بسم الله الرحمن الرحيم

BISMILLAH ARRAHMAN ARRAHEEM

Artificial Intelligence (CS-401)

Lecture 6: Games and Adversarial Search

Dr. Fahad Sherwani (Assistant Professor – FAST NUCES)

PhD in Artificial Intelligence

Universiti Tun Hussein Onn Malaysia

fahad.sherwani@nu.edu.pk

Can you plan ahead with these games

Typical simple case for a game

- **2-person** game
- Players alternate moves
- **Zero-sum**: one player’s loss is the other’s gain
- **Perfect information**: both players have access to complete information about the state of the game. No information is hidden from either player.
- **No chance** (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- But not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

Can we use …

- Uninformed serch?
- Heuristic Search?
- Local Search?
- Constraint based search?

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make
  - Compute new position resulting from each move
  - Evaluate each to determine which is best
  - Make that move
  - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board" (i.e., game state)
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position
  - Contrast with heuristic search where evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- Zero-sum assumption lets us use a single evaluation function to describe goodness of a board wrt both players
  - **f(n) >> 0**: position n good for me and bad for you
  - **f(n) << 0**: position n bad for me and good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

- Example of an evaluation function for Tic-Tac-Toe

  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]

  where a 3-length is a complete row, column, or diagonal

- Alan Turing's function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's

- Most evaluation functions specified as a weighted sum of position features

  $f(n) = w_1 * feat_1(n) + w_2 * feat_2(n) + ... + w_n * feat_k(n)$

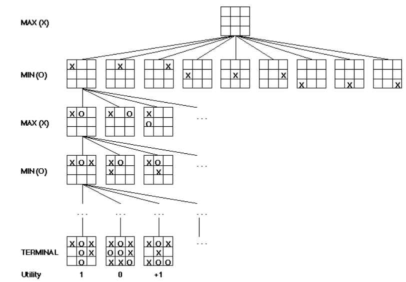
- Example features for chess are piece count, piece placement, squares controlled, etc.

- Deep Blue had >8K features in its evaluation function

That’s not how people play

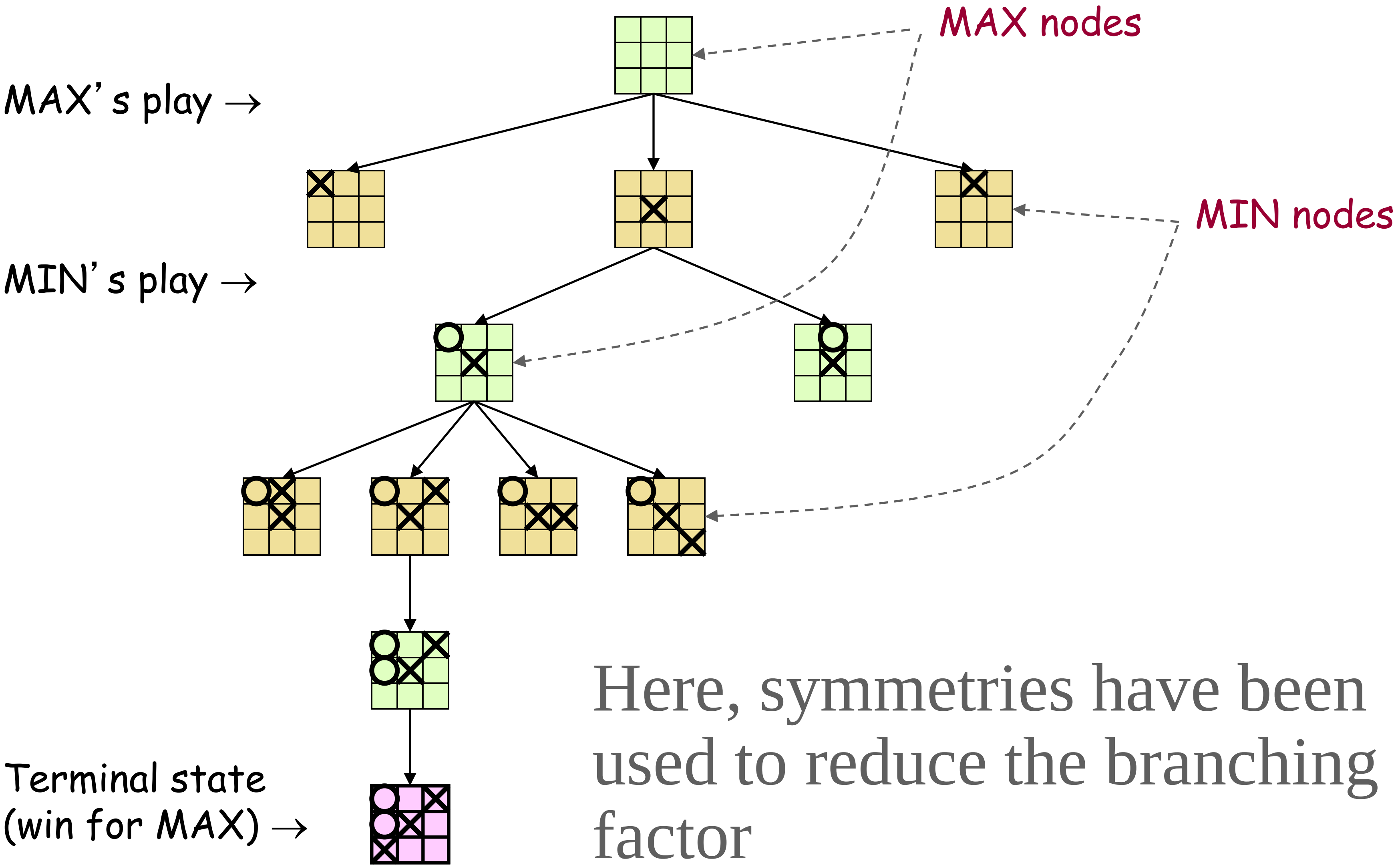
- People use “look ahead”
- i.e. enumerate actions, consider opponent’s possible responses, REPEAT
- Producing a complete **game tree** is only possible for simple games
- So, generate a partial game tree for some number of plys
  - Move = each player takes a turn
  - Ply = one player’s turn
- What do we do with the game tree?

Game trees



- Problem spaces for typical games are trees
- Root node represents the current board configuration; player must decide the best single move to make next
- **Static evaluator function** rates a board position **f(board)** a real, >0 for me <0 for opponent
- Arcs represent the possible legal moves for a player
- If it is **my turn** to move, then the root is labeled a "**MAX**" node; otherwise it is labeled a "**MIN**" node, indicating **my opponent's turn**.
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1

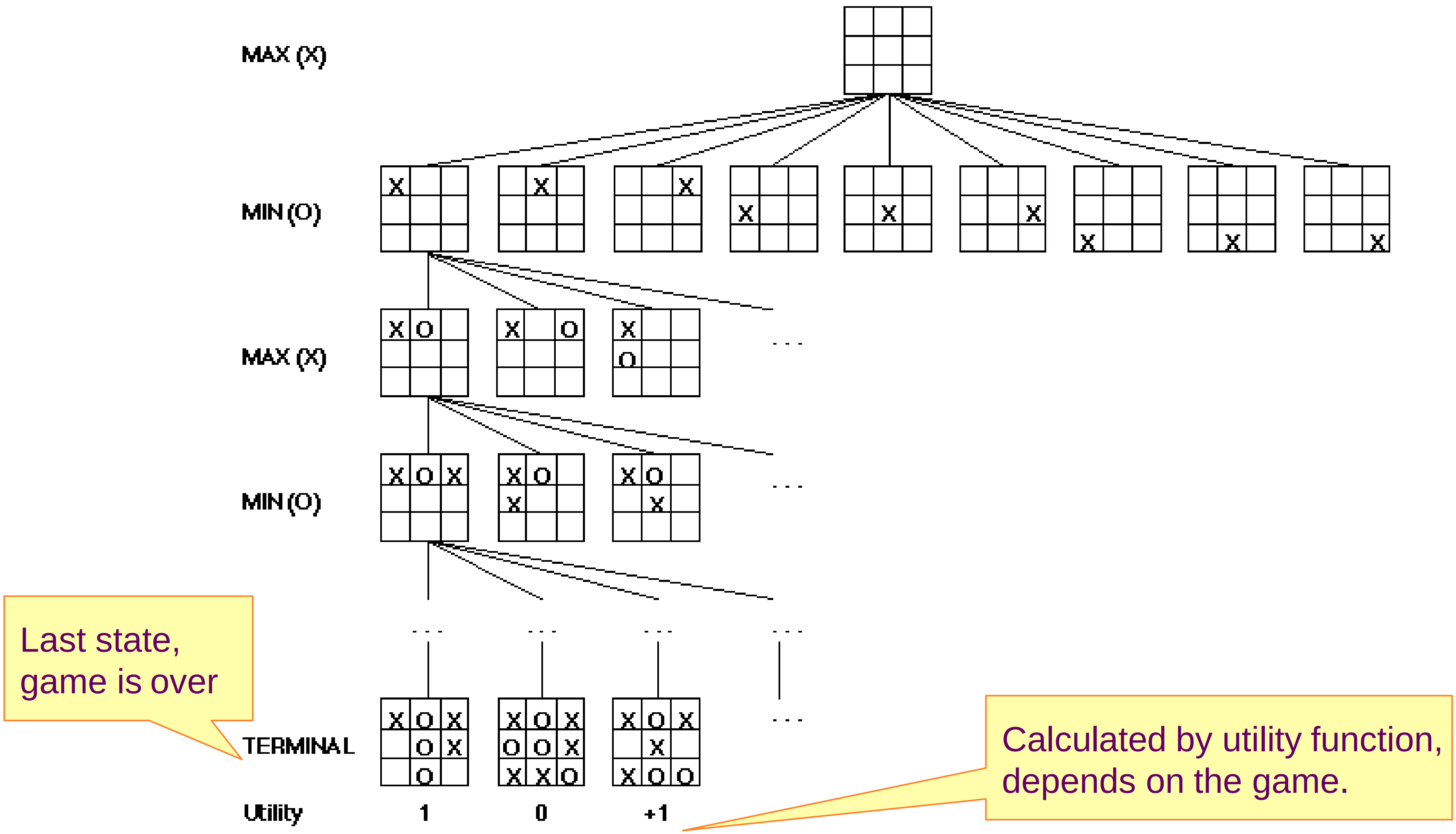
Game Tree for Tic-Tac-Toe



Here, symmetries have been used to reduce the branching factor

Game Tree (2-player, deterministic, turns)

How to see the game as a tree



Two-Person Perfect Information Deterministic Game

- Two players take turns making moves
- Board state fully known, deterministic evaluation of moves
- One player wins by defeating the other (or else there is a tie)
- Want a strategy to win, assuming the other person plays as well as possible

Computer Games

Playing games can be seen as a Search Problem

Multiplayer games as multi-agent environments.

Agents' goals are in conflict.

Mostly deterministic and fully observable environments.

Some games are not trivial search problems, thus needs AI techniques, e.g. Chess has an average branching factor of 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} *or* 10^{154} nodes.

Finding optimal move: choosing a good move with time limits.

Heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.

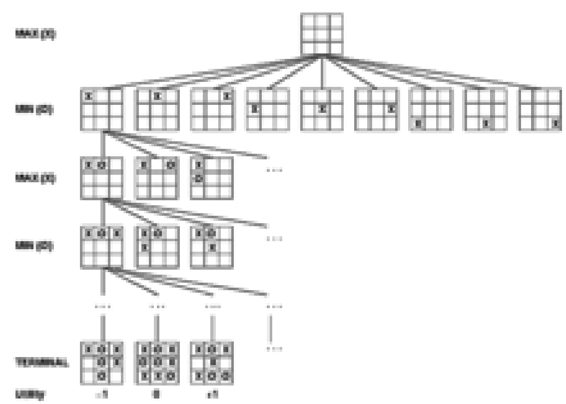
Minimax

Create a utility function

- Evaluation of board/game state to determine how strong the position of player 1 is.
- Player 1 wants to maximize the utility function
- Player 2 wants to minimize the utility function

Minimax Tree

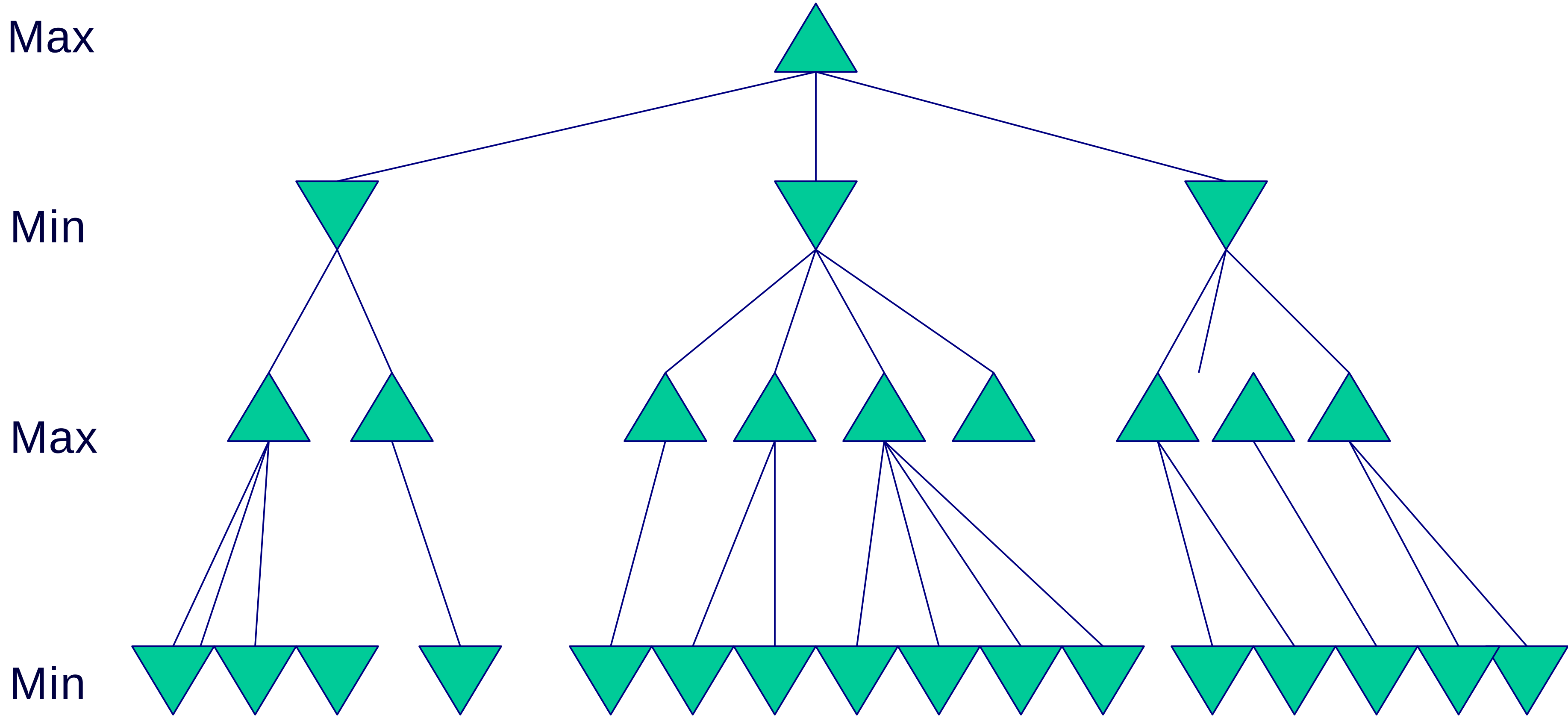
- Generate a new level for each move
- Levels alternate between “max” (player 1 moves) and “min” (player 2 moves)



Minimax Tree

You are Max and your enemy is Min.

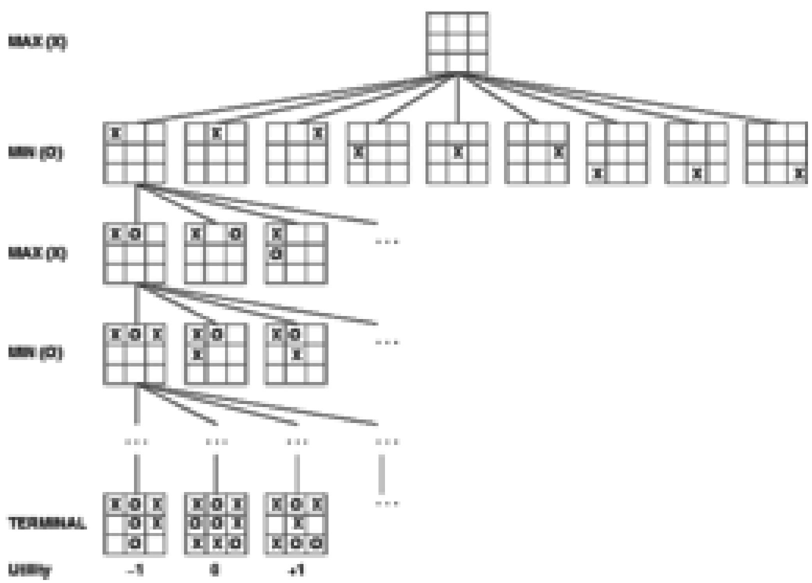
You play with your enemy in this way.



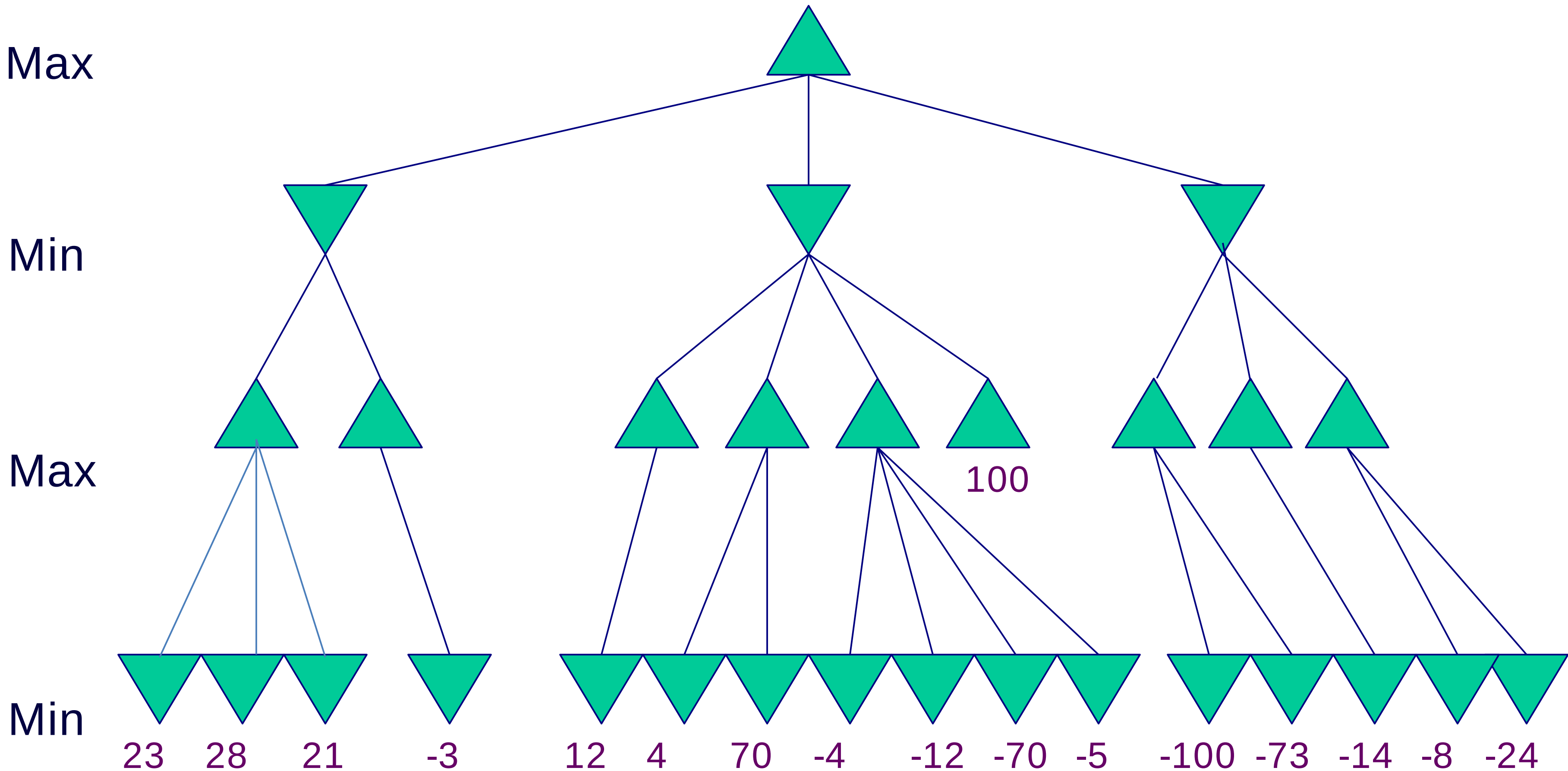
Minimax Tree Evaluation

Assign utility values to leaves

- Sometimes called “board evaluation function”
- If leaf is a “final” state, assign the maximum or minimum possible utility value (depending on who would win).
- If leaf is not a “final” state, must use some other heuristic, specific to the game, to evaluate how good/bad the state is at that point



Minimax Tree



Terminal nodes: values calculated from the utility function, evaluates how good/bad the state is at this point

Minimax procedure

- Create start node as a MAX node with current board configuration
- Expand nodes down to some **depth** (a.k.a. **ply**) of lookahead in the game
- Apply the evaluation function at each of the leaf nodes
- "Back up" values for each of the non-leaf nodes until a value is computed for the root node
  - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
  - At MAX nodes, the backed-up value is the **maximum** of the values associated with its children.
- Pick the operator associated with the child node whose backed-up value determined the value at the root

Minimax theorem

- Intuition: assume your opponent is at least as smart as you are and play accordingly. If he's not, you can only do better.
- Von Neumann, J: *Zur Theorie der Gesellschafts-spiele* Math. Annalen. **100** (1928) 295-320

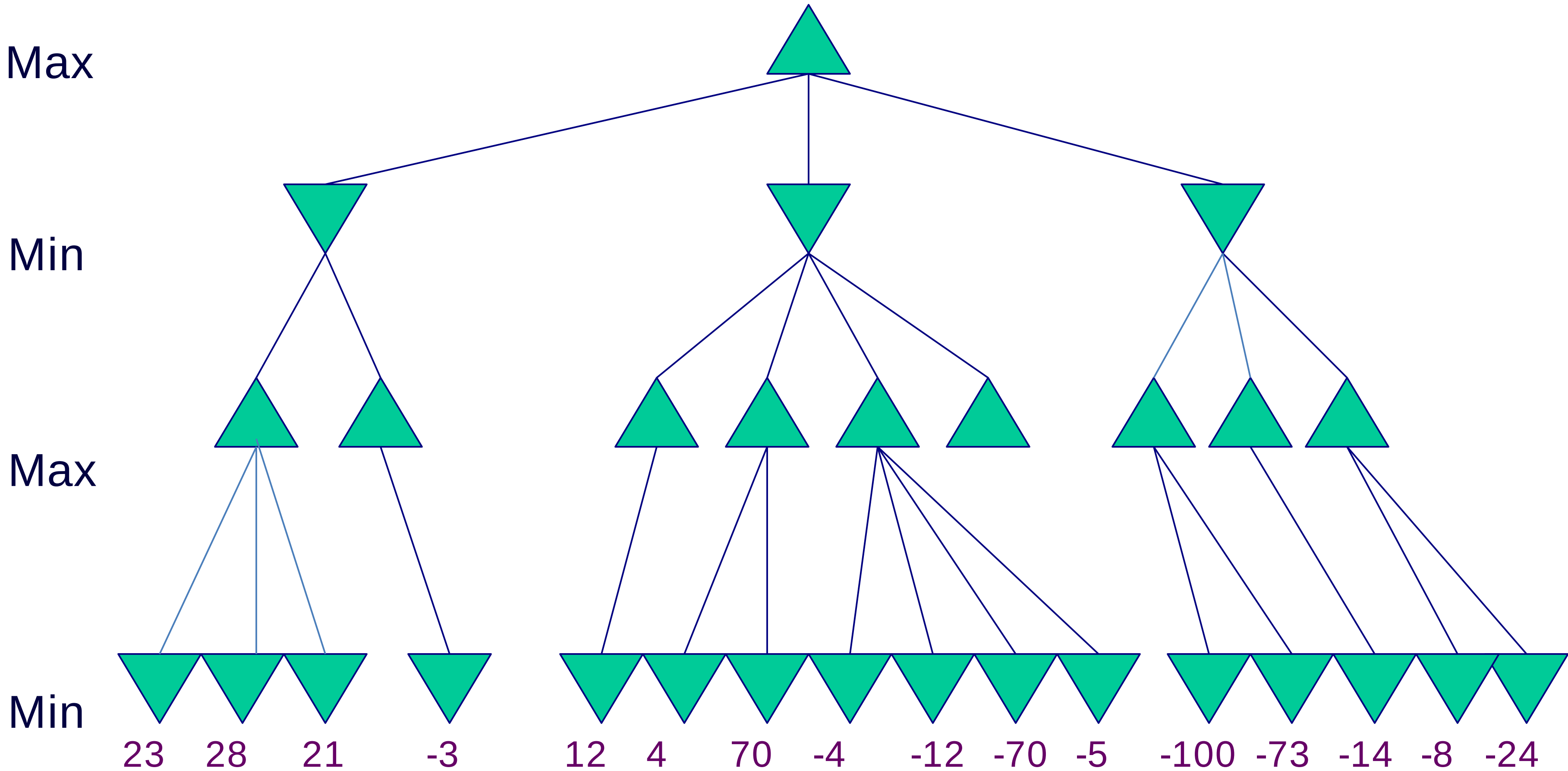
  For every two-person, zero-sum game with finite strategies, there exists a value V and a mixed strategy for each player, such that (a) given player 2's strategy, the best payoff possible for player 1 is V, and (b) given player 1's strategy, the best payoff possible for player 2 is –V.
- You can think of this as:
  - –Minimizing your maximum possible loss
  - –Maximizing your minimum possible gain

Minimax Tree Evaluation

For the MAX player

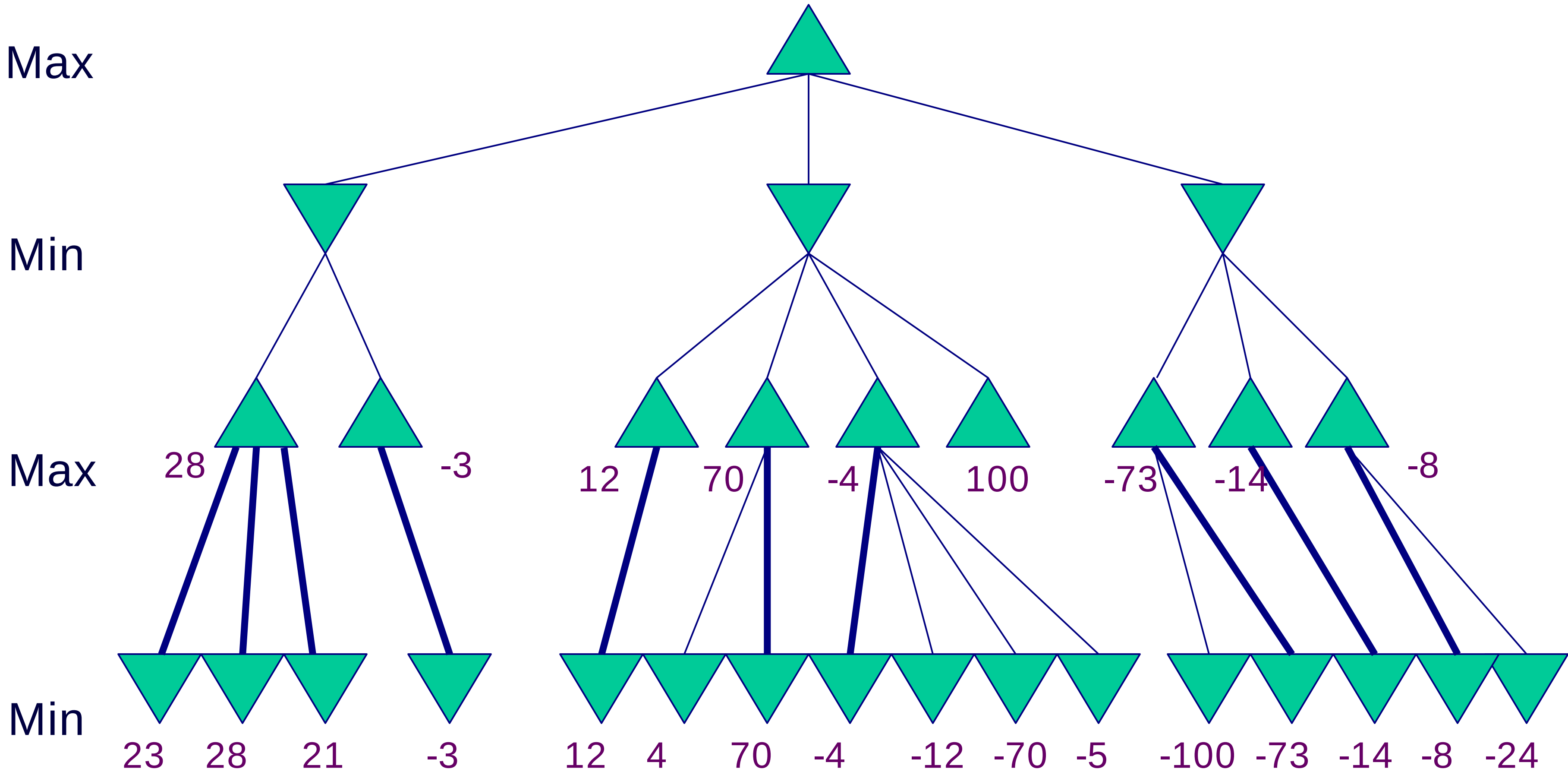
1. Generate the game as deep as time permits
2. Apply the evaluation function to the leaf states
3. Back-up values
   - At MIN assign minimum payoff move
   - At MAX assign maximum payoff move
4. At root, MAX chooses the operator that led to the highest payoff

Minimax Tree



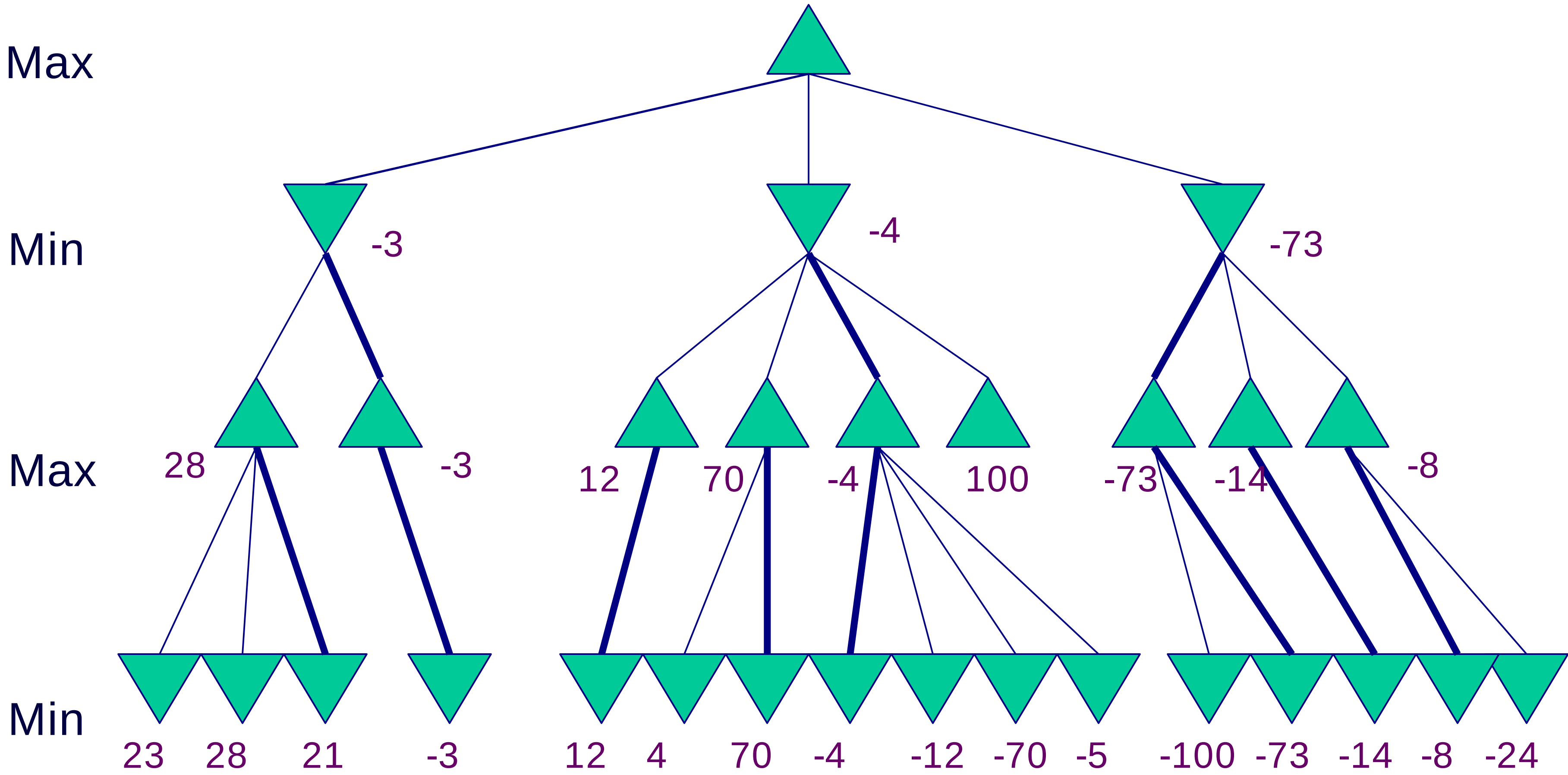
Terminal nodes: values calculated from the utility function

