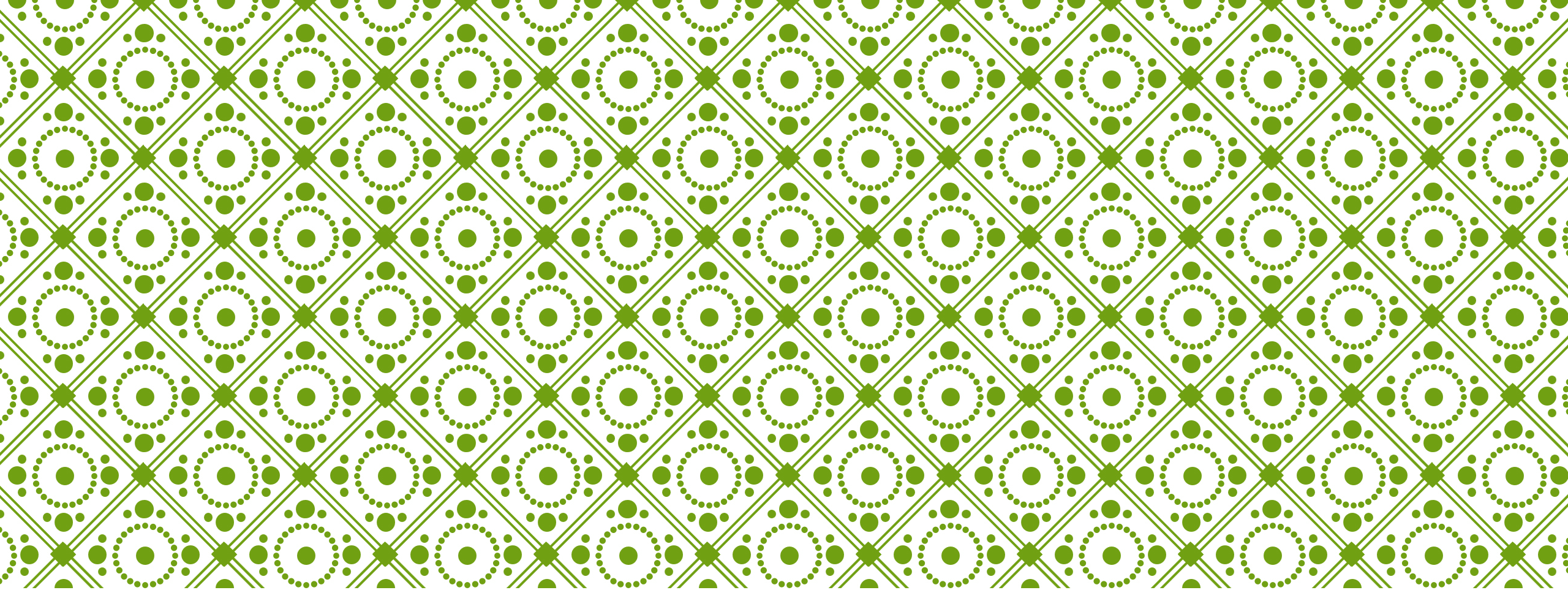
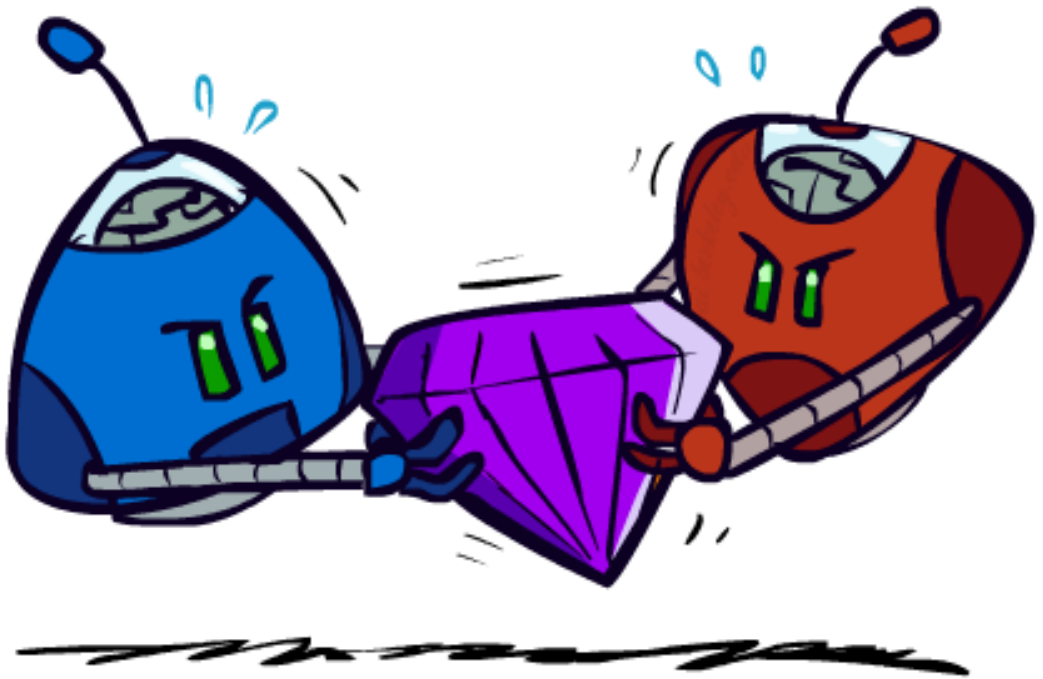


AI2002 – ARTIFICIAL INTELLIGENCE SPRING 2024 - LECTURE 19-21

Presented By: Mr. Sandesh Kumar

Adversarial Games

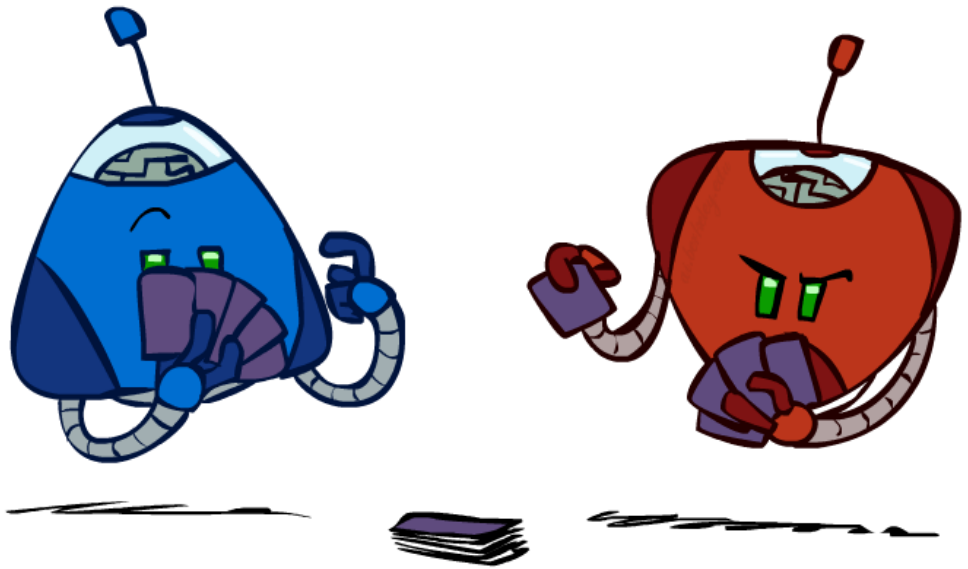


Games

- In **multiagent environments**, each agent needs to consider the actions of other agents and how they affect its own welfare.
- The agents can be **cooperative** or **competitive**.
- In **competitive environments**, the agents' goals are in conflict, giving rise to **Adversarial Search problems** — often known as **GAMES**.
- In mathematical **game theory**, a multiagent environment is treated as a **game**, the impact of each agent (economy) on the others is significant, regardless of whether the agents are ***cooperative*** or ***competitive***.
- In **AI**, the most common games are *deterministic*, *turn-taking*, *two-player*, **zero-sum games** of **perfect information** (such as chess).
  - In deterministic and fully observable environments in two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
  - If one player wins a game of chess, the other player necessarily loses.
  - It is this opposition between the agents'**utility functions** that makes the situation **adversarial**.

Types of Games

- Game = task environment with > 1 agent
- Axes:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - Two, three, or more players?
  - Teams or individuals?
  - Turn-taking or simultaneous?
  - Zero sum?



- Want algorithms for calculating a **strategy** (policy) which recommends a move from every possible state

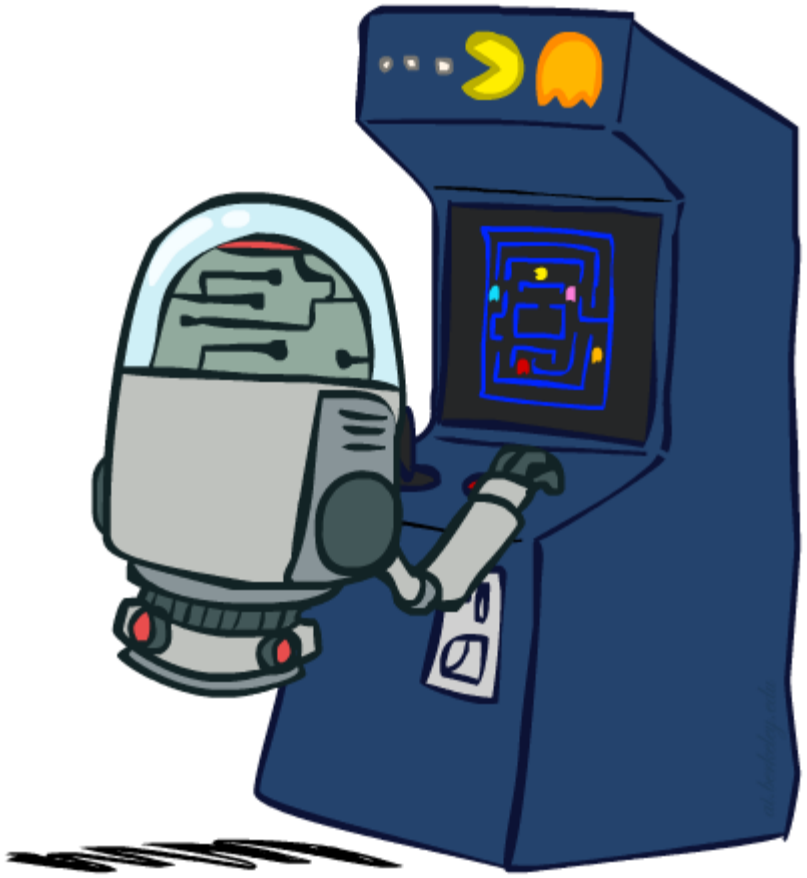
Games

- Games are too hard to solve.
  - Chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} nodes.
- Games require the ***ability to make decision even when calculating optimal decision is infeasible***.
- **Pruning** allows us to ignore portions of the search tree.
- **Heuristic evaluation functions** allow us to approximate **true utility of a state** without doing a complete search.
- Games such as backgammon includes elements of **imperfect information** because not all cards are visible to each player.
- Types of games

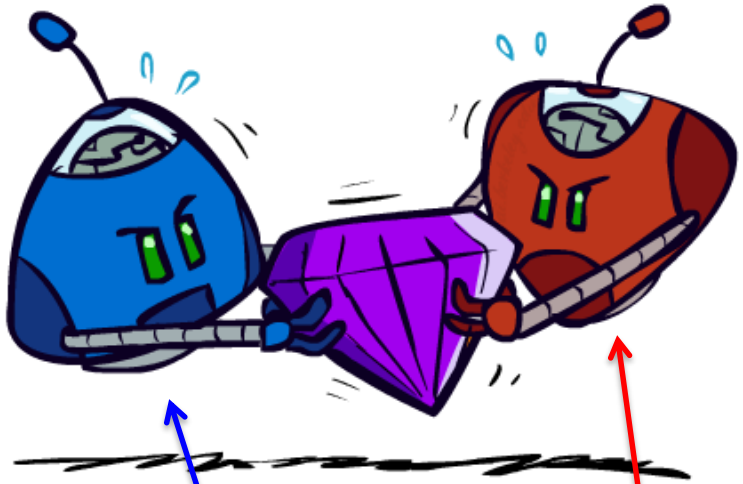
	deterministic	**chance**
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships,	bridge, poker, scrabble

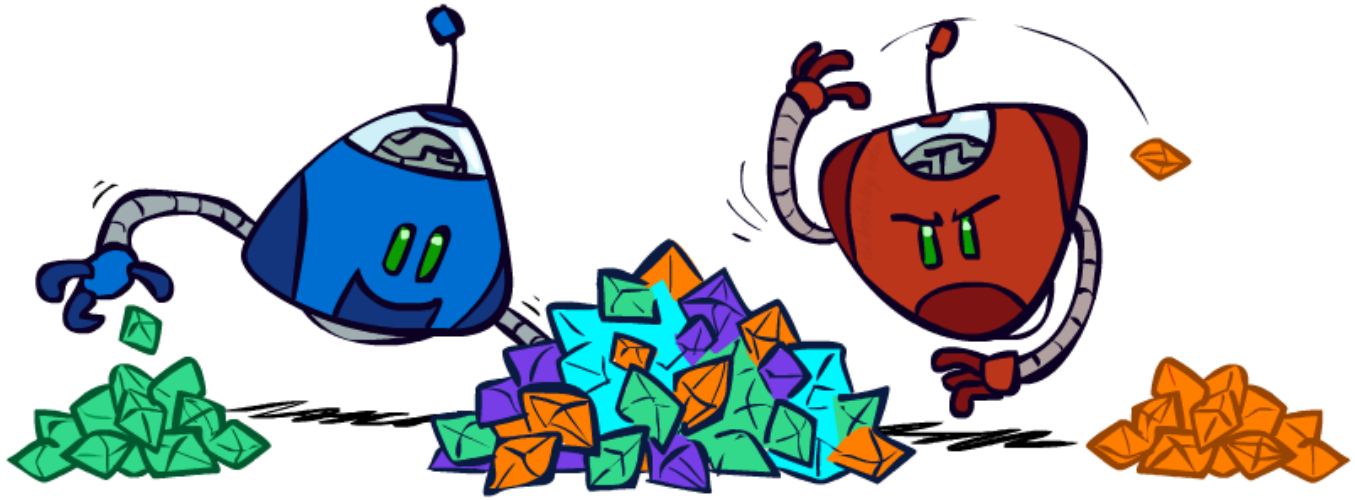
Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1…N} (usually take turns)
  - Actions: A (may depend on player/state)
  - Transition function: S x A → S
  - Terminal test: S → {true, false}
  - Terminal utilities: S x P → R
- Solution for a player is a policy: S → A



Zero-Sum Games

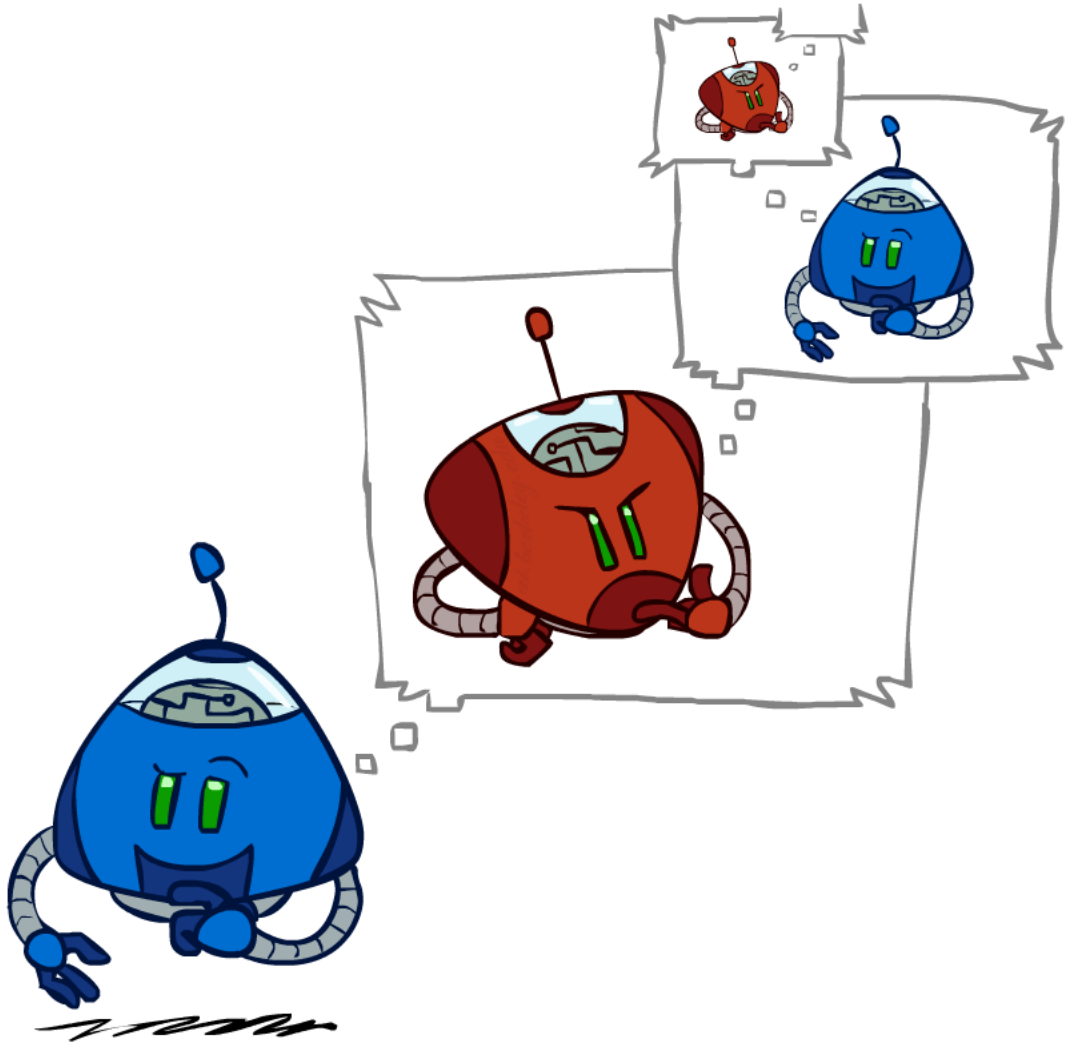




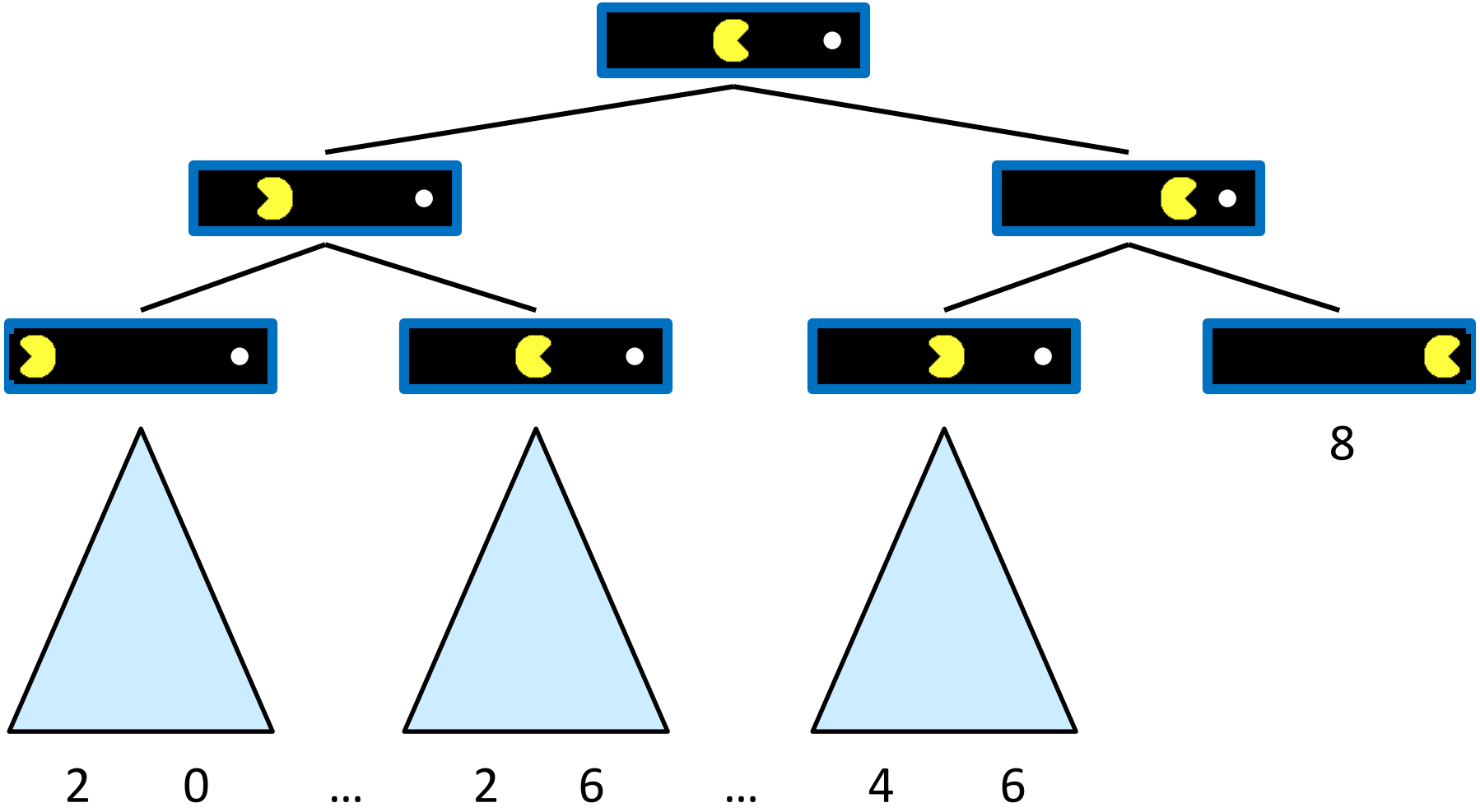
- Zero-Sum Games
  - Agents have ***opposite*** utilities
  - Pure competition:
    - One ***maximizes***, the other ***minimizes***

- General-Sum Games
  - Agents have ***independent*** utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible
- Team Games
  - Common payoff for all team members

Adversarial Search

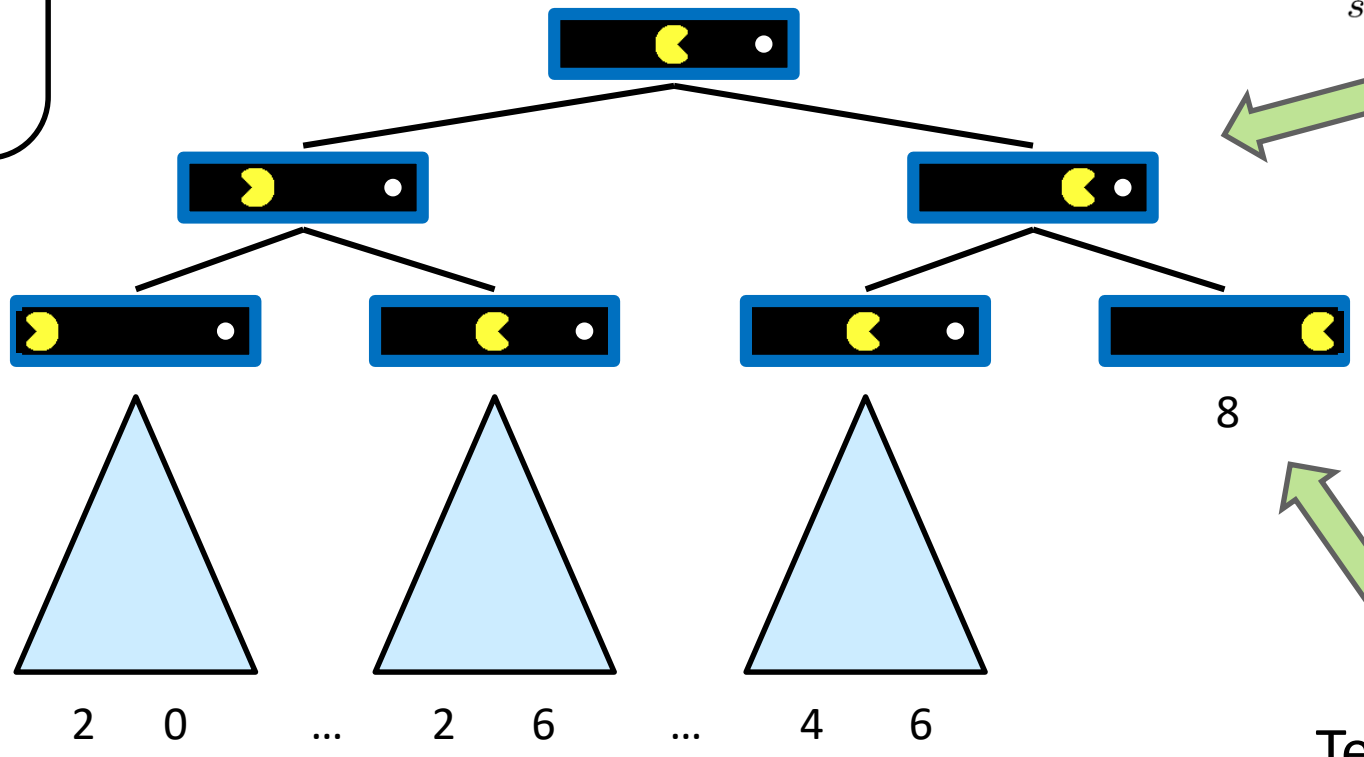


Single-Agent Trees



Value of a State

Value of a state:
The best achievable outcome (utility) from that state



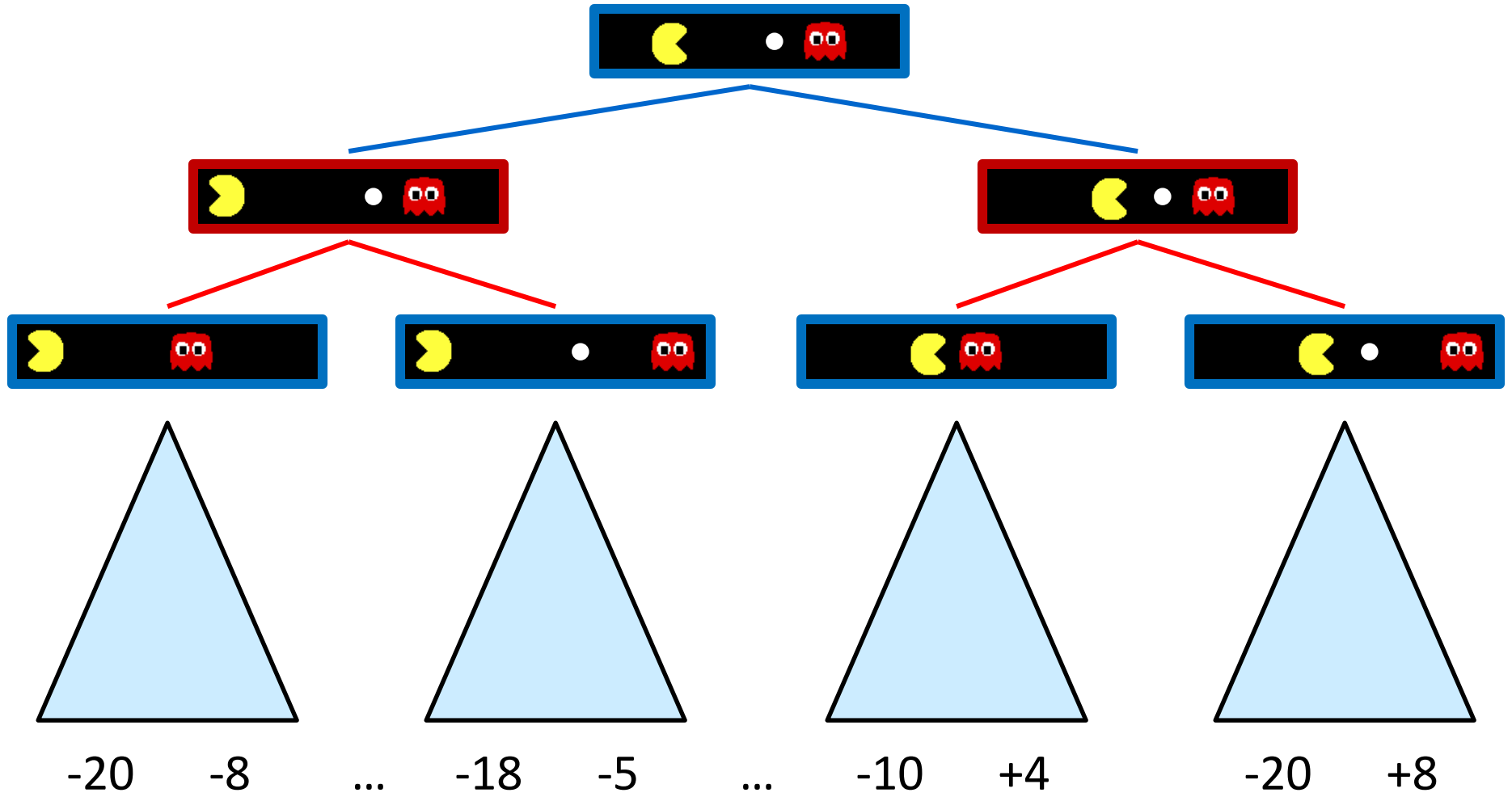
Non-Terminal States:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

$$V(s) = \text{known}$$

Adversarial Game Trees



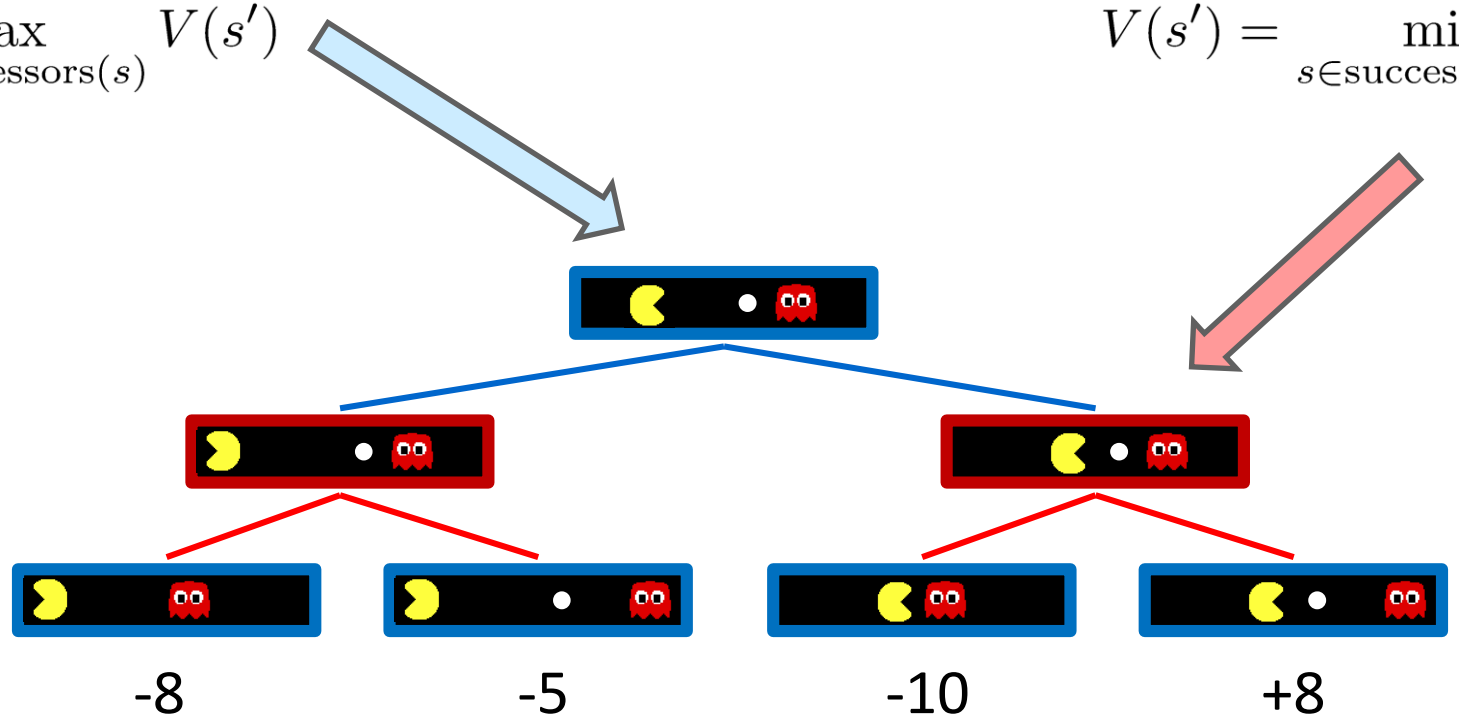
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

