Block 1 – Knowledge Representation and Search

Chapter 6:
Adversarial Search

Chapter 6- Index

1. Games: problem definition
2. Minimax algorithm
3. Alpha-beta pruning
4. Additional refinements

Bibliography

S. Russell, P. Norvig. ***Artificial Intelligence . A modern approach.*** Prentice Hall, 3rd edition, 2010 (Chapter 5) http://aima.cs.berkeley.edu/
N. J. Nilsson**.** ***Artificial Intelligence: A New Synthesis****.* Morgan Kaufmann Publishers, 1998 (Chapter 12)

Alternatively:
S. Russell, P. Norvig. ***Artificial Intelligence . A modern approach.*** Prentice Hall, 2nd edition, 2004 (Chapters 6) http://aima.cs.berkeley.edu/2nd-ed/

1. Games: problem definition

Game theory is widely studied by economists, mathematicians, financiers, etc.

Most studied games in Artificial Intelligence are:

- **Deterministic** (but strategic): luck does not play a role
- **Turn-taking**: play alternatively
- **Two-player**: person-person, person-computer
- **Zero-sum**: the gain (or loss) of a participant is exactly balanced by the loss (or gains) of the opponent. This opposition makes the situation *adversarial*.
- **Perfect information**: games are known and limited; each player knows perfectly the game evolution and what moves he and his opponent can make (observable)

Players are usually restricted to a small number of actions

Actions outcomes are defined by precise rules

Games are hard to solve. E.g.: chess has an average branching factor of about 35, and games often go by 50 moves by each player, so the search tree has about 35^{100} nodes …

Games require the ability to make some decision even when calculating the optimal decision is infeasible

1. Games: problem definition

We will consider games with two players: **MAX** and **MIN**. MAX moves first and they take turns moving until the game is over.

Initial state: includes the board position and identifies the player to move.

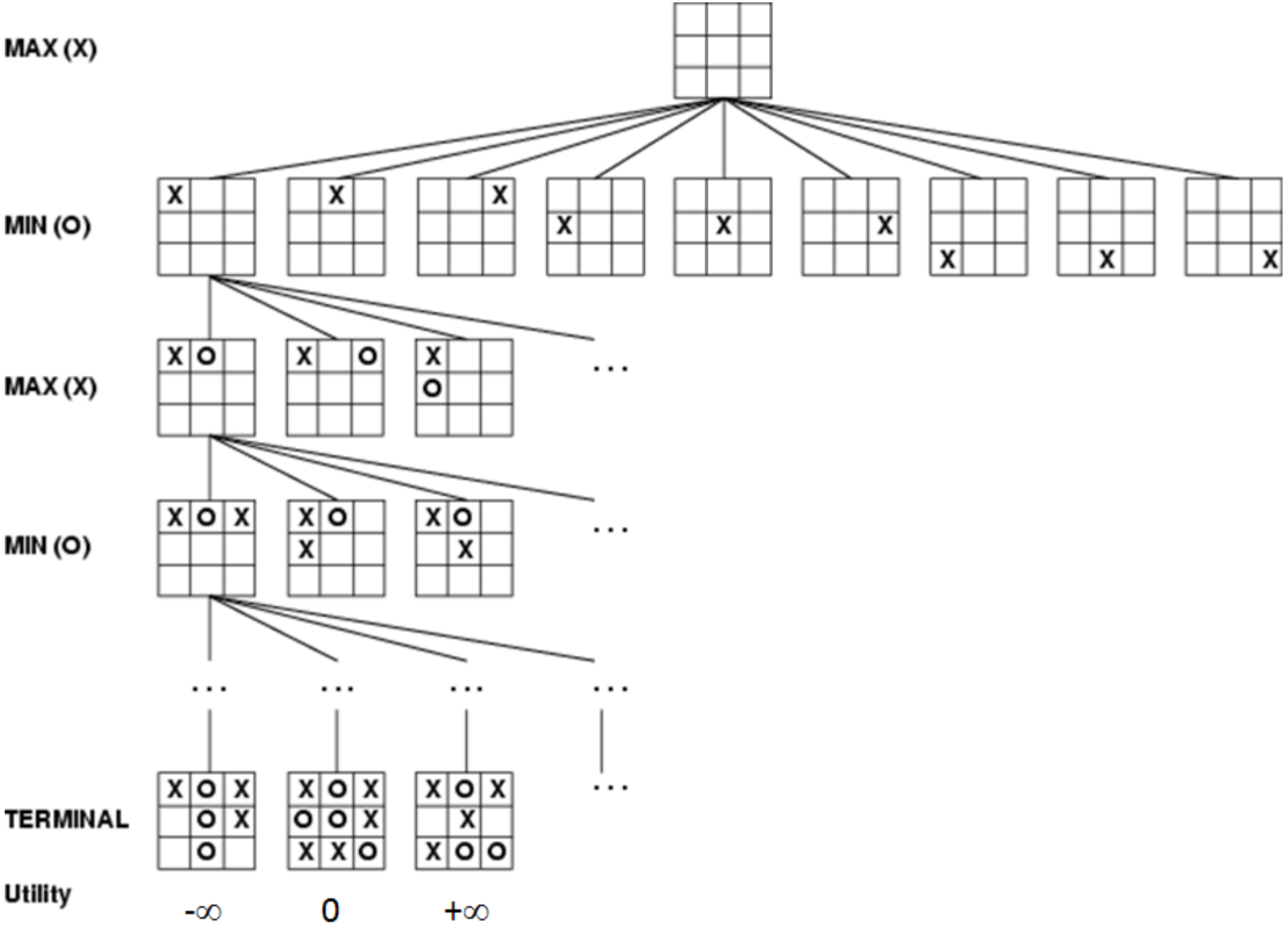
Actions (successor function) : legal moves that the player can take; each actions gives rise to a resulting state (new board position). Successor states represent the states that a player can reach in one move (one turn).

Terminal state (leaf nodes, goal state): determines when the game is over.

