Adversarial Search and Games

Chapter 5

Some material adopted from notes by Charles R. Dyer, U of Wisconsin-Madison

Why study games?

- Interesting, hard problems requiring minimal "initial structure"
- Clear criteria for success
- Study problems involving {hostile, adversarial, competing, cooperating} agents and uncertainty of interacting with the natural world
- People have used them to assess their intelligence
- Fun, good, easy to understand, PR potential
- Games often define very large search spaces, e.g.

  chess 35^{100} nodes in search tree, 10^{40} legal states

50 years of Computer chess history

- **1948**: Norbert Wiener describes how chess program can work using minimax search with an evaluation function
- **1950**: Claude Shannon publishes Programming a Computer for Playing Chess
- **1951**: Alan Turing develops *on paper* 1st program capable of playing full chess games (Turochamp)
- **1958**: first program plays full game on IBM 704 (loses)
- **1962**: Kotok & McCarthy (MIT) 1st program to play credibly
- **1967**: Greenblatt's Mac Hack Six (MIT) defeats a person in regular chess tournament play
- **1997**: IBM's Deep Blue beats world champ Gary Kasparov

State of the art

- **1979 Backgammon:** BKG (CMU) tops world champ
- **1994 Checkers**: Chinook is the world champion
- **1997 Chess**: IBM Deep Blue beat Gary Kasparov
- **2007 Checkers:** solved (it's a draw)
- **2016 Go**: AlphaGo beat champion Lee Sedol
- **2017 Poker:** CMU's Libratus won $1.5M from top poker players in a casino challenge
- **20?? Bridge**: Expert bridge programs exist, but no world champions yet

1997

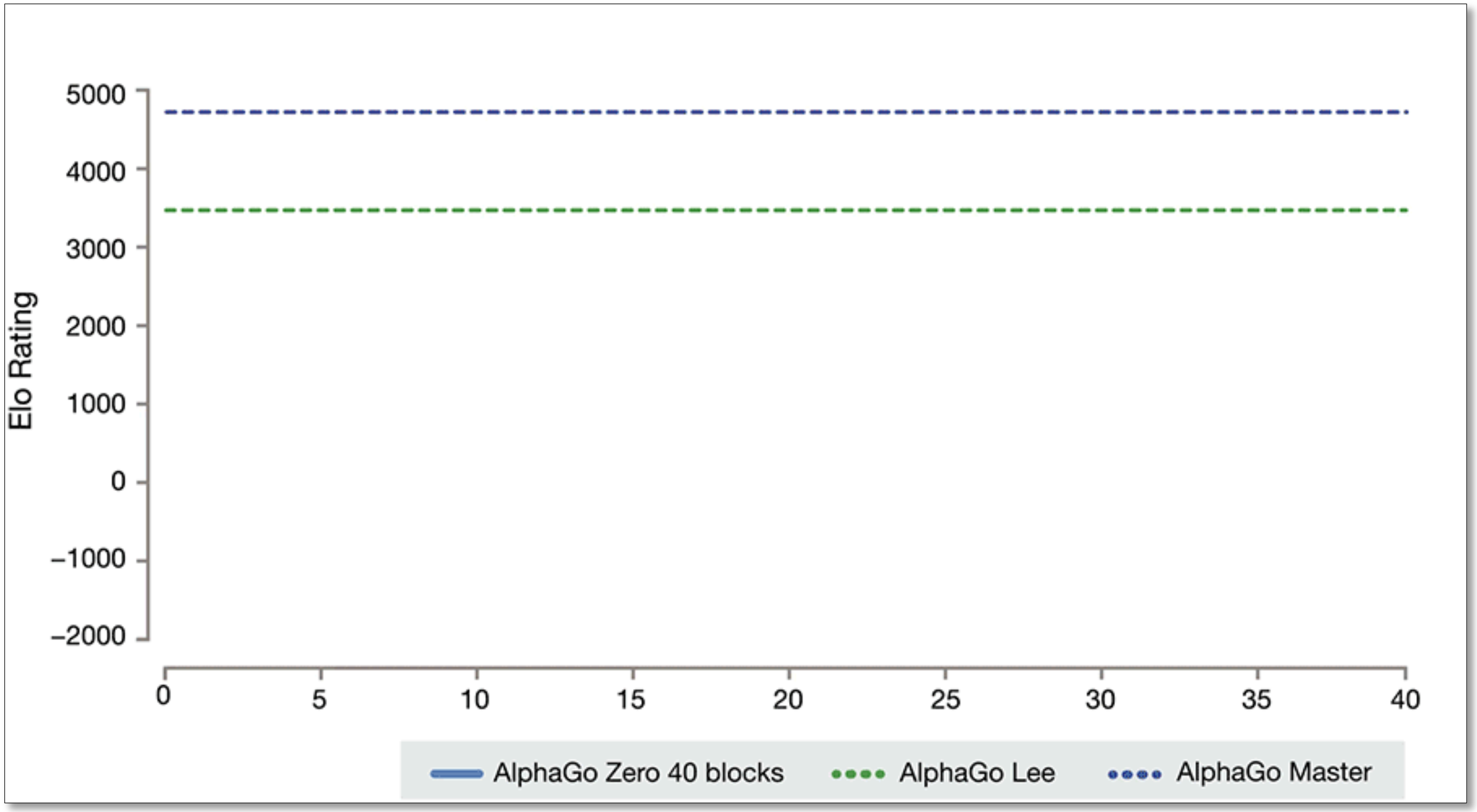
Chess Grand Master Garry Kasparov, left, comtemplates his next move against IBM's Deep Blue chess computer while Chung-Jen Tan, manager of the Deep Blue project looks on iduring the first game of a six-game rematch between Kasparov and Deep Blue in this file photo from 1997. The computer program made history by becoming the first to beat a world chess champion, Kasparov, at a serious game. Photo: Adam Nadel/Associated Press

Google DeepMind
Challenge Match
8 - 15 March 2016
AlphaGo
AlphaGo
Lee Sedol
Google DeepMind
Challenge Match
8 - 15 March 2016

2016

AlphaGo Zero learns on its Own

AlphaGo Zero was not trained on human games, but used reinforcement learning while playing against itself



AlphaGo - The Movie

Highly recommended 2017 award-winning documentary, **free on** YouTube

New: Human beats AI by tricking it to Blunder

- Researchers trained their own AI "adversaries" to search for weaknesses in KataGo in 2023
  - KataGo is considered a state-of-the-art GO playing system
- See arXiv paper "Adversarial Policies Beat Superhuman Go AIs"

"Notably, our adversaries do not win by learning to play Go better than KataGo – in fact, our adversaries are easily beaten by human amateurs. Instead, our adversaries win by tricking KataGo into making serious blunders. … Our results demonstrate that even superhuman AI systems may harbor surprising failure modes."

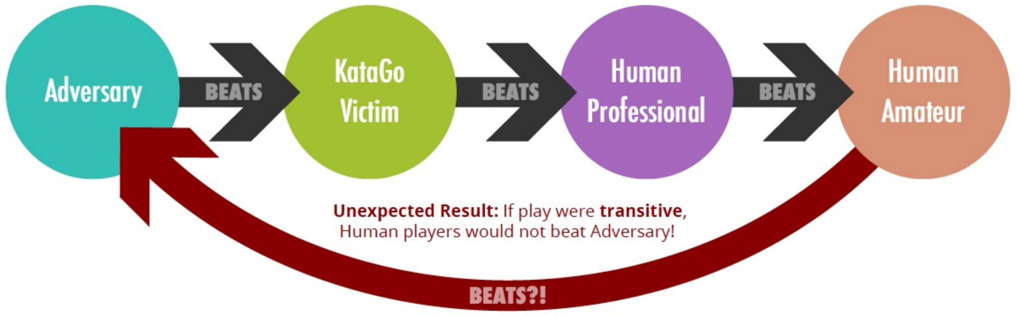

Adversary
BEATS
KataGo Victim
BEATS
Human Professional
BEATS
Human Amateur
Unexpected Result: If play were transitive, Human players would not beat Adversary!
BEATS?!

Figure 2.1: A human amateur beats our adversarial policy (Appendix I.1) that beats KataGo. This non-transitivity shows the adversary is not a generally capable policy, and is just exploiting KataGo.

How can we do it?

Classical vs. Machine Learning approaches

- We'll look first at the classical approach used from the 1940s to 2010
- Then at newer statistical approaches, of which AlphaGo is an example
- And reinforcement learning, used by Facebook's ReBel for Texas Hold'em
- These all share some techniques

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 state of game. No information hidden from either player.
- **No chance** (e.g., using dice, shuffled cards) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello, …
- But not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

Can we use …

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

None of these model the fact that we have an **adversary** …

How to play a game, v1

- 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 for you
  - 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 each resulting position

Evaluation function

- **Evaluation function** or static* **evaluator** used to evaluate the "goodness" of a game position

  Contrast with heuristic search, where evaluation function estimates **cost** from start to goal passing through given node

- Zero-sum assumption permits single function to describe goodness of board for both players
- "me" = player doing the evaluation
  - **f(p) >> 0**: position p good for me; bad for you
  - **f(p) << 0**: position p bad for me; good for you
  - **f(p) near 0**: position p is a neutral position
  - **f(p) = +infinity**: win for me
  - **f(p) = -infinity**: win for you

***static:** snapshot in time

Evaluation function examples

- **For Tic-Tac-Toe**

  f(n) = [# my open 3lengths] - [# your open 3lengths]

  Where an **open 3length** is complete row, column or diagonal with no opponent marks

- **Alan Turing's function for chess**
  - **f(n) = w(n)/b(n)** where w(n) = sum of point value of white's pieces and b(n) = sum of black's
  - Traditional chess piece values: pawn:1; knight:3; bishop:3; rook:5; queen:9