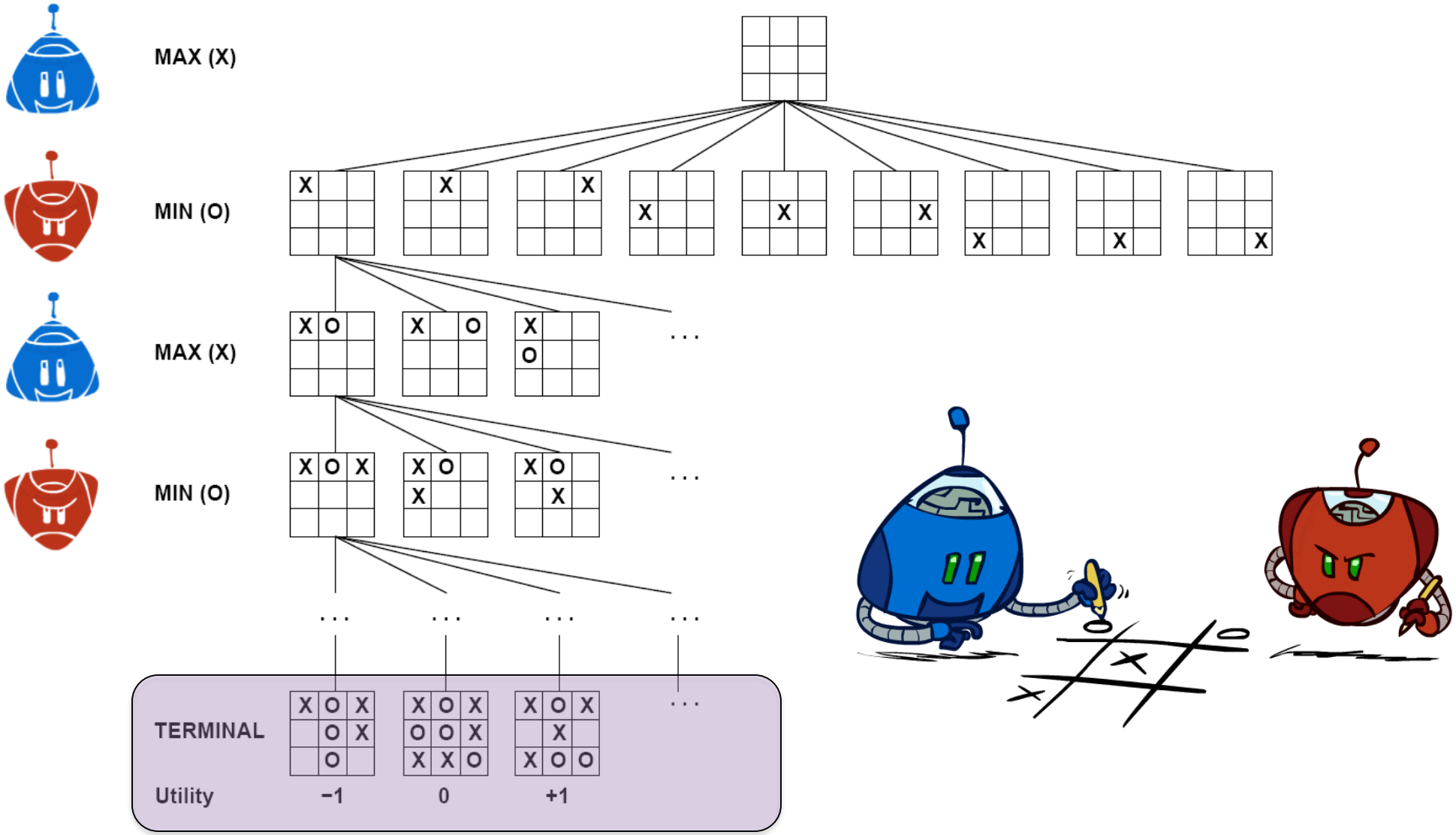
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



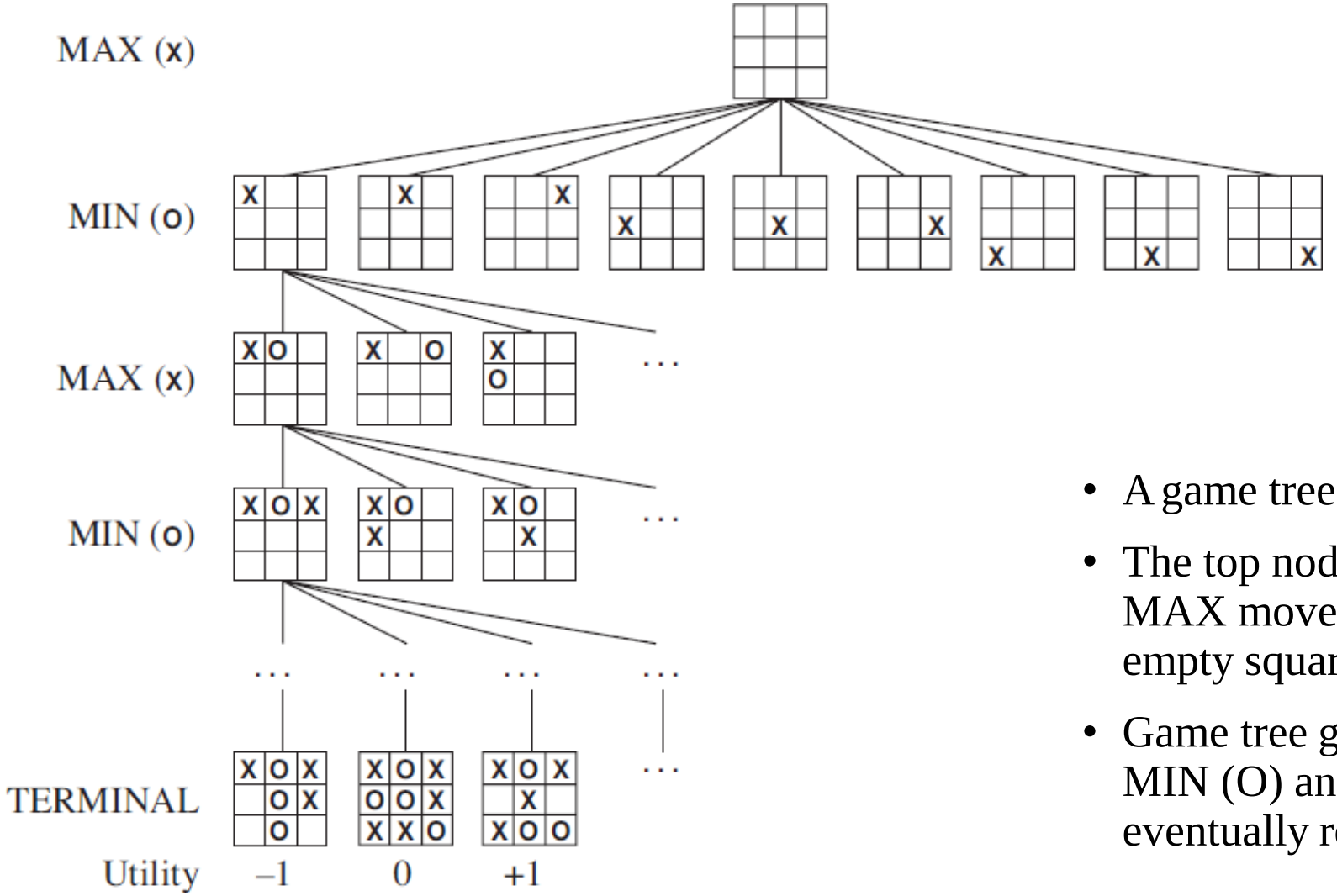
Games as Search Problem

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called **objective function** or **payoff function**), defines the final numeric value for a game that ends in terminal state s for a player p.
  - In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2 .
  - A **zero-sum game** is (constant-sum is a better term) defined as one where the total payoff to all players is the same for every instance of the game.
  - Chess is zero-sum because every game has payoff of either 0+1, 1+0 or 1/2+1/2.

Games as Search Problem

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the **nodes are game states** and the **edges are moves**.
  - For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.
  - But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.
- We have two players: MAX and MIN.
  - MAX (our player) moves first in the game.
- Regardless of size of game tree, MAX's (our player's) job to search for a good move in search tree.
  - A search algorithm does not explore the full game tree, and examines enough nodes to allow a player to determine what move to make.

Game Tree



- A game tree for the game of tic-tac-toe.
- The top node is the initial state, and MAX moves first, placing an X in an empty square.
- Game tree gives alternating moves by MIN (O) and MAX (X), until eventually reaching terminal states.

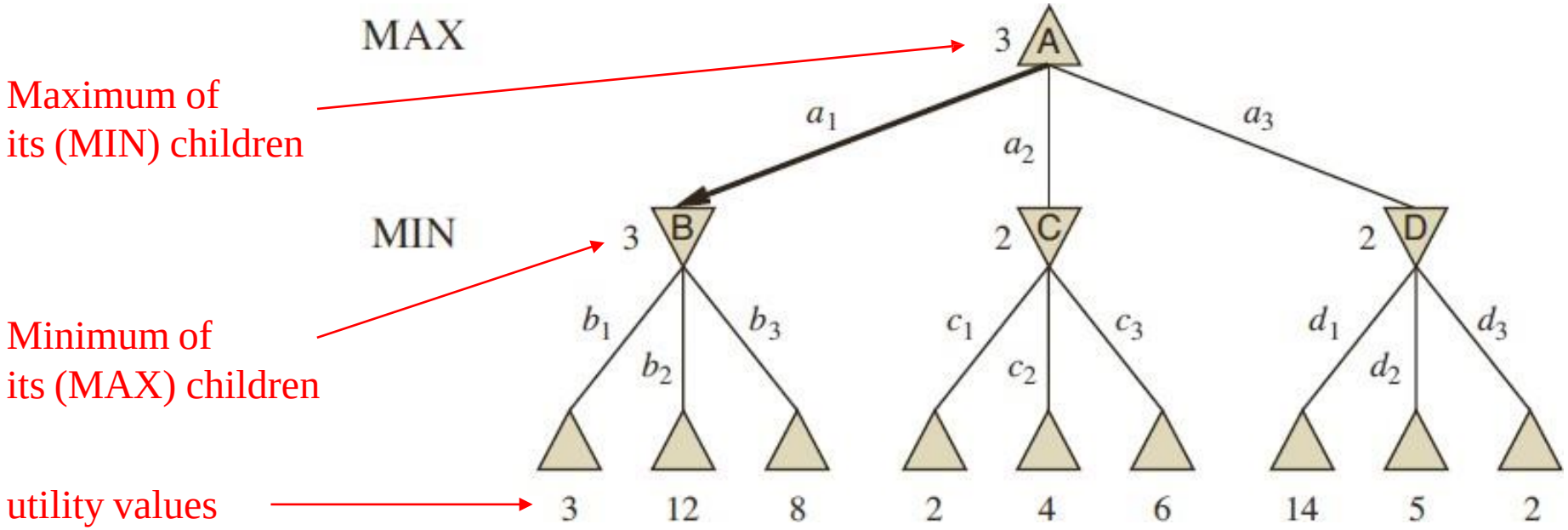
Optimal Decisions in Games

- **Optimal Solution** in a search problem is a sequence of actions leading to a ***goal state*** (***a terminal state that is a win).***
- MAX must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN.
- Given a game tree, **optimal strategy** can be determined from **minimax value** of each node n, and it is called as MINIMAX(n).
- **The minimax value of a node** ***is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game***.
- The **minimax value of a terminal state** is just its ***utility***.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

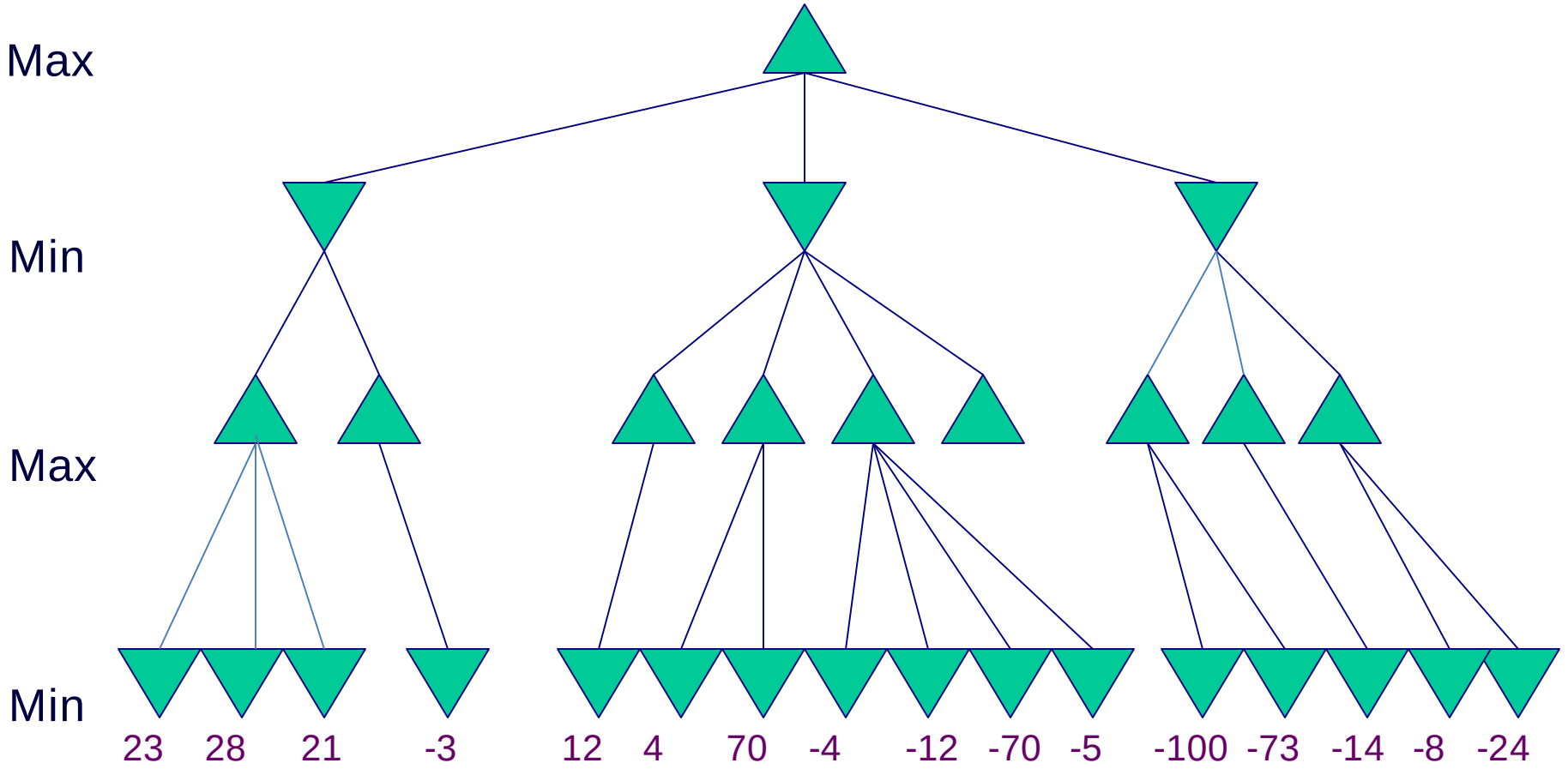
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

MINIMAX

- Δ nodes are "MAX nodes," in which it is MAX's turn to move, and ∇ nodes are "MIN nodes."
- Terminal nodes show the utility values for MAX; other nodes are labeled with their minimax values.
- MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.

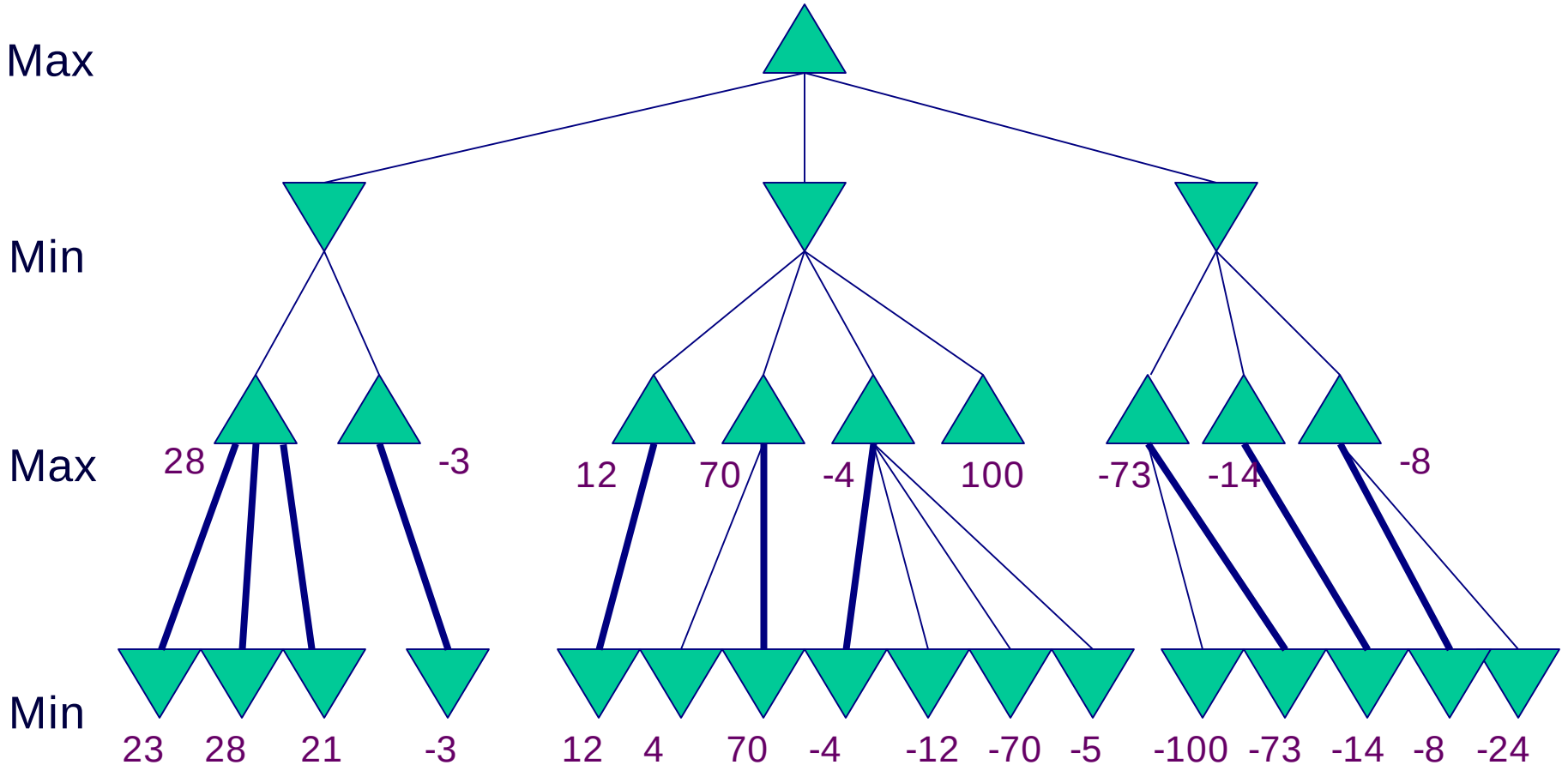


Minimax Tree



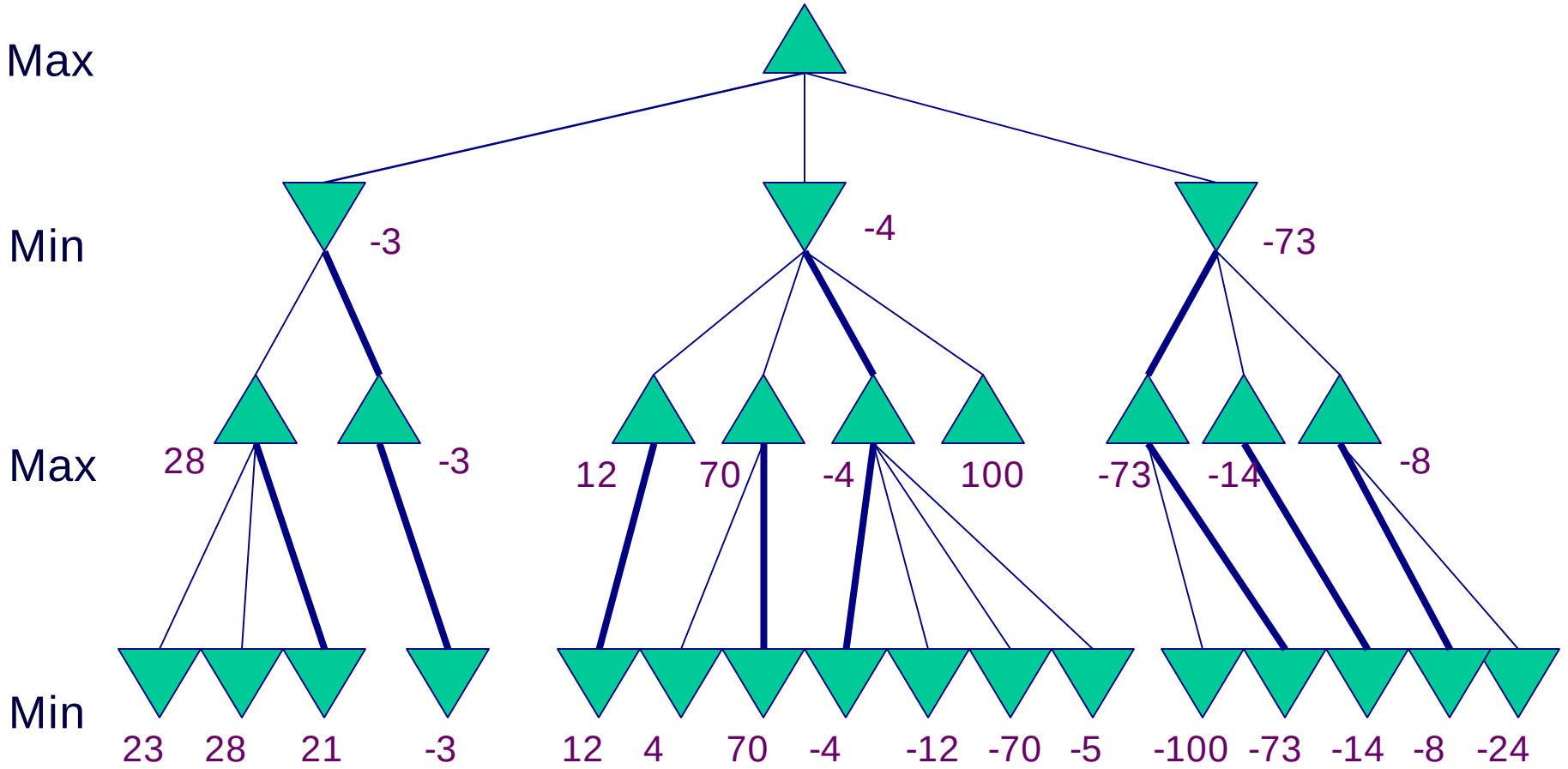
Terminal nodes: values calculated from the utility function

