Lectures 6: Adversarial Search (Game Playing)

Artificial Intelligence

CS-6364

Adversarial Search

- Zero-Sum Games (Perfect Information)
- Games with imperfect real-time Decisions
- Chance Games

Game Playing

- The goals of competitive agents that evolve in multiple-agent environments are **<u>in competition</u>**
  - adversarial search (game playing)
- Why is the study of games part of AI?
  - scientific reasons: easy to represent, difficult to solve, need smart searching through huge search spaces
  - non-scientific reason: humans simply love to play
- Simplifying assumptions about games:
  - deterministic environment (but...backgammon)
  - fully observable environment (but...poker)
  - two players, i.e. MAX and MIN (but...multi-player card games)
  - zero-sum games: the scores of the players at the end of the game are equal and opposite, e.g. +1 for MAX, -1 for MIN

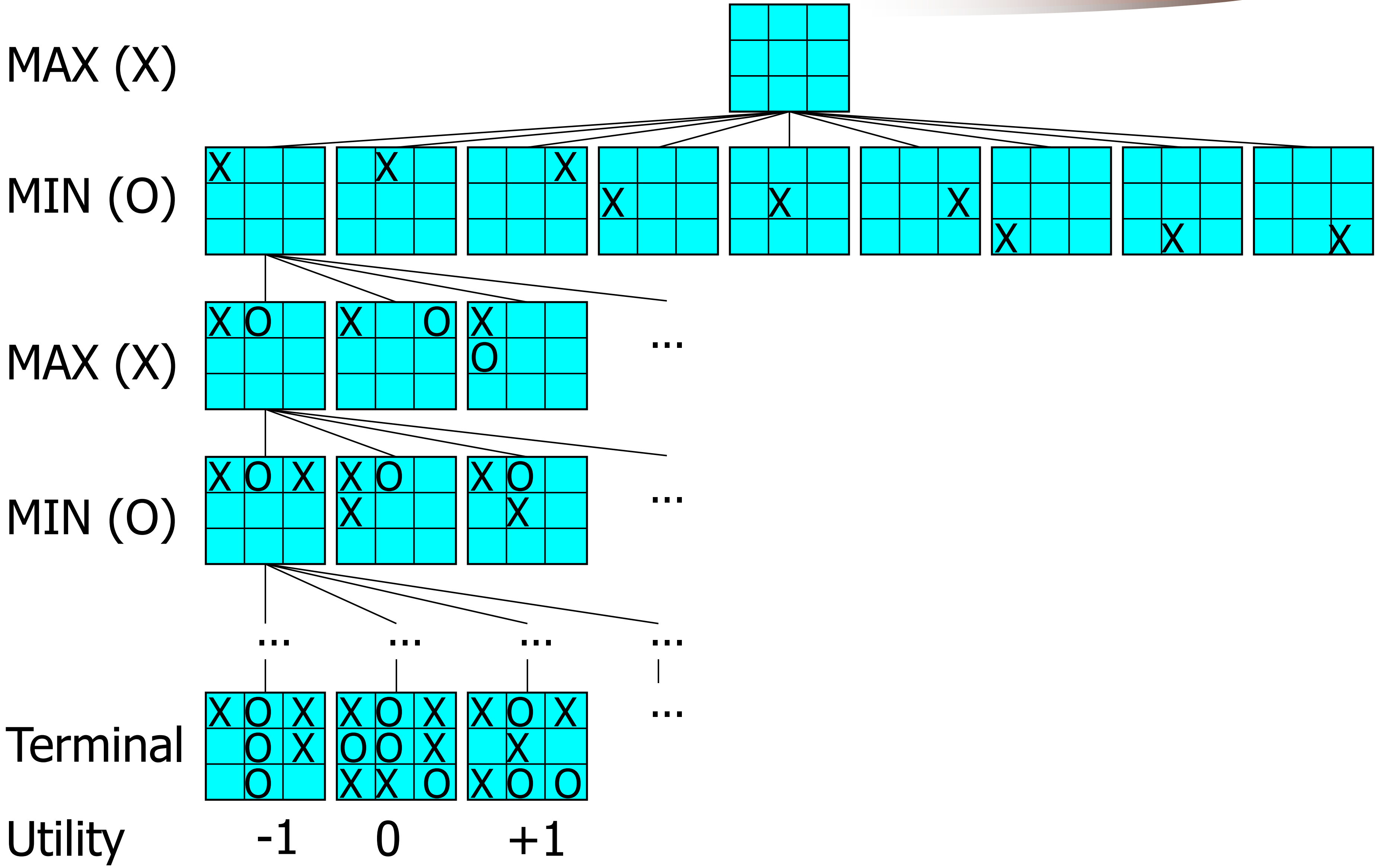
Game playing

- Game playing is a search problem where you can have:
  - perfect decisions
  - imperfect decisions
- Pruning is a technique to eliminate parts of the game tree
- Pruning allows you to ignore portions of the search tree that make no difference to the final choice

Representing Games

- **Initial state**
  - board position; also specify which player moves first
- **Successor function**
  - return pairs (move, successor state)
- **Terminal test**
  - check if game over
  - states where terminal test succeeds are terminal states
- **Utility function**
  - also called objective function or payoff function
  - assign a numeric value to each terminal state
  - winner is awarded, loser is penalized
  - e.g., +1 for winning, 0 for draw, -1 for losing in chess
- **Game search tree**
  - defined by the initial state and legal moves

Game of TIC-TAC-TOE



Perfect decisions

- We will assume that we have 2 players - MAX and MIN. MAX moves first. At the end of the game, the winner is awarded points or the loser gets penalty points. This will become our utility function for our agent program.
- In a normal search, MAX will search for a *move sequence* to reach a winning terminal state.

Note: MAX has to find a strategy to reach a winning position regardless of what MIN does

Game strategy

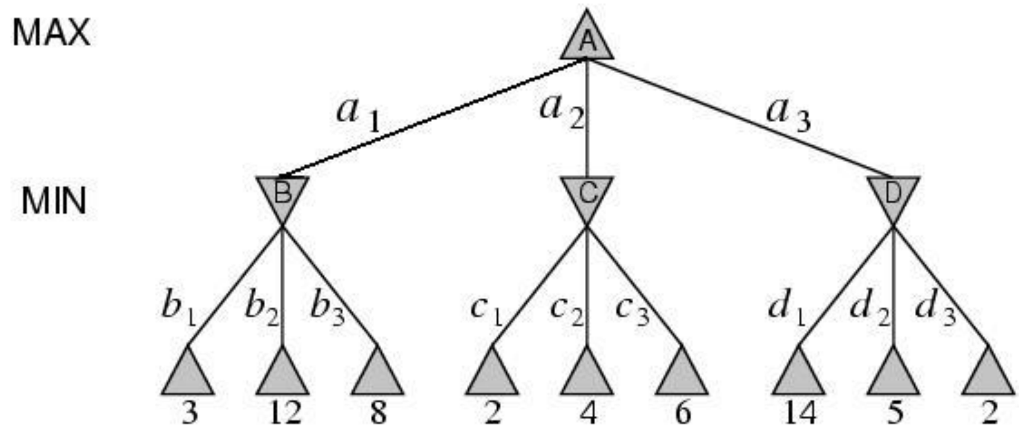
In a normal search problem, MAX would have to find a sequence of moves that leads to a terminal state that is a winner

 Unfortunately, MIN has something to say too!!

 MAX must find a strategy that leads to winning regardless of what MIN does!

Game Strategies

- One "move" consists of two "plies"
  - each player moves once (such a move is called a "ply")
- Each player takes into account the other player's possible moves
  - MAX moves in the initial state, then it moves to another state resulting from any possible response move of MIN etc.



a simplified game search tree

A is the initial state

- MAX can move a_1, a_2, a_3

If move a_2 is selected, then state C is reached

In response to a_2, MIN can move c_1, c_2, c_3

The utilities of terminal states are known

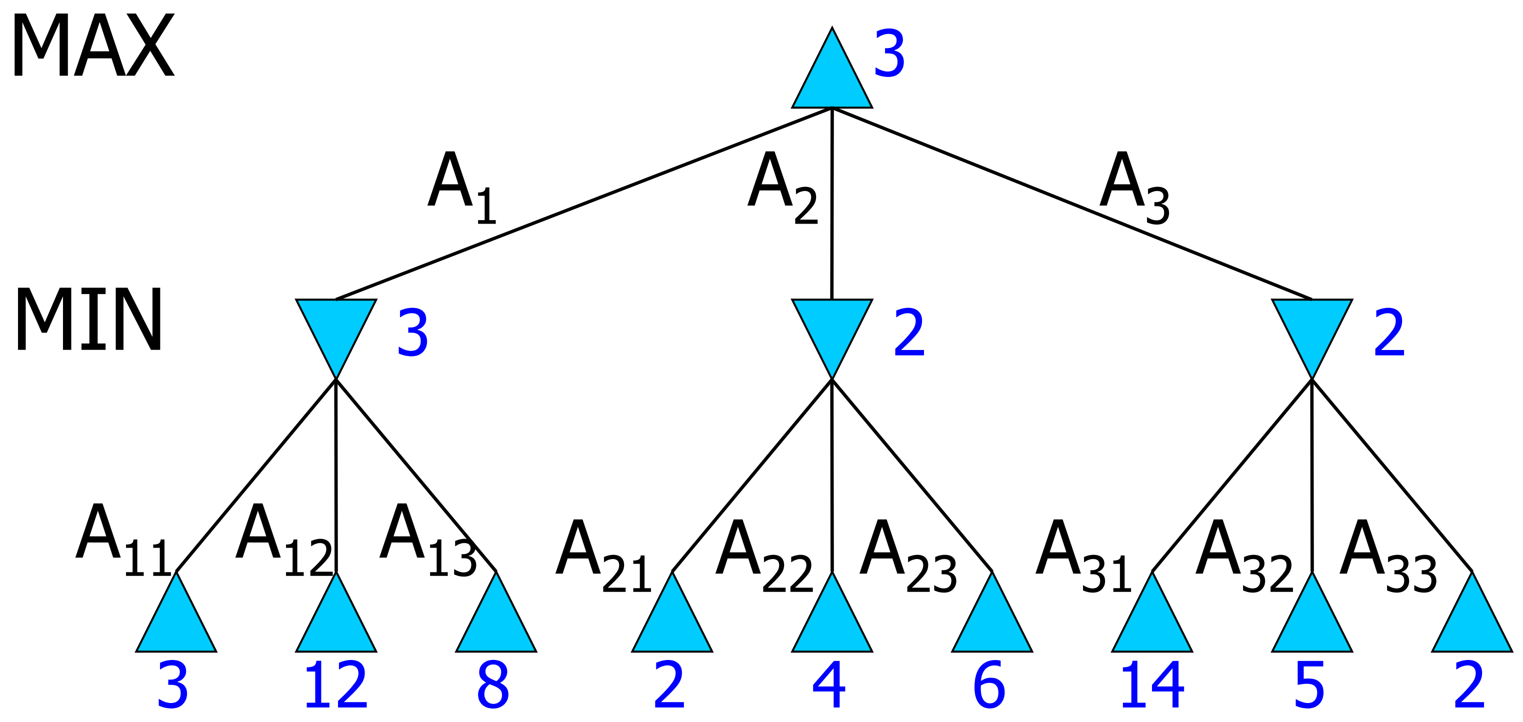
How should MAX play?

Optimal Strategies

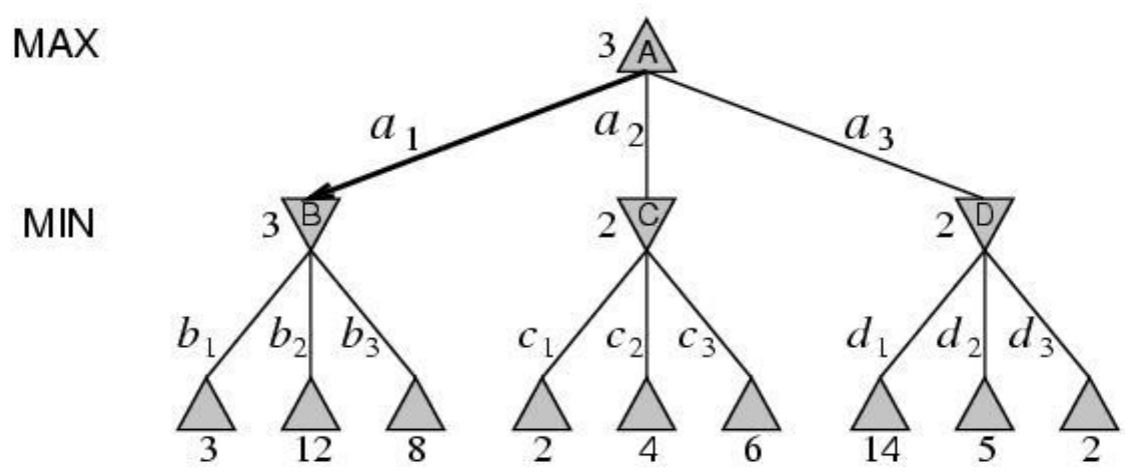
- At each step, select the move that leads to the terminal state with the highest utility for the player
  - if the other player (MIN) is assumed to make its best move every time, then the resulting strategy of the player (MAX) is optimal
- Estimate the utility of the nodes even if they are not terminal nodes
  - compute minimax value of each node
  - MAX selects the move to a state of maximum value
  - MIN selects the move to a state of minimum value

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{utility}(n), & \text{if } n \text{ is a terminal state} \\ \max_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s), & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successors}(n)} \text{MINIMAX-VALUE}(s), & \text{if } n \text{ is a MIN node} \end{cases}$$

Game of TIC-TAC-TOE



Using minimax to Play Tic-Tac-Toe



- Initial state of the game: node A
- How should MAX play? Select a_1, a_2 or a_3?

- Assuming MIN plays optimally, MIN will select the moves that minimize the utility

  → minimax value is 3 for B, 2 for C, 2 for D (min. of successors)
- MAX selects the moves that maximize the utility

  → minimax value is 3 for A (max. of successors B, C and D)

→ In the initial state, MAX should select the action a_1

...and MIN should respond by selecting the action b_1

Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

return action corresponding to best move from this state

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MAX(v, MIN-VALUE(s))
    return v
```

back-up the values of terminal states

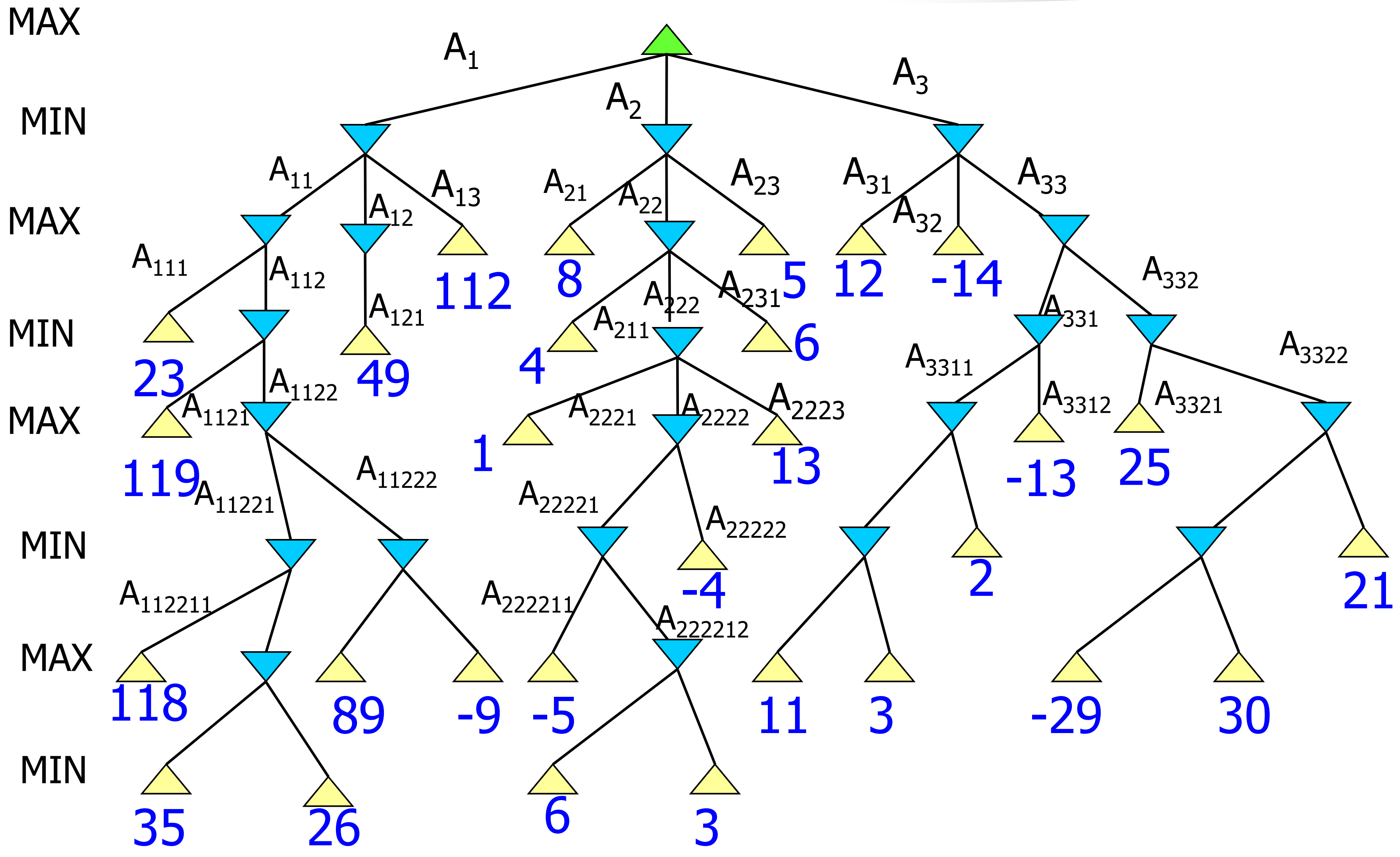
find successor with highest utility (minimax) value

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MIN(v, MAX-VALUE(s))
    return v
```