Evaluation function examples

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

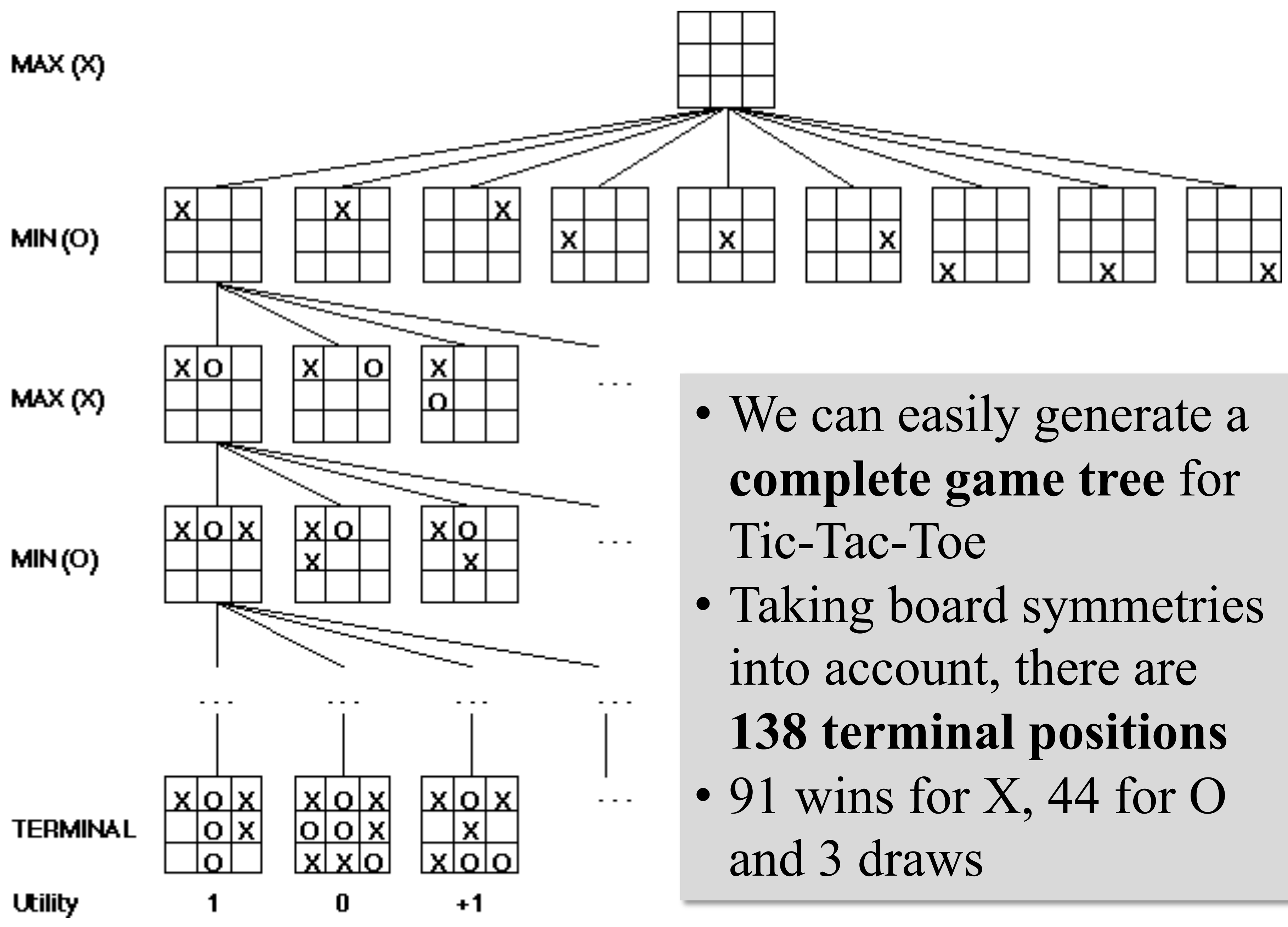
  $f(n) = w_1 * feat_1(n) + w_2 * feat_2(n) + ... + w_n * feat_k(n)$

- Typical chess features: piece count, piece values, piece placement, squares controlled, …
- IBM's chess program Deep Blue (circa 1996) had >8K features in its evaluation function!
- We can **learn weights** from choices made by expert players in real games (lots of data for chess and other popular games!)

But that's not how people play (1)

- People also use *look ahead*, *i.e.*

  enumerate actions, consider opponent's possible responses, REPEAT
- Producing a *complete* **game tree** only possible for simple games
- A complete tree has all possible moves for each position with each leaf being a draw or a win for one player
  - We need a graph if there can be loops (as in chess)

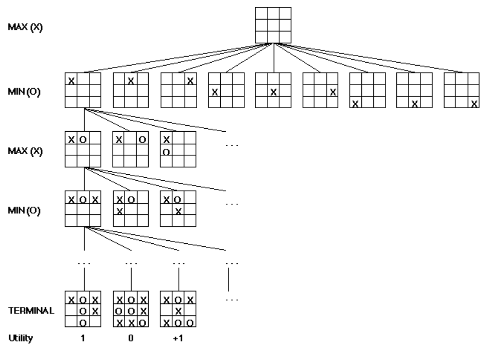


- We can easily generate a **complete game tree** for Tic-Tac-Toe
- Taking board symmetries into account, there are **138 terminal positions**
- 91 wins for X, 44 for O and 3 draws

But that's not how people play (2)

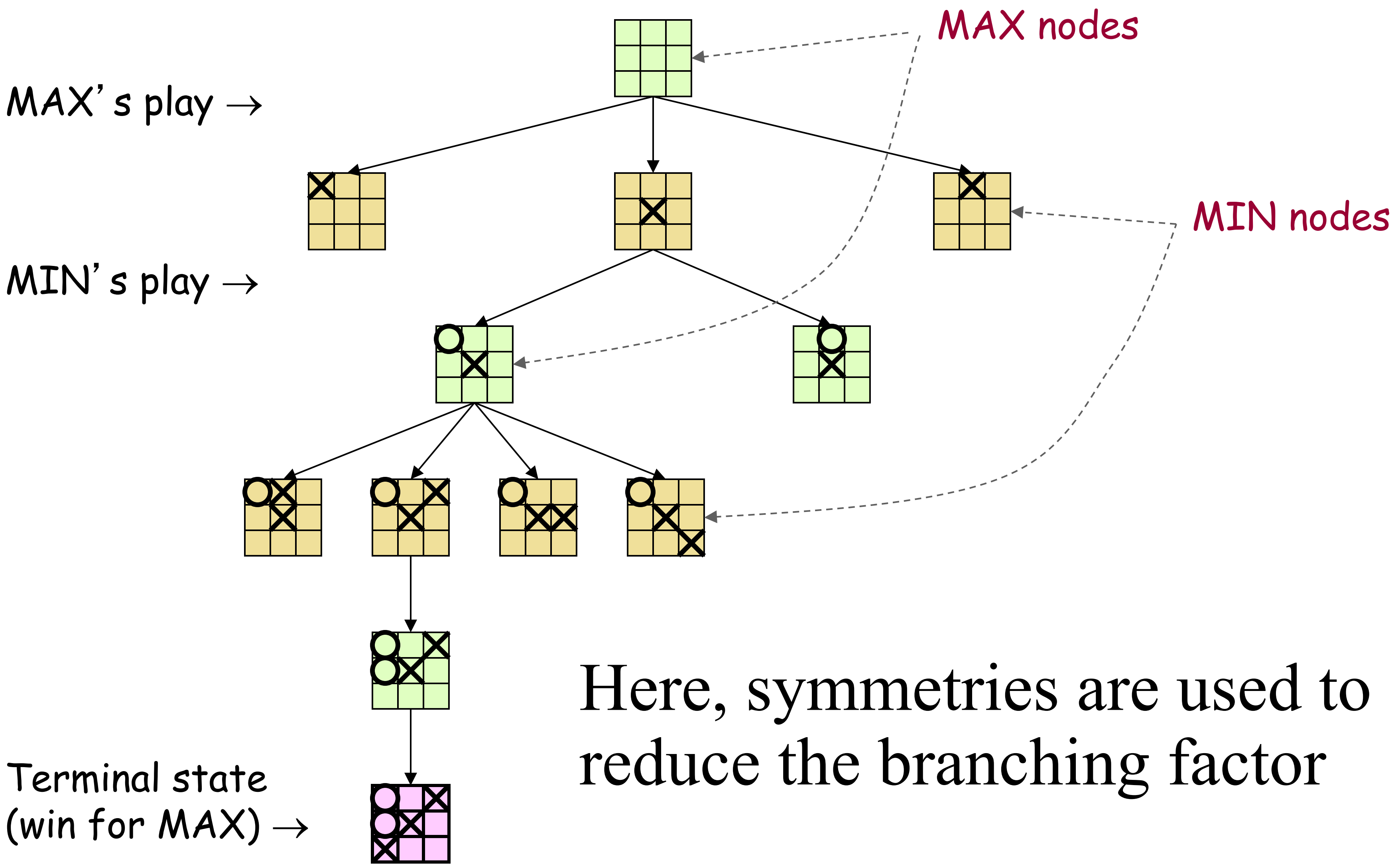
- For **non-simple games** we generate a **partial game tree** for some number of plies
- For games, we say…
  - Move = each player takes a turn
  - Ply = one player's turn
- How far we can "look ahead" depends mostly on the branching factor
  - How many moves a player might have
  - Checkers ≈ 6.4; Chess ≈ 35
- What do we do with the partial game tree?

Game trees



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

Game Tree for Tic-Tac-Toe

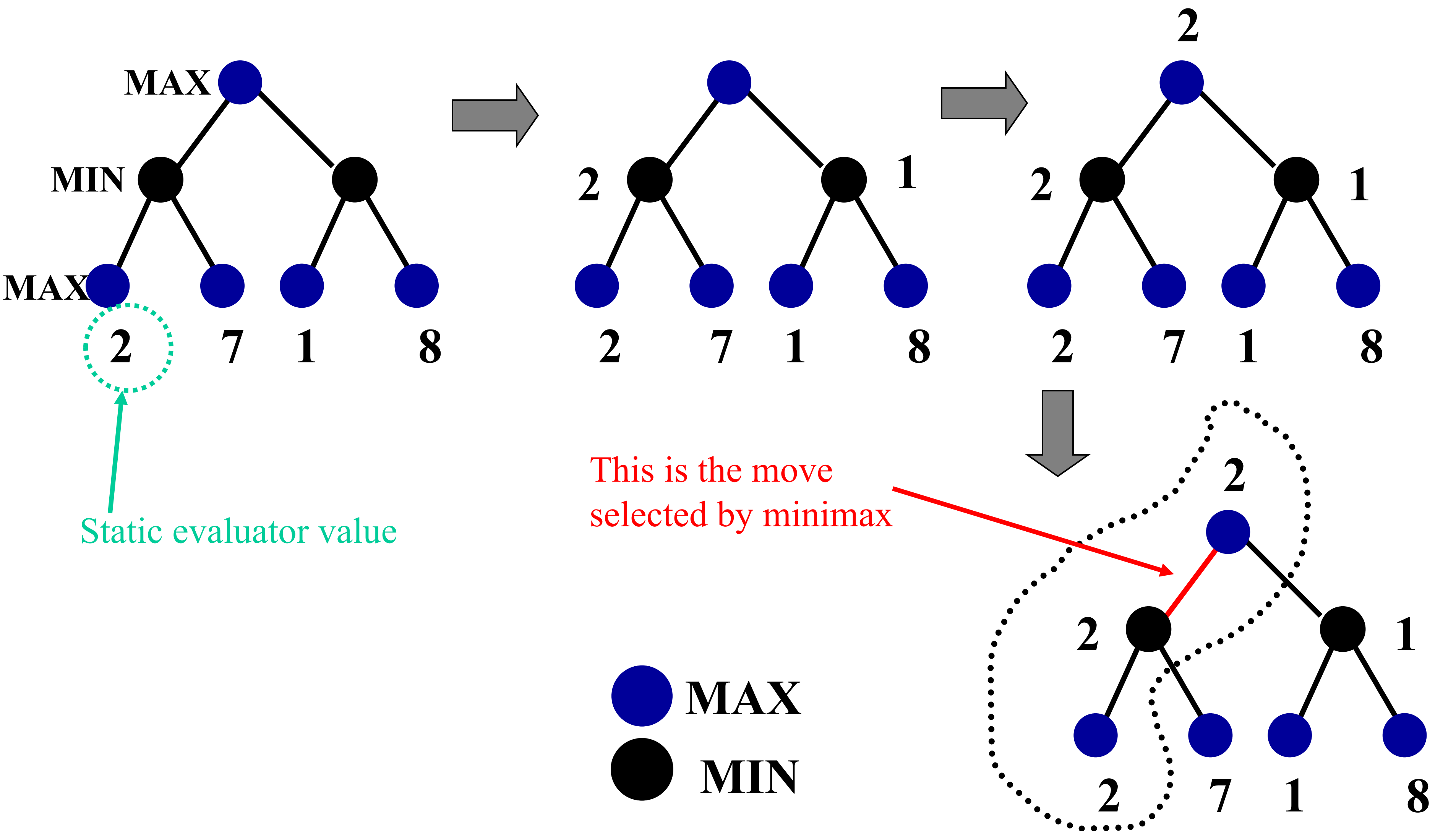


Here, symmetries are used to reduce the branching factor

Minimax procedure

- Create MAX node with current board configuration
- Expand nodes to some **depth** (e.g., five plies) of lookahead in game
- Apply evaluation function at each **leaf** node
- **Back up** values for each non-leaf node until value is computed for the root node
  - At MIN nodes: value is **minimum** of childrens' values
  - At MAX nodes: value is **maximum** of childrens' values
- **Choose move** to child node whose backed-up value determined value at root

