V. Games

Two-player, turn-taking, perfect-informed, finite and deterministic, zero-sum games

- Two players take turns according to given rules until the game is over.
- The game is in a fully observable environment, i.e., the players know completely what both players have done and can do.
- The number of the rules and length of the plays of the game are finite and the effect of each rule is determined uniquely. Outcomes of the game do not depend on chance at all.
- The sum of the payoff values of the players at the end of the game is always zero. (Its special case is when one player wins, the other player necessarily loses. Draw is also possible.)

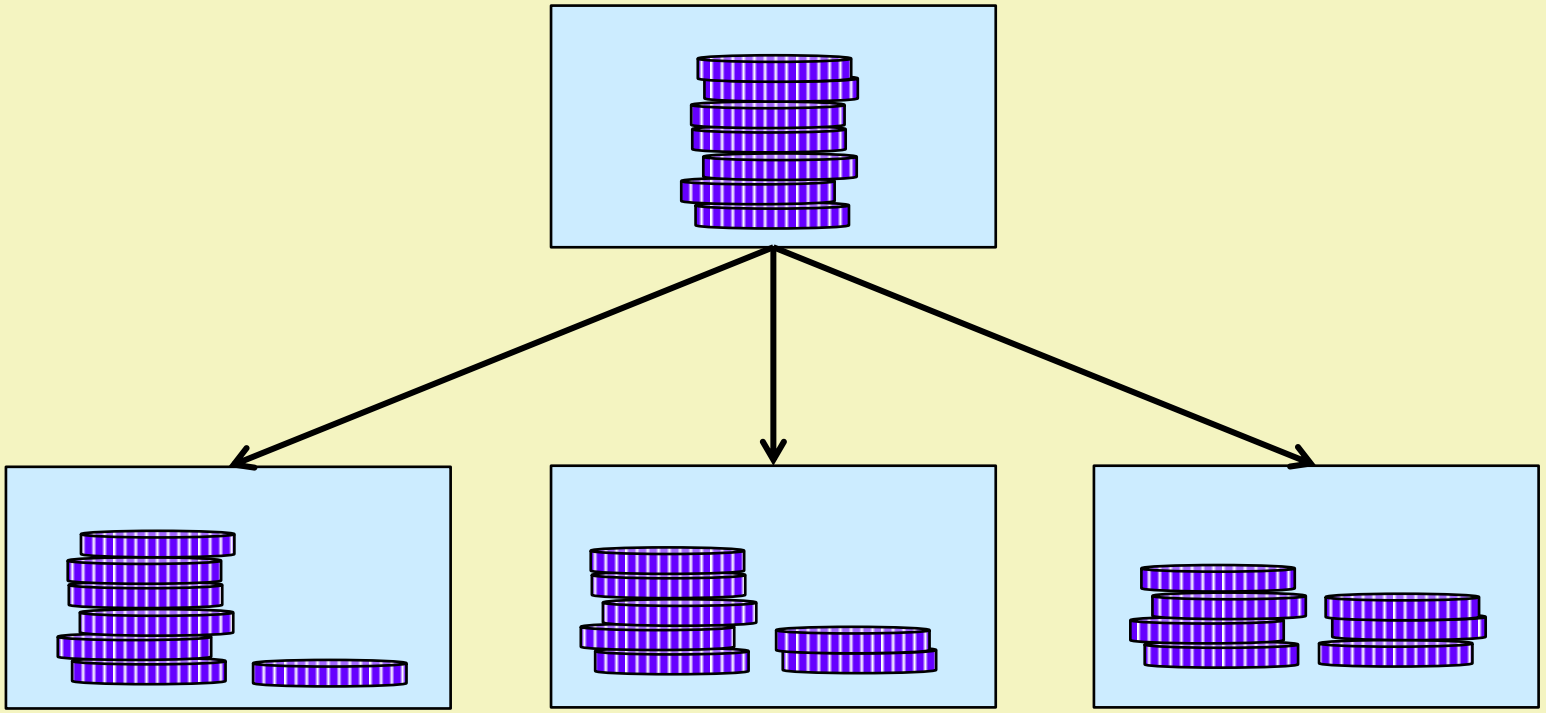
State space representation

- state – configuration + player next to move
- operator – legal move
- initial state – initial configuration + first player
- terminal state – terminal configuration + next player
- payoff functions – both players (*A* and *B*) have got:

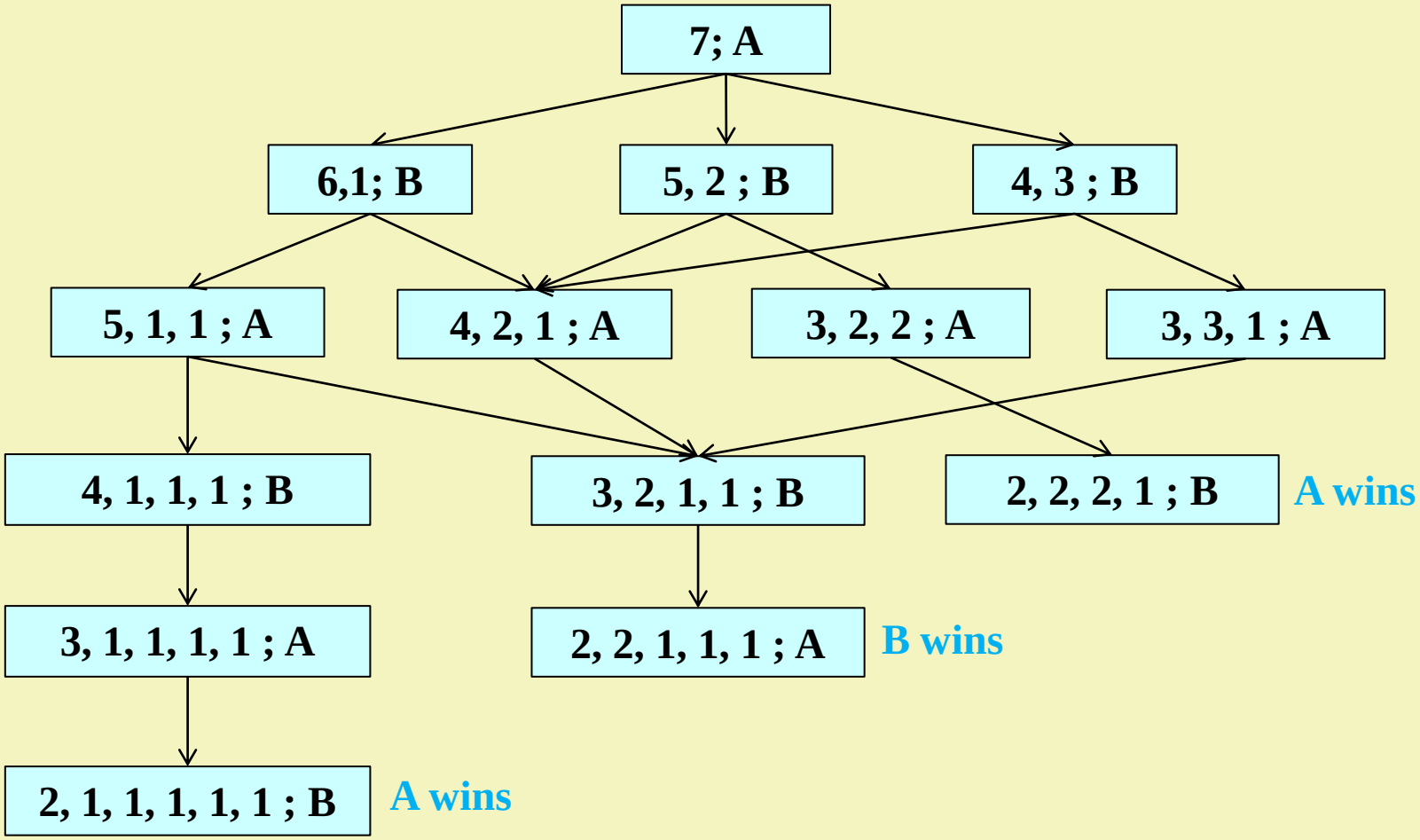
$$p_A, p_B : \text{terminal states} \longrightarrow \mathbb{R}$$

  - In a zero-sum two-player game: $p_A(t) + p_B(t) = 0$
  - In special zero-sum two-player game:
    - $p_A(t) = +1$ if *A* wins at the state *t* (goal state, winning state for *A*)
    - $p_A(t) = -1$ if *A* loses at the state *t* (losing state for *A*)
    - $p_A(t) = 0$ if the state *t* is a draw