Minimax Tree

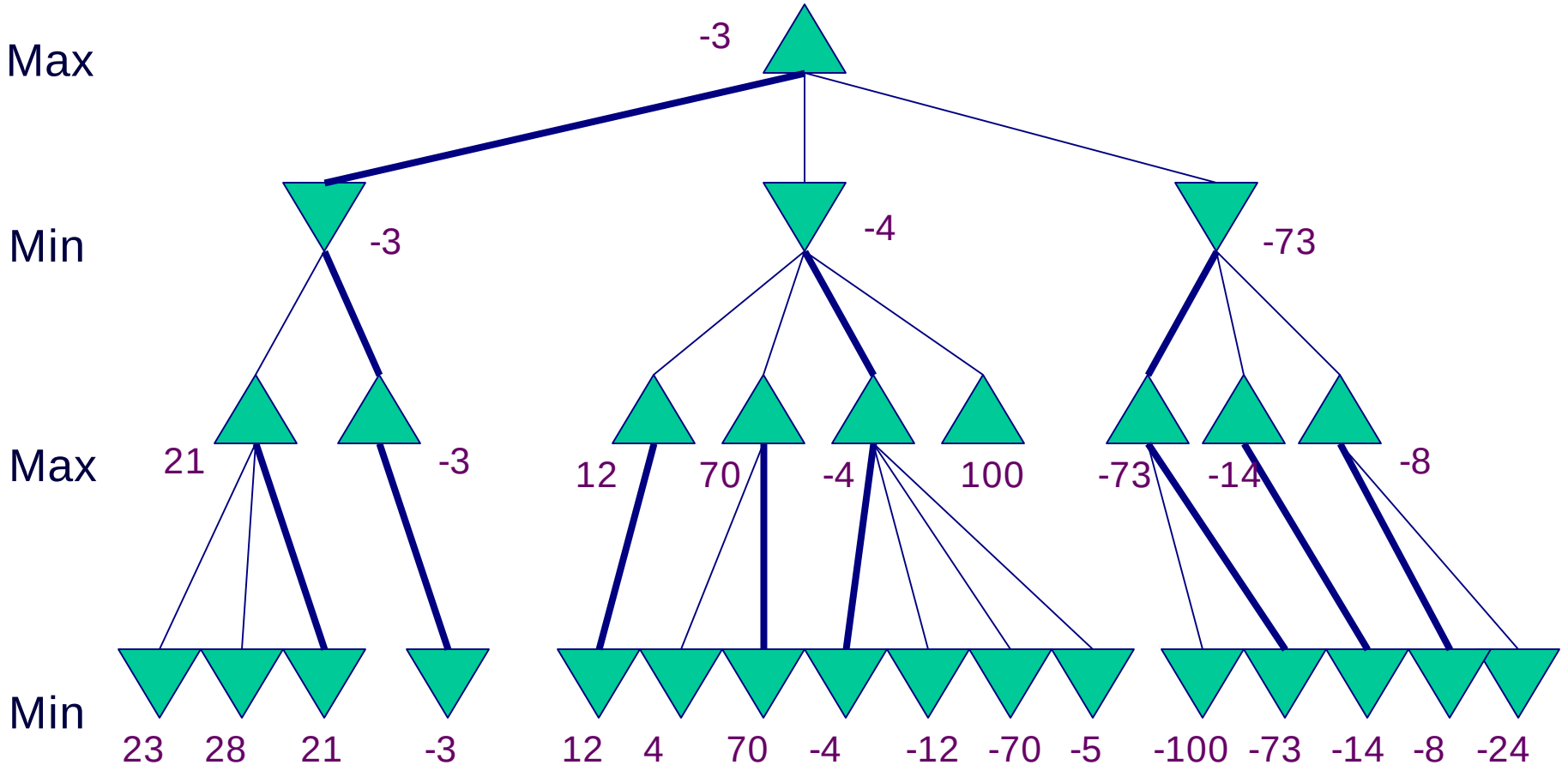


Other nodes: values calculated via minimax algorithm

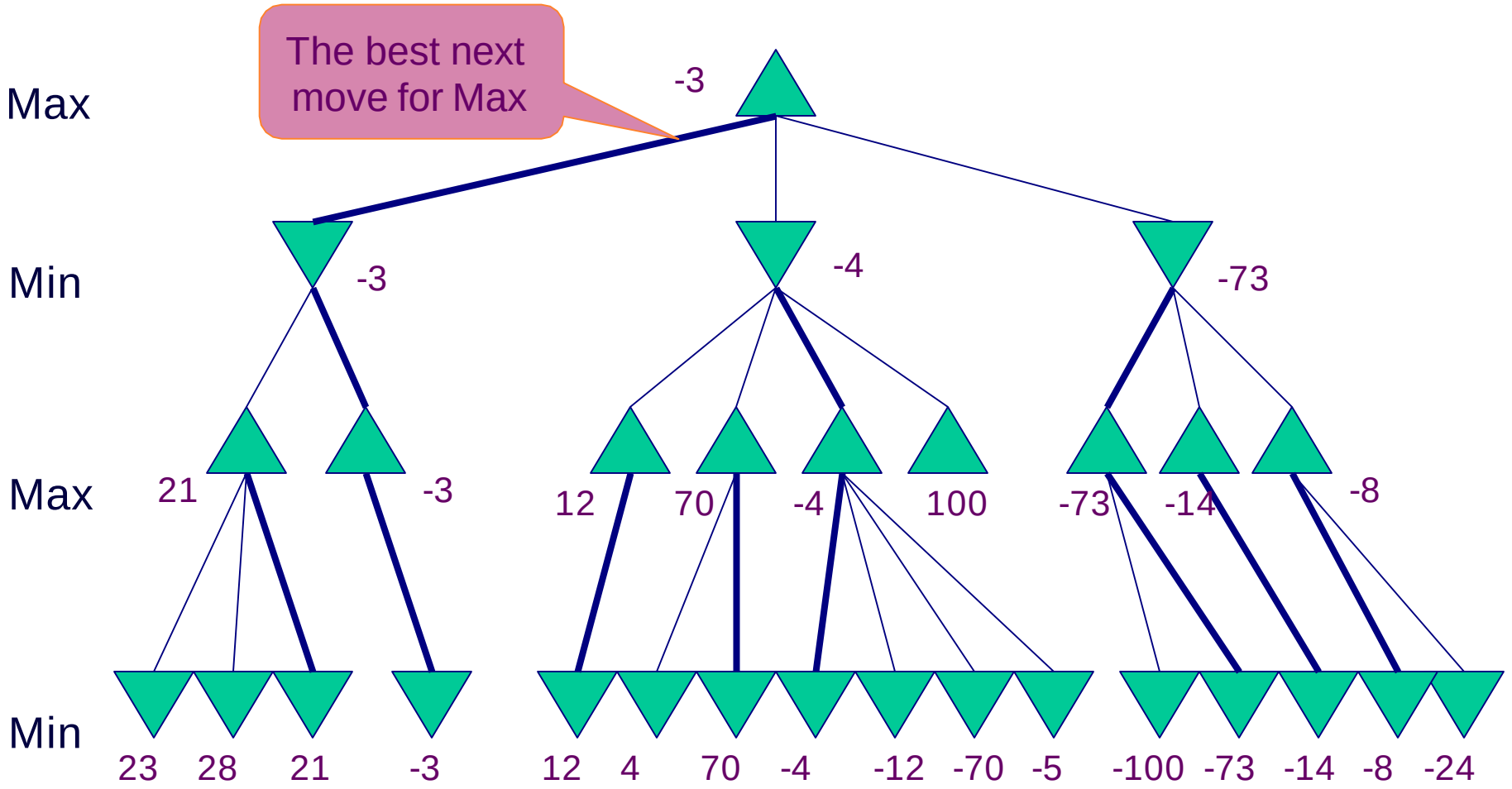
Minimax Tree



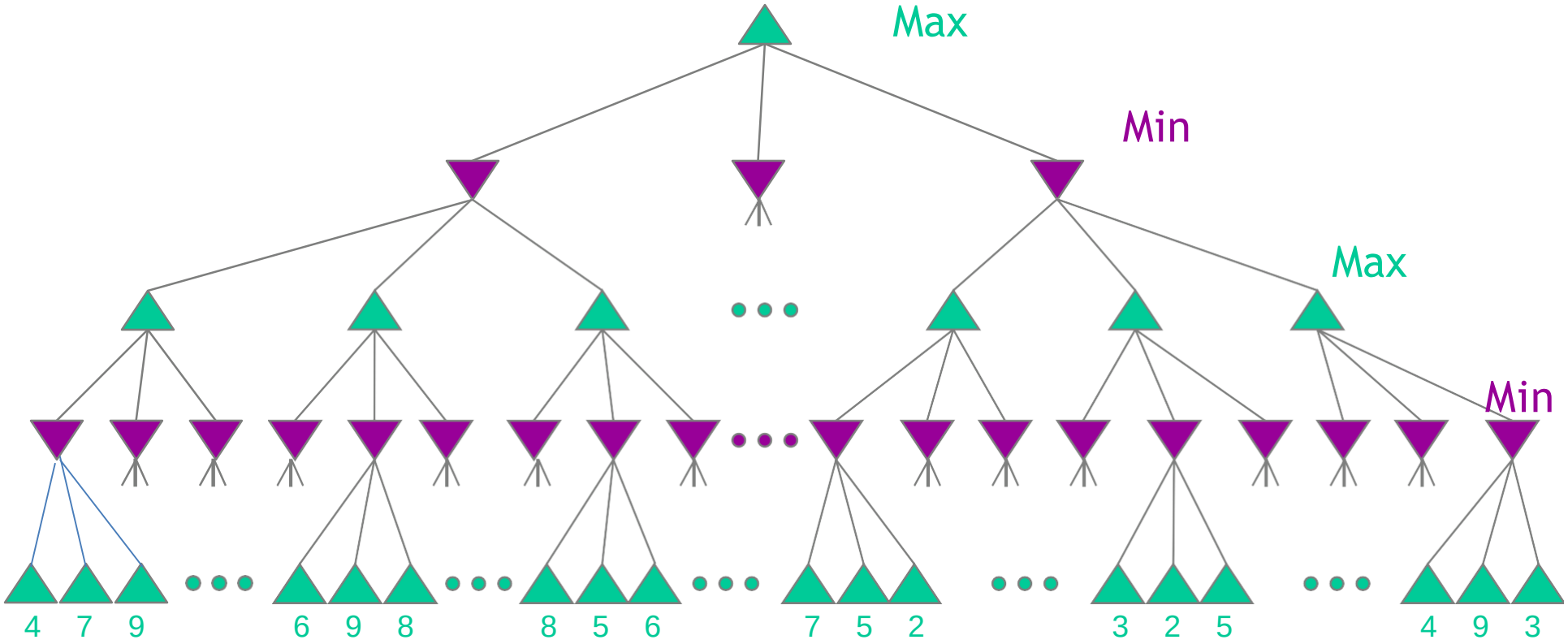
Minimax Tree



Minimax Tree

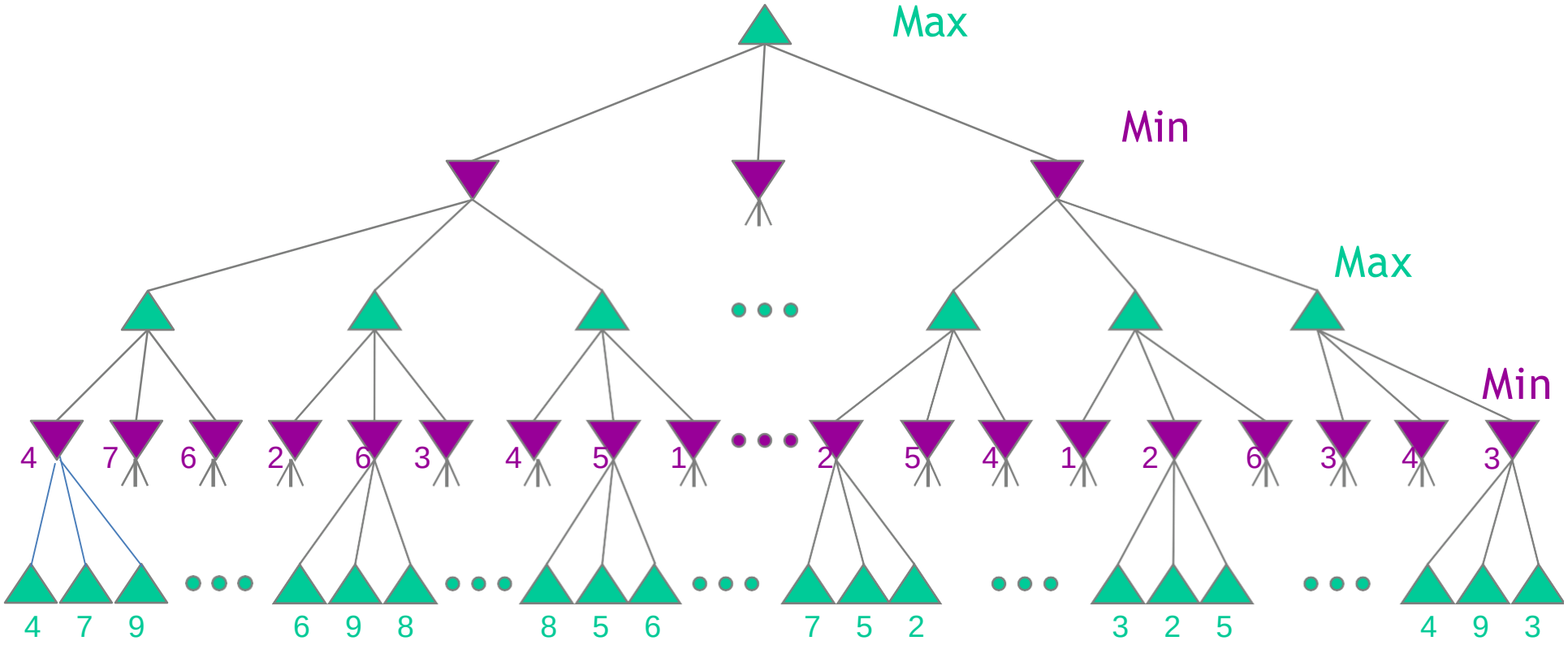


MiniMax Example-2



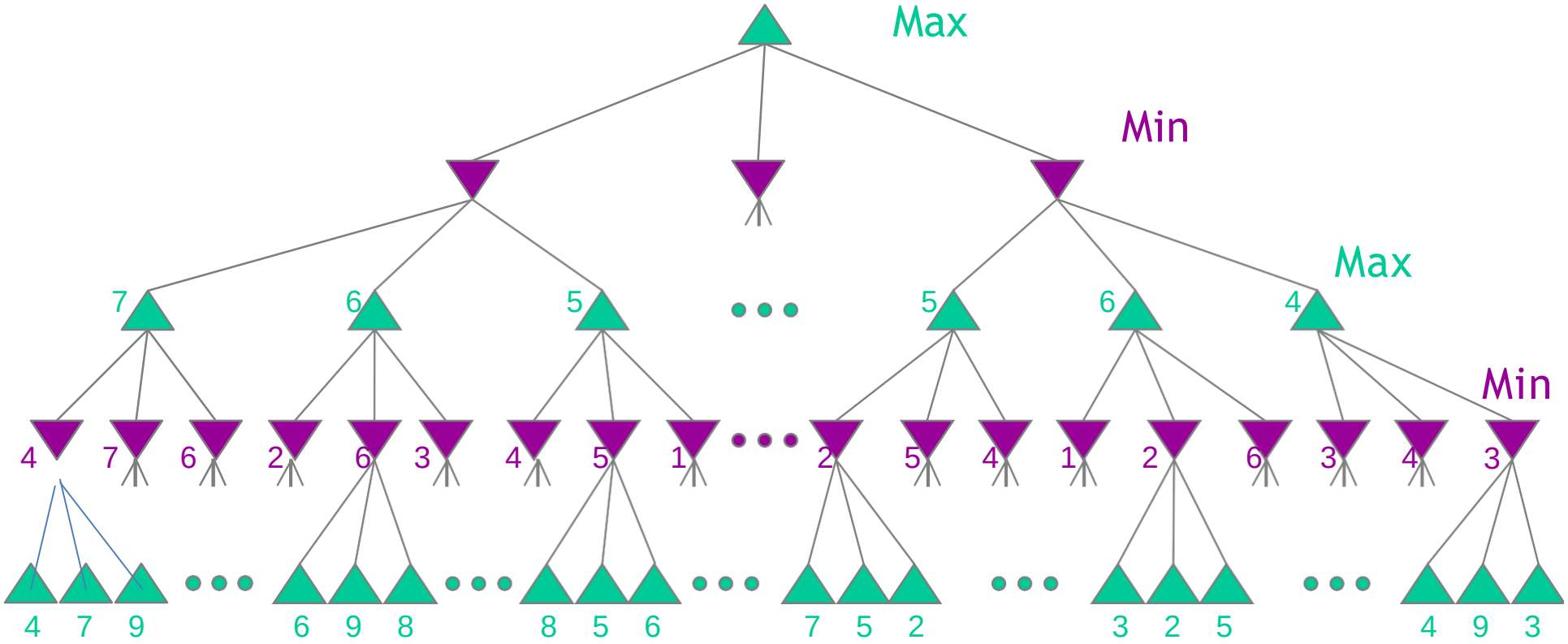
Terminal nodes: values calculated from the utility function

MiniMax Example-2

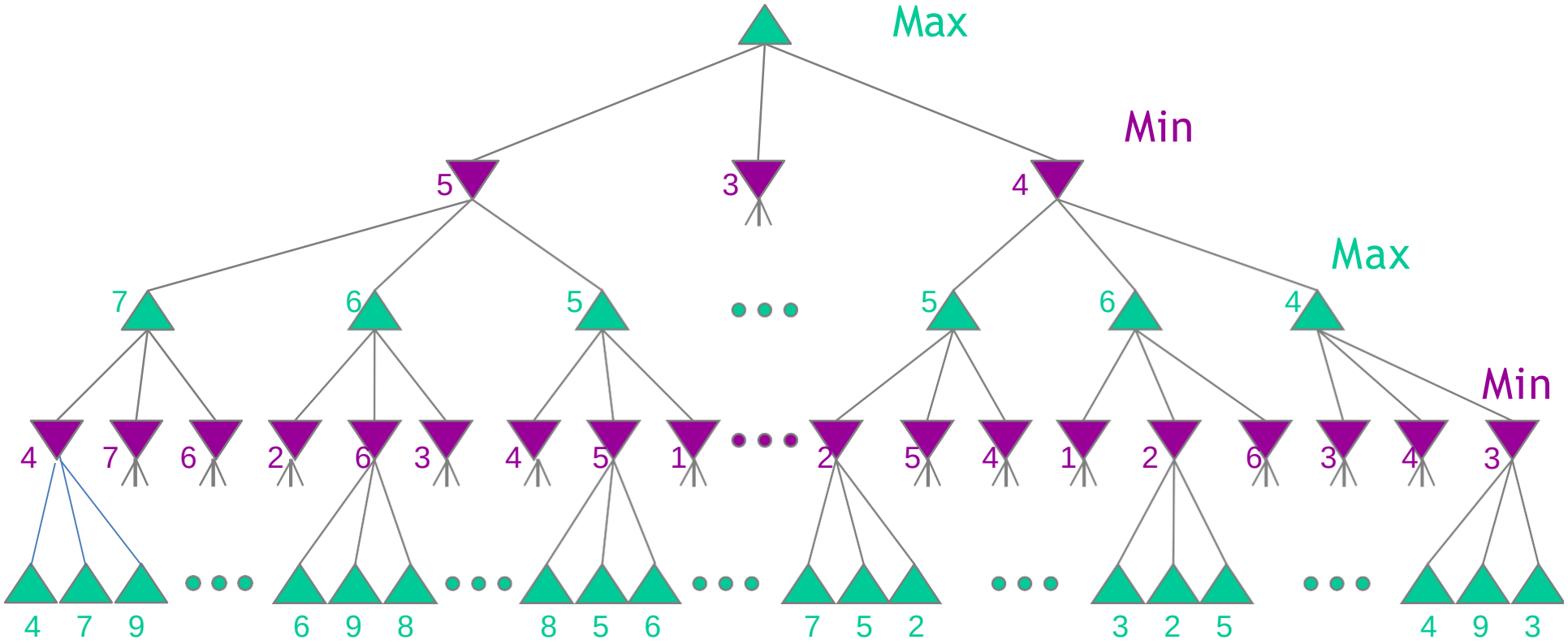


Other nodes: values calculated via minimax algorithm

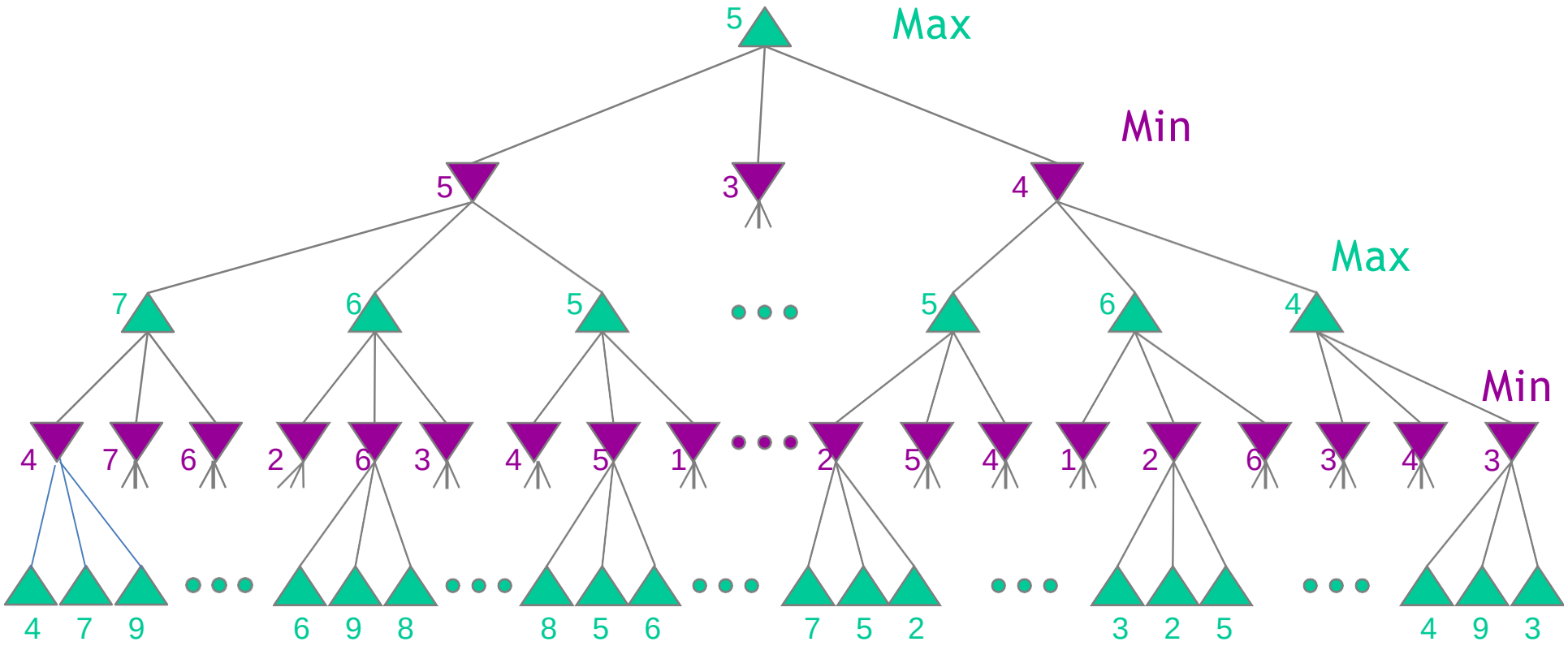
MiniMax Example-2



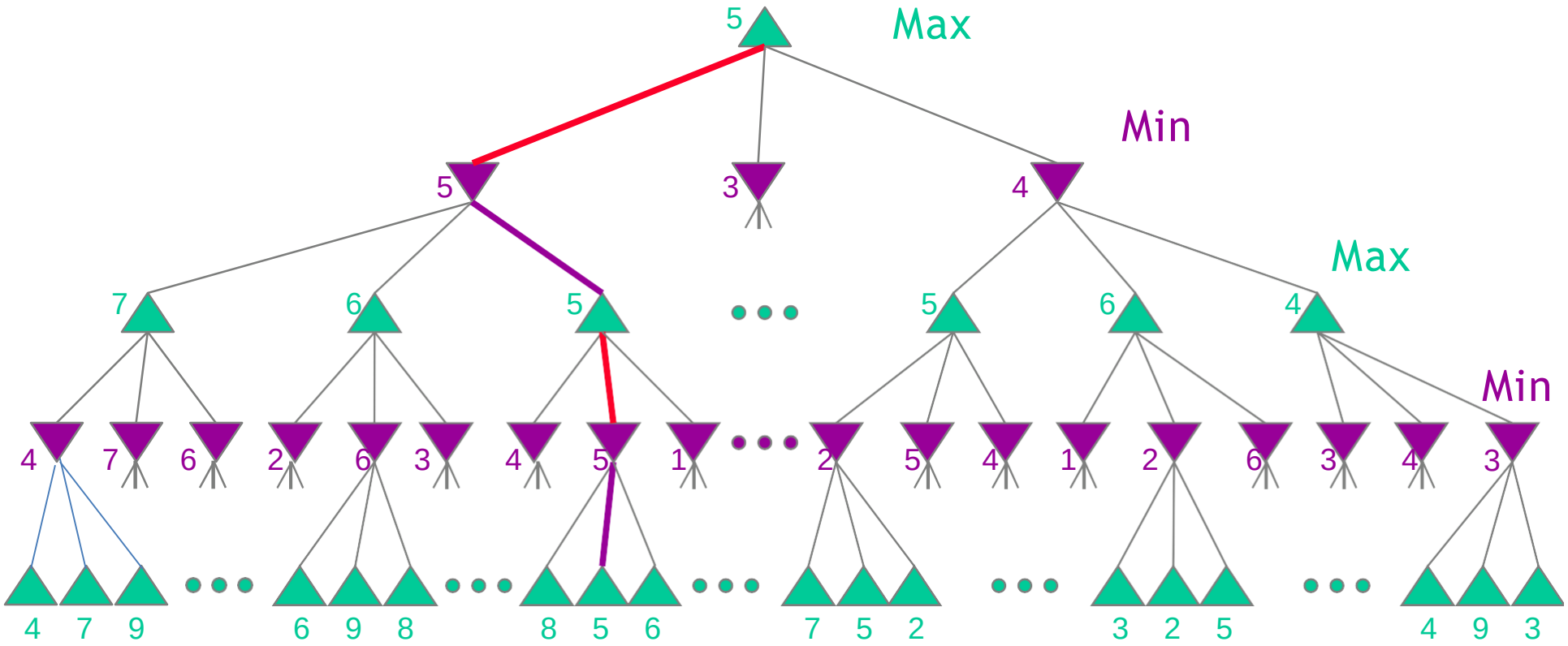
MiniMax Example-2



MiniMax Example-2



MiniMax Example-2



Moves by Max and countermoves by Min

Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

- Minimax Algorithm returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility.
- The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

Properties of Minimax Algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree.

Complete: **YES**, if tree is finite

Optimal: **YES**, against an optimal opponent.

Time complexity: **$O(b^m)$**

Space complexity: **O(bm)** (depth-first exploration) Algorithm generates all actions at once

- For real games, the time cost is totally impractical, but minimax algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.
- For chess, $b \approx 35$, $m \approx 100$ for reasonable games ➔ exact solution completely infeasible

Alpha–Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can’t eliminate the exponent, but we can effectively cut it in half.
- It is possible to compute **correct minimax decision** without looking at every node in game tree.
- **Alpha–Beta pruning** prunes away branches of the minimax tree that cannot possibly influence the final decision and it returns the same move as minimax algorithm would.

- ***α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.***
- ***β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.***

Pruning the Minimax Tree

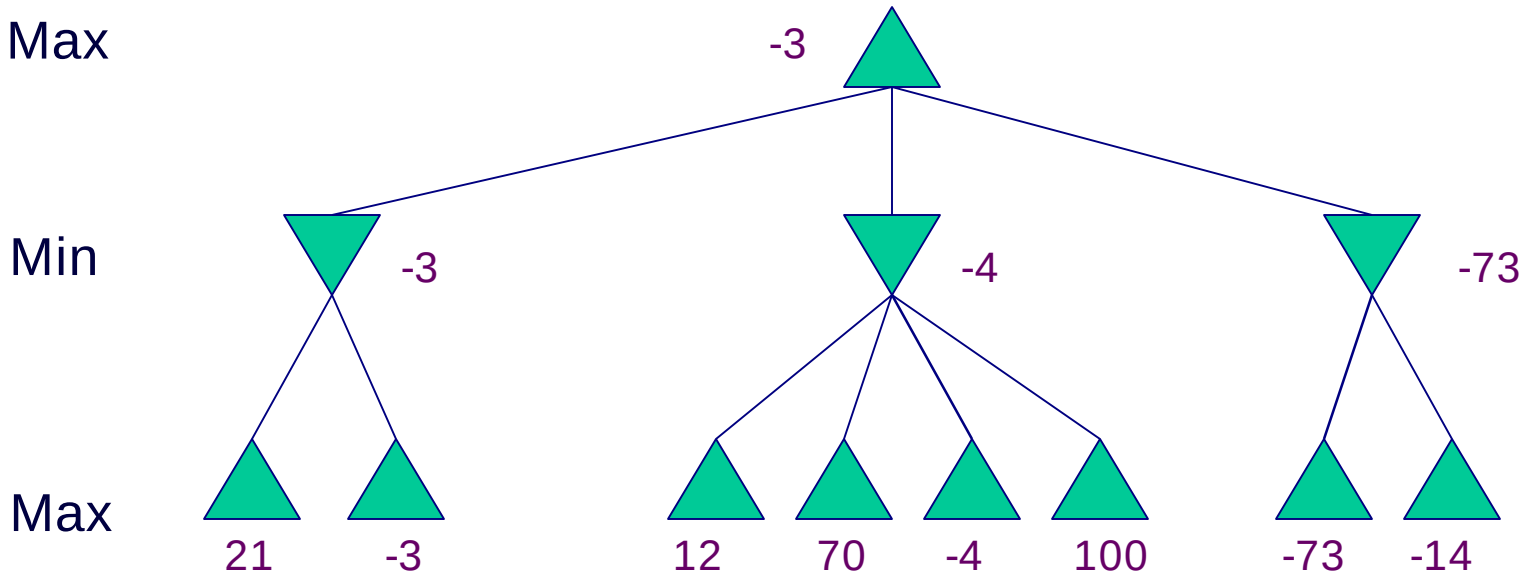
Since we have limited time available, we want to avoid unnecessary computation in the minimax tree.

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

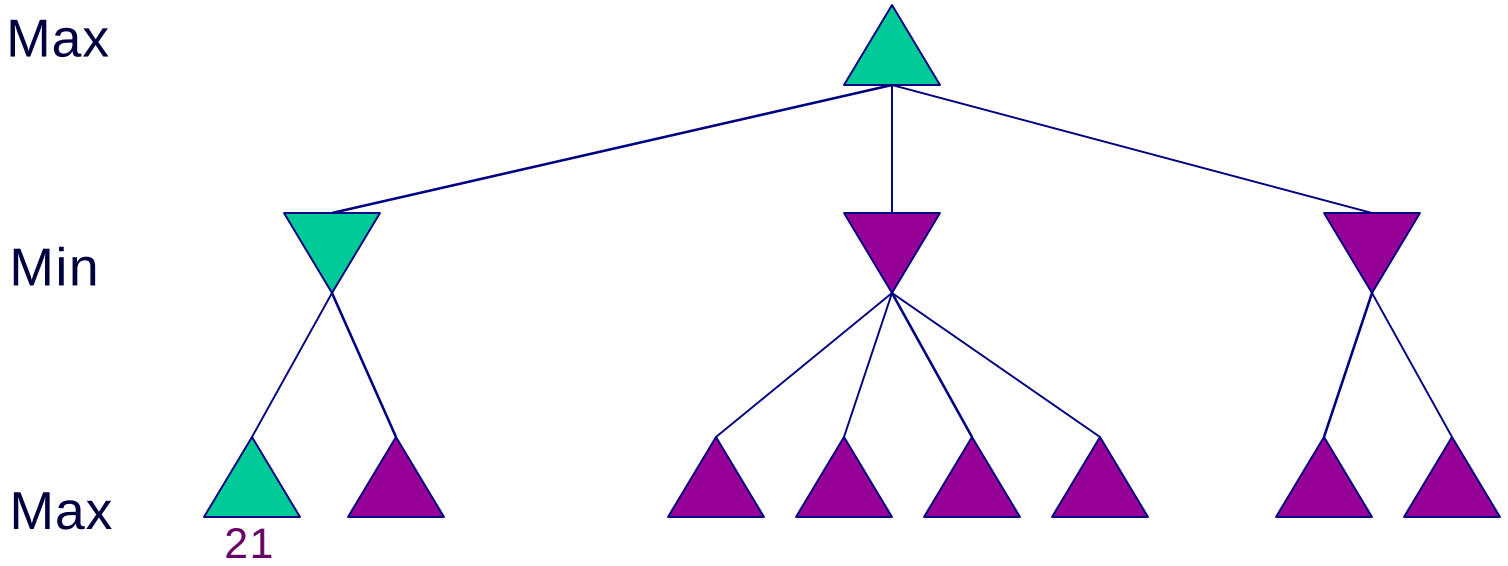
α Cuts

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

α Cut Example

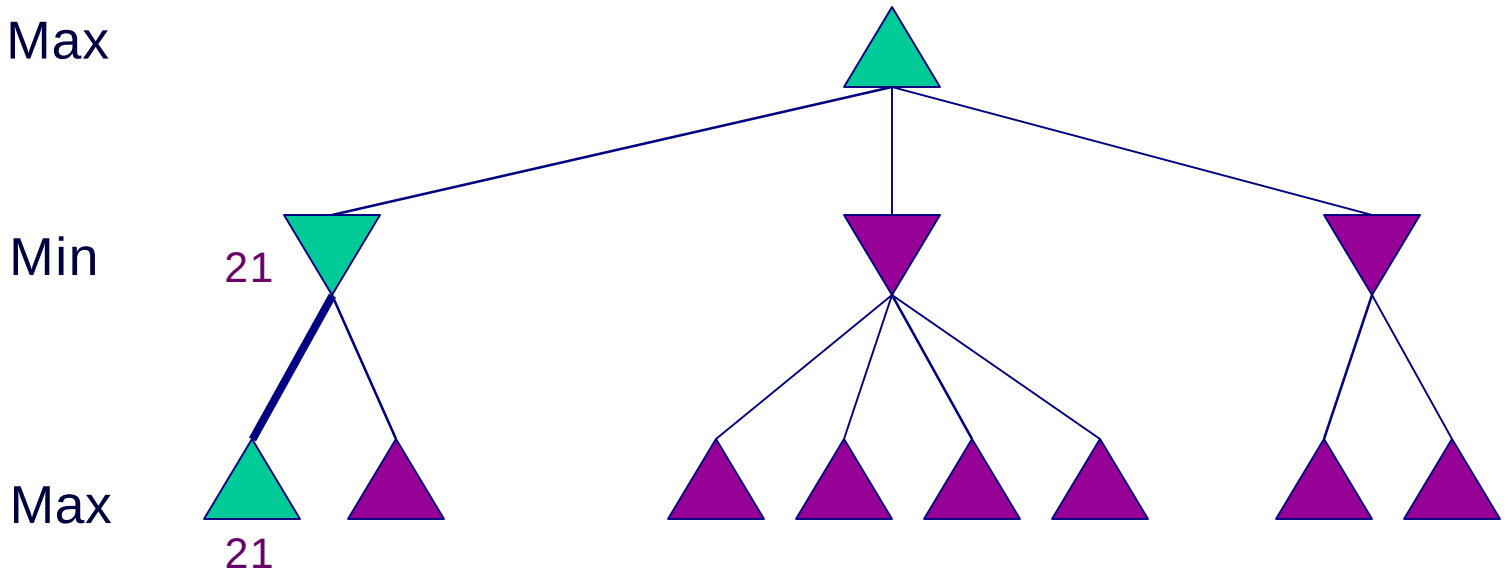


α Cut Example



Depth first search along path 1

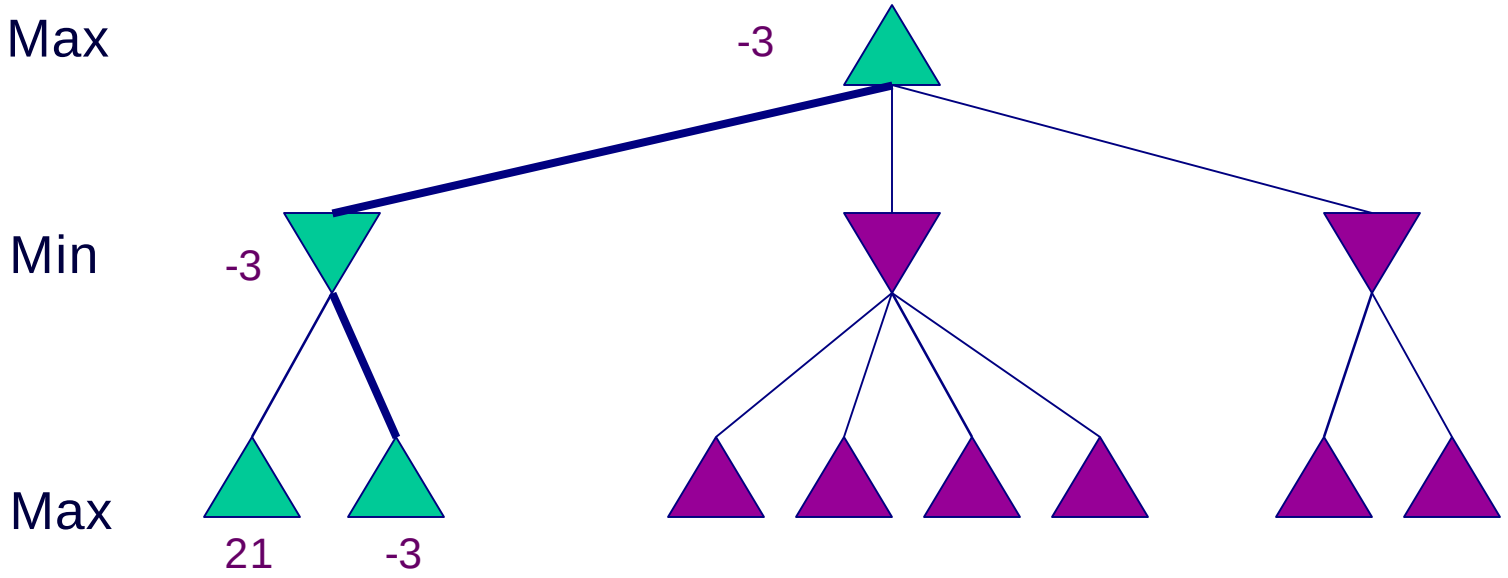
α Cut Example



21 is minimum so far (second level)