Minimax Algorithm

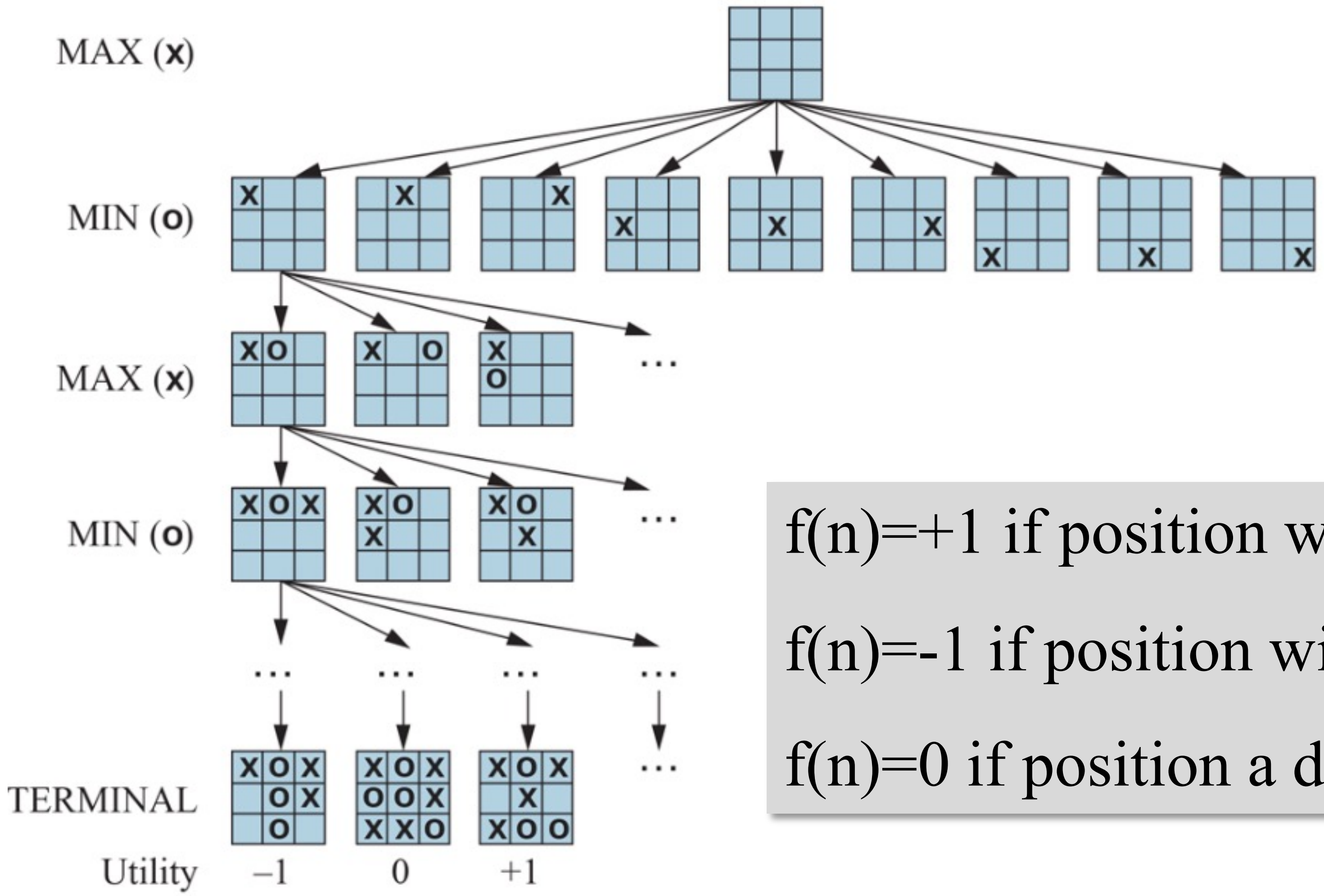


Minimax theorem

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

  For every 2-person, 0-sum game with finite strategies, there is a value V and a mixed strategy for each player, such that (a) given player 2's strategy, best payoff possible for player 1 is V, and (b) given player 1's strategy, best payoff possible for player 2 is –V.
- You can think of this as:
  - Minimizing your maximum possible loss
  - Maximizing your minimum possible gain

Partial Game Tree for Tic-Tac-Toe



f(n)=+1 if position win for X

f(n)=-1 if position win for O

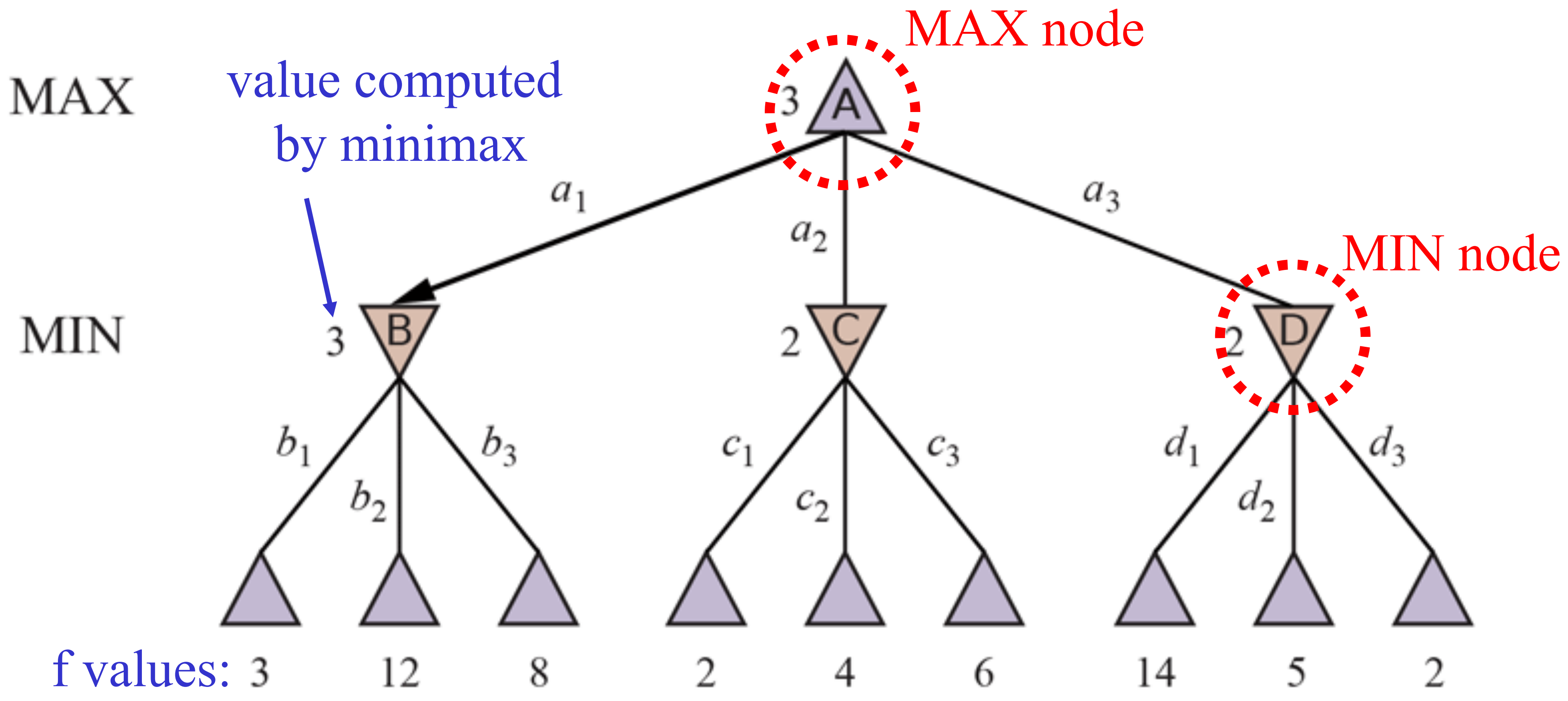
f(n)=0 if position a draw

Partial game tree for tic-tac-toe. Top node is the initial state, and max moves first, placing an X in an empty square. Only part of the tree shown, giving alternating moves by min (O) and max (X), until we reach terminal states, which are assigned utilities {-1,0,+1} for {loose, draw, win}

Why backed-up values?

- Why not just use a good static evaluator metric on immediate children
- **Intuition:** if metric is good, doing look ahead and backing up values with Minimax should be better
- Non-leaf node N's backed-up value is value of best state MAX can reach at depth **h** if MIN plays *well*
  - "plays well": same criterion as MAX applies to itself
- If **e** is good, then backed-up value is better estimate of STATE(N) goodness than **e**(STATE(N))
- Use lookahead horizon **h** because time to choose a move is typically limited

Minimax Tree Again

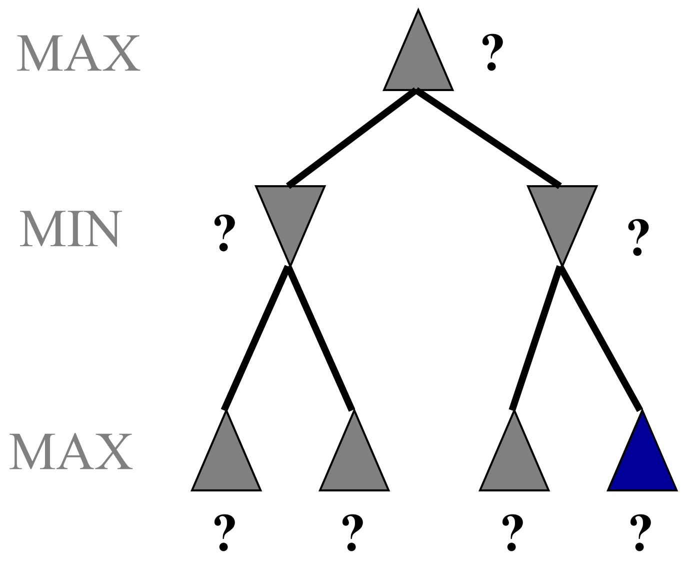


Two-ply game tree. △ nodes are "max nodes," in which it is max's turn to move, and ∇ nodes are "min nodes." The terminal nodes show utility values for max; the other nodes are labeled with their minimax values. max's best move at root is a1 because it leads to state with the highest minimax value. min's best reply is b1 since it leads to the state with the lowest minimax value.

Is that all there is to simple games?