Minimax Tree

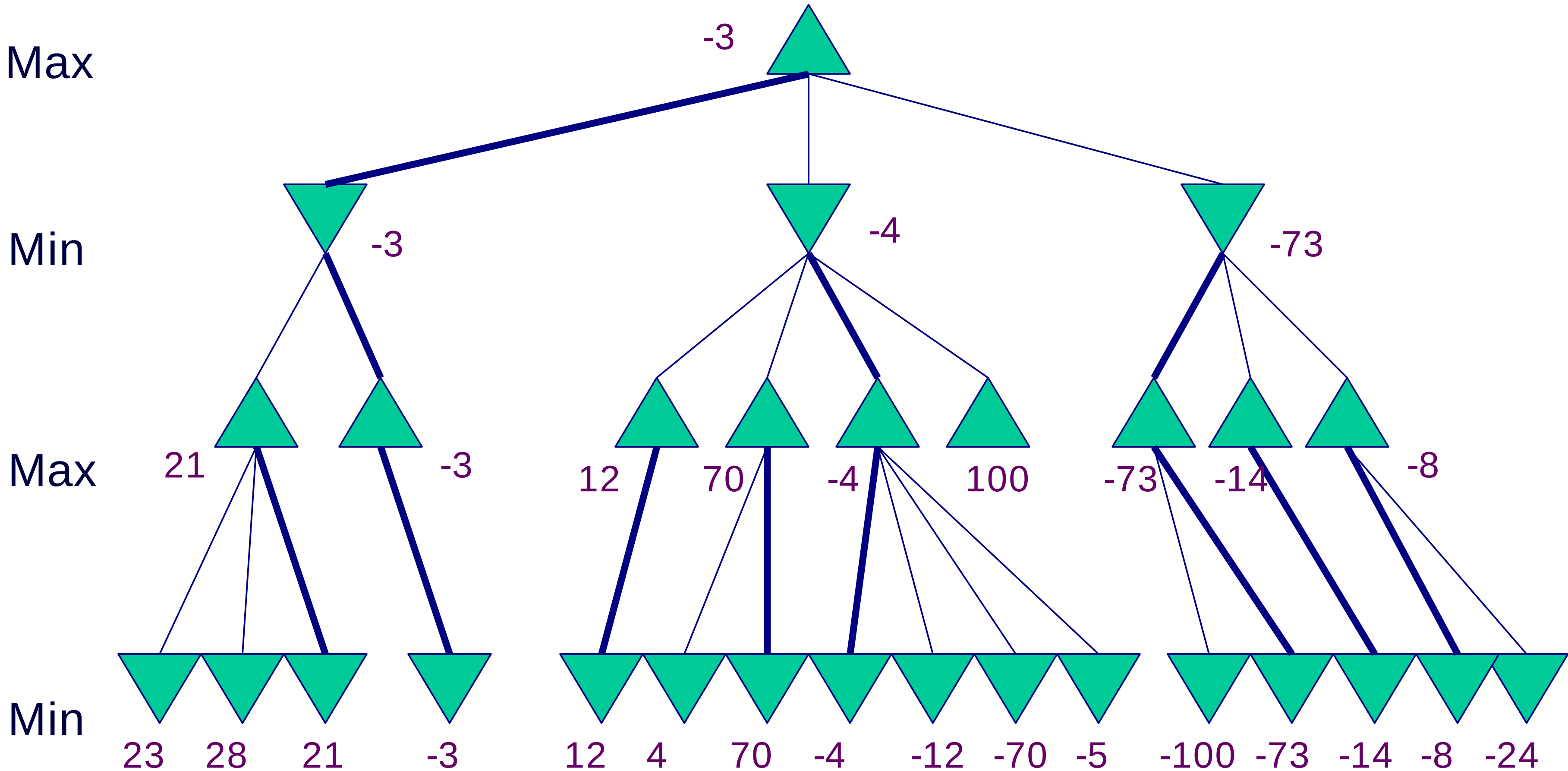


Other nodes: values calculated via minimax algorithm

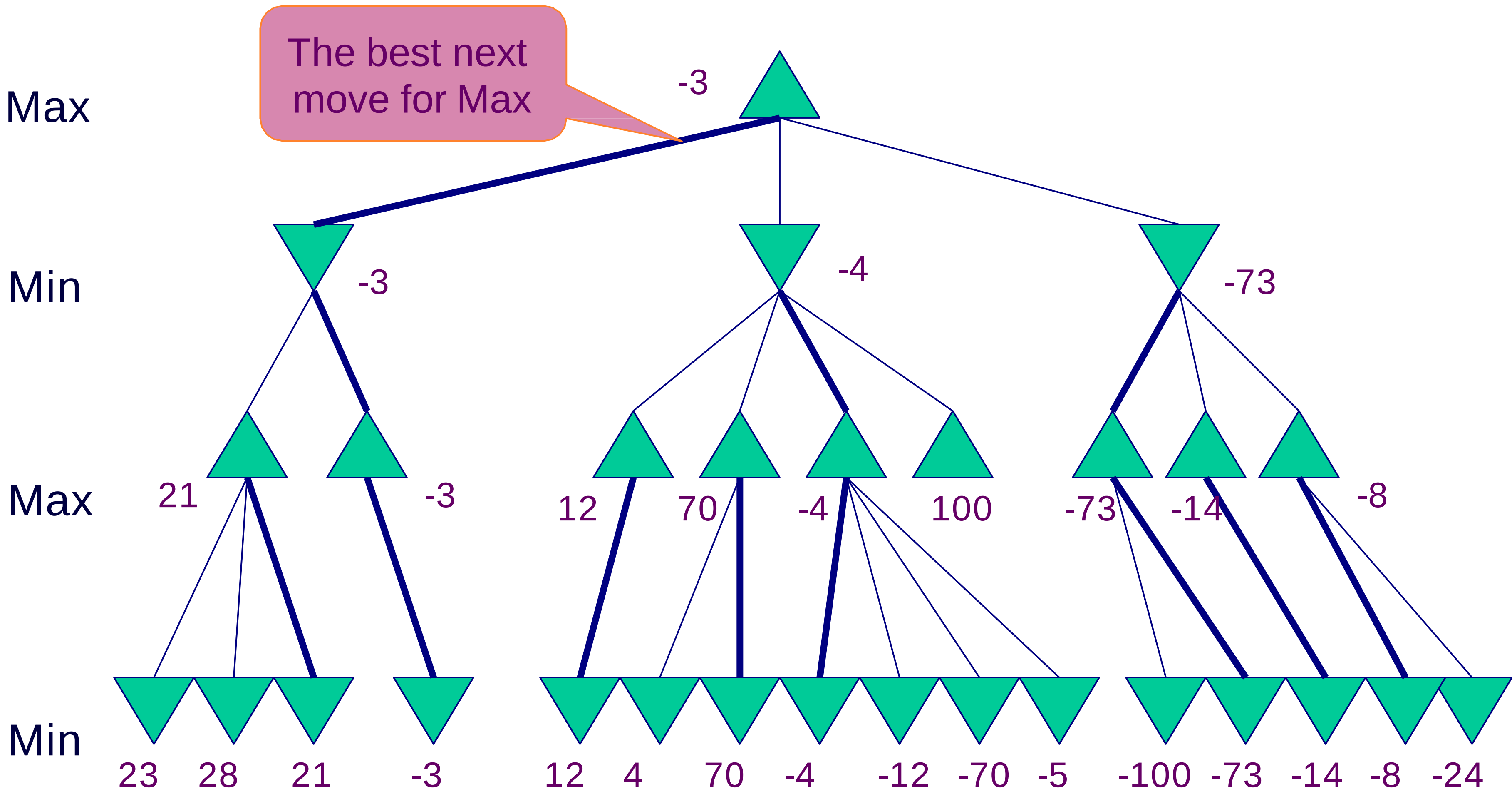
Minimax Tree



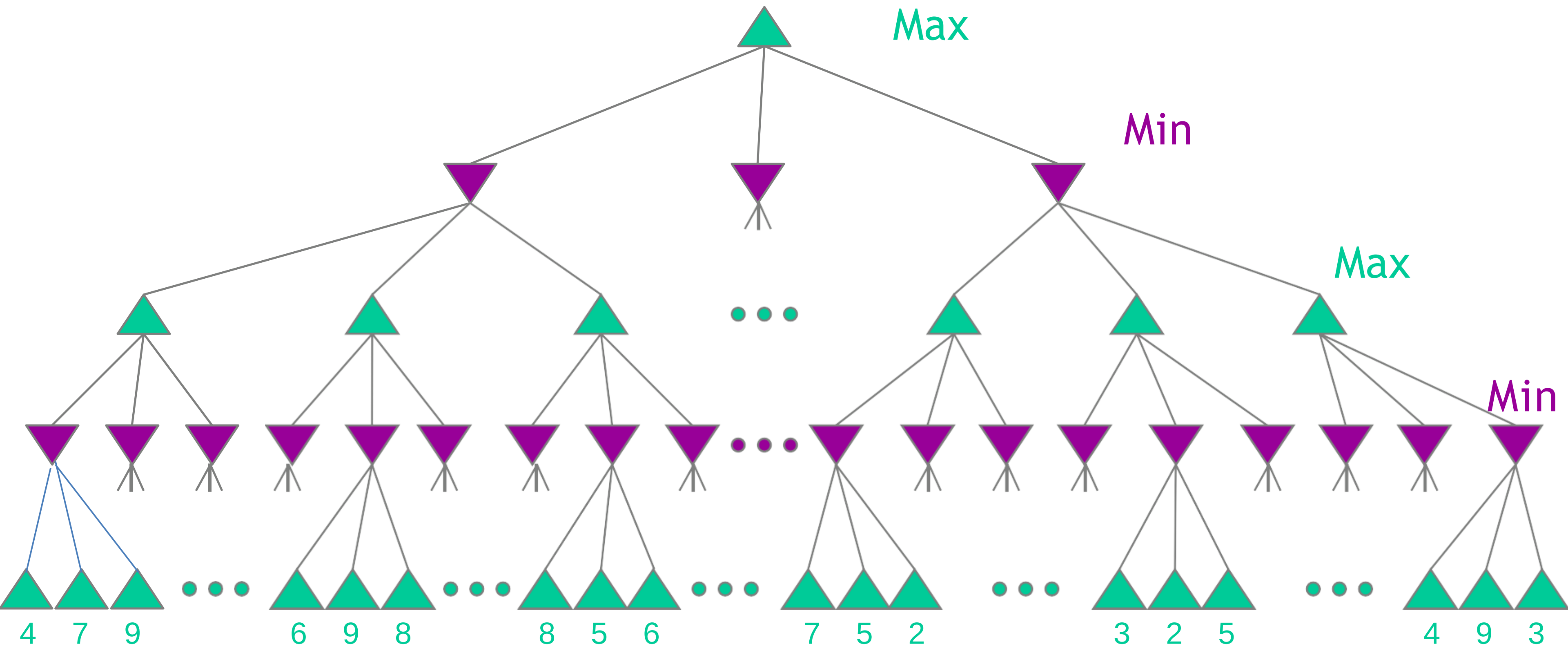
Minimax Tree



Minimax Tree

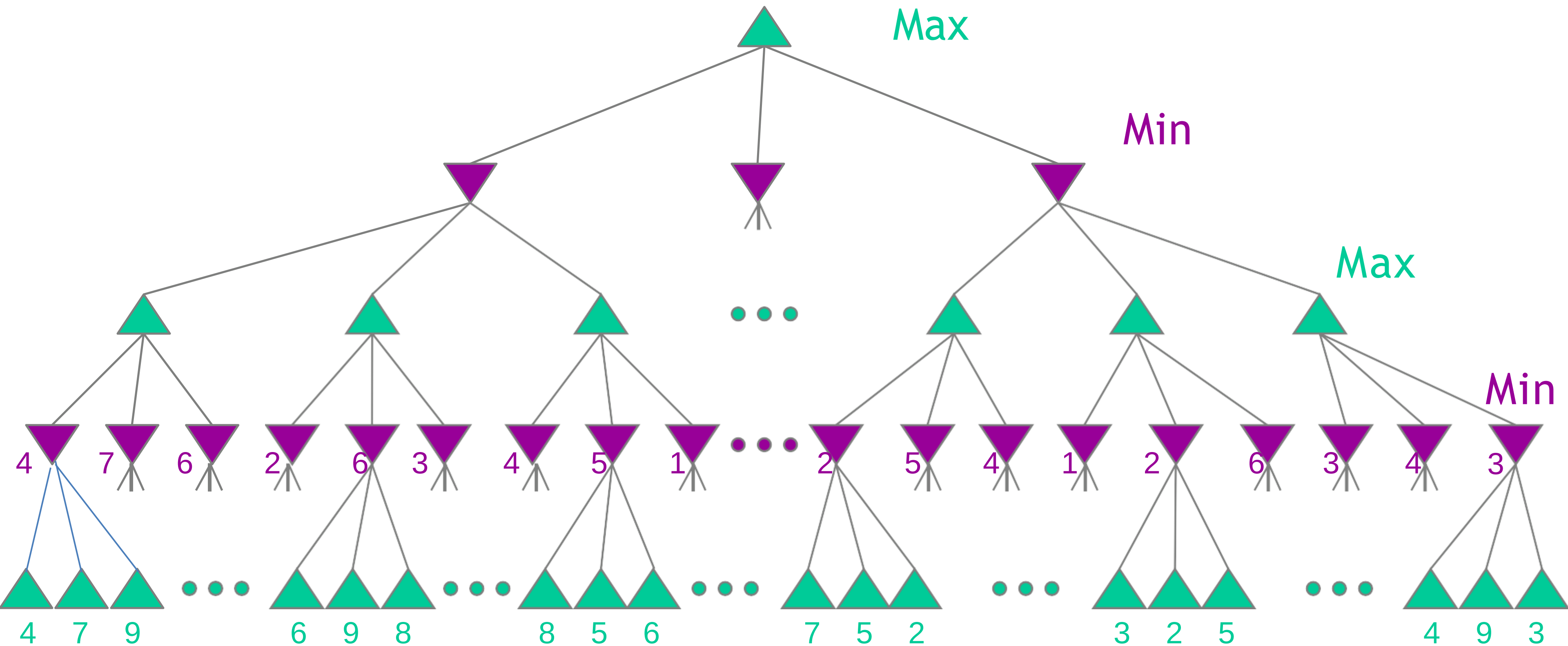


MiniMax Example-2



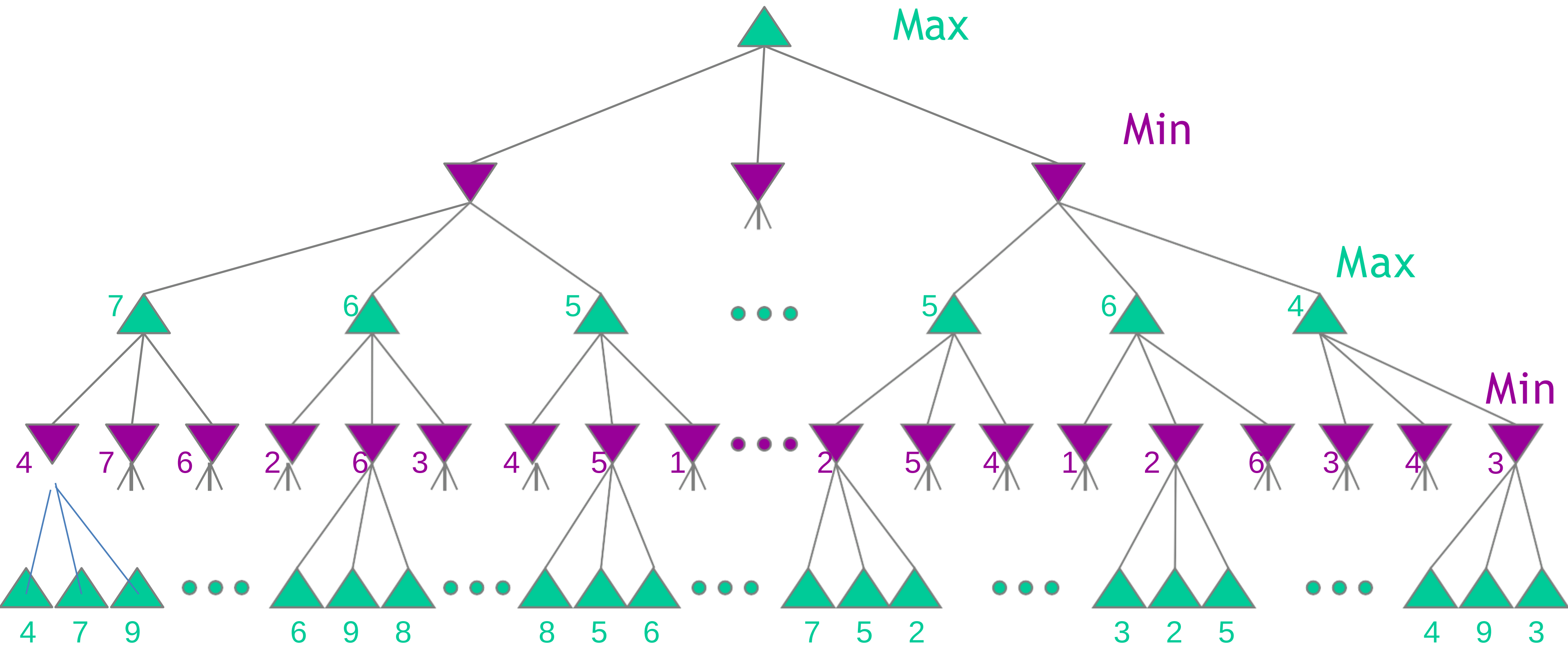
Terminal nodes: values calculated from the utility function

MiniMax Example-2

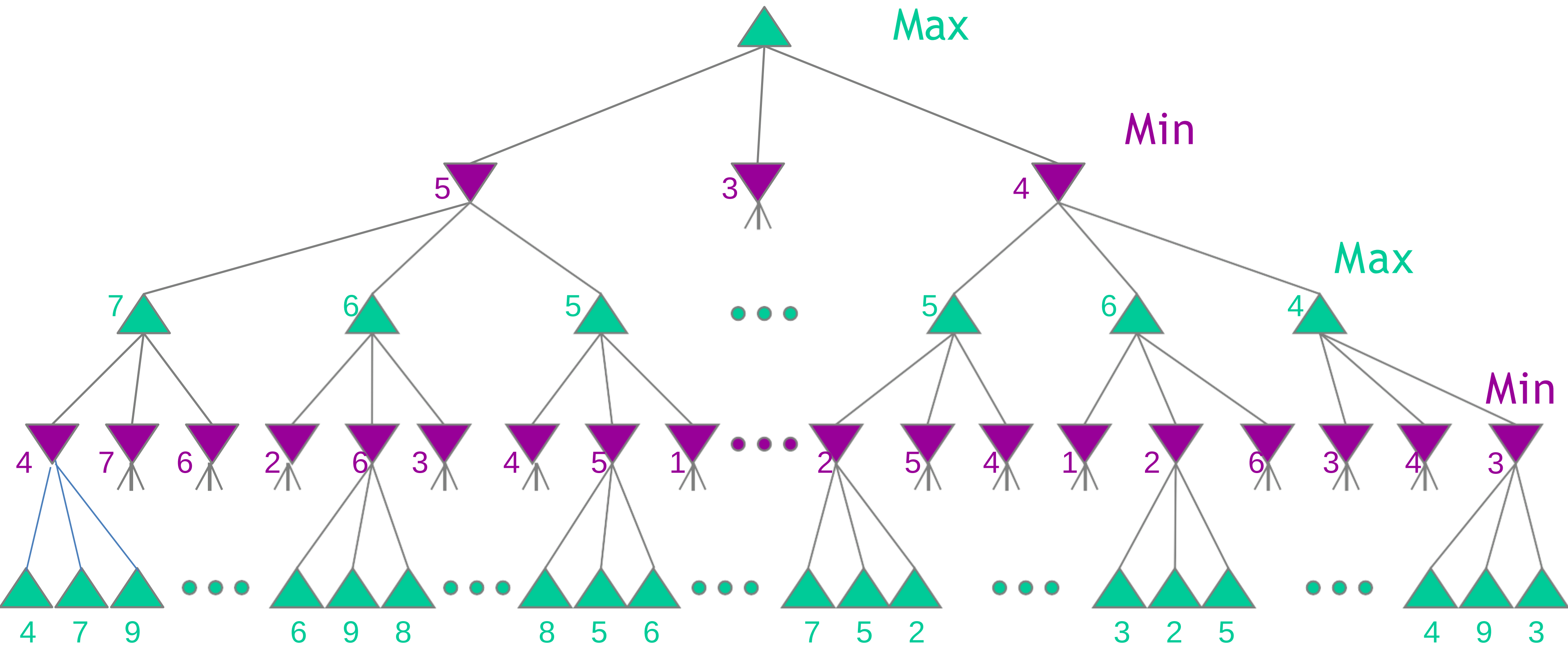


Other nodes: values calculated via minimax algorithm

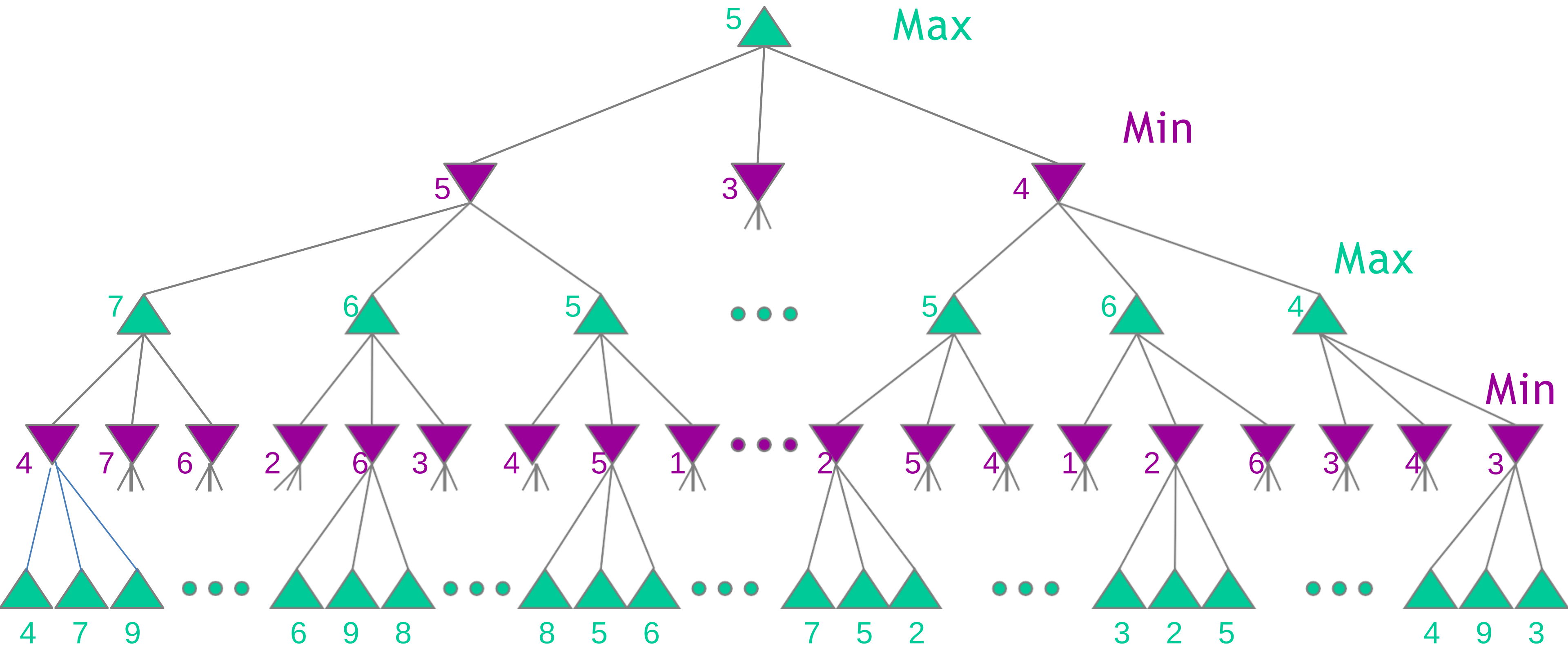
MiniMax Example-2



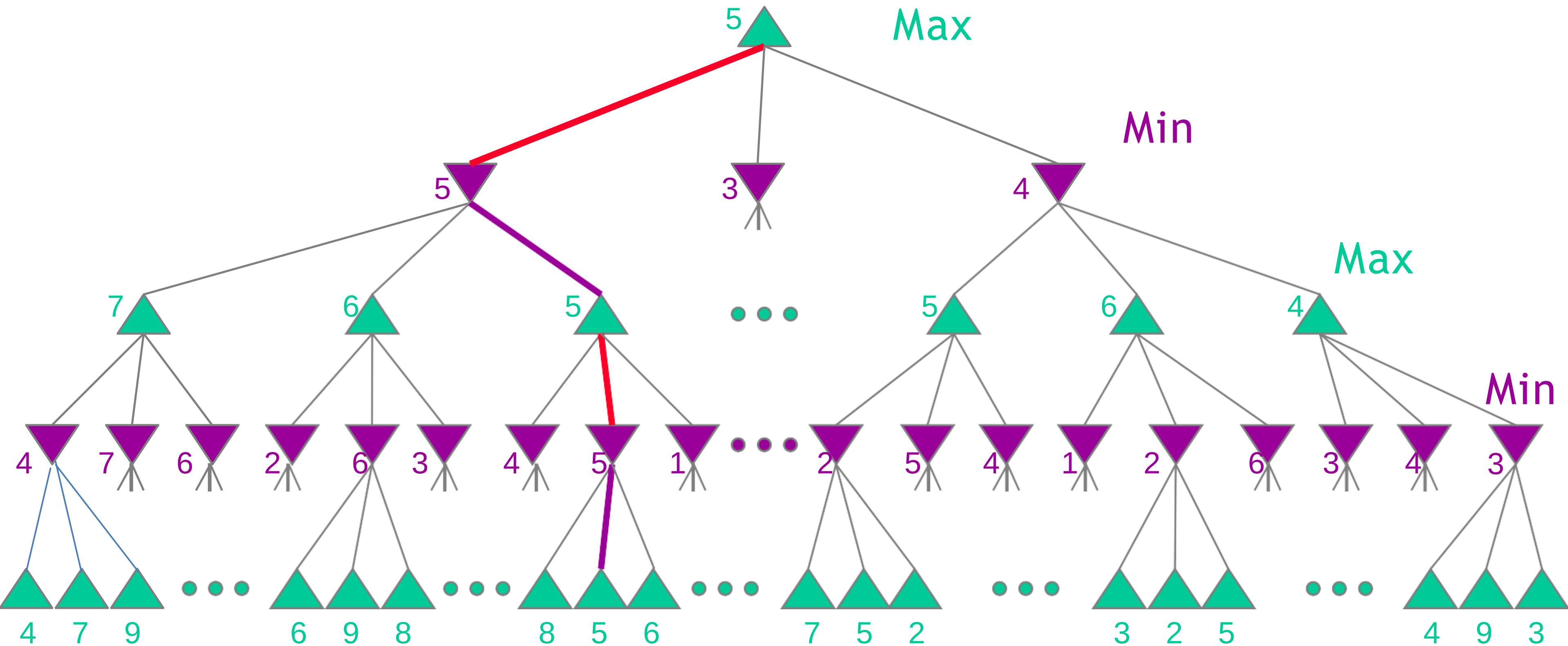
MiniMax Example-2



MiniMax Example-2



MiniMax Example-2



Moves by Max and countermoves by Min

Properties of MiniMax

Complete: Yes (if tree is finite)

Optimal: Yes (against an optimal opponent)

Time complexity: A complete evaluation takes time b^m

Space complexity: A complete evaluation takes space bm

(depth-first exploration)

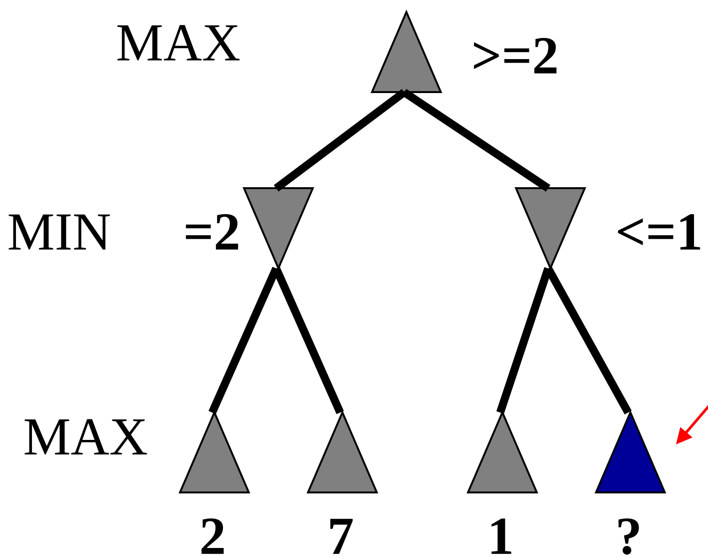
For chess, b ≈ 35, m ≈100 for "reasonable" games
→ exact solution completely infeasible, since it's too big

Instead, we limit the depth based on various factors, including time available.

Alpha-Beta Pruning Algorithm

Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through **alpha-beta pruning**
- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston



- We don't need to compute the value at this node.
- No matter what it is, it can't affect the value of the root node.

Pruning the Minimax Tree

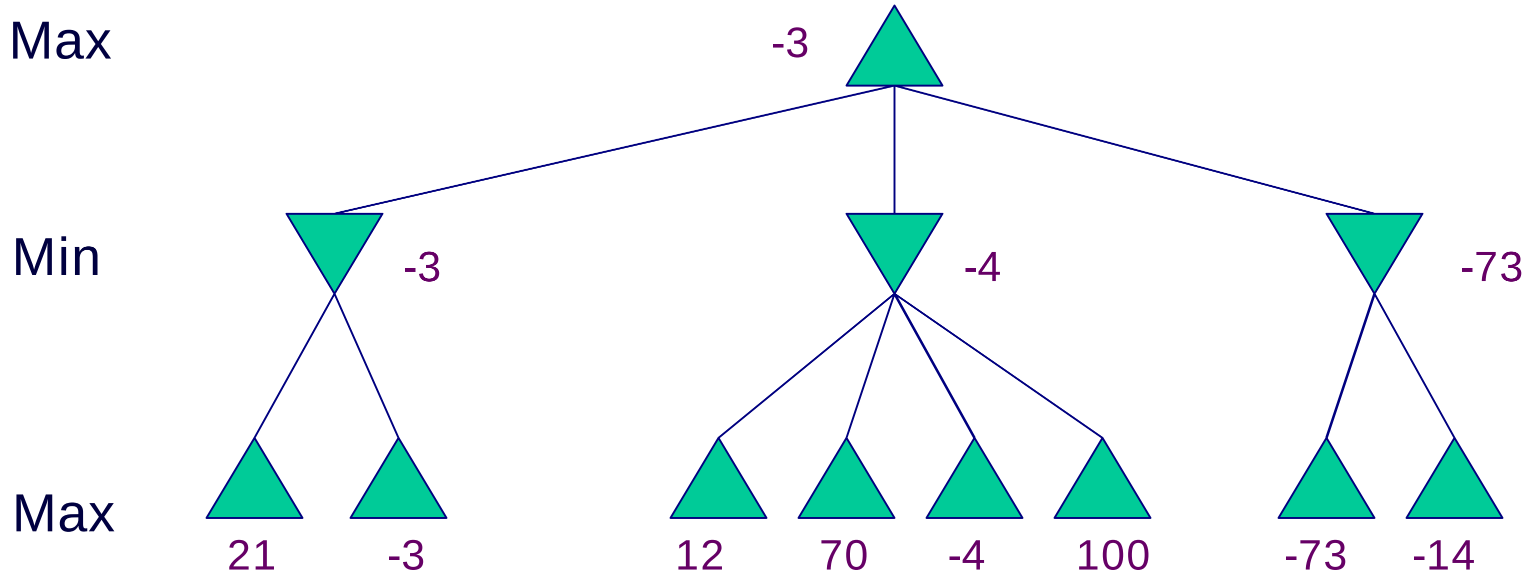
Since we have limited time available, we want to avoid unnecessary computation in the minimax tree.

Pruning: ways of determining that certain branches will not be useful.

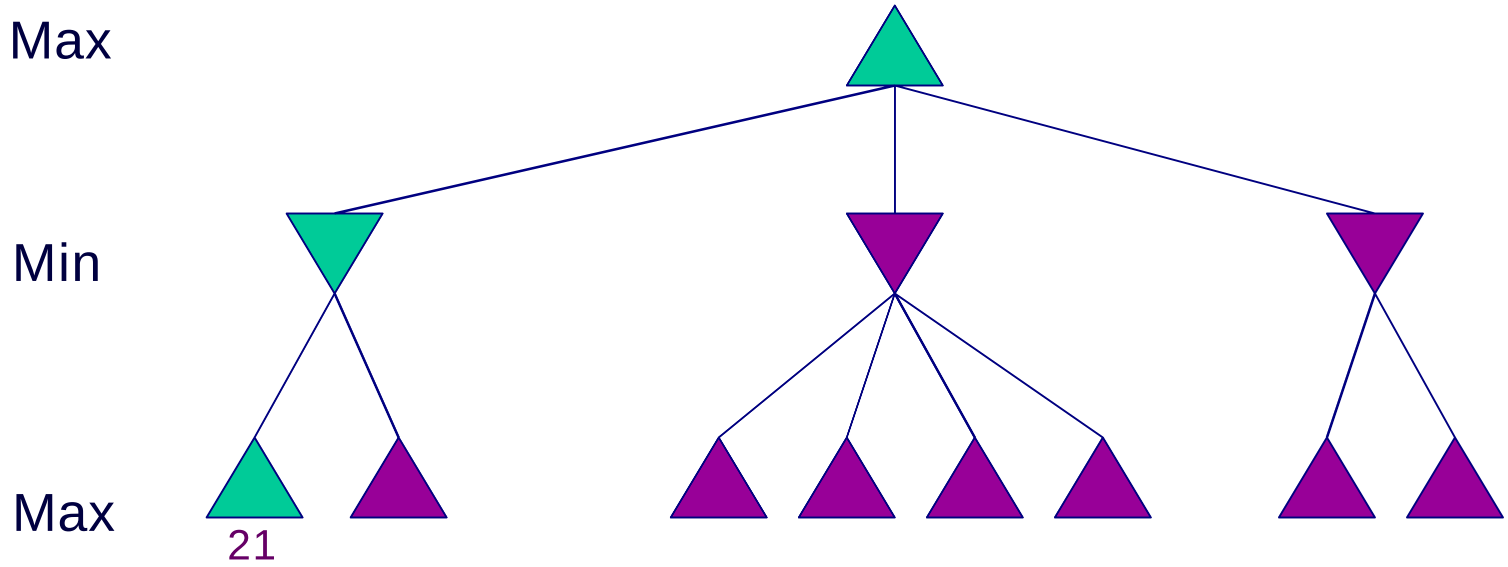
α Cuts

If the current max value is greater than the successors min value, don't explore that min subtree any more.

