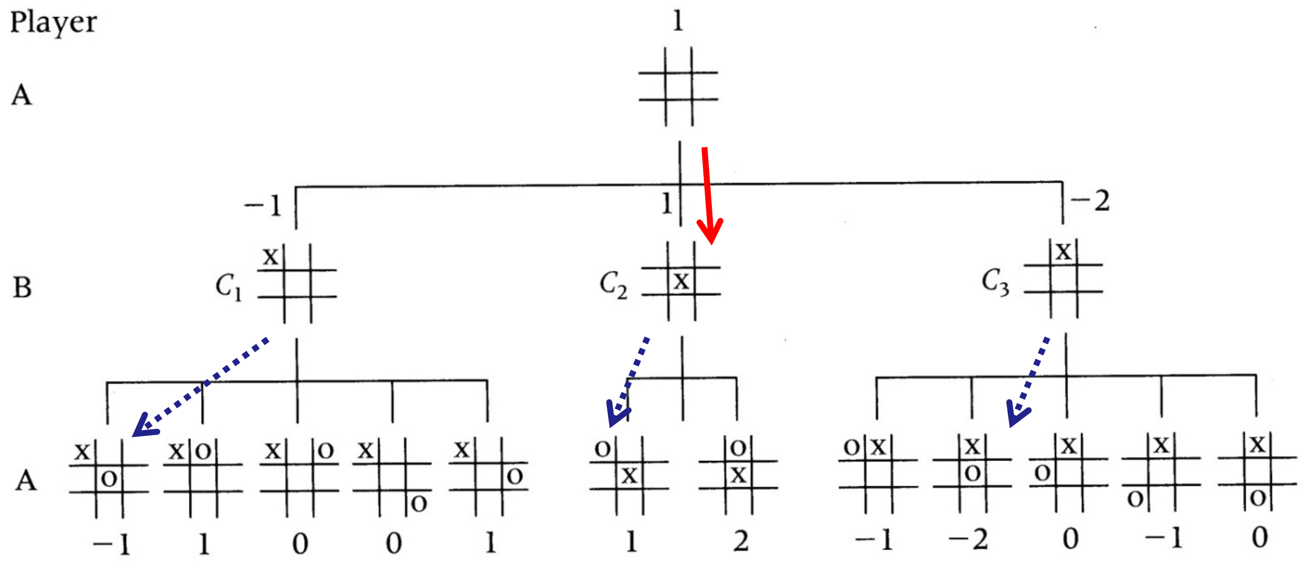
- From the start-position S, assume that A has looked at three of its subtrees (from the left). A has then found a winning node U (marked +1). *Then the value of V and W does not matter.*
- This is a simple version of *alpha-beta cutoff (pruning)*

Usually, the game tree is too large to traverse



- One then usually searches to a certain depth, and then estimate (with some heuristic function) how good the situation is for A at the nodes at that depth. We then usually also use other numbers that -1, 0 and +1.
- In the figure above we go to depth 2.
- The heuristic function above is: Number of «winning lines for A» minus the same for B (this is given below each leaf nodes).
- The best move for A from the start position is therefore (according to this heuristic) to go to C_2.

Usually, the game tree is too large to traverse



However, this heuristic is not good later on in the game. It does not take into account that winning is better than any heuristic. We therefore, in addition, give winning nodes the value +∞ (and here 9 is fine).

This will give quite a good strategy. But, as said above: tic-tac-toe will end in a tie if both players play perfectly.

o	o	x
x	x	o
o	x	x

We have to add that the tie-situation (e.g. the one to the right) gets the value 0. Thus, if we fully analyze the game, the value of the root node will be 0.

