Game Theory

Exploring a Search Space

GAME TREES - example games (tic-tac-toe, chess, checkers, etc.)

Game Concepts

- Two Player Games (player **A** and player **B**, **A** goes first)
- Alternating Moves
- Legal Moves (assume finite)
- Win/Loss/Draw Definitions
- Board or Game Configuration C_i
  - Terminal Configuration - Win/Lose/Draw (one or many)
  - Non-terminal Configuration - all others

Sequence of Moves and Countermoves

A sequence of board or game configurations $C_1 \ldots C_m$ is **valid** if

i. C_1 is a starting configuration of the game
ii. C_i, $0<i<m$ are non-terminal configurations
iii. C_{i+1} is obtained from C_i by a legal move

by player **A** if *i* is odd

by player **B** if *i* is even

- If C_m is a terminal configuration, $C_1 \ldots C_m$ is an **instance** of the game. All possible instances of a finite game may be represented by a **game tree** (all the possible plays of the game, not necessarily a data structure)

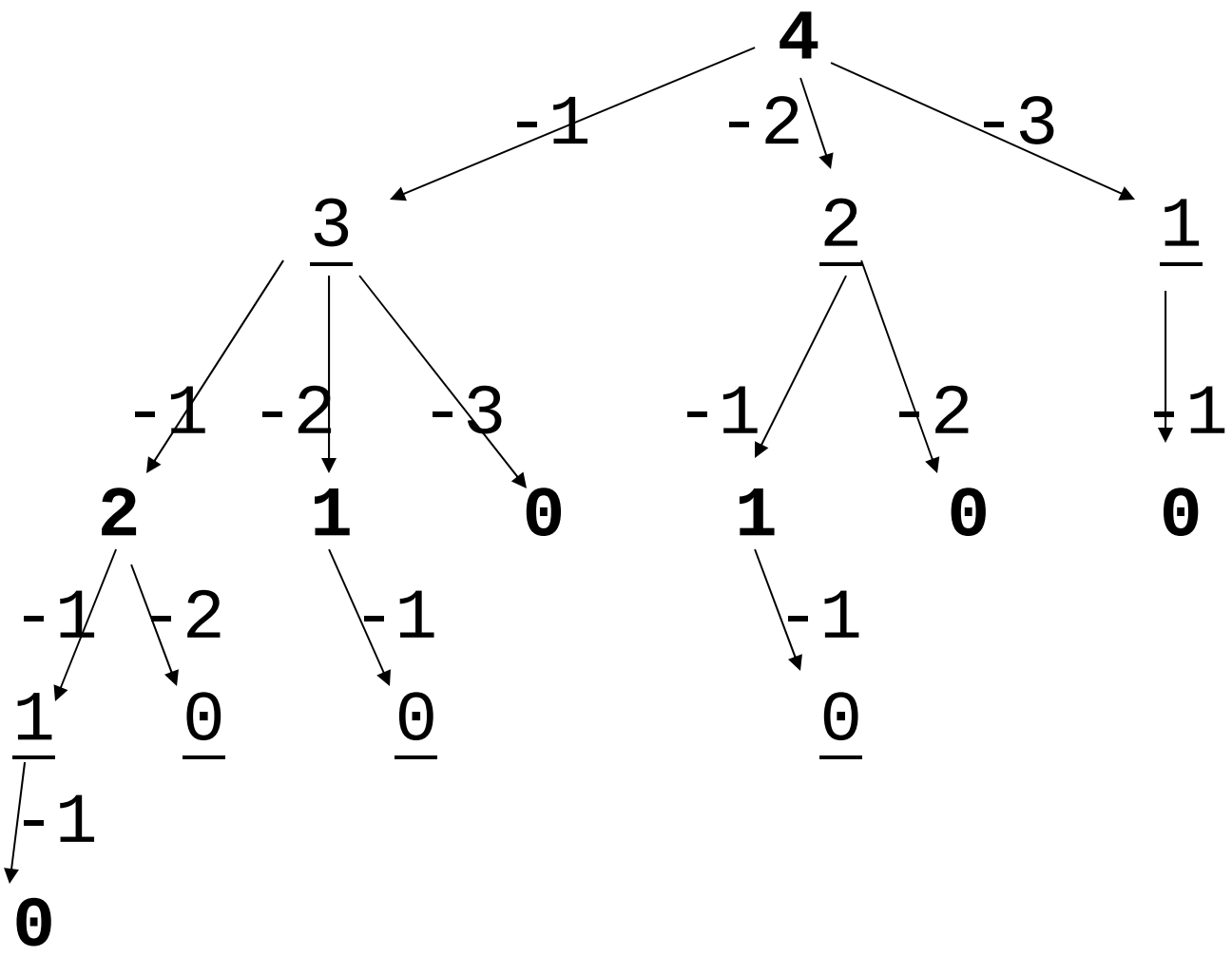
Example: NIM

- start with *n* toothpicks (C_1 = starting configuration of the game); players **A** and **B** alternate moves (player **A** first)
  - Legal Moves: remove 1 or 2 or 3 toothpicks
  - cannot remove more toothpicks than remaining
- Win/Loss: player removing the last toothpick loses; no draw; only one type of terminal configuration (no toothpicks left); tic-tac-toe has many possible terminal configurations