α Cut Example

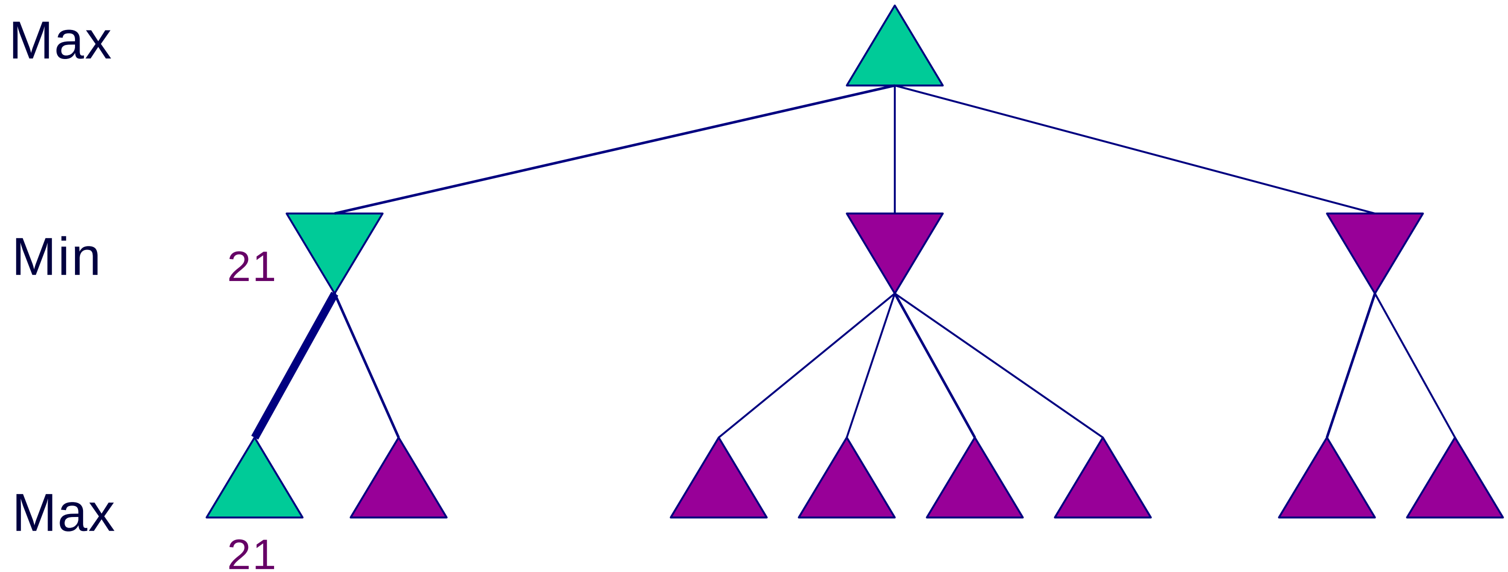


α Cut Example



Depth first search along path 1

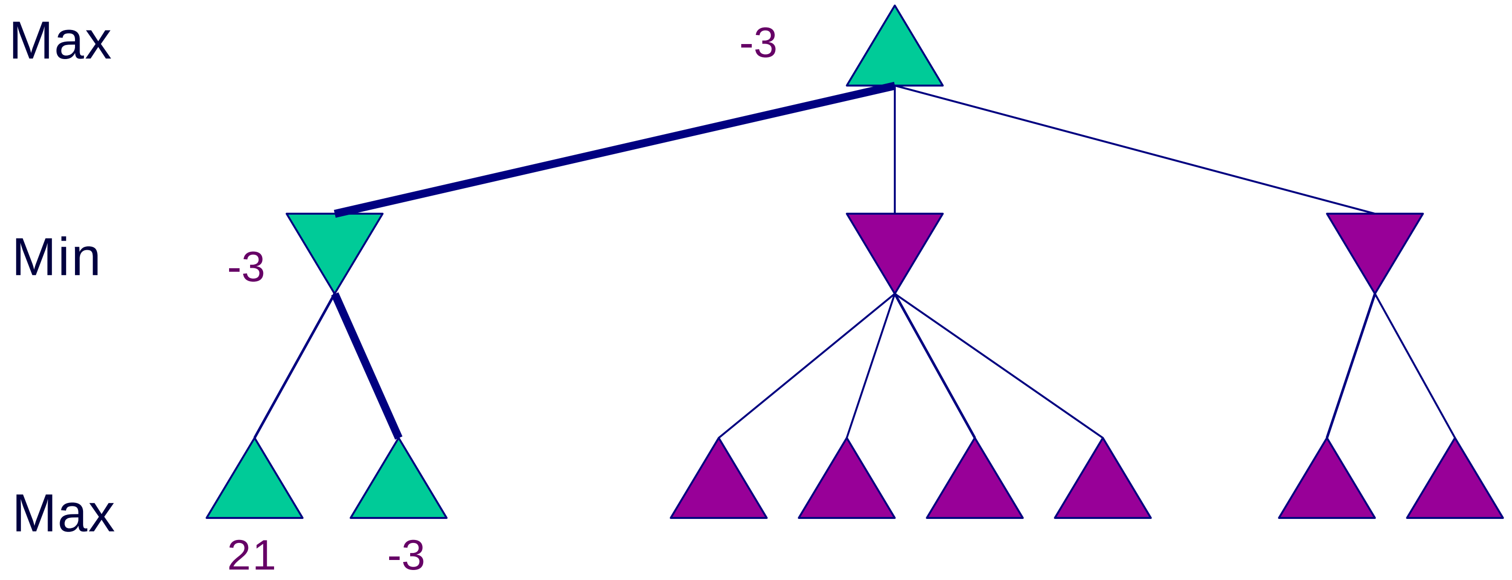
α Cut Example



21 is minimum so far (second level)

Can't evaluate yet at top level

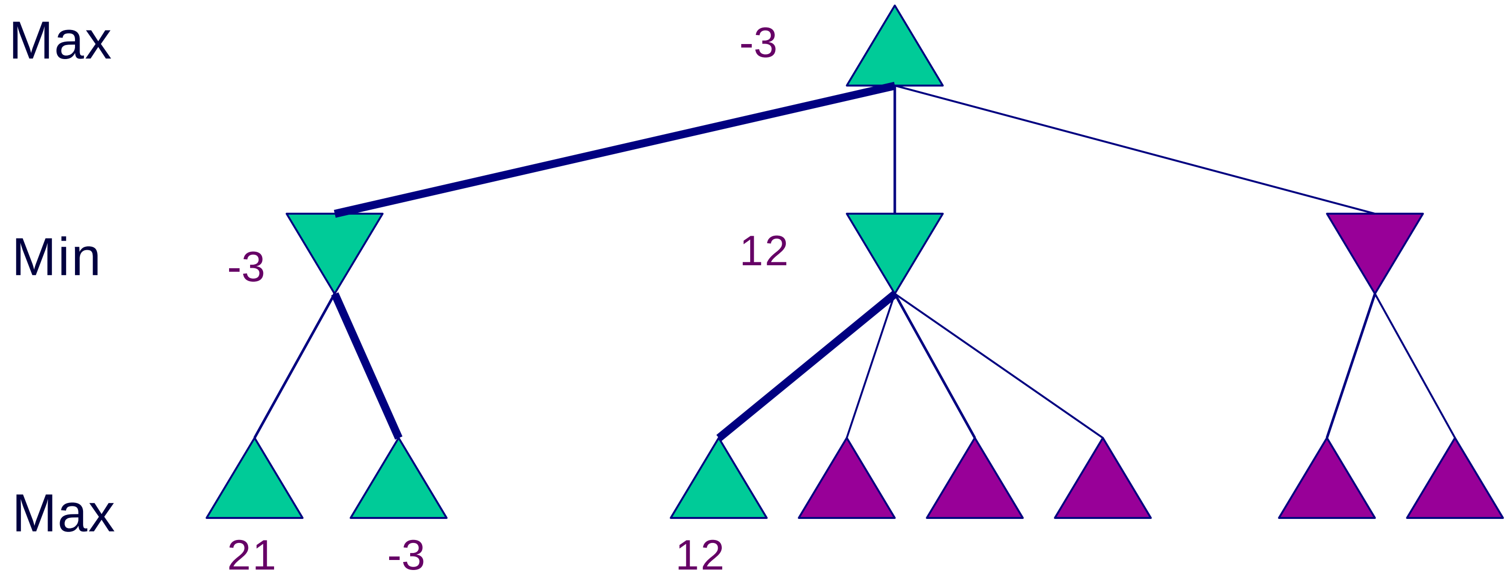
α Cut Example



-3 is minimum so far (second level)

-3 is maximum so far (top level)

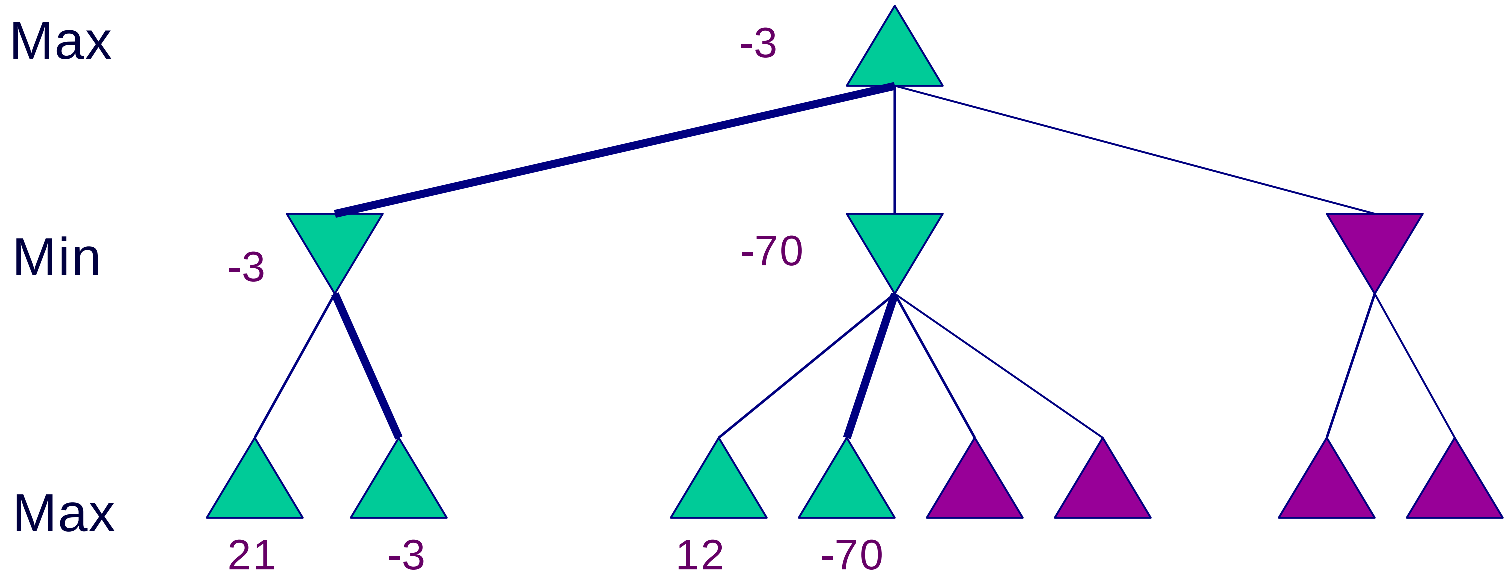
α Cut Example



12 is minimum so far (second level)

-3 is still maximum (can’t use second node yet)

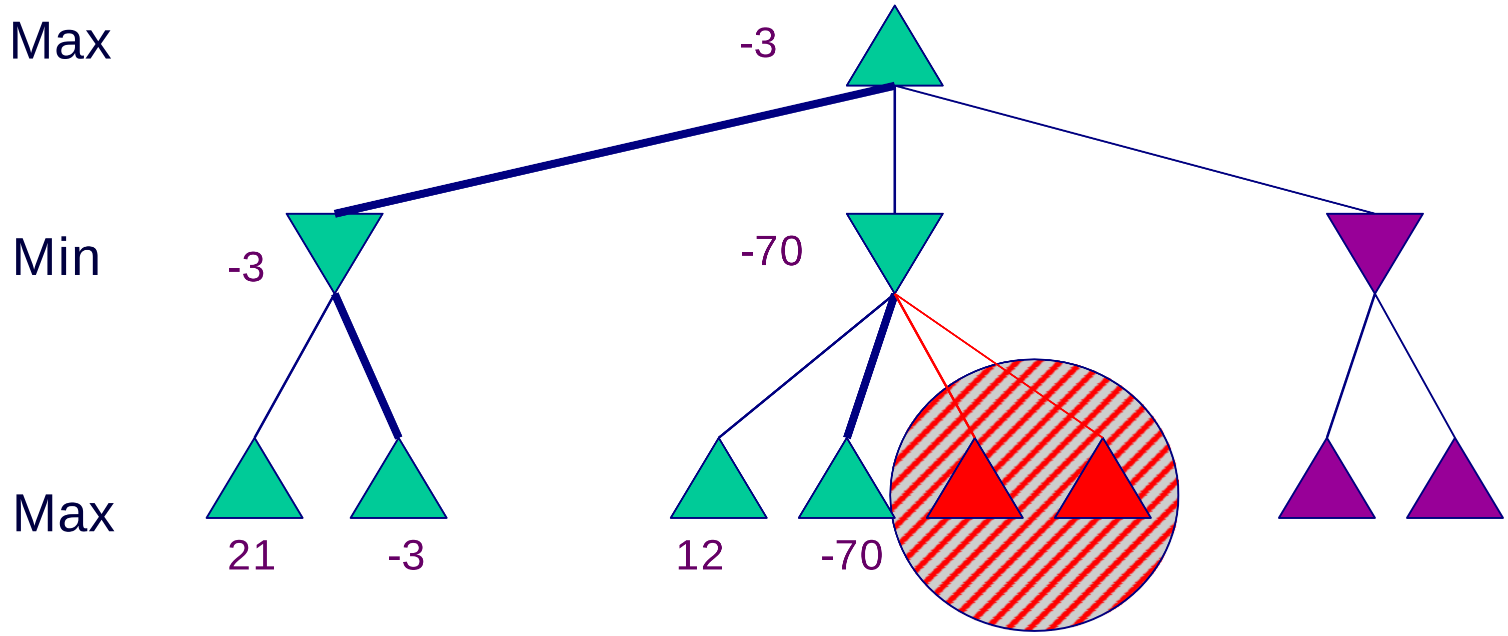
α Cut Example



-70 is now minimum so far (second level)

-3 is still maximum (can't use second node yet)

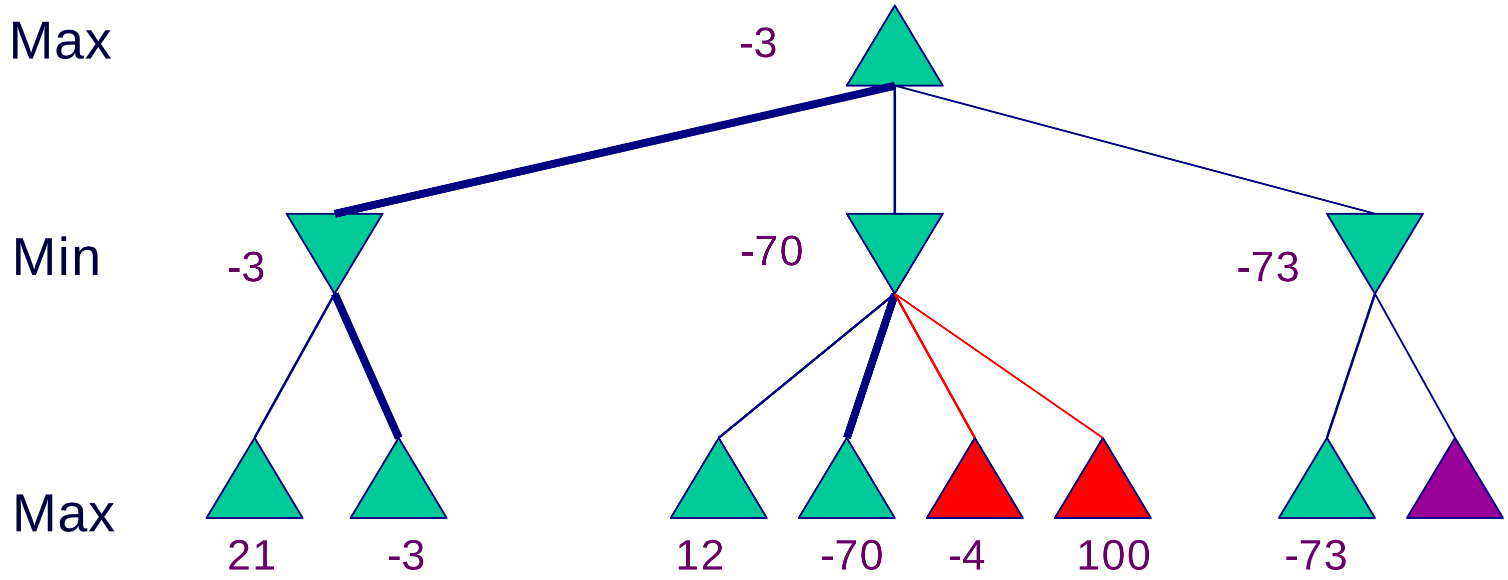
α Cut Example



Since second level node will never be > -70, it will never be chosen by the previous level

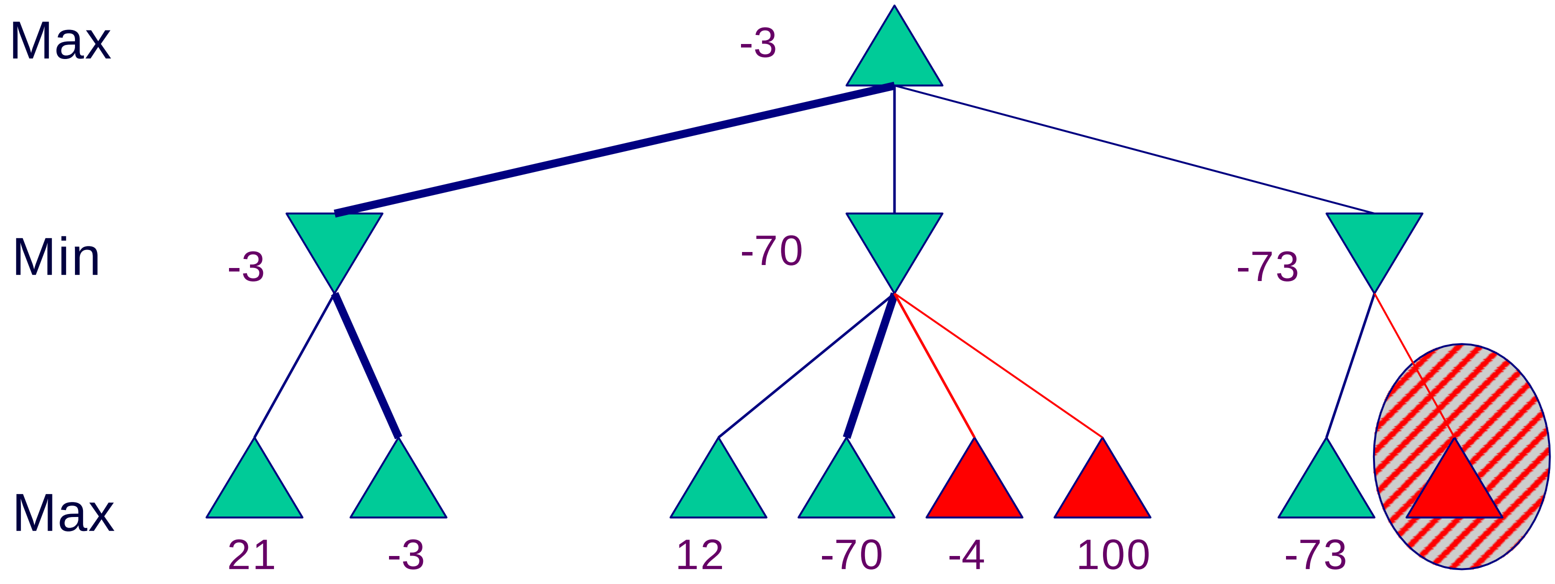
We can stop exploring that node

α Cut Example



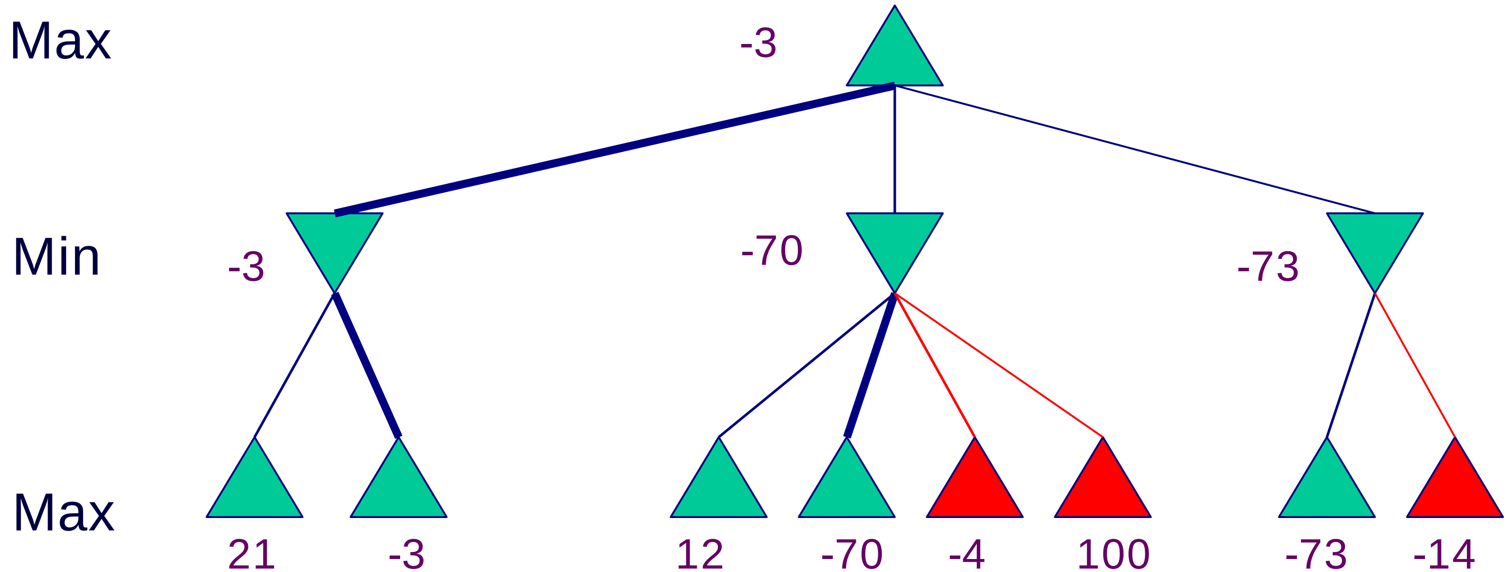
Evaluation at second level is again -73

α Cut Example



Again, can apply α cut since the second level node will never be > -73, and thus will never be chosen by the previous level

α Cut Example



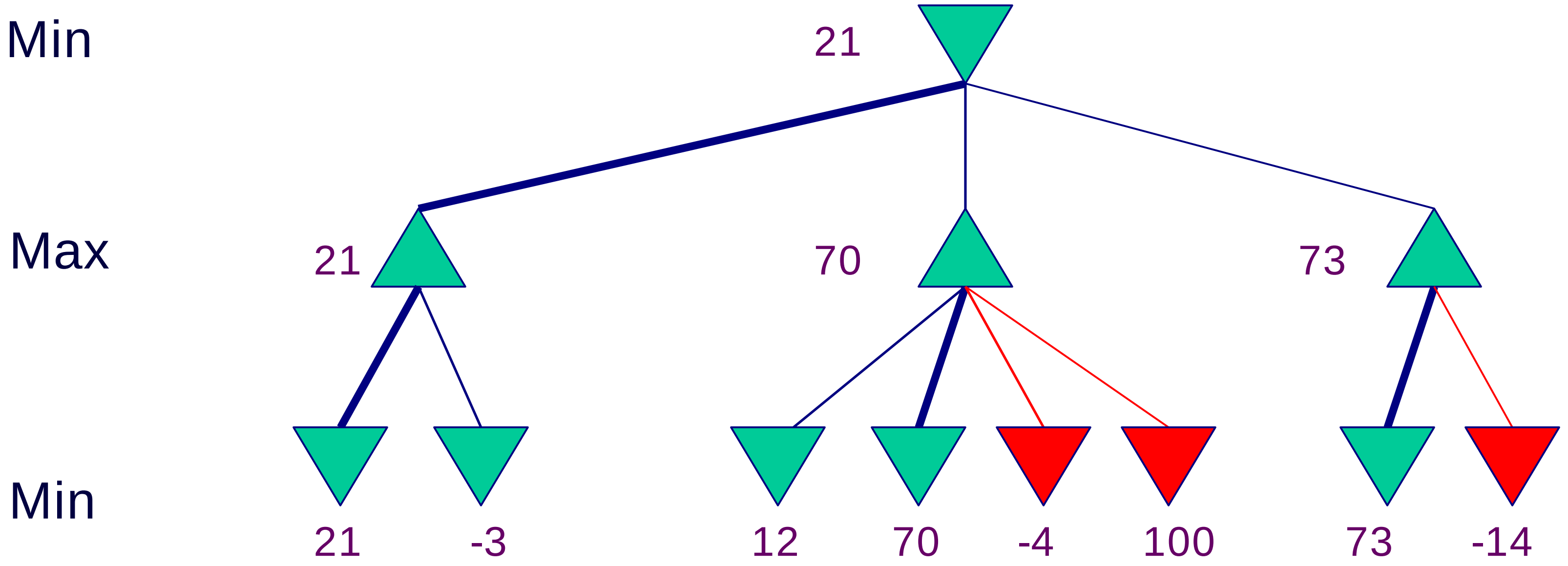
As a result, we evaluated the Max node without evaluating several of the possible paths

β Cuts

Similar idea to α cuts, but the other way around

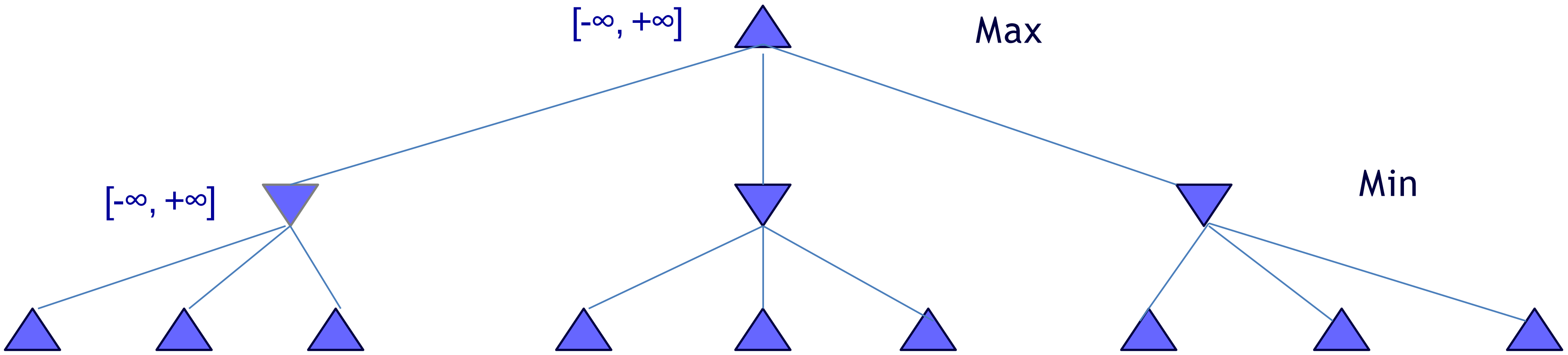
If the current minimum is less than the successor's max value, don't look down that max tree any more

β Cut Example



Some subtrees at second level already have values > min from previous, so we can stop evaluating them.

Alpha-Beta Example 2

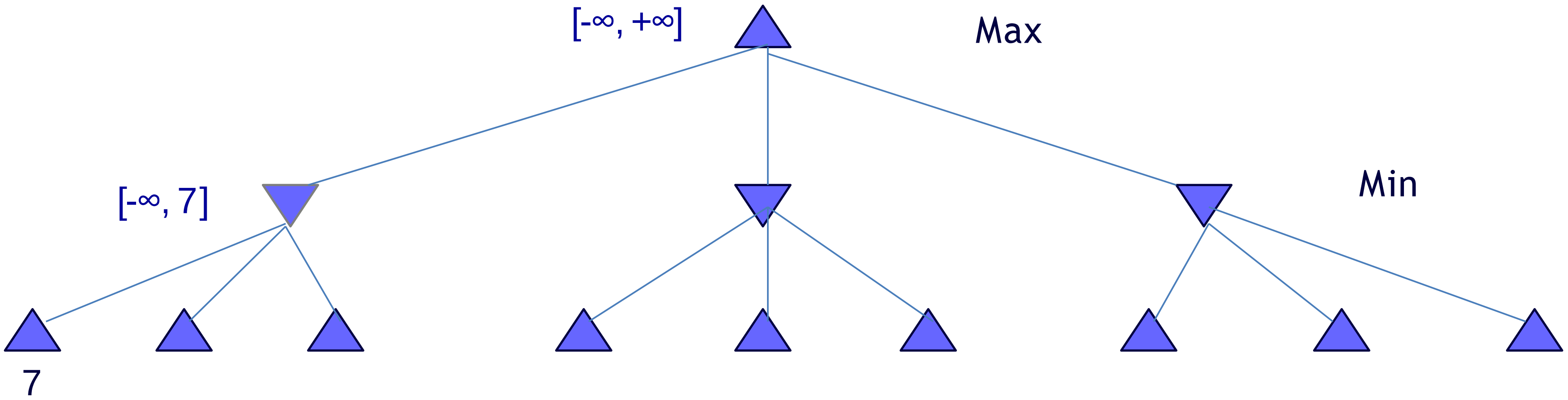


α best choice for Max ?

β best choice for Min ?

- we assume a depth-first, left-to-right search as basic strategy
- the range of the possible values for each node are indicated
  - initially [-∞, +∞]
  - from Max's or Min's perspective
  - these *local* values reflect the values of the sub-trees in that node; the *global* values α and β are the best overall choices so far for Max or Min

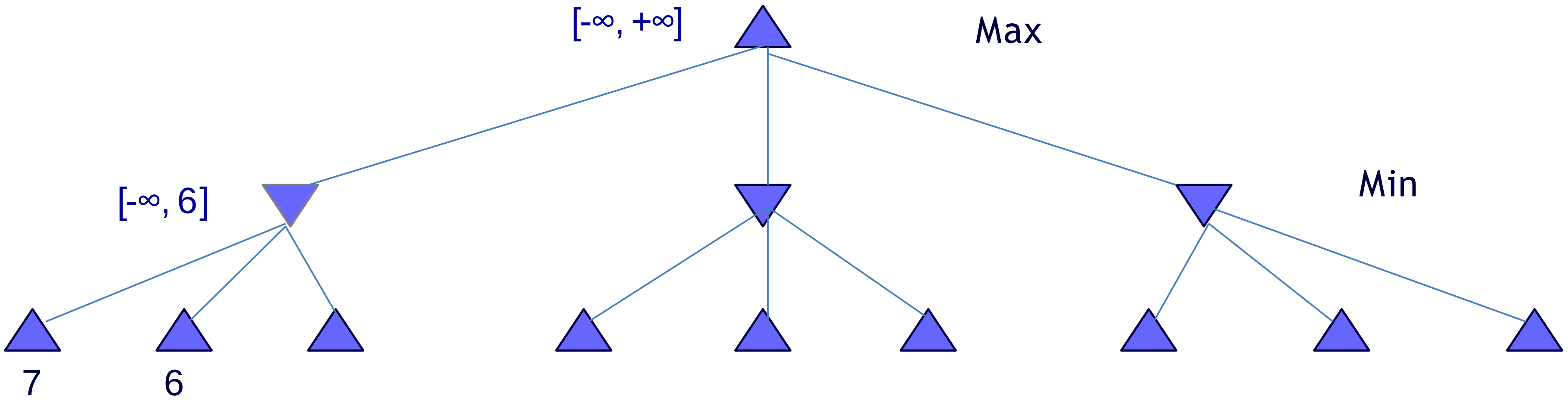
Alpha-Beta Example 2



α best choice for Max ?