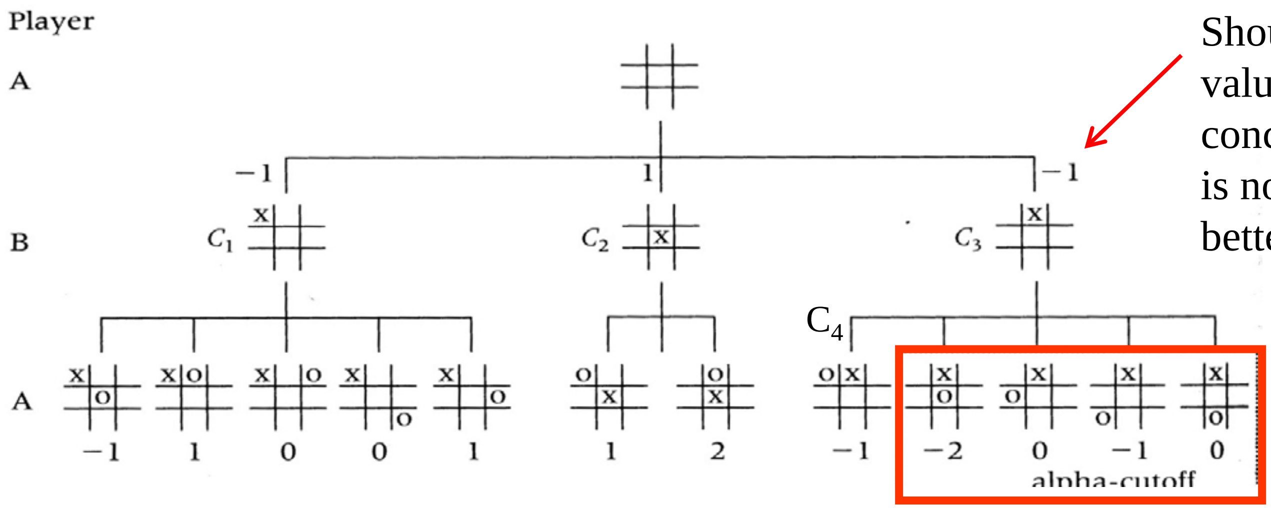
NOTE: The difficult choice for a game-programmer is between searching *very deep* or using a *good, but time consuming*, heuristic function!

Alpha-beta cutoff (pruning)

This technique is only usable for two-player games!

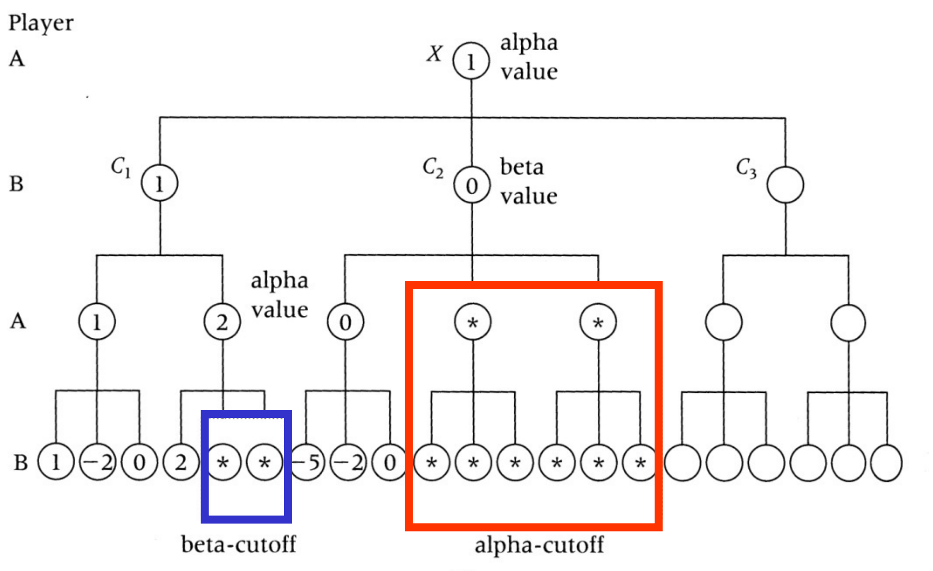
Intuitively Alpha-beta-cutoff goes as follows (assuming it is A's move):

- A will consider all the possible moves from the current situation, one after the other
- A has already seen a move in which it can obtain the value u (after C_1 og C_2, u = 1)
- A looks at the next potential move, which leads to situation C_3
- However, A then observes that from C_3, *B has a very good move* (= bad for A) that seen from A has the value v = -1. Then the value of C_3 cannot be better than -1 (independent of what the other subtrees of C_3 gives (as B will minimize at C_3).
- As v < u, player A has no interest in looking for even better moves for B from situation C_3. A already knows that it has a better move than to C_3, which is to C_2.