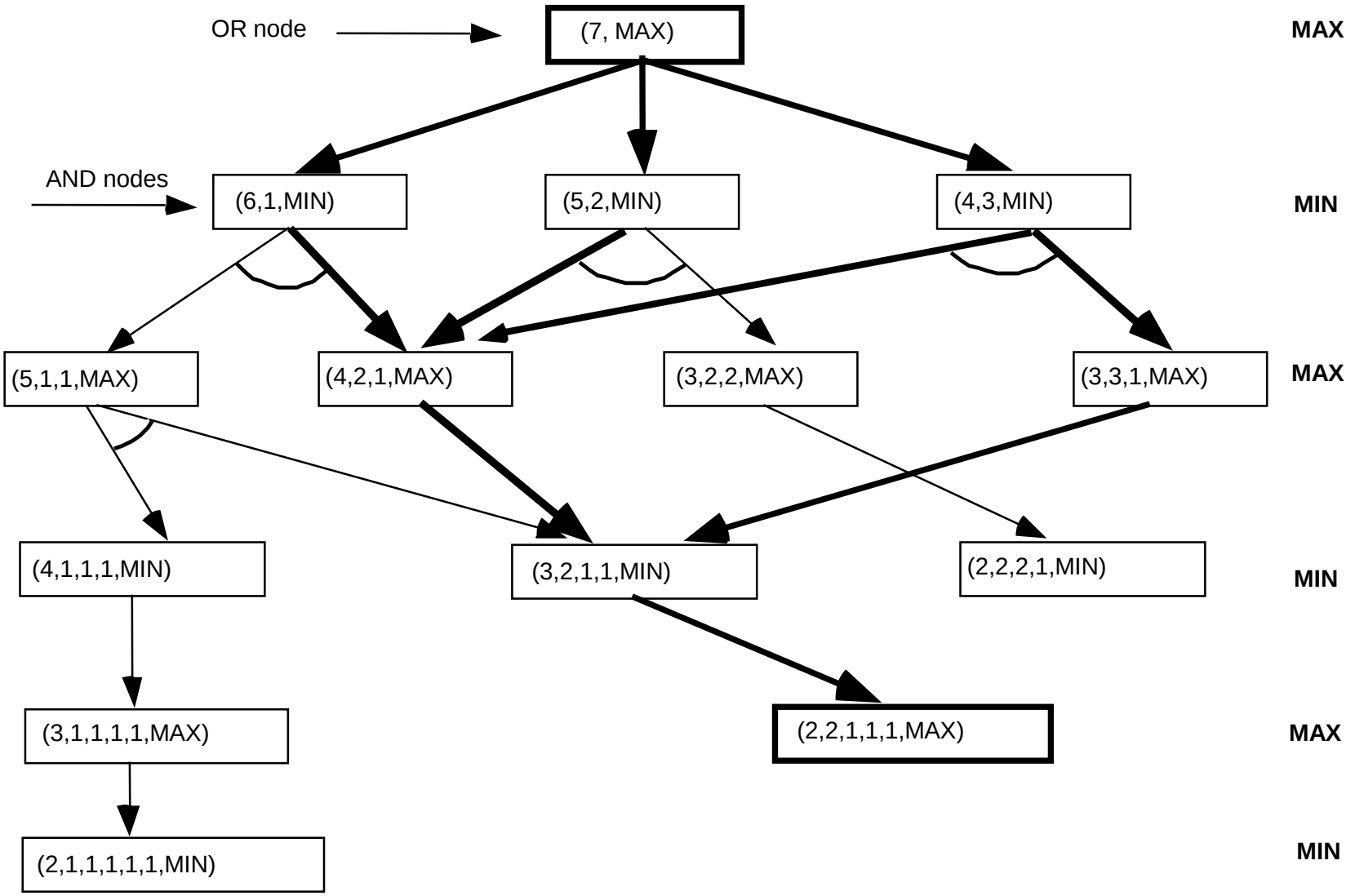
Utility function: also called objective function or payoff function; it gives a numerical value for the terminal states. The outcome is a win (+∞), a loss (-∞) or a draw (0)

- The initial state and the legal moves for each player define the game tree for the game.
- MAX is the initial player. Players move alternatively. The numbers on each leaf node indicates the **utility value of the terminal state from the point of view of MAX**. High values are good for MAX and bad for MIN.
- Objective: use the search tree (particularly the utility of terminal nodes) to determine the best move for MAX.

1. Games: problem definition



1. Games: problem definition: the Grundy's game



1. Games: problem definition

Problems:

- In general, game playing can be solved by applying search techniques on AND/OR graphs

- However, even for simple games, a complete search turns out to be intractable

- Even though different techniques can be applied in order to reduce search, these are proved not to be enough.

Solutions:

- Increase the heuristic information of the method at the cost of perhaps losing admissibility. The goal is to approach $b^* \cong 1$. (This is not feasible in some games as there is very little knowledge on the problem).

- Prune the search tree at a certain point and use methods to select the best initial move (MINIMAX, α–β).

2. Minimax algorithm

When searching for the optimal solution for MAX (best initial move for MAX), MIN has something to say about it. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an **infallible opponent**.

It is infeasible, even for simple games, to draw the entire game tree

Minimax procedure

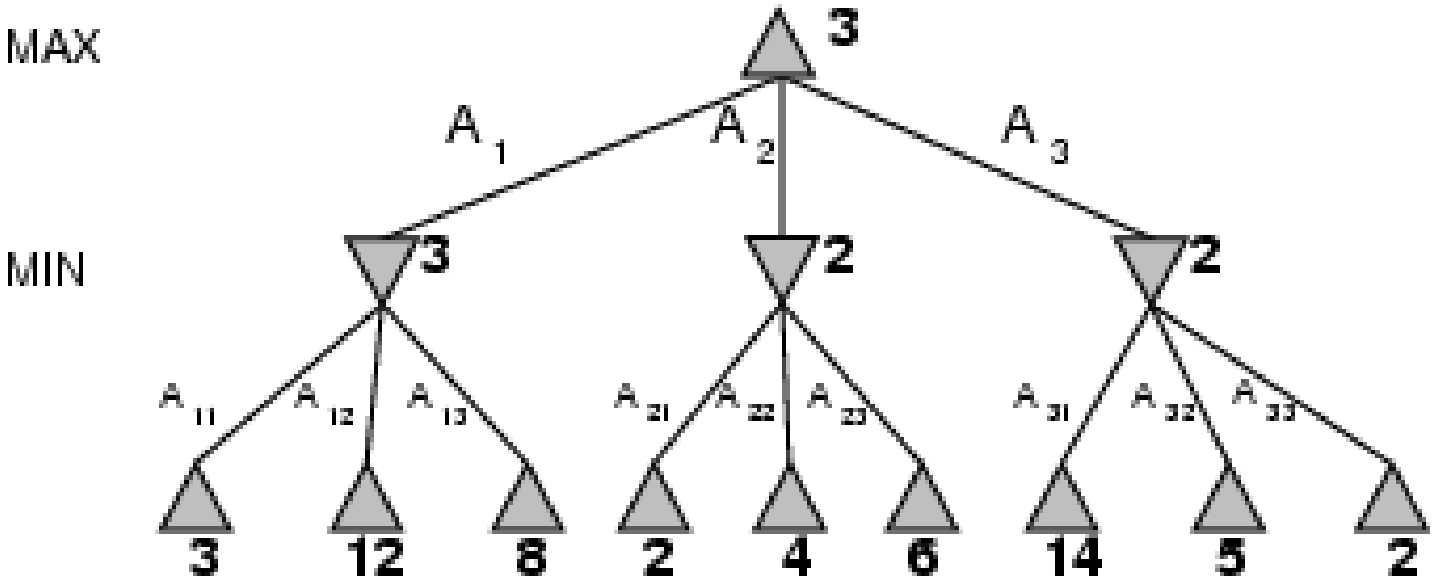
1. Generate the game tree up to a certain depth level in breadth-first fashion, all the way down to the terminal states of the maximum depth level.
2. Apply the utility function to each terminal state to get its value.
3. Use the utility values of the terminal states to determine the utility of the nodes one level higher up in the search tree (depth-first). Continue backing up the utility values from the leaf nodes toward the root, one layer at a time (depth-first). MIN nodes take the minimum utility value of its successors; MAX nodes take the maximum utility value of its successors.
4. Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the state with the highest utility value. That is, it chooses the action corresponding to the best possible move, the move that leads to the outcome with the best utility (maximum utility) assuming that the opponent plays to minimize utility.