Can't evaluate yet at top level

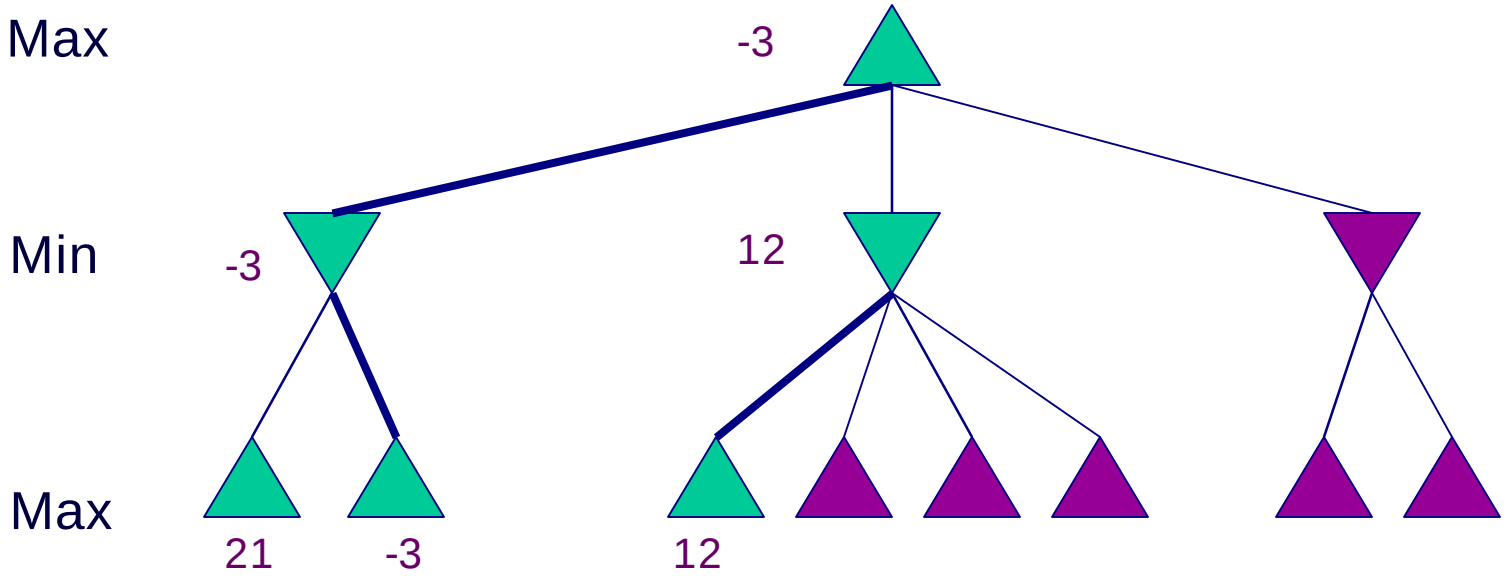
α Cut Example



-3 is minimum so far (second level)

-3 is maximum so far (top level)

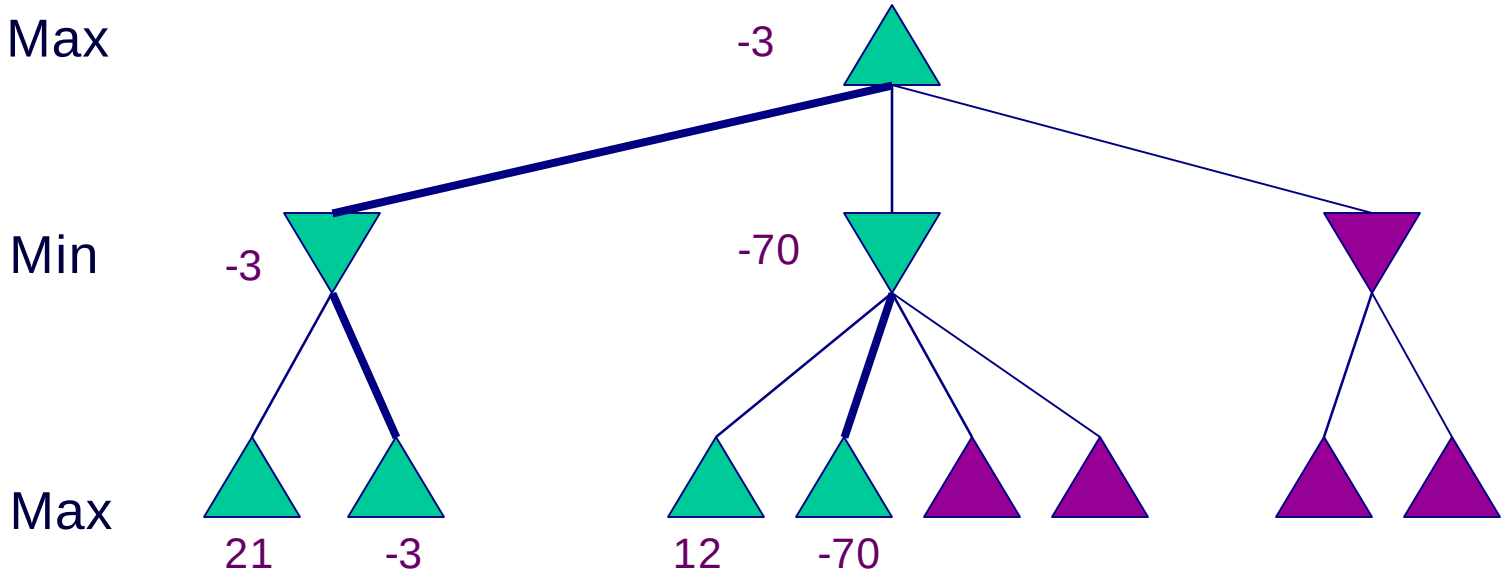
α Cut Example



12 is minimum so far (second level)

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

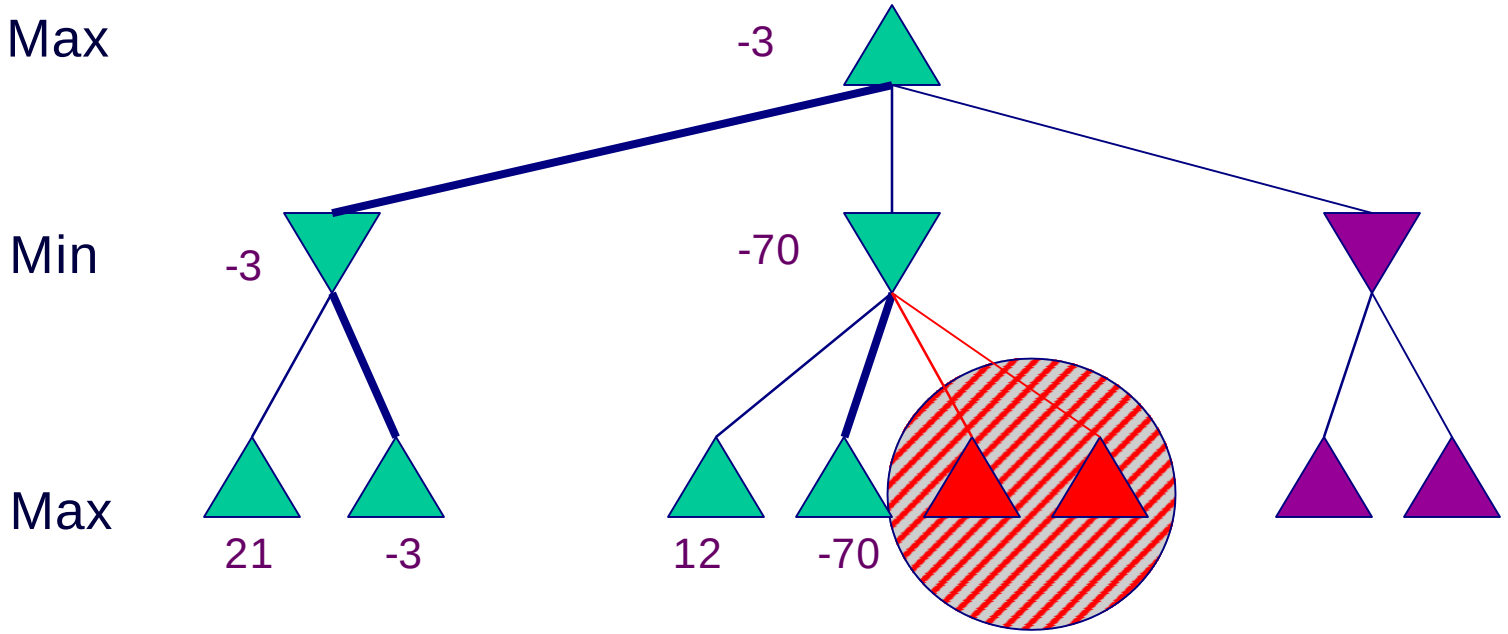
α Cut Example



-70 is now minimum so far (second level)

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

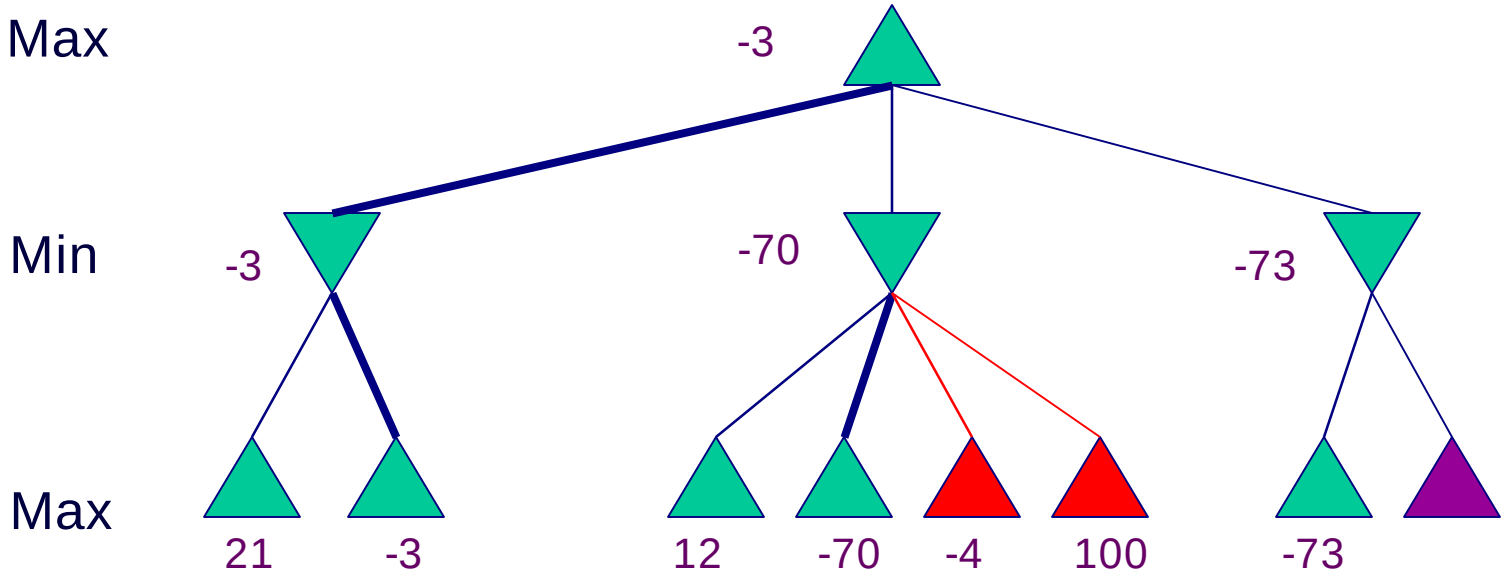
α Cut Example



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

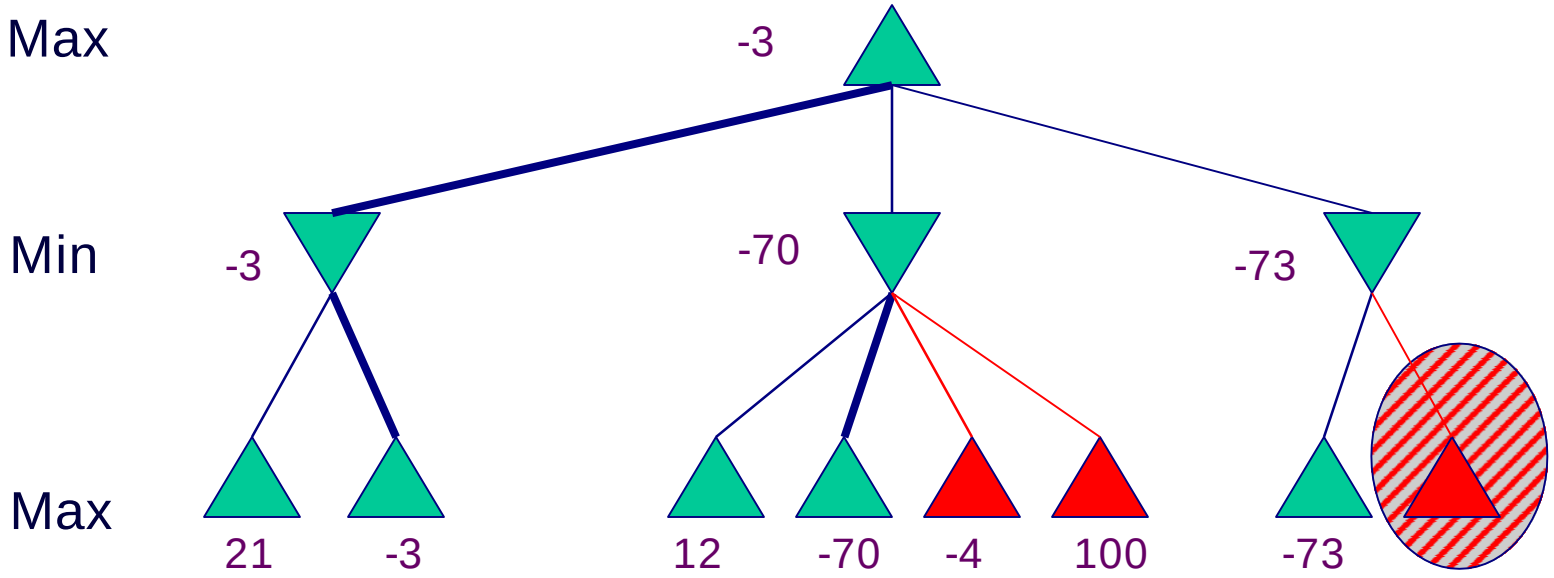
We can stop exploring that node

α Cut Example



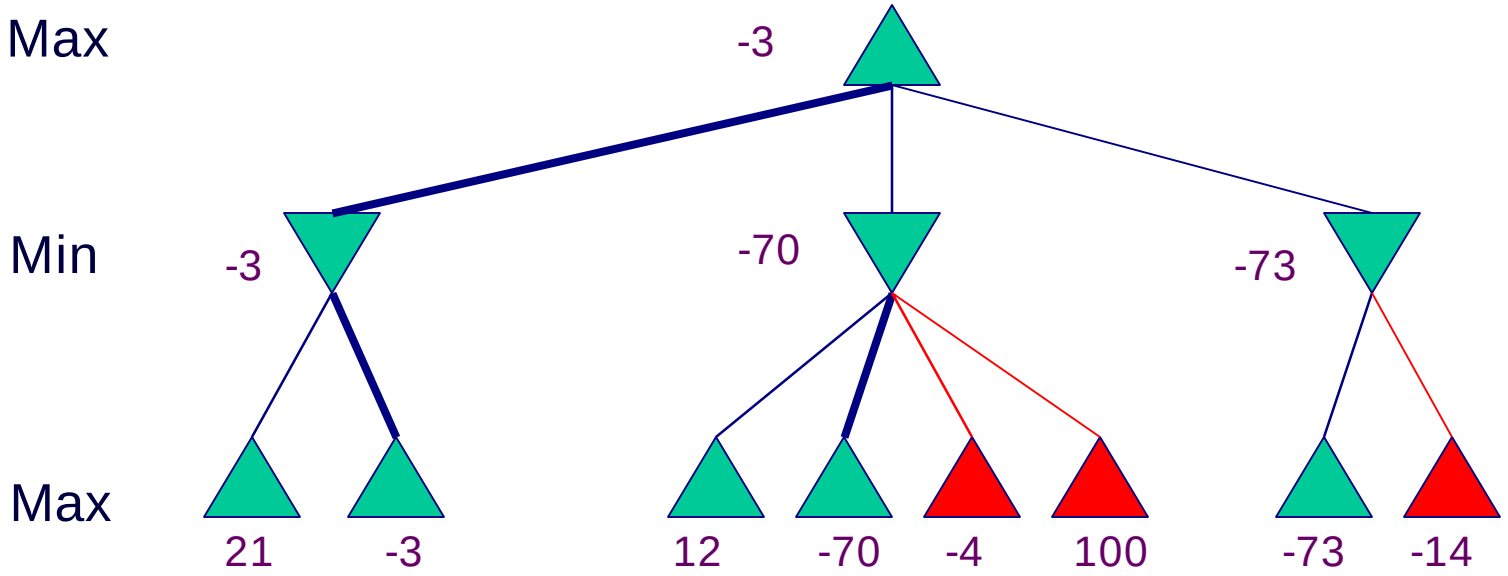
Evaluation at second level is again -73

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



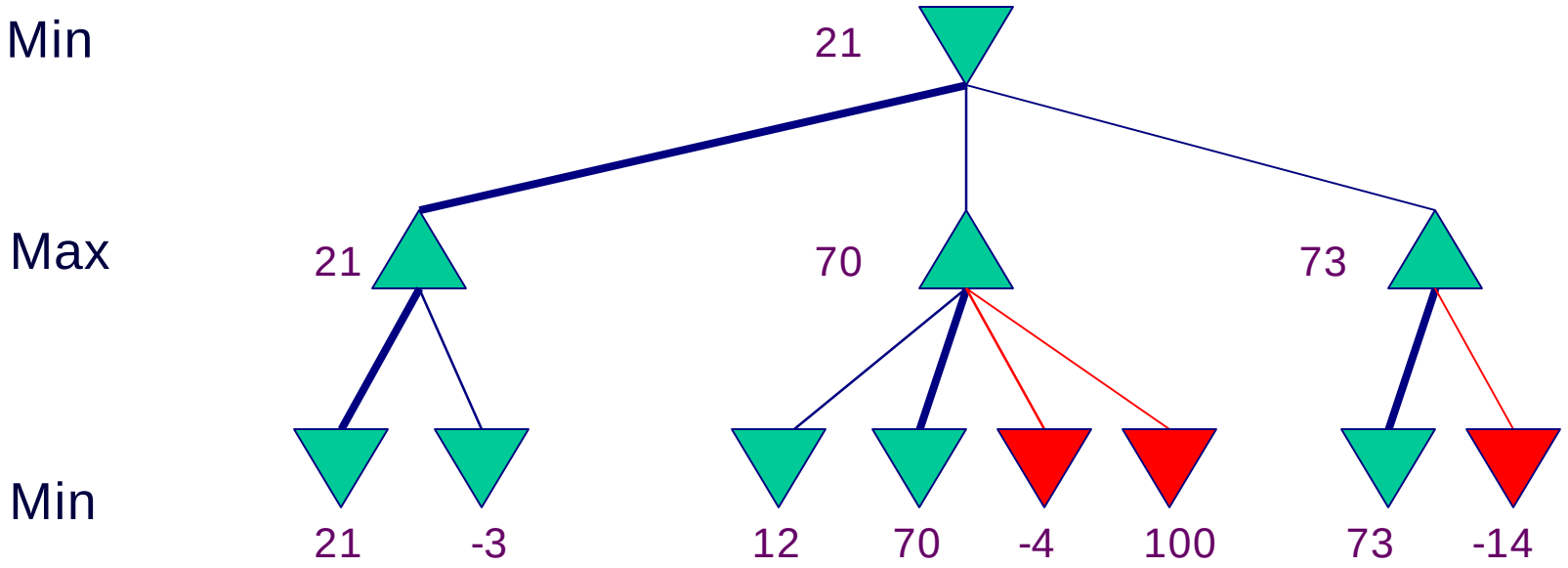
As a result, we evaluated the Max node without evaluating several of the possible paths

β Cuts

Similar idea to α cuts, but the other way around

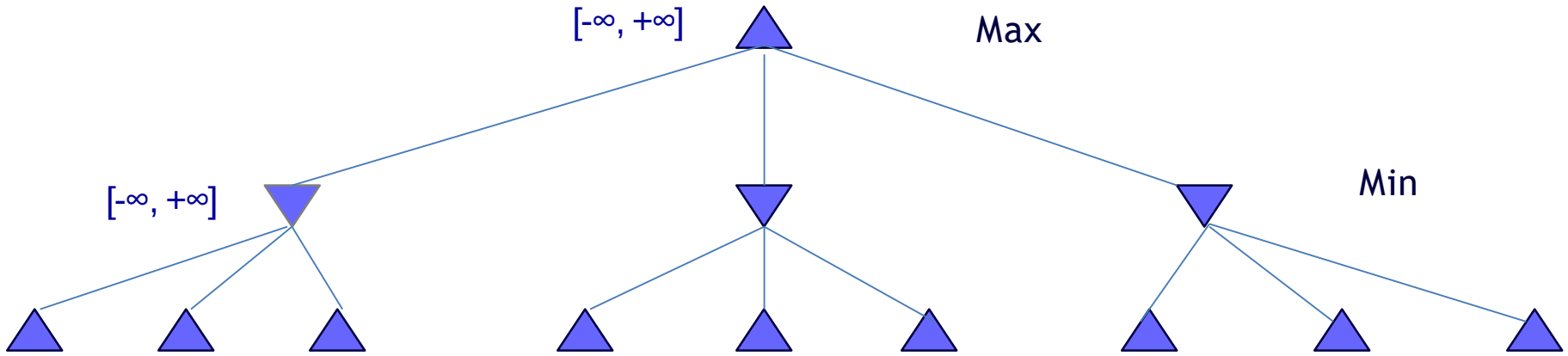
If the current minimum is less than the successor's max value, don't look down that max tree any more

β Cut Example



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

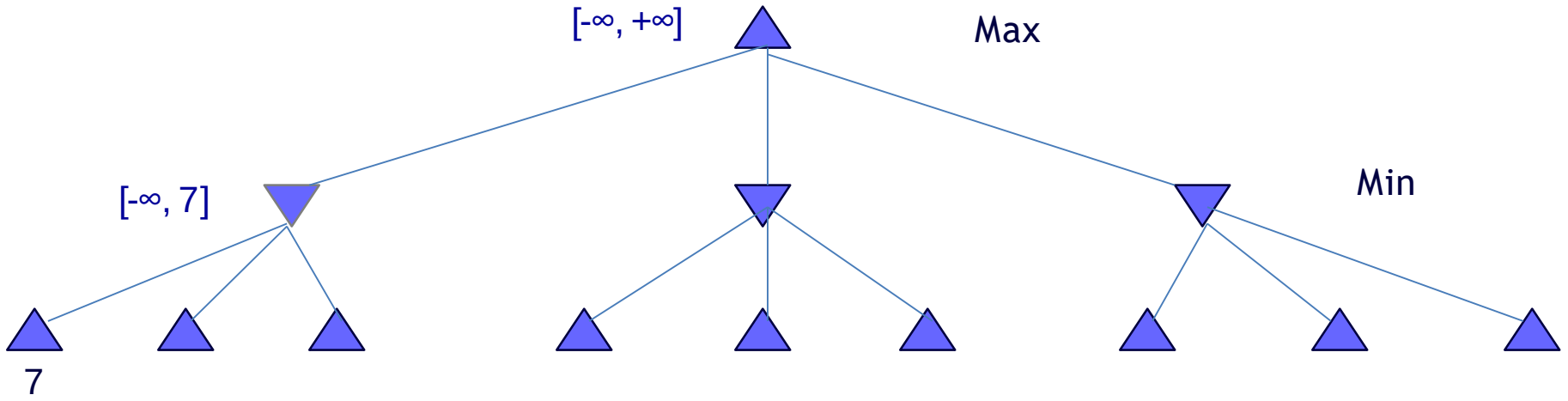
Alpha-Beta Example 2



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'sor Min'sperspective
  - 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

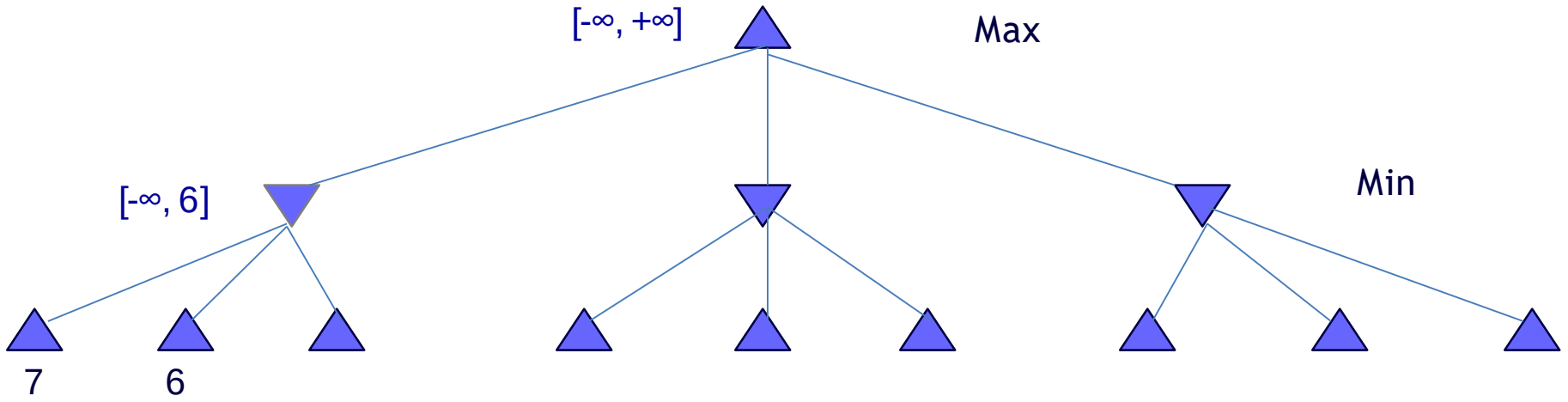
Alpha-Beta Example 2



α best choice for Max ?

β best choice for Min 7

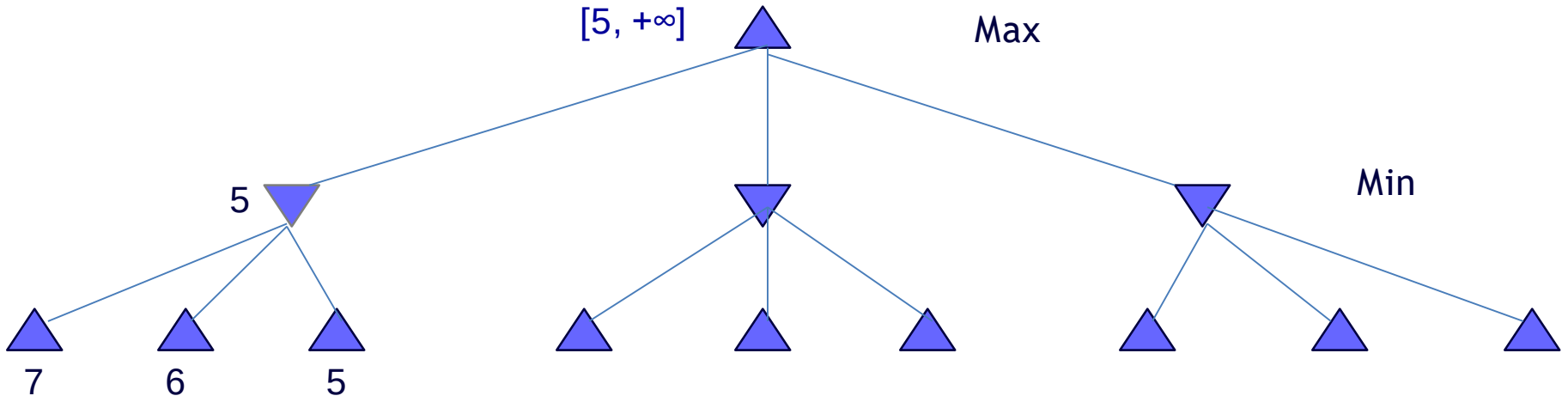
Alpha-Beta Example 2



α best choice for Max ?