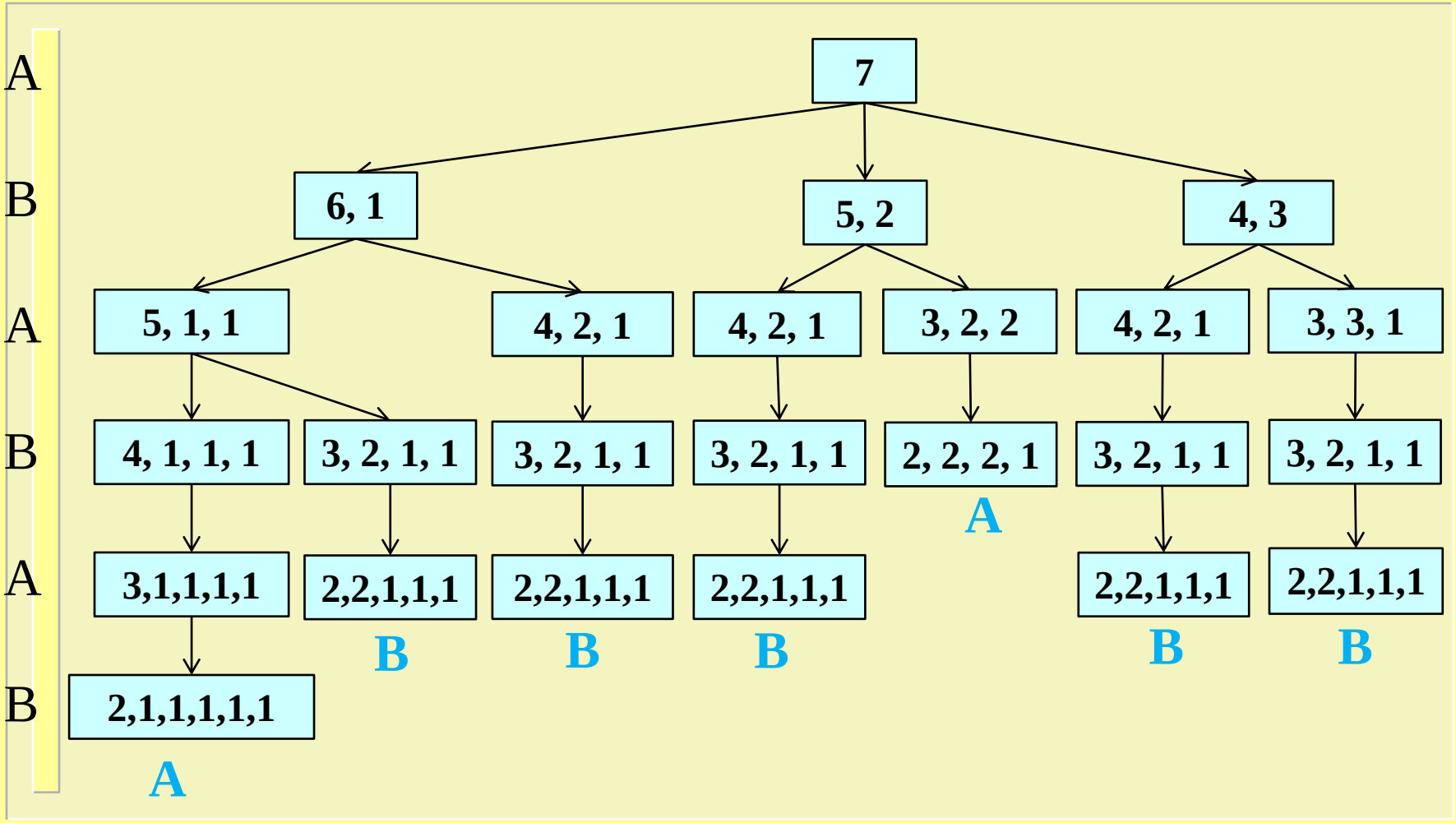
Grundy mum's game



Grundy mum's game graph



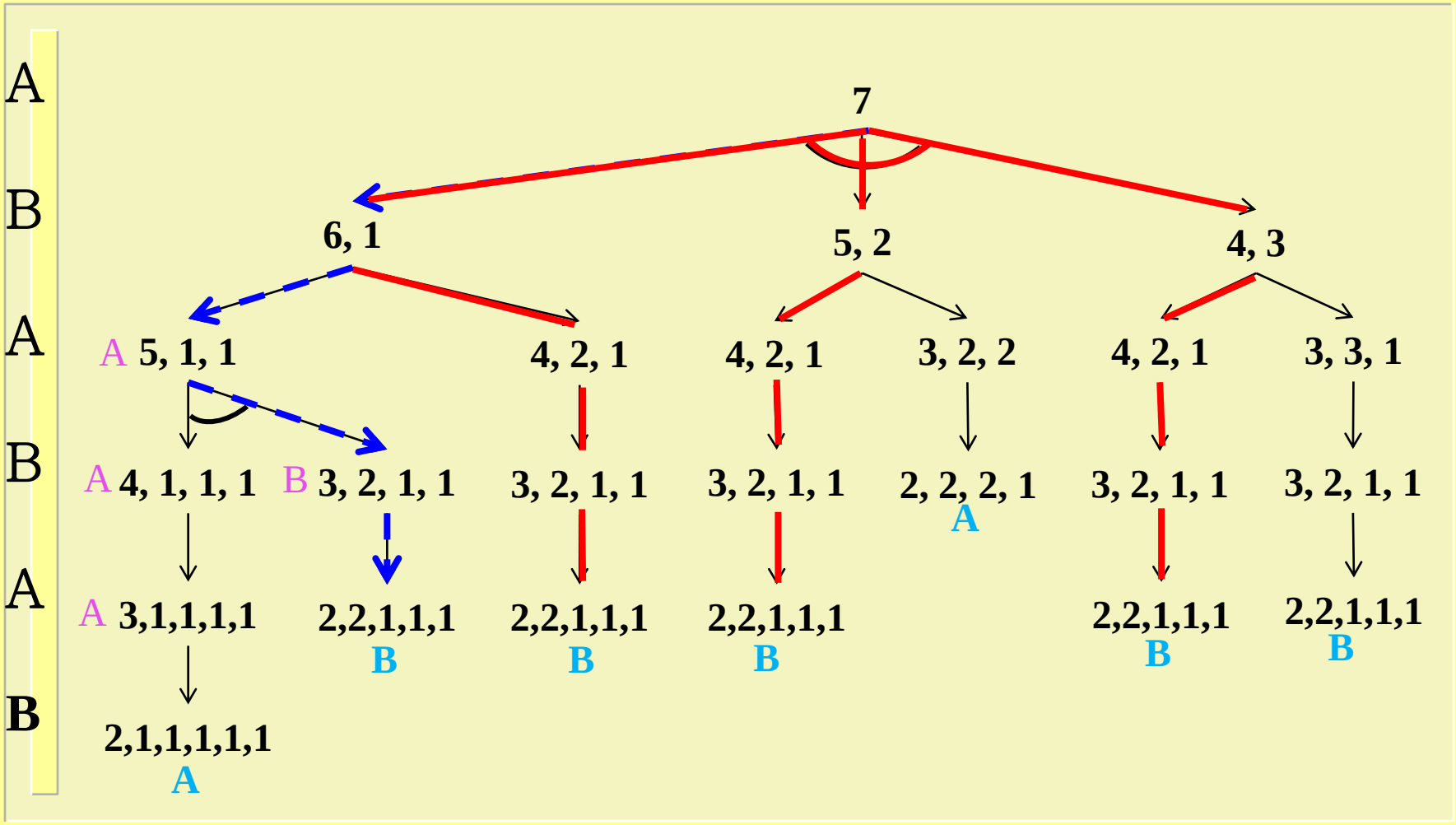
Grundy mum's game tree



Game tree

- node – configuration
  (the same configuration may occur in several nodes)
- level – player (the levels of A and B alternate)
- arc – step (level by level)
- root – start configuration (first player)
- leaf – terminal configuration
- branch – play of the game