2. Minimax algorithm

Utility values of nodes (minimax values of nodes)

Minimax-value(n)=

$$\begin{cases} \text{Utility}(n) & \text{if } n \text{ is a terminal node} \\ \max_{s \in \text{Successors}(n)} \text{Minimax-value}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successors}(n)} \text{Minimax-value}(s) & \text{if } n \text{ is a MIN node} \end{cases}$$

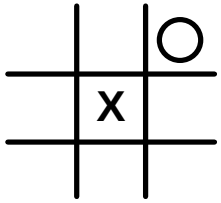


2. Minimax algorithm: tic-tac-toe

Max: X **MIN: 0**

f(n) =
(number of complete rows, columns or diagonals that are still open for MAX) – (number of complete rows, columns and diagonals that are still open for MIN)

f(n)= +∞, if it is a win for MAX
f(n)= - ∞, if it is a win for MIN



f(n)= 5 – 4 = 1

2. Minimax algorithm: tic-tac-toe

Search tree:

- Depth: 2 levels
- Symmetric positions are not represented

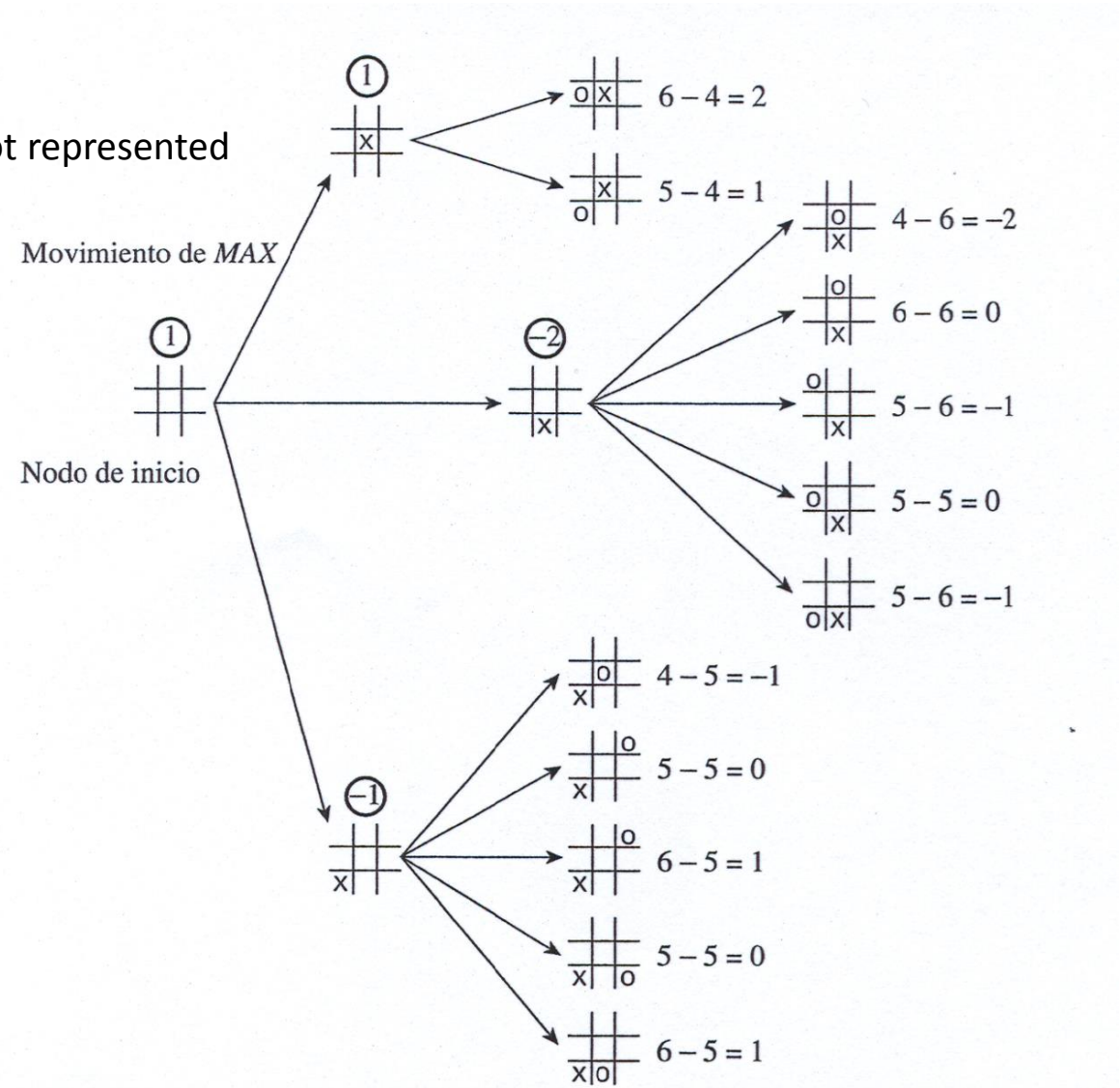
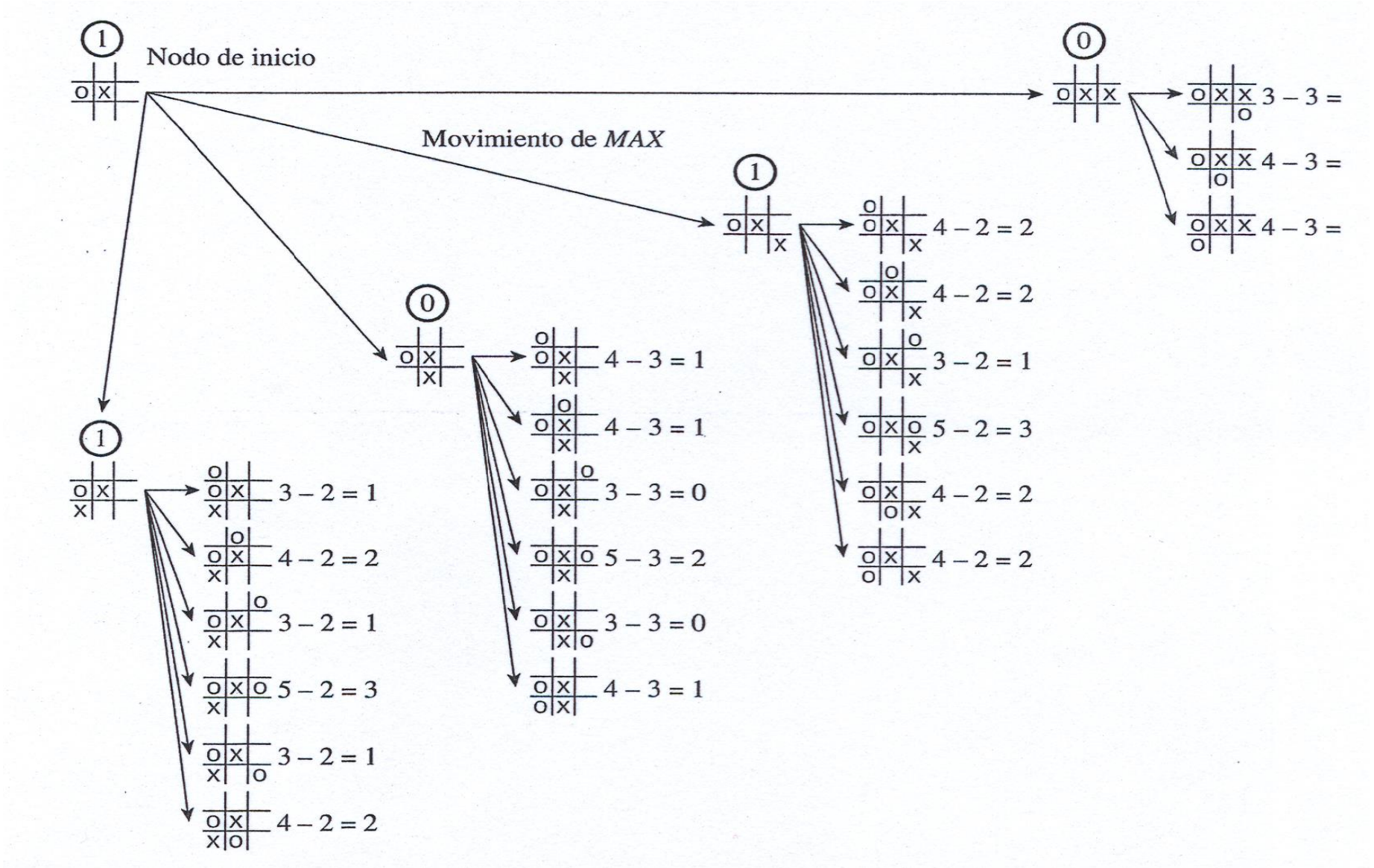
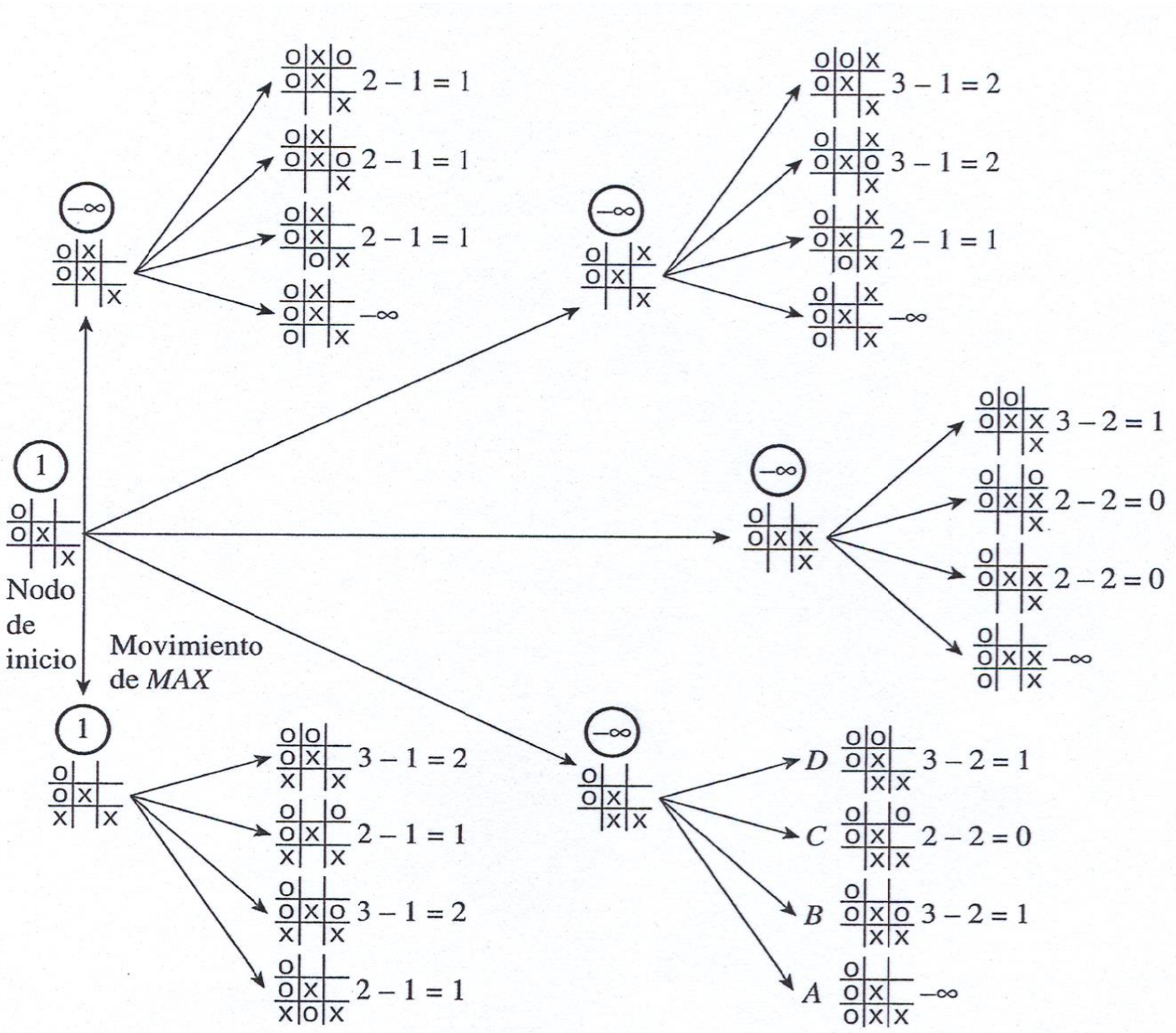