β best choice for Min 6

Alpha-Beta Example 2

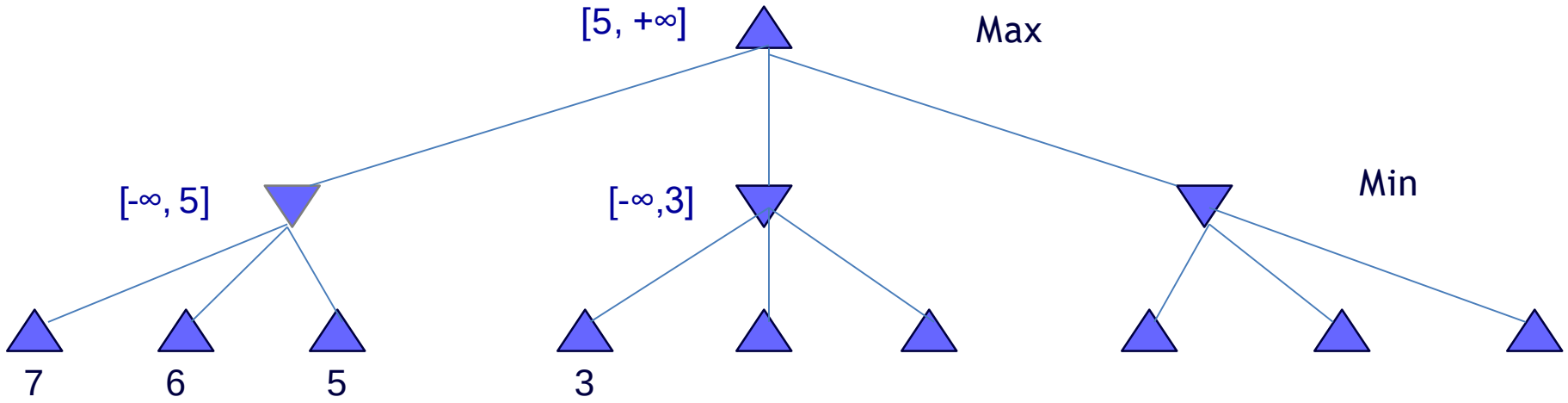


α best choice for Max 5

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

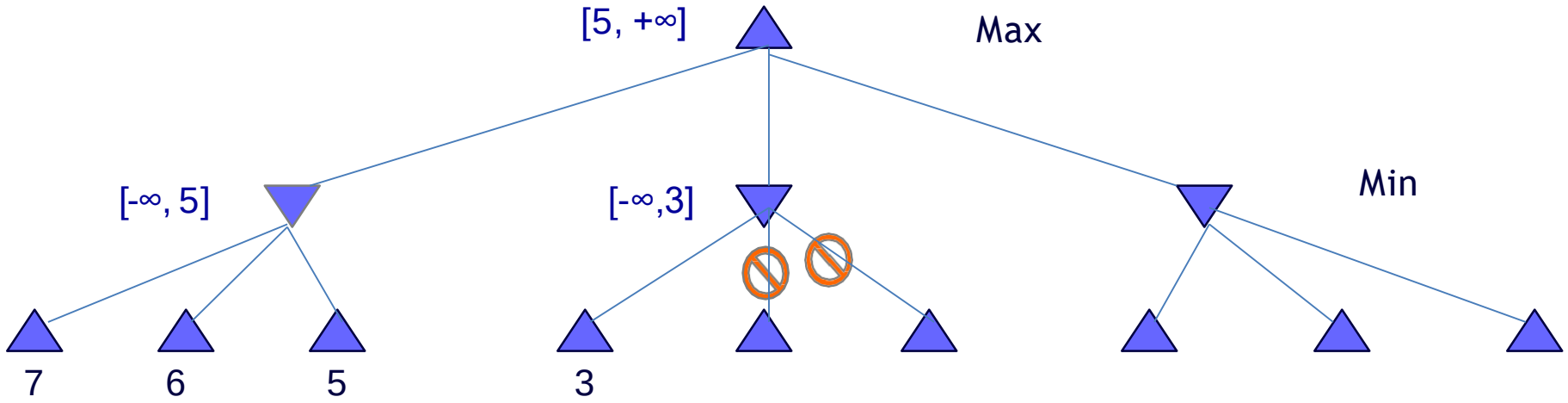


α best choice for Max 5

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

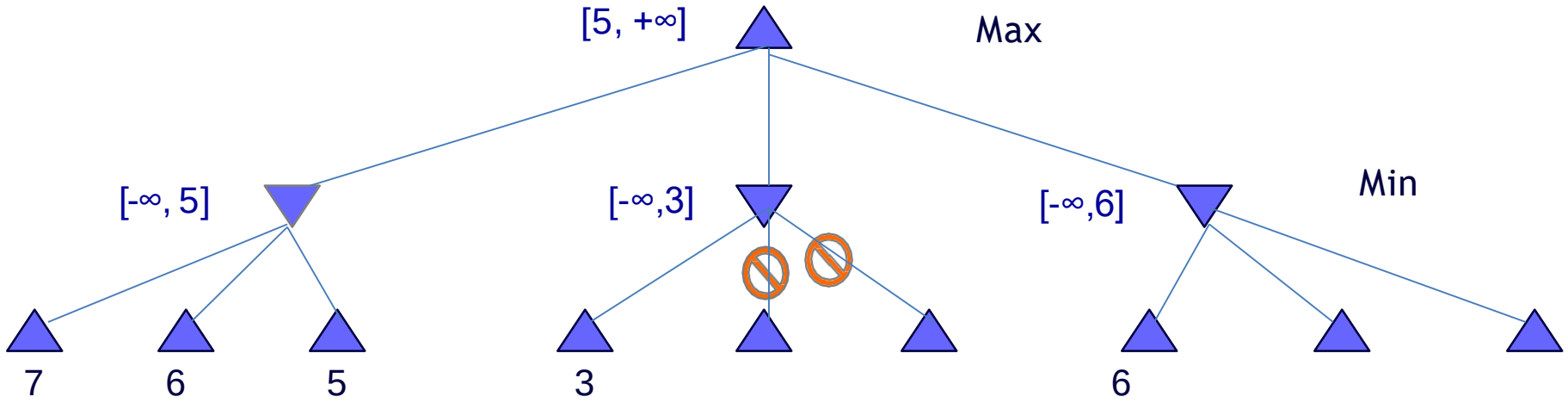


α best choice for Max 5

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

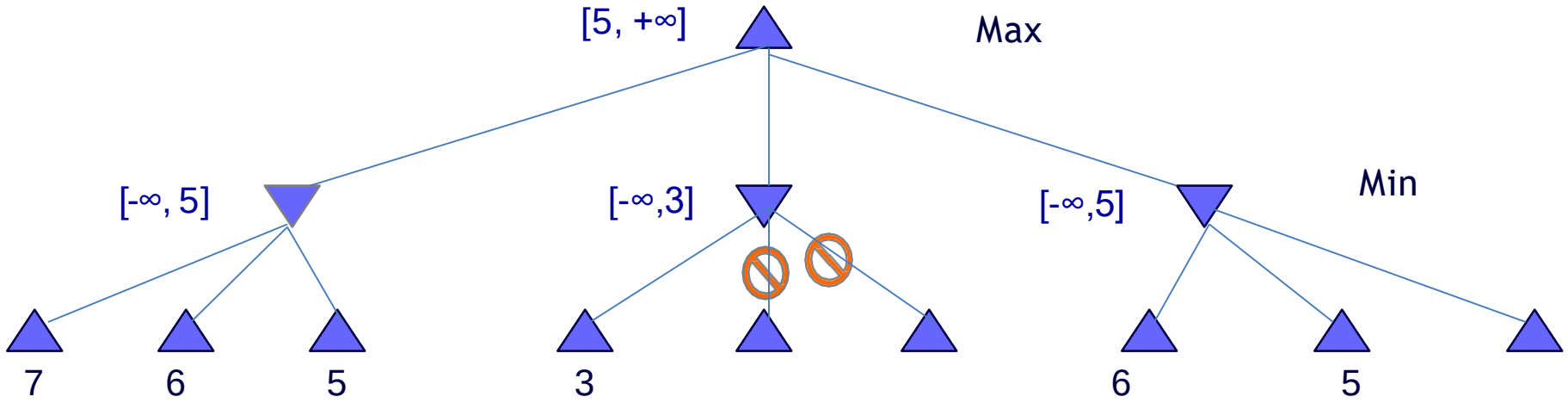


α best choice for Max 5

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

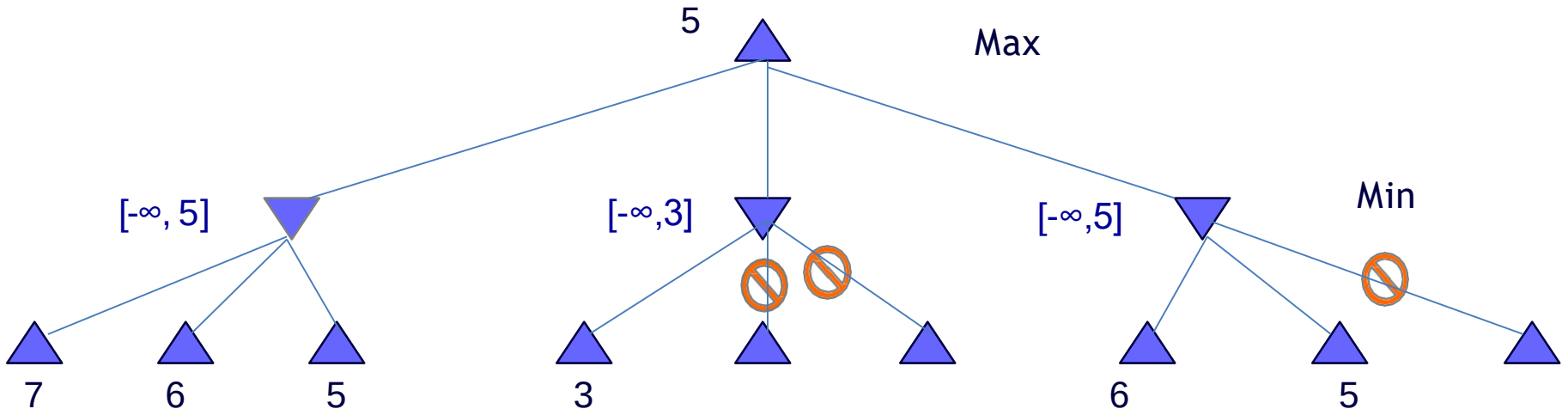


α best choice for Max 5

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

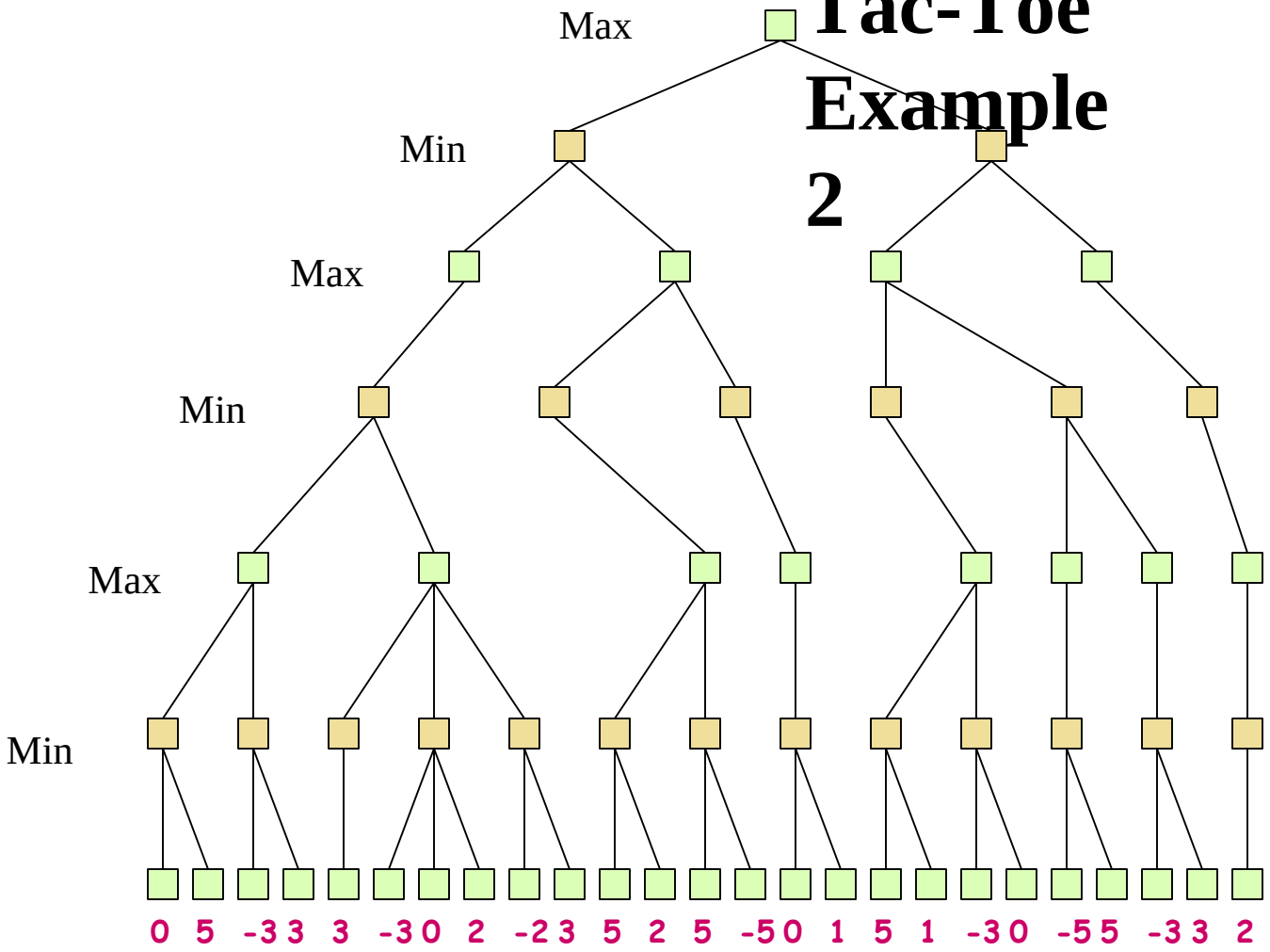


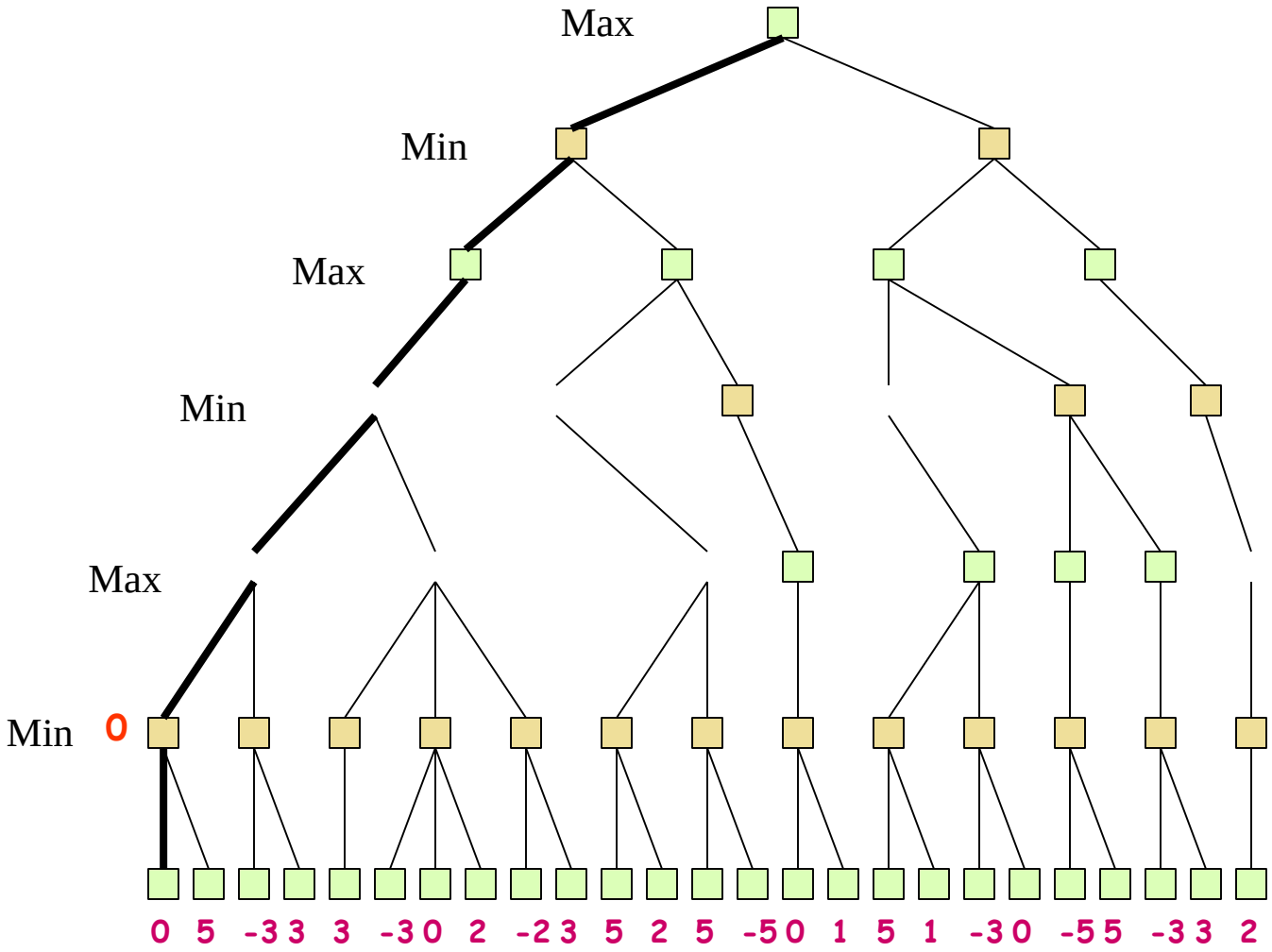
α best choice for Max 5

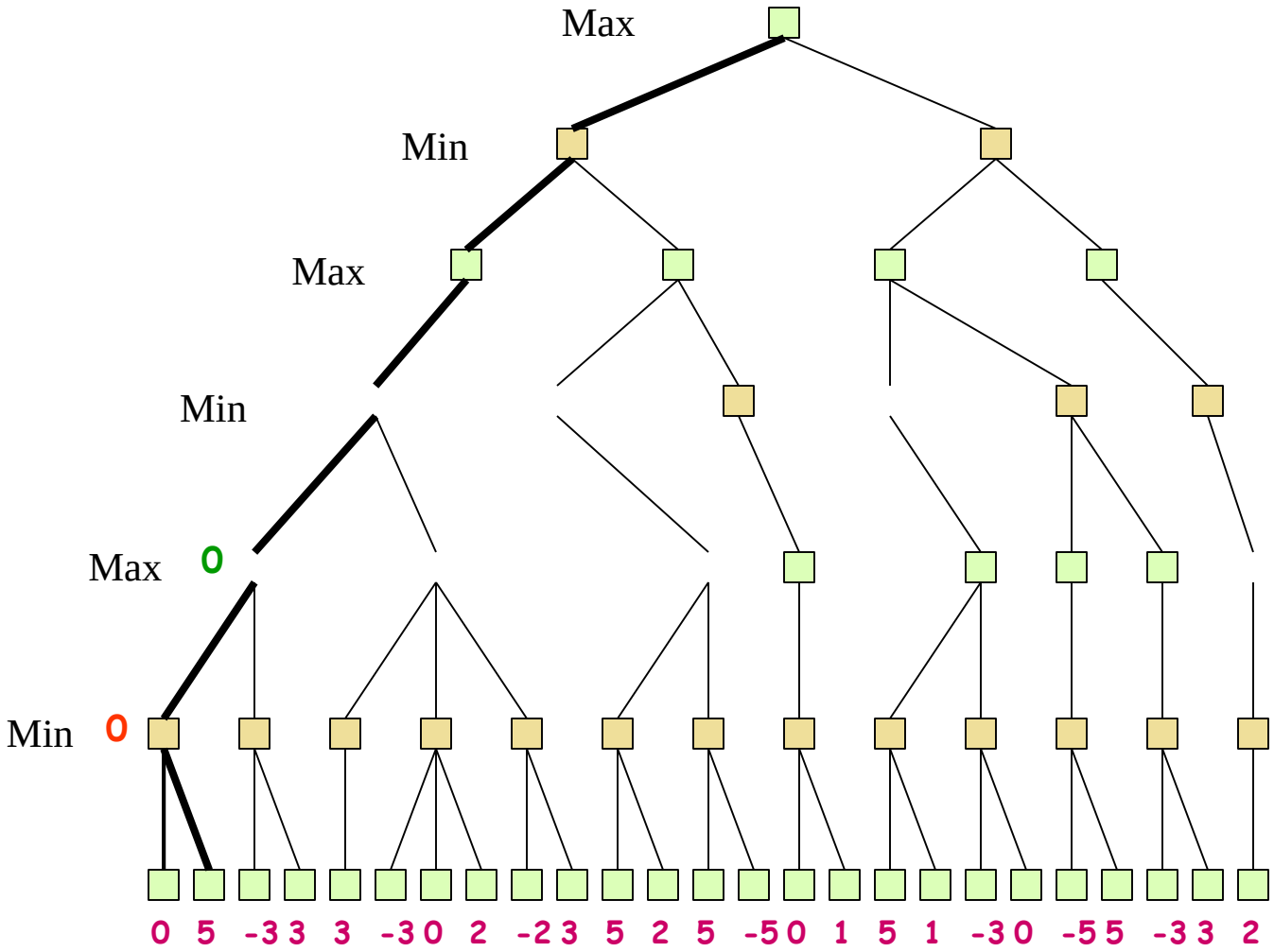
β best choice for Min 3

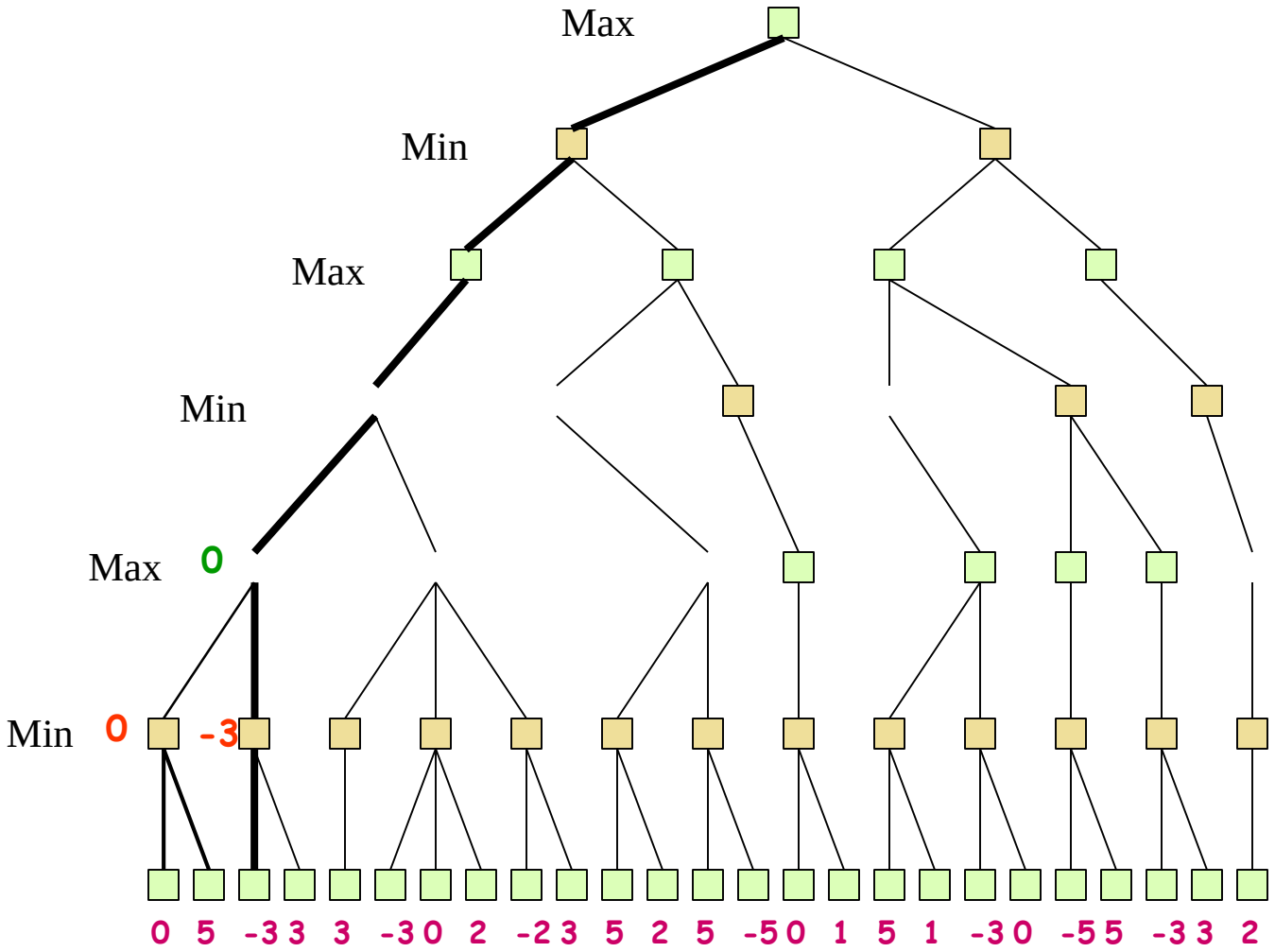
- Min could continue searching this sub-tree to see if there is a value that is less than the current worst alternative in order to give Max as few choices as possible
- this depends on the specific implementation
- Max knows the best value for its sub-tree