β best choice for Min 7

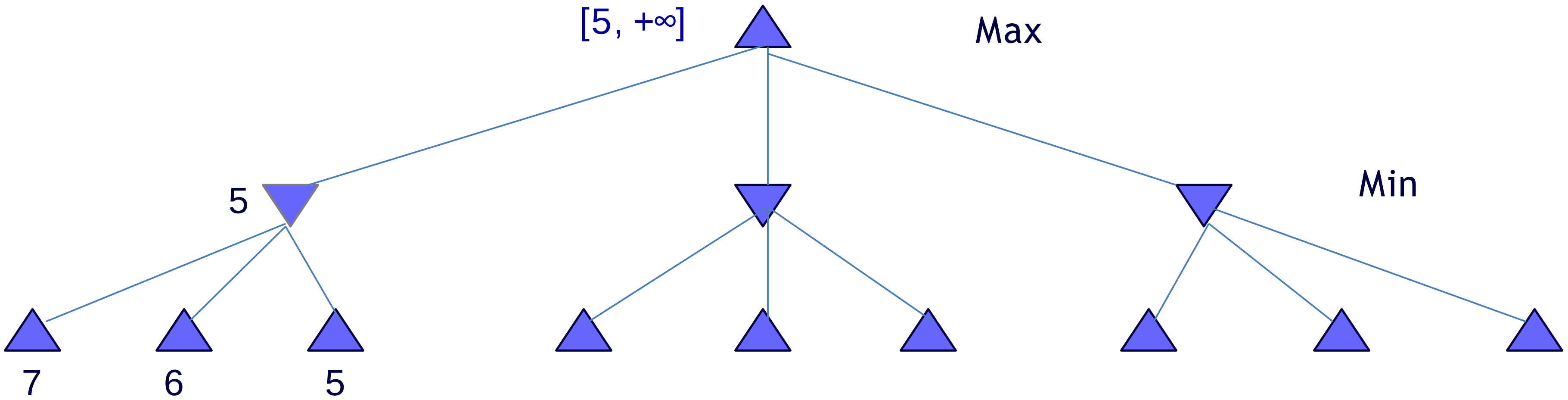
Alpha-Beta Example 2



α best choice for Max ?

β best choice for Min 6

Alpha-Beta Example 2

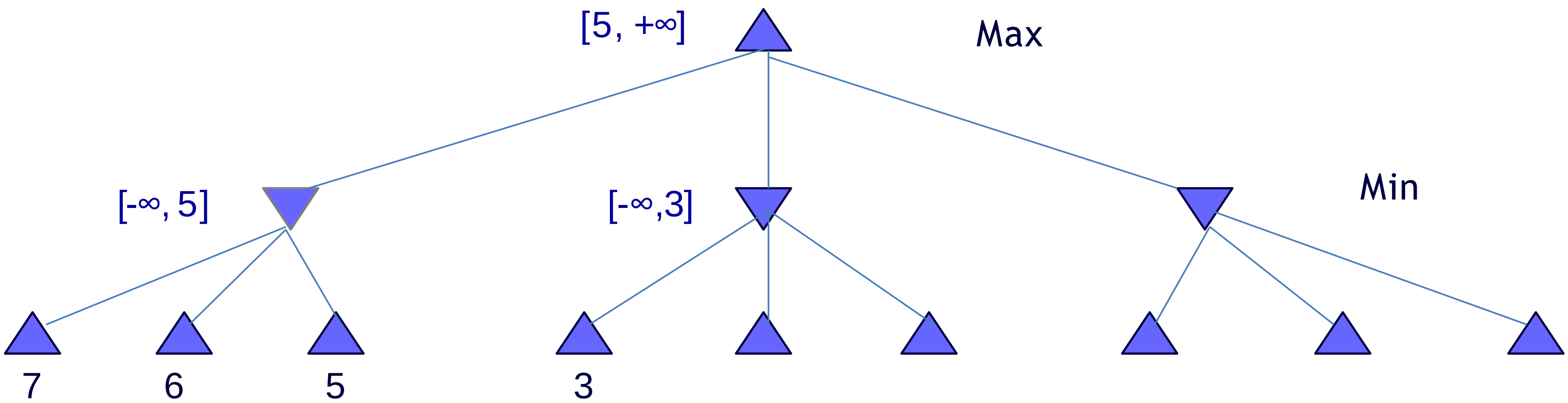


α best choice for Max 5

β best choice for Min 5

- Min obtains the third value from a successor node
- this is the last value from this sub-tree, and the exact value is known
- Max now has a value for its first successor node, but hopes that something better might still come

Alpha-Beta Example 2

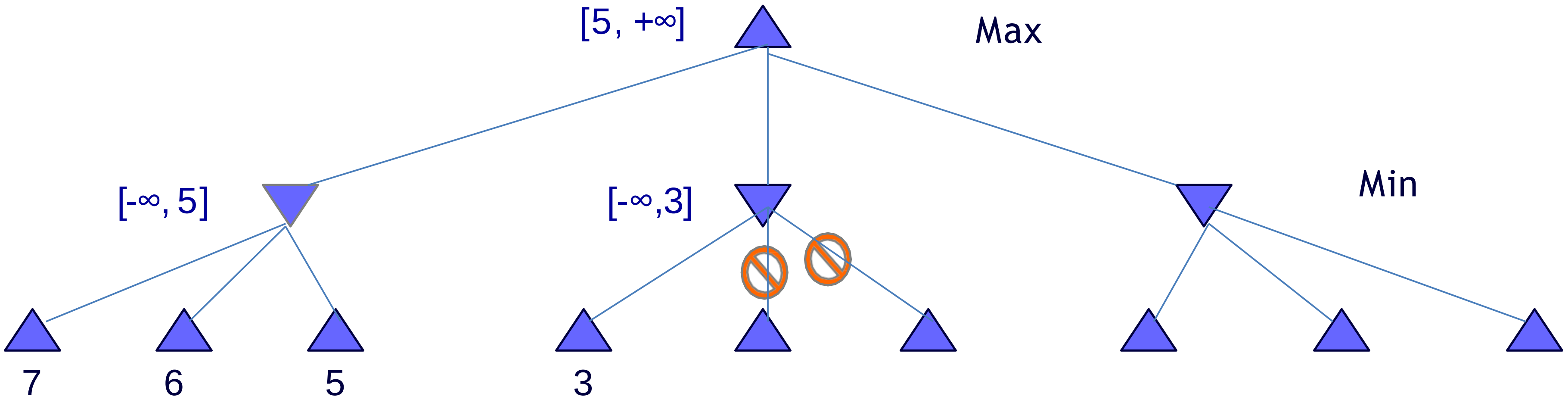


α best choice for Max 5

β best choice for Min 3

– Min continues with the next sub-tree, and gets a better value
– Max has a better choice from its perspective, however, and will not consider a move in the sub-tree currently explored by min
– Initially [-∞, +∞]

Alpha-Beta Example 2

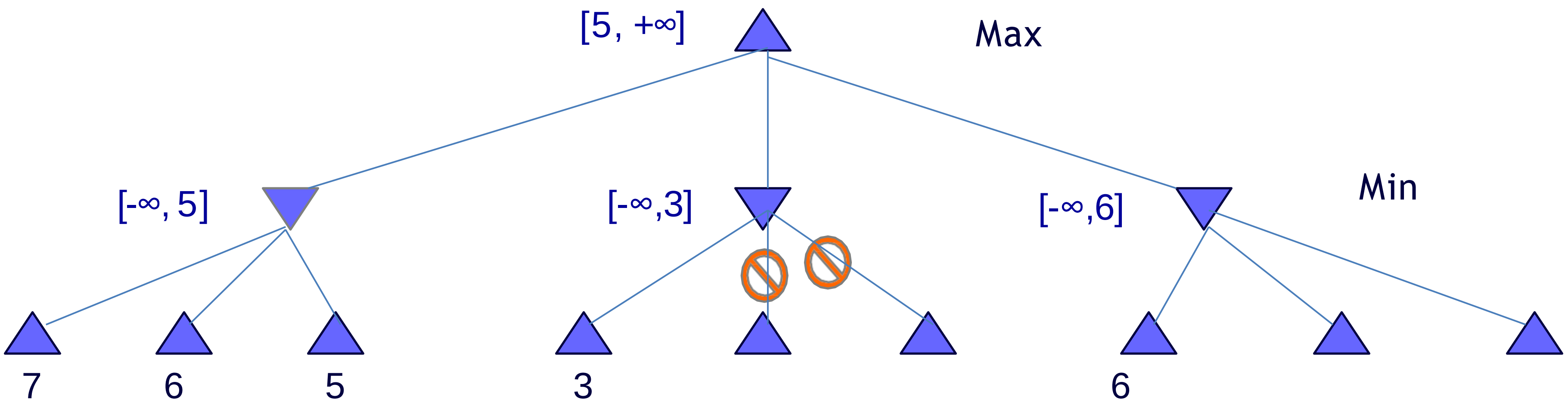


α best choice for Max 5

β best choice for Min 3

- Min knows that Max won't consider a move to this sub-tree, and abandons it
- this is a case of pruning, indicated by

Alpha-Beta Example 2

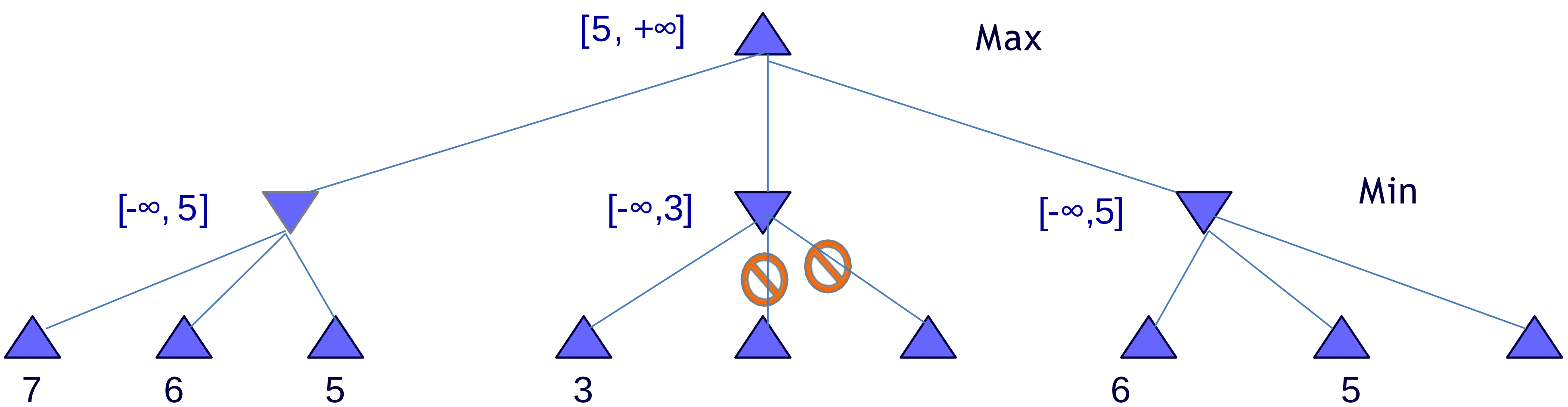


α best choice for Max 5

β best choice for Min 3

- Min explores the next sub-tree, and finds a value that is worse than the other nodes at this level
- if Min is not able to find something lower, then Max will choose this branch, so Min must explore more successor nodes

Alpha-Beta Example 2

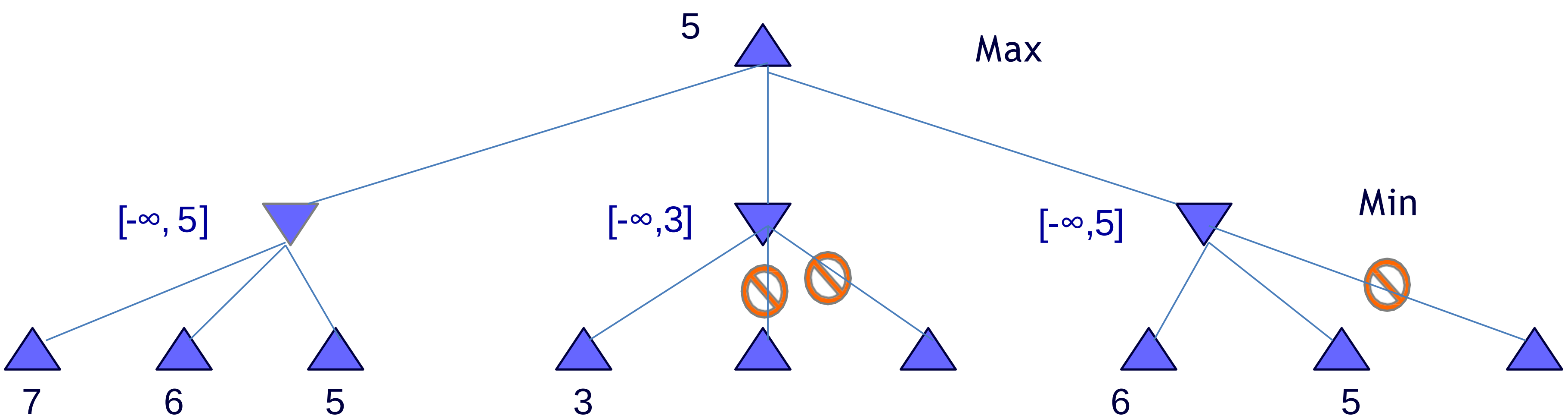


α best choice for Max 5

β best choice for Min 3

- Min is lucky, and finds a value that is the same as the current worst value at this level
- Max can choose this branch, or the other branch with the same value

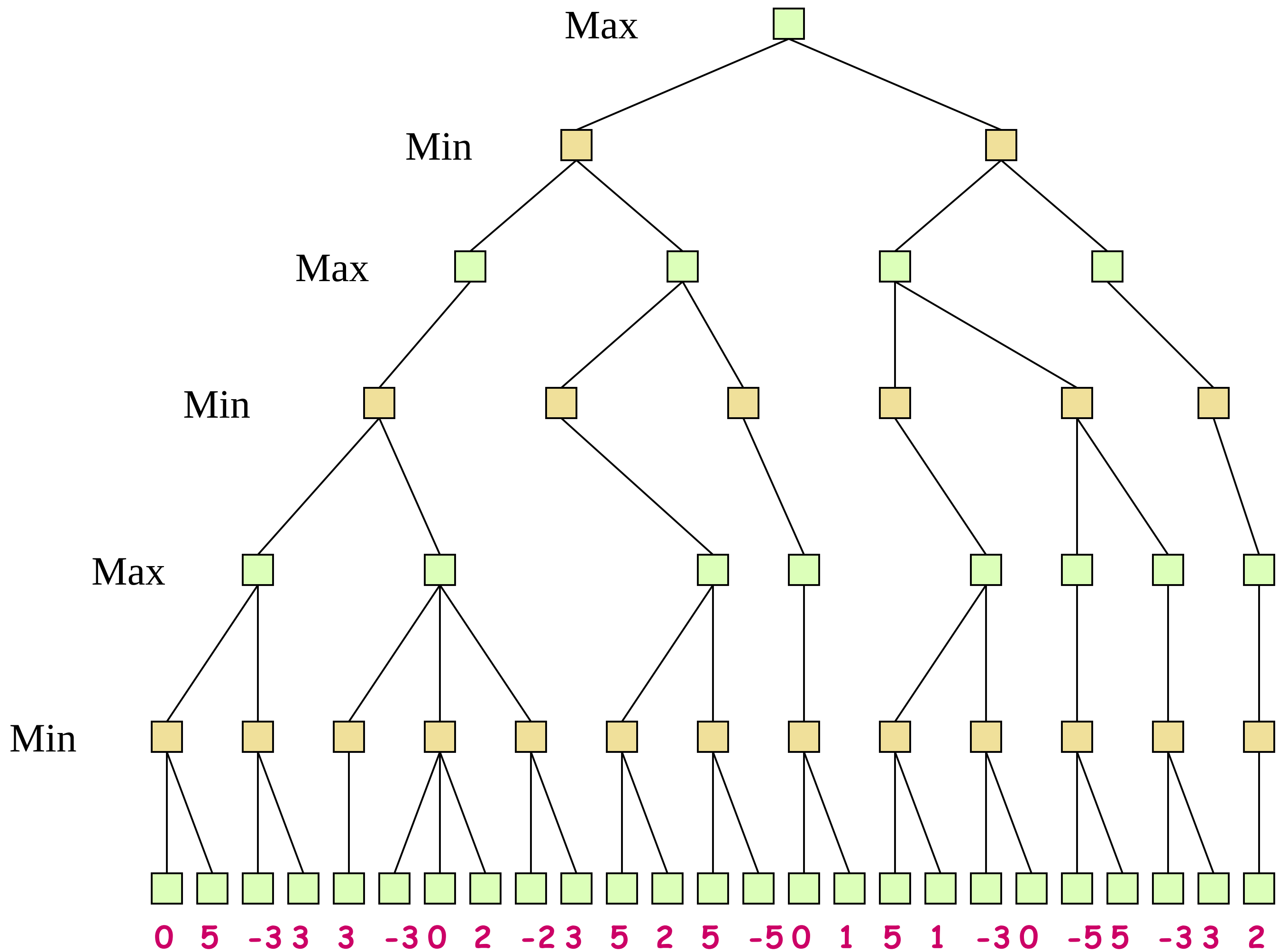
Alpha-Beta Example 2

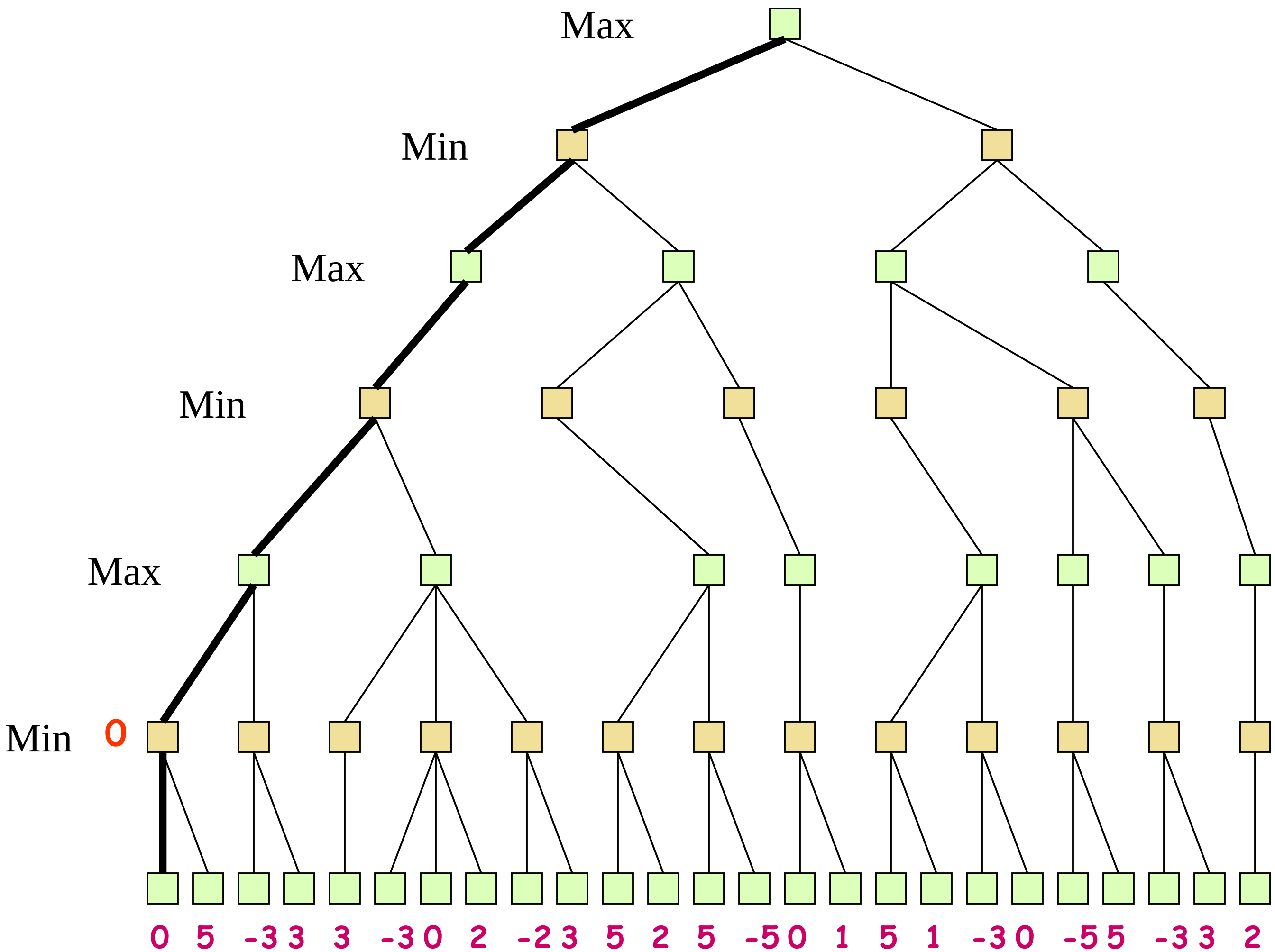


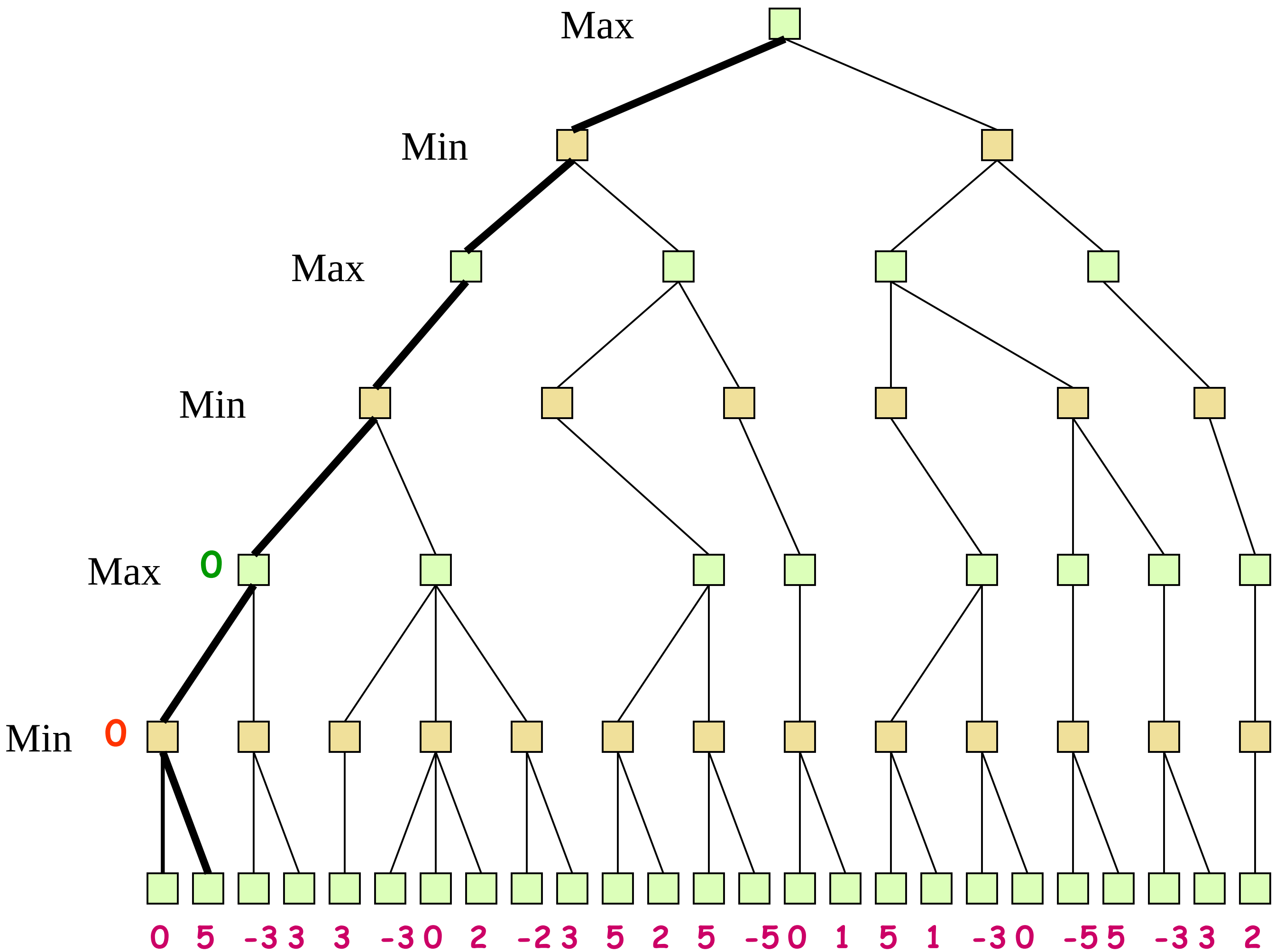
α best choice for Max 5
β best choice for Min 3

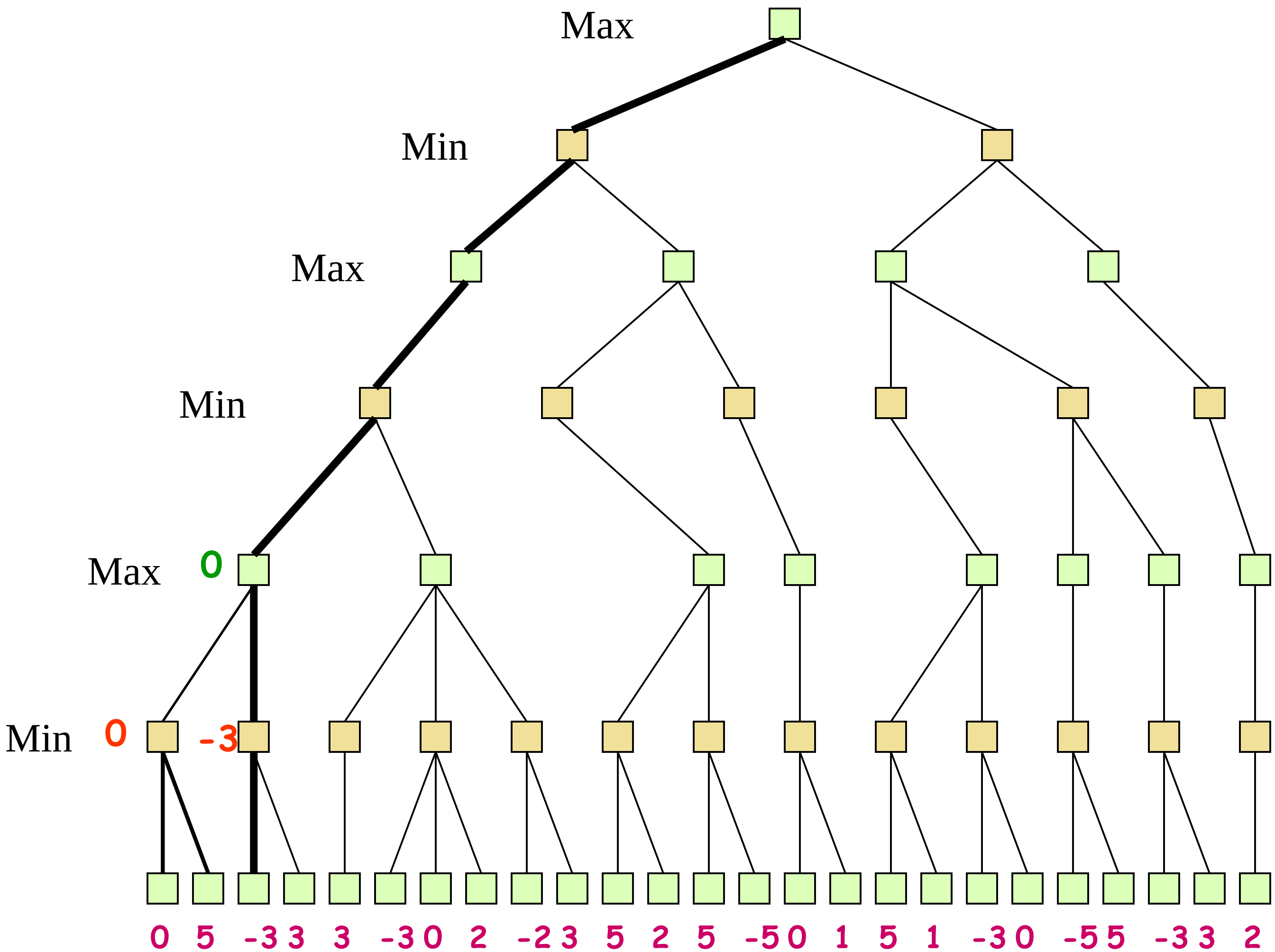
- Min could continue searching this sub-tree to see if there is a value that is less than the current worst alternative in order to give Max as few choices as possible
- this depends on the specific implementation
- Max knows the best value for its sub-tree