*How can the player **B** win?*



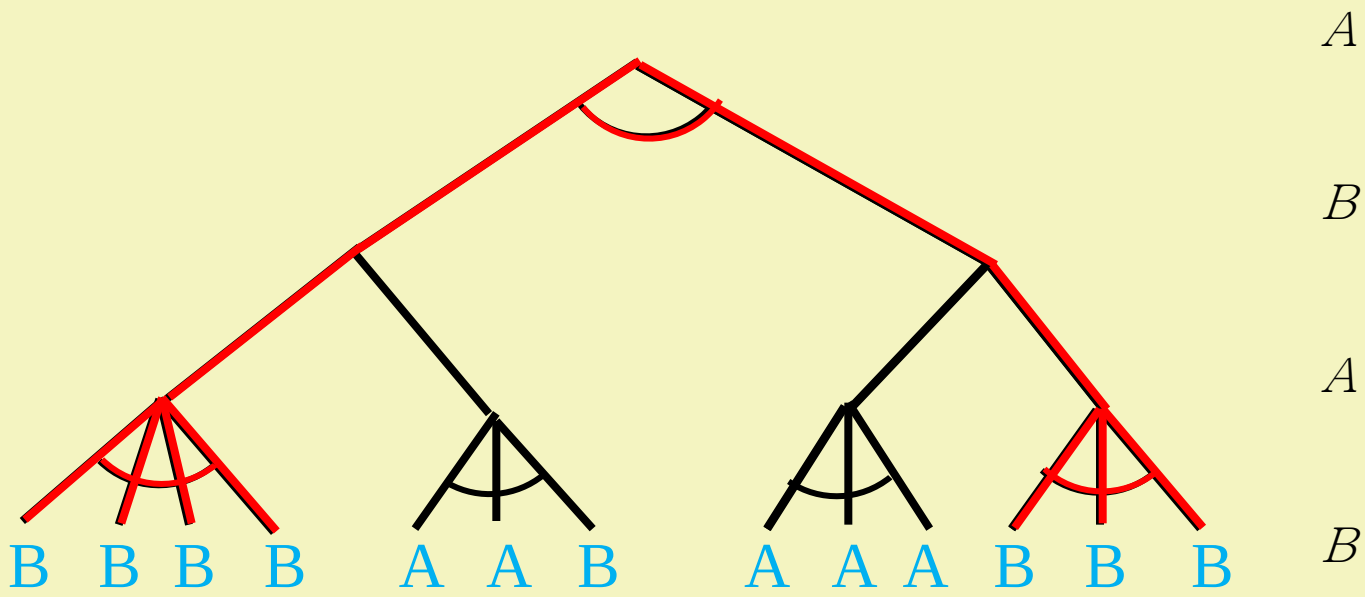
Winning strategy

- The winning strategy (or non-losing strategy) of a player shows his moves that lead to win (or at least draw) no matter what the opposite player does.
- The winning strategy is not one play of the game but a collection of plays leading to win, and one of these plays can be realized by the player who has got winning strategy.

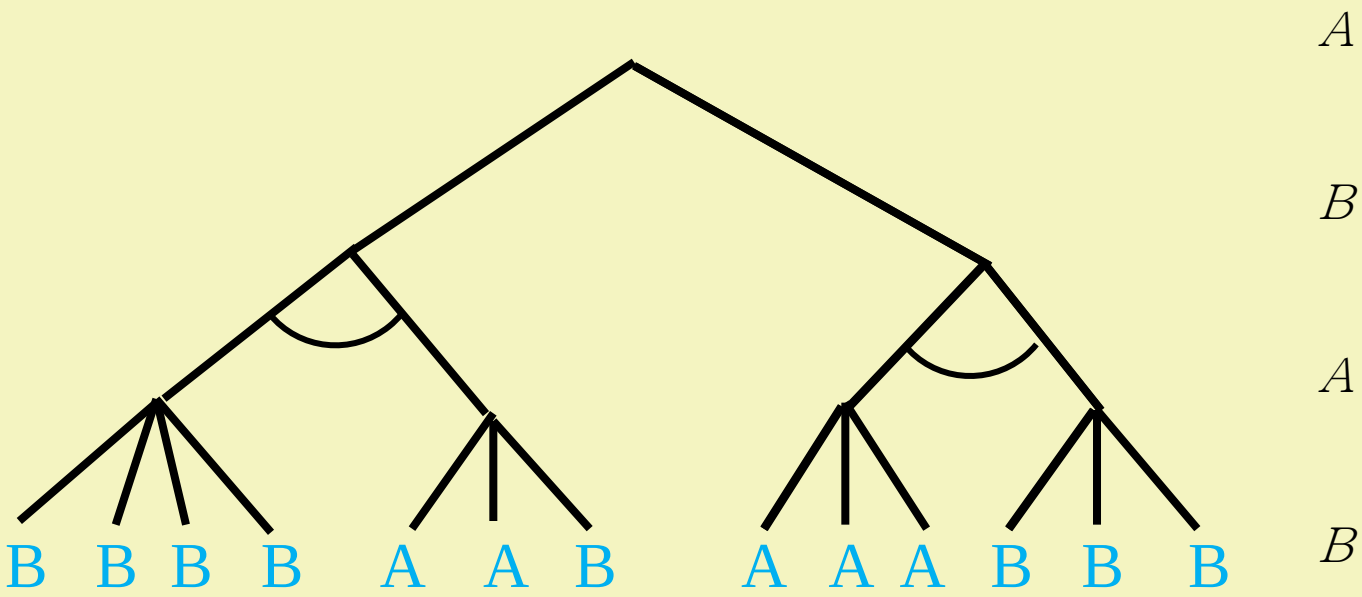
Remarks

- The winning (or non-losing) strategy of one player can be described by a special sub-tree of the AND/OR tree constructing from the point of view of that very player.
- This sub-tree (hyper-path) contains all successors of the node on the opponent's level (AND connection) and only one successor of our level (OR connection) and it leads from the root to winning (non-losing) leaf nodes.
- Searching for a winning strategy is a special hyper-path finding problem.