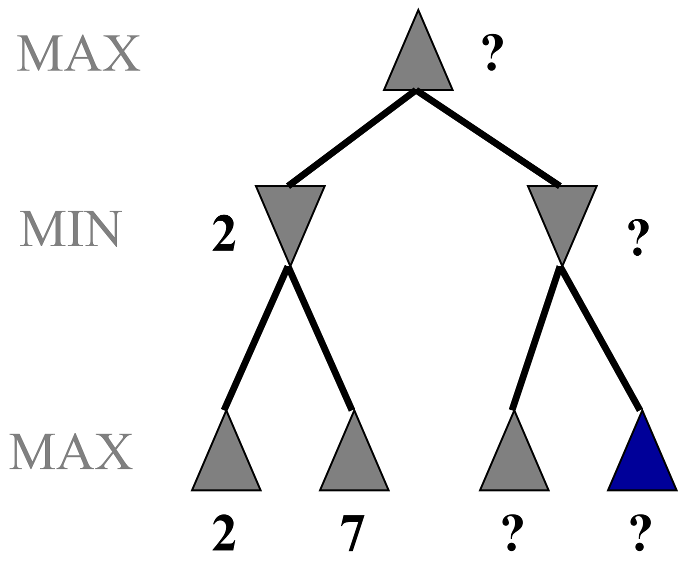
Alpha-beta pruning

- Improve performance of the minimax algorithm through **alpha-beta pruning**
- A simple approach is to compute tree to some depth, e.g., two plies



Alpha-beta pruning

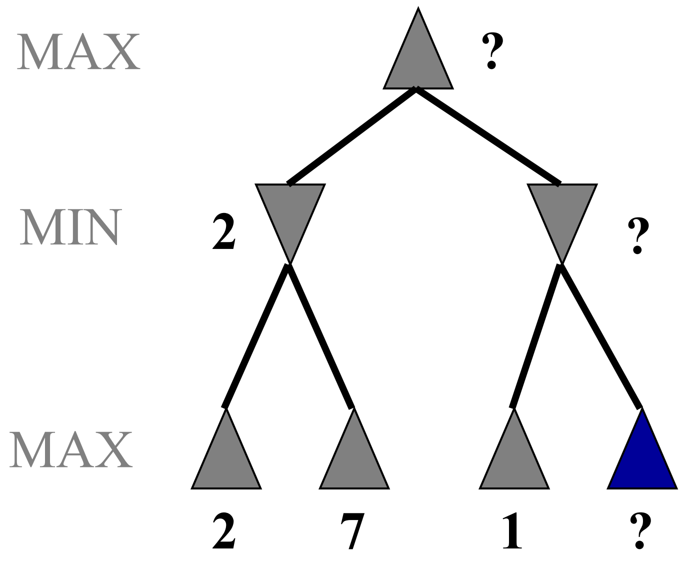
- Improve performance of the minimax algorithm through **alpha-beta pruning**
- A simple approach is to compute tree to some depth, e.g., two plies



- We then compute the values of each leaf node, from left to right using the static evaluator function
- We can compute the values of an interior node whenever we know all of its children's nodes

Alpha-beta pruning

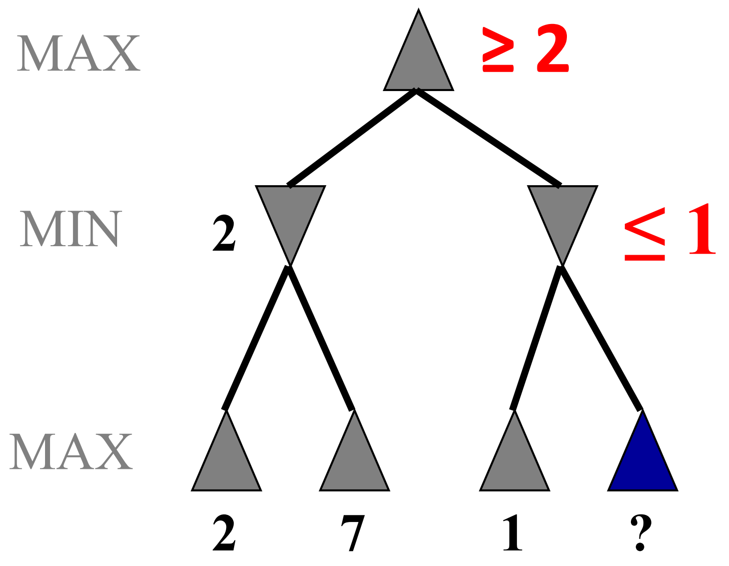
- Improve performance of the minimax algorithm through **alpha-beta pruning**
- A simple approach is to compute tree to some depth, e.g., two plies



- But sometimes, we can put a bound on an unknown value
- Computing the bound from what we do know

Alpha-beta pruning

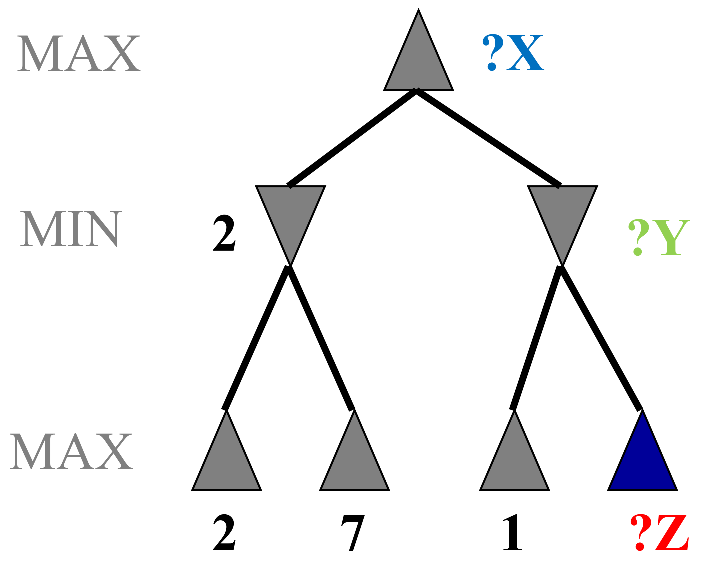
- Improve performance of the minimax algorithm through **alpha-beta pruning**
- A simple approach is to compute tree to some depth, e.g., two plies



- But sometimes, we can put a bound on an unknown value
- Computing the bounds from the values and other bounds we do know

Alpha-beta pruning

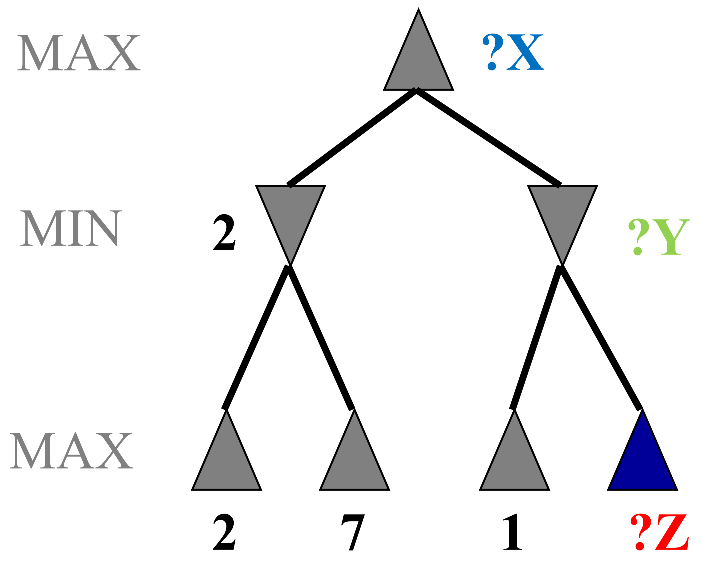
- Improve performance of the minimax algorithm through **alpha-beta pruning**
- *"If you have an idea that is surely bad, don't take the time to see how truly awful it is"* **-**Pat Winston (MIT)



- Need not **compute** the value of ?Z
- Know value of ?Y will be smaller of 1 and ?Z, i.e., **?Y ≤ 1 & ?Y≤?Z**
- No matter what ?Z is, it can't affect value of ?X
- MAX can get a 2 by choosing its first move

Alpha-beta pruning

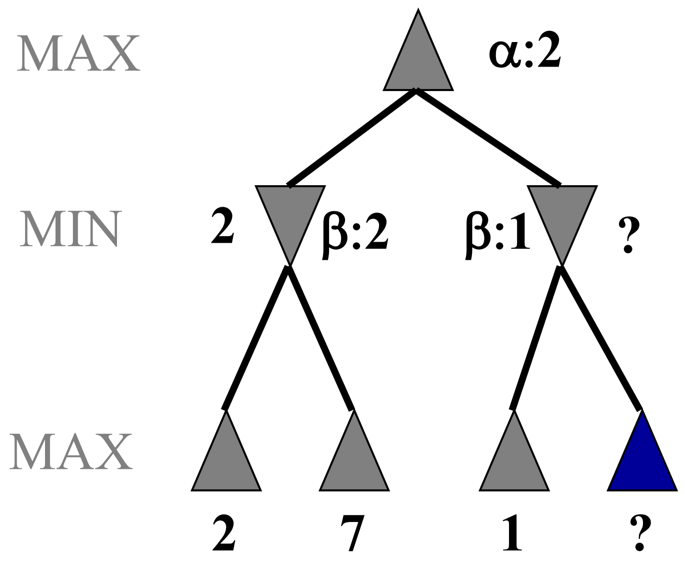
- Improve performance of the minimax algorithm through **alpha-beta pruning**
- *"If you have an idea that is surely bad, don't take the time to see how truly awful it is"* -Pat Winston (MIT)



- Need not **compute** the value of ?Z
- Know value of ?Y will be smaller of 1 and ?Z, i.e., **?Y ≤ 1 & ?Y≤?Z**
- No matter what ?Z is, it can't affect value of ?X
- MAX can get a 2 by choosing its first move

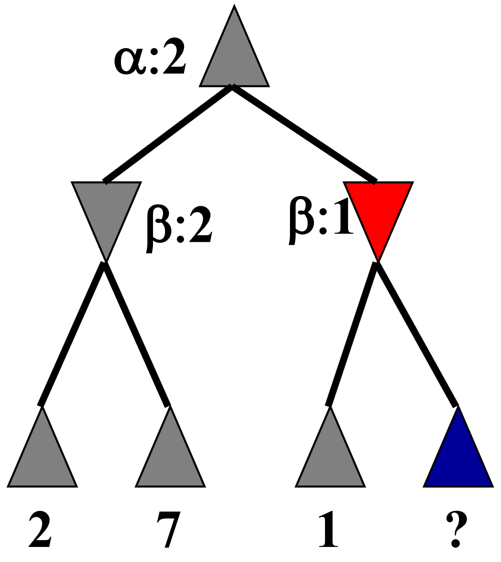
Alpha-beta pruning

- Improve performance of the minimax algorithm through **alpha-beta pruning**
- *"If you have an idea that is surely bad, don't take the time to see how truly awful it is"* -Pat Winston (MIT)



- The alpha-beta algorithm maintains bounds on non-leaf nodes, **alpha** for Max nodes and beta for Min nodes
- A Max node's **alpha** is a **lower bound** on its final value
- A Min node's **beta** is a **upper bound** on its final value

Alpha-beta pruning

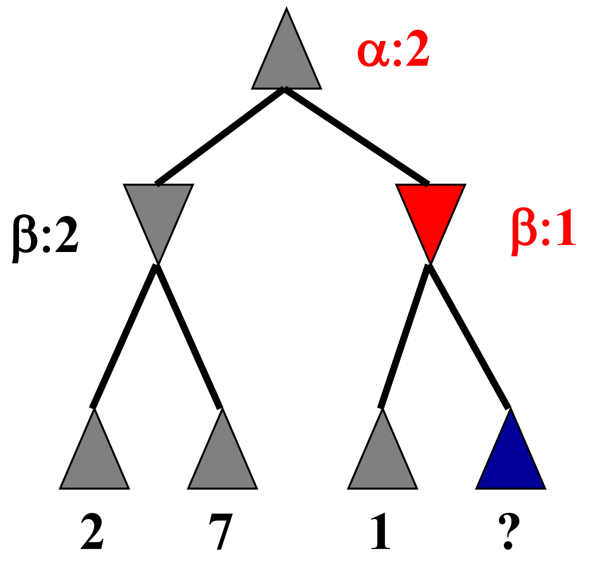


- Traverse tree in depth-first order
- At **MAX** node n, **alpha(n)** = max value so far in immediate subtree