Alpha-Beta Tic-Tac-Toe Example 2

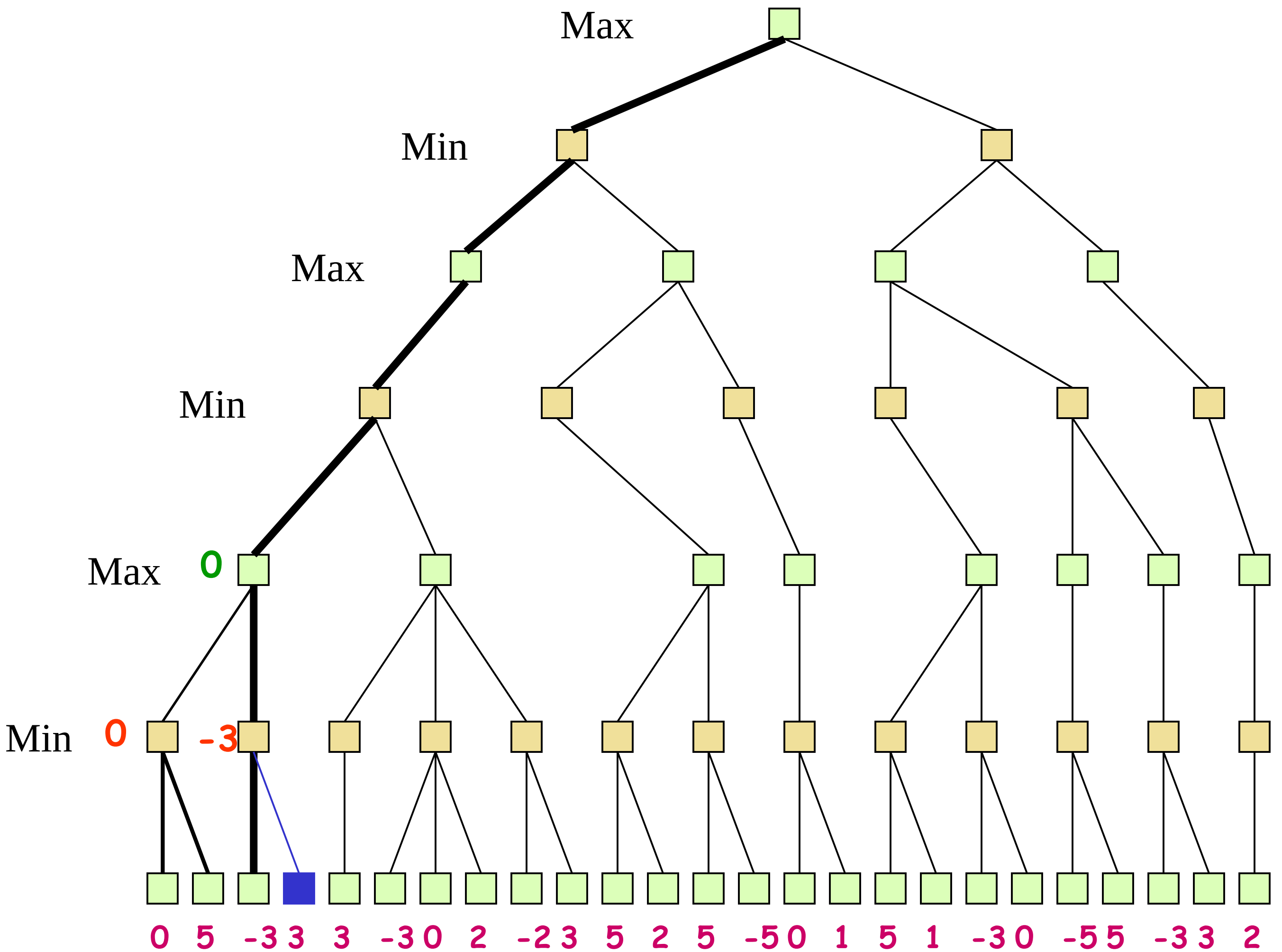


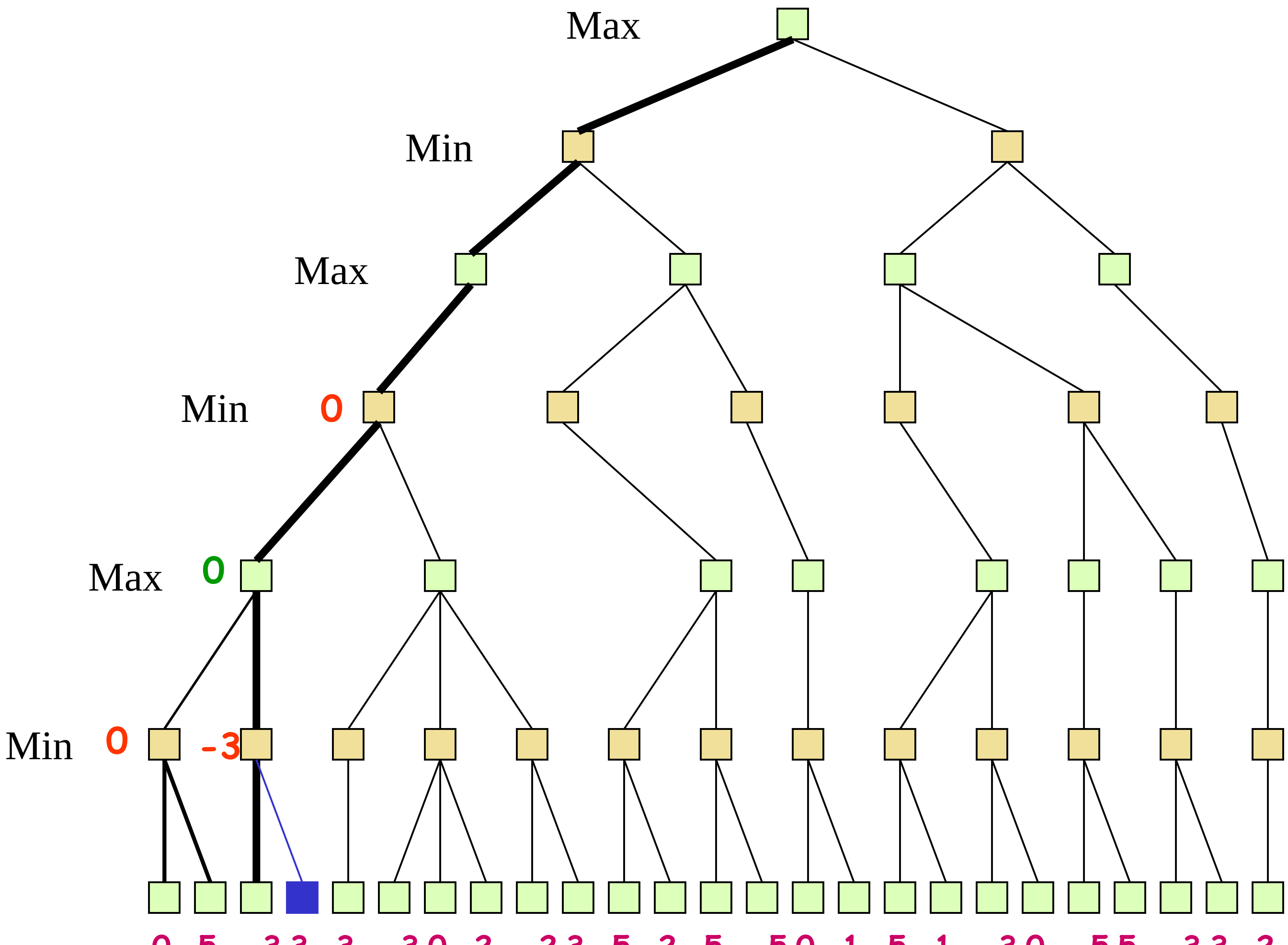

Max
Min
Max
Min
Max
Min
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


Max
Min
Max
Min
Max
0
Min
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

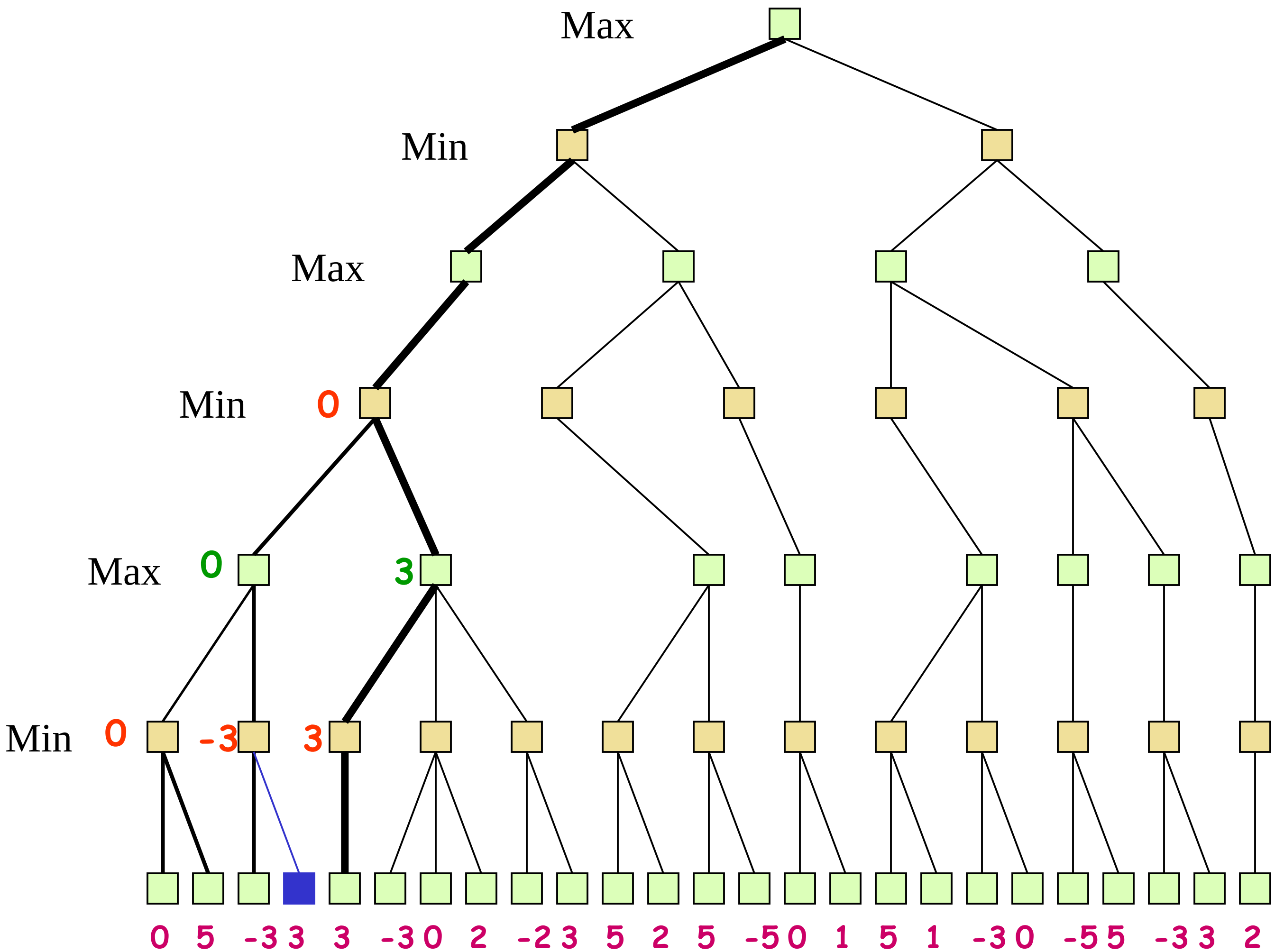


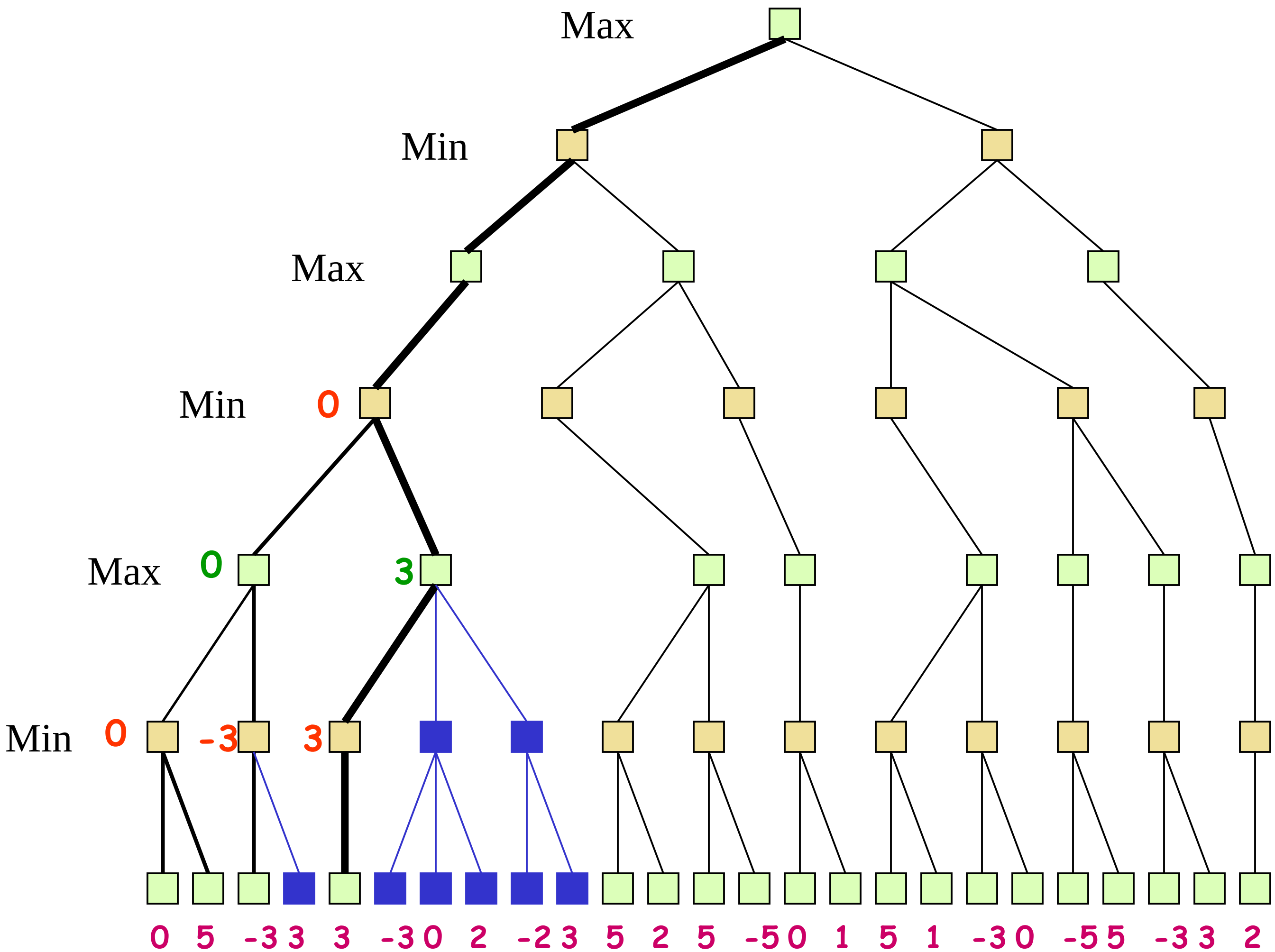
Max
Min
Max
Min
Max
0
Min
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


Max
Min
Max
Min
Max
0
Min
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

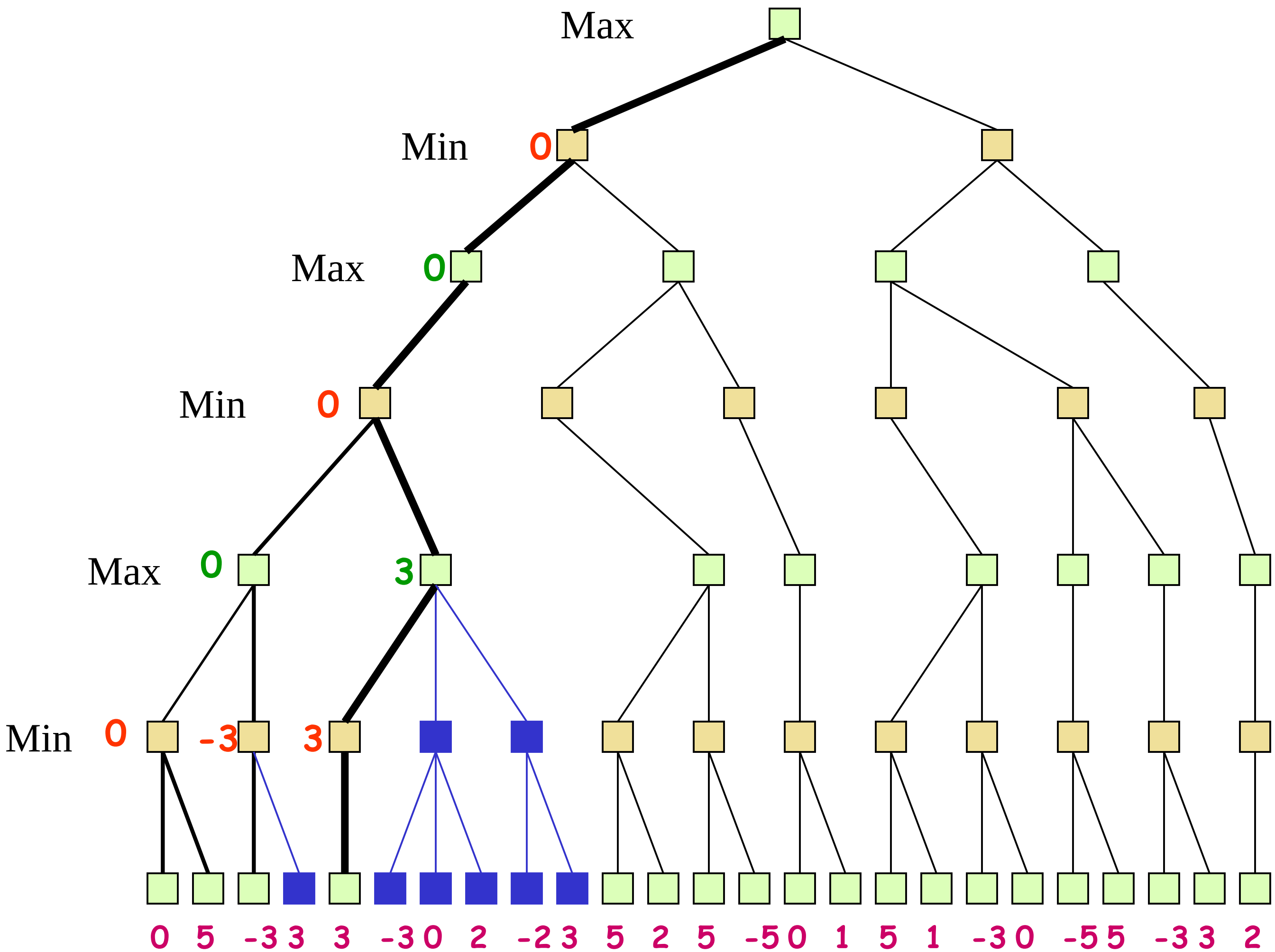


Max
Min
Max
Min
0
Max
0
Min
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

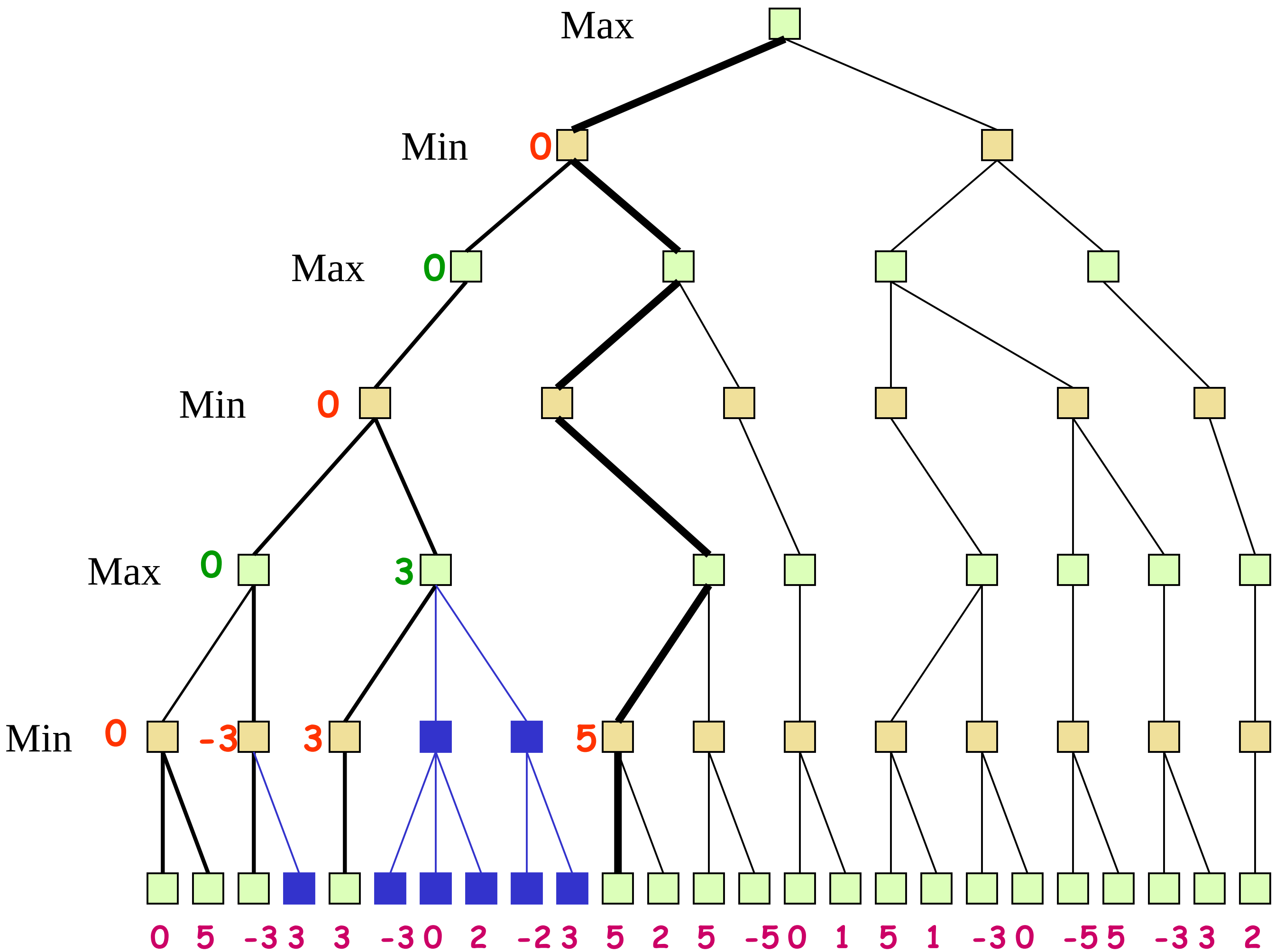

Max
Min
Max
Min
0
Max
0
3
Min
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



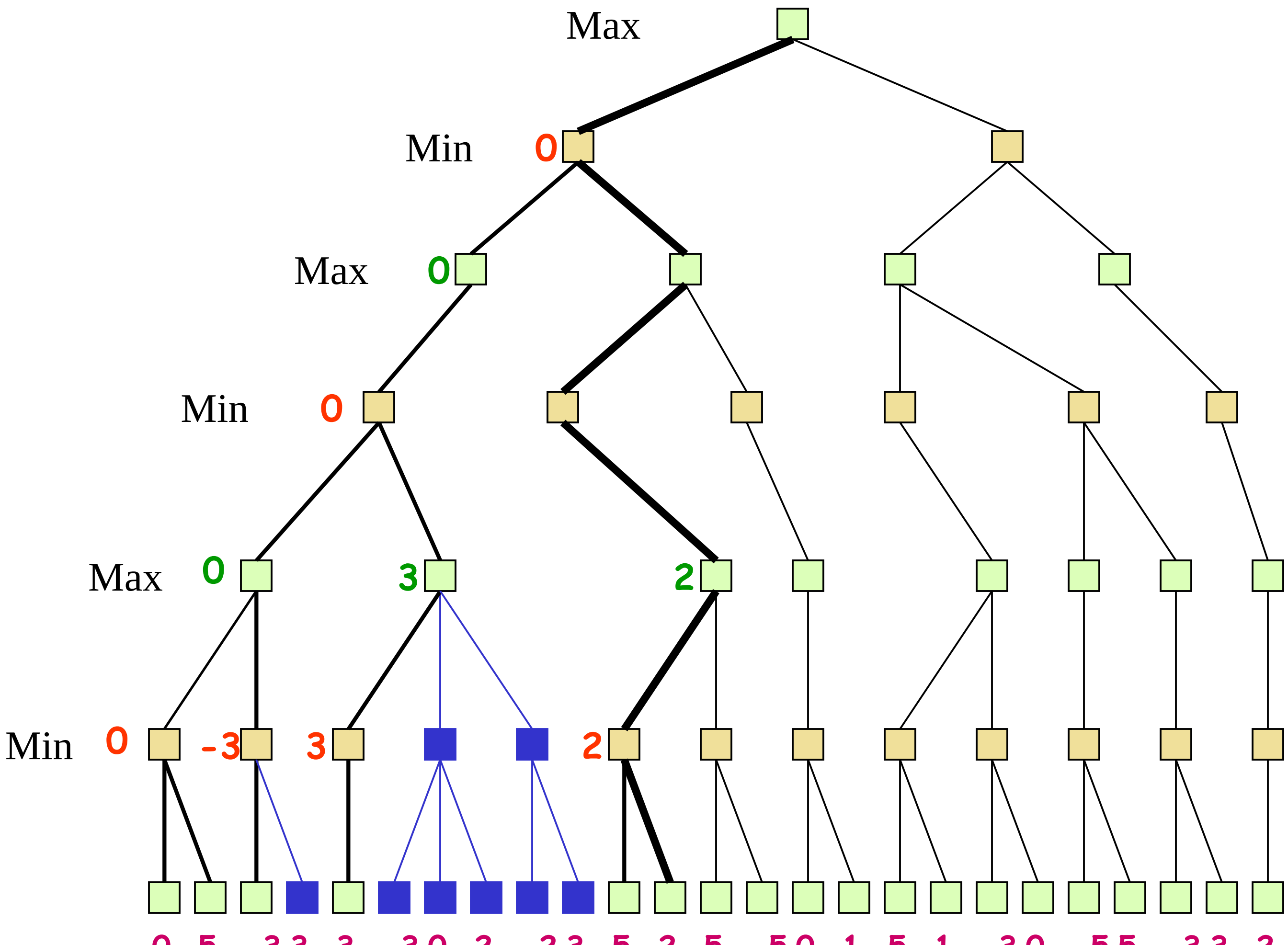
Max
Min
Max
Min
0
Max
0
3
Min
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



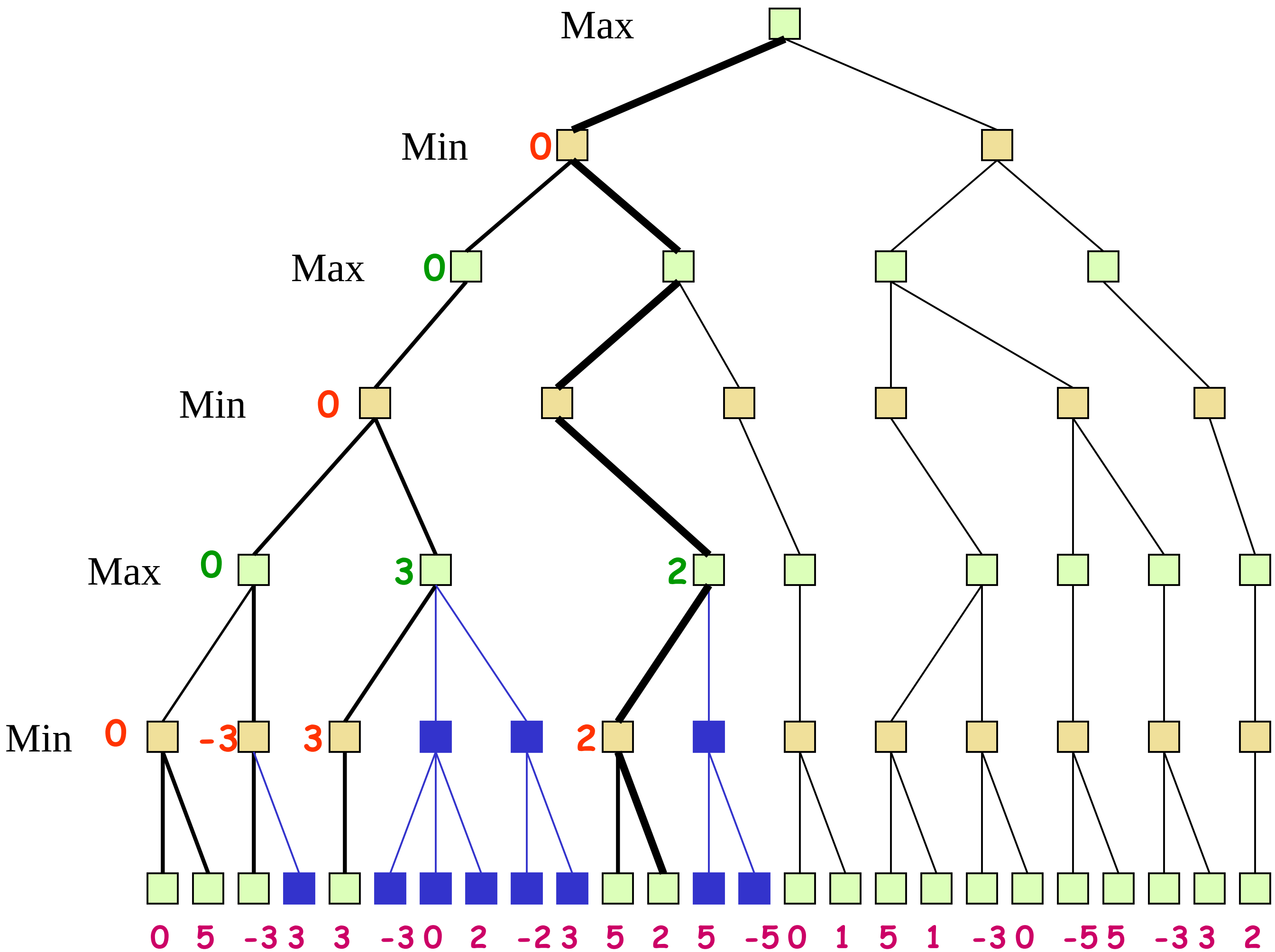
Max
Min
0
Max
0
Min
0
Max
0
3
Min
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



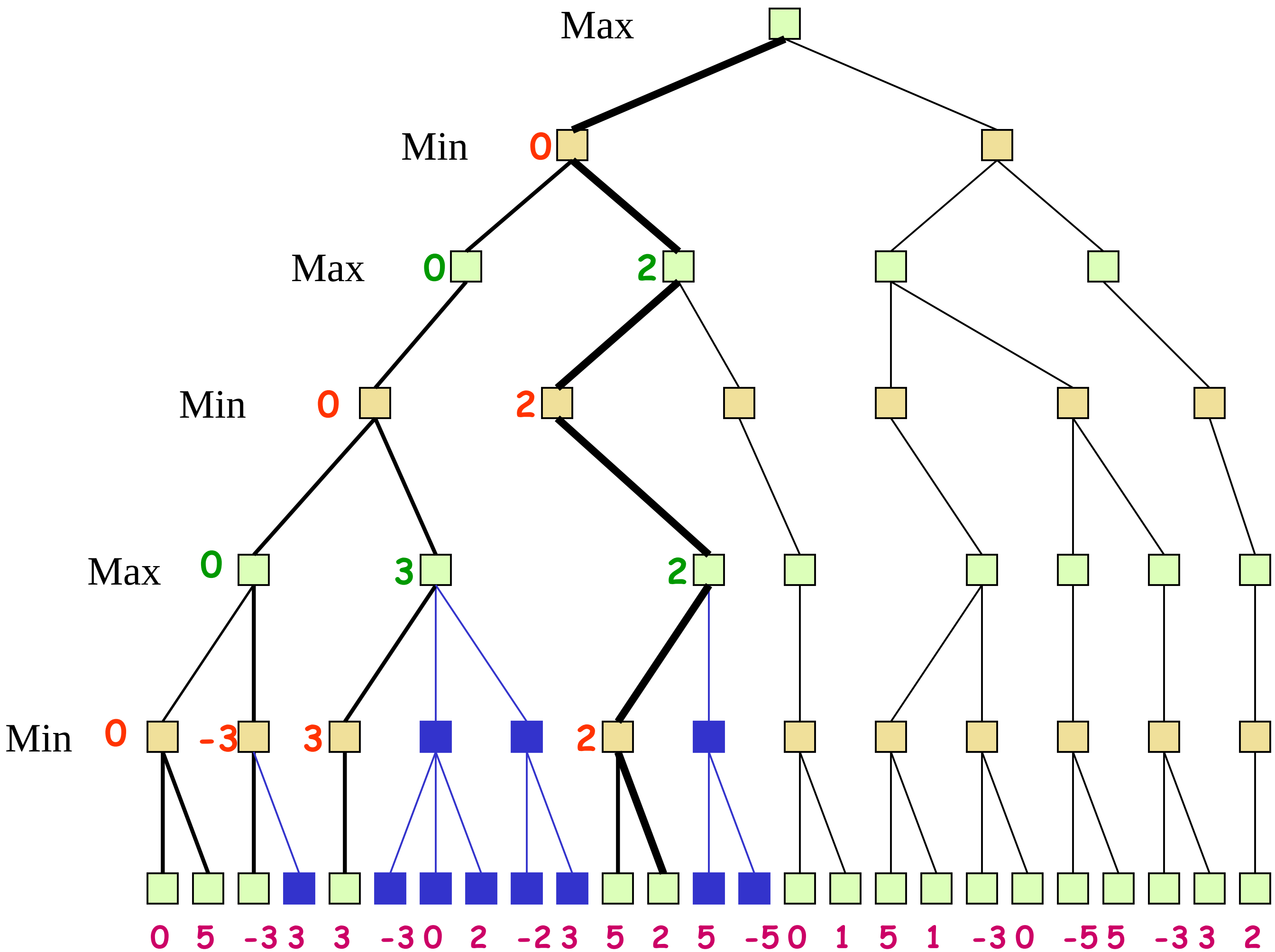
Max
Min
Max
Min
Max
Min
0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



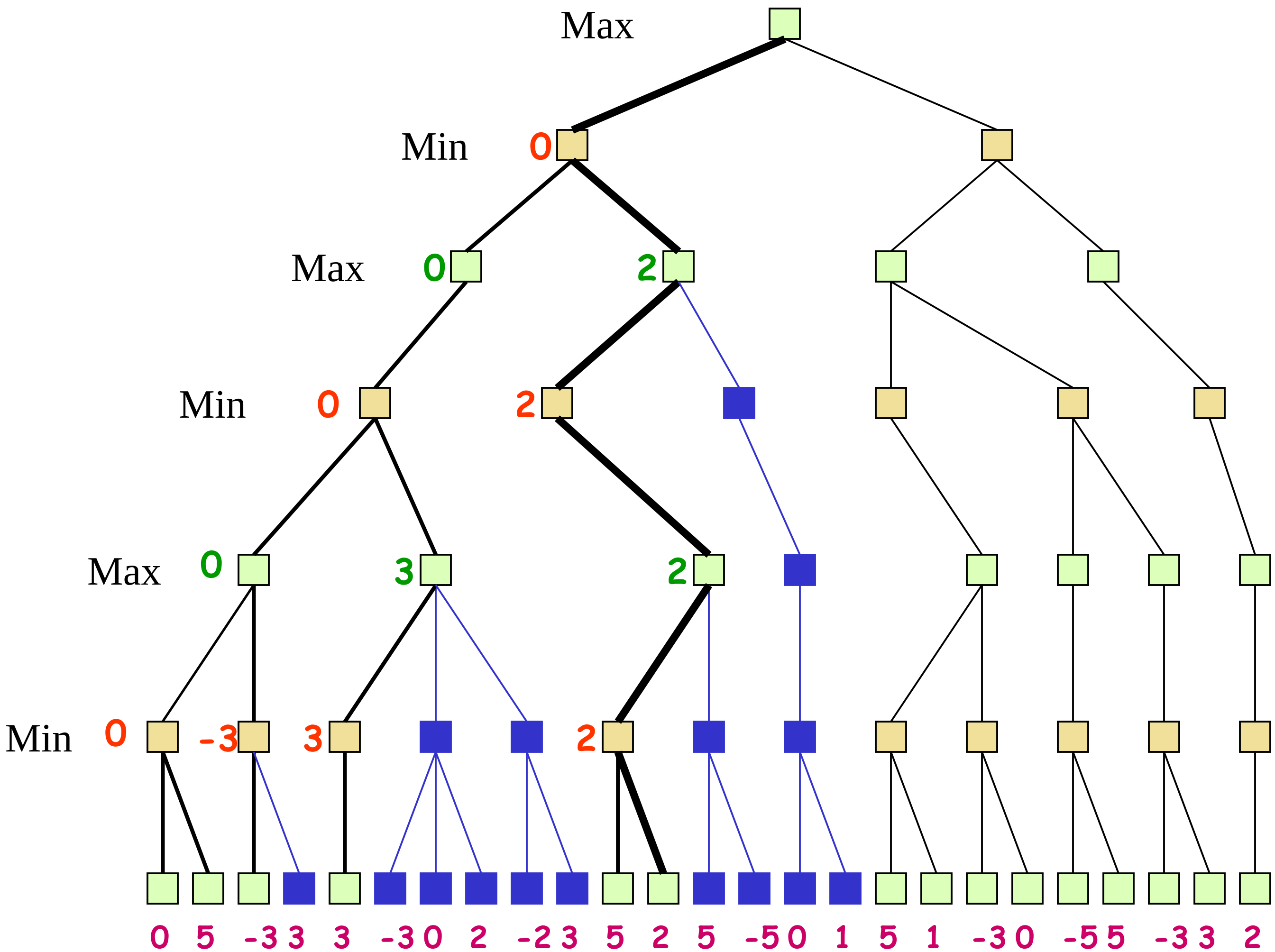
Max
Min
0
Max
0
Min
0
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