back-up the values of terminal states

find successor with lowest utility (minimax) value

Another Game

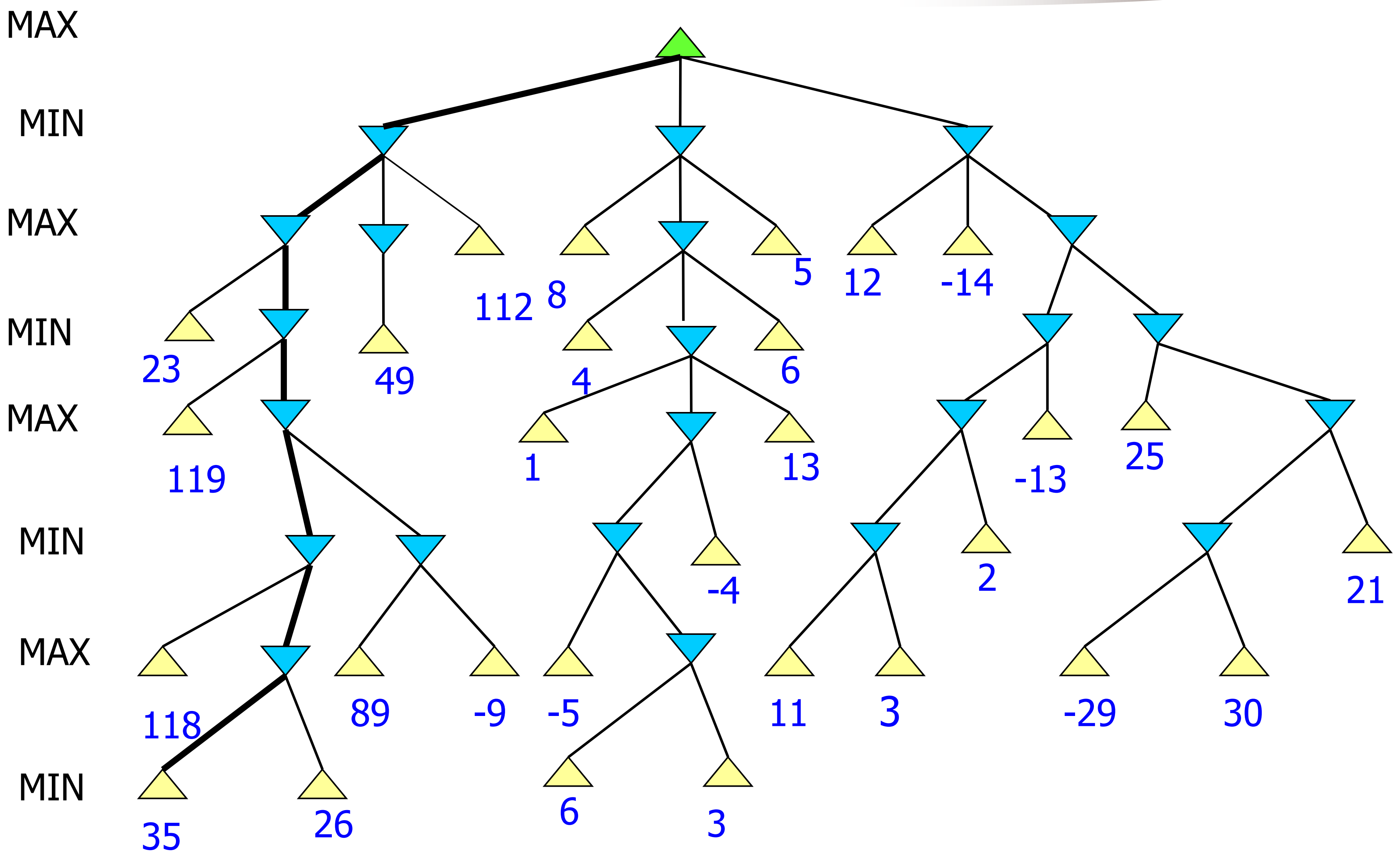


How Minimax works?

- To help MAX find the optimal strategy, we will use Minimax Algorithm, for now. This algorithm has 5 steps:

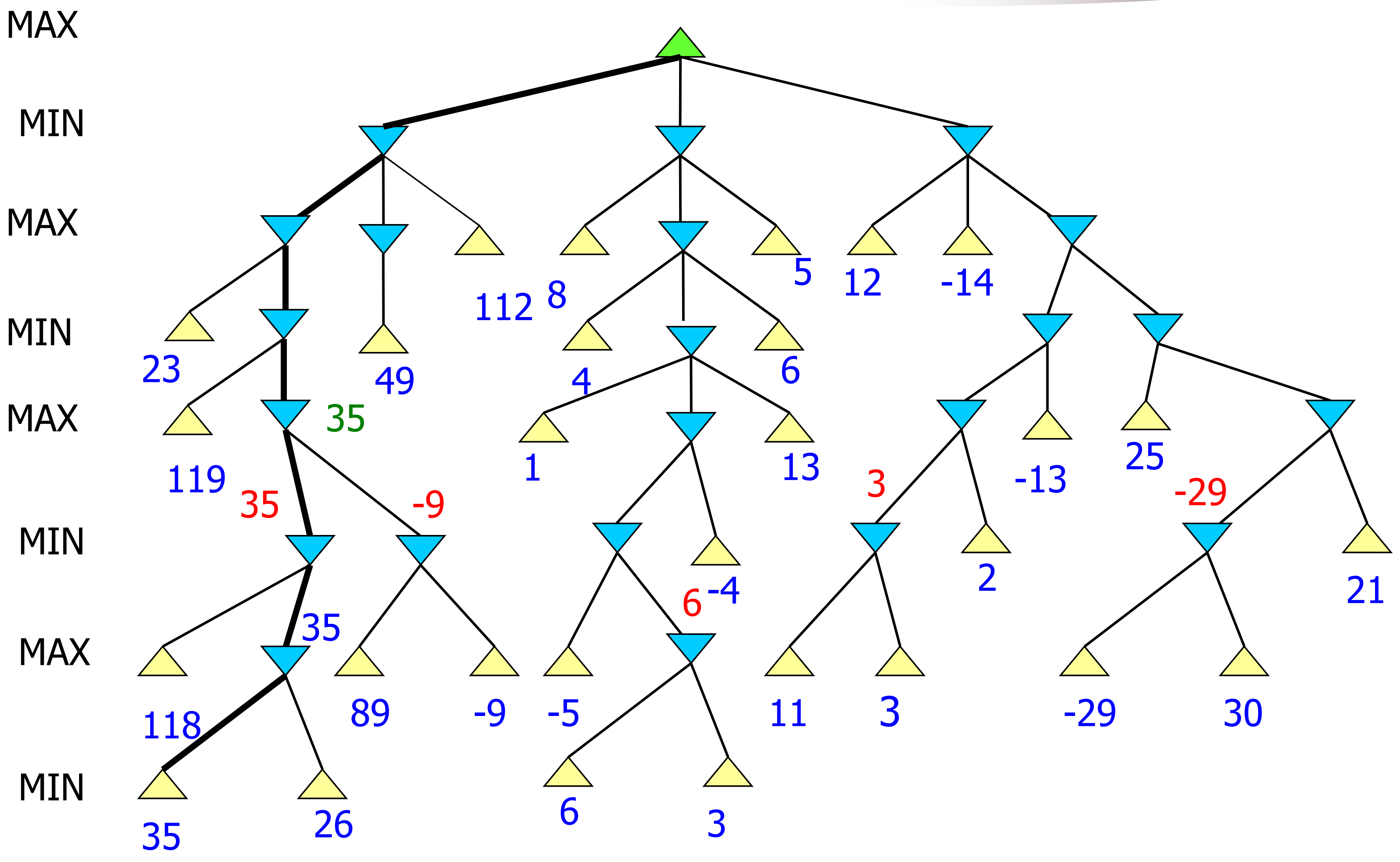
1. Generate the WHOLE game tree
2. Apply the utility function to all terminal states to get their utility value
3. Use the terminal state utility value to determine their parent's utility
4. Repeat step 3 for each higher level until you get to the root of the game tree
5. Pick a move that leads to the highest utility value

Who wins???