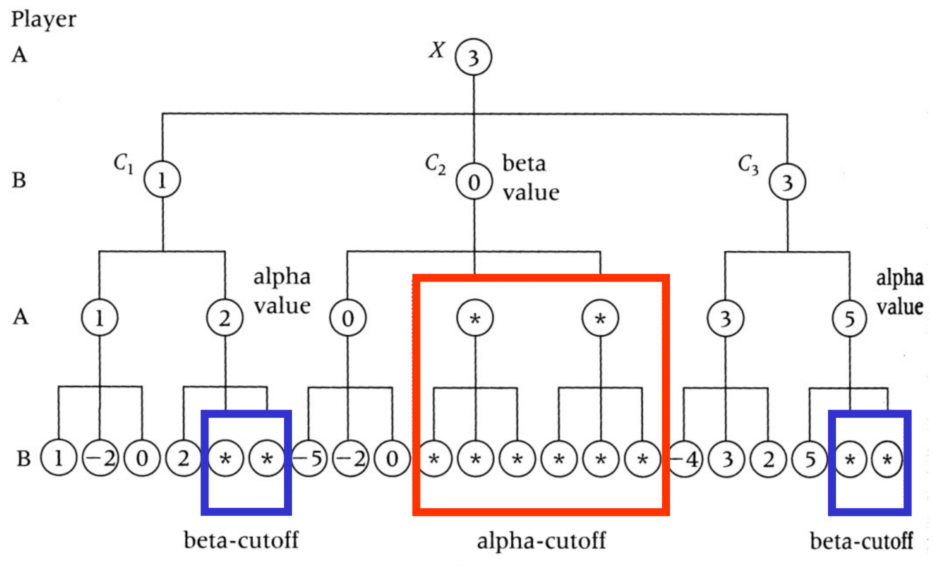


Should have become -2, but value -1 is enough for A to conclude that a move to C_3 is not the best (to C_2 is better)

Examples showing alpha-beta cutoff



(a)



(b)

- When A considers the next move:
  - Cutoffs from A-situations is called alpha-cutoffs.
  - Corresponding cutoffs from B-situations are called beta-cutoffs.
- The figures to the left shows alpha- and beta-cutoffs at different stages of a DF-search of a game tree.

- When implementing alpha-beta-cutoffs during a DF-search, it is usual to switch viewpoints between the levels.
  - Then we can always *maximize* the value.
  - But we have to negate all values for each new level.
- Such an implementation is given at the next slide.

Alpha-beta-search (negating the values for each level)

```
real function ABNodeValue (
    X,          // The node we compute alpha/beta value for. Children: C[1],C[2]… C[k]
    numLev,     // Number of levels left
    parentVal,// The alpha-beta-value from the parent node (-LB from the parent)
                // Returned value: The final alpha/beta-value for the node X
{
    real LB;  //  Current Lower Bound for the alpha/beta value of this node (X)

    if <X is a terminal node> or numLev = 0 then {
        return <An estimate of the quality of the situation (the heuristic)>;
    } else {
        LB := - ABNodeValue(C[1], NumLev-1, ∞);
        for i :=  2 to k do {
            if  LB >= parentValue  then {
                return LB;                   // Cutoff, no further calculation
            }
            else {
                LB := max(LB, - ABNodeValue(C[i], Numlev-1, - LB) );
            }
        }
    }
    return LB;
}
```

Start the recursive call to calculate value for the (current) rootnode (down to depth 10) by calling
`ABNodeValue(rootnode, 10, -∞)`

Misprints in the textbook

- There are some simple misprints in the program at page 741 in the textbook (probably not corrected in any edition):
  - ”AB” is missing in the name of the procedure in the recursive call.
  - A right parenthesis is missing at the end of the line where **max** is called.
- These errors are corrected on the previous slide!