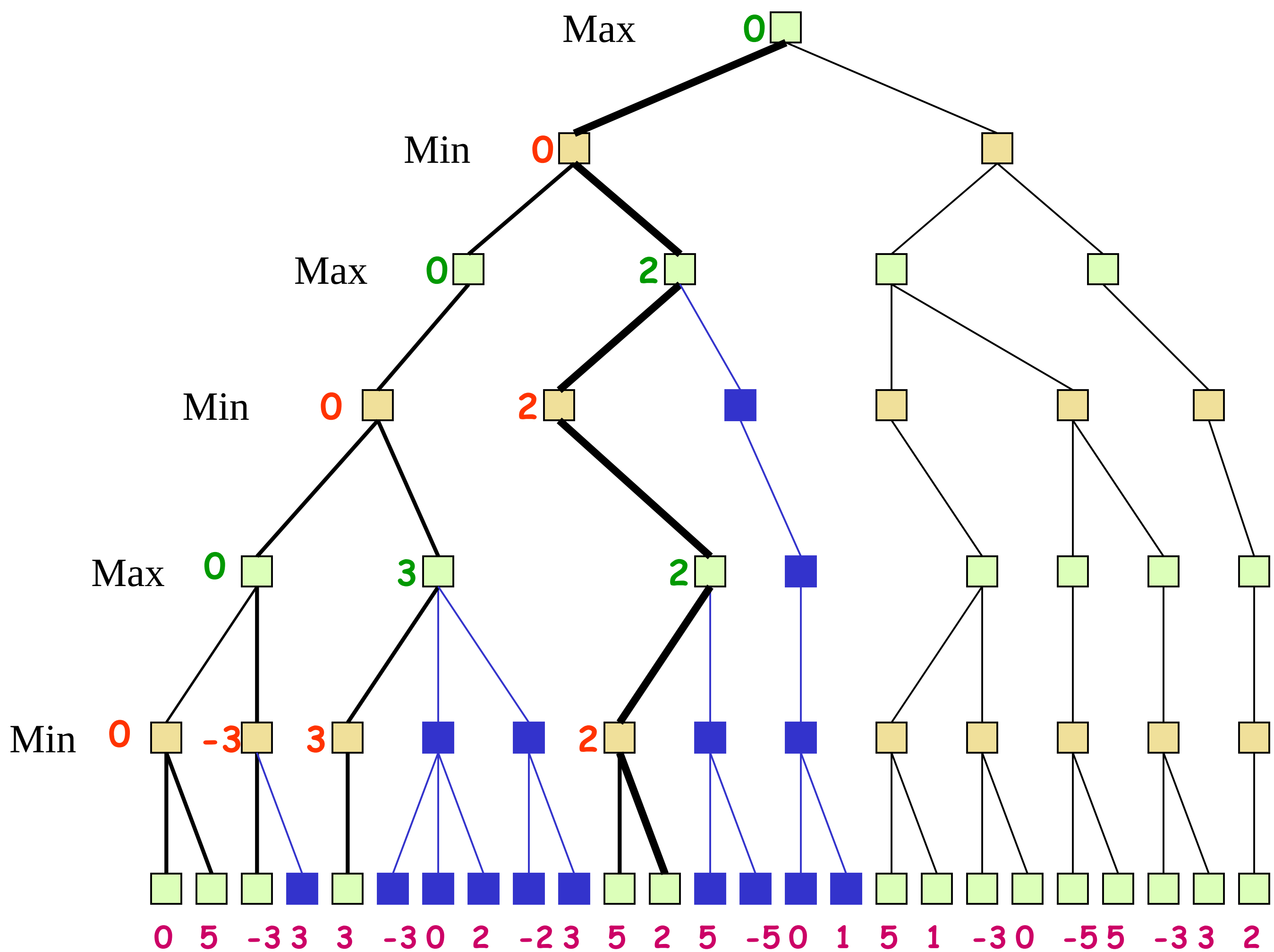


Max
Min
Max
Min
Max
Min
0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

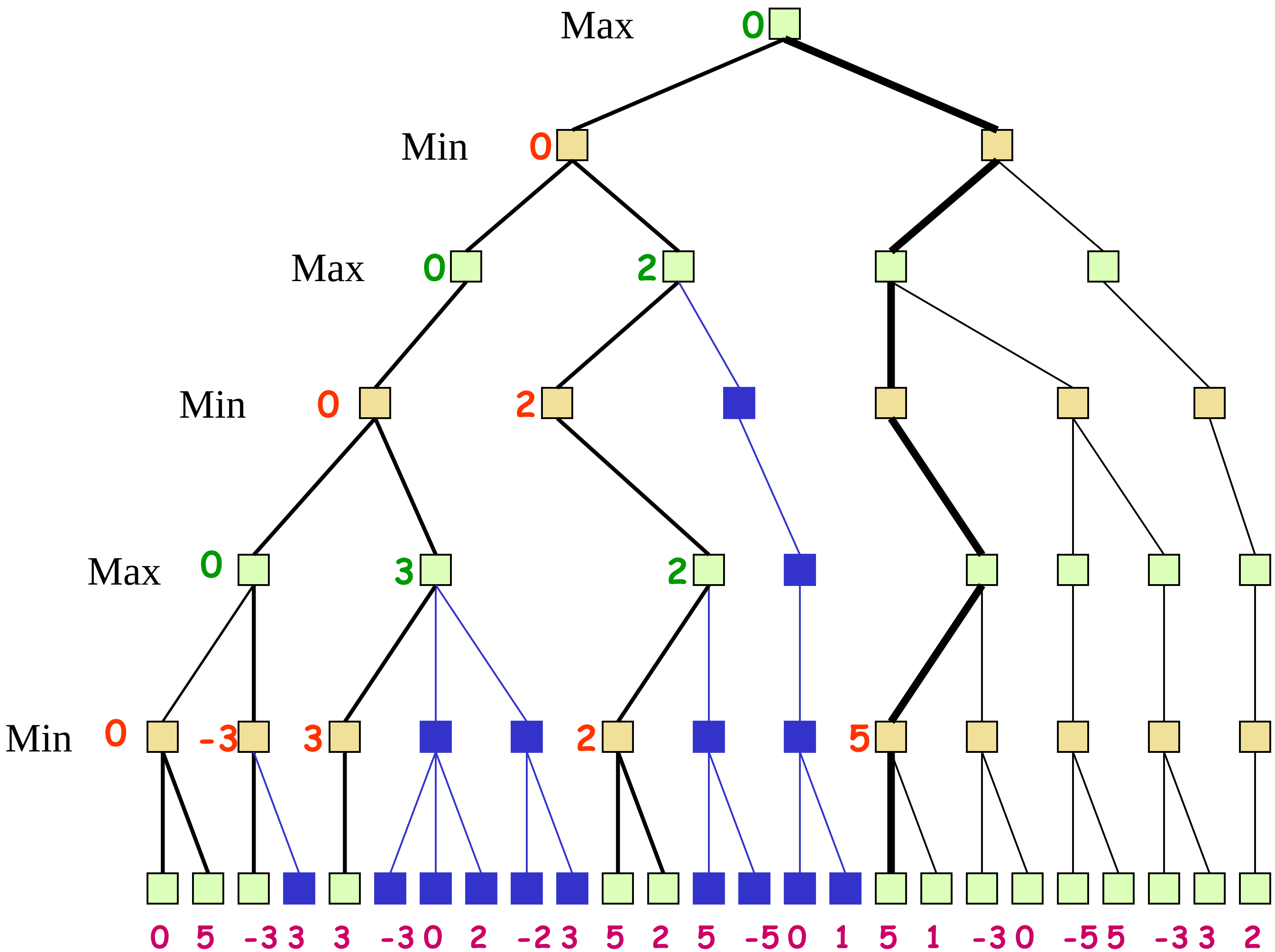


Max
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

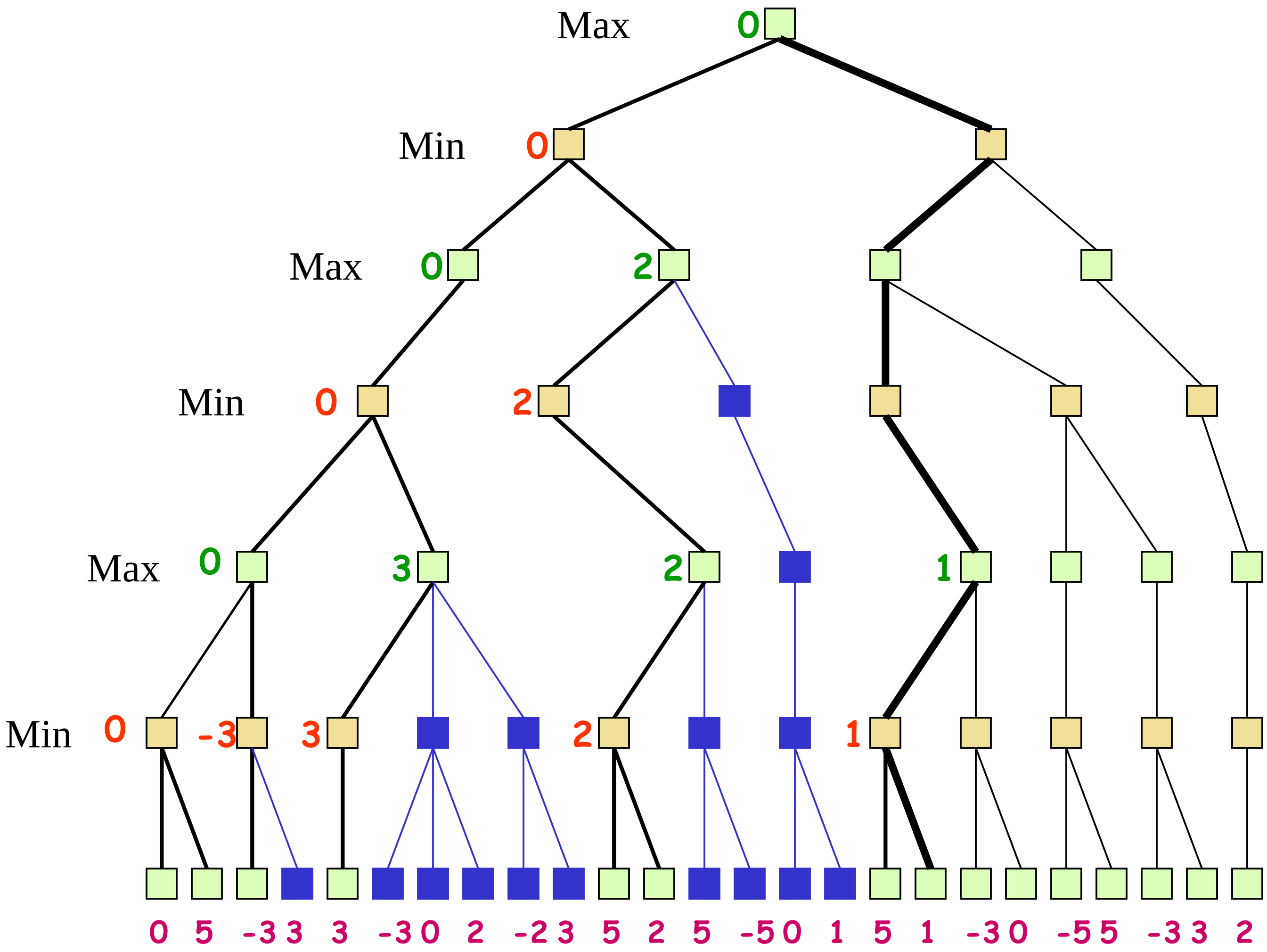

Max
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

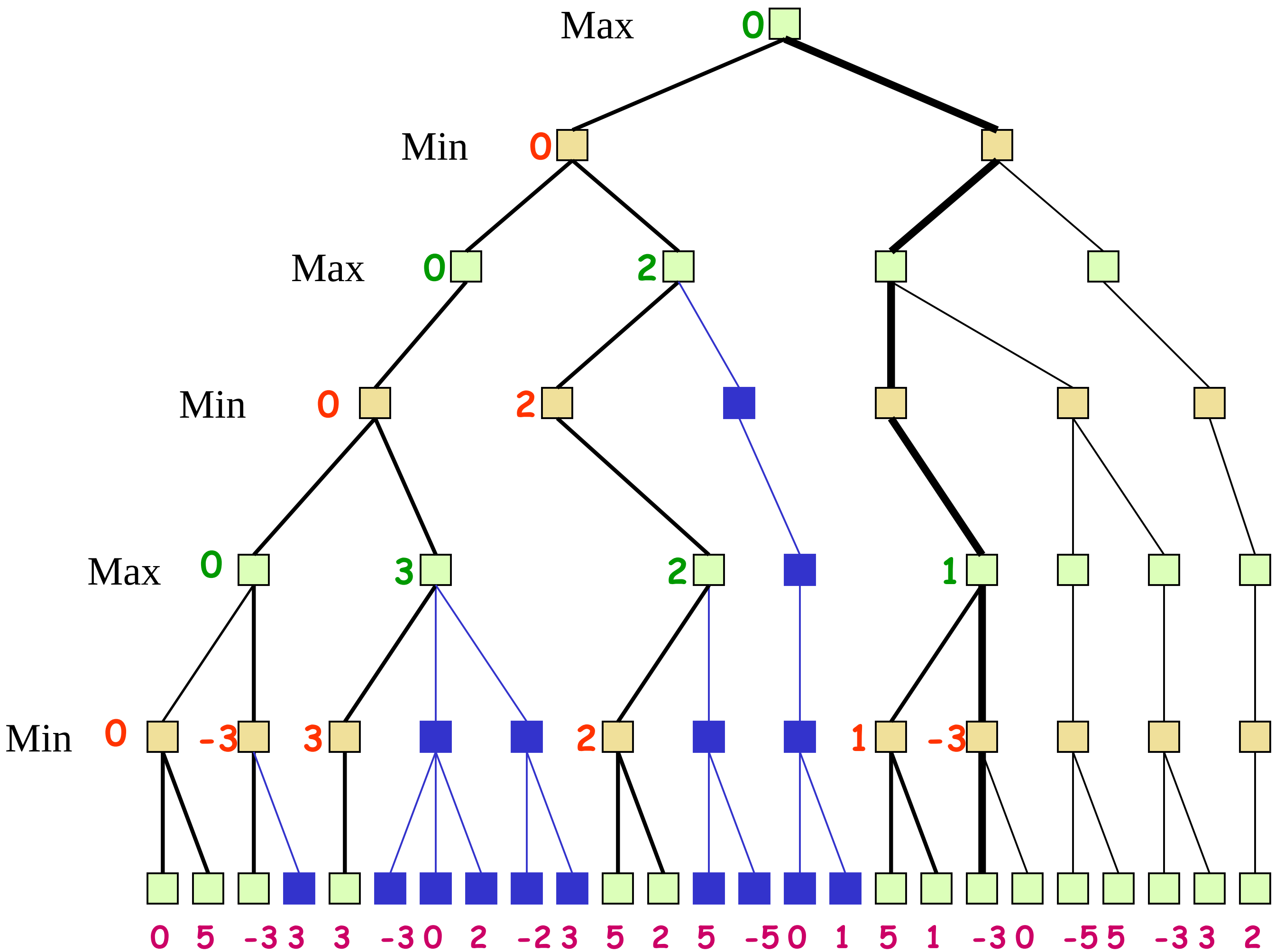


Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

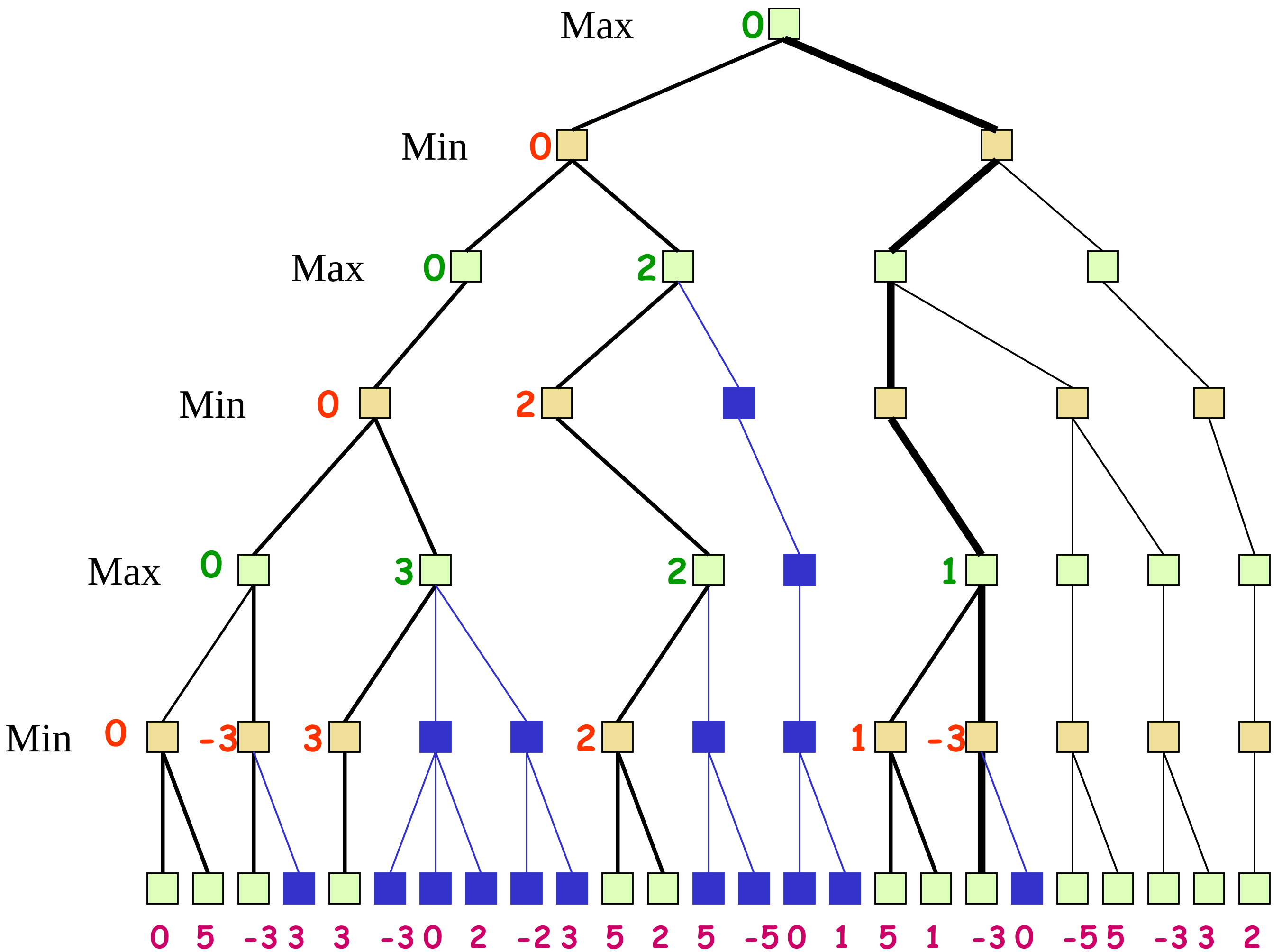


Max
Min
Max
Min
Max
Min
0
0
0 2
0 2
0 3 2
0 -3 3 2 5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

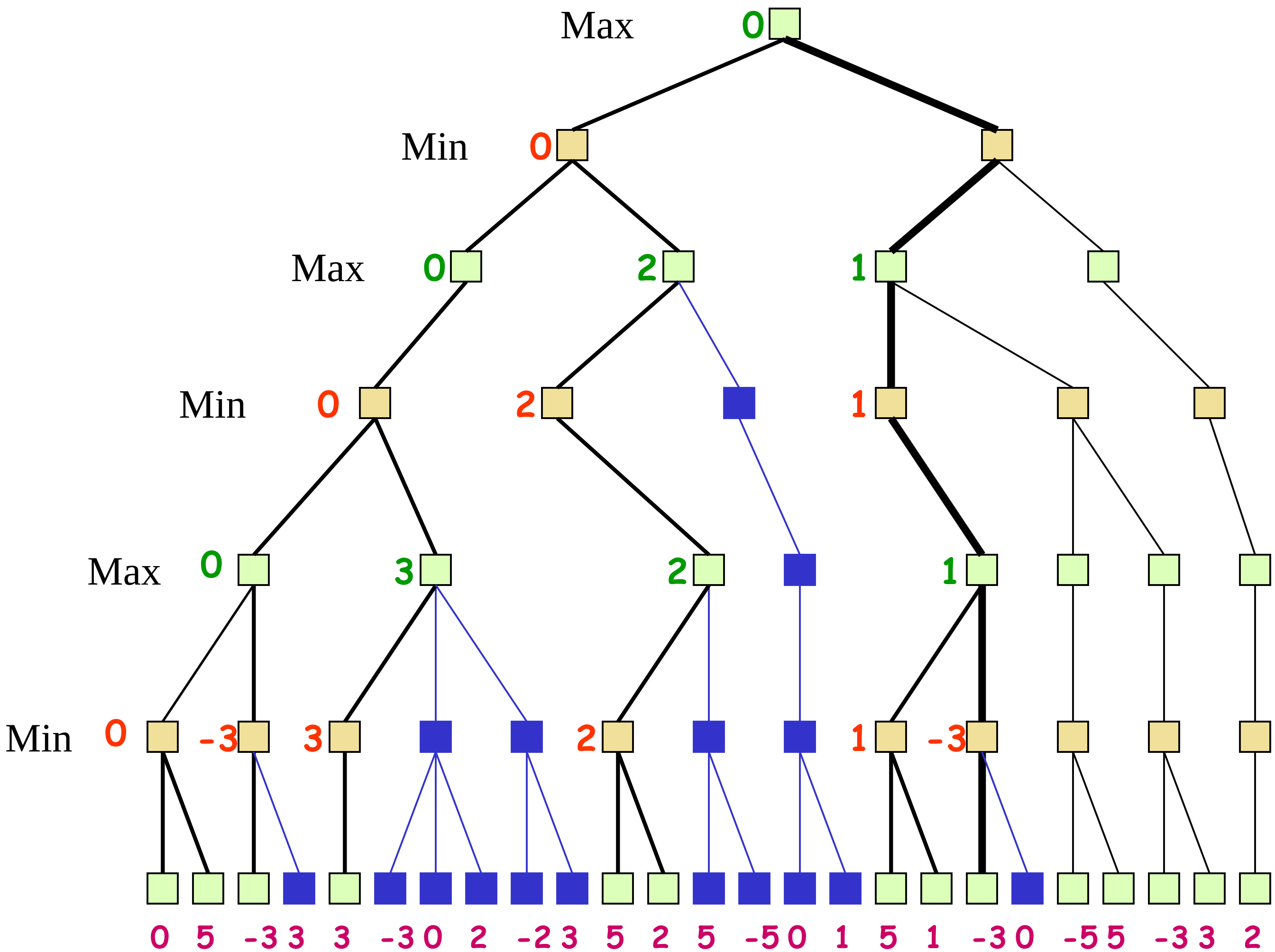

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
1
Min
0
-3
3
2
1
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



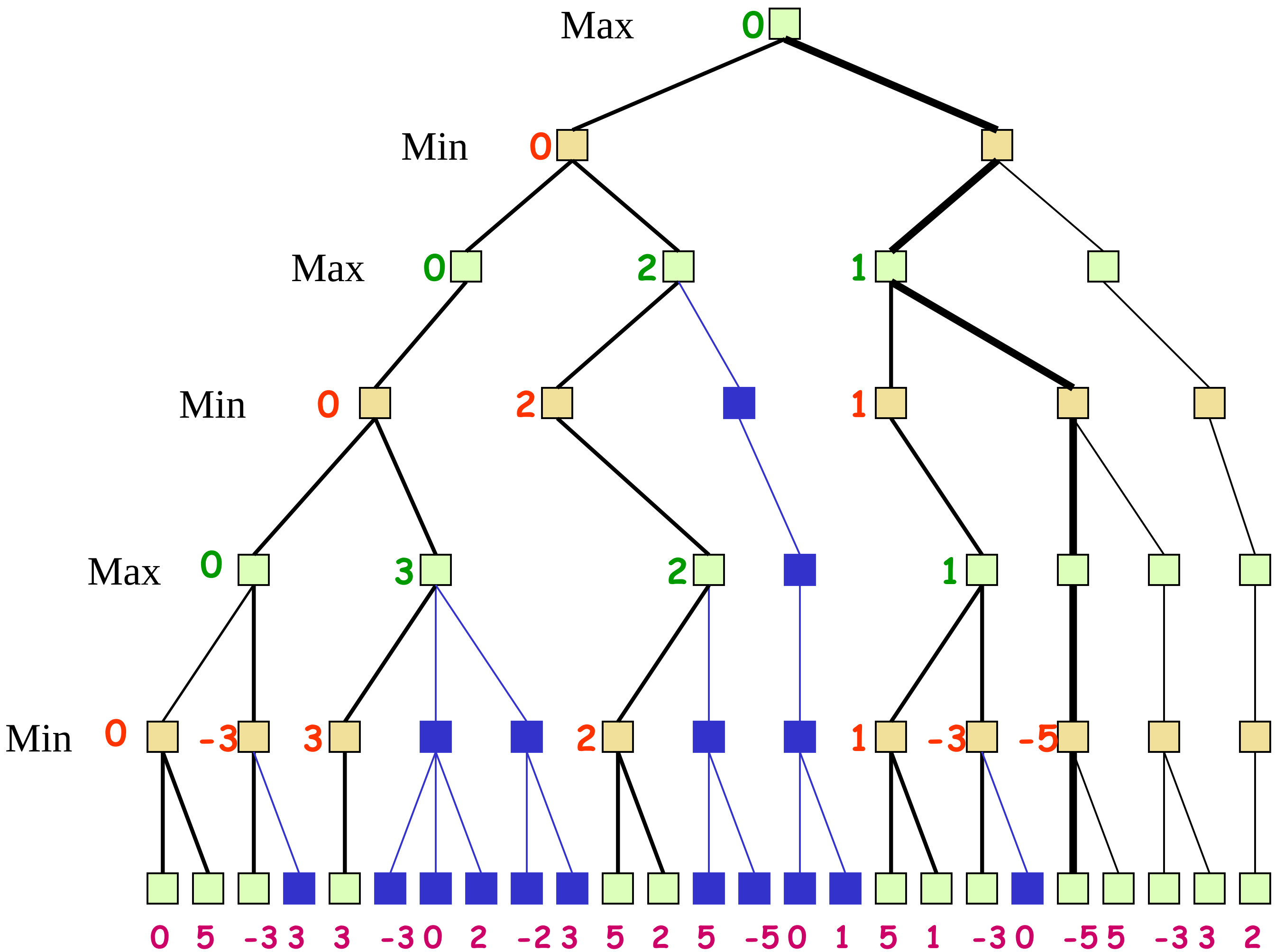
Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
1
Min
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



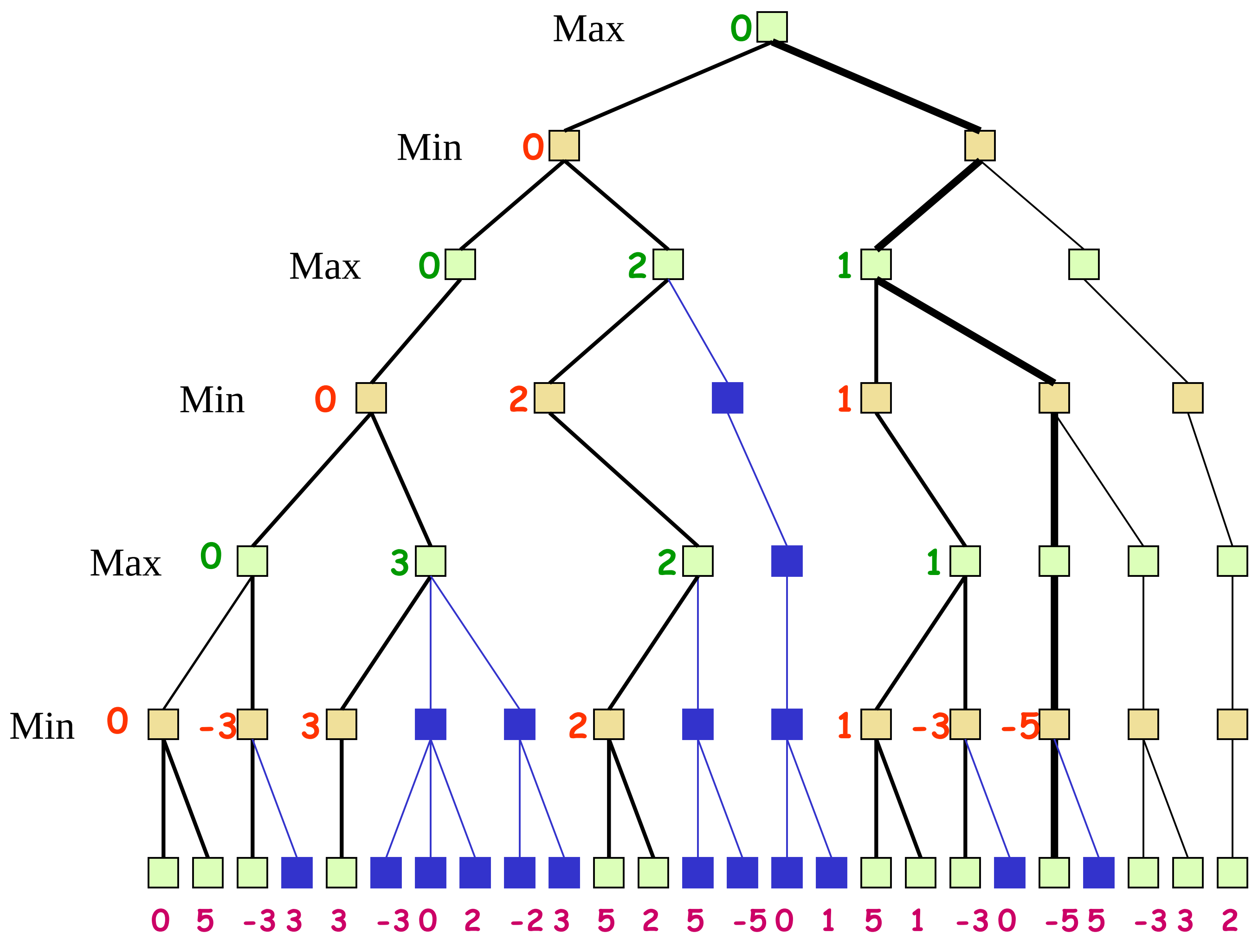
Max
Min
Max
Min
Max
Min
0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



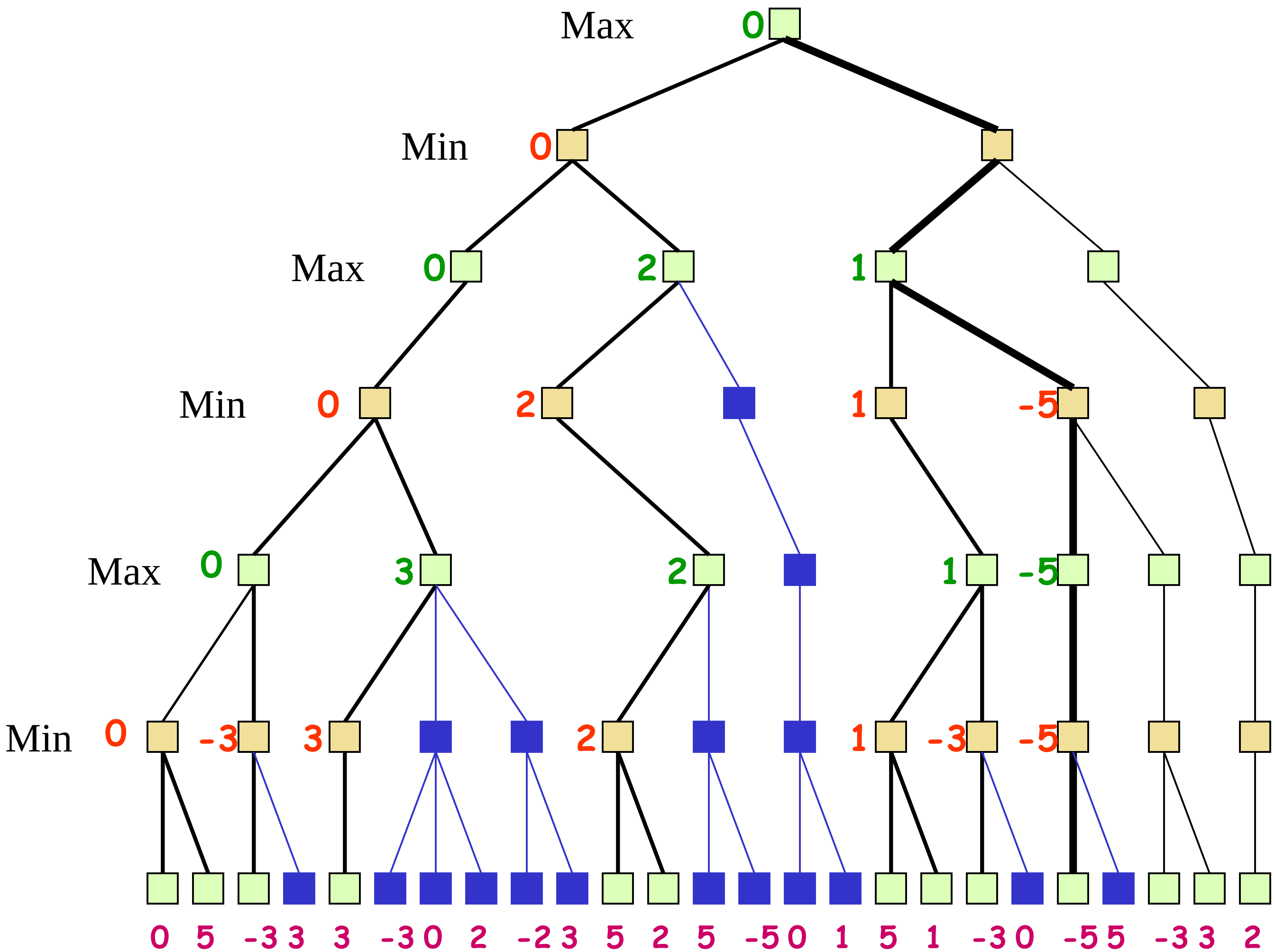
Max
Min
Max
Min
Max
Min
0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