Figura 12.3.

2. Minimax algorithm: tic-tac-toe



2. Minimax algorithm: tic-tac-toe



2. Minimax algorithm: summary

1. Generate the game tree up to a certain level (breadth-first expansion).
2. Apply the utility function to each terminal state to get its value
3. Back up the values to the nodes one level higher up in the tree (depth-first exploration):
   MAX node: largest value of its successors
   MIN node: smallest value of its successors
4. If MAX is the initial node, it will choose the largest backed-up value.

Procedure utility:

The selection of the best first move is made taking into account the game evolution in deeper levels: the backed-up values of the nodes are much more informed than the result of applying directly the utility function over these nodes

Procedure efficiency:

- depth level in the game tree
- utility function.

3. α-β pruning

The problem with Minimax search is that the number of game states it has to examine is exponential in the number of moves.

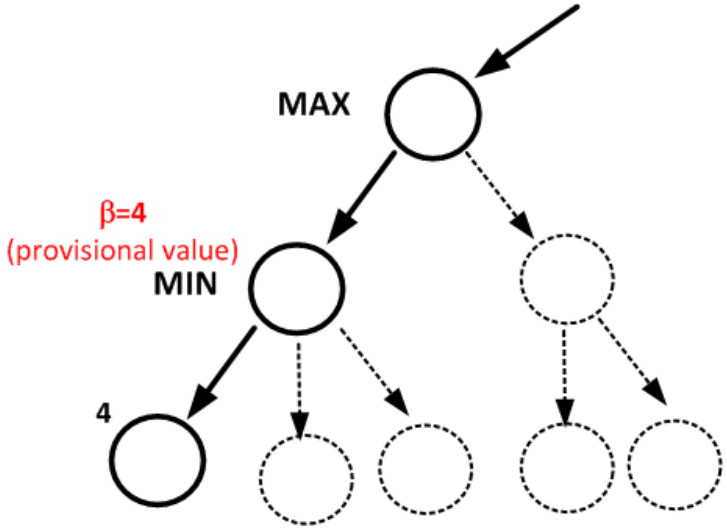
It is possible to compute the correct Minimax decision without looking at every node in the search tree. That is, we can eliminate large parts of the tree from consideration.

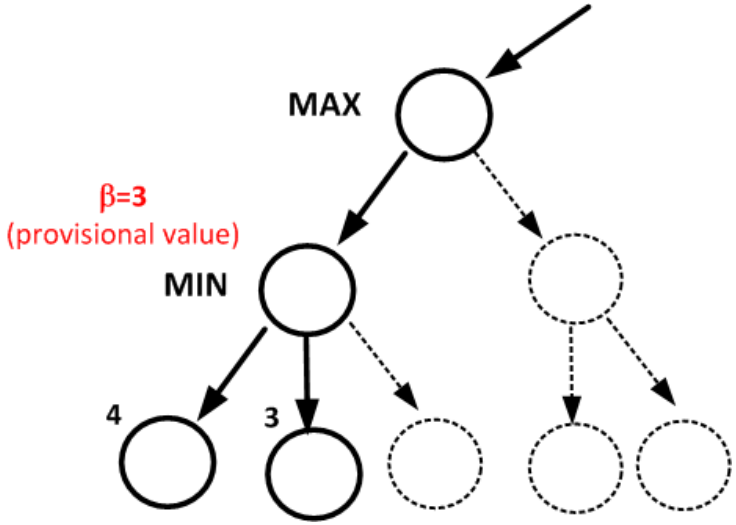
α-β pruning returns the same move as Minimax would, but prunes away branches that cannot possibly influence the final decision.