- C_i is described by the number of toothpicks left
  - *i* odd means it is player **A** move
  - *i* even means it is player **B** move

6-NIM GAME TREE

- value on edge = # of toothpicks removed
- value in a node = # of toothpicks remaining
- degree of a node = # children = # legal moves possible
- **A** can win only on **odd** levels (node with **0** means **B** took last toothpick)
- **B** can win only on even levels (node with 0 means **A** took last toothpick)

Sample 4-Nim



The complete game tree maps out all possibilities (instances) for the game. Every path down the tree from the root is a play of the game. Game trees can also be used to determine next move a player should make. From C_1, assuming **A** wants to win, **A** should make the move that maximizes **A**'s chance to win. Use an evaluation function which assigns a numeric value to a game configuration . This is a measure of value or worth of game configuration for player **A**.

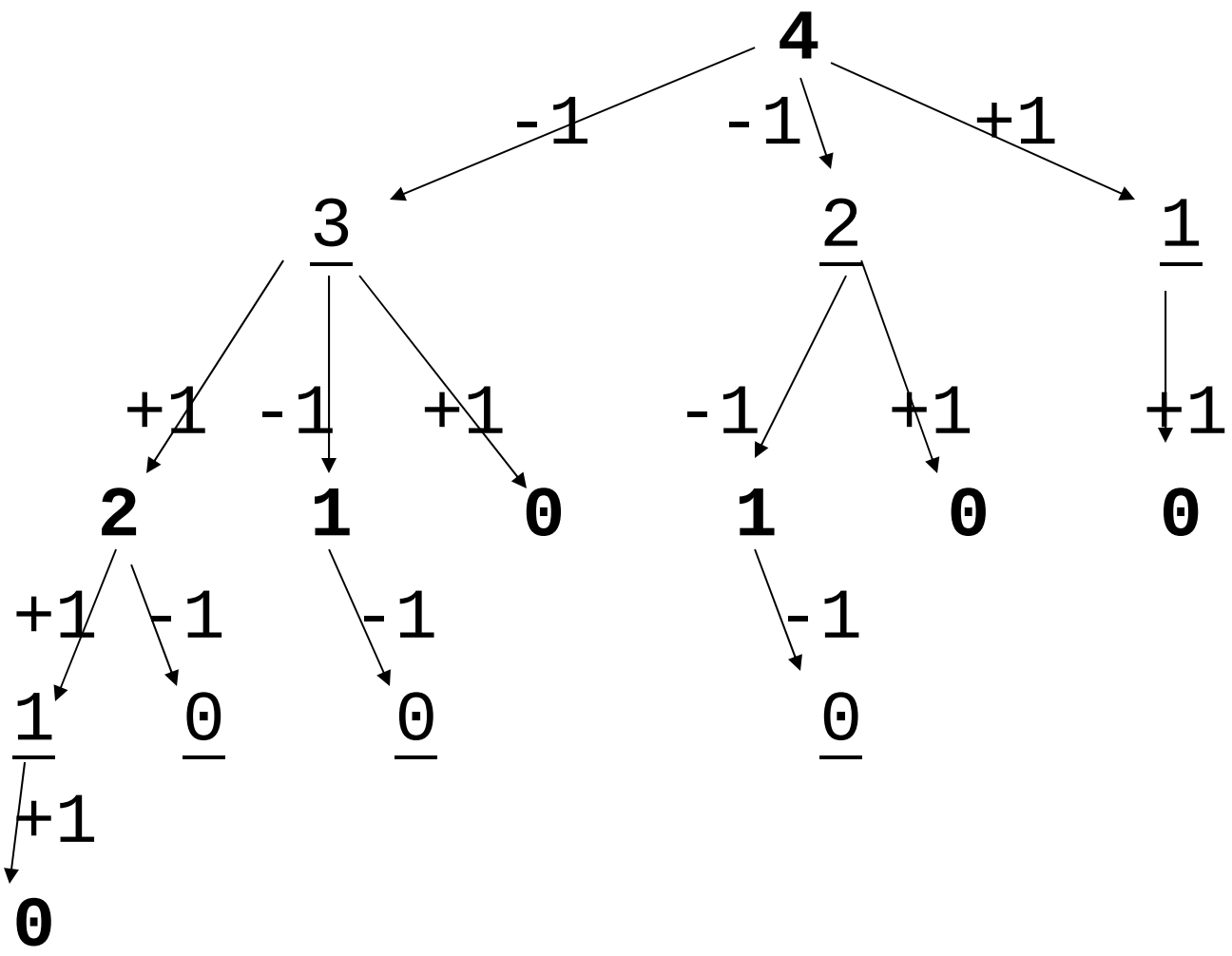
In small game trees, since it is possible to map out game instances all the way to terminal configurations, it is sufficient to assign an evaluation function only at terminal configurations and then somehow propagate these values up the tree to other, non-terminal game configurations. For example:

$$E(X) = \begin{cases} 1 & \textit{if config X wins for A} \\ -1 & \textit{if config X wins for B} \end{cases}$$

If one assumes both **A** and **B** want to win, **A** wants to maximize the game configuration value at **A**-move nodes (**bold)**, and **B** wants to minimize the game configuration value at **B**-move nodes (<u>underscore</u>). The following function describes this Max-Min procedure for transferring the game configuration values up the tree.

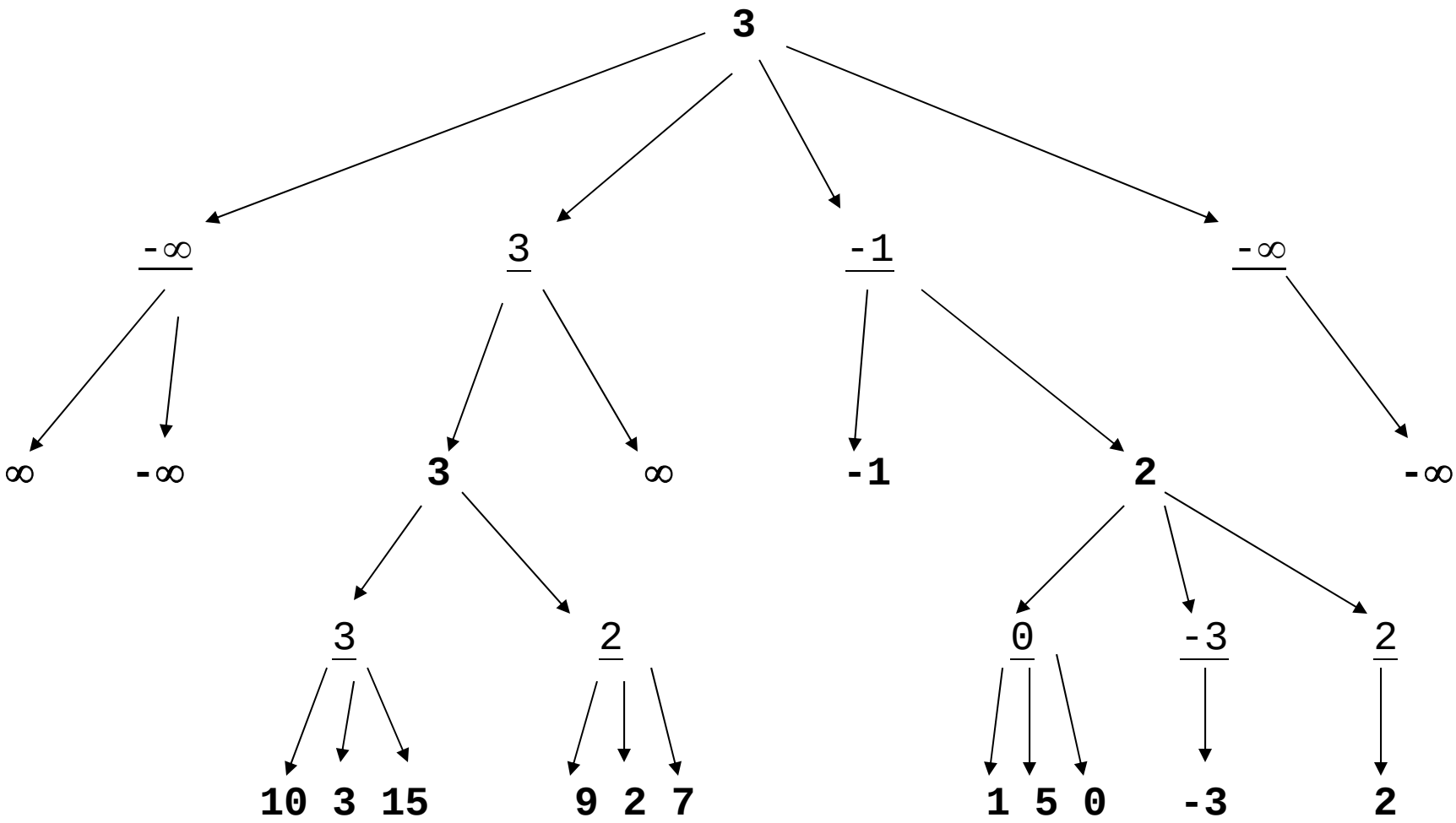
$$V(X) = \begin{cases} \max\limits_{i=\{childrenofX\}} V(X_i) & if \;\; X \; is \; on \; odd \; level \\ \min\limits_{i=\{childrenofX\}} V(X_i) & if \;\; X \; is \; on \; even \; level \end{cases}$$

Sample 4-Nim with configuration values



Above Max-Min Procedure is easy to implement for small game trees when it is possible to look ahead all the way to terminal configurations. For larger games (i.e. chess, checkers), one may only be able to look ahead a few levels in the game tree because of the large number of legal moves and countermoves at each level. This requires us to develop evaluation functions, for non-terminal game configurations, using heuristic understandings of the game to weight various board configurations for player **A**.

Also the function used to transfer the values up the tree may be enhanced from the basic Max-Min procedure described above. The efficiency of both will determine how many levels of the game tree can be searched in the limited amount of time given to determine a move. Trade-off between deeper searching and more ornate evaluation functions will have to be considered.


3
-∞
3
-1
-∞
∞
-∞
3
∞
-1
2
-∞
3
2
0
-3
2
10 3 15
9 2 7
1 5 0
-3
2

Assume player **A** is the computer and rewrite the algorithm to compute in a fairly simple recursive form.

$$V'(X) = \begin{cases} e(X) & \textit{if } X \textit{ is a leaf of the subtree generated} \\ \max\limits_{i=\{childrenofX\}} \{-V'(X_i)\} & \textit{if } X \textit{ is not a leaf of the subtree generated} \end{cases}$$

where e(x)=E(x) if is a position from which **A** is to move and e(x)= -E(x) otherwise.

Below is pseudo-code for a recursive procedure to evaluate by generating only levels of the game tree beginning with as the root. This is a post-order traversal because the value of a node can be determined only after its children have been evaluated.