*Searching for winning strategy in the AND/OR tree of the player **B***



*Searching for winning strategy in the AND/OR tree of the player **A***

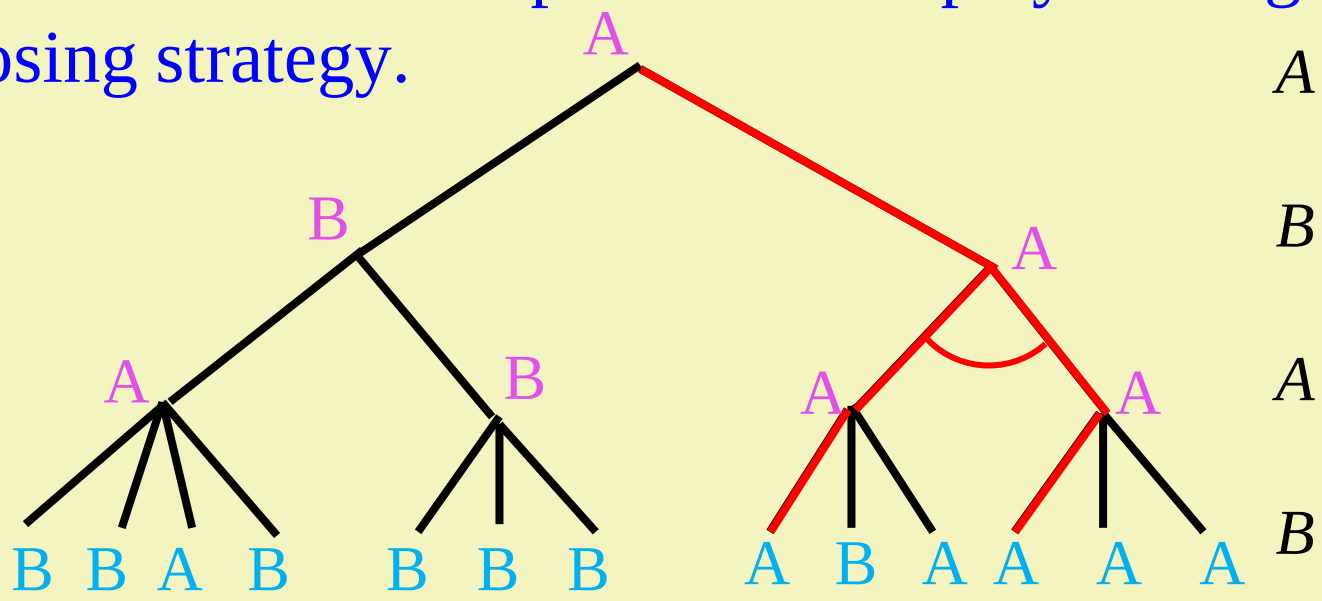


There is no winning strategy for A.

Only one player may have got winning strategy.

Theorem

- In each two-player, perfect-informed, finite and deterministic, zero-sum games where there is no draw one of the players must have got winning strategy.
- In case the draw is also possible: one player has got a non-losing strategy.



Sub-tree evaluation

- Finding the winning strategy is hopeless in the larger game tree.
- Instead of searching for a winning strategy the algorithms are investigated that can suggest a good next step for us.
- These algorithms build up a sub tree of the game tree starting from the current state and try to estimate the beneficial of the leaf nodes of this sub-tree and thereafter calculates our next step based on these values.

Evaluation function

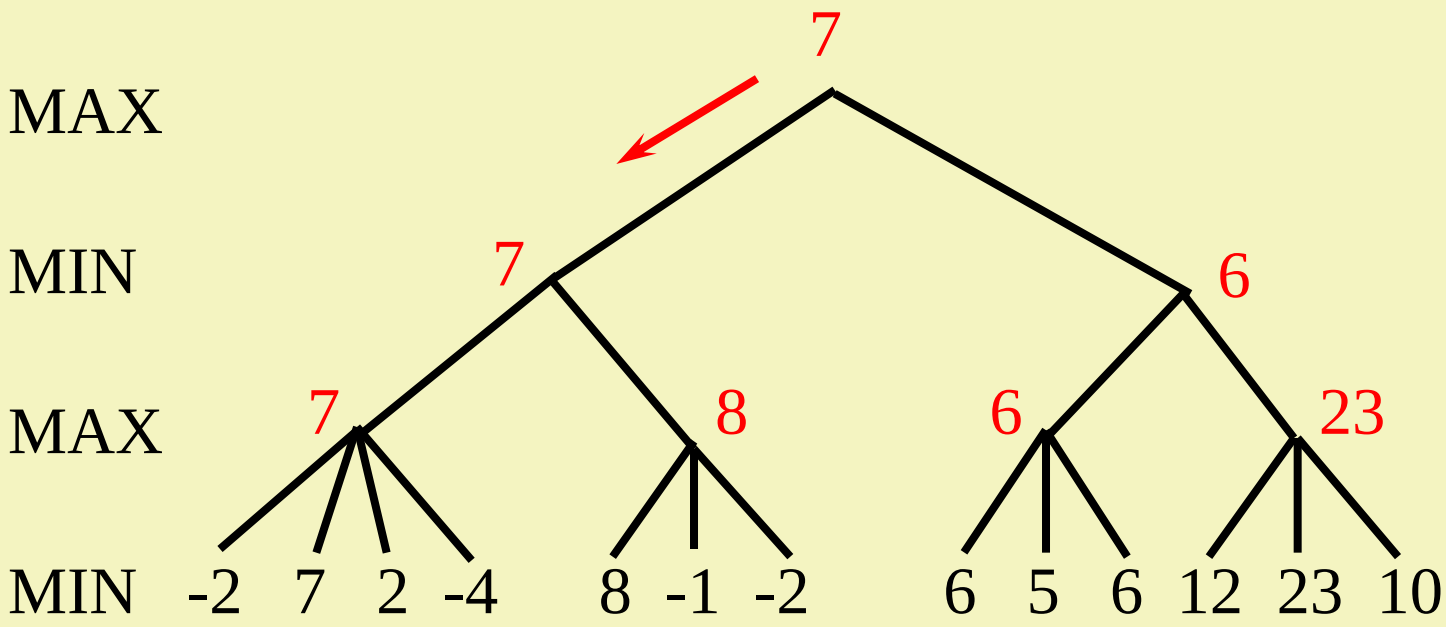
- This is a heuristic function that can measure the beneficial of the states from our point of view against our opponent.

$$f : \text{States} \rightarrow [-1000 .. 1000]$$

The minimax algorithm

1. Starting from the current state, some levels of the game tree are built up (depending on the time or the storage-space limit).
   - Let the player representing us be MAX and our opponent be MIN.
2. The leaves of this sub-tree must be evaluated based on the evaluation function.
3. Values of the inner nodes are computed from their successors upwards level by level. These backed-up values are
   - the maximum of the children on our (MAX) levels
   - the minimum of the children on the opponent's (MIN) levels
4. Our next step will be towards the successor of the current state (this is the root of the sub-tree) which gives up the largest backed-up value.