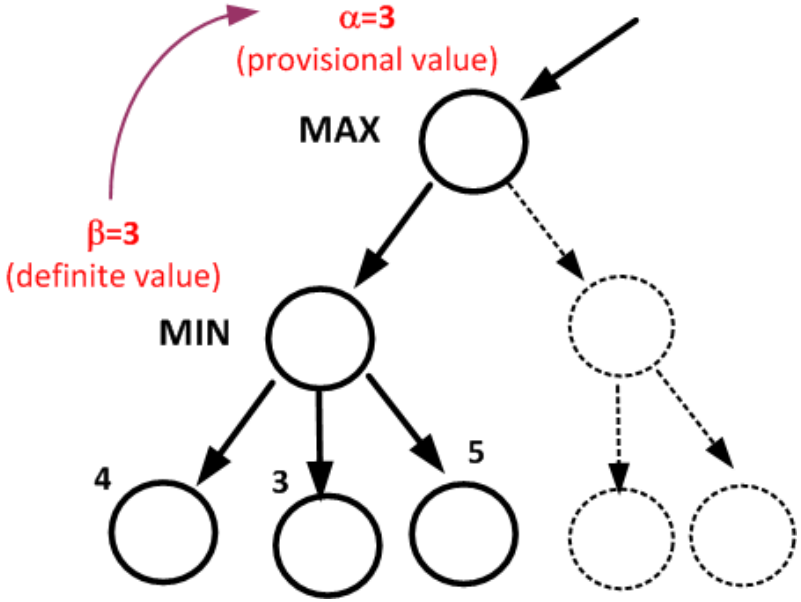
The MINIMAX procedure separates completely the processes of search-tree generation and position evaluation -> inefficient strategy.

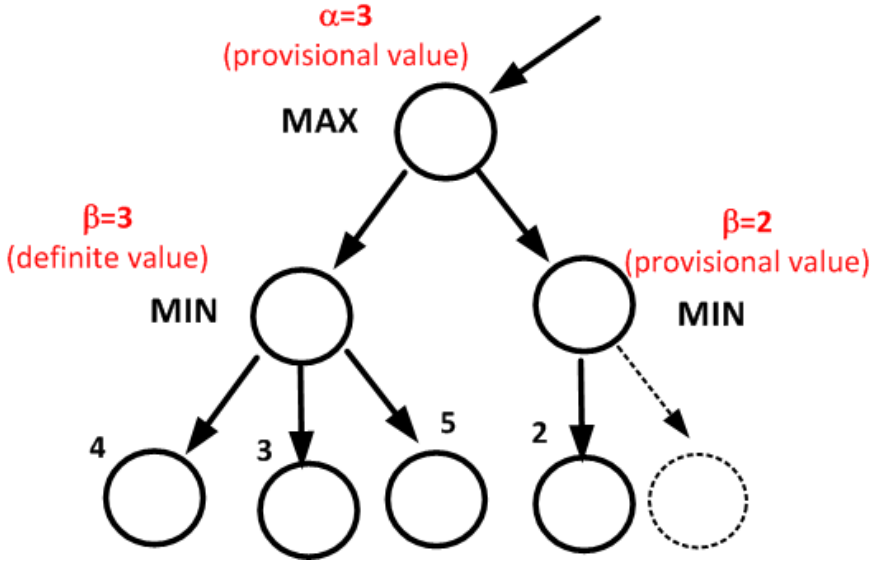
The α-β pruning is based on a procedure that generates a terminal node and, simultaneously, evaluates it and backs up its value (depth-first generation and evaluation/exploration)