  Alpha values start at -∞ and only increase
- At **MIN** node n, **beta(n)** = min value found so far

  Beta values start at +∞ and only decrease
- **Beta cutoff**: stop search below MAX node N (i.e., don't examine more descendants) if alpha(N) >= beta(i) for some MIN node ancestor i of N
- **Alpha cutoff:** stop search below MIN node N if beta(N)<=alpha(i) for a MAX node ancestor i of N

Alpha-beta pruning

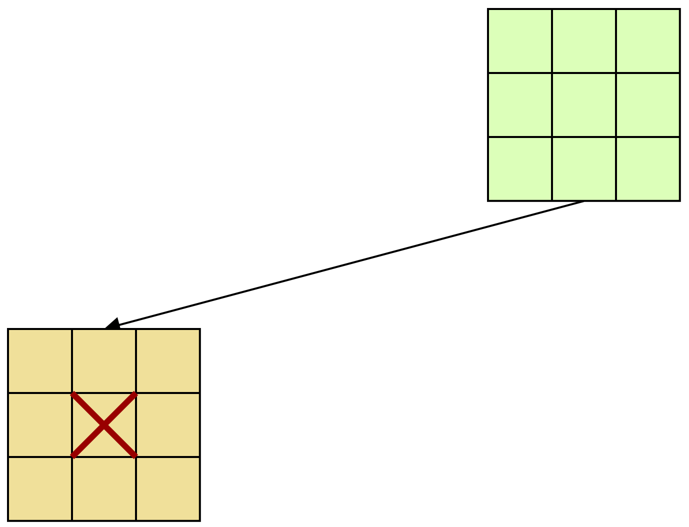


- Traverse tree in depth-first order
- At **MAX** node n, **alpha(n)** = max value found so far

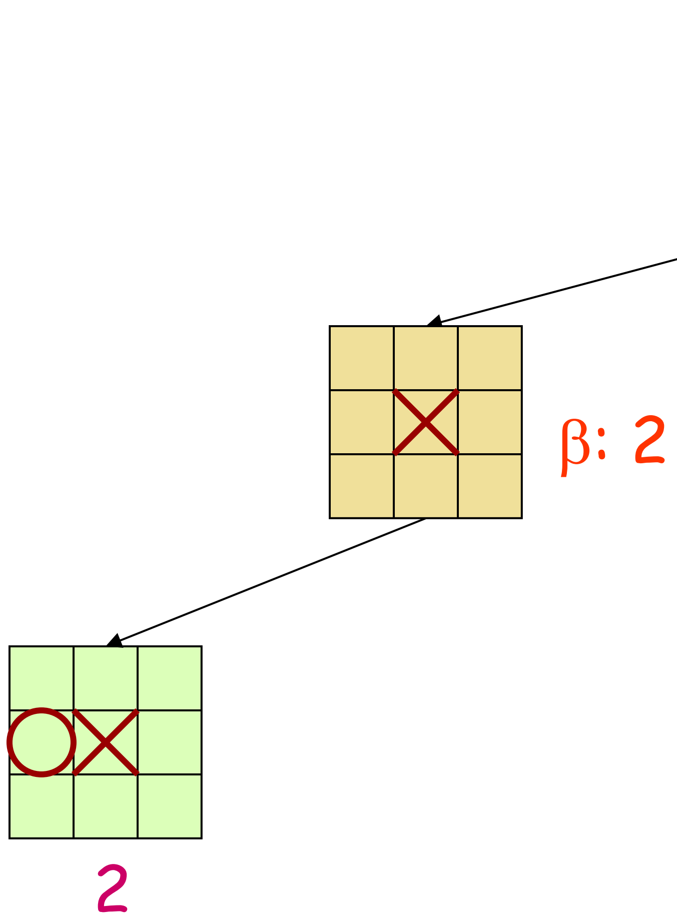
  Alpha values start at -∞ and only increase
- At **MIN** node n, **beta(n)** = min value found so far

  Beta values start at +∞ and only decrease
- **Beta cutoff**: stop search below MAX node N (i.e., don’t examine more descendants) if alpha(N) >= beta(i) for some MIN node ancestor i of N
- **Alpha cutoff:** stop search below MIN node N if beta(N)<=alpha(i) for a MAX node ancestor i of N

Alpha-Beta Tic-Tac-Toe Example



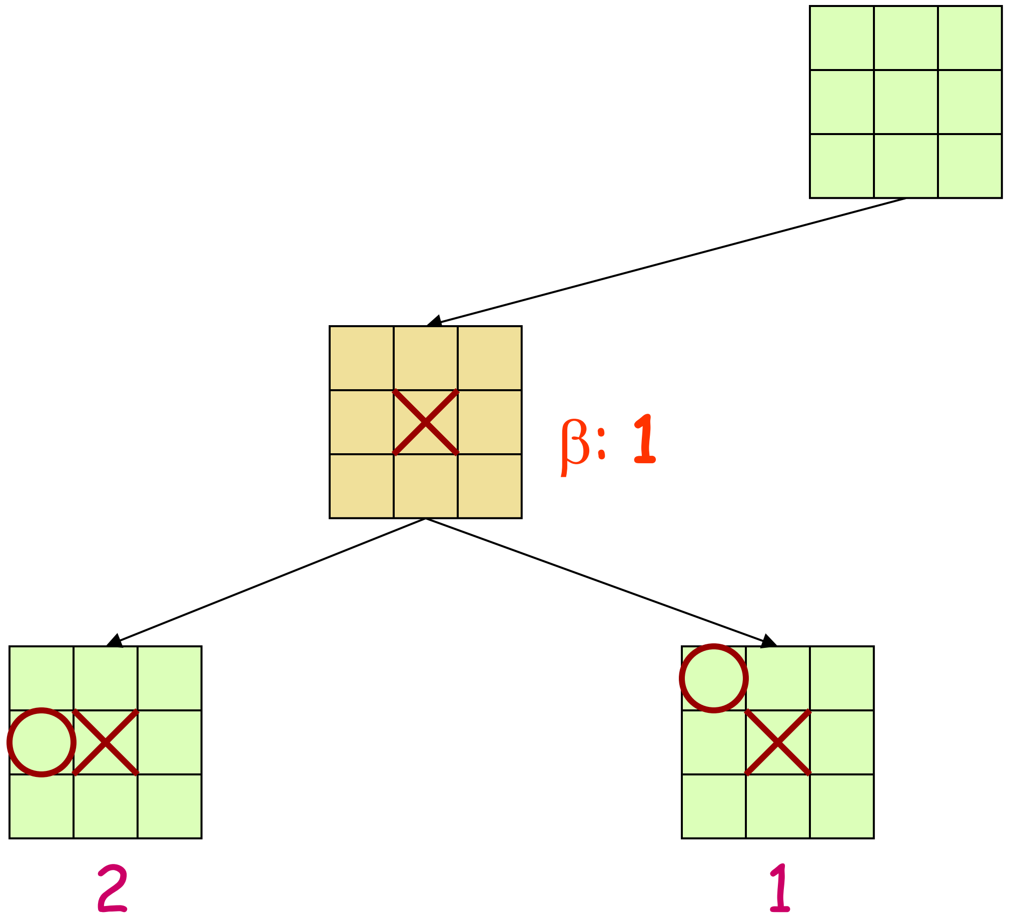
Alpha-Beta Tic-Tac-Toe Example



F = X's open lines – O's open lines

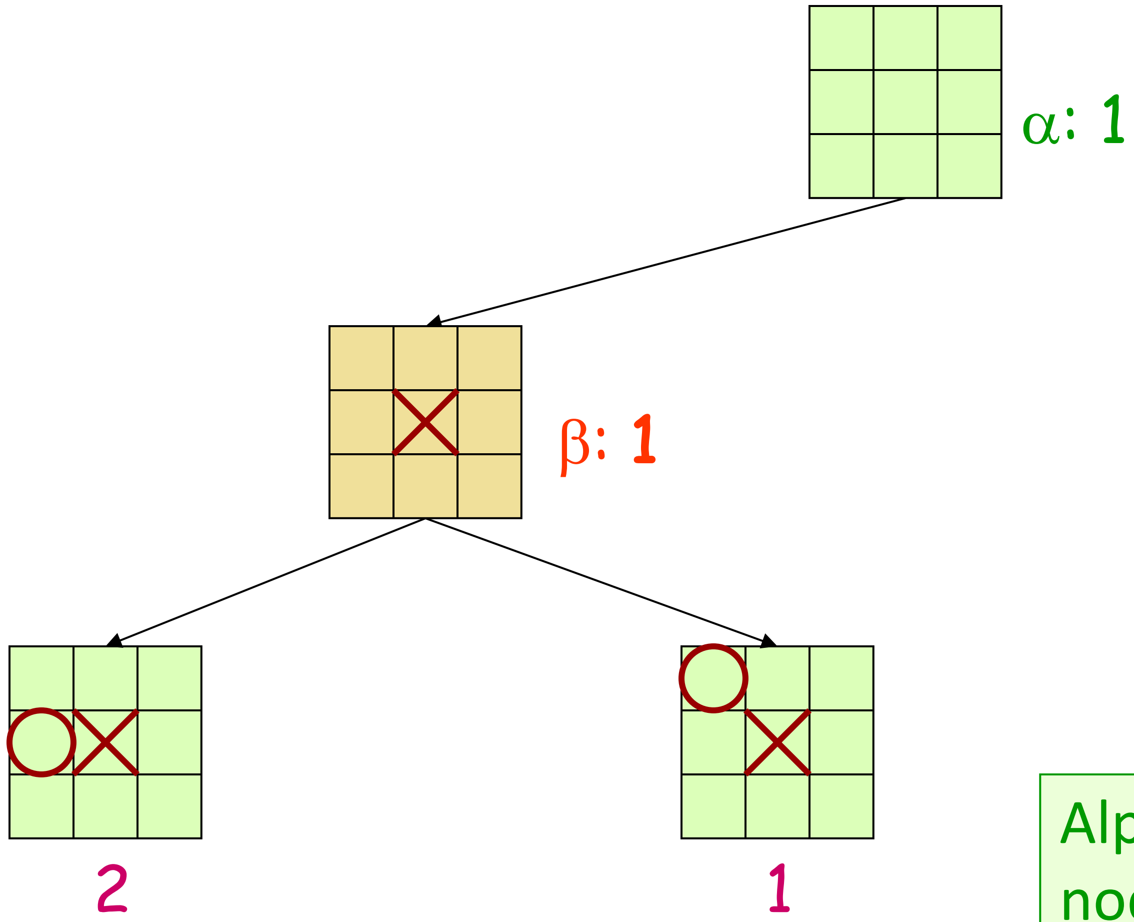
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example