Example

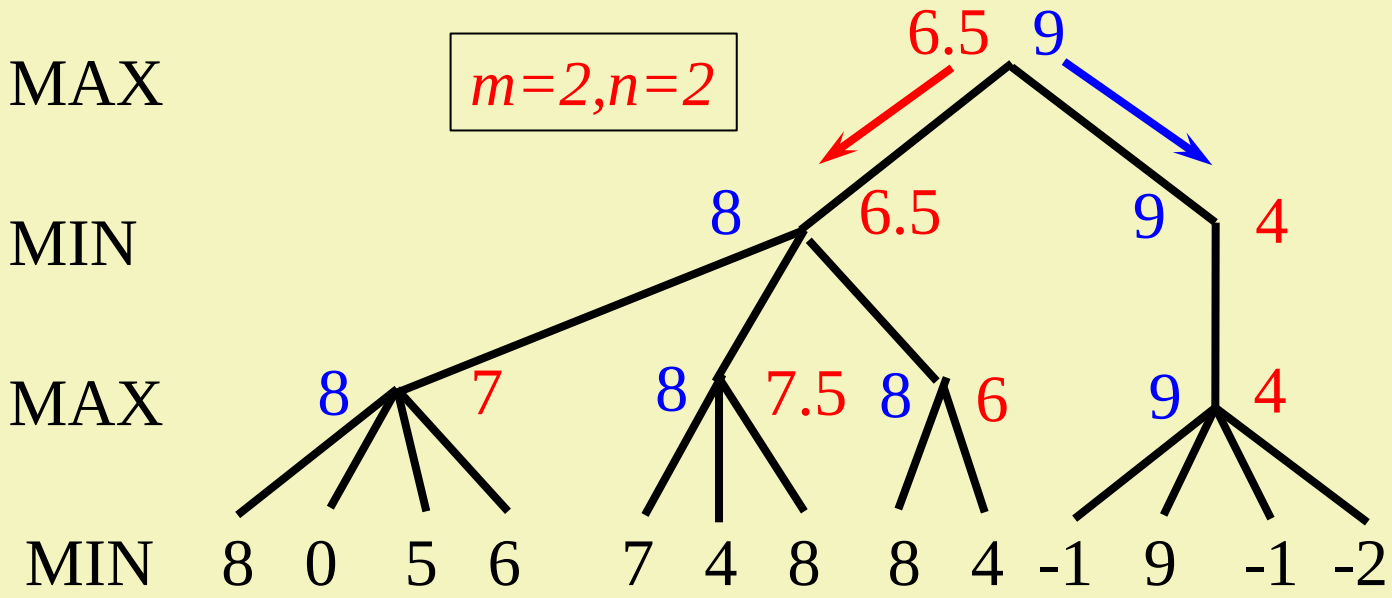


Remark

- We must repeat this algorithm whenever it is our turn to play since our opponent may not move what we expect.
  - He/she can use other depth bound
  - He/she can use other evaluation function
  - He/she can use other algorithm
  - He/she can miss

(m,n) average evaluation

- This method can rectify the miscalculations of the evaluation function. The backed-up values are
  - the average of the *m* largest child values on MAX levels
  - the average of the *n* smallest child values on MIN levels