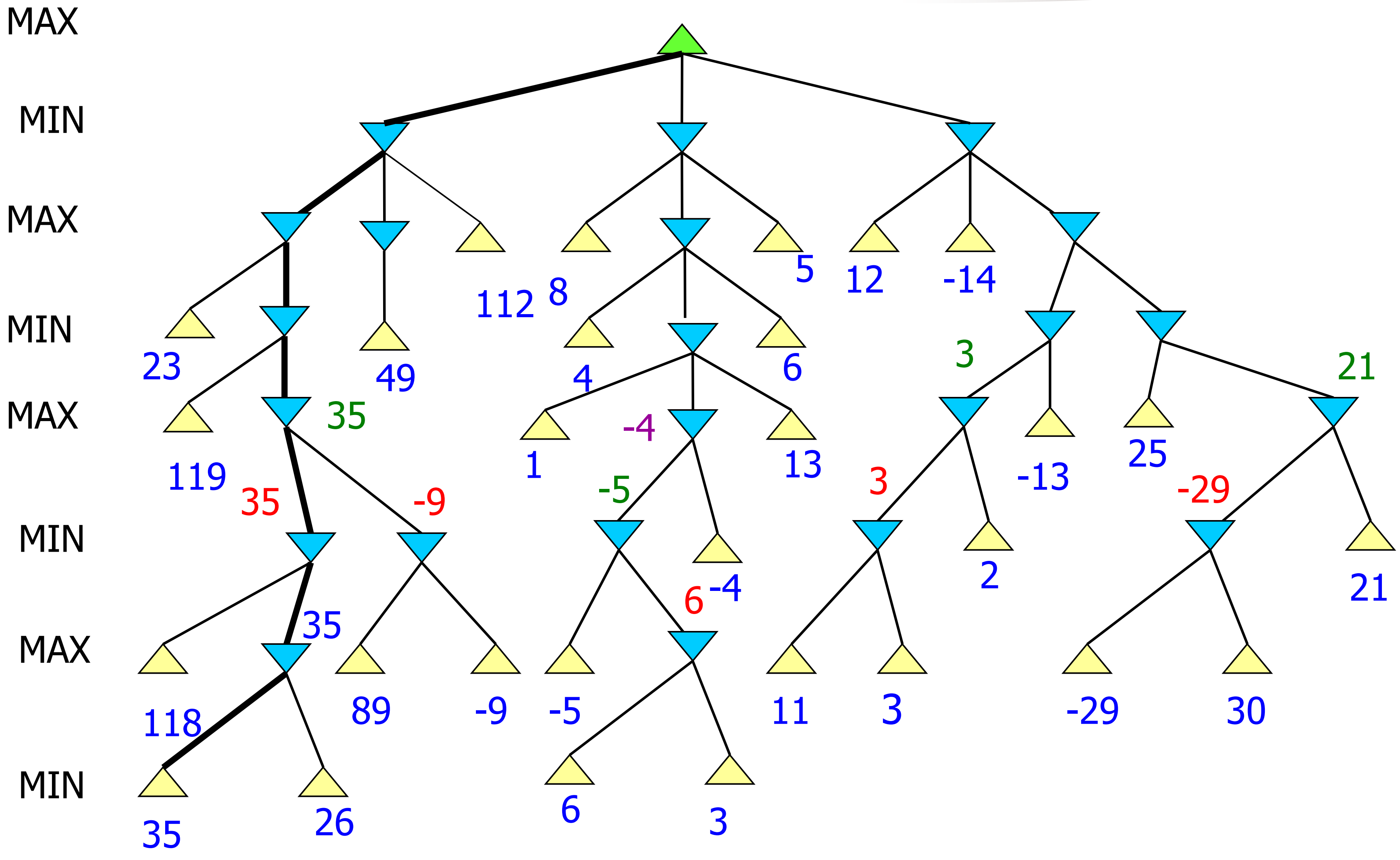


MAX
MIN
MAX
MIN
MAX
MIN
MAX
MIN
23
49
112
8
5
12
-14
4
6
119
1
13
-13
25
-4
2
21
118
89
-9
-5
11
3
-29
30
35
26
6
3

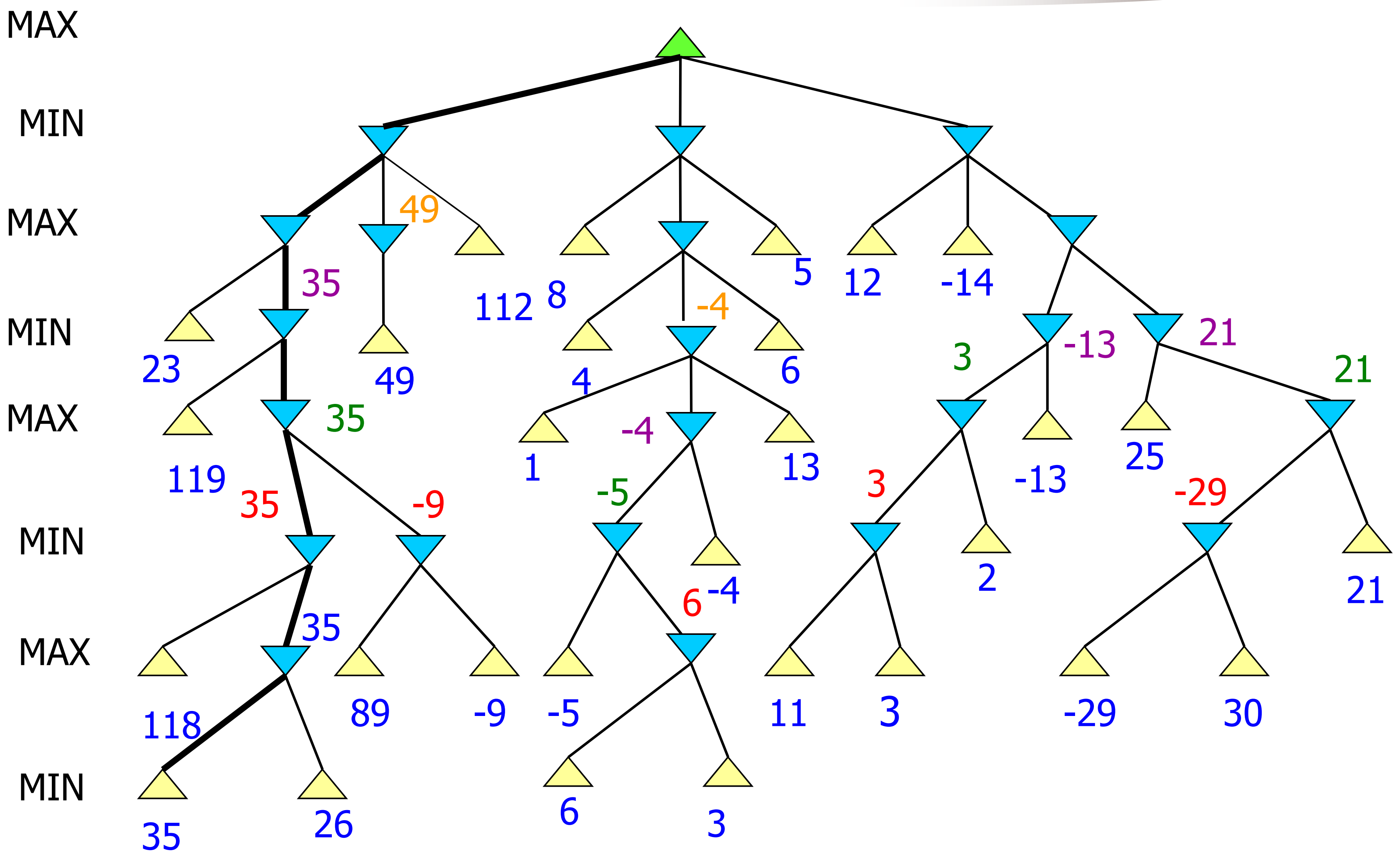
Who wins??? – Determine utilities



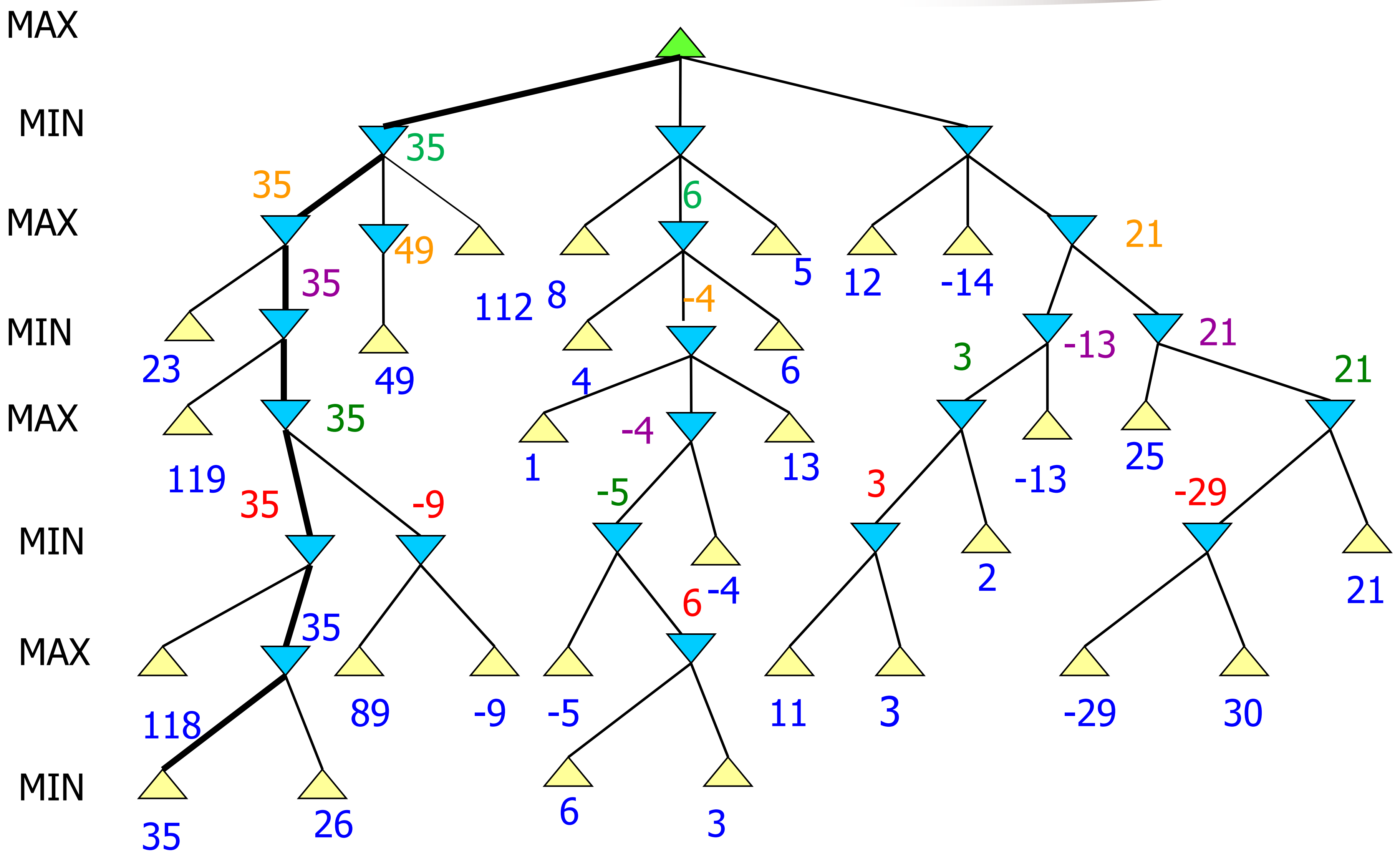
Next


MAX
MIN
MAX
MIN
MAX
MIN
MAX
MIN
23
119
35
35
35
118
35
26
49
112
-9
89
-9
8
4
1
-4
-5
-5
6
-4
6
3
5
6
13
12
-14
3
3
3
11
3
-13
2
25
21
-29
21
-29
30

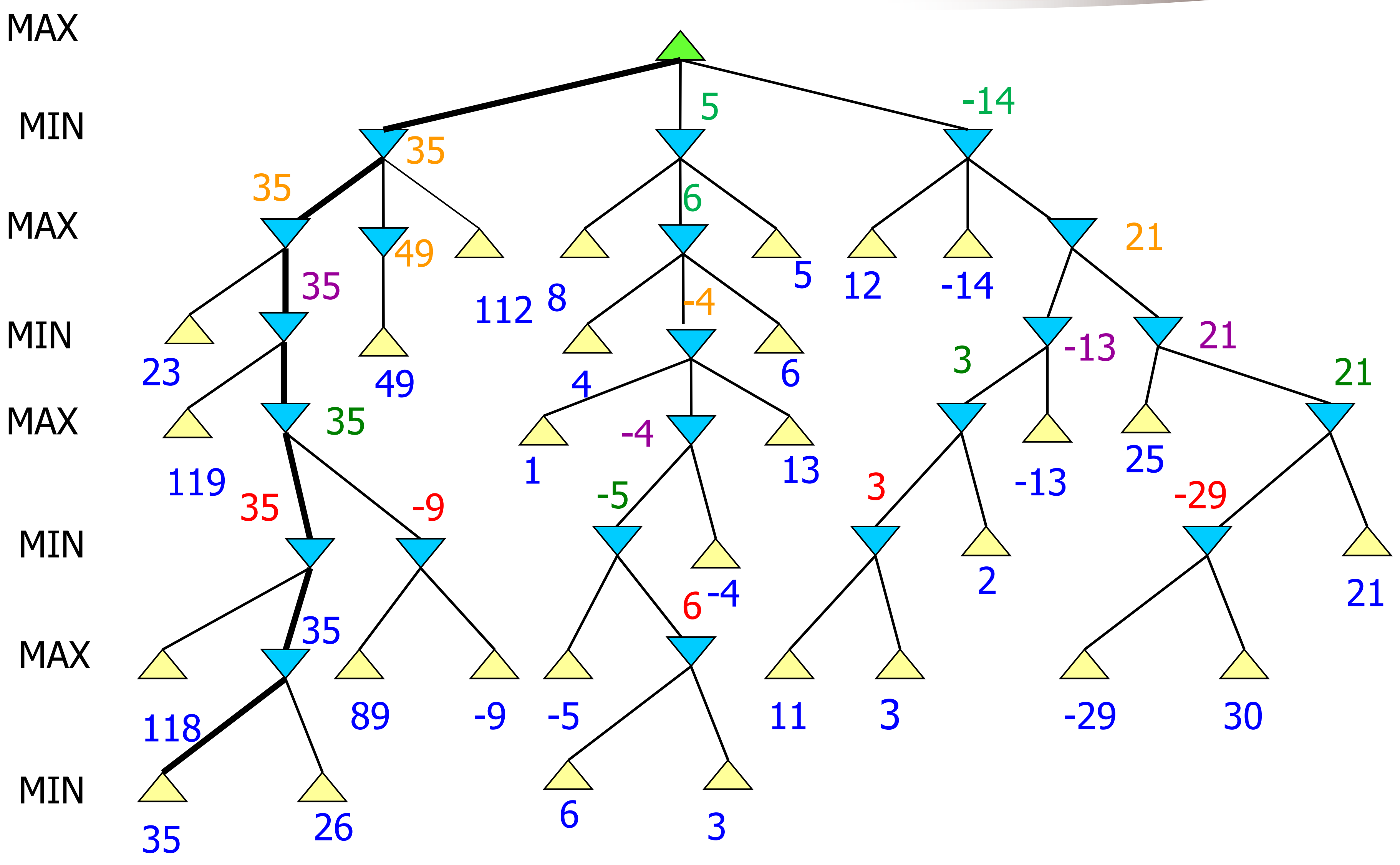
More


MAX
MIN
MAX
MIN
MAX
MIN
MAX
MIN
49
35
112
8
5
12
-14
-4
23
49
4
6
3
-13
21
35
21
4
-4
119
1
13
25
35
-9
-5
3
-13
-29
6
-4
2
21
35
118
89
-9
-5
11
3
-29
30
35
26
6
3

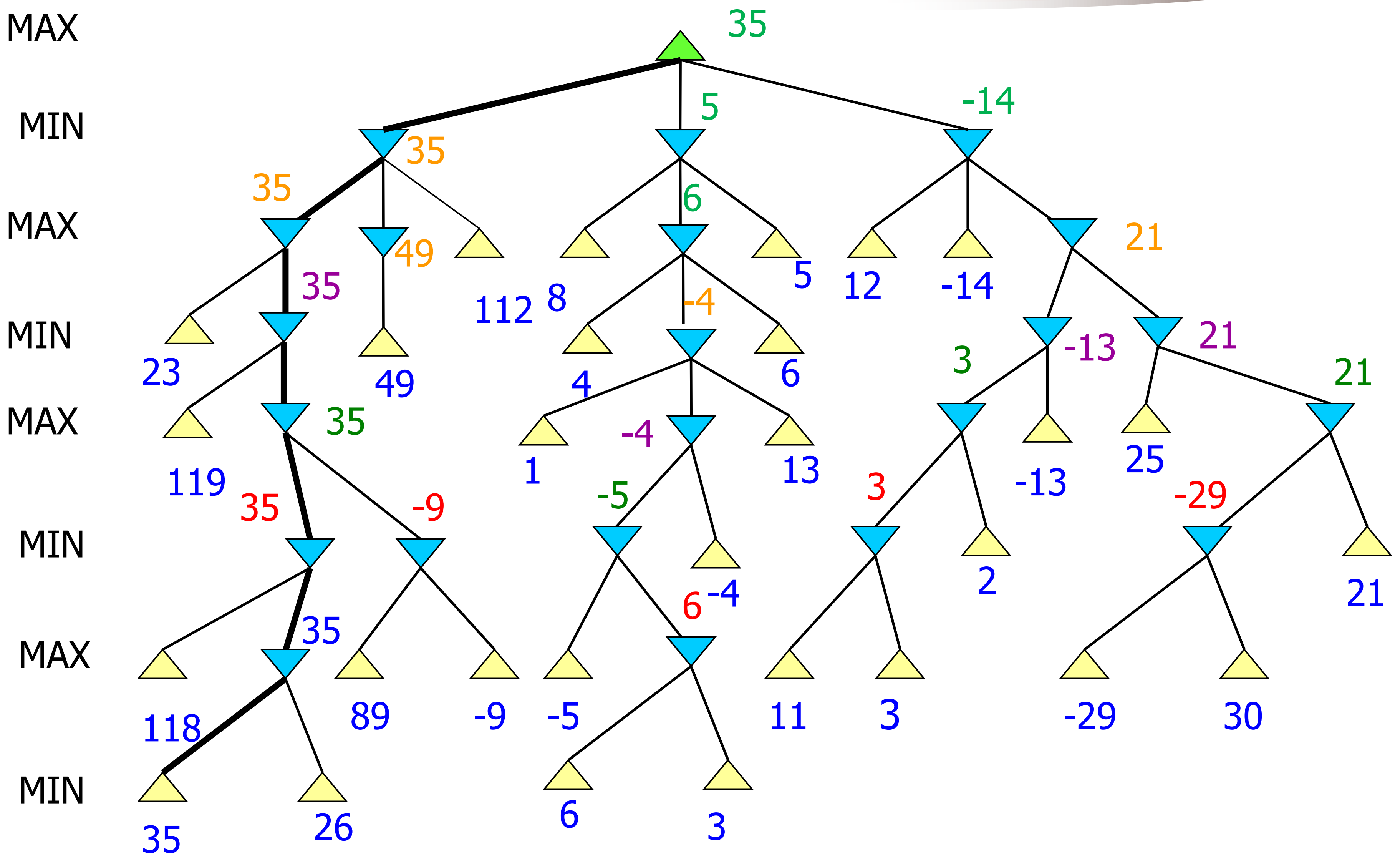
Minimax values



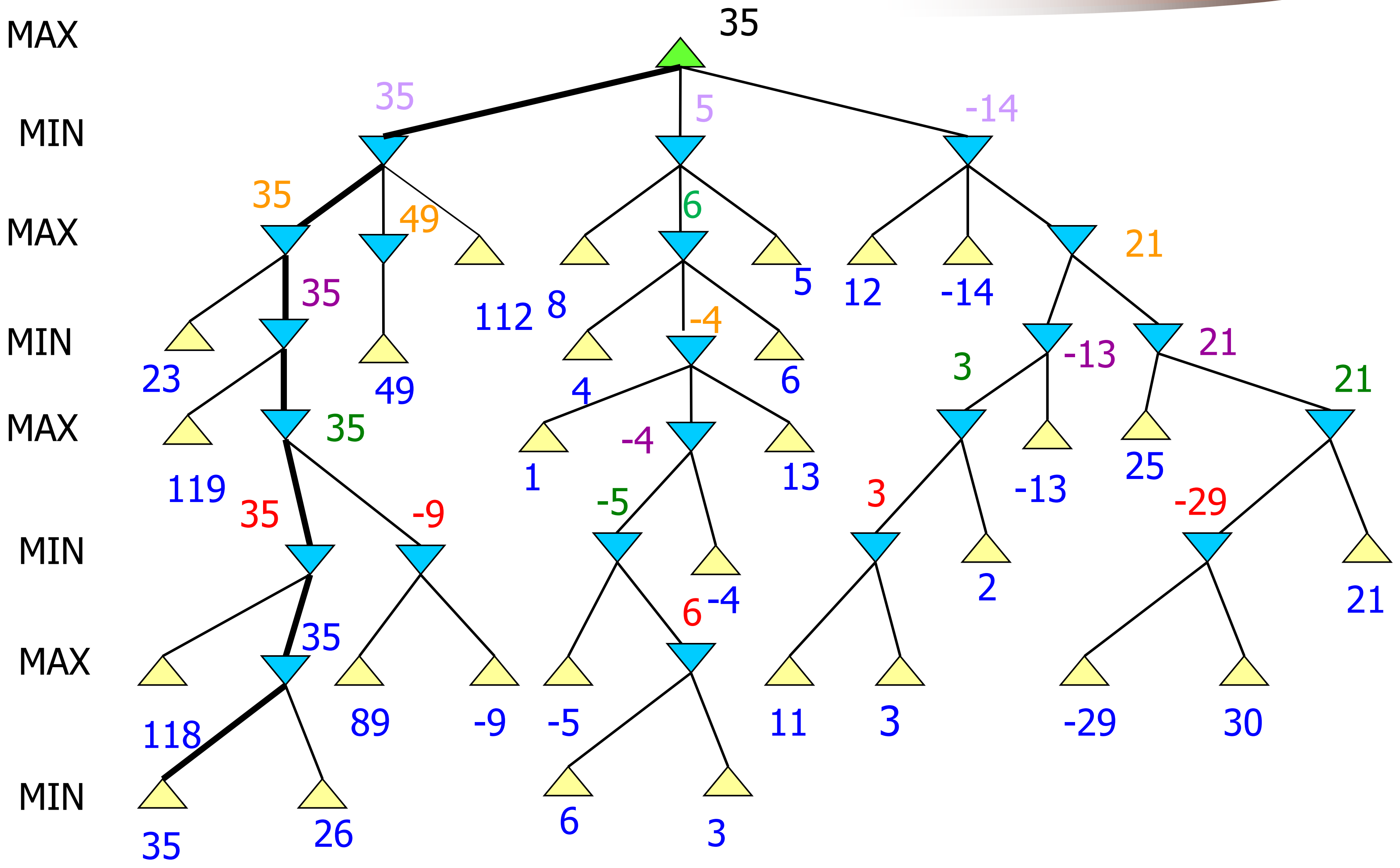
More Minimax values



Informed Decision

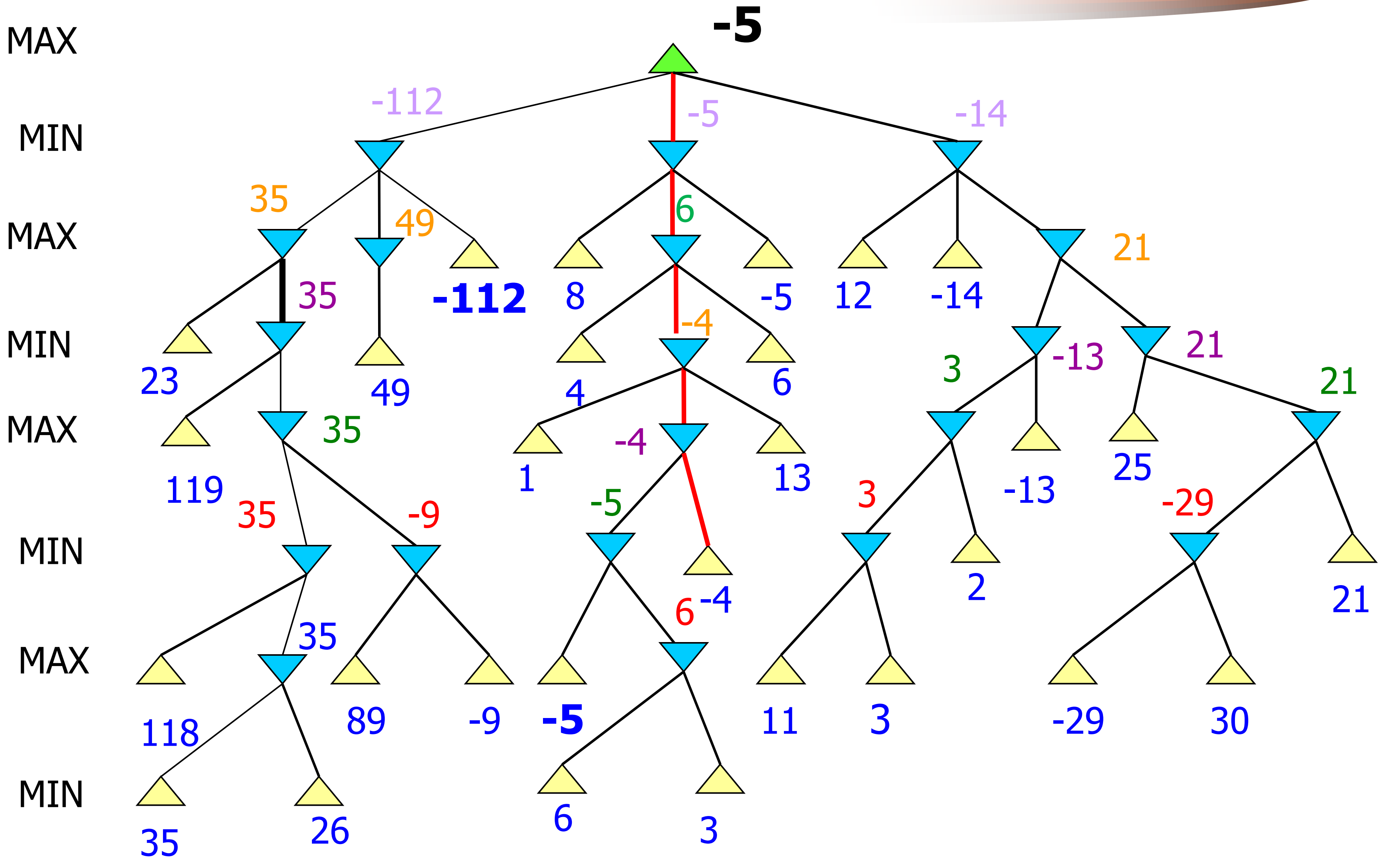


Who won???