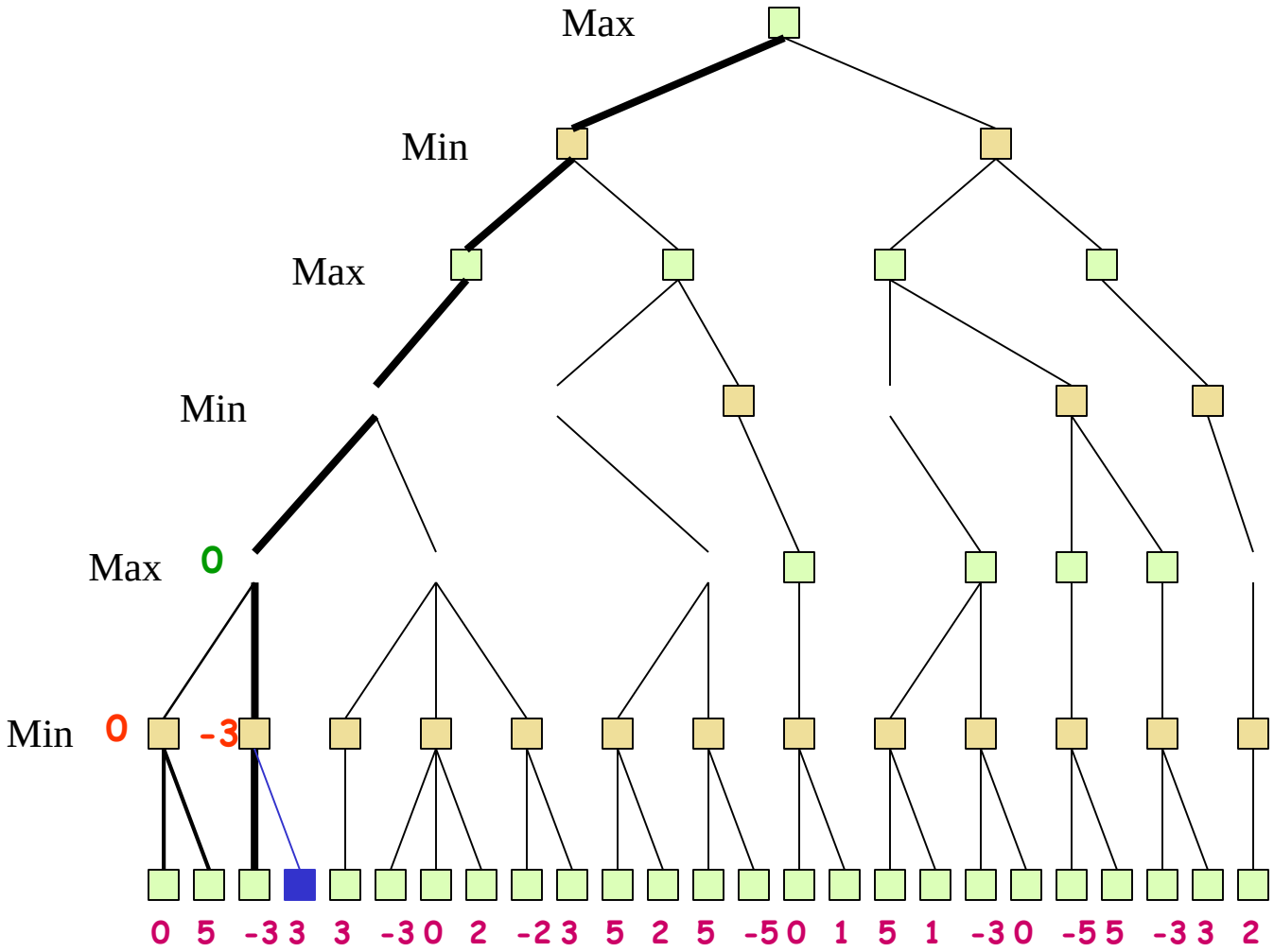
Alpha-Beta Tic-Tac-Toe Example 2

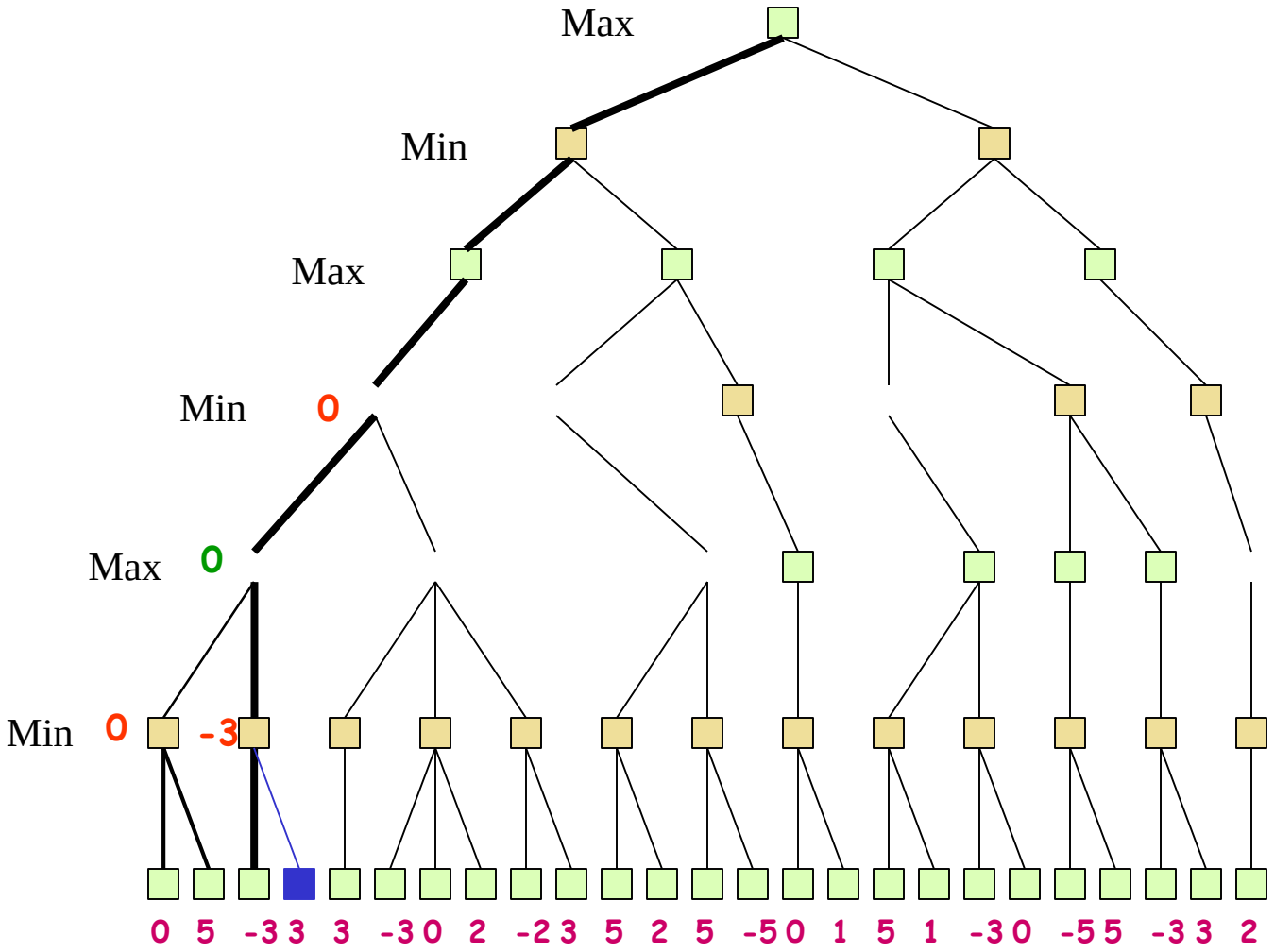


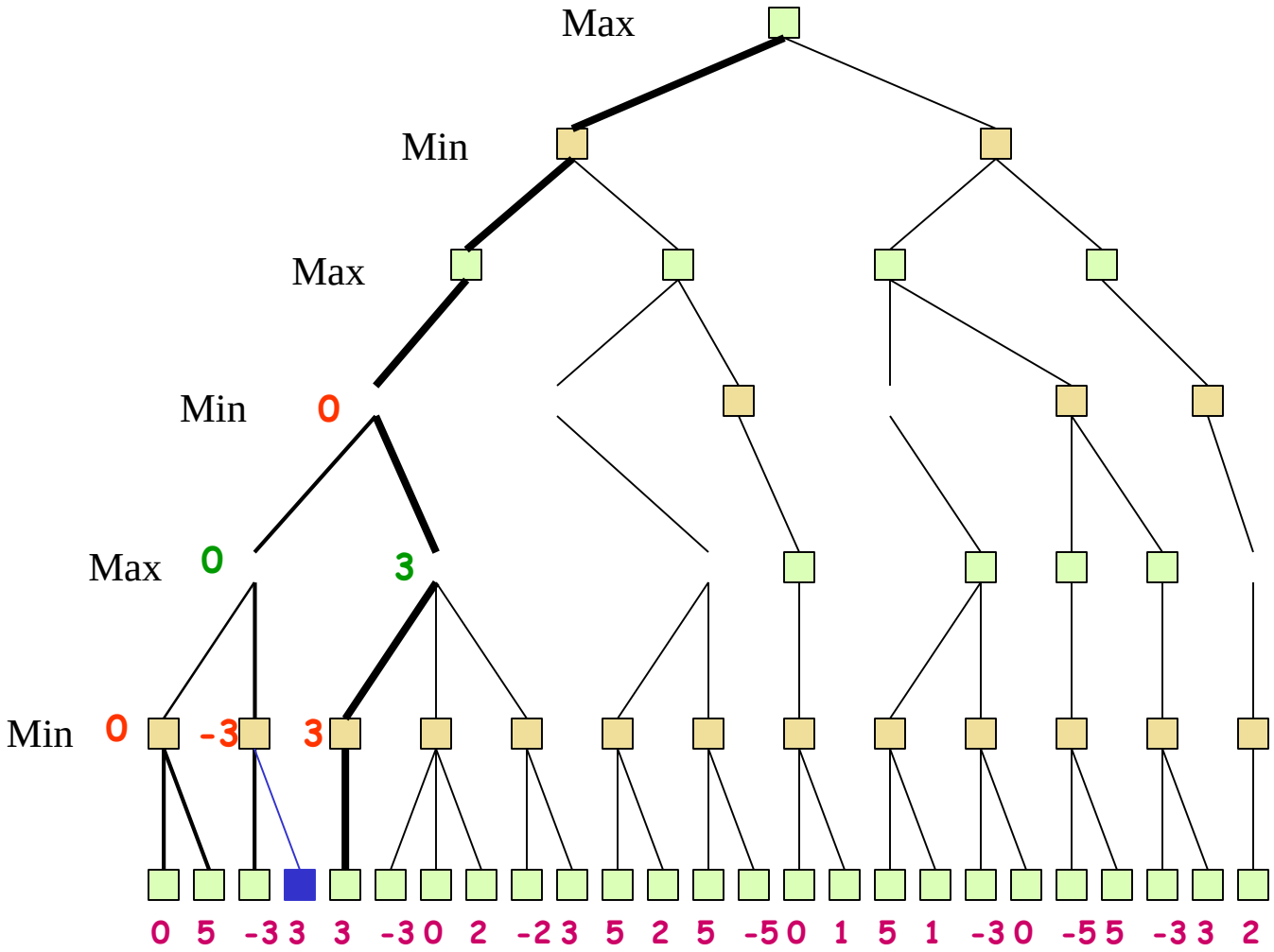

Max
Min
Max
Min
Max
Min
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

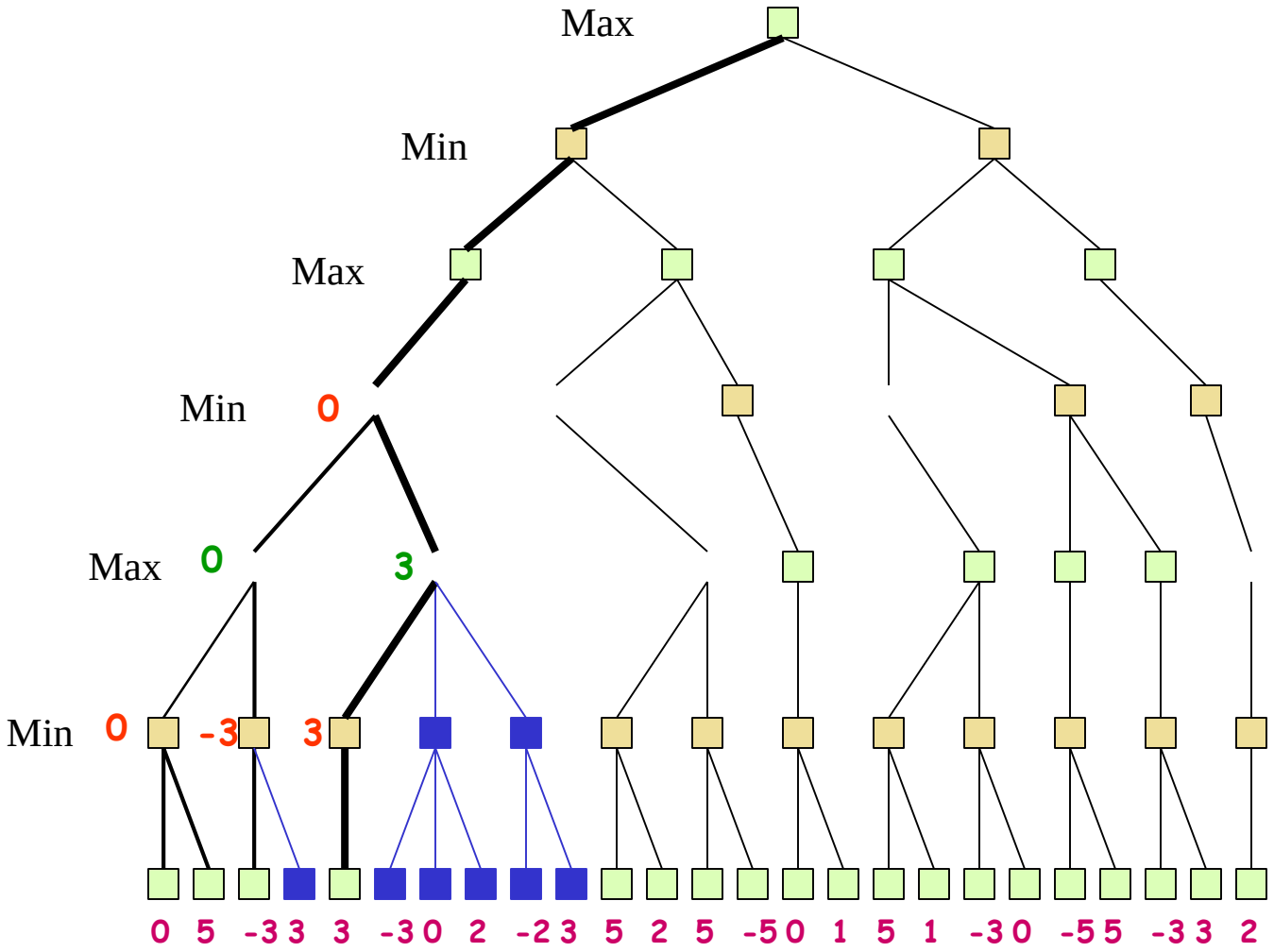

Max
Min
Max
Min
Max 0
Min 0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

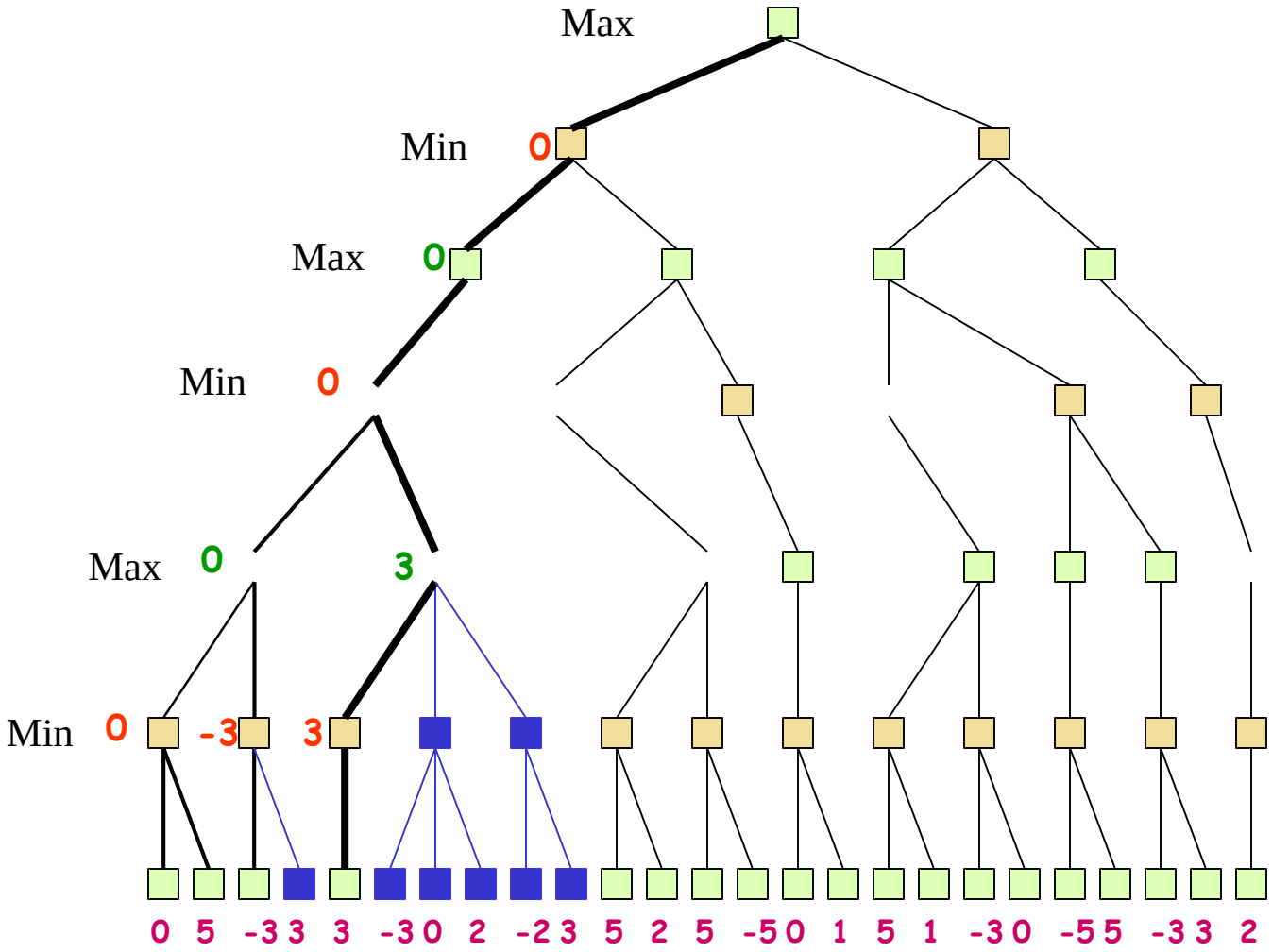

Max
Min
Max
Min
Max 0
Min 0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

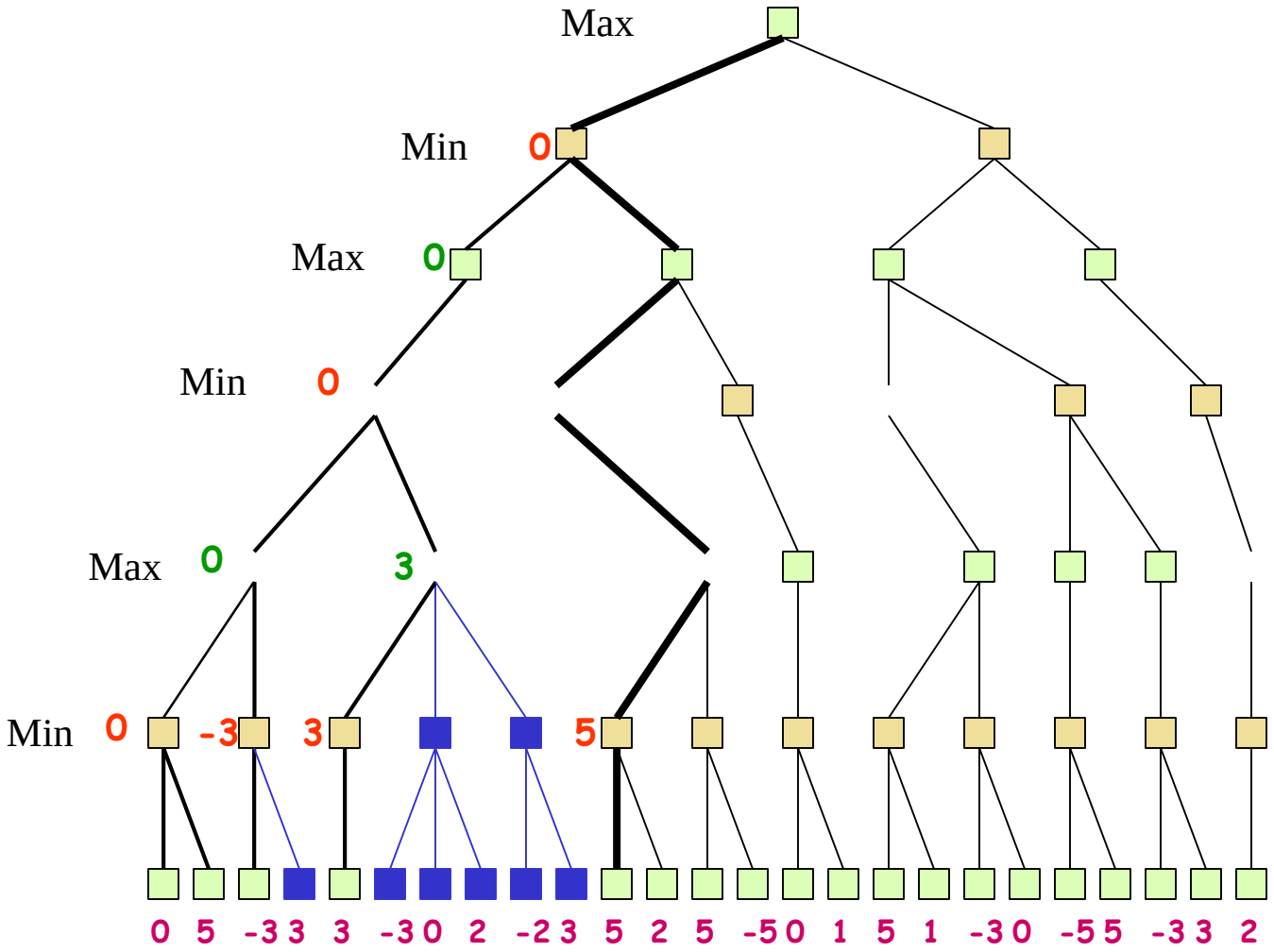

Max
Min
Max
Min
Max 0
Min 0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

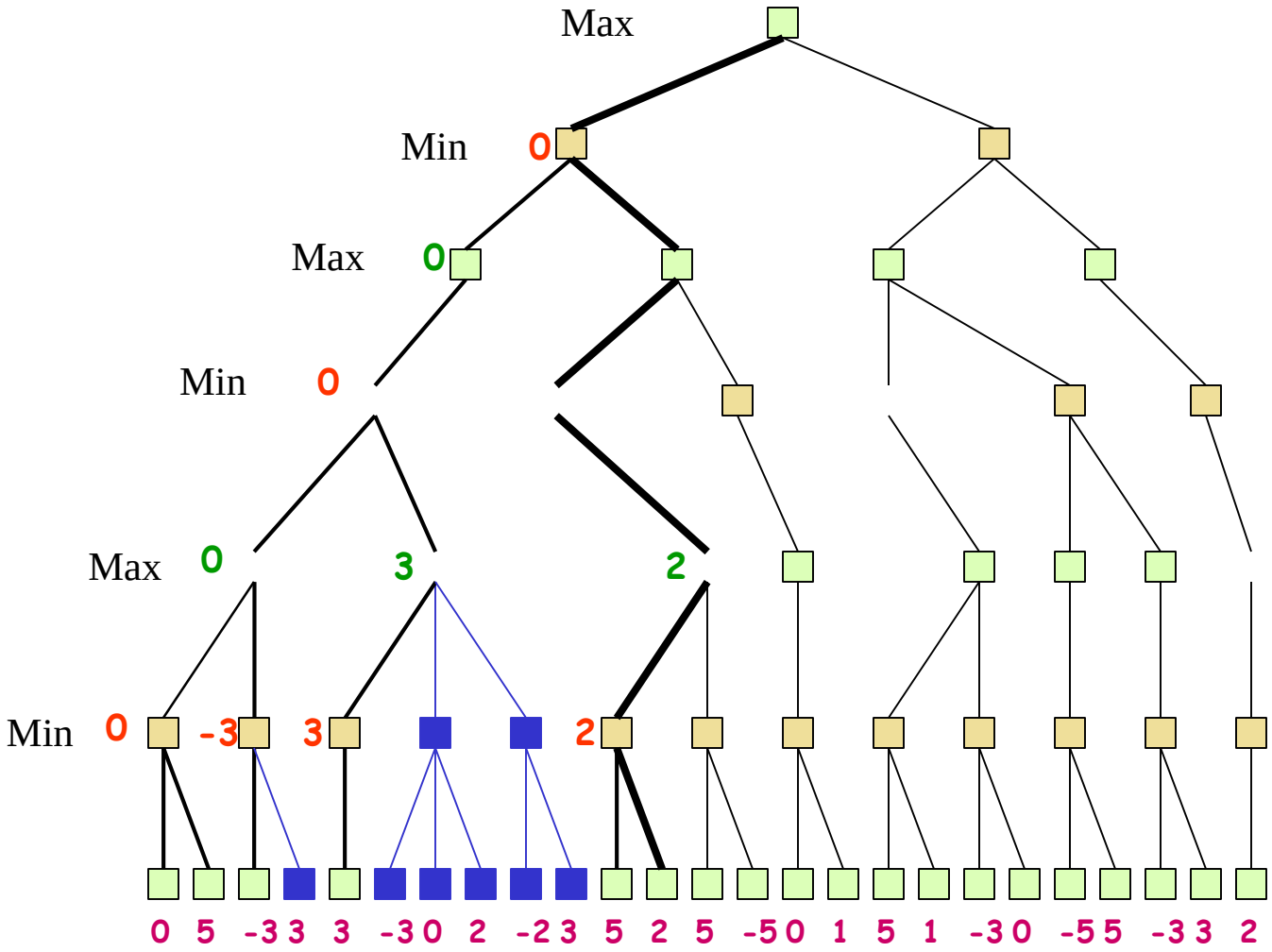

Max
Min
Max
Min 0
Max 0
Min 0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

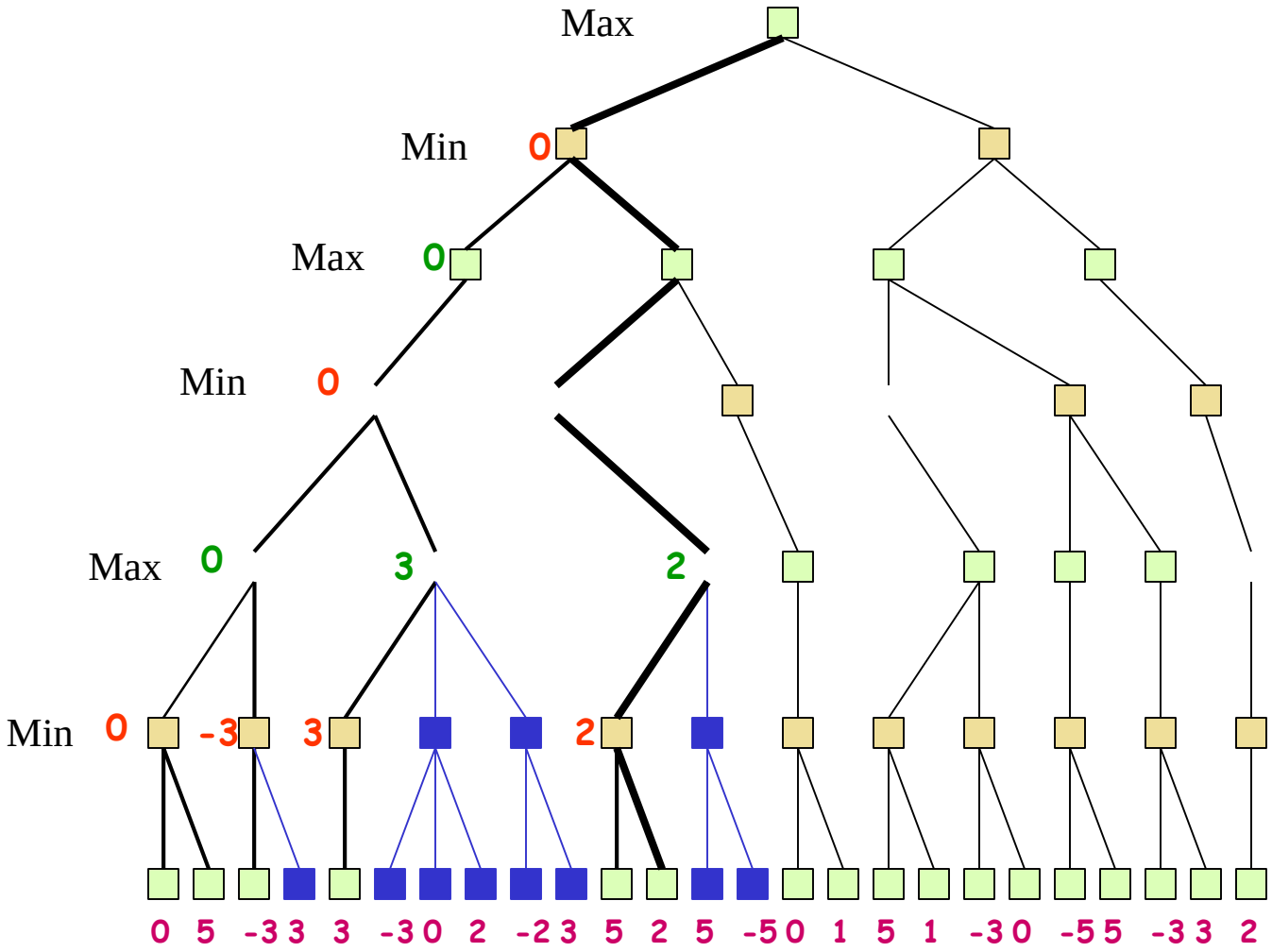

Max
Min
Max
Min 0
Max 0 3
Min 0 -3 3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

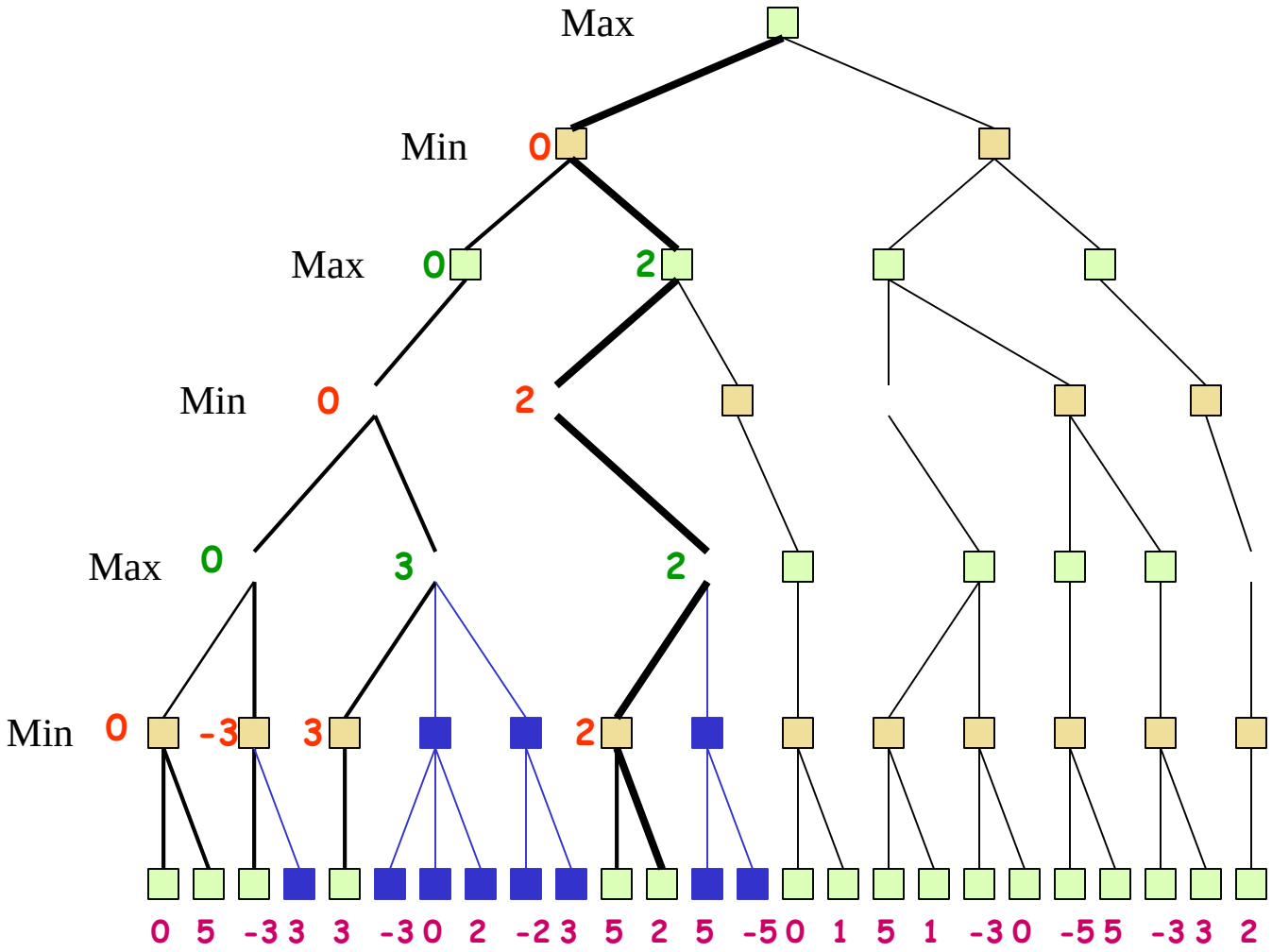

Max
Min
Max
Min 0
Max 0 3
Min 0 -3 3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

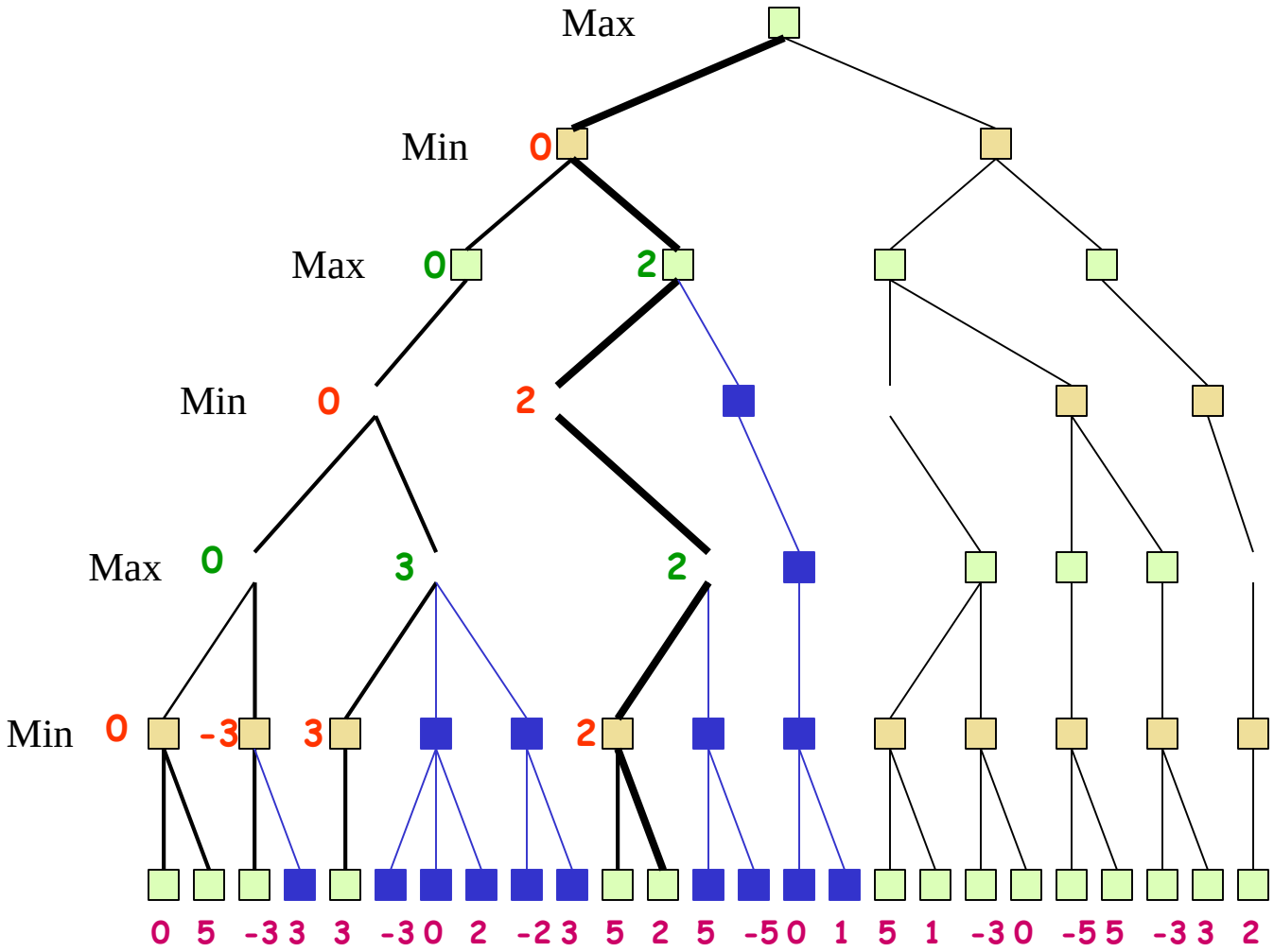

Max
Min
0
Max
0
Min
0
Max
0
3
Min
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