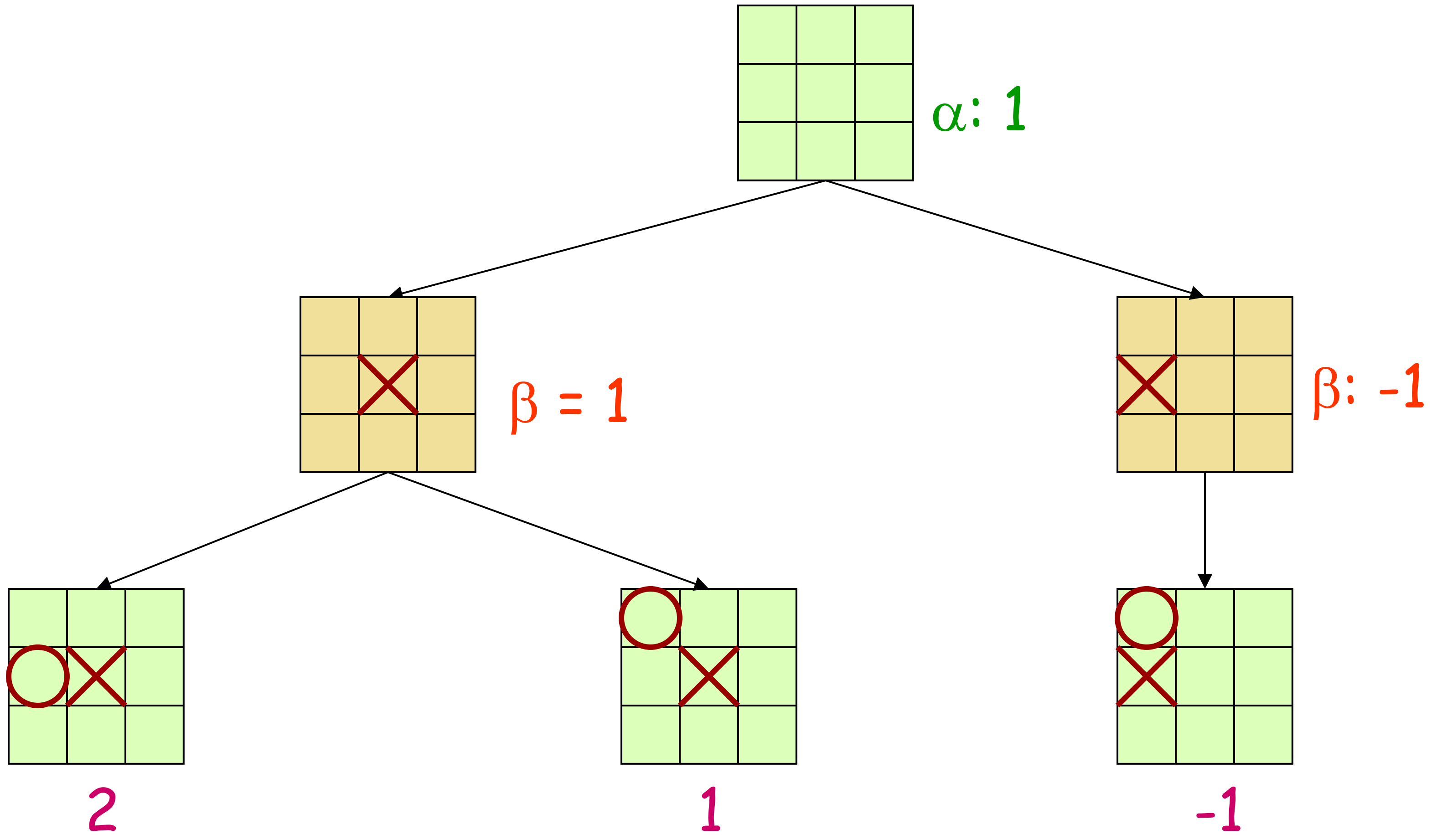
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

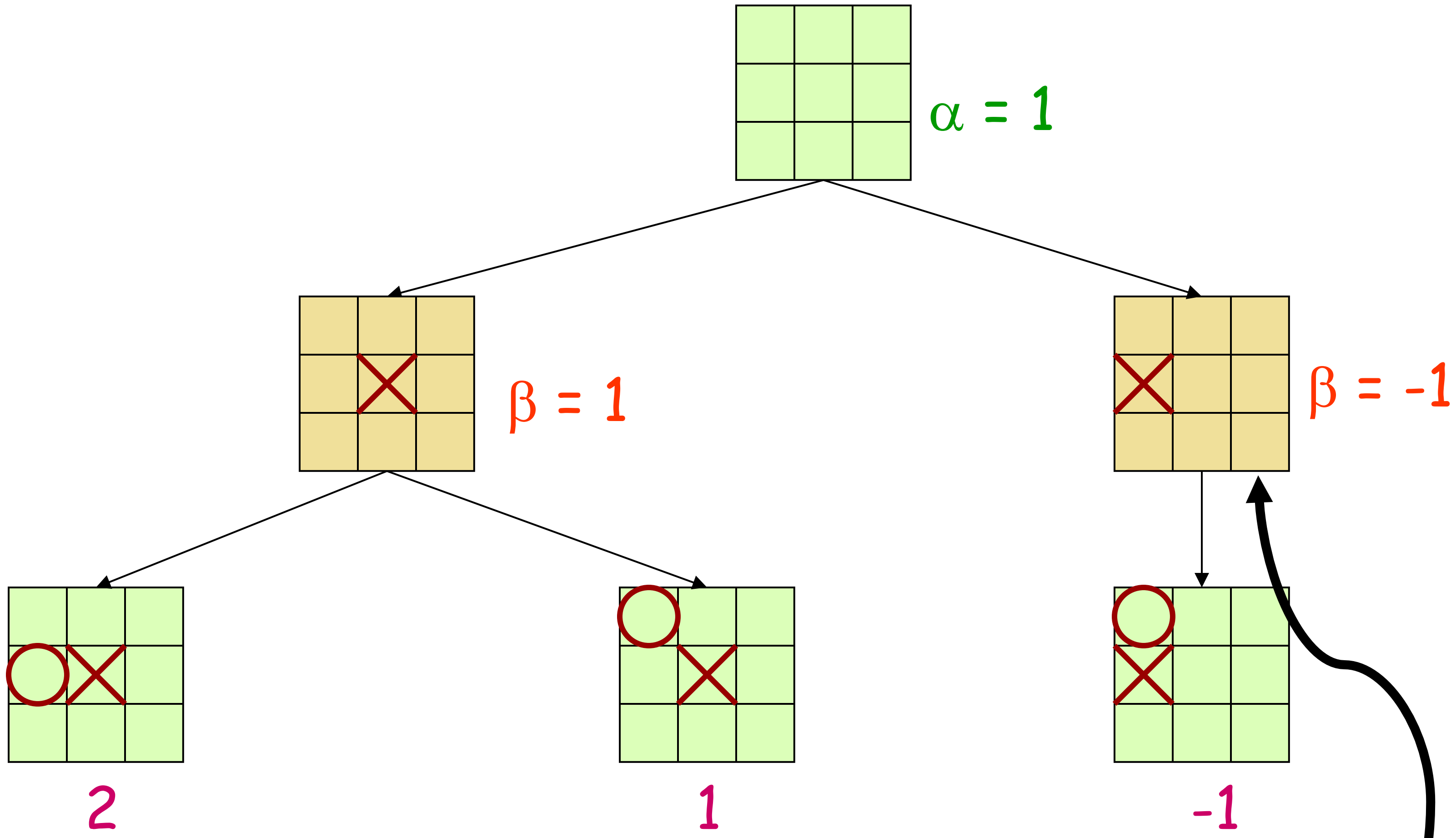


Alpha value of MAX node is **lower** bound on final backed-up value; it can never decrease