MAX
MIN
MAX
MIN
MAX
MIN
MAX
MIN
35
35
5
-14
35
49
6
21
35
112
8
5
12
-14
-4
23
49
4
6
3
-13
21
35
4
-4
21
119
1
13
25
35
-9
-5
3
-13
-29
6
-4
2
21
35
118
89
-9
-5
11
3
-29
30
35
26
6
3

Statistics are not important! MIN wins!!



Analysis of minimax

- Perform depth-first search of game tree
- Space complexity? $O(bm)$
  - where m is the maximum depth of the tree
  - complexity reduced to $O(m)$ if only one successor is generated at a time
- Time complexity? $O(b^m)$
  - prohibitive for real games

Properties of Minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (depth-first exploration)

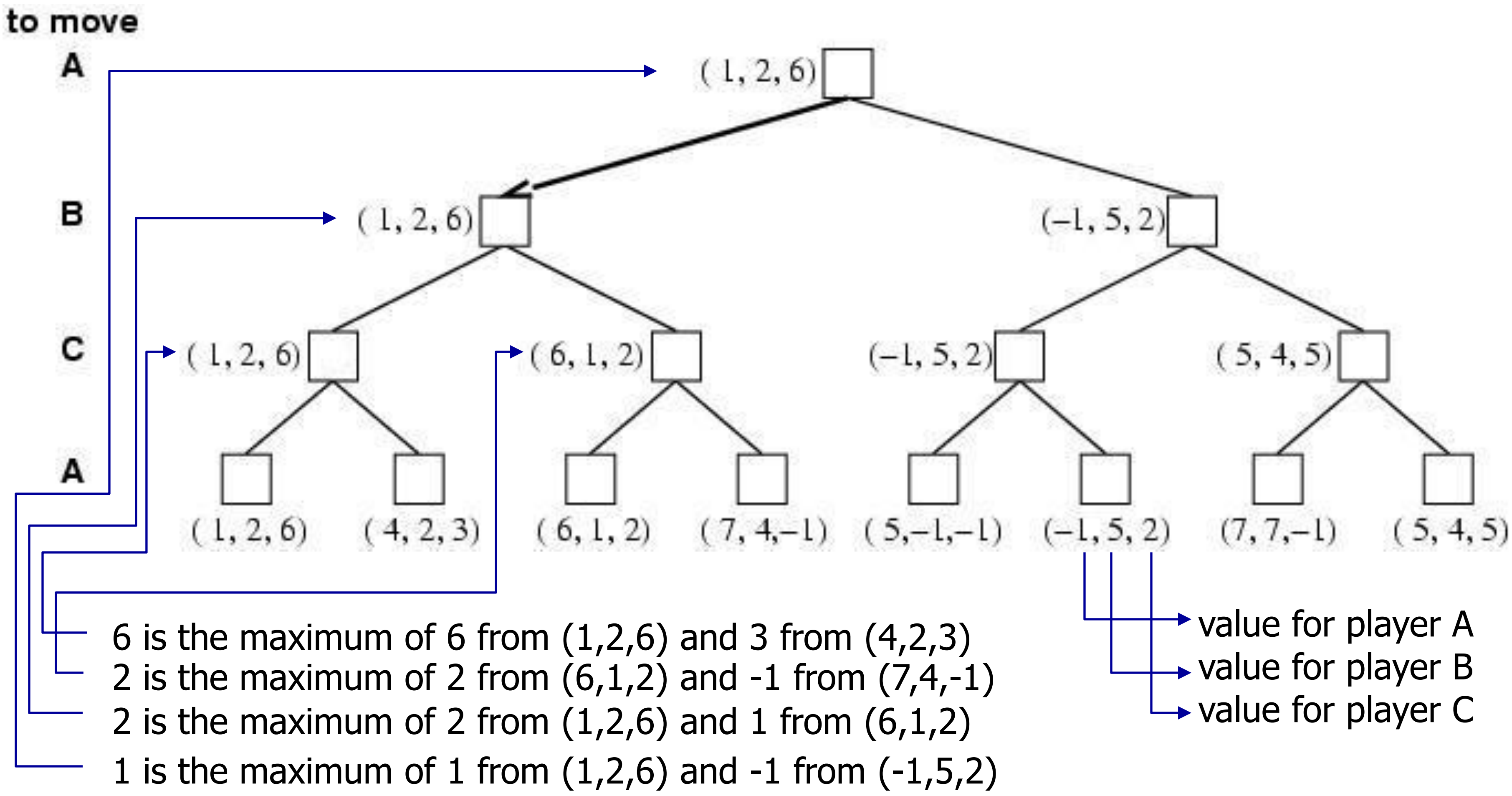
For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
→ exact solution completely infeasible

Multi-Player Games and minimax

- Extend minimax to games with more than two players:

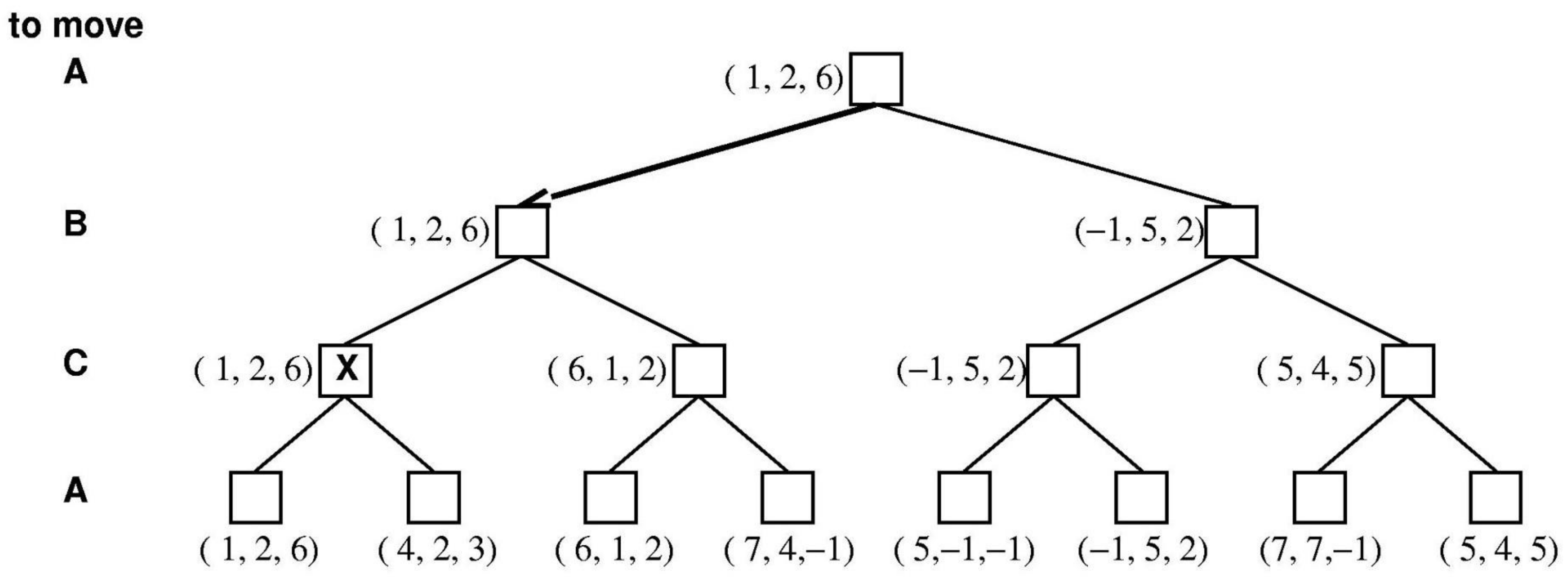
- Compute an **utility** **vector** (rather than an utility value) for each node
  - the utility vector contains a value for each player
- At each step, select the vector that has the maximum utility value for the current player that is about to move
  - back-up the **entire** **vector** (not only the value for this player)

Example of Multi-Player minimax



6 is the maximum of 6 from (1,2,6) and 3 from (4,2,3)
2 is the maximum of 2 from (6,1,2) and -1 from (7,4,-1)
2 is the maximum of 2 from (1,2,6) and 1 from (6,1,2)
1 is the maximum of 1 from (1,2,6) and -1 from (-1,5,2)

First 3-ply of TIC-TAC-TOE



Problems with MINIMAX

- The number of game states it has to examine is exponential in the number of moves.
  - Would be nice to try to evaluate the correct minimax decision without looking at the whole tree

- *Alpha-beta pruning of the game tree*
  - *How?*

Alpha-beta pruning

Utilizes minimax search, but keeps track of more information.

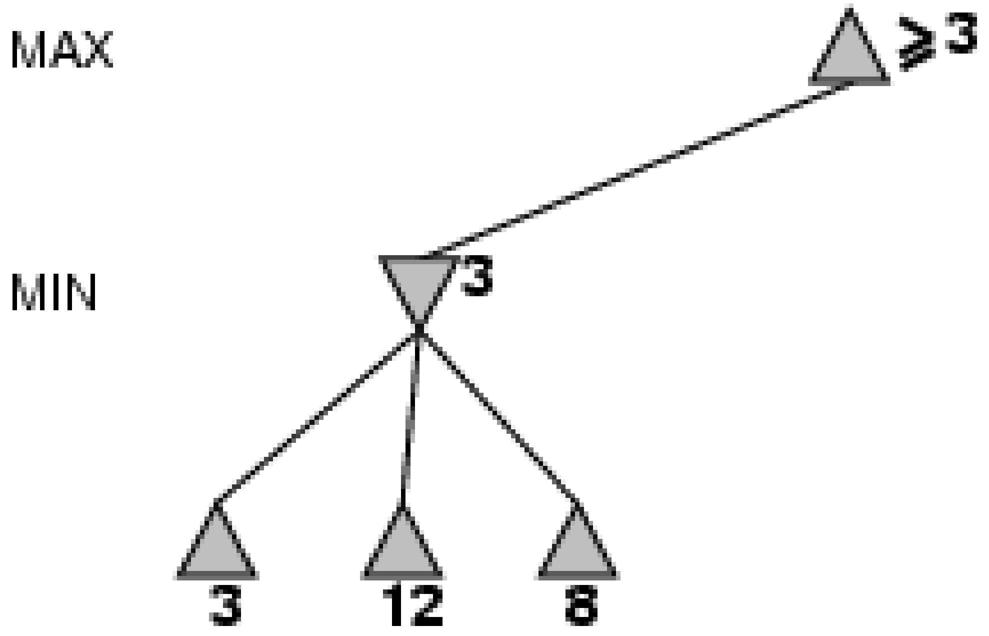
Alpha-beta **pruning** eliminates branches of a standard minimax tree that cannot affect the final outcome of the move.

What is the idea?

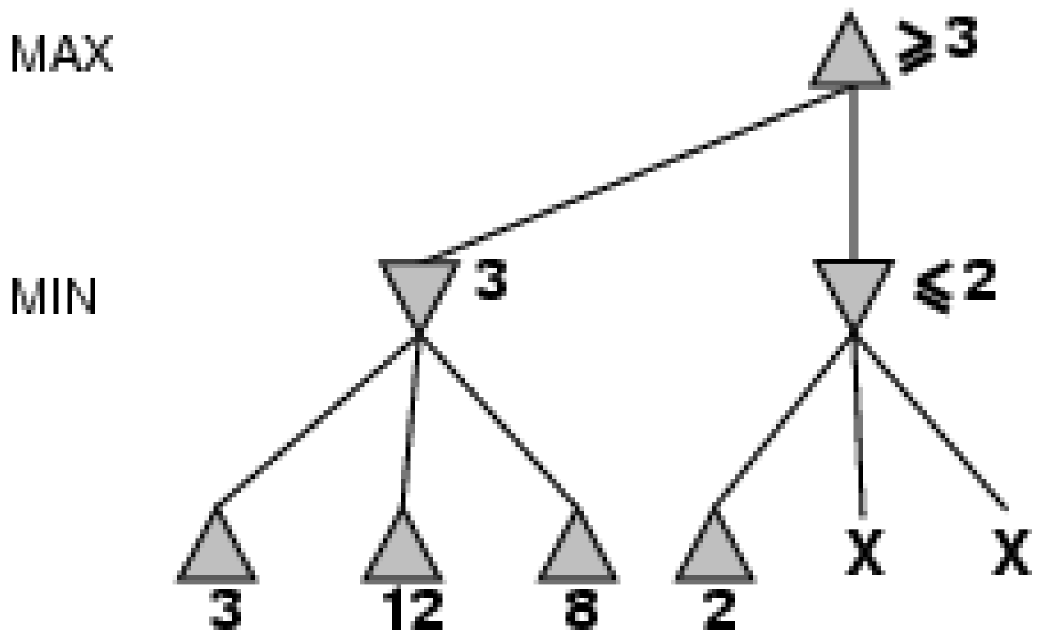
Computes the minimax decision without looking at every node in the tree. How?