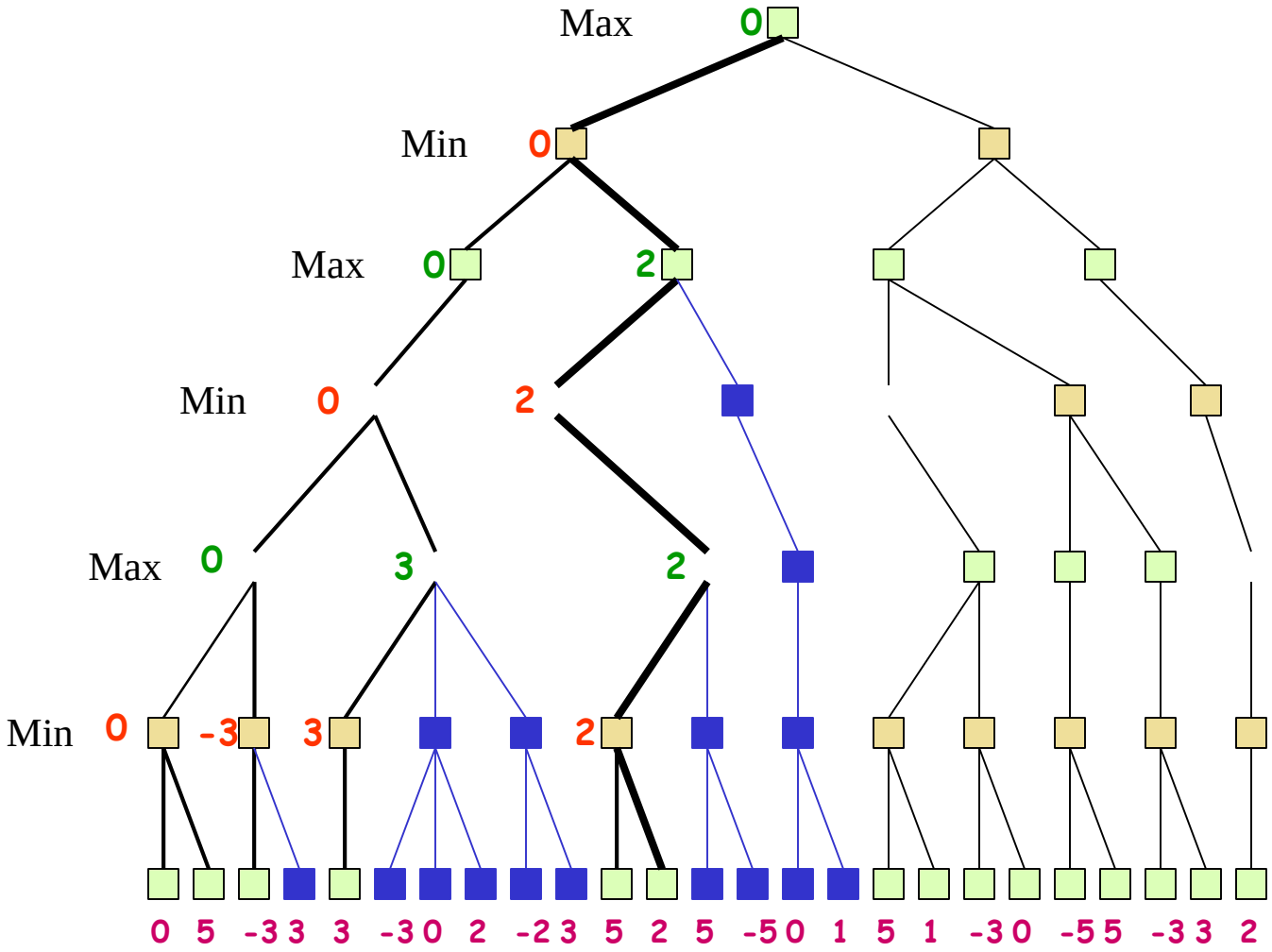

Max
Min 0
Max 0
Min 0
Max 0 3
Min 0 -3 3 5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

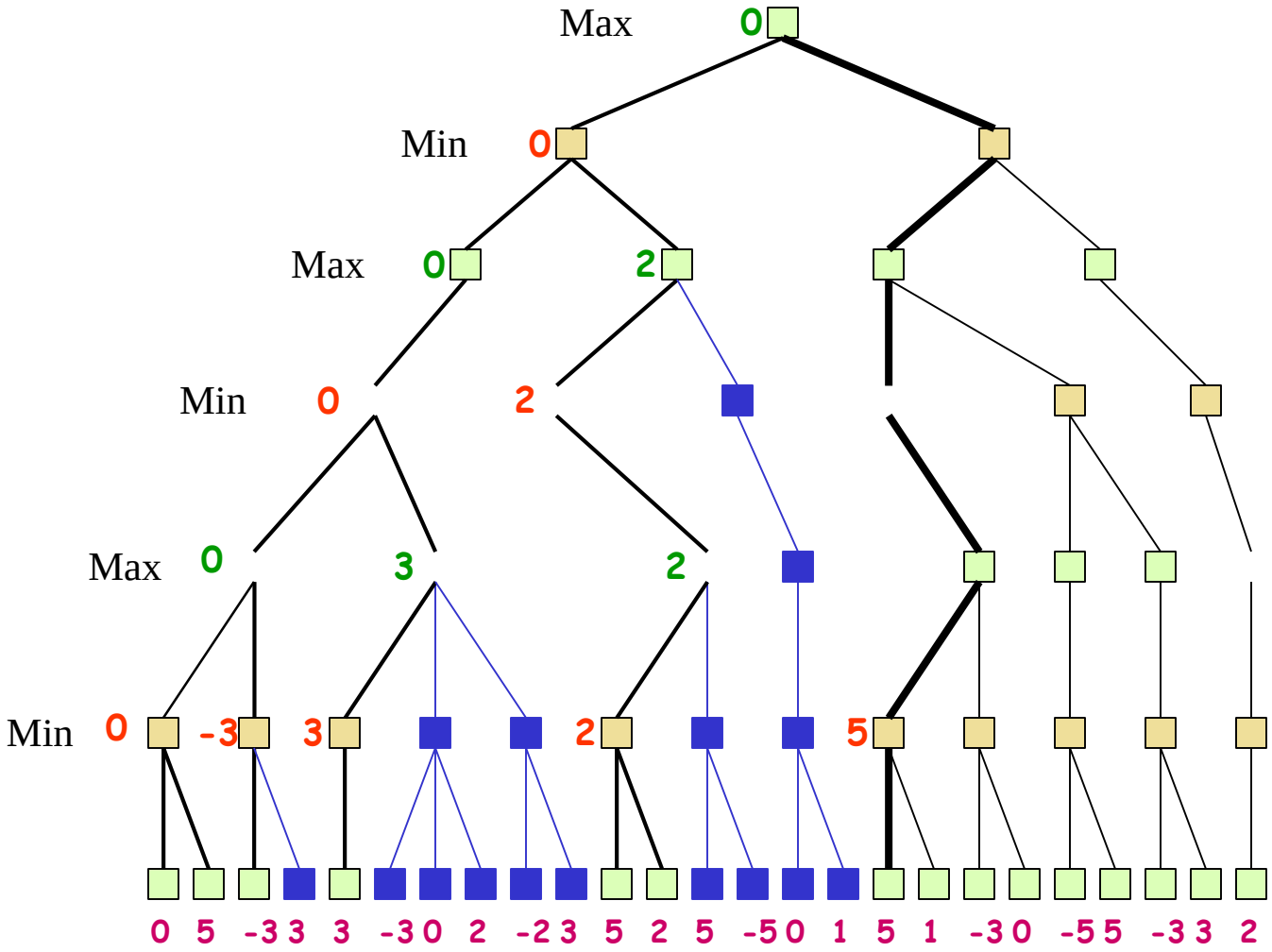

Max
Min
0
Max
0
Min
0
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

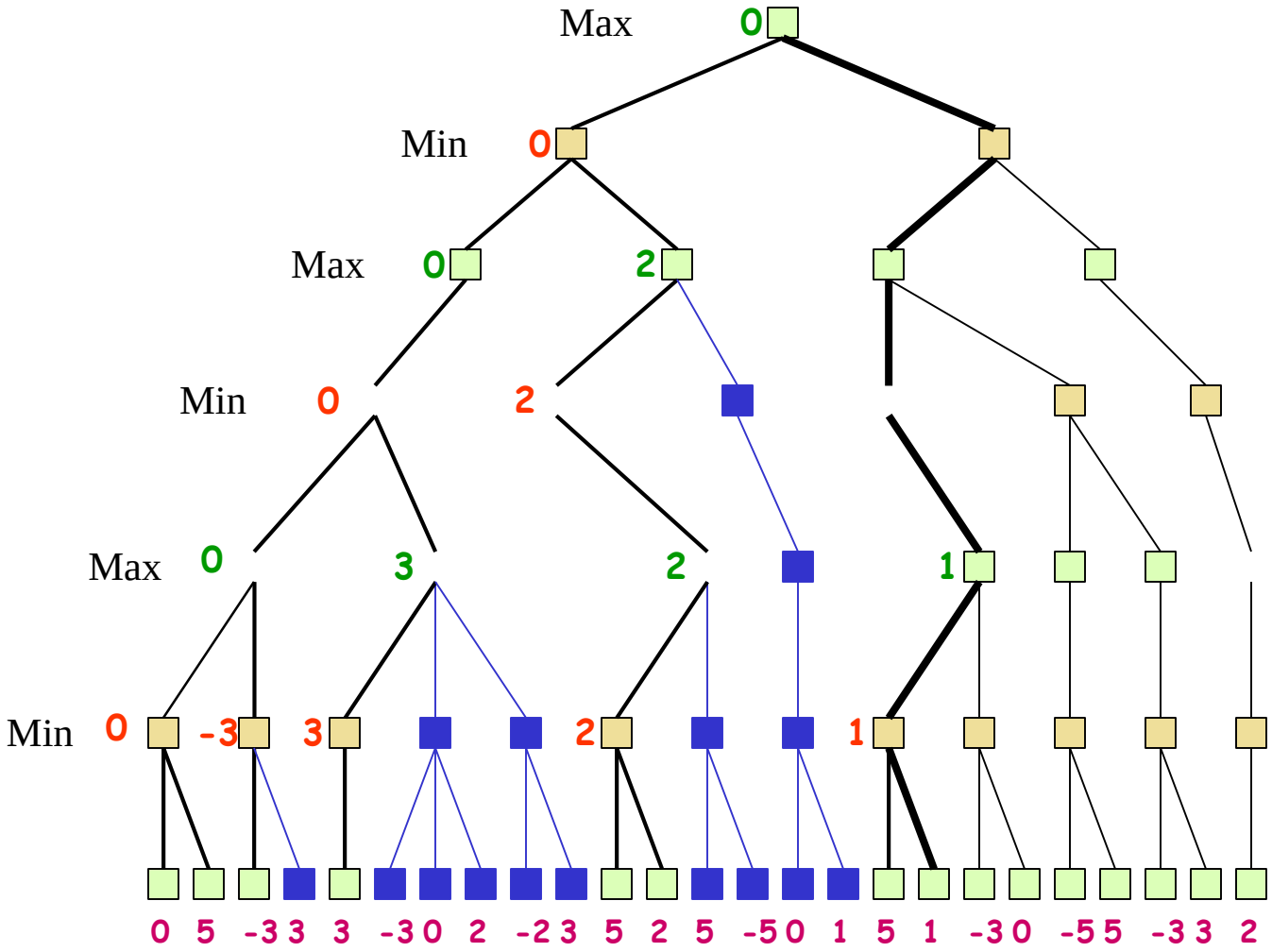

Max
Min 0
Max 0
Min 0
Max 0 3 2
Min 0 -3 3 2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

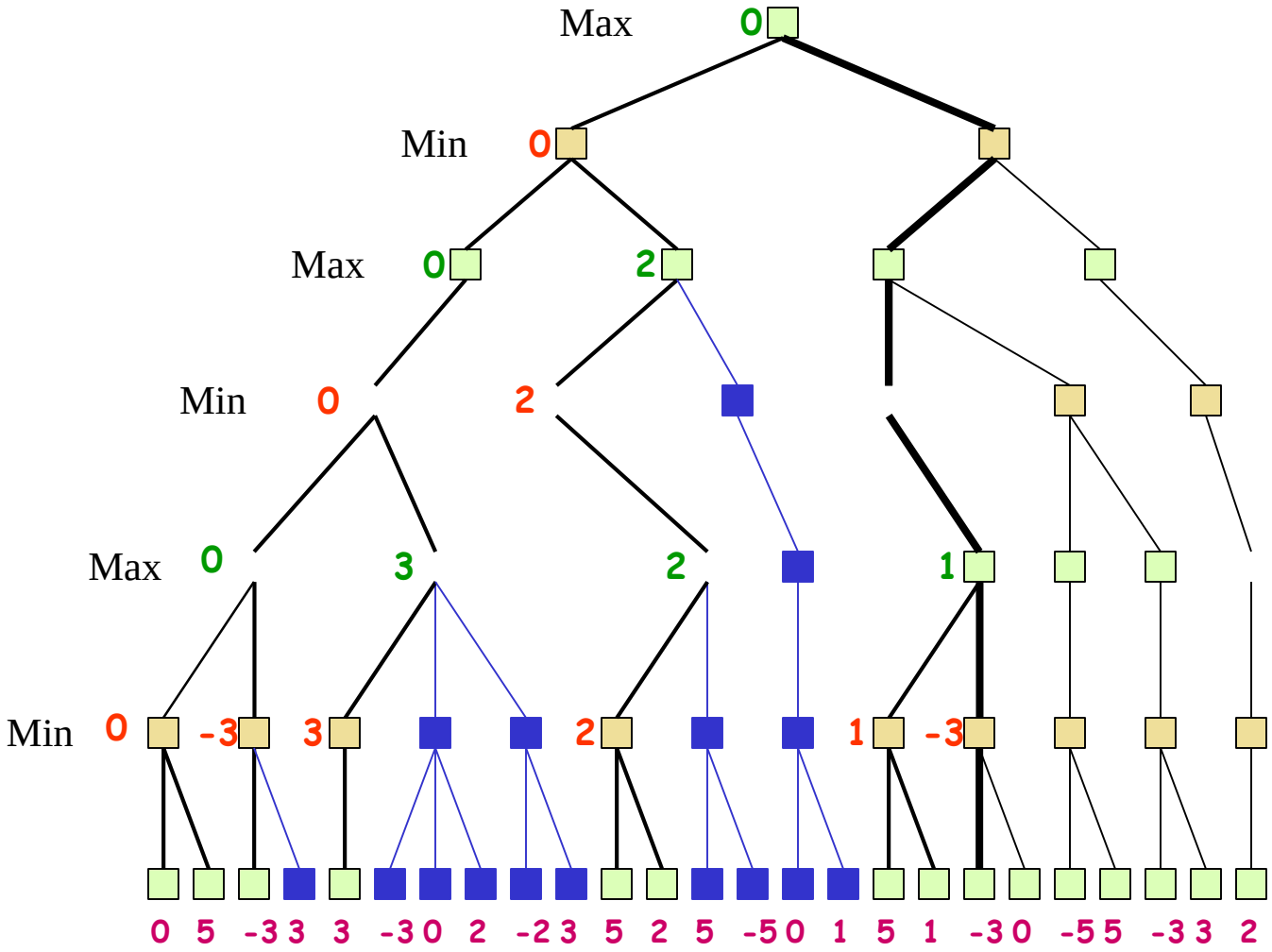

Max
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

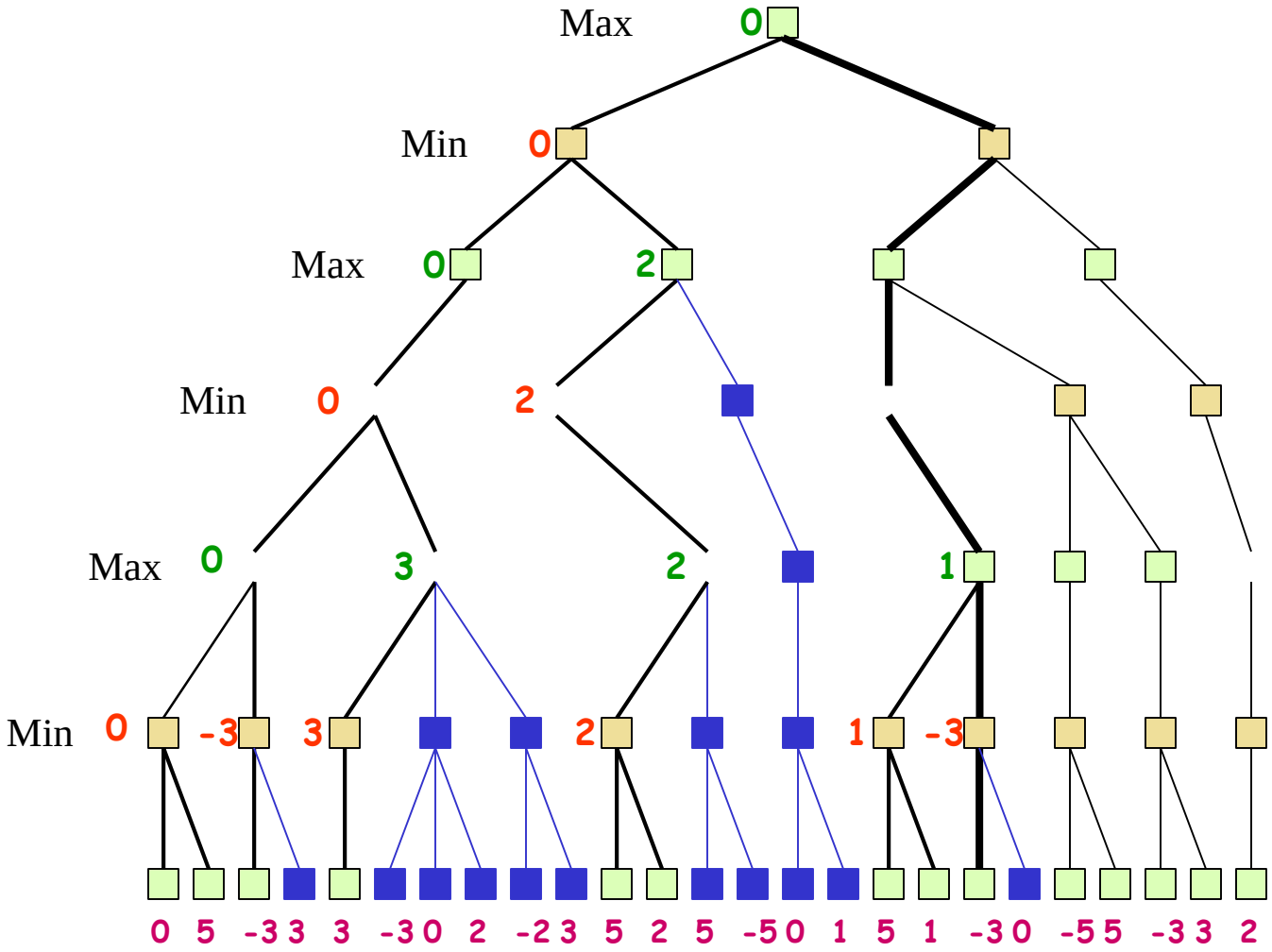

Max
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

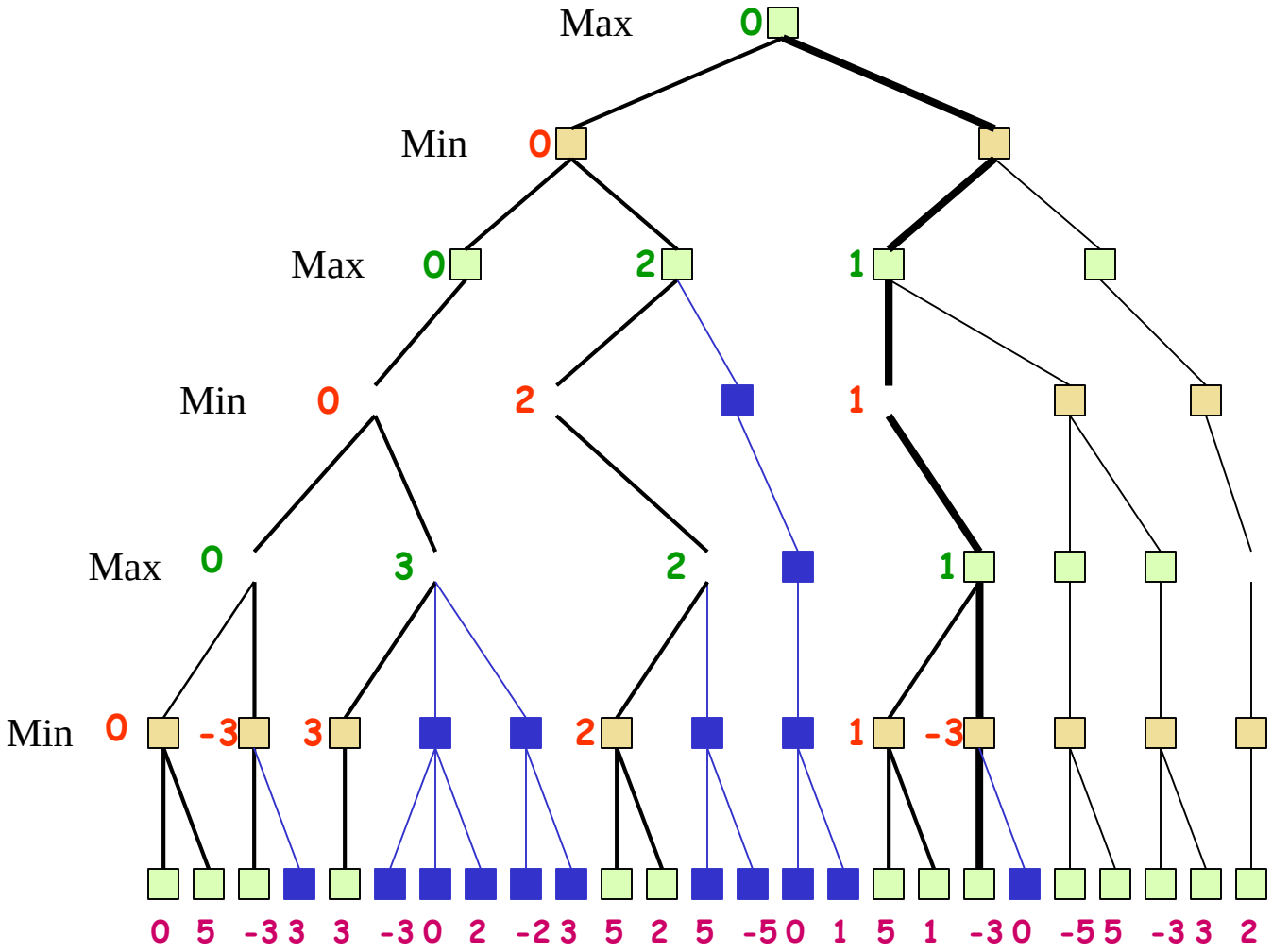

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

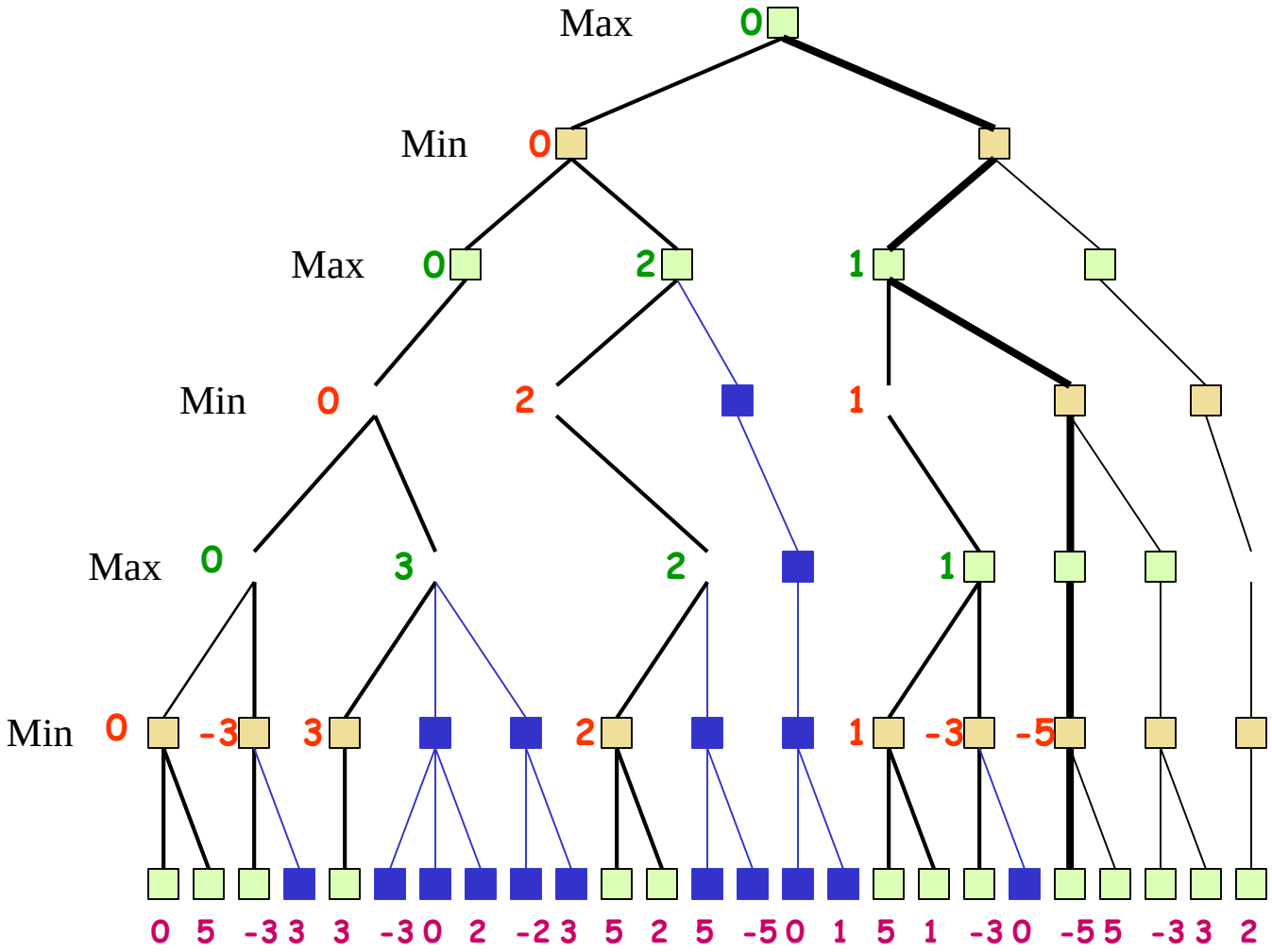

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
Min
0
-3
3
2
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

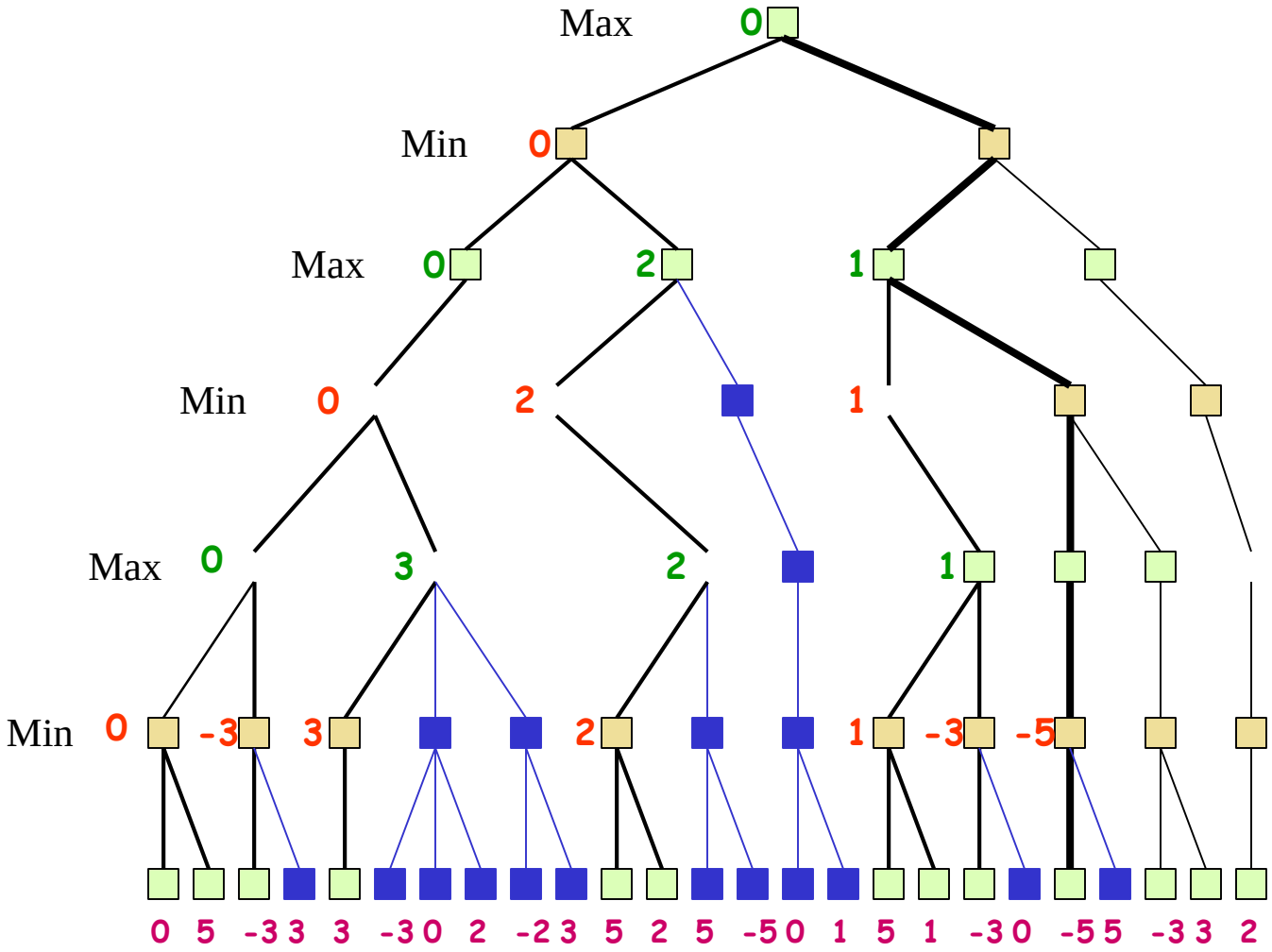

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
1
Min
0
-3
3
2
1
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