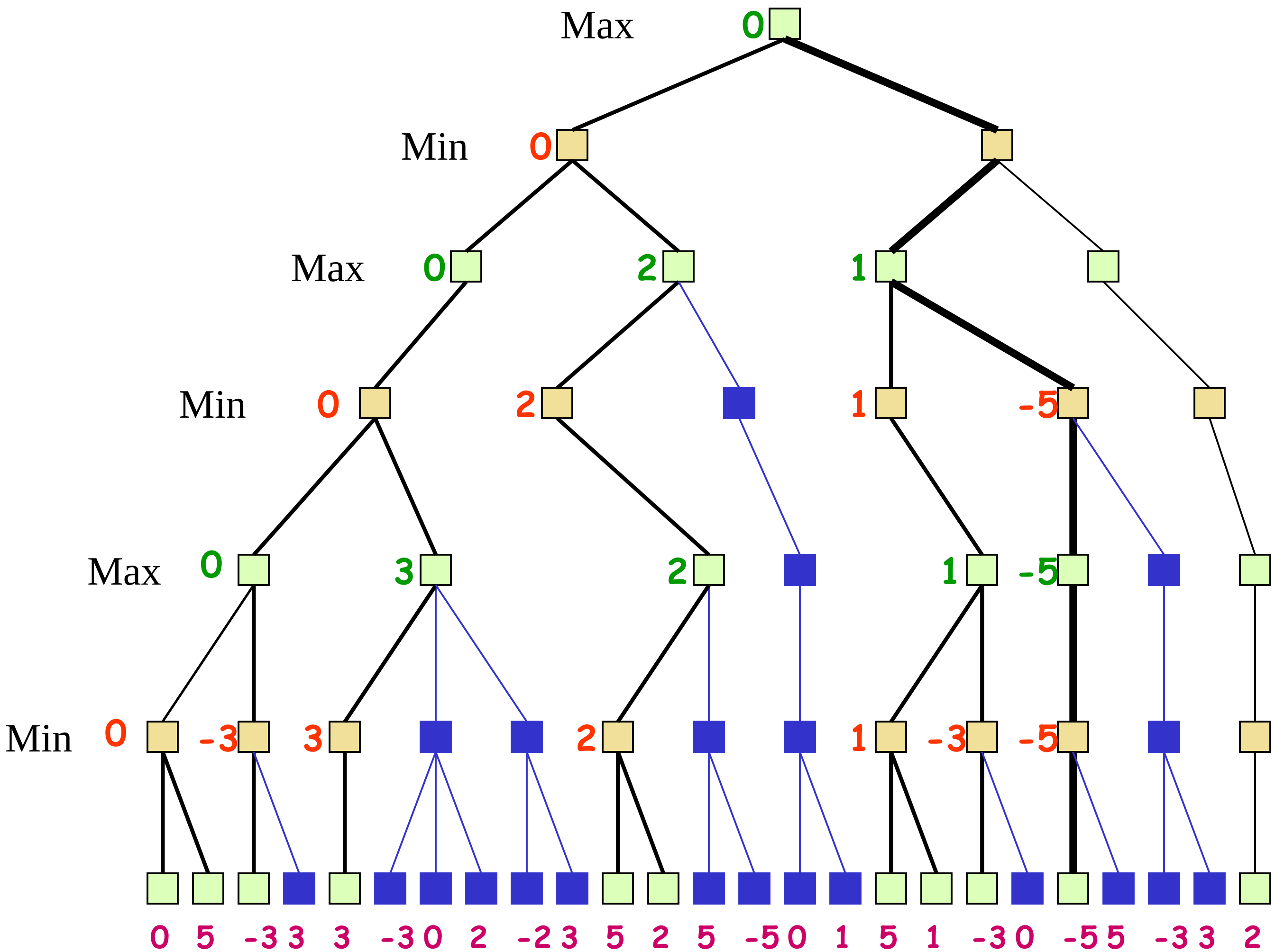
Max
0
Min
0
Max
0 2 1
Min
0 2 1
Max
0 3 2 1
Min
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

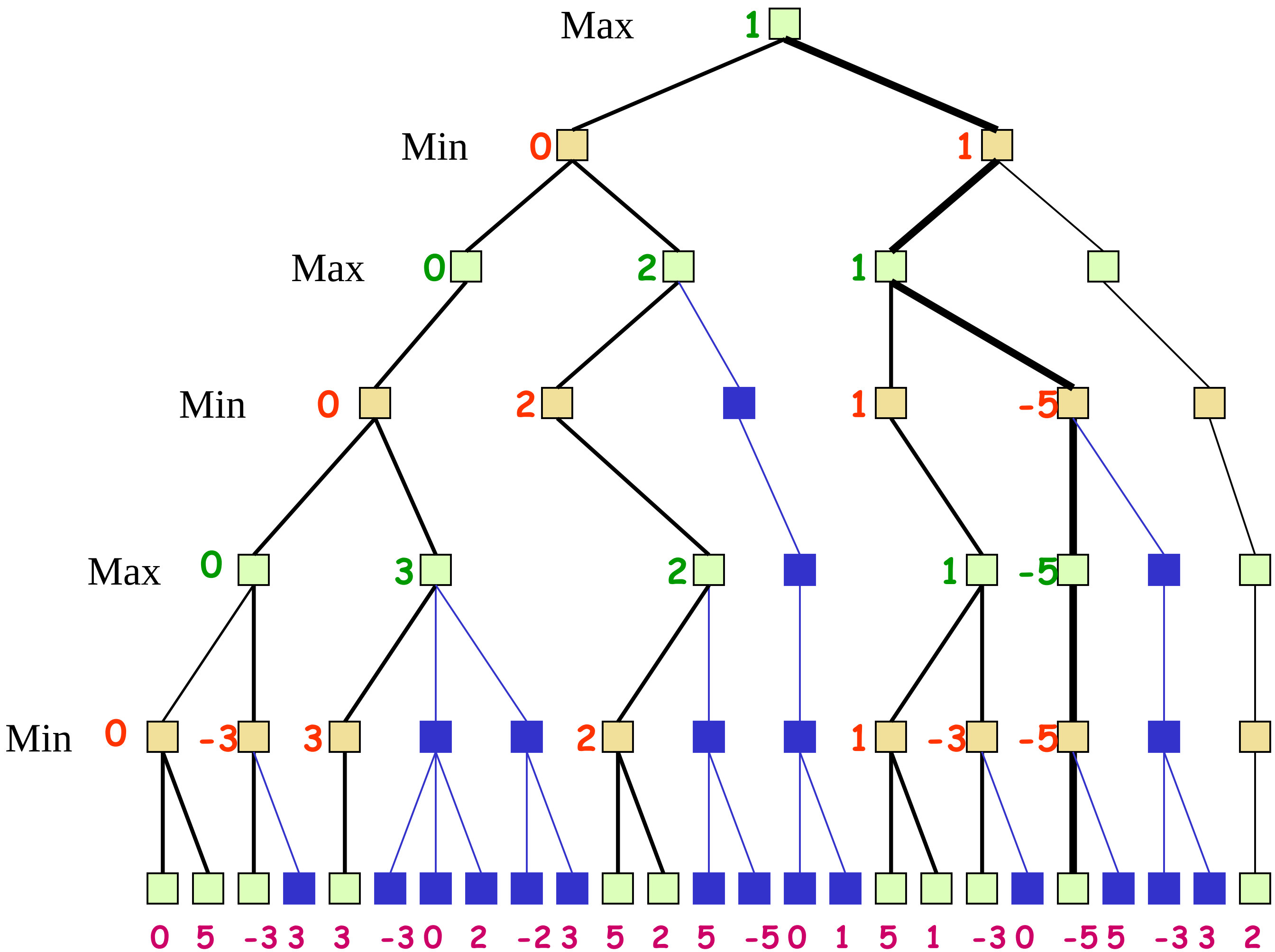


Max
Min
Max
Min
Max
Min
0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

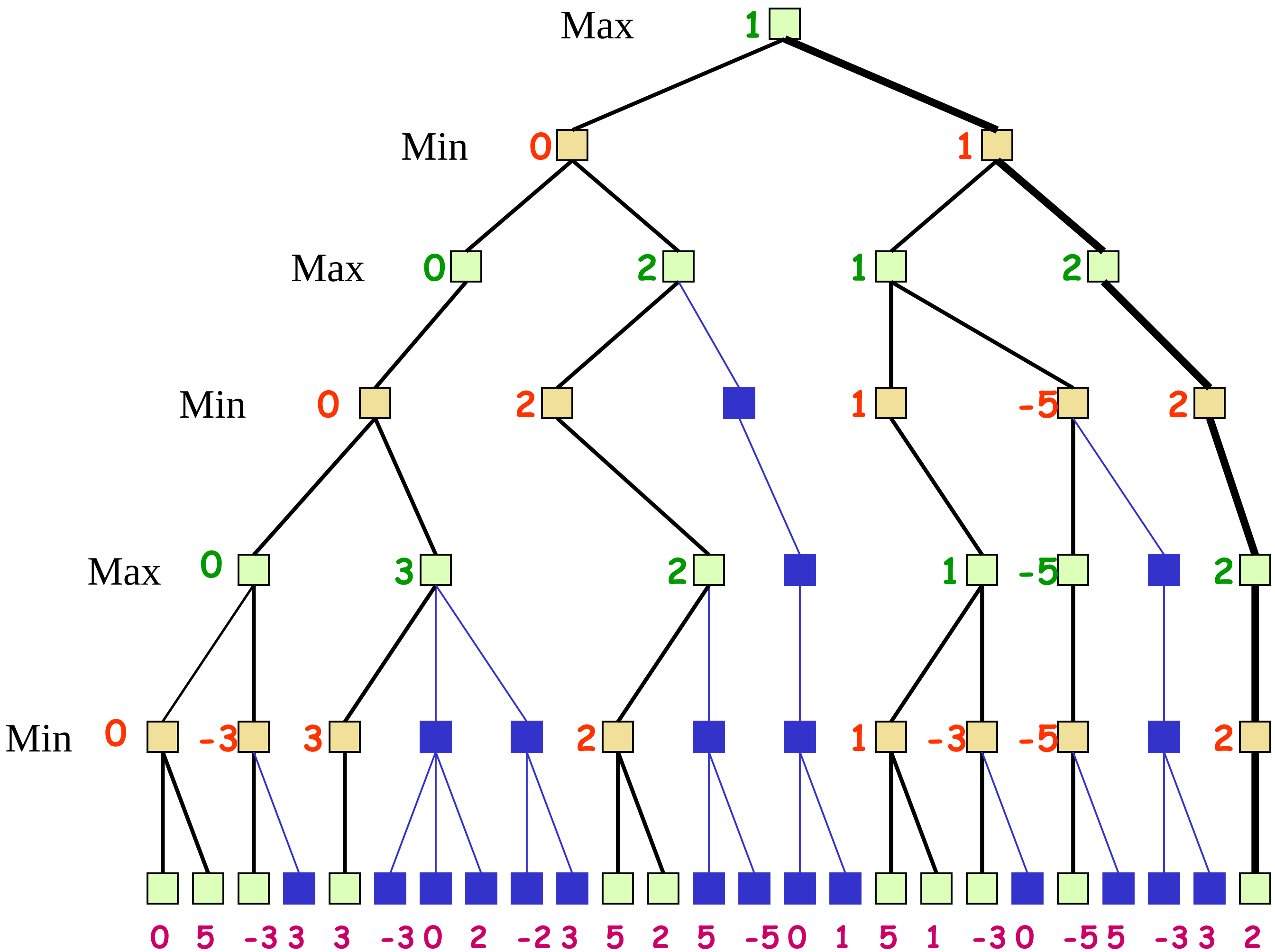


Max
Min
Max
Min
Max
Min
0
0
0 2 1
0 2 1 -5
0 3 2 1 -5
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

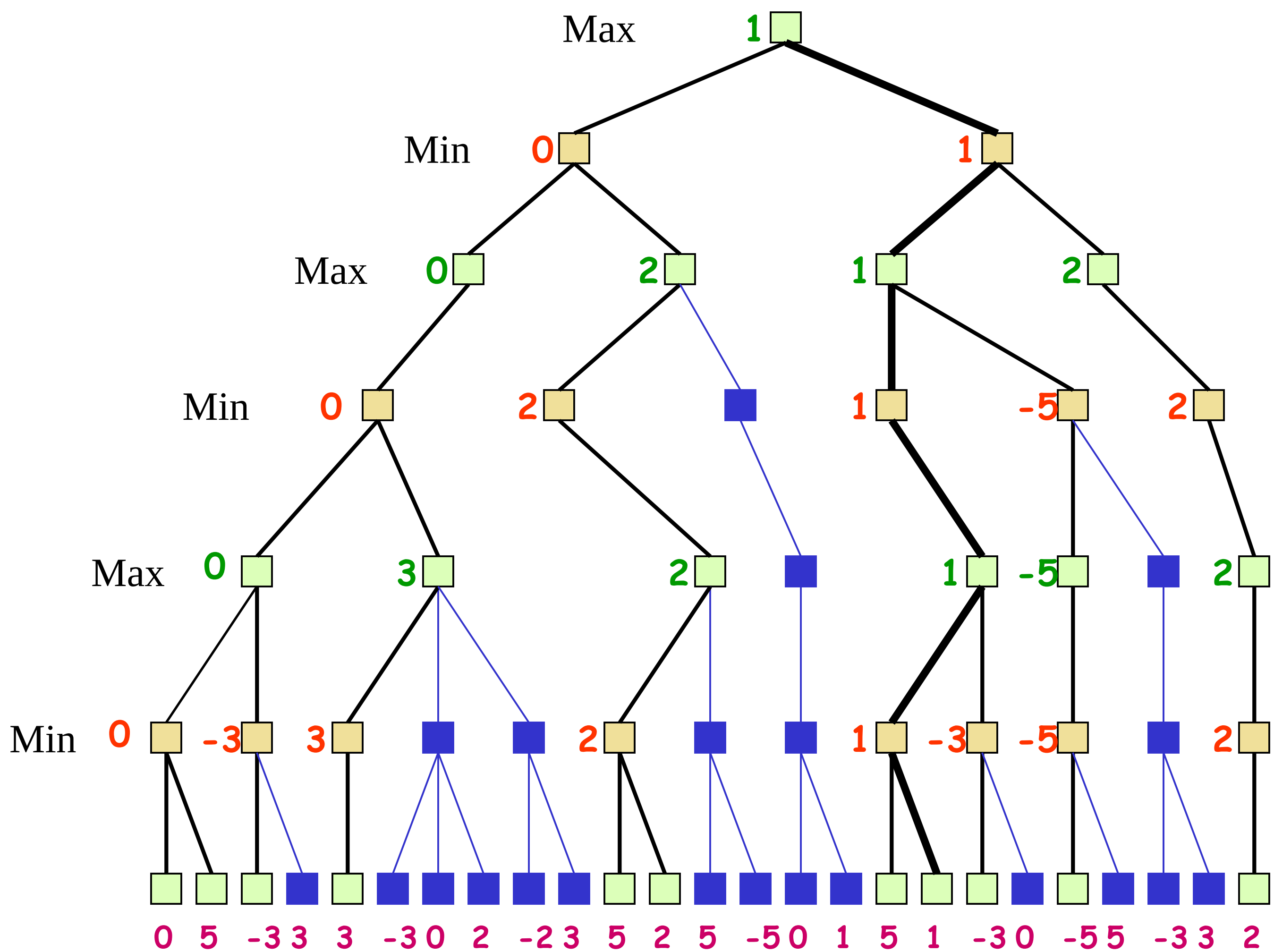

Max
Min
Max
Min
Max
Min
0
0
0
2
1
0
2
1
-5
0
3
2
1
-5
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



Max
Min
Max
Min
Max
Min
1
0
1
0
2
1
0
2
1
-5
0
3
2
1
-5
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



Max
Min
Max
Min
Max
Min
1
0
1
0
2
1
2
0
2
1
-5
2
0
3
2
1
-5
2
0
-3
3
2
1
-3
-5
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



Max
Min
Max
Min
Max
Min
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

α-β Pruning

Pruning by these cuts does not affect final result

– May allow you to go much deeper in tree

"Good" ordering of moves can make this pruning much more efficient

– Evaluating "best" branch first yields better likelihood of pruning later branches

– Perfect ordering reduces time to bm/2 instead of O(bd)

– i.e. doubles the depth you can search to!

α-β Pruning

Can store information along an entire *path*, not just at most recent levels!

Keep along the path:

α: best MAX value found on this path

(initialize to most negative utility value)

β: best MIN value found on this path

(initialize to most positive utility value)

Pruning at MAX node

α is possibly updated by the MAX of successors evaluated so far

If the value that would be returned is ever > β, then stop work on this branch

If all children are evaluated without pruning, return the MAX of their values

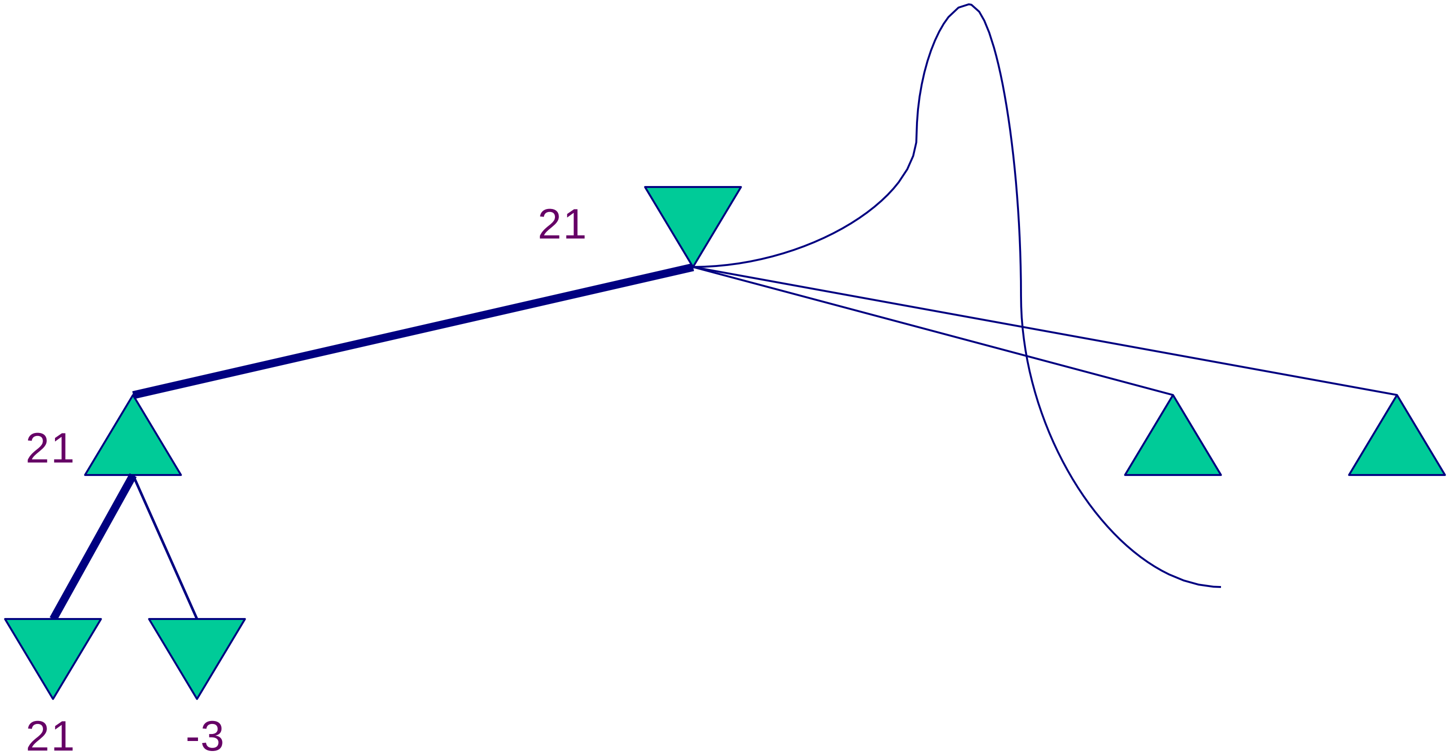
Pruning at MIN node

β is possibly updated by the MIN of successors evaluated so far

If the value that would be returned is ever < α, then stop work on this branch

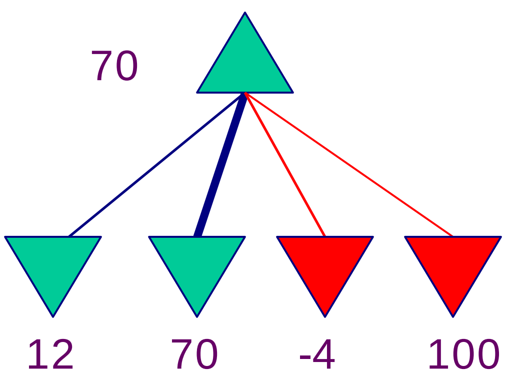
If all children are evaluated without pruning, return the MIN of their values

Idea of α-β Pruning



We know β on this path is 21

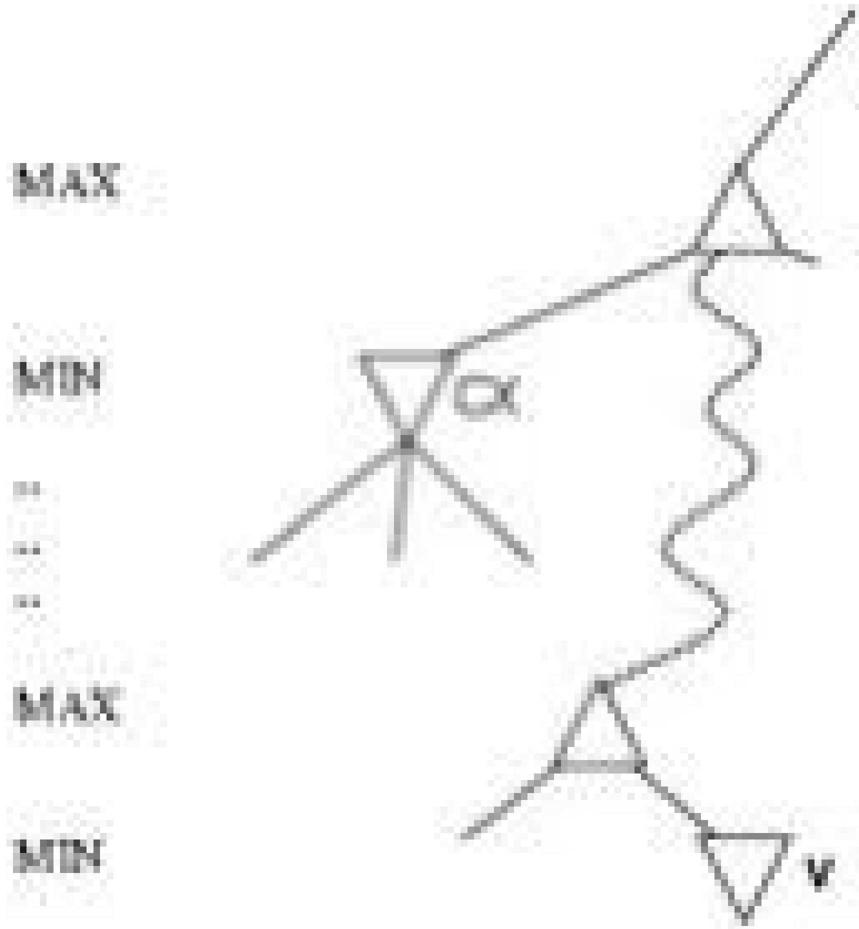
So, when we get max=70, we know this will never be used, so we can stop here



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*



Imperfect Decisions

Complete search is impractical for most games

Alternative: search the tree only to a certain depth

- Requires a cutoff-test to determine where to stop
  - Replaces the terminal test
  - The nodes at that level effectively become terminal leave nodes
- Uses a heuristics-based evaluation function to estimate the expected utility of the game from those leave nodes.

Utility Evaluation Function

Very game-specific

Take into account knowledge about game

"Stupid" utility

– 1 if player 1 wins

– -1 if player 0 wins

– 0 if tie (or unknown)

– Only works if we can evaluate complete tree

– But, should form a basis for other evaluations

Utility Evaluation

Need to assign a numerical value to the state

- Could assign a more complex utility value, but then the min/max determination becomes trickier.

Typically assign numerical values to lots of individual factors:

- a = # player 1's pieces - # player 2's pieces
- b = 1 if player 1 has queen and player 2 does not, -1 if the opposite, or 0 if the same
- c = 2 if player 1 has 2-rook advantage, 1 if a 1-rook advantage, etc.

Utility Evaluation

The individual factors are combined by some function

Usually a linear weighted combination is used:

- $u = \alpha a + \beta b + \chi c$
- Different ways to combine are also possible

Notice: quality of utility function is based on:

- What features are evaluated
- How those features are scored
- How the scores are weighted/combined

Absolute utility value doesn't matter – relative value does.

Evaluation Functions

If you had a perfect utility evaluation function, what would it mean about the minimax tree?

You would never have to evaluate more than one level deep!

Typically, you can't create such perfect utility evaluations, though.

Evaluation Functions for Ordering

As mentioned earlier, order of branch evaluation can make a big difference in how well you can prune

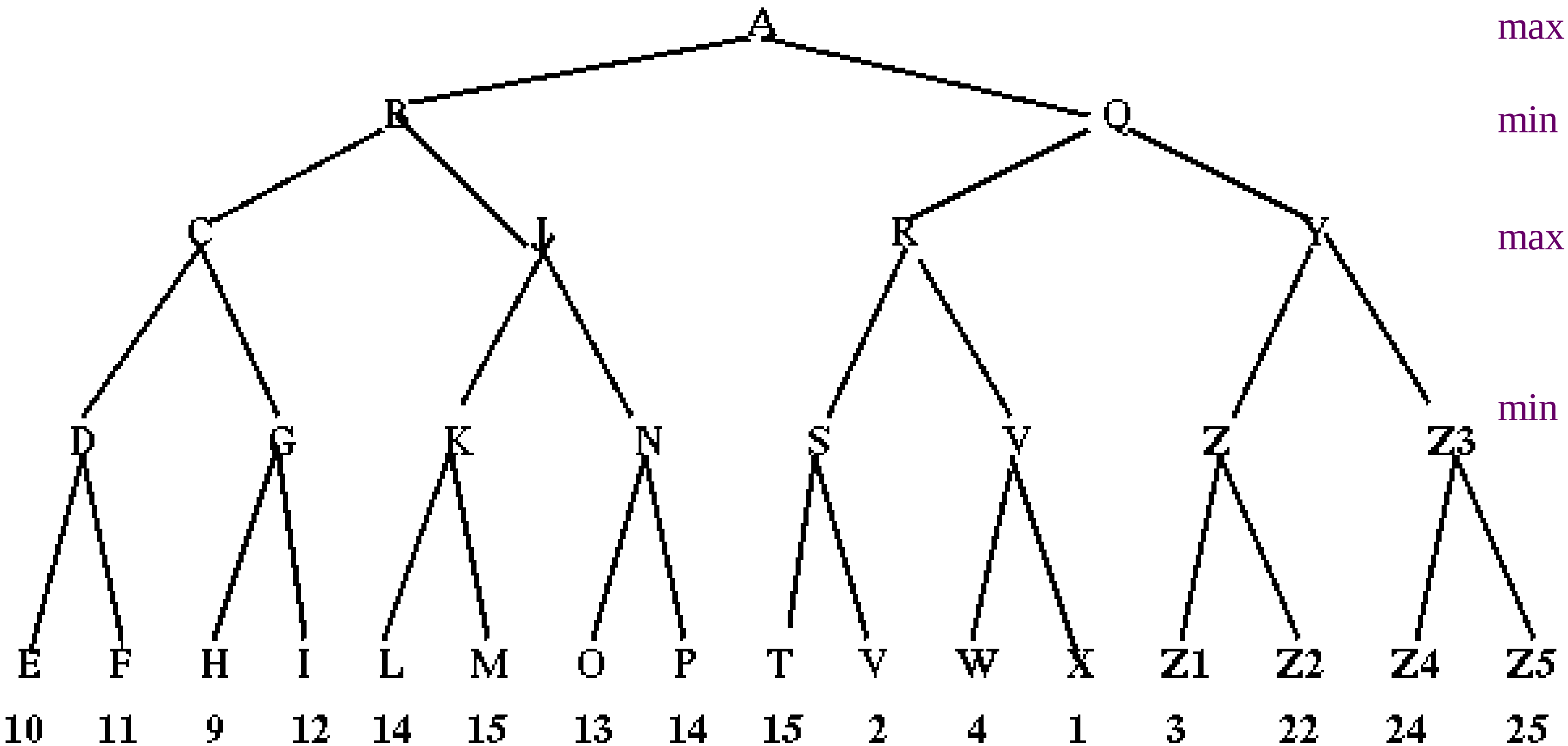
A good evaluation function might help you order your available moves:

- Perform one move only
- Evaluate board at that level
- Recursively evaluate branches in order from best first move to worst first move (or vice-versa if at a MIN node)

The following are extra Examples

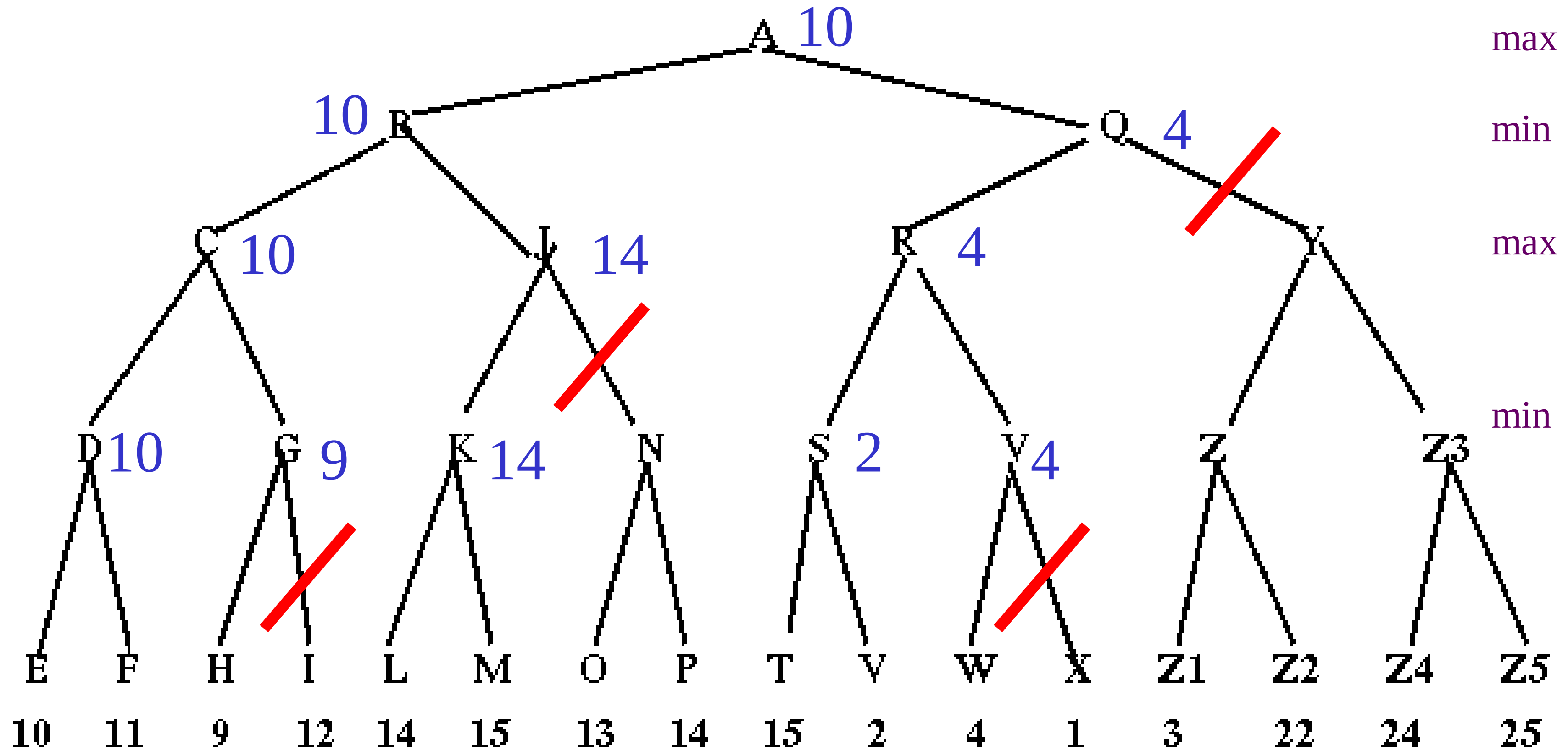
(Self Study)

Exercise



A
B
Q
C
J
R
Y
D
G
K
N
S
V
Z
Z3
E
F
H
I
L
M
O
P
T
V
W
X
Z1
Z2
Z4
Z5
10
11
9
12
14
15
13
14
15
2
4
1
3
22
24
25
max
min
max
min

Exercise (Solution)



Checkers Case Study

- Initial board configuration
  - Black single on 20
    single on 21
    king on 31
  - Red single on 23
    king on 22
  - Evaluation function