- ➢ it eliminates branches from the tree
  - a process known as pruning

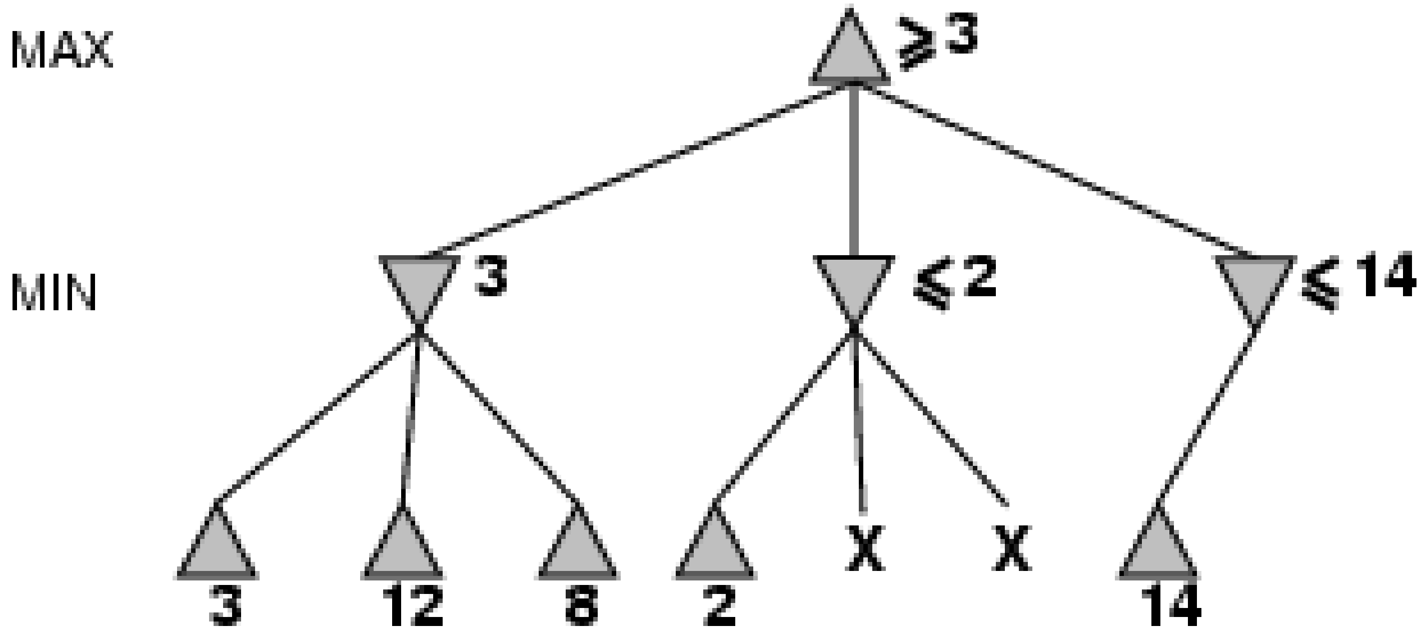
α-β pruning example



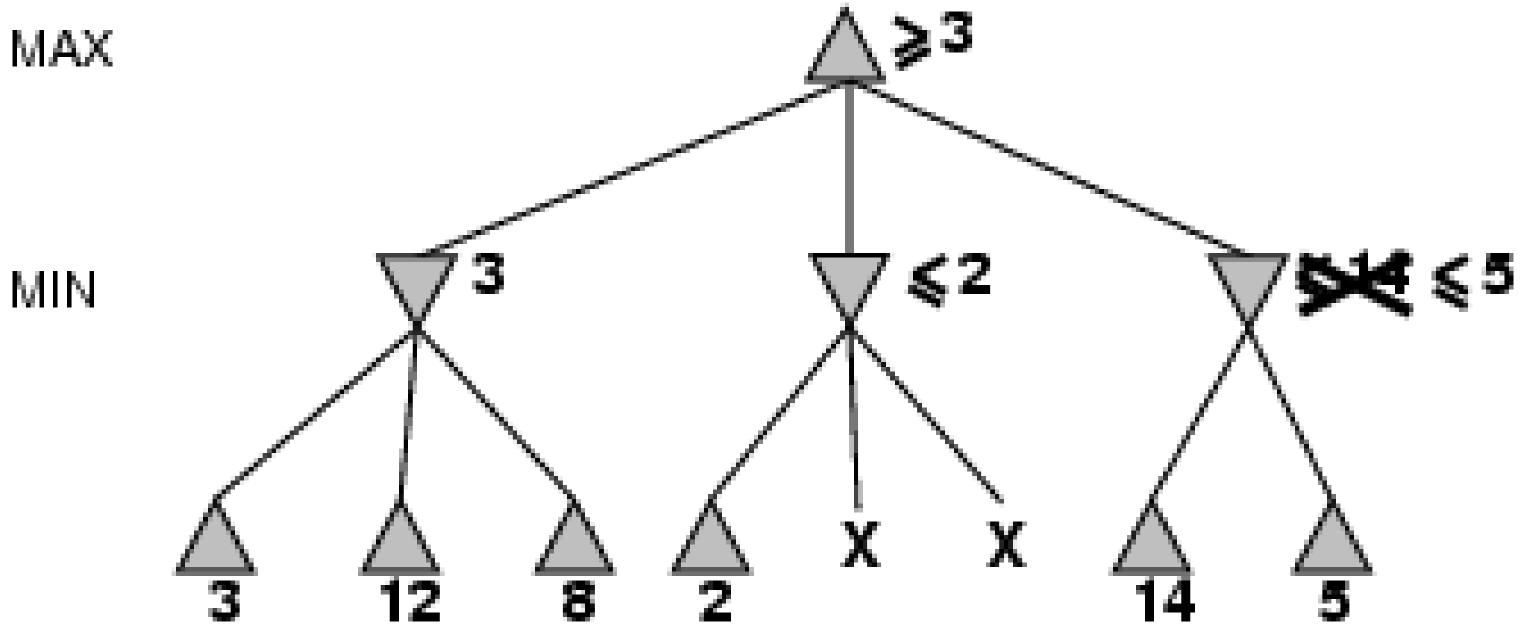
α-β pruning example



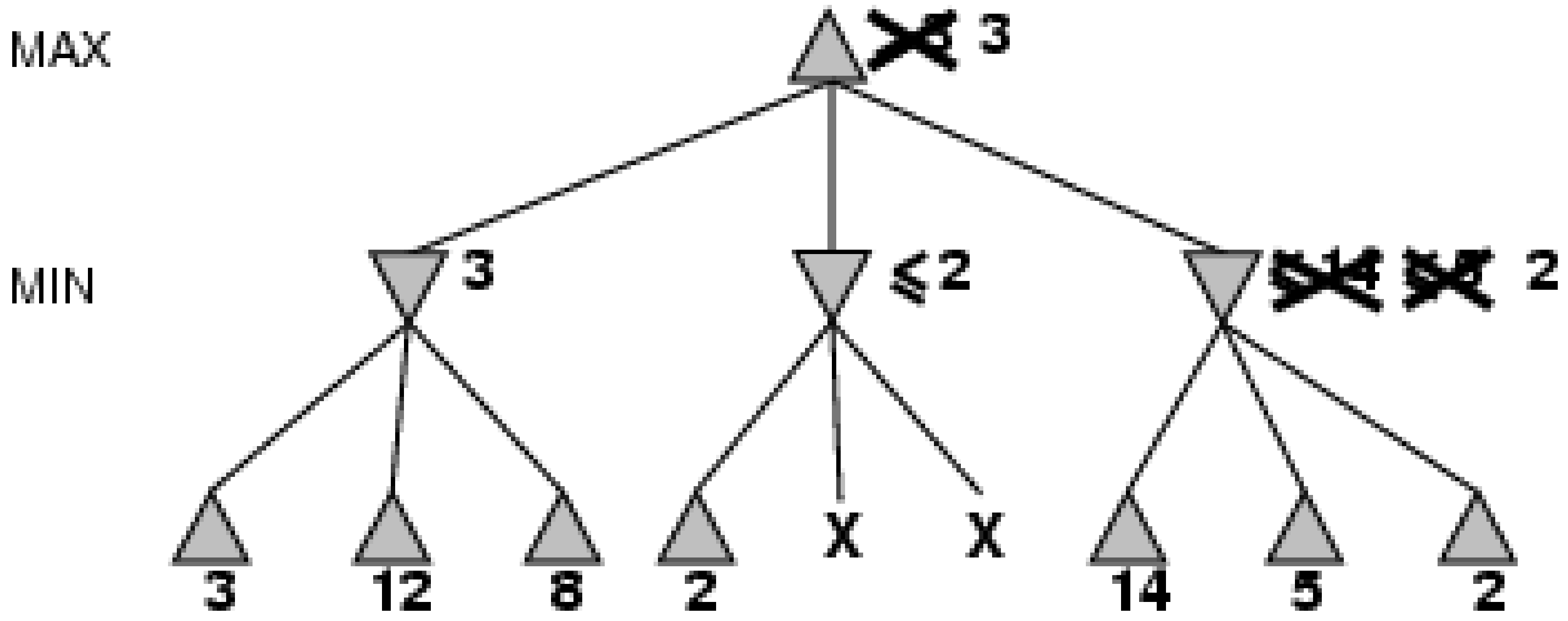
α-β pruning example



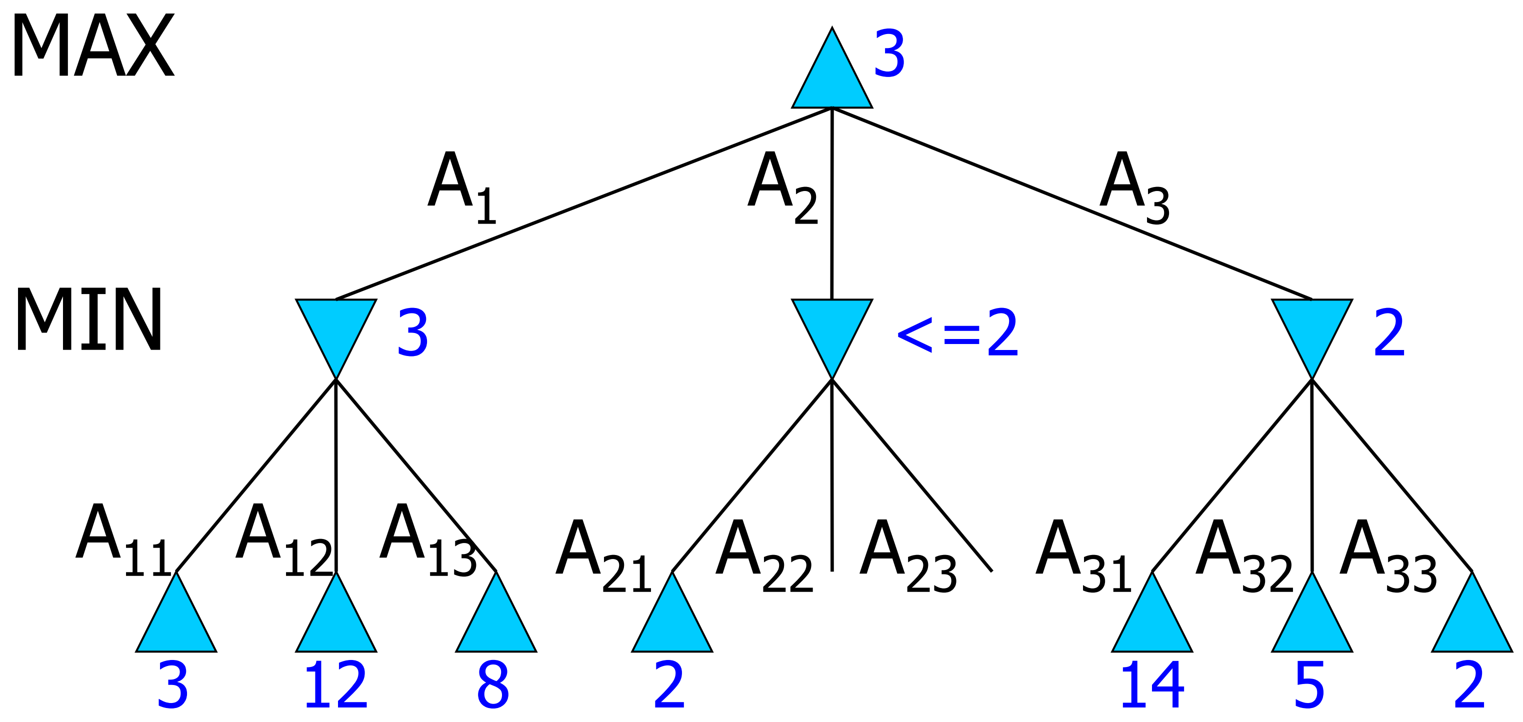
α-β pruning example



α-β pruning example



How it works

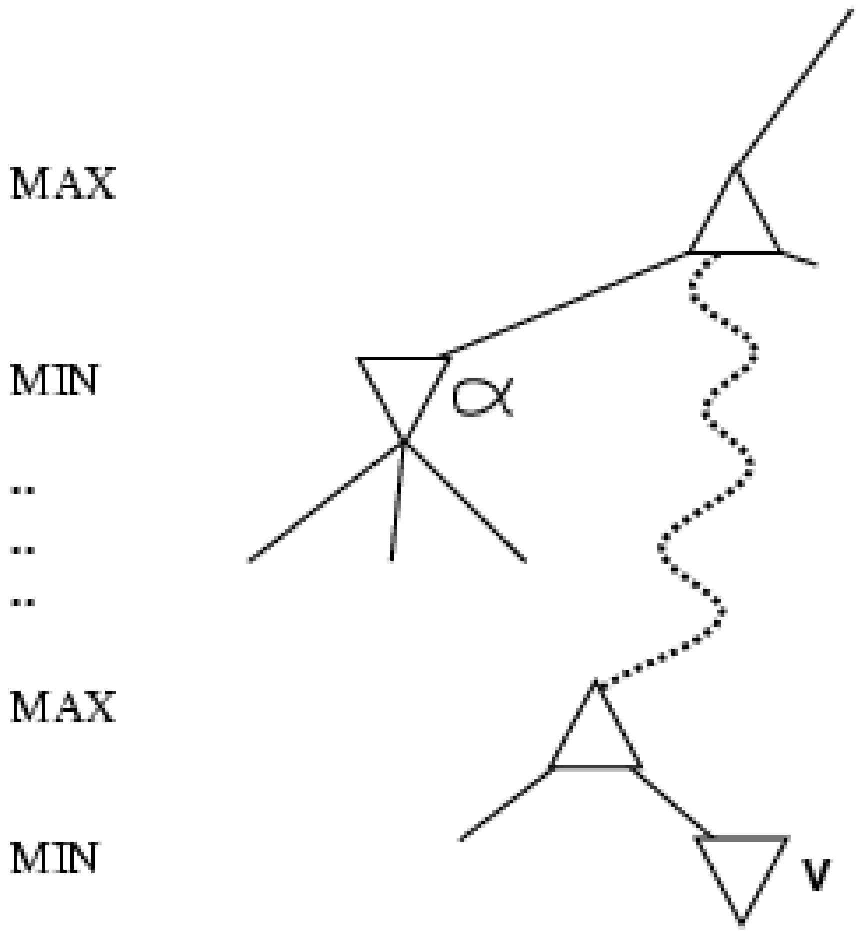

MAX
MIN
3
A1
A2
A3
3
<=2
2
A11
A12
A13
A21
A22
A23
A31
A32
A33
3
12
8
2
14
5
2

Why is it called α-β?

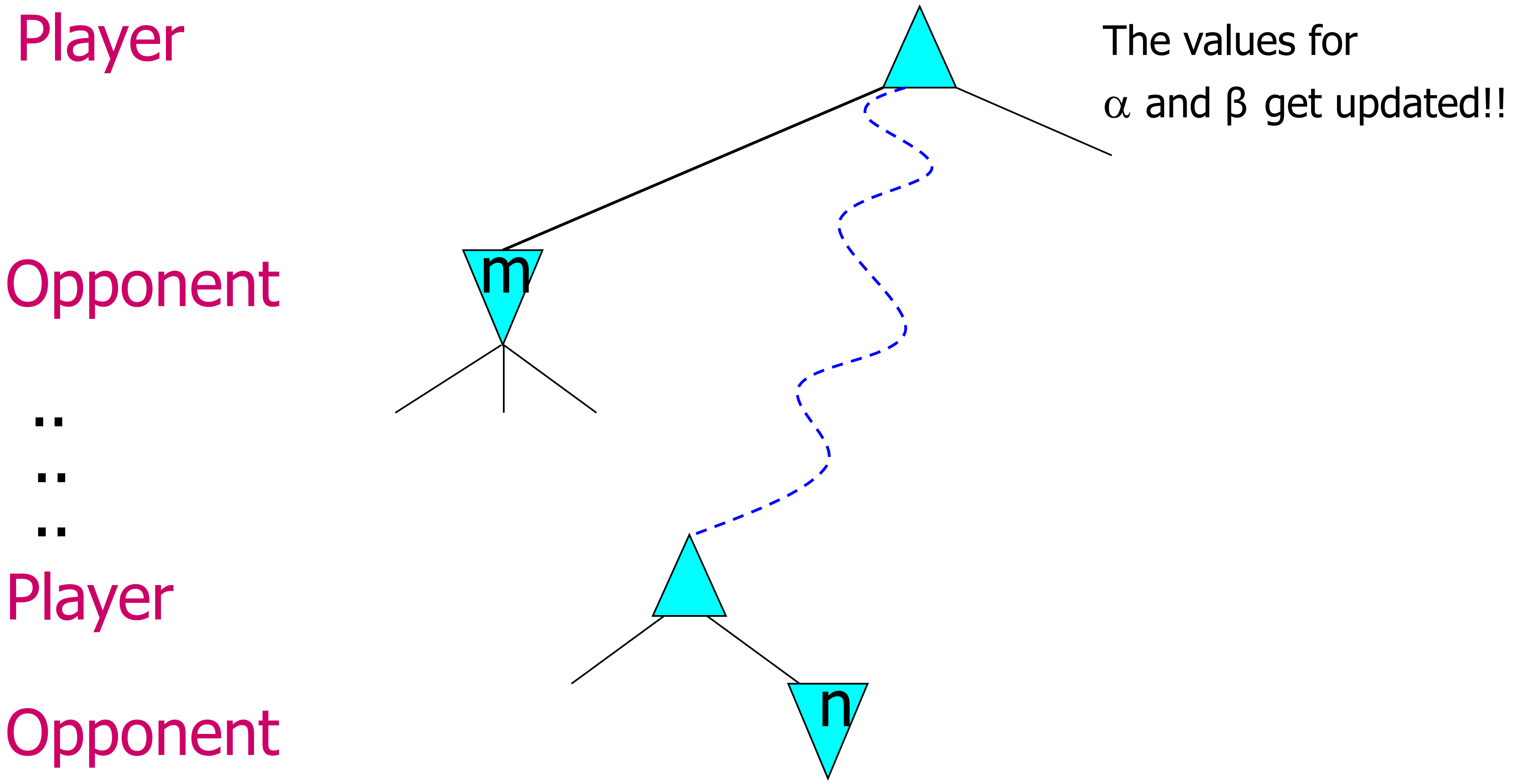
- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*
-
- If *v* is worse than α, *max* will avoid it
-

  → prune that branch

- Define β similarly for *min*



General principle



The values for

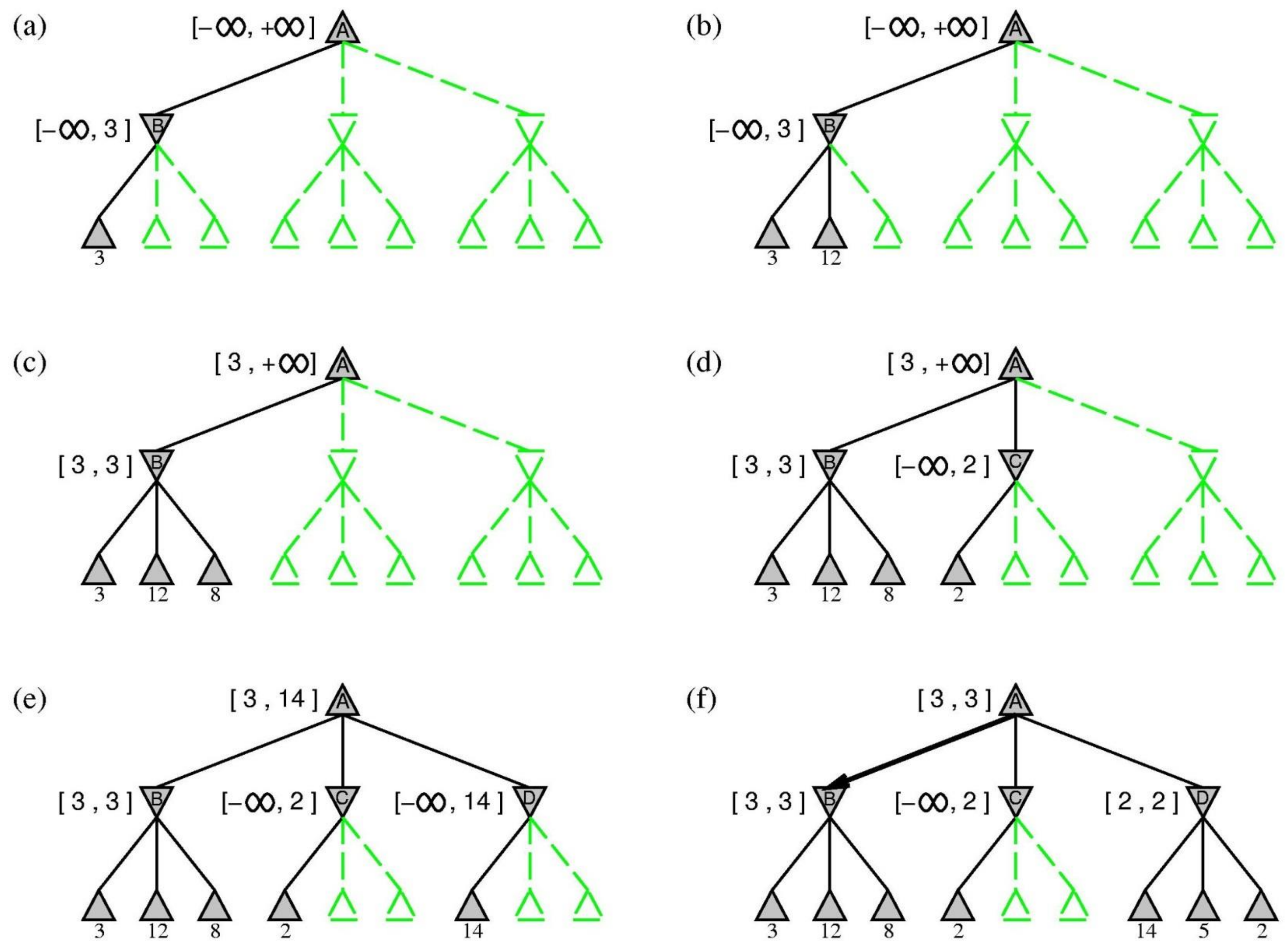
α and β get updated!!