Alpha-Beta Tic-Tac-Toe Example

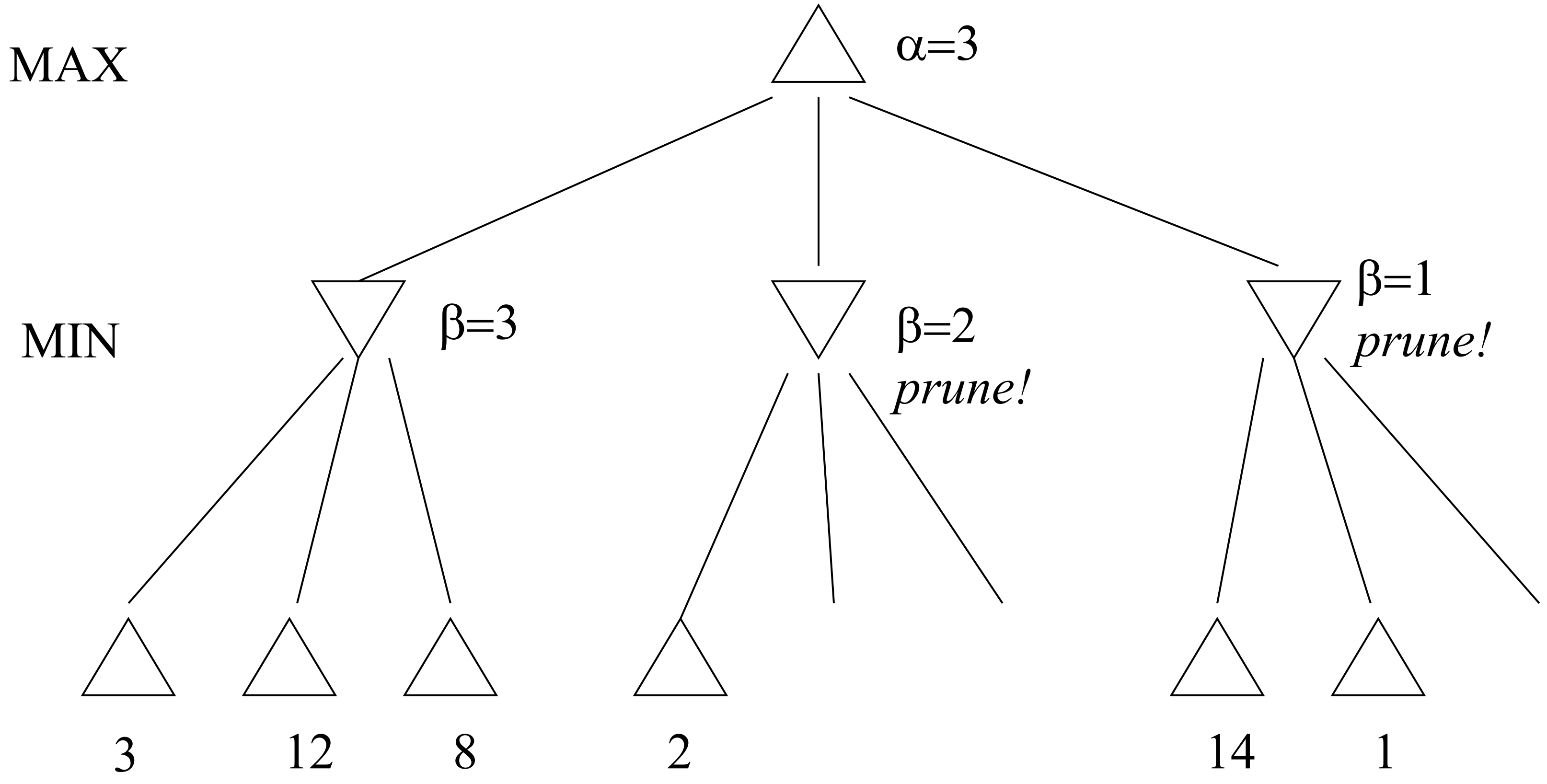


Alpha-Beta Tic-Tac-Toe Example

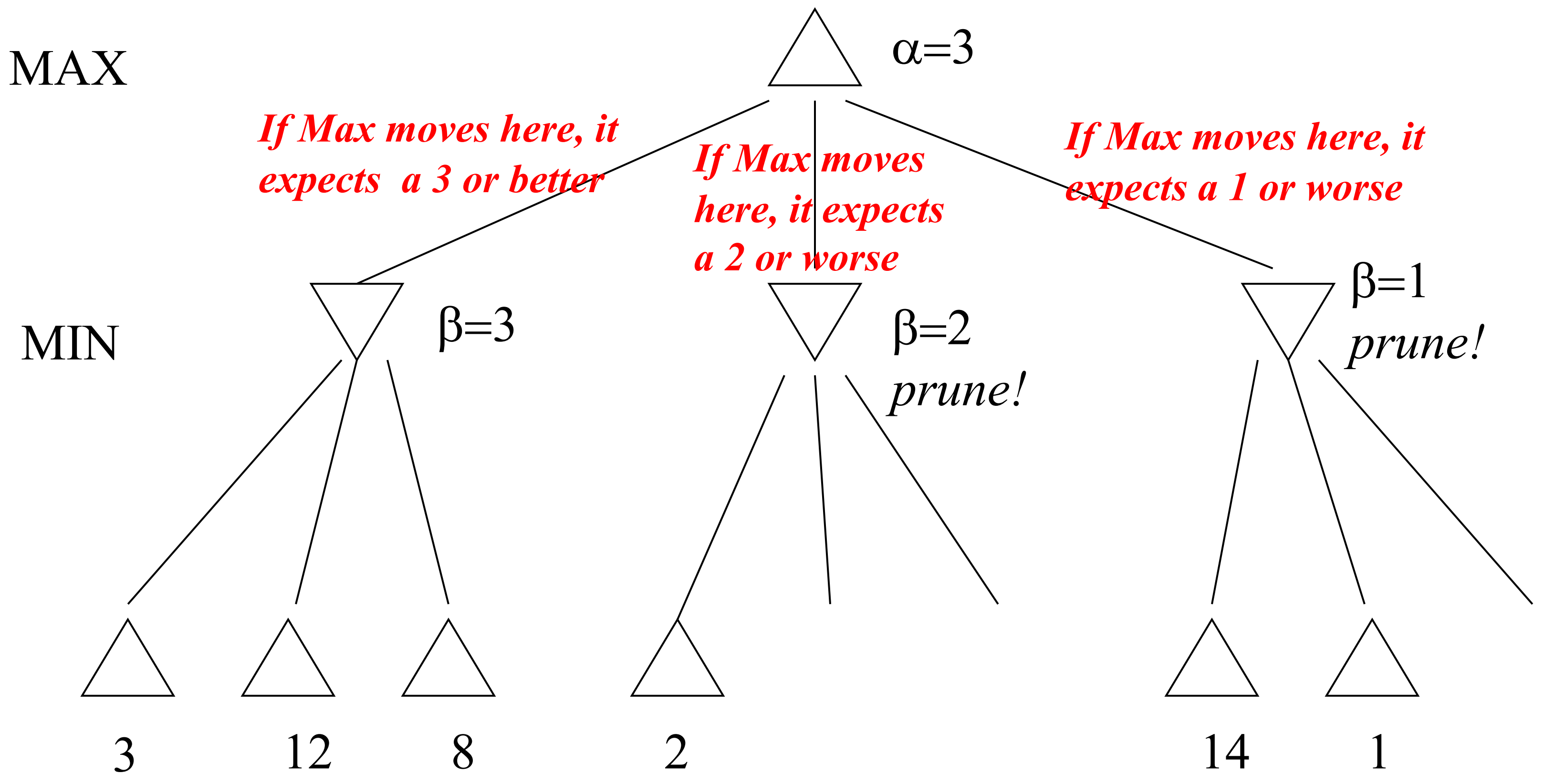


Discontinue search below a MIN node whose beta value ≤ alpha value of one of its MAX ancestors