    $E(s) = (5\,x_1 + x_2) - (5r_1 + r_2)$

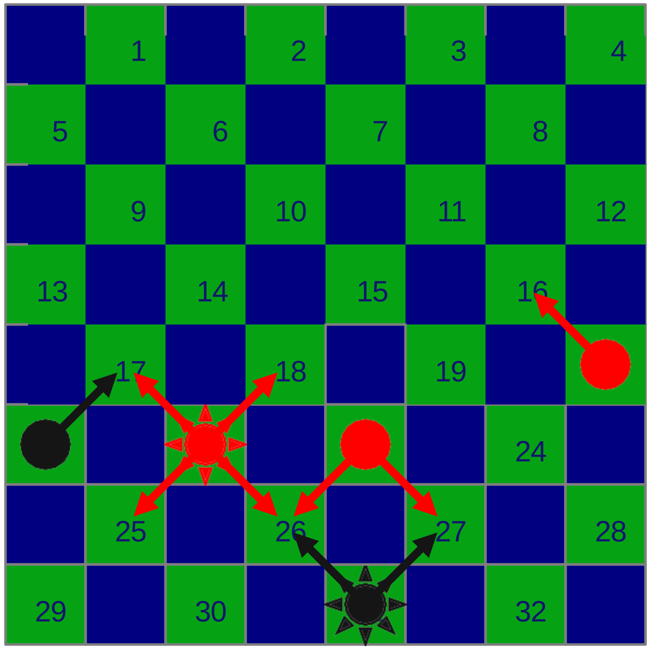
    where

    x_1 = black king advantage,

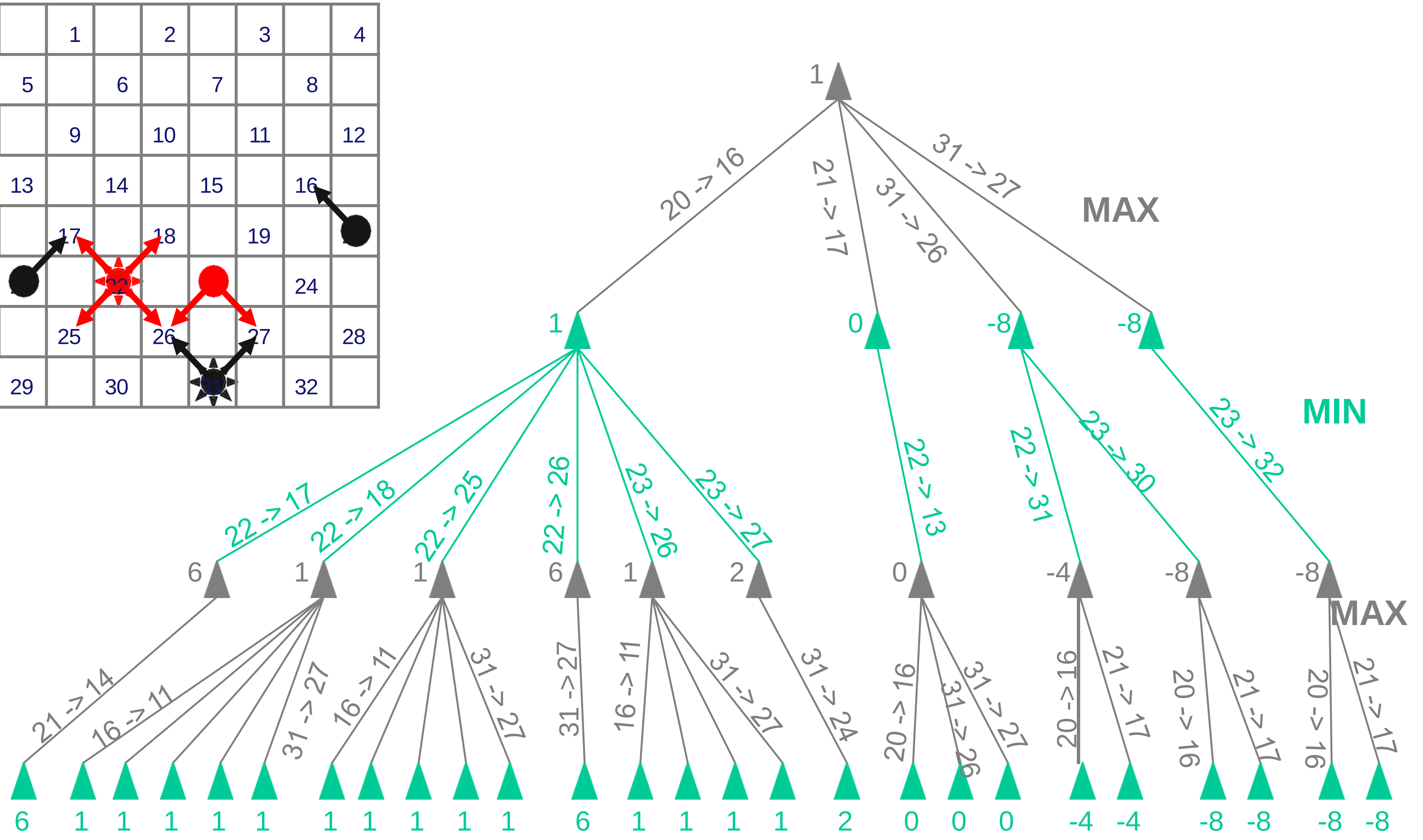
    x_2 = black single advantage,

    r_1 = red king advantage,

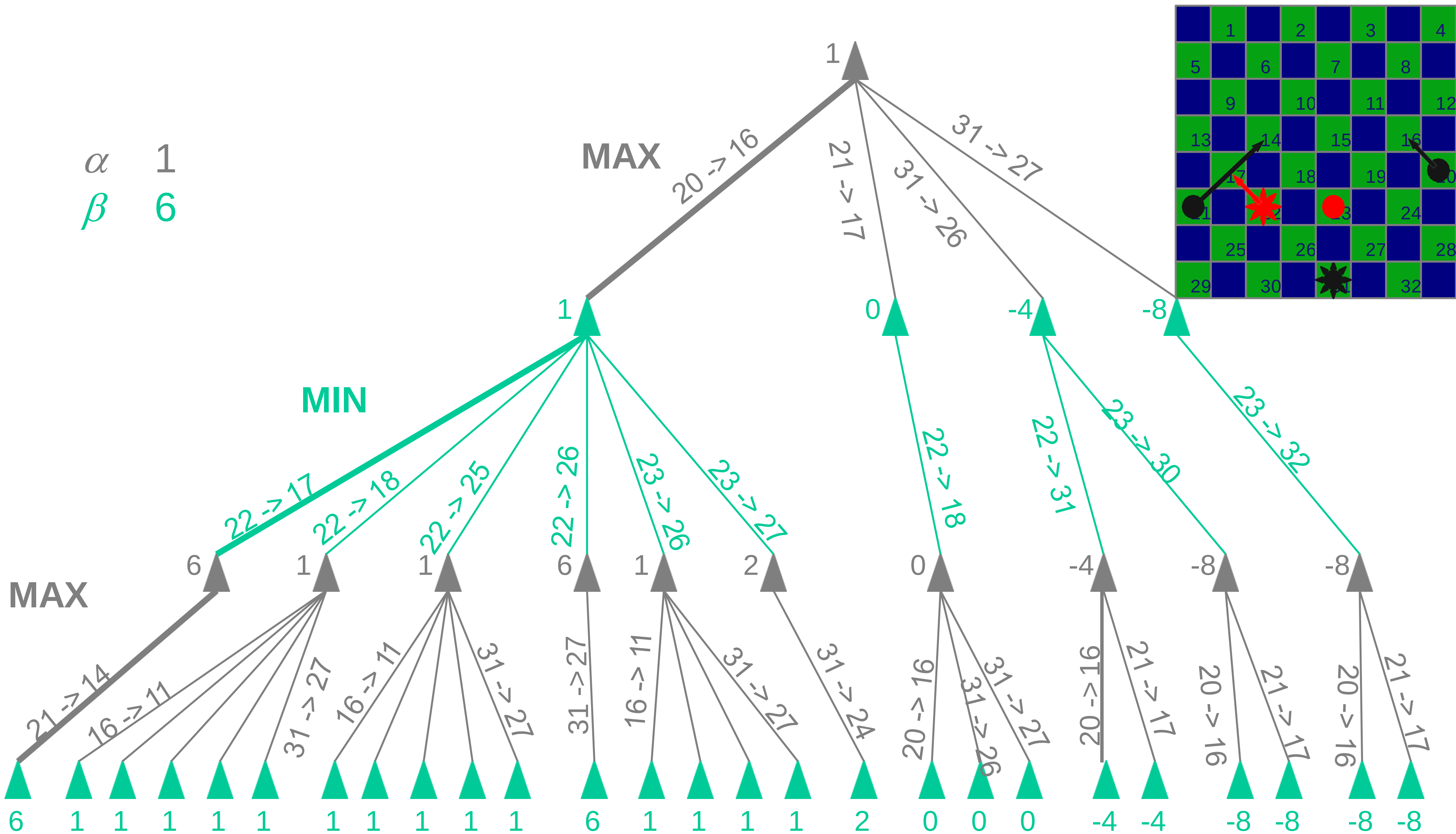
    r_2 = red single advantage



Checkers MiniMax Example



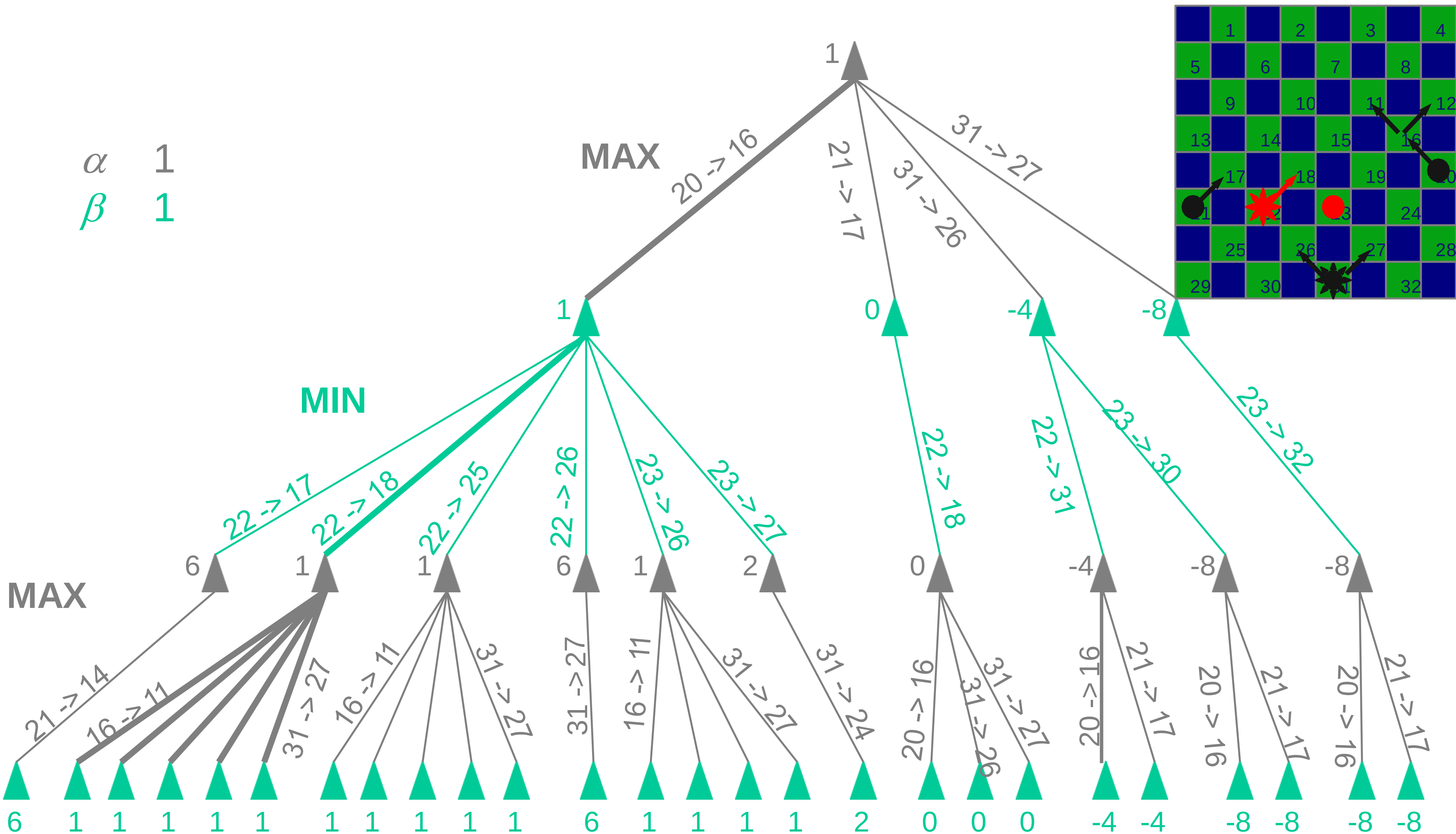
Checkers Alpha-Beta Example



Checkers Alpha-Beta Example

α 1

β 1

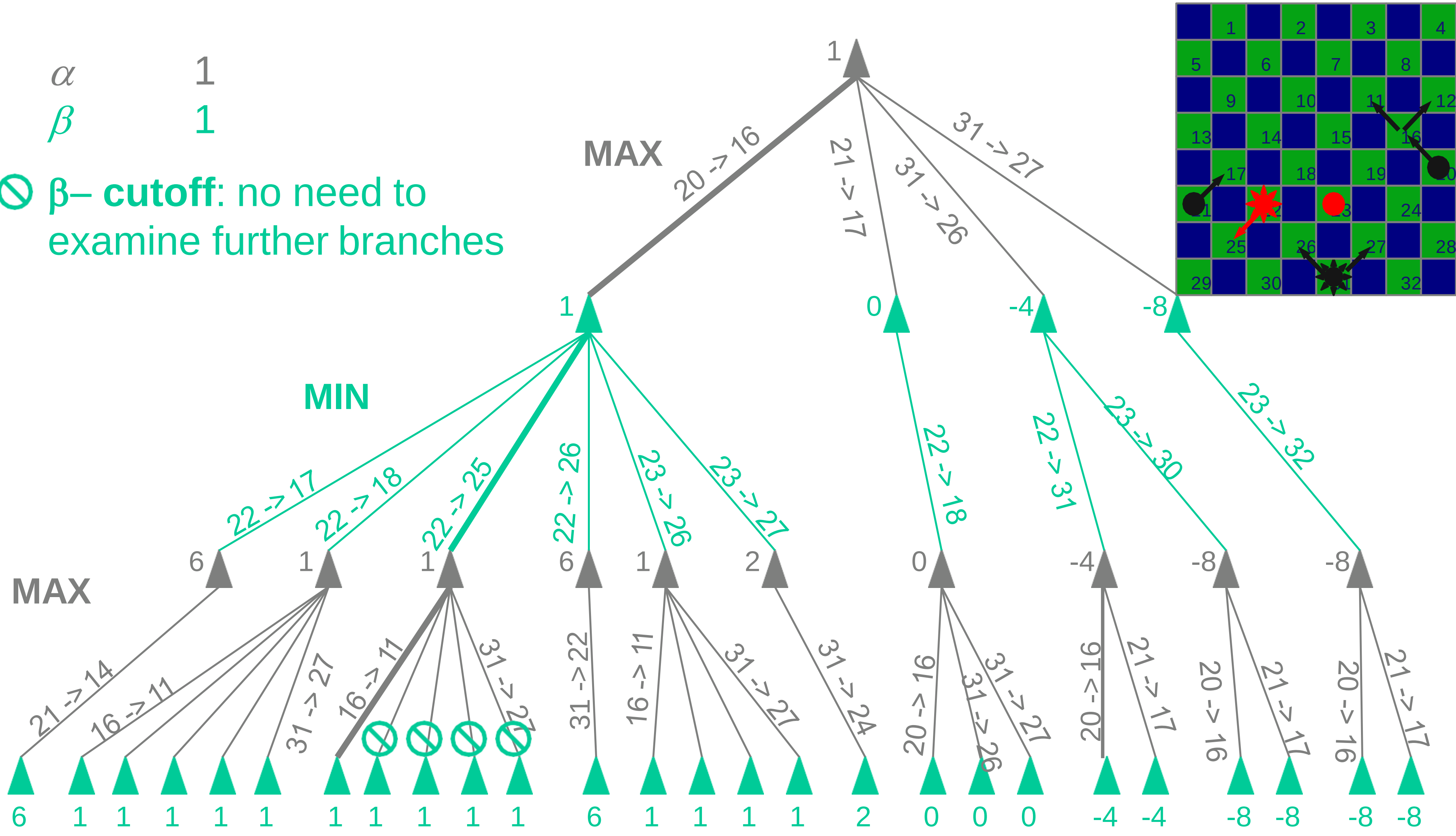


Checkers Alpha-Beta Example

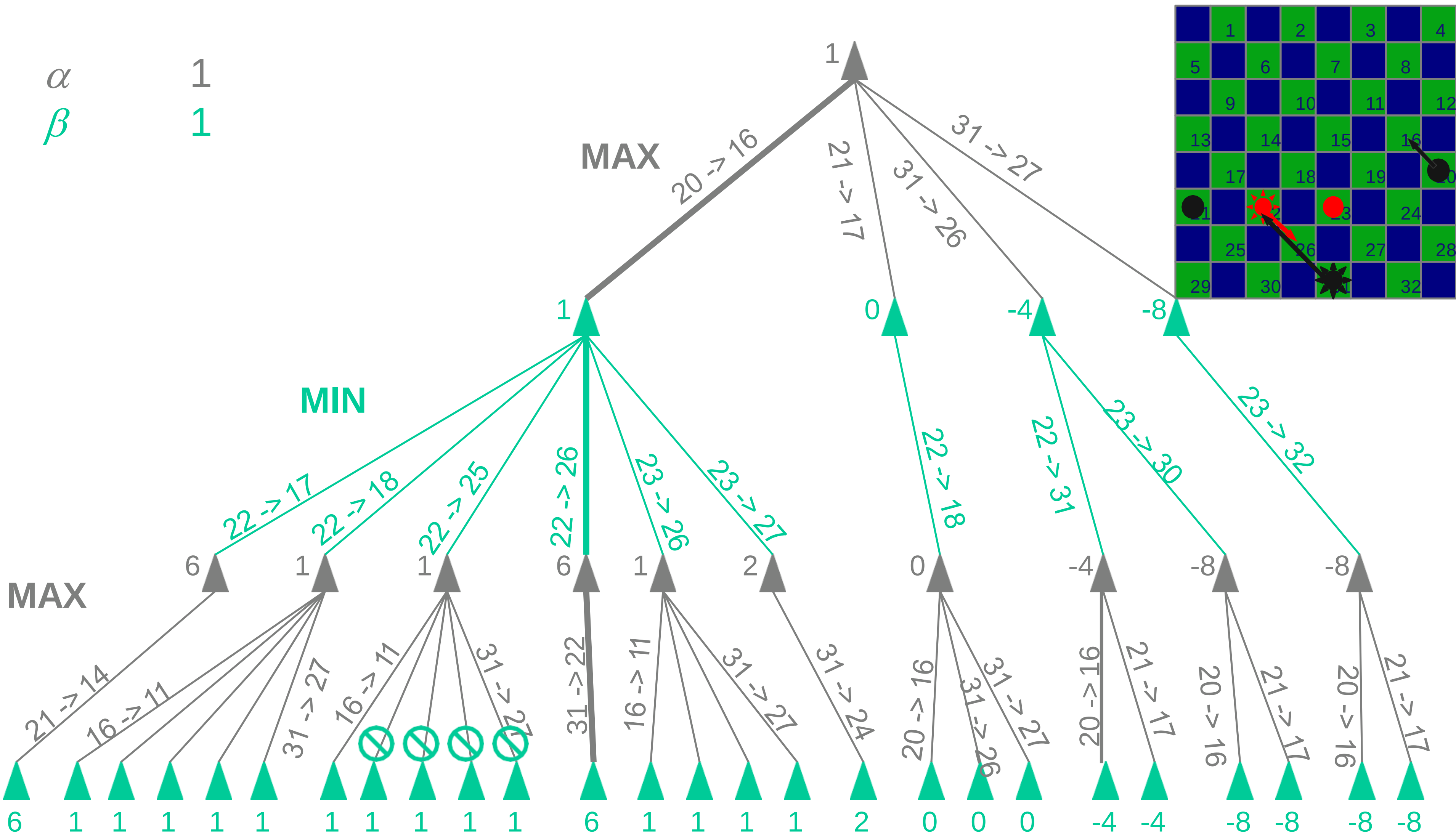
α 1

β 1

β– **cutoff**: no need to examine further branches



Checkers Alpha-Beta Example

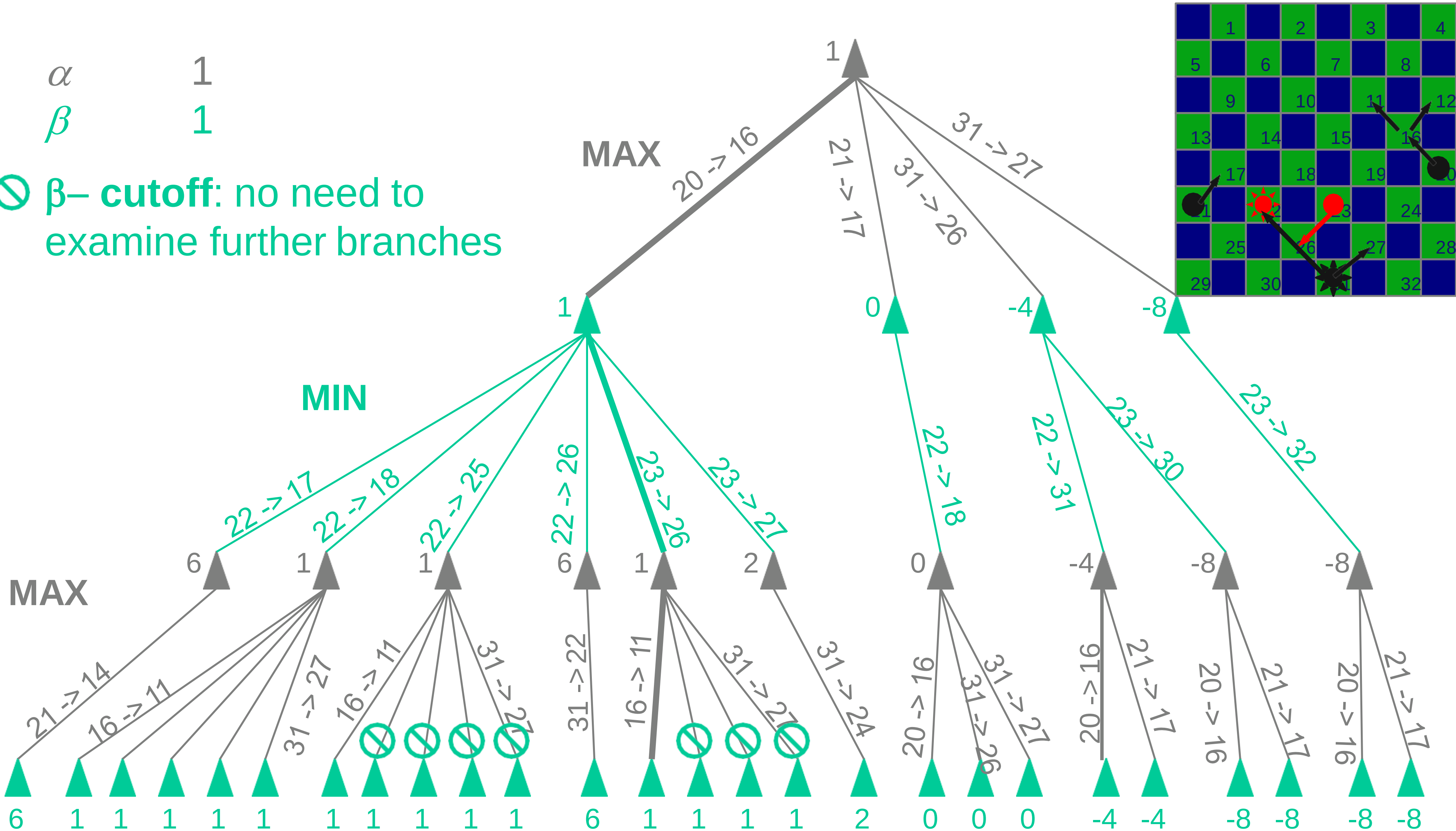


Checkers Alpha-Beta Example

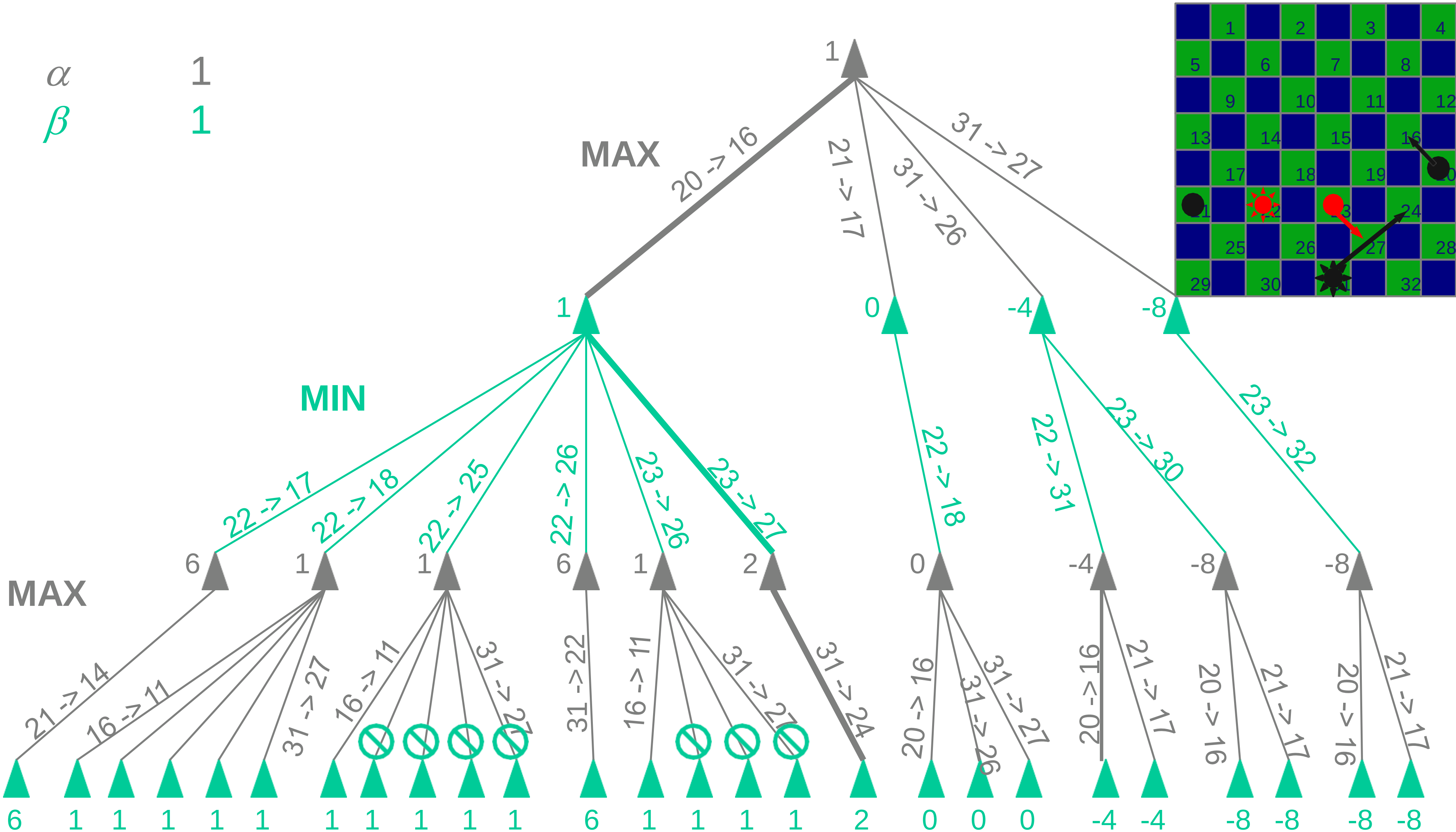
α 1

β 1

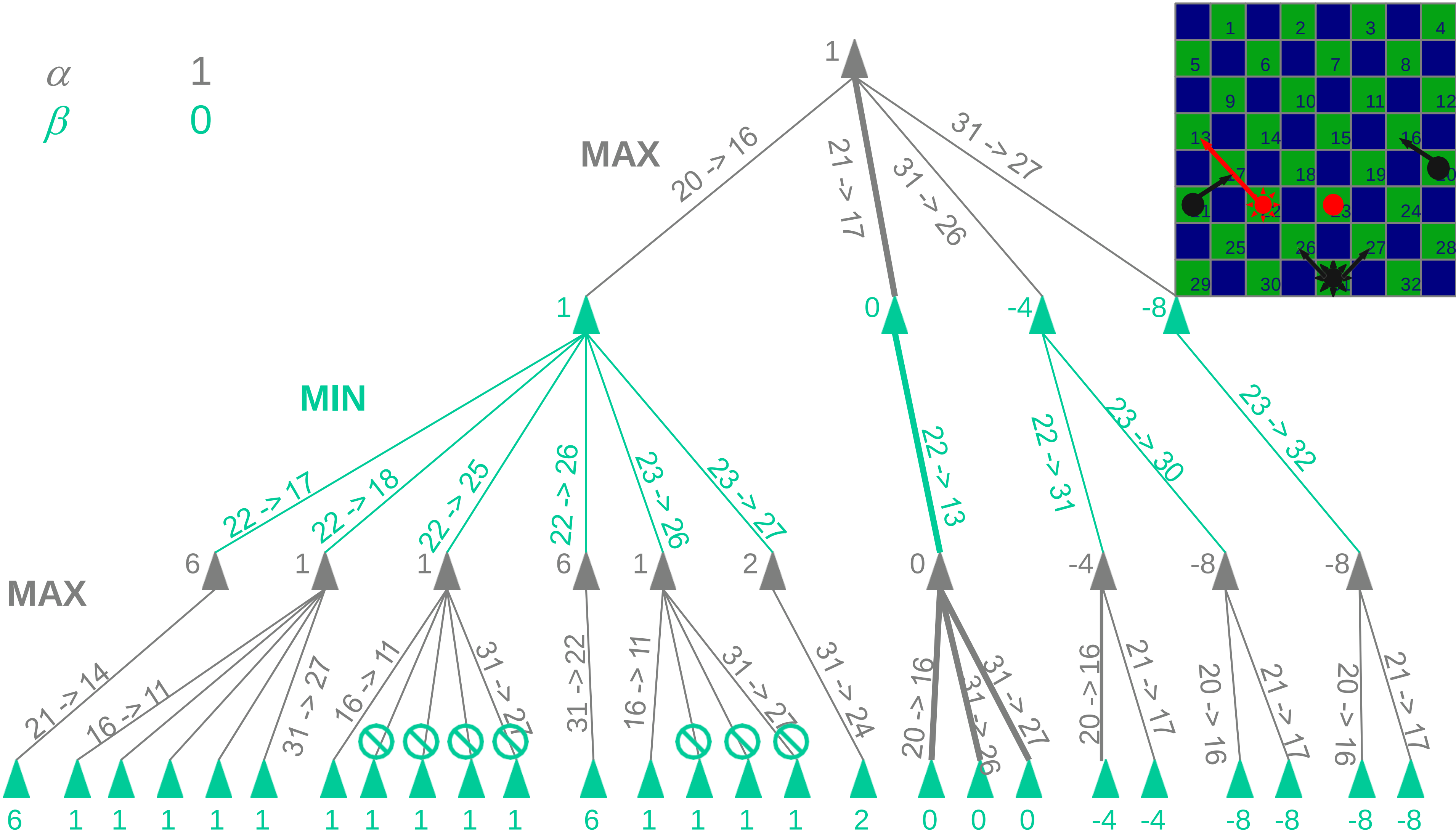
β– cutoff: no need to examine further branches



Checkers Alpha-Beta Example



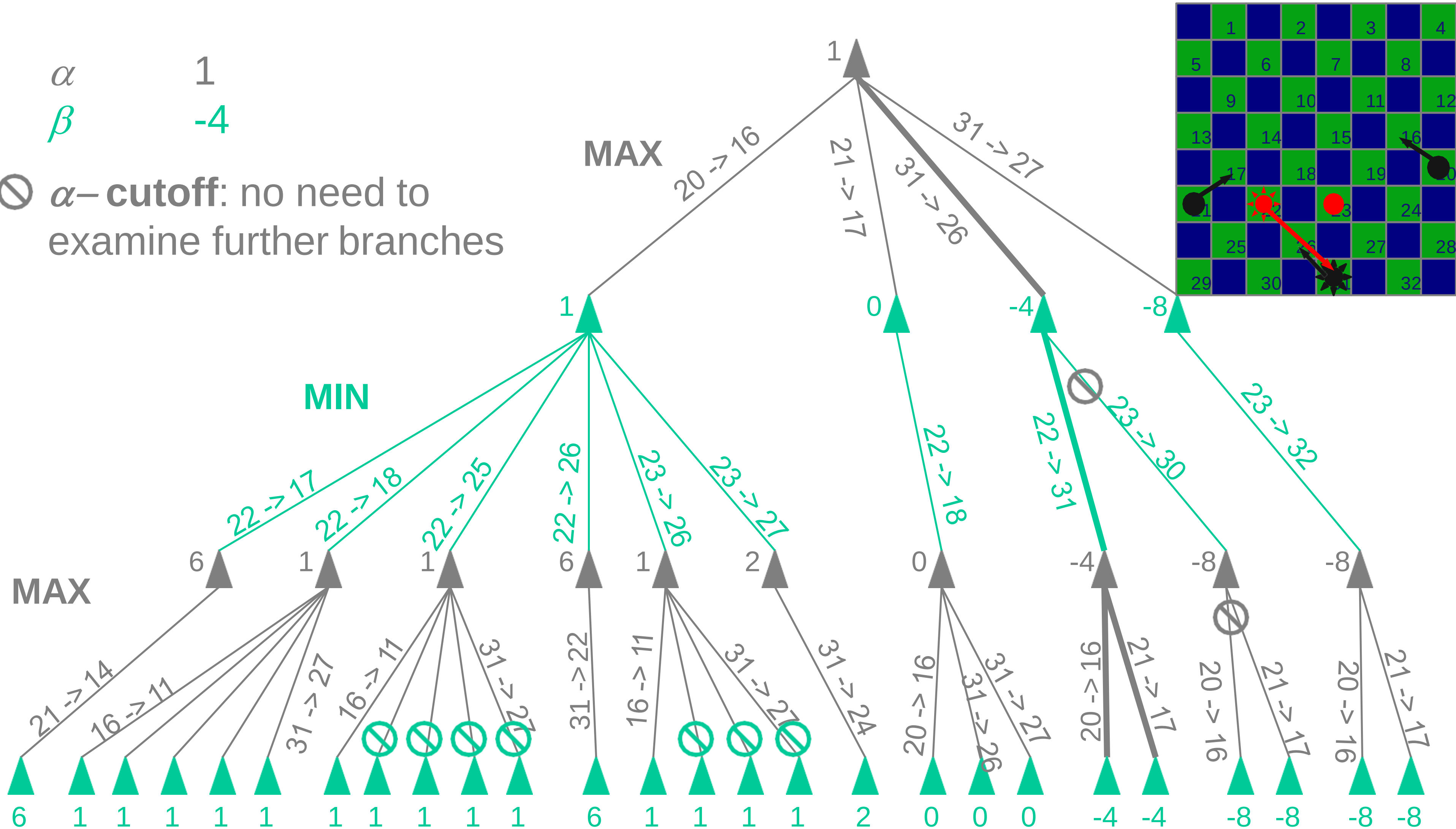
Checkers Alpha-Beta Example



Checkers Alpha-Beta Example

α 1
β -4

$\alpha-$ **cutoff**: no need to examine further branches



Checkers Alpha-Beta Example