The idea: if *m* is better than *n* for Player, we will never get to *n* in play

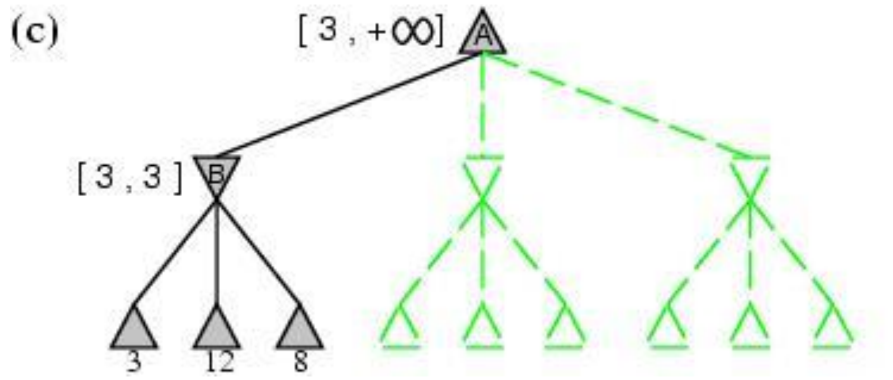
Name of alpha-beta

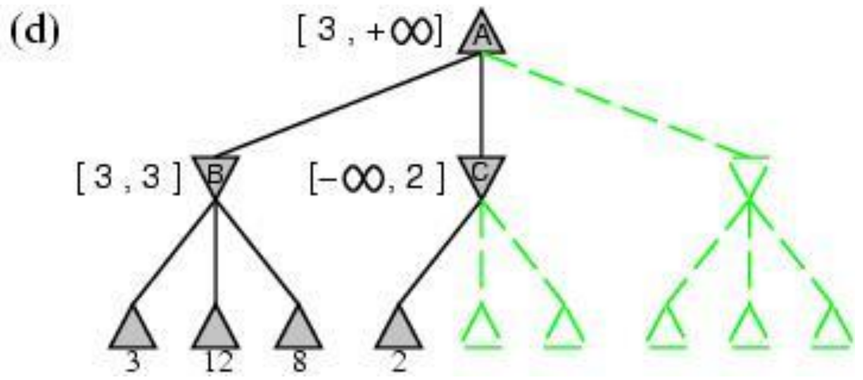
- From two parameters that describe the backed-up values that appear anywhere along the path:
  - α = the value of the best (highest-value) choice we have found so far at any choice point along the path for MAX
  - β = the value of the best (lowest-value) choice we have found so far for at any choice point along the path for MIN

Alpha-beta pruning



A Concrete Example of Alpha-Beta Pruning

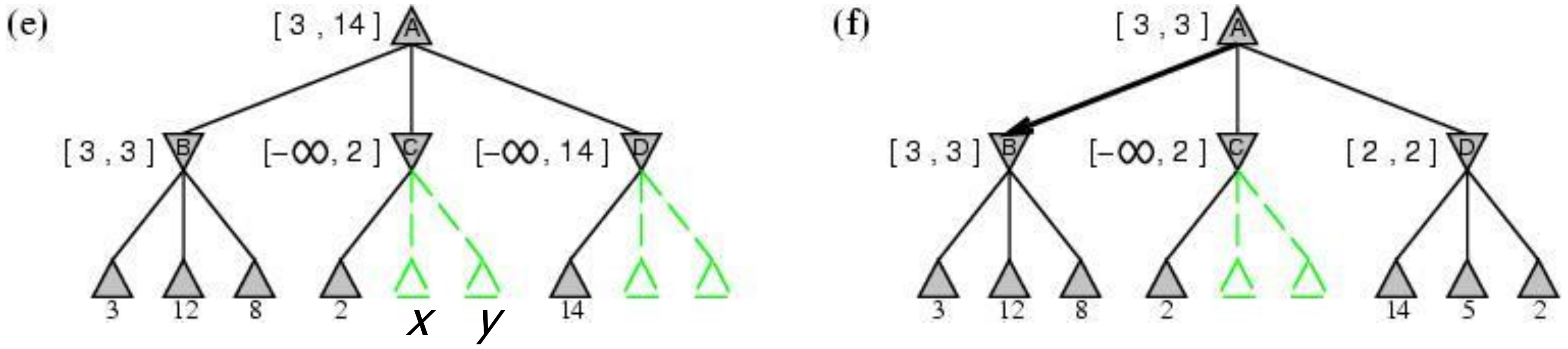




- After (c):
  - all descendants of B have been visited → value of B is 3
- After (d):
  - C is a MIN node; the first descendant of C has been visited

  → the minimax value of C will be at most 2 (it can be lower than 2 since not all descendants of C have been visited)
  - ..but A is a MAX node that will choose the highest-utility descendant → when MAX must choose in A, it may choose B but definitely not C → prune rest of the tree under C

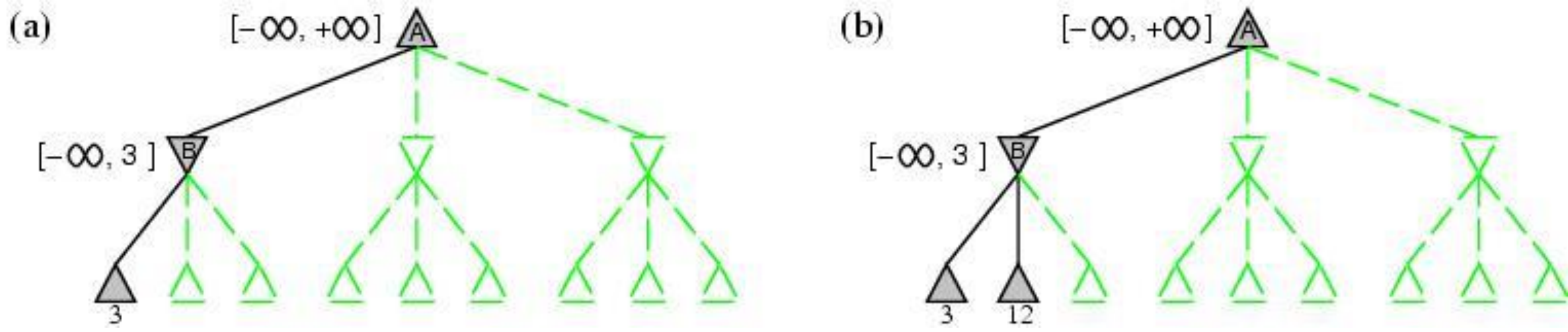
Simplification of *MINIMAX-VALUE*



- MINIMAX-VALUE (root) = max (min(3,12,8), min(2,x,y), min(14,5,2))
  = max (3, min(2,x,y),2)
  =max (3,z,2) where $z \leq 2$
  = 3

The value of the root and the minimax decision are independent of the values pruned at leaves *x* and *y*

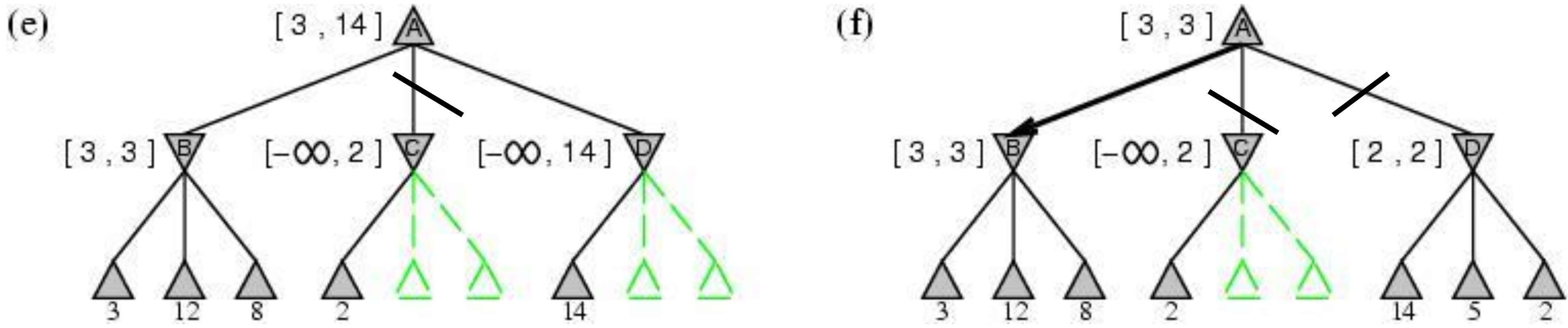
How do we prune the game tree?



- After (a):
  - B is a MIN node
  - the first descendant of B has been visited

  → the minimax value of B will be at most 3 (it can be lower than 3 since not all descendants of B have been visited)
- After (b):
  - the second descendant of B has been visited
  - no impact on the value of B

How do we prune the game tree?



- After (e):
  - D is a MIN node; the first descendant of D has been visited

    → the minimax value of D will be at most 14 (it can be lower than 14 since not all descendants of D have been visited)
- After (f):
  - all descendants of D have been visited → value of D is 2
  - value of A is the maximum of its descendants → value of A is 3

Alpha-beta search

```
function ALPHABETA-SEARCH (state)
                         returns an action
Inputs: state, current state in game

    v ← MAX-VALUE(state, $-\infty$, $+\infty$ )
return the action in SUCCESSORS[state] with value v
```

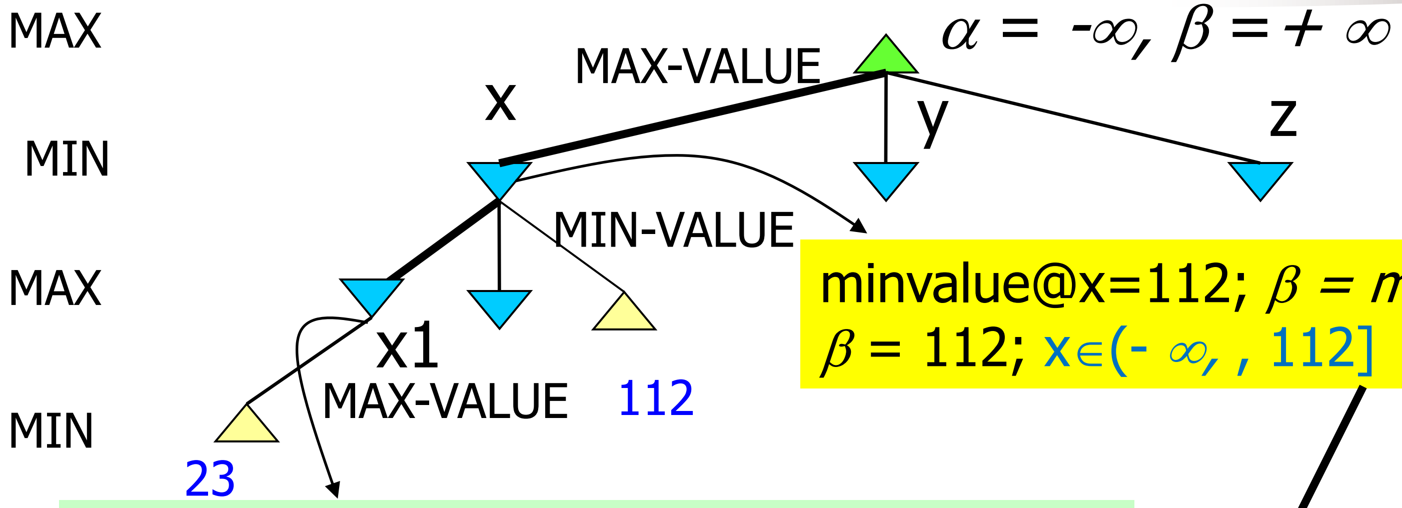
```
function MAX-VALUE (state, $\alpha$ , $\beta$ ) returns a utility value
Inputs: state, current state in game
            $\alpha$, the best value for MAX along the path to state
            $\beta$, the best value for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← $-\infty$
    for a, s in SUCCESSORS[state] do
            v ← MAX(v, MIN-VALUE(s, $\alpha$, $\beta$))
            if $v \geq \beta$ then return v
            $\alpha \leftarrow$ MAX ($\alpha$ , v)
    return v
```

```
function MIN-VALUE (state, $\alpha$ , $\beta$ ) returns a utility value
Inputs: state, current state in game
            $\alpha$, the best value for MAX along the path to state
            $\beta$, the best value for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← $+\infty$
    for a, s in SUCCESSORS[state] do
            v ← MIN(v, MAX-VALUE(s, $\alpha$, $\beta$))
            if $v \leq \alpha$ then return v
            $\beta \leftarrow$ MIN($\beta$ , v)
return v
```

MORE α - β decisions



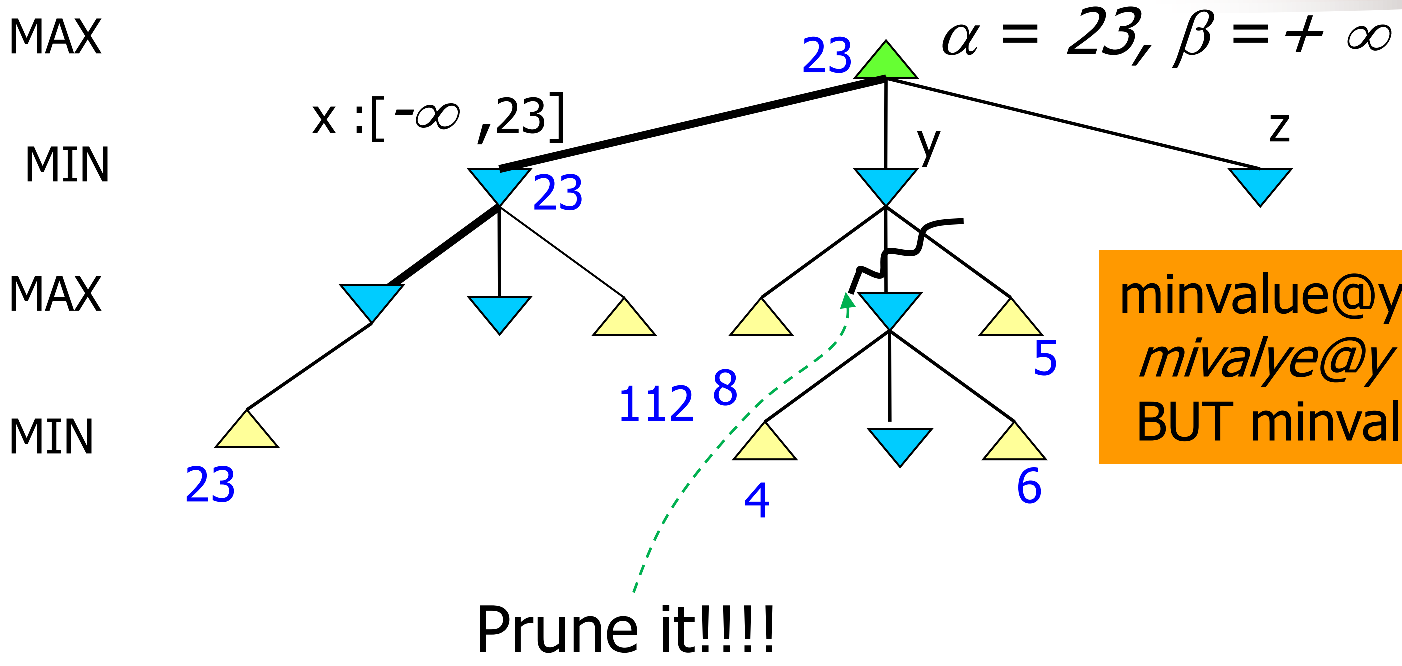
minvalue@x=112; β = *min(112, + ∞)*
β = 112; x∈(- ∞, , 112]

maxvalue@x1=23; α = *max(23, -∞)*
α = 23; x1∈[23, ∞)

x∈(- ∞, , 23]

minvalue@x=23; β = *min(23, β=112)*
β = 23; x∈(- ∞, , 23]

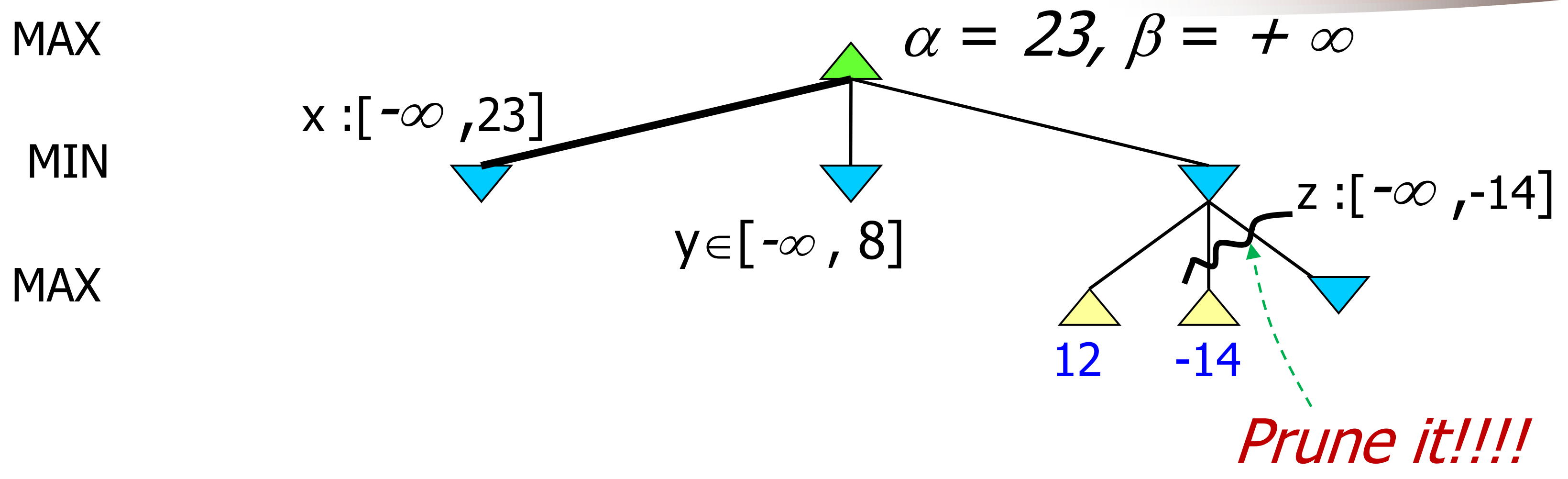
Even More α - β pruning



minvalue@y=8;
mivalye@y = min(8, ∞)=8
BUT minvalue @y ≤ α

Means pruning of remaining successors!!!