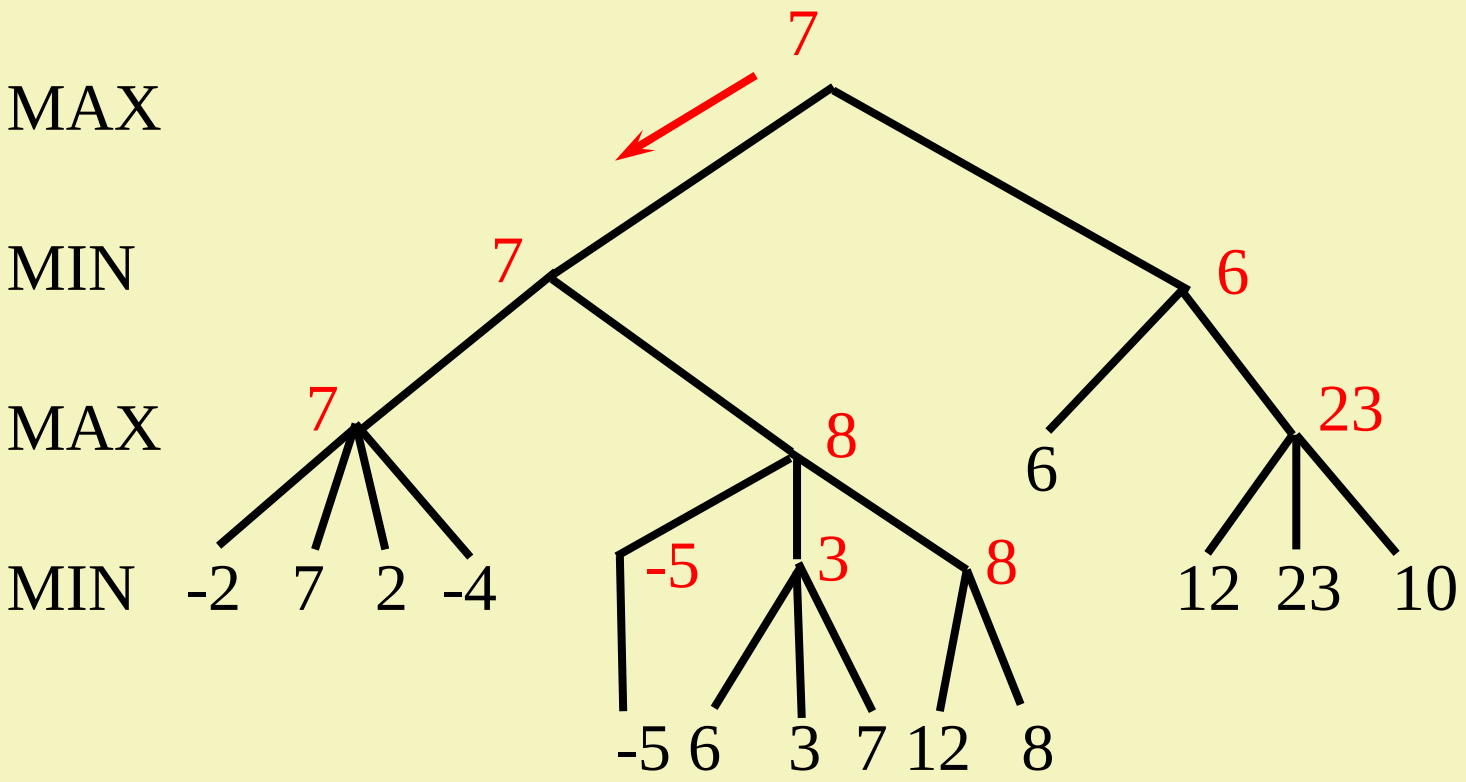
Various depth bound evaluation

- The aim of this method is that the values of the leaf nodes on all branches of the sub-tree show reliable values.
- The evaluation value of a node is unreliable (uncertain) if its child's value significantly differs, i.e., (stationary test)

$$|f(current) - f(child)| > K$$

- In these cases the building of the sub-tree under the given depth must be continued from the unreliable nodes until the stationary test is false.

Example



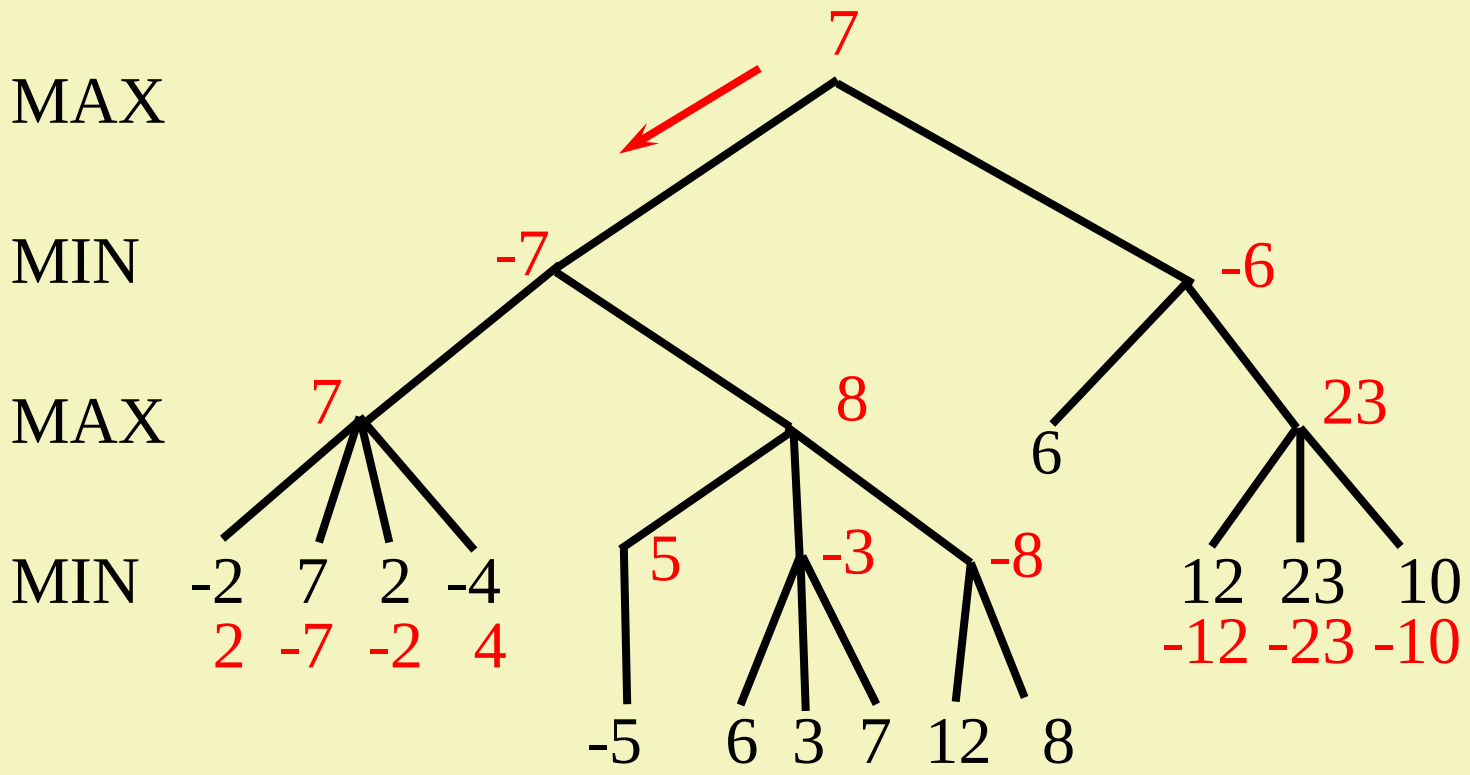
Selecting evaluation

- If the essential moves and the marginal moves can be separated, then it is enough to build up the sub-tree of the game using only the essential moves.
- This idea reduces the memory space of the evaluation.
- This selection needs some heuristics.