3. α-β pruning

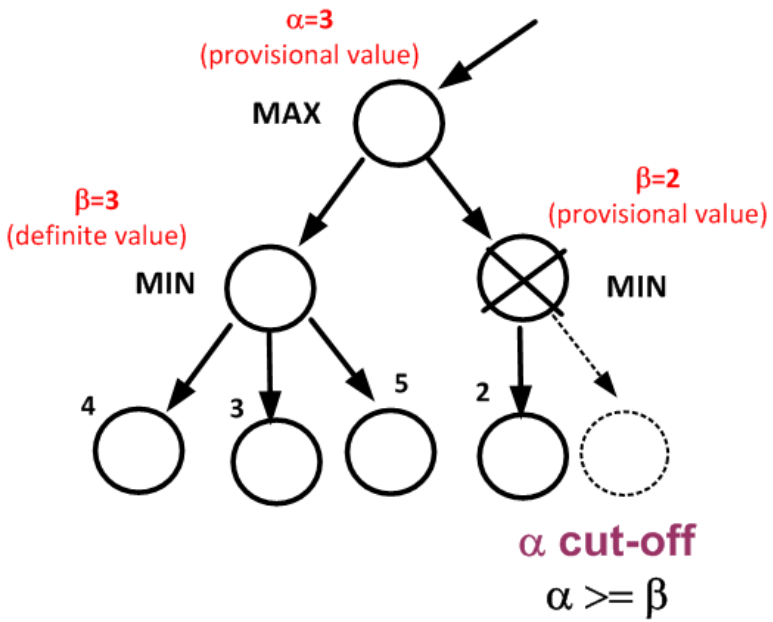

MAX
β=4
(provisional value)
MIN
4

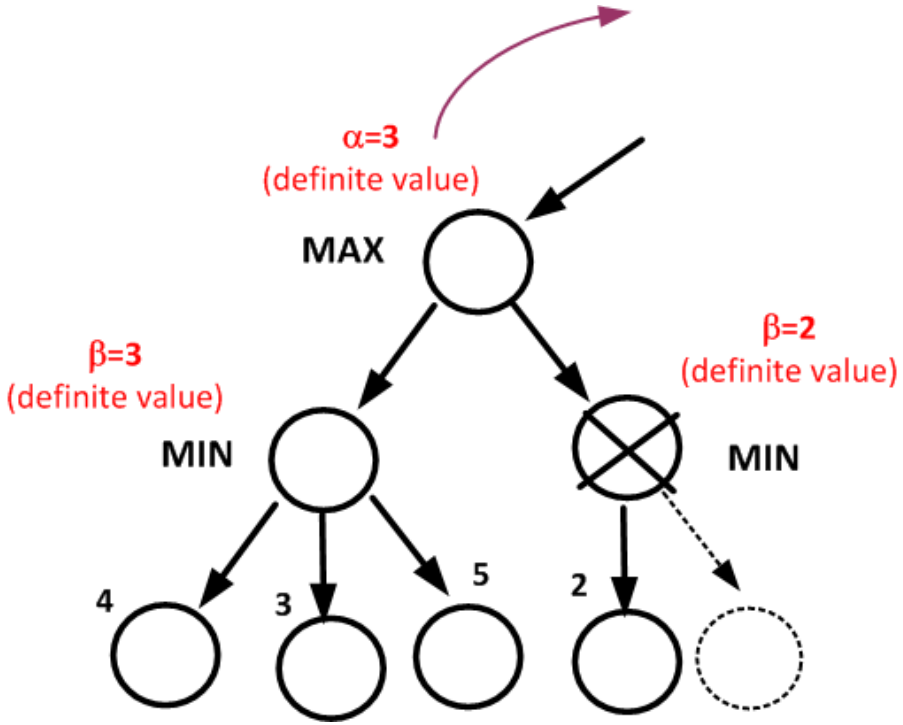

MAX
β=3
(provisional value)
MIN
4
3


α=3
(provisional value)
MAX
β=3
(definite value)
MIN
4
3
5

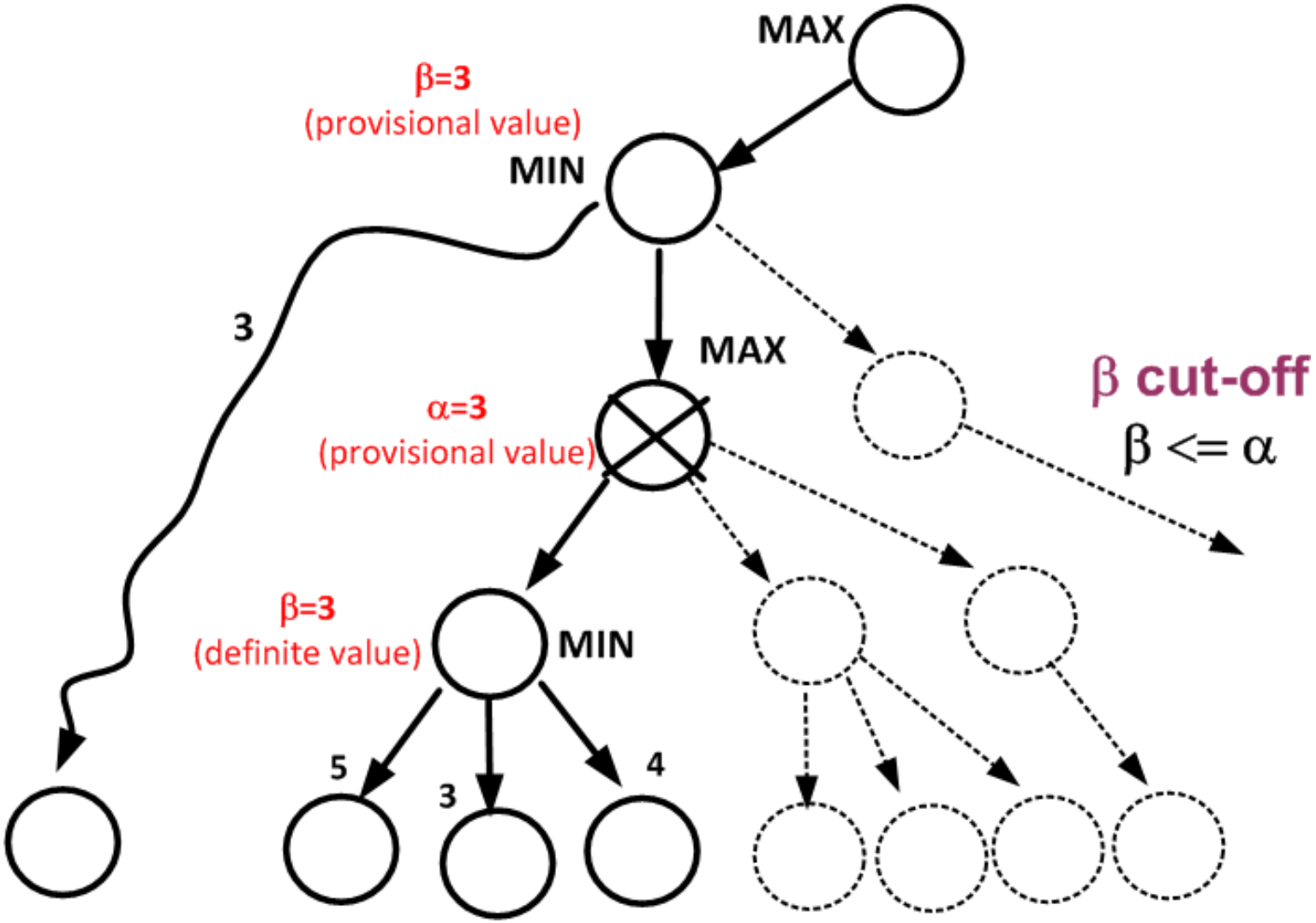

α=3
(provisional value)
MAX
β=3
(definite value)
β=2
(provisional value)
MIN
MIN
4
3
5
2

3. α-β pruning: α cut-off

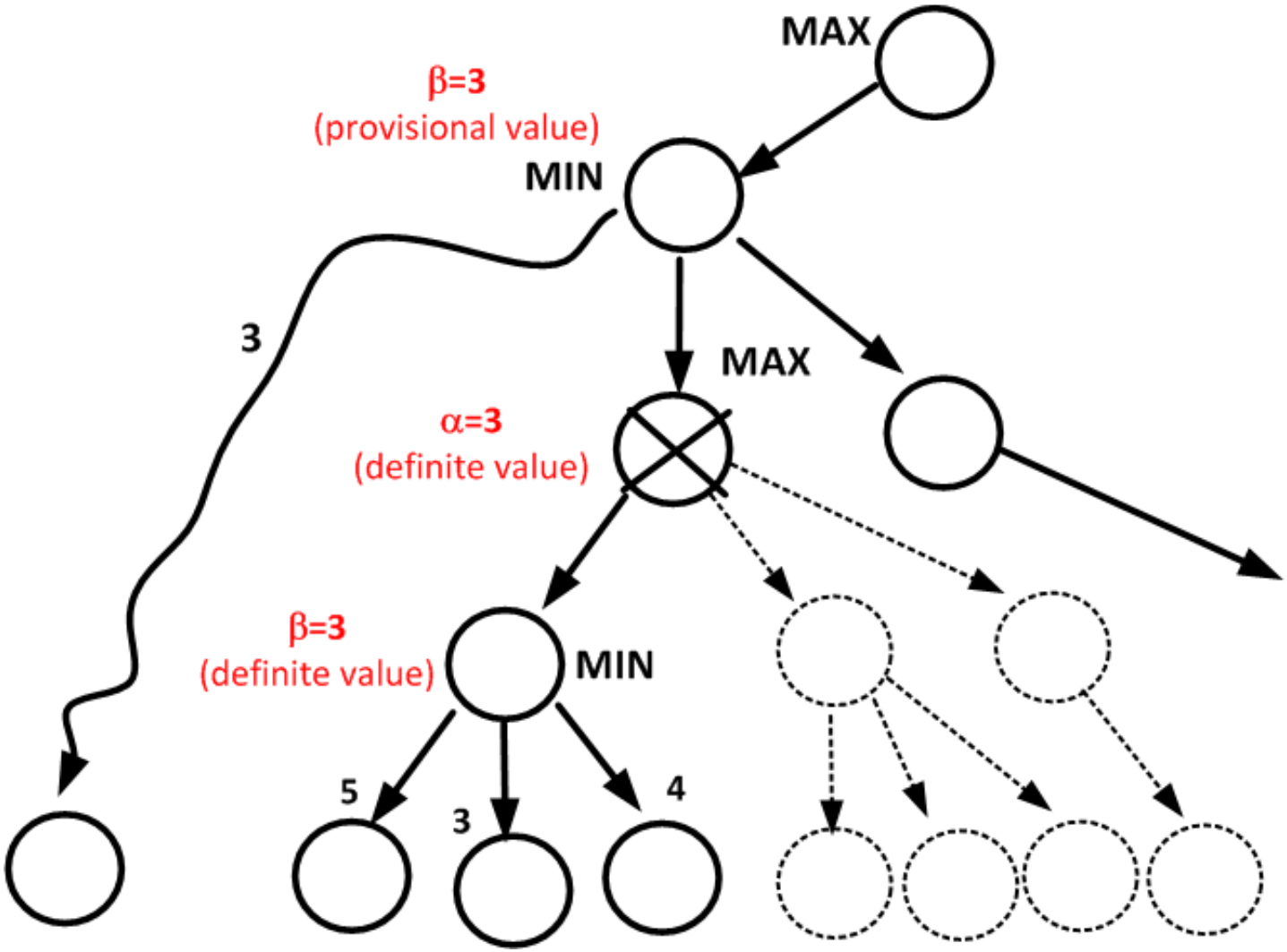




3. α-β pruning: β cut-off



3. α-β pruning: β cut-off



3. α-β pruning: algorithm

1) Initially:

MAX nodes: α values (initially $\alpha = -\infty$)
MIN nodes: β values (initially $\beta = +\infty$)

2) Generate a terminal node in depth-first manner. Compute its utility value.

3) Back up the value to the parent node and keep the best value for the parent.

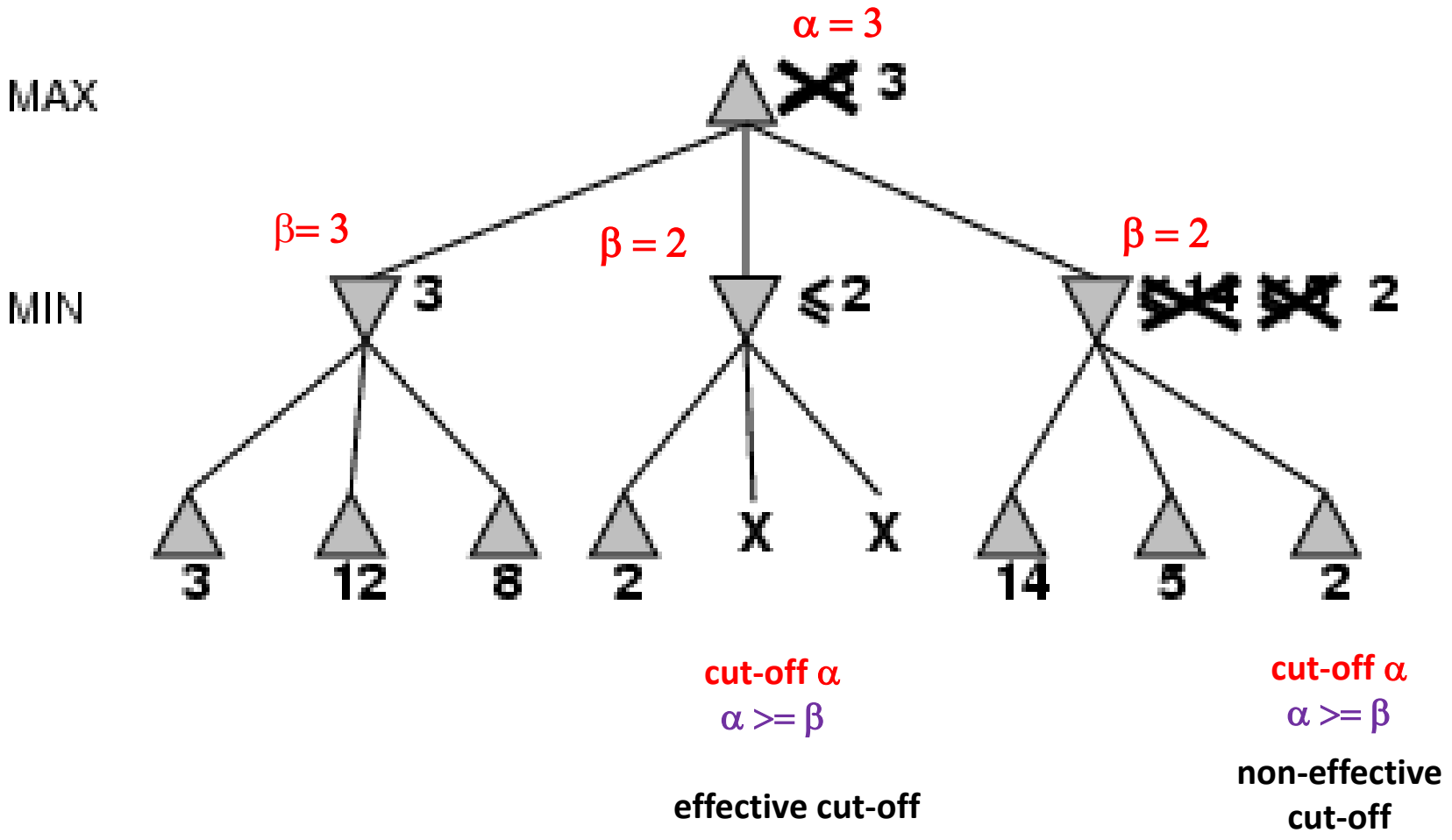
4) Cut-off question. Whenever a node (MAX or MIN) is given a backed-up value, this value is compared with all its contrary predecessor values in order to apply an α cut-off or β cut-off.

- if a cut-off is produced, go to 3

5) Provisional/definite value:

- If it is a provisional value, go to 2
- If it is definite value, go to 3

3. α-β pruning: example 1



3. α-β pruning: characteristics

MAX nodes are assigned α values:

- The α values of MAX nodes are lower bounds and they can never decrease so they can only be assigned values >= α
- The α value of a MAX node is set equal to the current largest final backed-up value of its successors

MIN nodes are assigned β values:

- The β values of MIN nodes are upper bounds and they can never increase so they can only be assigned values <= β
- The β value of a MIN node is set equal to the current smallest final backed-up value of its successors