Final More pruning



minvalue@z=12; minvalue@z =min(12, β)=12
BUT minvalue@z =12$\leq$ α = *23*
Prune the remaining successors!!!

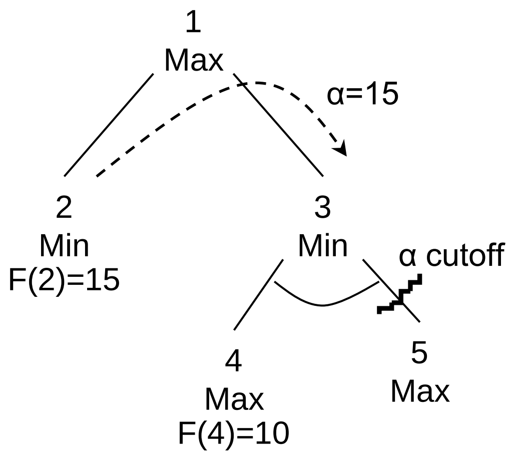
Alpha-Beta Pruning Procedure

The idea is to use the branch and bound strategy **to eliminate a path that is worse than the one already found**.

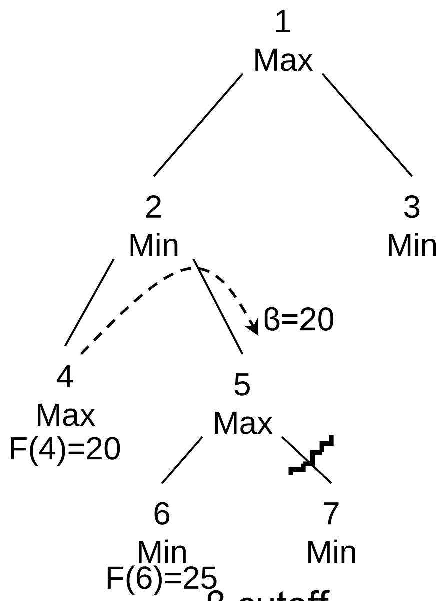
Here it is necessary to modify the branch and bound to include *two bounds*; one for each player.

α—is the lower bound of maximizing player

β—is the upper bound of minimizing player

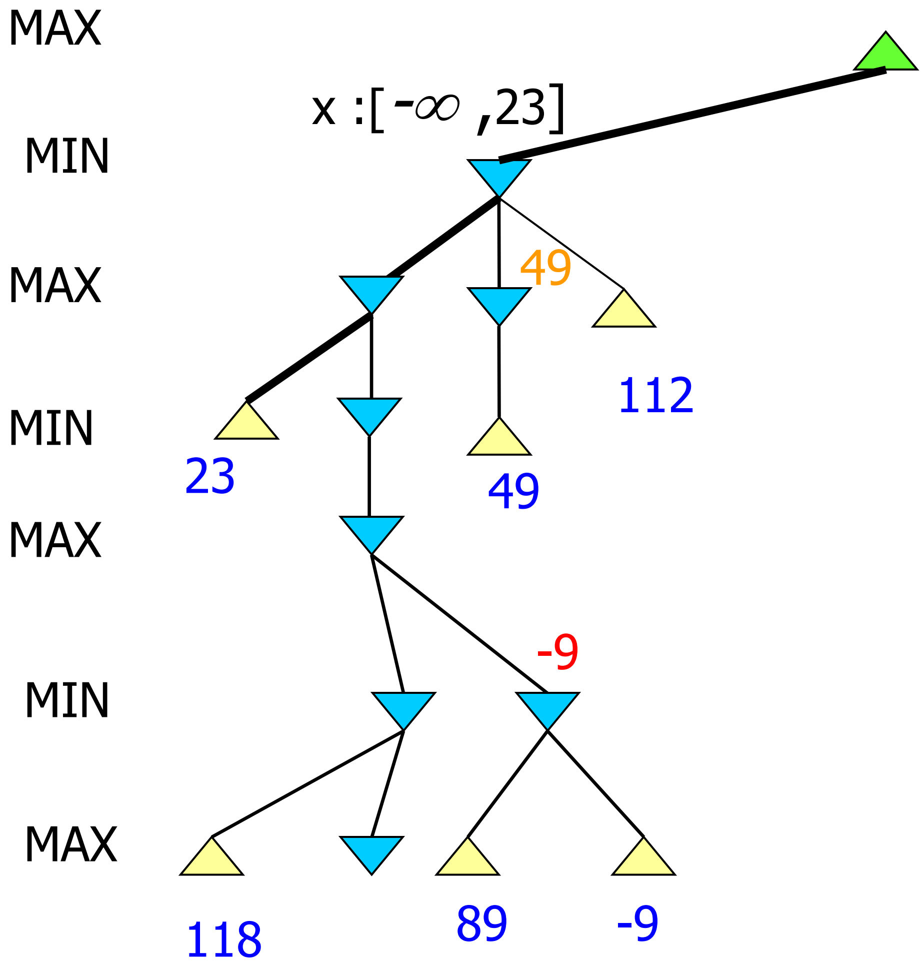


α cutoff
(pruning)



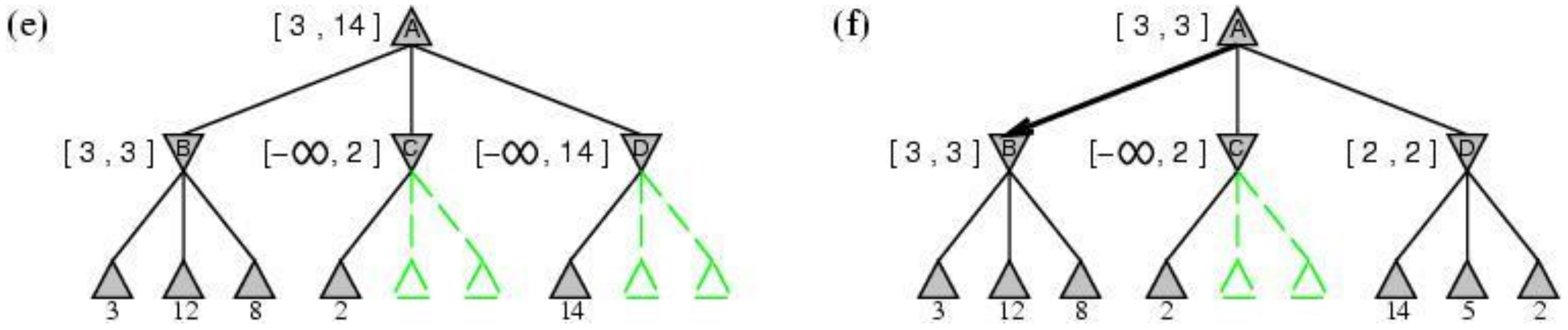
ß cutoff
(pruning)

Searching on a pruned tree



Move Ordering

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the game states are examined!!!!



We could not prune D in (e) because the worse successors were generated first!!! Try to examine first successors that are likely to be best!

The best moves are called **Killer moves**!

Imperfect decisions

- When you cannot generate the entire game tree whether because of time constraints, or the tree is too big to generate, you cannot use the Minimax algorithm as stated earlier.

- You must use a heuristic function to evaluate the leaf nodes that exist at the level where you had to stop. We call this the EVALUATION FUNCTION. In addition, you replace the terminal test with a CUTOFF TEST.

Imperfect decisions

Evaluation functions

The evaluation function returns an estimate of the EXPECTED UTILITY at a game position.

Most Evaluation Functions work by calculating various *features* of the game state.

Example: In chess, we can have features for: # of white pawns, #black horses, etc.

The features define various categories of *equivalence classes of states*: states in which each category has the same value of the features! The Evaluation Function can return a value that reflects the proportion of states with each outcome.

Example: In chess, 72% of states encountered in 2 pawns categories vs states with 1 pawn category led to win (utility = +1) 20% to loss (utility=0) and 8% to draw (utility=1/2)

$\rightarrow$*Expected Value=(0.72*$\times$(+1))+ *(0.2*$\times$(0))+ *0.08*$\times$(1/2))=0.76

Material advantage evaluation functions

Material advantage evaluation functions assign values to features of a game. For chess this could be the number of each type of piece on the board. These features are combined in a linear equation of the form:

$$\text{EVAL}(s)=\mathbf{w_1 f_1 + w_2 f_2 + \ldots + w_n f_n}$$

where **w** represents the weight, or importance, assigned to that feature and **f** represents each feature.

Cutoff Test

- Claude Shannon’s paper *Programming Computer for Playing Chess* (1950) proposed that programs should cutoff search early and apply a heuristic evaluation function to states in the search , effectively turning non-terminal nodes into terminal leaves!

- Alter *minimax* or *alpha-beta* in 2 ways:

1. Replace the utility function by a heuristic evaluation function EVAL, which estimates the position’s utility;
2. Replace the terminal test by a cutoff test that decides when to apply EVAL.

The heuristic minimax for state s and maximum depth d:

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if cutoff-test}(s,d) \\ \max_{a\in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if Player}(s)=\text{MAX} \\ \min_{a\in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if Player}(s)=\text{MIN} \end{cases}$$

Cutting off search

- The easiest way to cut off search is to set a fixed depth. This is very risky however, as you are not taking into account the status of the game.

- A slightly better way is to use an iterative deepening algorithm. You allow it to search until it has to stop due to time constraints. It returns the best move found so far. This will allow you to use all available time, but it still examines every node in the game tree at each level it explores

Quiescent search

- An even better way is to look at how quickly the evaluation function is changing and cut off your search when it has not changed much for a couple of moves, we call this a **QUIESCENT POSITION**.

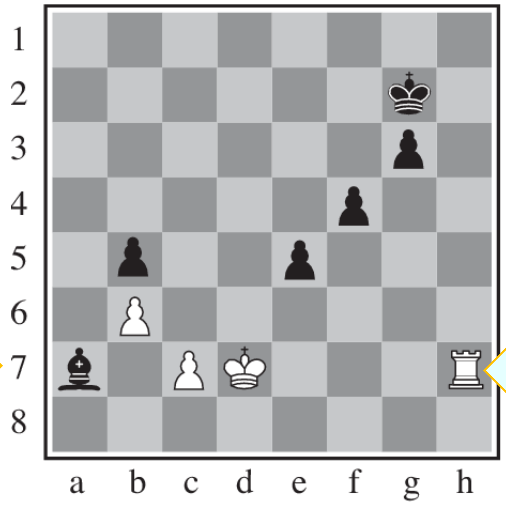
- The additional search used to find quiescent positions is called quiescent search.

Horizon Effect

- It arises when the game program faces an opponent move that causes serious damage and ultimately it is unavoidable.

Example: Consider this chess game. The black is to move.

There is no way for the black bishop to escape!!!!



The white rook can have a sequence of moves that can capture the black bishop"
1/ move to h1
2/ move to a1
3/ move to a2
4/ move to a7 and capture the black bishop!

BUT black has a sequence of moves that pushes the capture of the black bishop over the horizon::
1/ checks the white by moving the pawn from e5 to e6 → that pawn will be captured
2/ black checks again moving the pawn from f4 to f5 → that pawn will be captured
BLACK thinks that he has saved the bishop with the price of 2 pawns. It has pushed the capture of the bishop beyond the horizon.

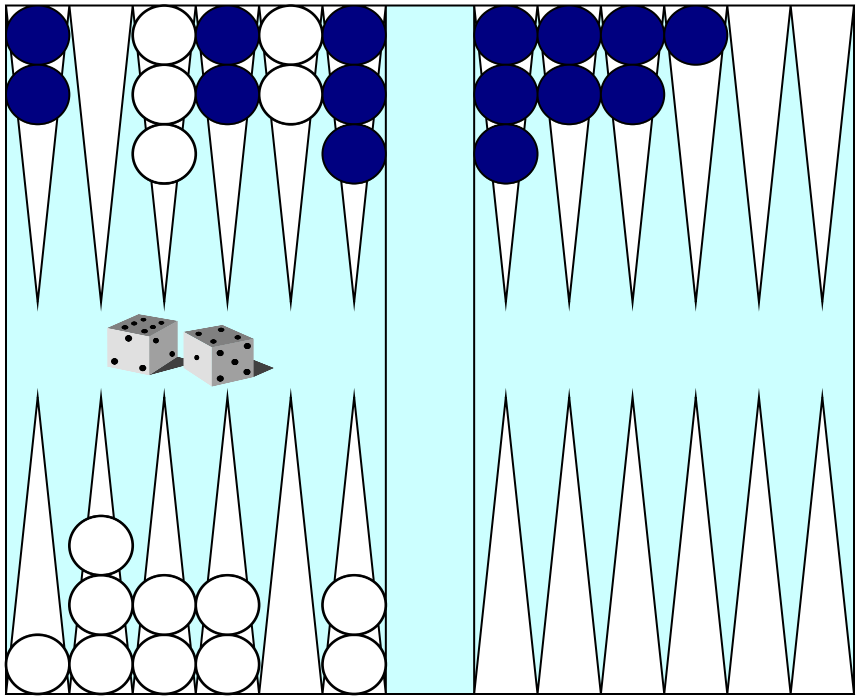
The horizon problem

 Difficult to eliminate
Arises when the program is facing a move by the opponent that causes damage and is unavoidable

Solution: Singular extensions = moves that are clearly better than other moves in a given position.

Games that include an element of chance

- Include a random element, e.g. throwing of dice



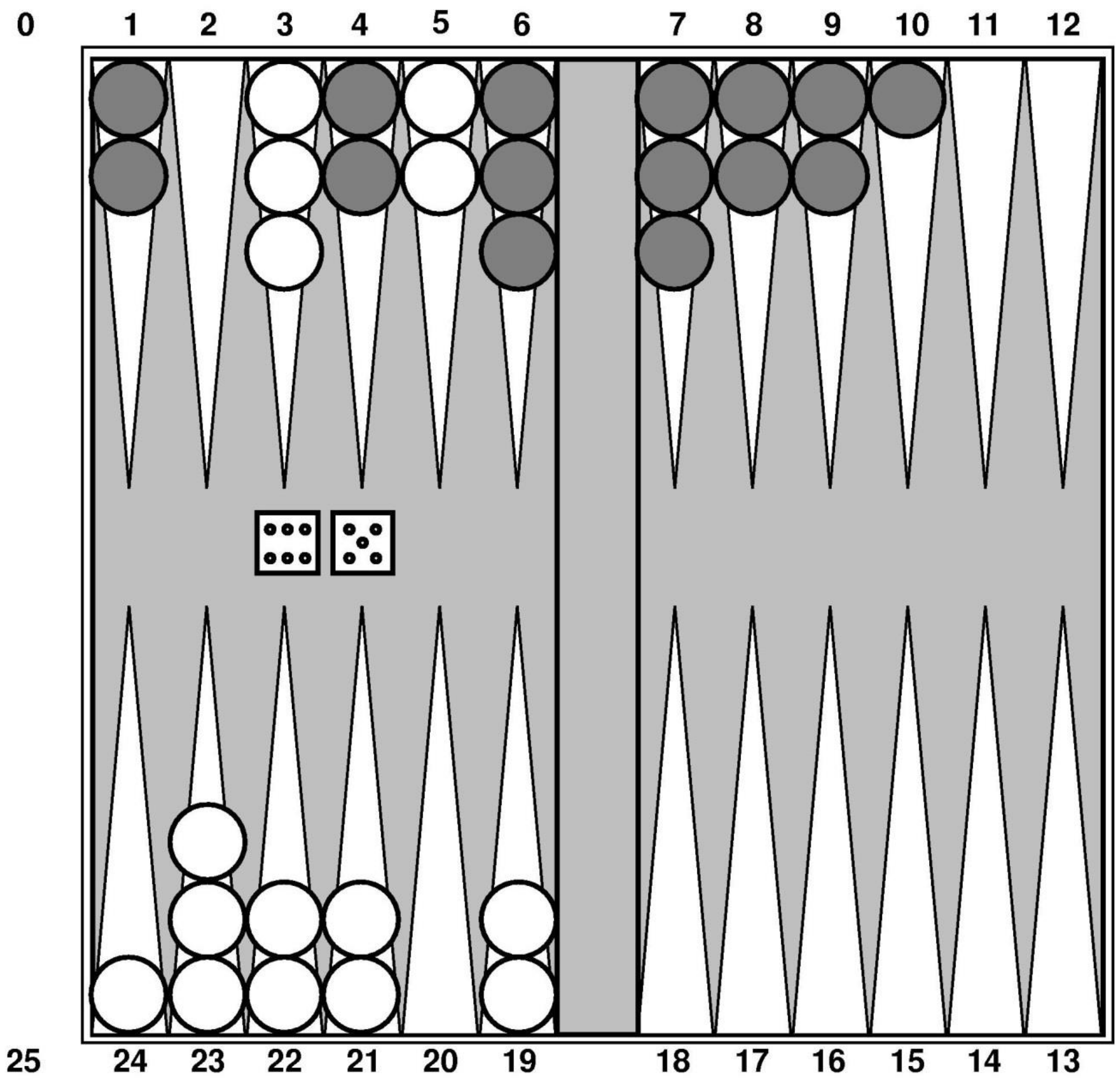
Backgammon is a typical game that combines luck and skill

A typical backgammon position.
The goal: move all of one's pieces off the board

WHITE moves clockwise toward 25, BLACK moves Counter-clockwise toward 0.