Negamax algorithm

- Its implementation is easier than minimax.
  - Initially take the negation of the leaf values on the opposite (MIN) levels.
  - Back up the values upwards level by level:
    $backed\text{-}up\ value = max(-child_1, ..., -child_n)$

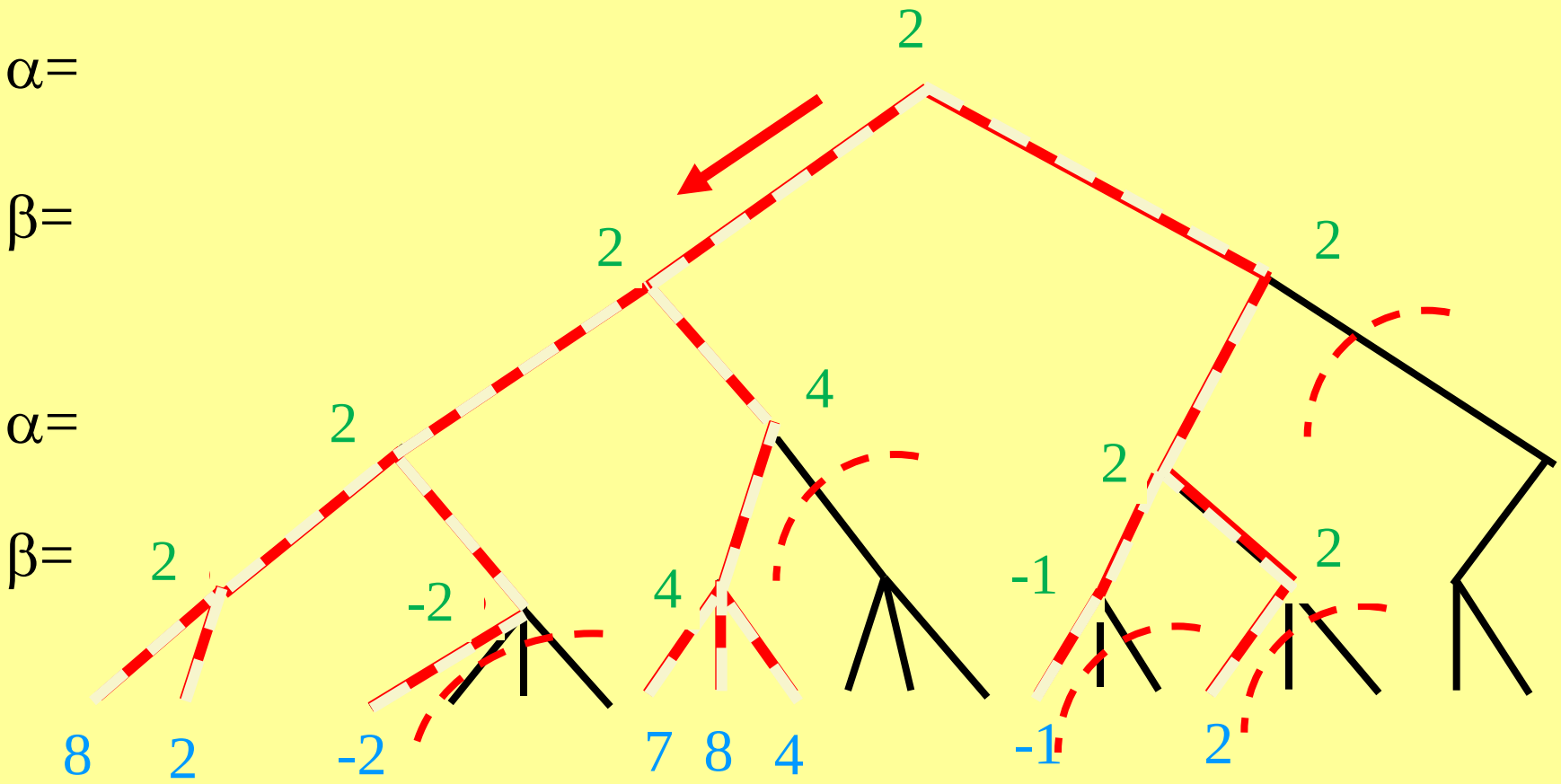
Example


MAX
MIN
MAX
MIN
7
-7
-6
7
8
6
23
-2 7 2 -4
2 -7 -2 4
5
-3
-8
12 23 10
-12 -23 -10
-5 6 3 7 12 8

Alpha-beta algorithm

- It traverses the sub-tree according to the backtracking algorithm.
- The nodes of the current path have got temporary values:
  - on MAX's levels: α value (lower limit),
  - on MIN's levels: β value (higher limit)
- Fore step: $\alpha=-\infty$ and $\beta=+\infty$.
- Back step: $\alpha=\max(\alpha, \text{child})$ or $\beta=\min(\beta, \text{child})$
- Cutting condition: if there are an α and β value on the current path so that $\alpha \geq \beta$.

Example



Discussion

- The result of the alpha-beta pruning is equal to the result of minimax method. (If several equal values run up to the root, the „left most” direction must be chosen.)
- Memory space: only one path.
- Running time: better than minimax because of cutting.
  - Average case: expected value of the number of branches that must be investigated before cutting is *2*
  - Optimal case: in a sub-tree with branching factor *b* with depth *d*, the number of the leaves evaluated: $\sqrt{b^d}$

Two player game software

- Negamax algorithm based on selecting, various depth bound and average evaluation with alpha-beta pruning
- A frame program is needed that accepts the moves of the user and generates the moves of the computer.
- Special features must be built in (initial settings, helps, hints, saving and reloading games etc.)
- Graphical user interface is very important.
- It is not worth anything without good heuristics (in the evaluation function and the selection)