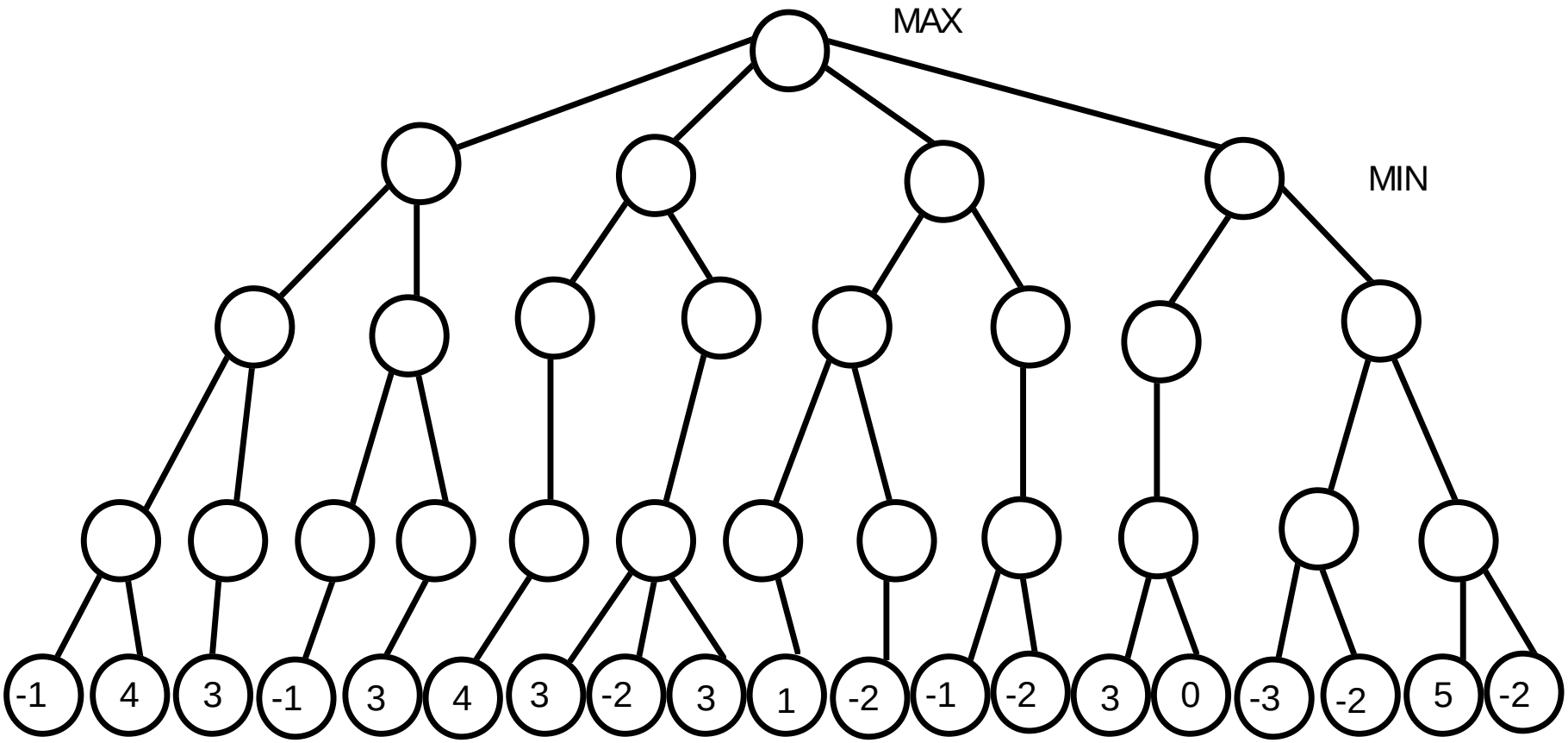
Search can be discontinued below any MIN node having a β value less than or equal to the α value of any of its MAX node ancestors (α cut-off)

Search can be discontinued below any MAX node having an α value greater than or equal to the β value of any of its MIN node ancestors (β cut-off)

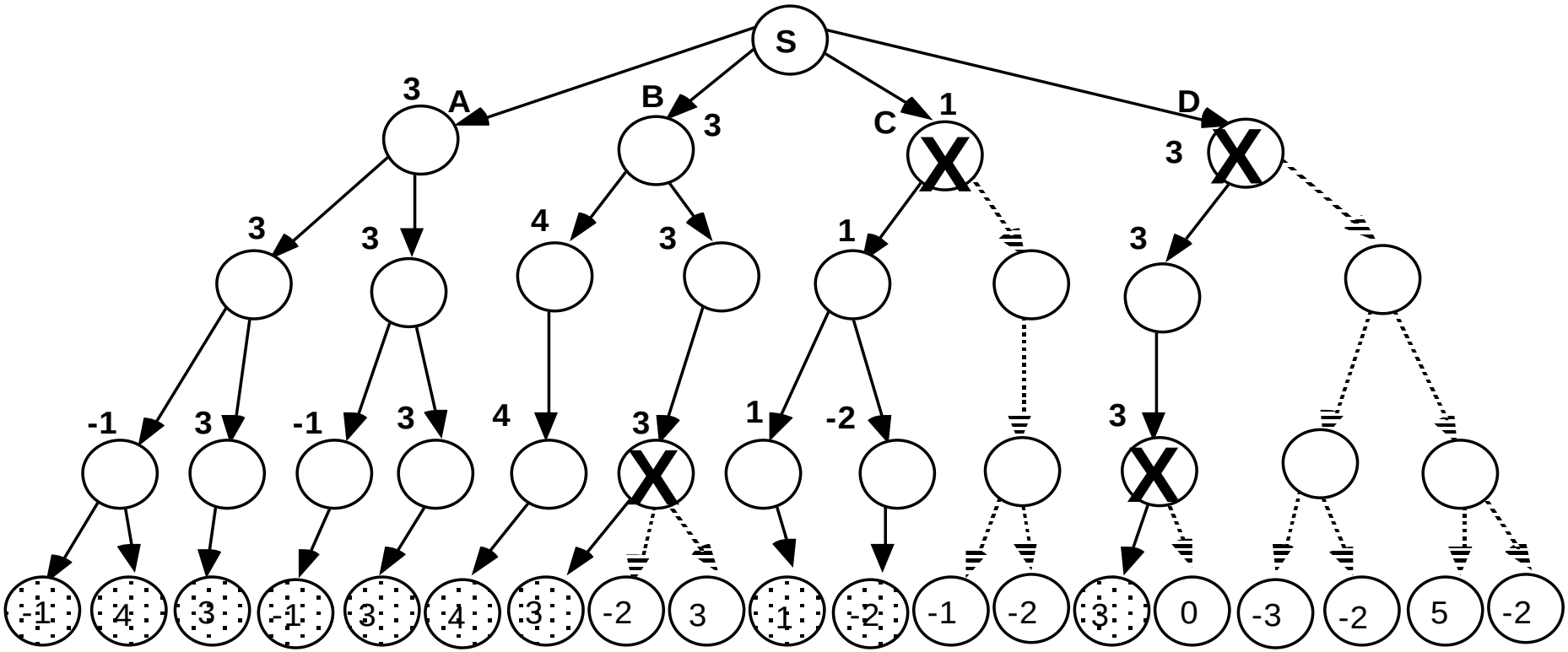
The final values of nodes in an α- β procedure are the same as in MINIMAX , except for those nodes whose values come from an α cut-off (values could be lower) or β cut-off (values could be greater).

If a cut-off is produced in a node selectable by the initial MAX node, the node will not be chosen in the game as its value is improved by another selectable node which has not been cut off.

3. α-β pruning: example 2



3. α-β pruning: example 2



3. α-β pruning: efficiency

The search efficiency of the α- β procedure depends on the number of cut-offs that can be made during a search

The number of cut-offs depends on the degree to which the early α and β values approximate the final backed-up values, that is, the moment to which the α and β values allow to make a cut-off.

The final value of the start node is set equal to the static value of a terminal node (best node at the expansion depth level): if this node appears soon, the number of cut-offs in the depth-first search will be optimal thus generating the minimal search-tree.

The number of terminal nodes of depth **d** that would be generated by optimal α- β search is about the same as the number of terminal nodes that would have been generated by MINIMAX at depth **d/2**. Therefore, in the same time α- β allows to explore twice as much as the MINIMAX procedure.

4. Additional refinements

There are different variants of α–β to find the best terminal node as soon as possible and maximize the number of cut-offs:

1. Ordered expansion at each level
2. Preliminary ordering

In game playing it is necessary:

- to use more intelligent procedures to prune the search tree rather than loads of computation
- to use more intelligent procedures to get a dynamic algorithm behaviour

Examples:

- to have a set of typical moves (a database of moves, common initial moves, preferences over tiles, etc.)
- to have examples of board patterns and best moves for those positions
- to have game strategies, habits of the opponent, mistakes (wrong moves), etc.
- modifications/utility function learning methods