RULES: a piece can move to any position unless there are multiple opponent pieces there;
If there is an opponent, it is captured and it must Start over!!!

An example position

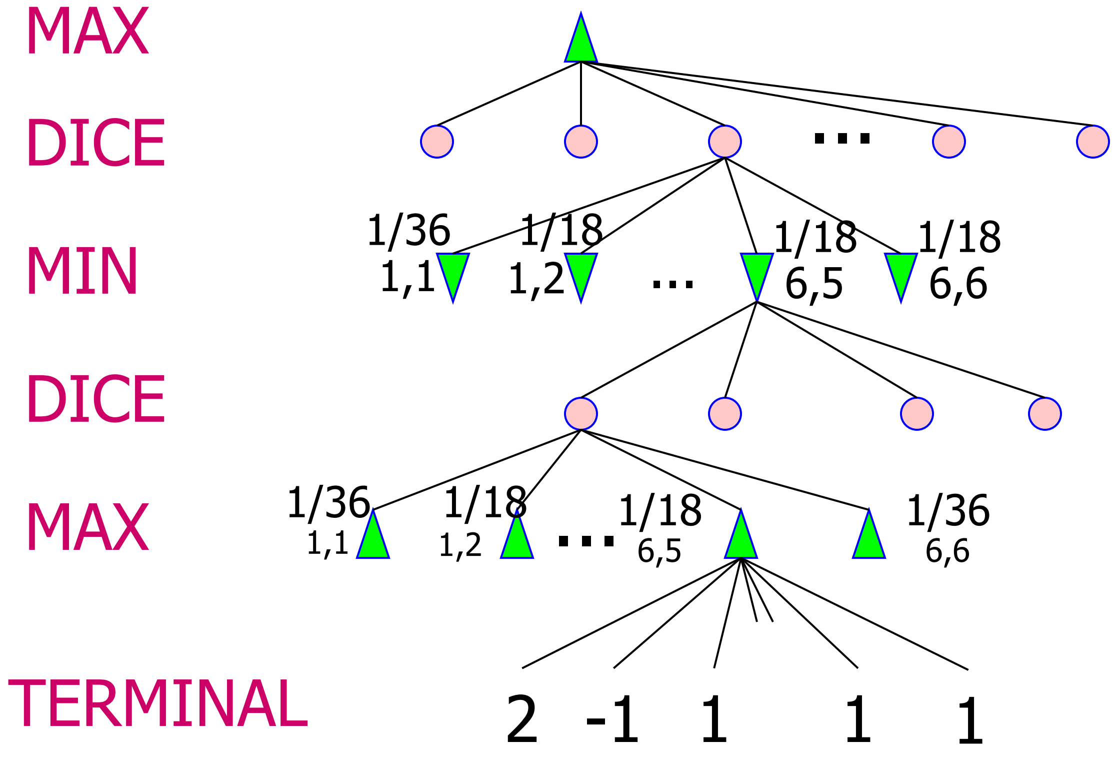


White has rolled 6-5

It must chose among 4 legal moves:
1/ (5-10, 5-11)
2/ (5-11,19-24)
3/ (5-10, 10-16)
4/ (5-11,11-16)

GAMES that include an element of chance

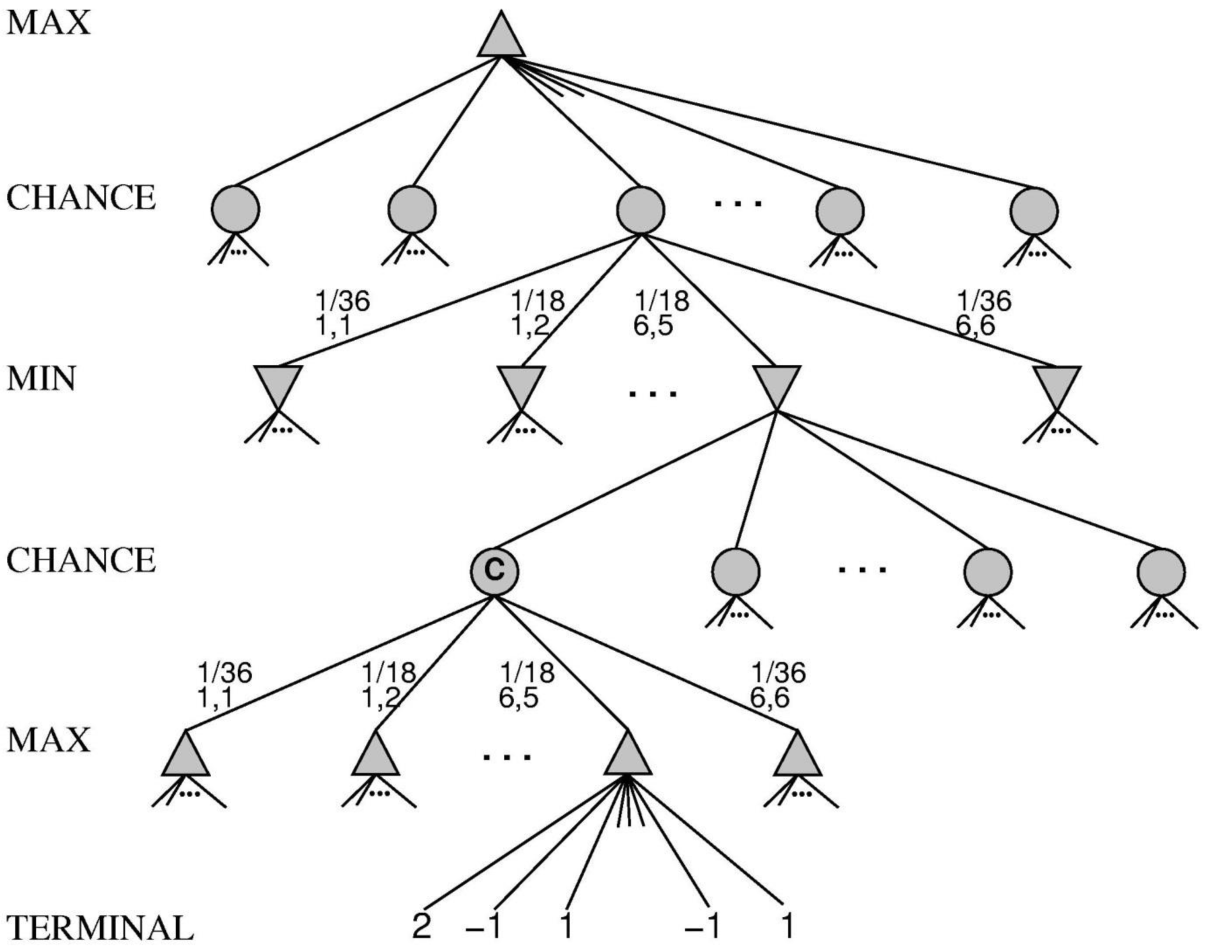
In backgammon, a game tree must include chance nodes in addition to MIN and MAX nodes



There are 36 ways to roll the dice
Each equally likely!!!

Because (2,3) is the same as (3,2)
There are only 21 distinct rolls

Schematic Tree for a backgammon position



There are 36 ways to
Roll twice, each equally
likely
BUT, because 5-6 is the
Same as 6-5 – there are
*21=36-6*5/2 distinct rolls*

The six doubles have a 1/36 chance
Of coming up, the other 21-6=15
Distinct rolls have a chance of 1/18

How correct decisions are made

- *Pick the move that leads to the best position*

Problem: The resulting positions do not have minimax values – but they have *expected values*!!!

  - The expectation is taken over all the possible dice rolls that could occur

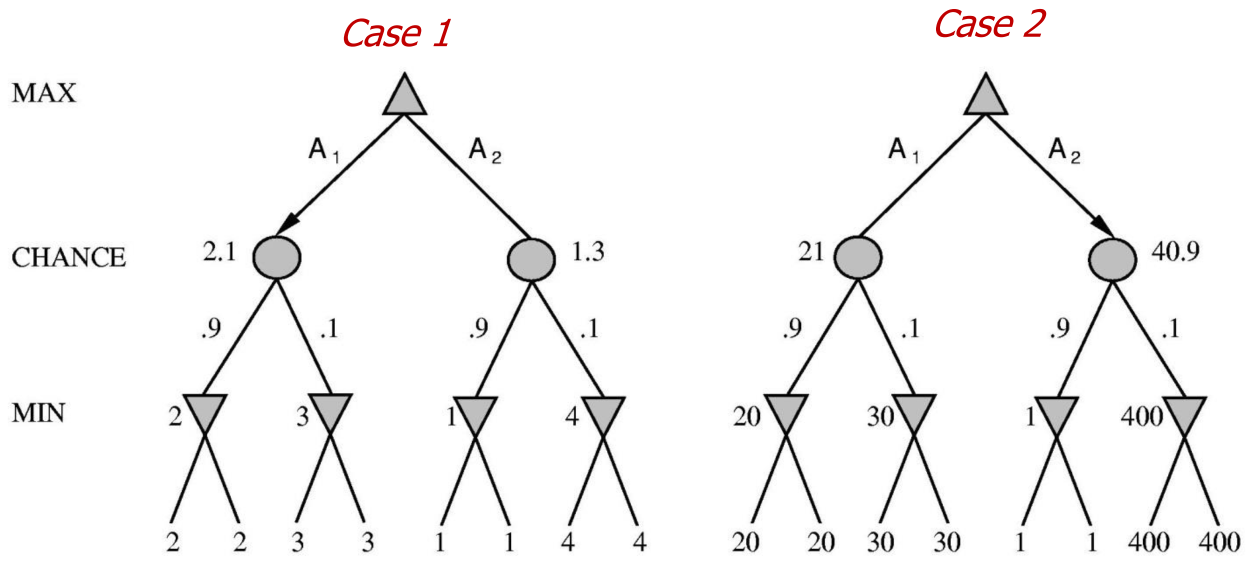
- *Generalized minimax value*

Expectimax value

- Terminal nodes and MAX and MIN nodes work exactly like for MINIMAX
- CHANCE nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls:

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{utility}(n), & \text{if } n \text{ is a terminal state} \\ \max_{s \in \text{Successors}(n)} \text{EXPECTIMINIMAX}(s), & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successors}(n)} \text{EXPECTIMINIMAX}(s), & \text{if } n \text{ is a MIN node} \\ \sum_{s \in \text{Successors}(n)} P(s) * \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a CHANCE node} \end{cases}$$

Position evaluation in games with chance nodes



2 cases: the presence of chance nodes means that one has to be more careful about what the ***evaluation values mean!!!***

Case 1: 2*0.9+3*0.1=2.1 **is better than** 0.9+0.4=1.3 → select A_1
Case 2: 20*0.9+30*0.1=21 **is worse than** 0.9+400*0.1=40.9 → select A_2

Complexity of expectimax

- If the program knew in advance all the dice rolls that would occur for the rest of the game, solving the game would be just like solving a game without dice.

- MINIMAX does it in $O(b^m)$ time
- Expectimax also considers all the possible dice-roll sequences, it takes $O(b^m n^m)$ where n is the number of distinct rolls.

Apply alpha-beta pruning

We have seen that we can prune
MIN and MAX nodes
Can we prune CHANCE nodes as well???

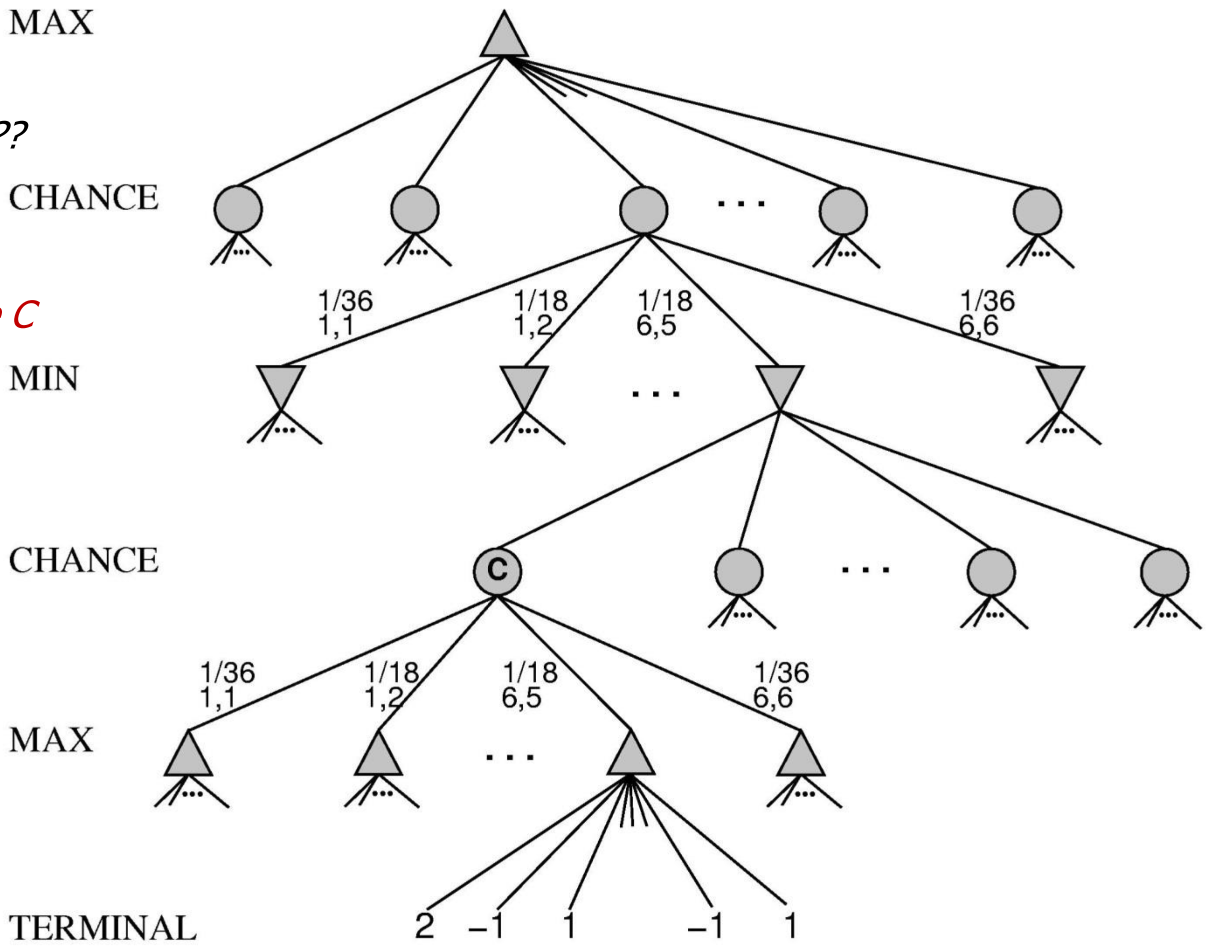
Use ingenuity:

- *take the example of the value of node C*
- *The value of C is the average of its children's values*

PUT bounds on the possible values of the utility functions – these will create bounds on the average as well

<u>Alternative:</u>

Use Monte Carlo simulation



Card Games

- Characterized by *stochastic partial observability – missing information is generated randomly*
- *<u>Examples:</u> bridge, hearts, poker*
- *At first sight – they resemble dice games – incorrect analogy!!!*
- *However – we could consider all the possible deals of the invisible cards – solve all of them and chose the move that has the best outcome averaged over all the deals.*

Chose a move based on possible deals

- Suppose that each deal *s* occurs with probability *P(s)*, then the move we want is given by:

$$\boxed{\operatorname*{arg\,max}_{a} \sum_{s} P(s) MINIMAX(RESULT(s,a))}$$

What if the # of deals is very large???

- For example – for playing bridge, there are 2 unseen hands of 13 cards each – the # of deals is 10,400,600!
- Use a Monte Carlo approximation:
  - Instead of adding up all the deals, we take a random sample of *N* deals, where the probability of a deal *s* appearing in a sample is proportional to:

$$\boxed{\operatorname*{arg\,max}_a \frac{1}{N}\sum_{i=1}^{N} MINIMAX(RESULT(s_i, a))}$$

Games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

- Othello: human champions refuse to compete against computers, who are too good.

- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.