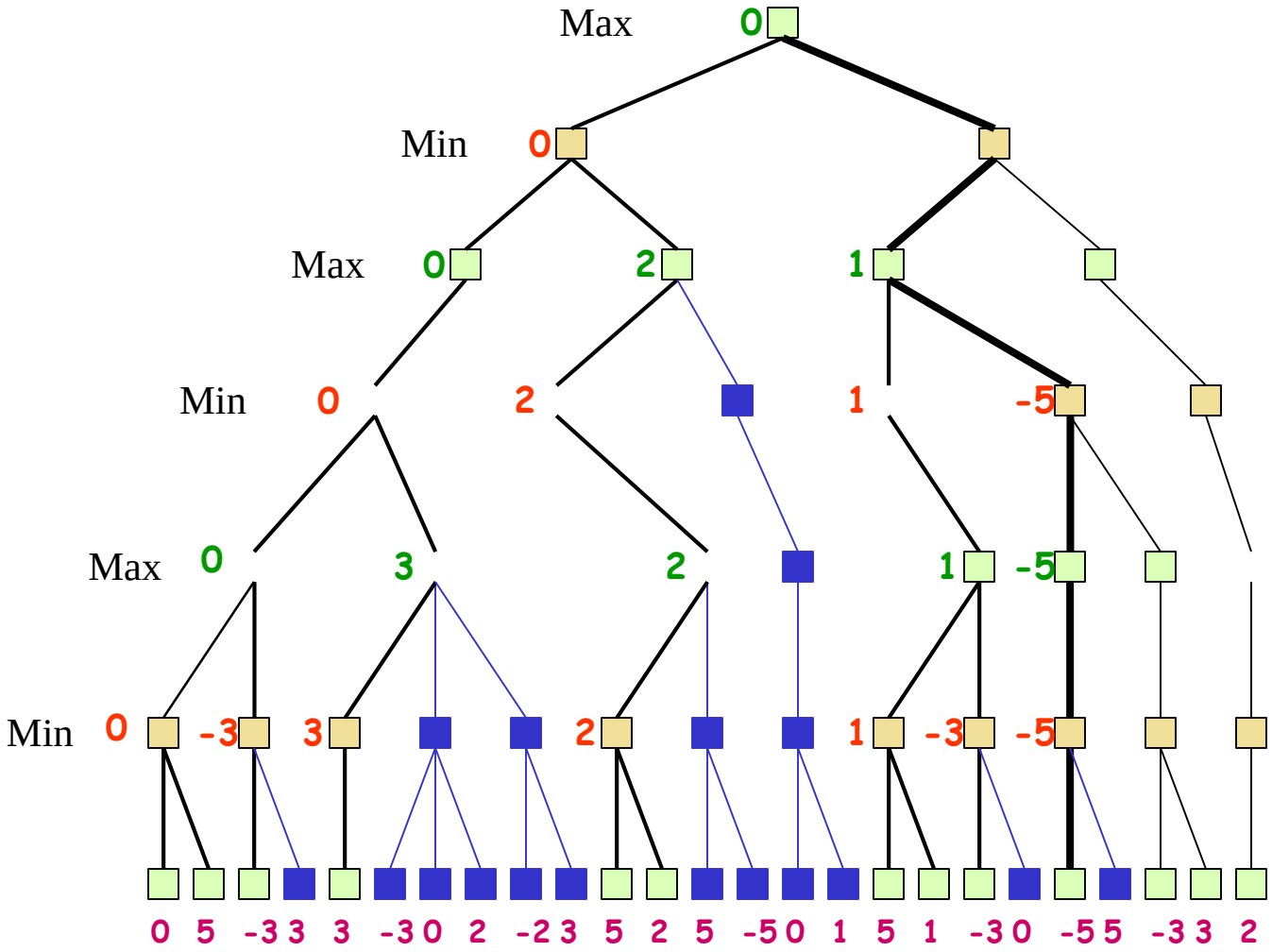

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
1
Min
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

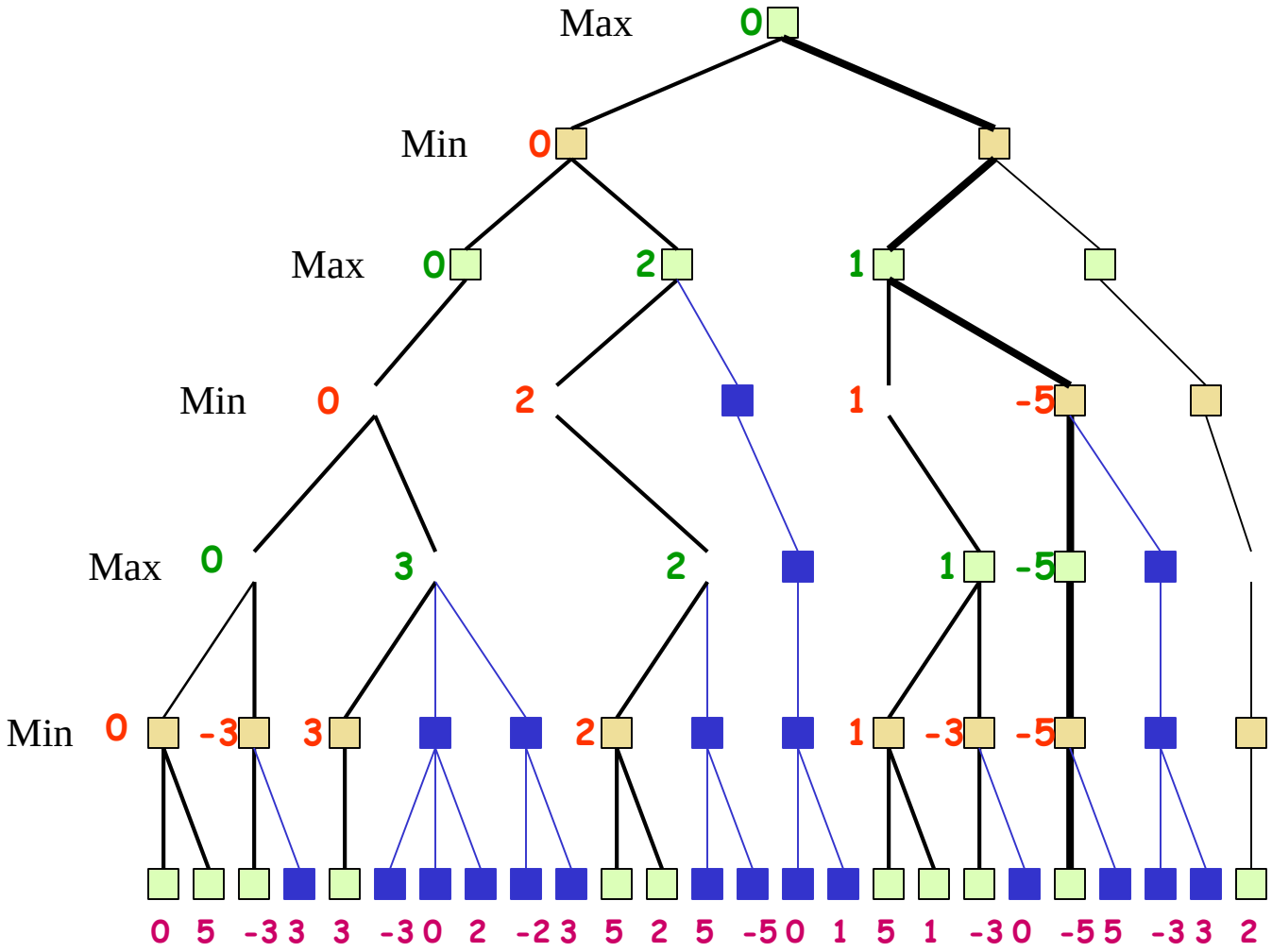

Max
0
Min
0
Max
0
2
Min
0
2
Max
0
3
2
1
Min
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

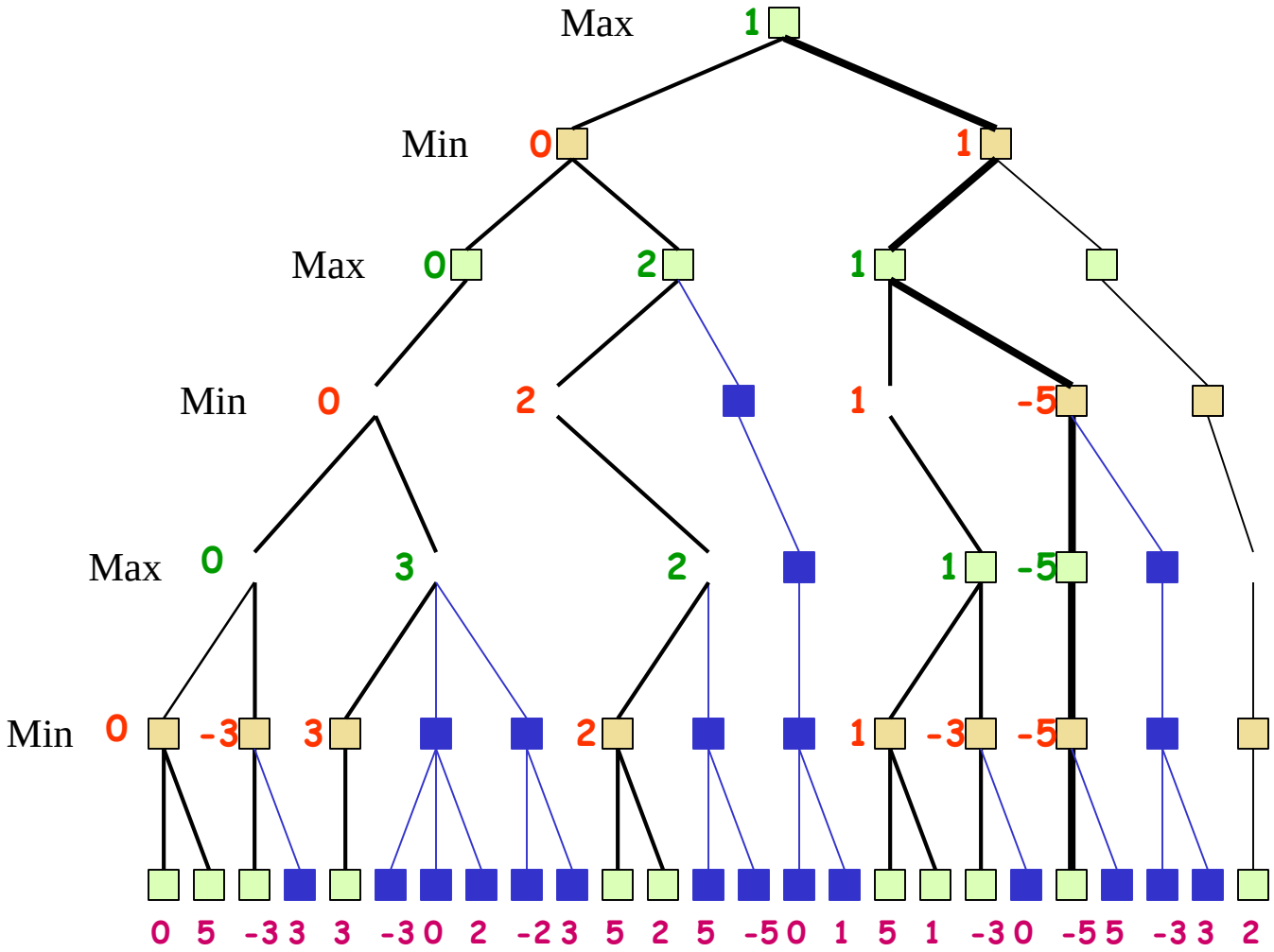

Max
0
Min
0
Max
0
2
1
Min
0
2
1
Max
0
3
2
1
Min
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

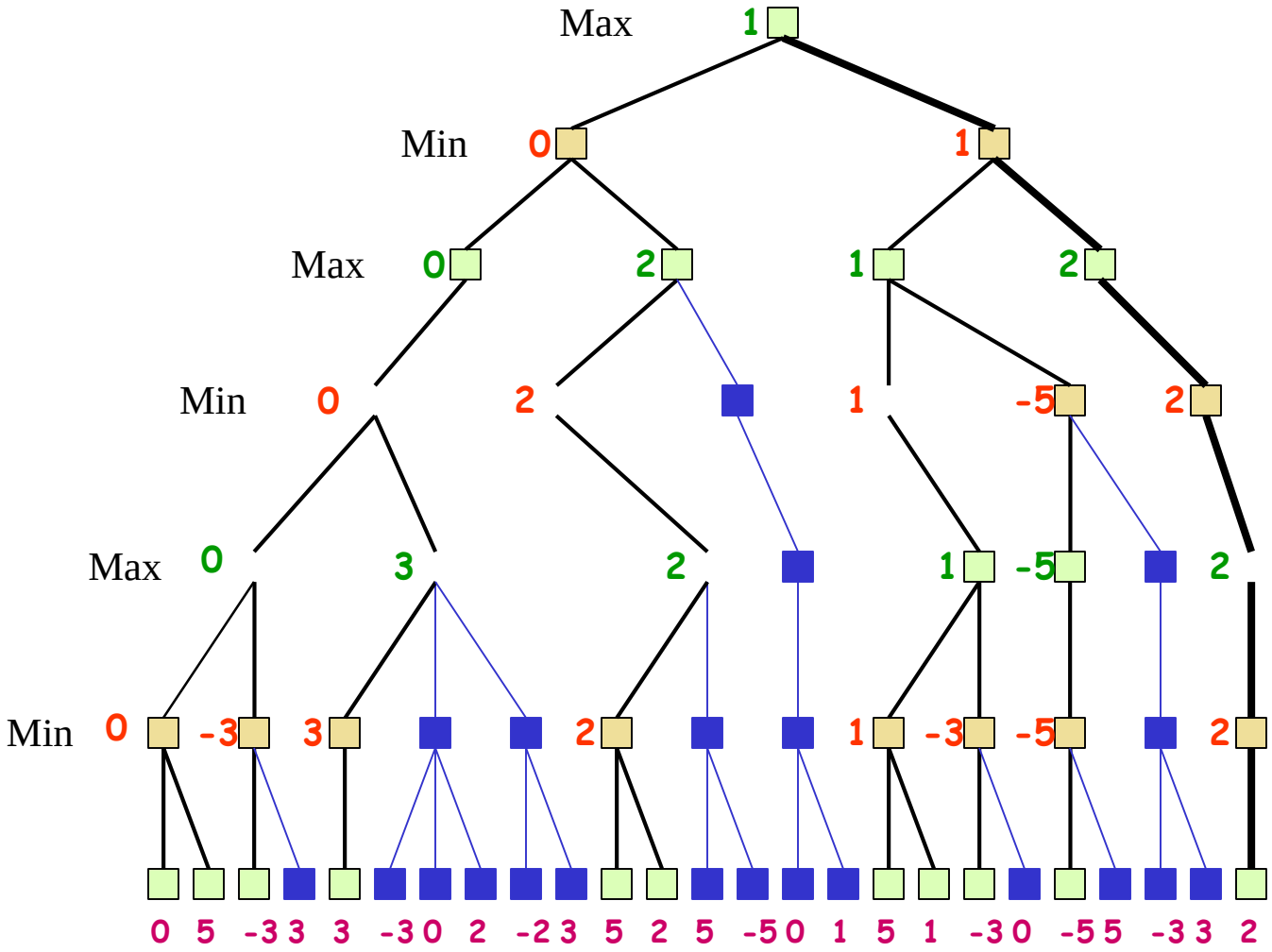

Max
0
Min
0
Max
0
2
1
Min
0
2
1
Max
0
3
2
1
Min
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

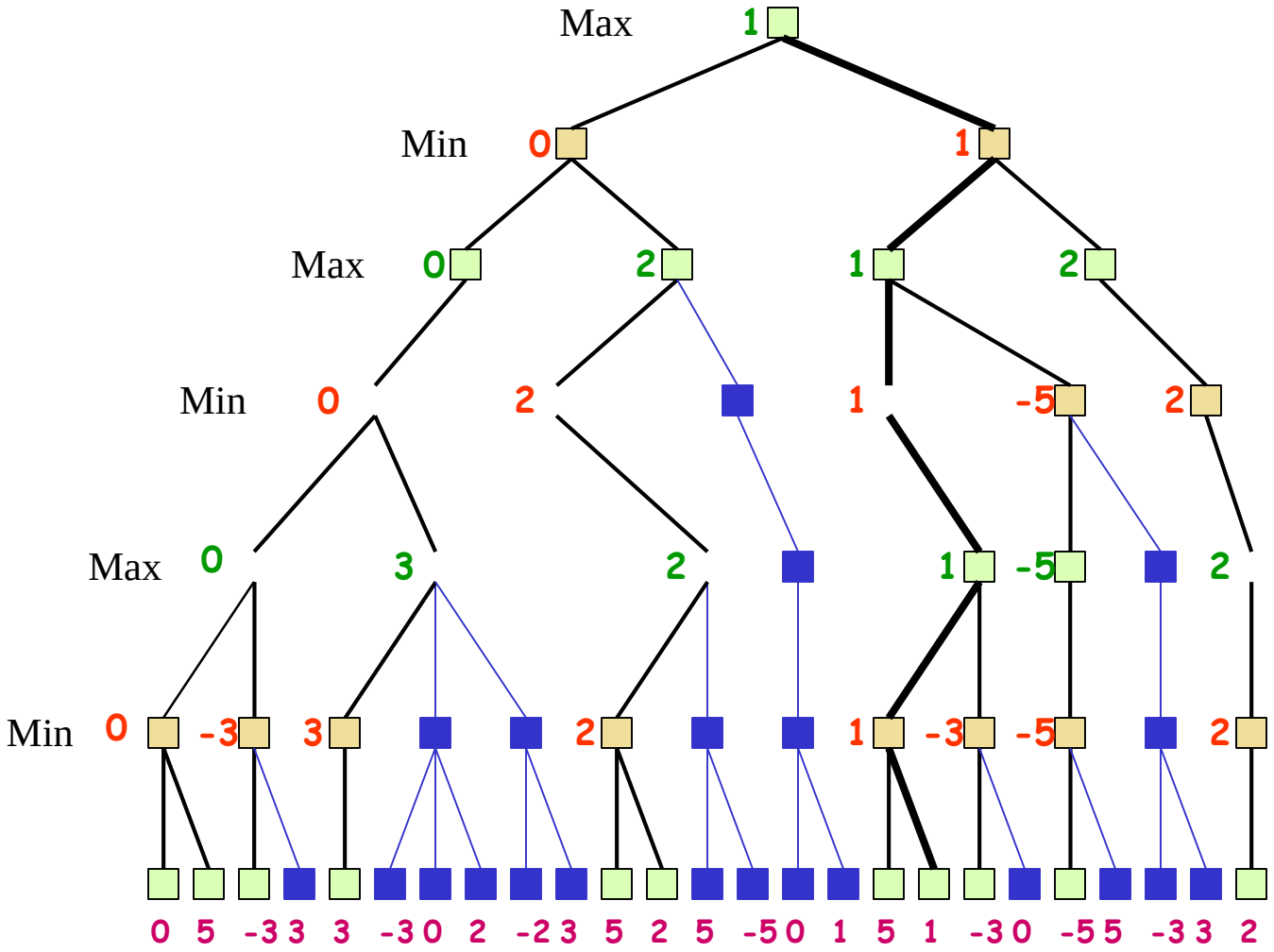

Max
0
Min
0
Max
0
2
1
Min
0
2
1
Max
0
3
2
1
Min
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

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

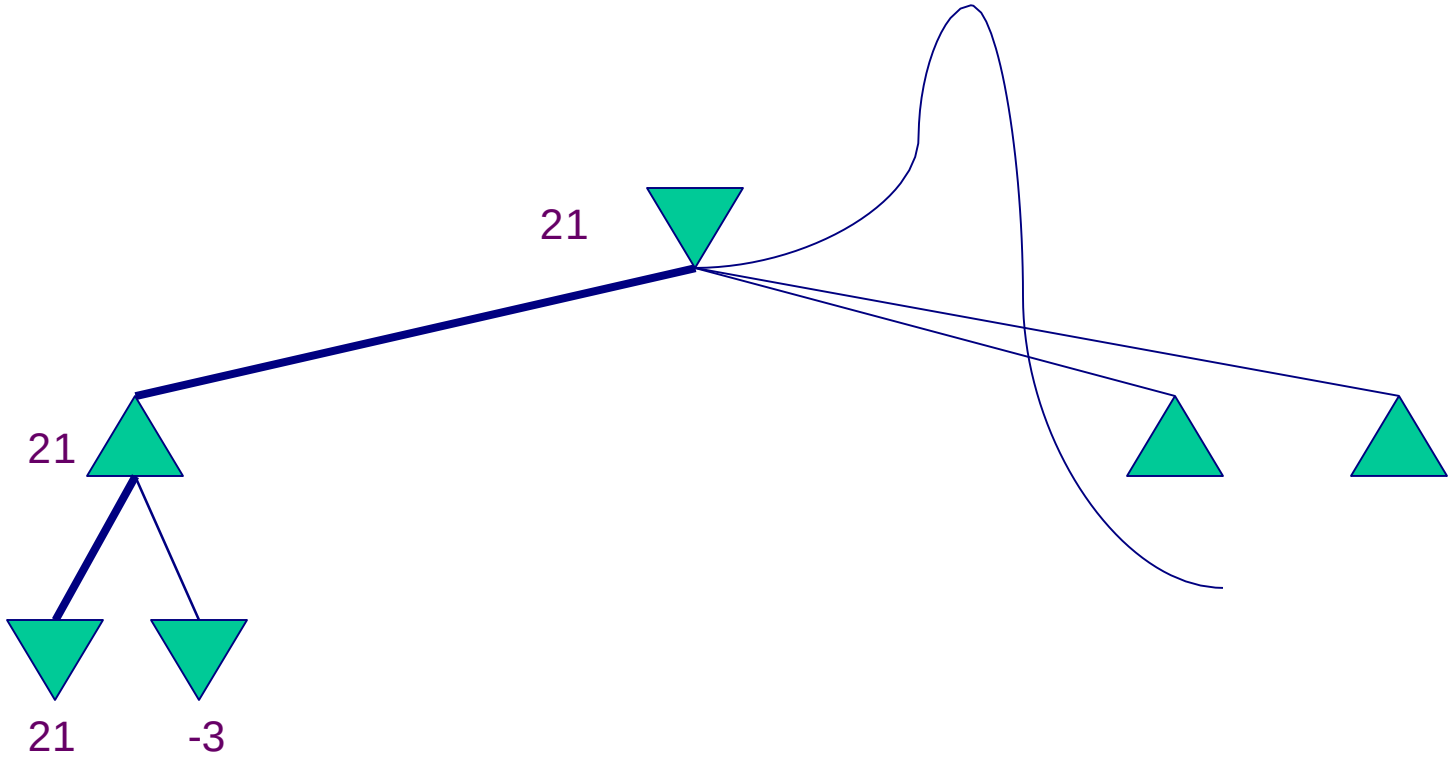
Pruning at MIN node

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

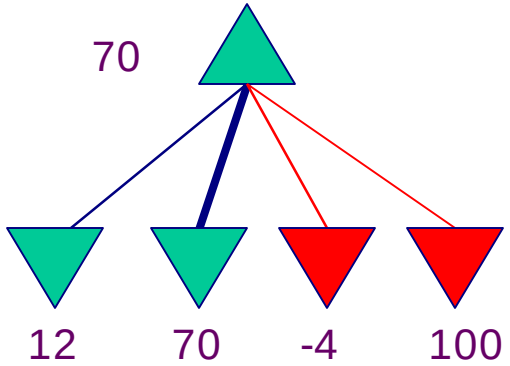
If all children are evaluated without pruning, return the MIN of their values

Idea of α-β Pruning



We know β on this path is 21

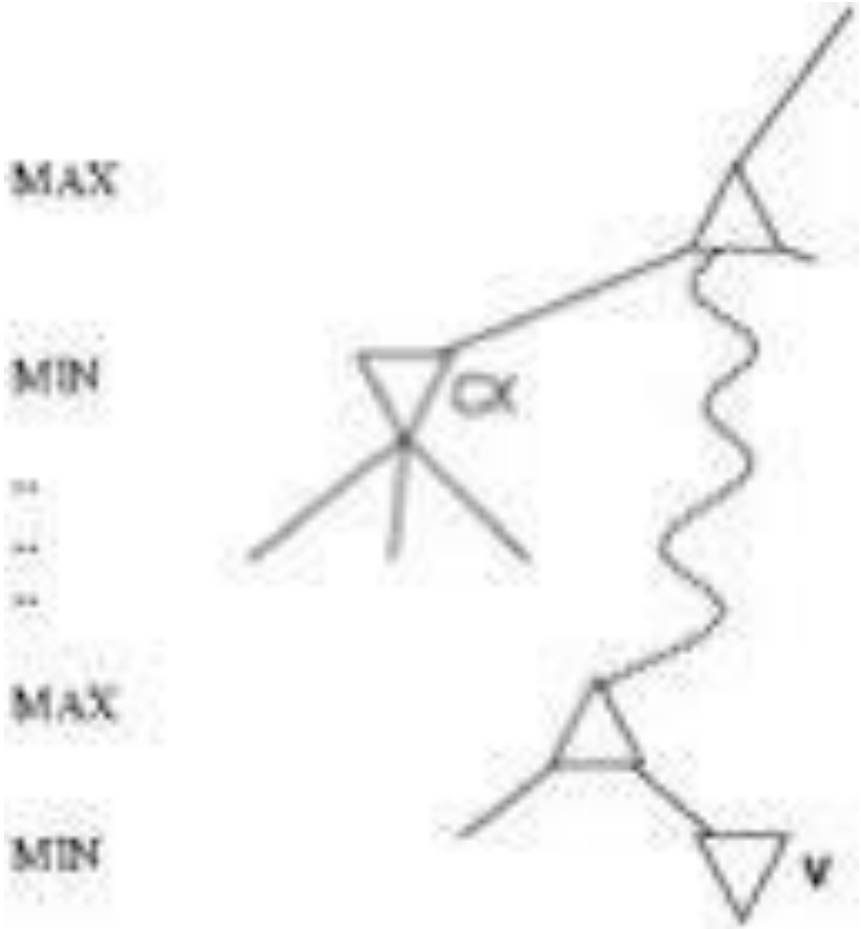
So, when we get max=70, we know this will never be used, so we can stop here



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*



Alpha–Beta Search Algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

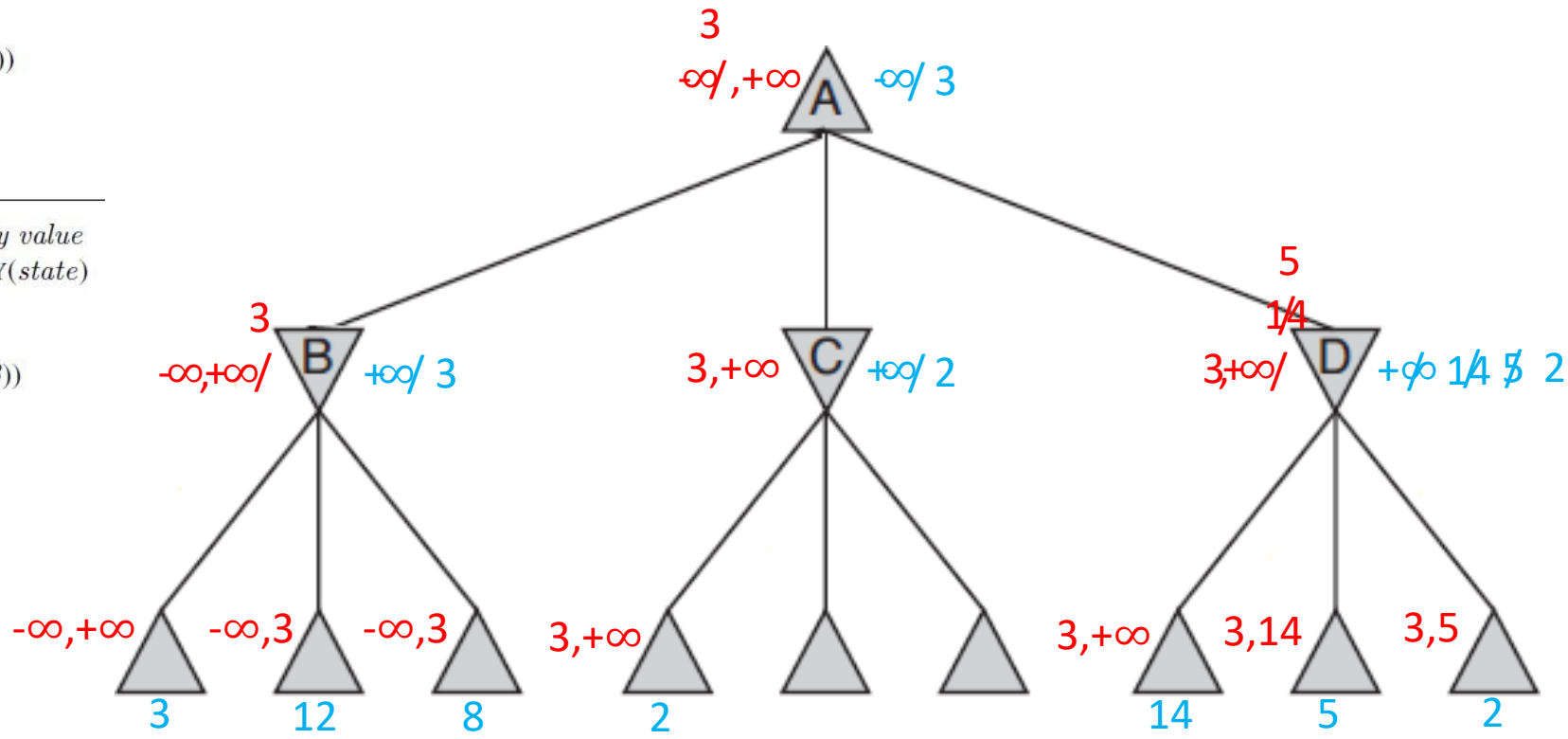
```
function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Alpha–Beta Search Algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```



Properties of Alpha–Beta Search Algorithm

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With *perfect ordering*, **time complexity = $O(b^{m/2})$** ➔ doubles solvable depth
- Unfortunately, 35^{50} is still impossible! (for chess)