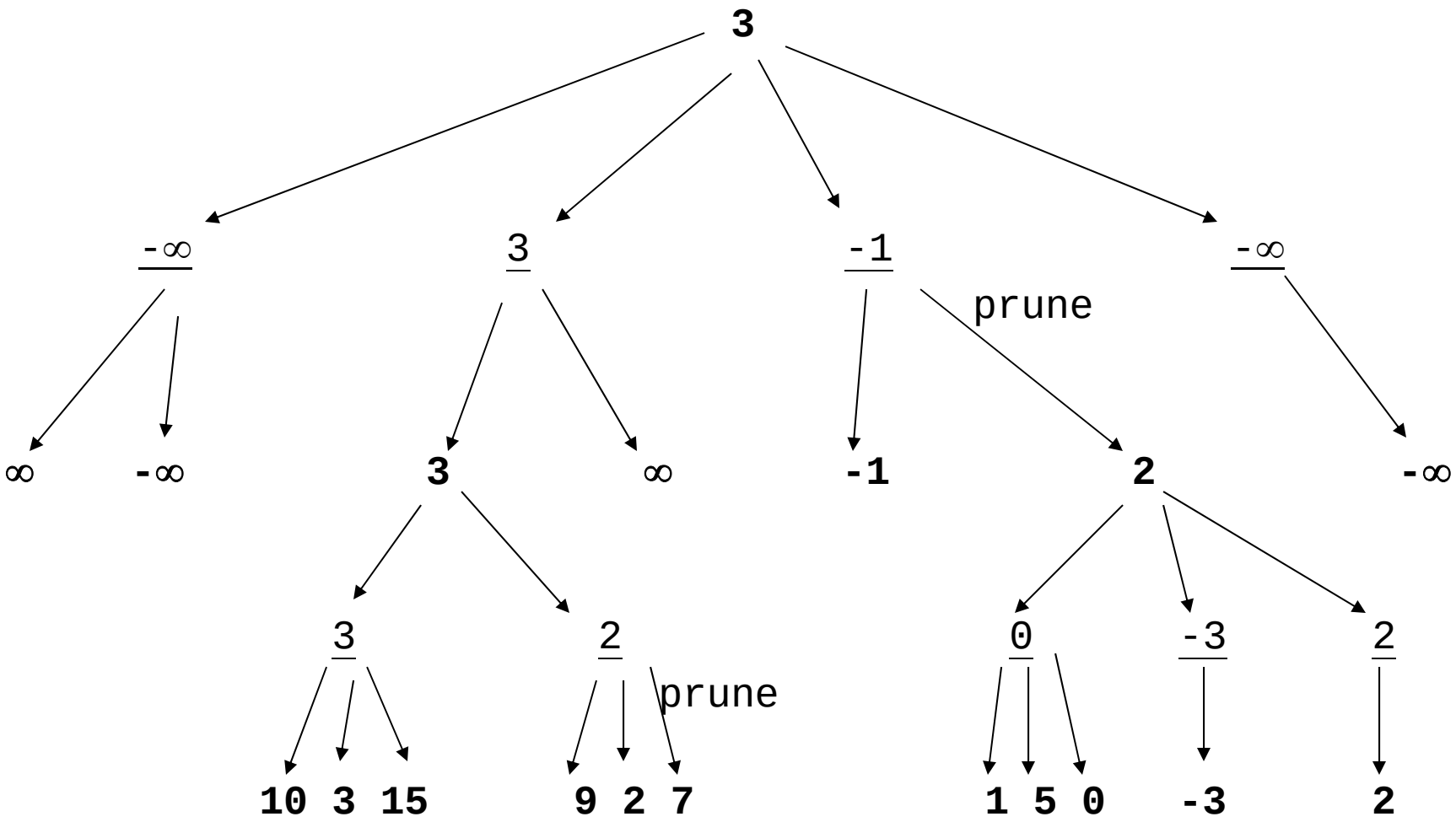
```
VE(X, L)
if X is terminal or L=0
   then return E(X)   // if on B level, return –E(X)
temp = - VE(C1, L-1)  // traverse the first sub-tree //
for i = 2 to #children(X)     // remaining sub-trees //
    temp =  max[temp, -VE(Ci, L-1)]

return (temp)
```

Alpha-Beta Pruning

- ALPHA CUTOFF - The *alpha* value of a maximum position (**A** player) is defined to be the minimum possible value for that position as determined so far.

If the absolute value of a min position (**B** player) is determined to be less that or equal to the *alpha* value of its parent, then no more children of this min position need exploring.


3
-∞
3
-1
-∞
prune
∞
-∞
3
∞
-1
2
-∞
3
2
prune
0
-3
2
10 3 15
9 2 7
1 5 0
-3
2