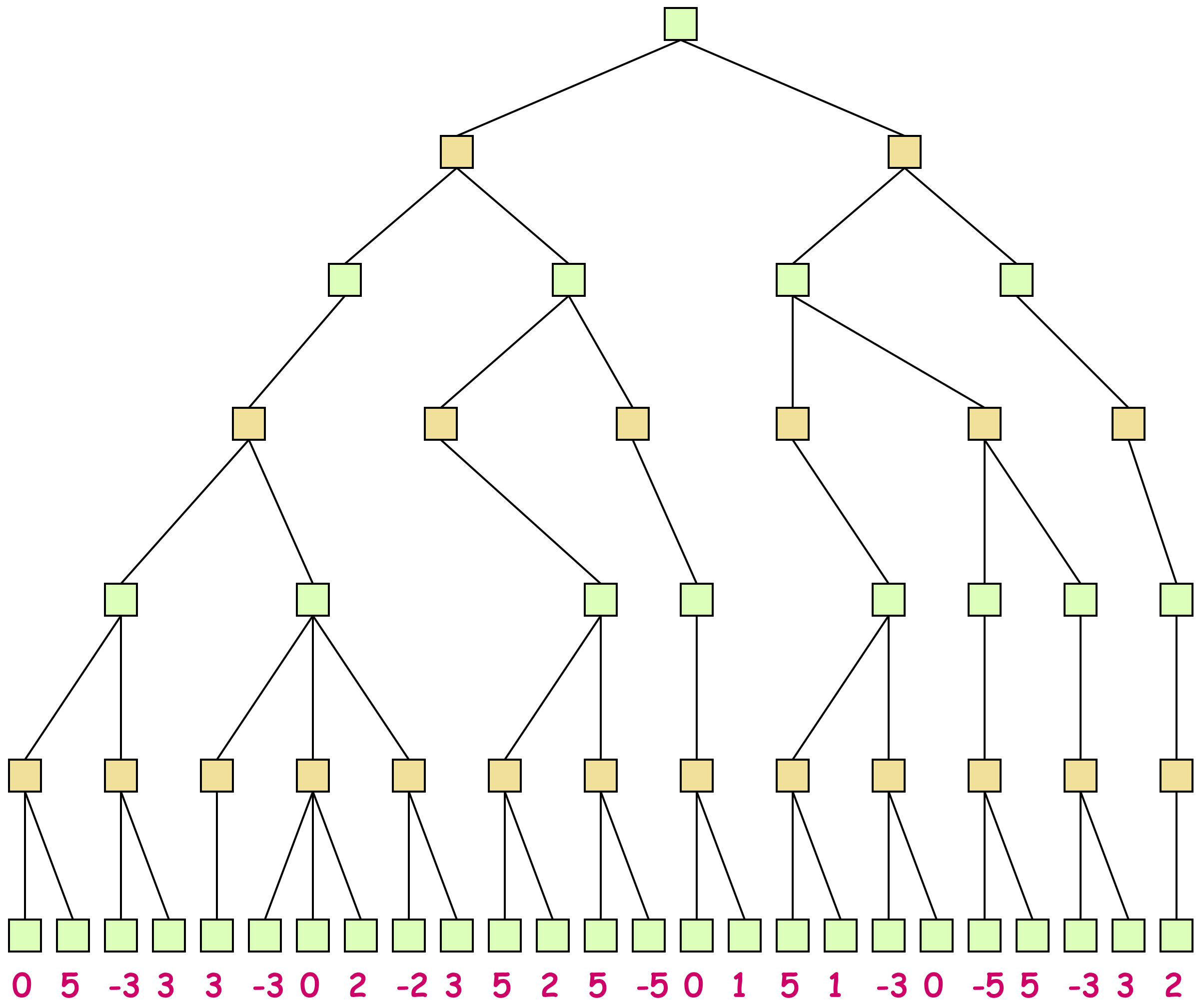
Another alpha-beta example

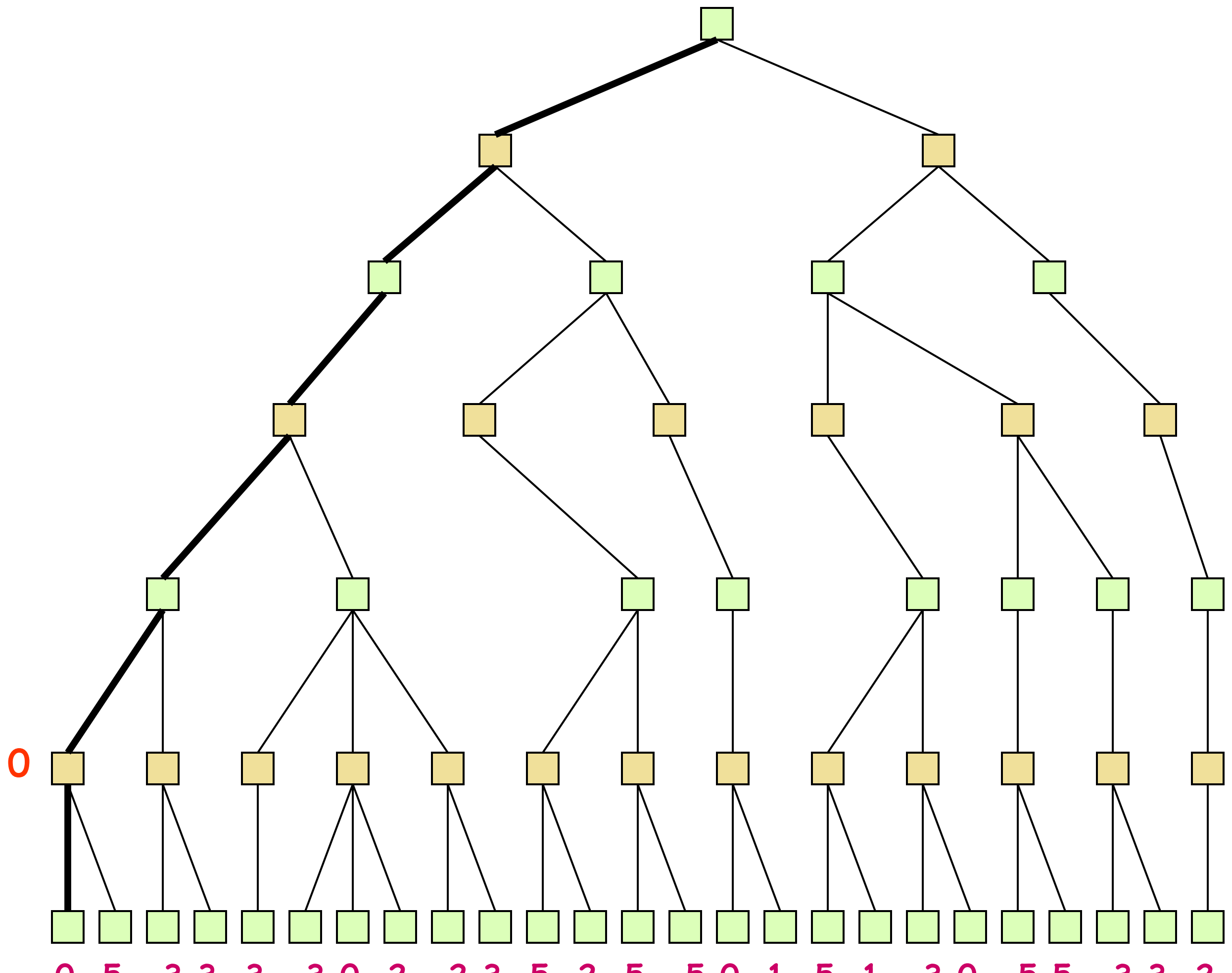


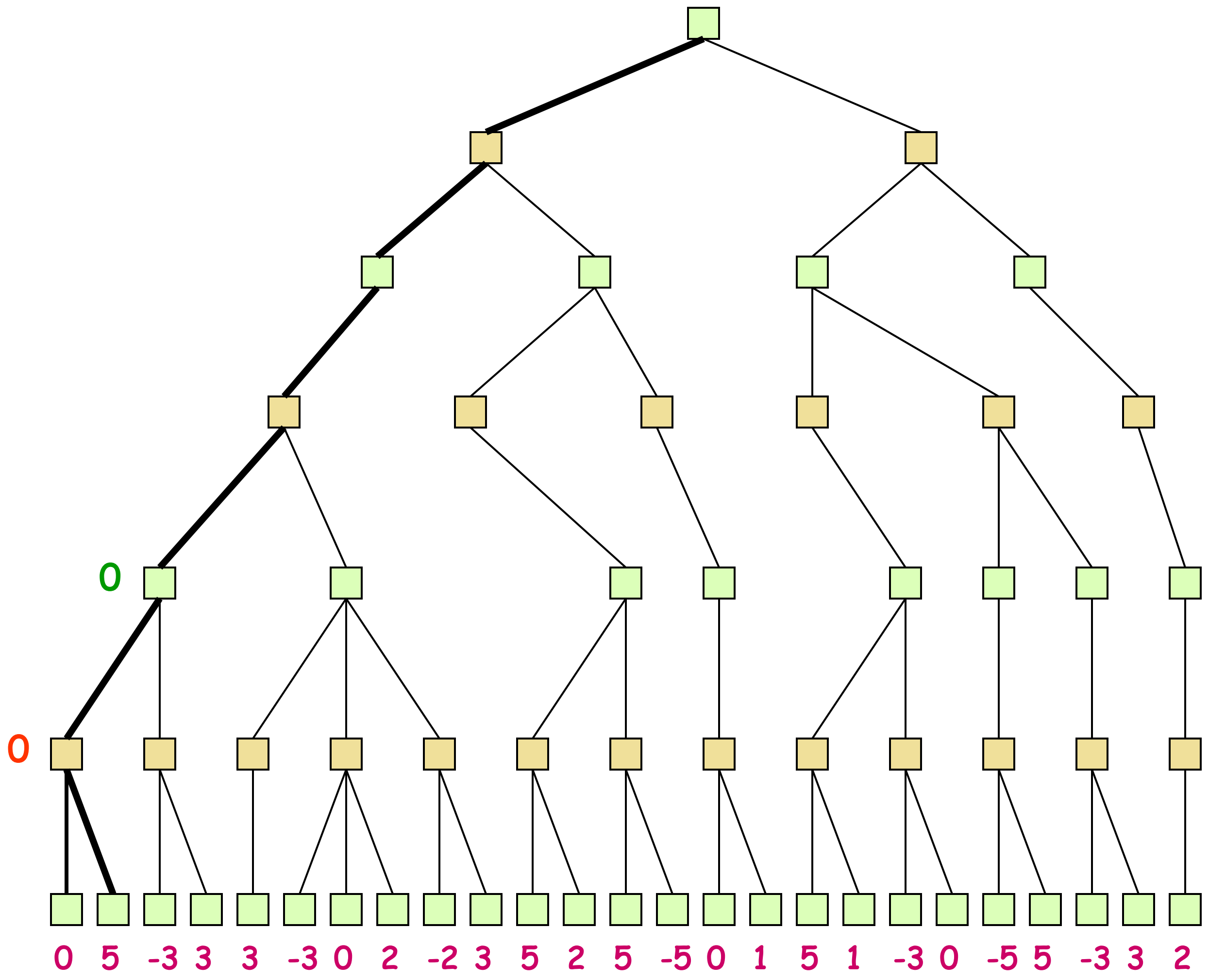
Another alpha-beta example

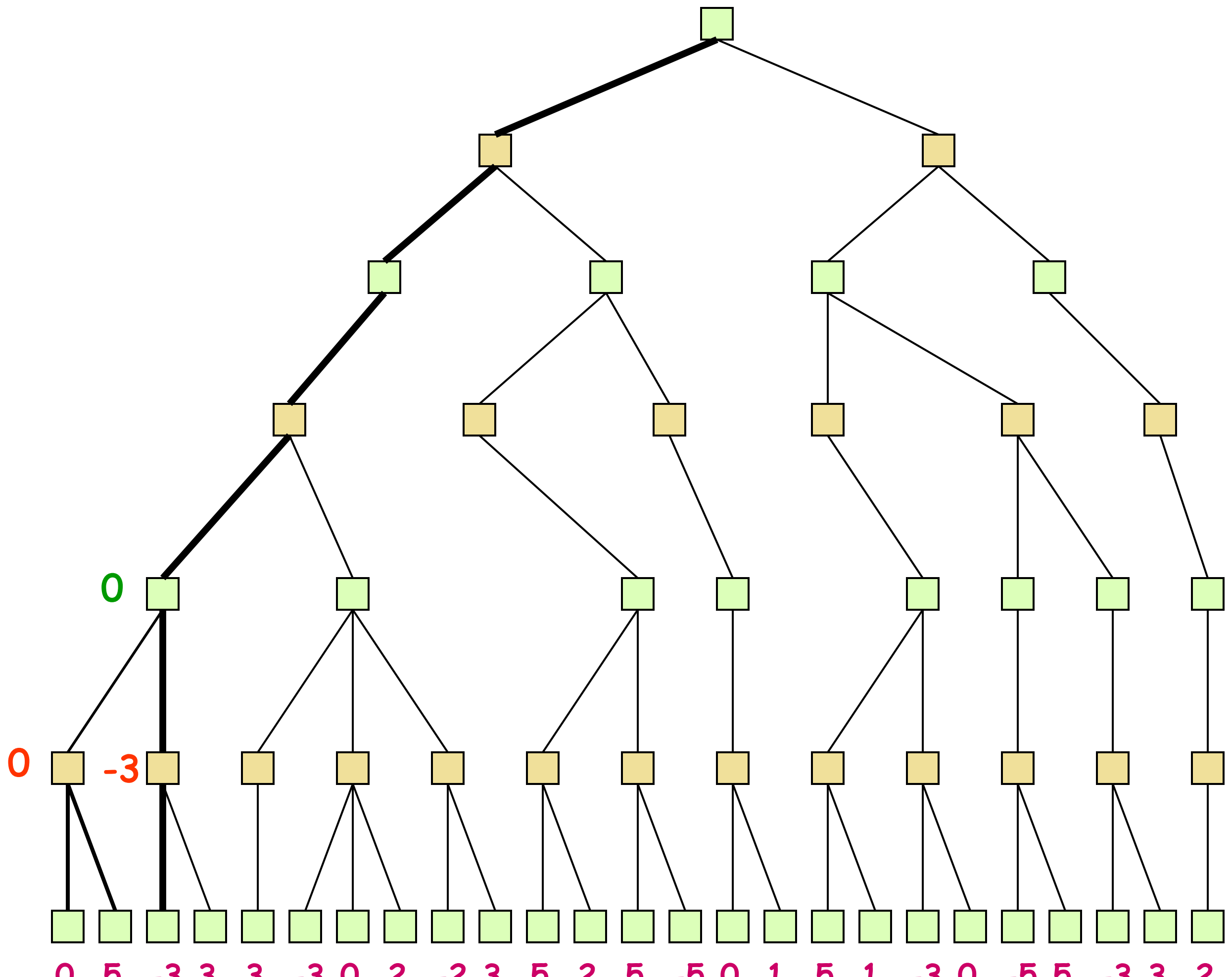

MAX
MIN
α=3
If Max moves here, it expects a 3 or better
If Max moves here, it expects a 2 or worse
If Max moves here, it expects a 1 or worse
β=3
β=2
prune!
β=1
prune!
3
12
8
2
14
1

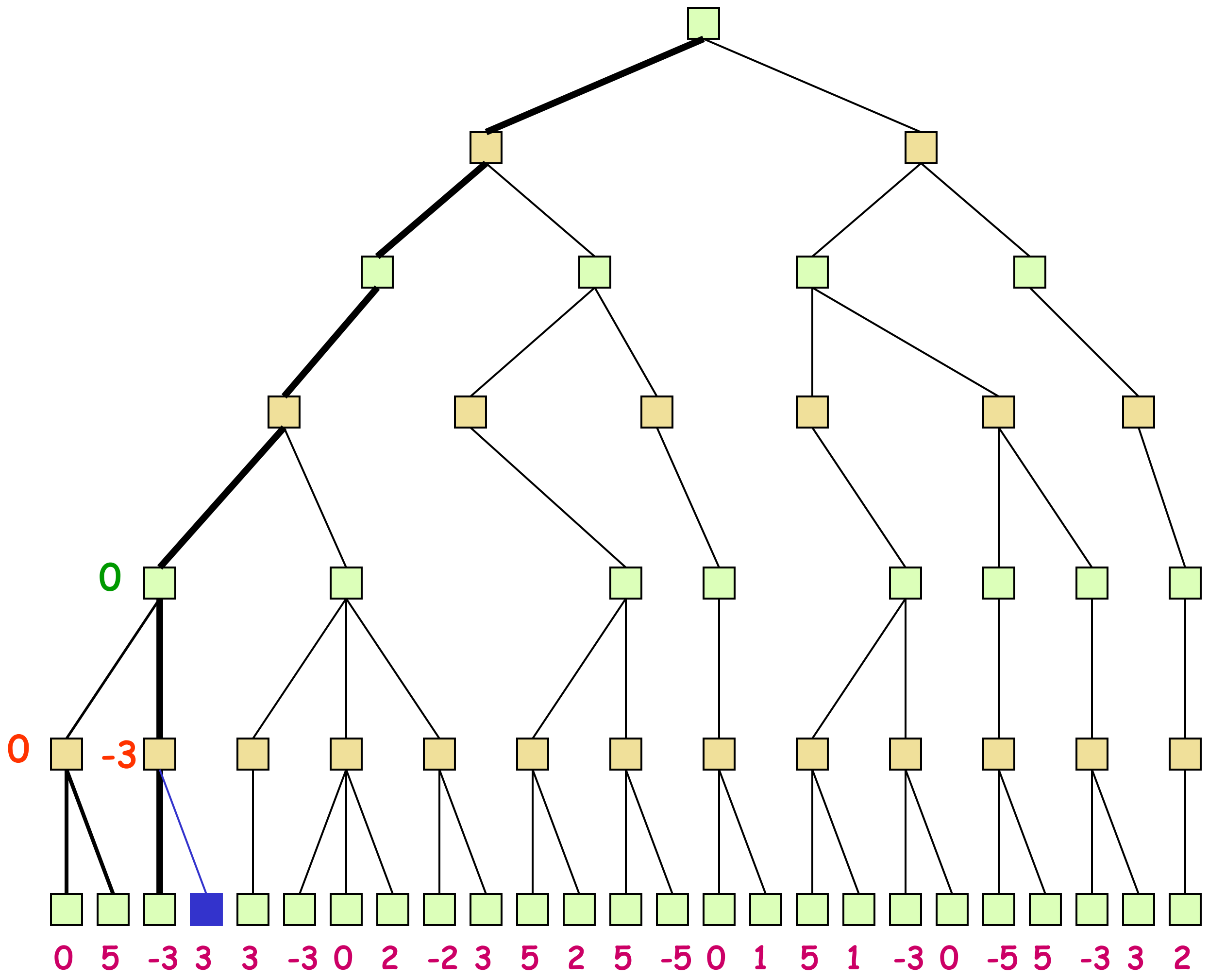
Alpha-Beta Tic-Tac-Toe Example 2

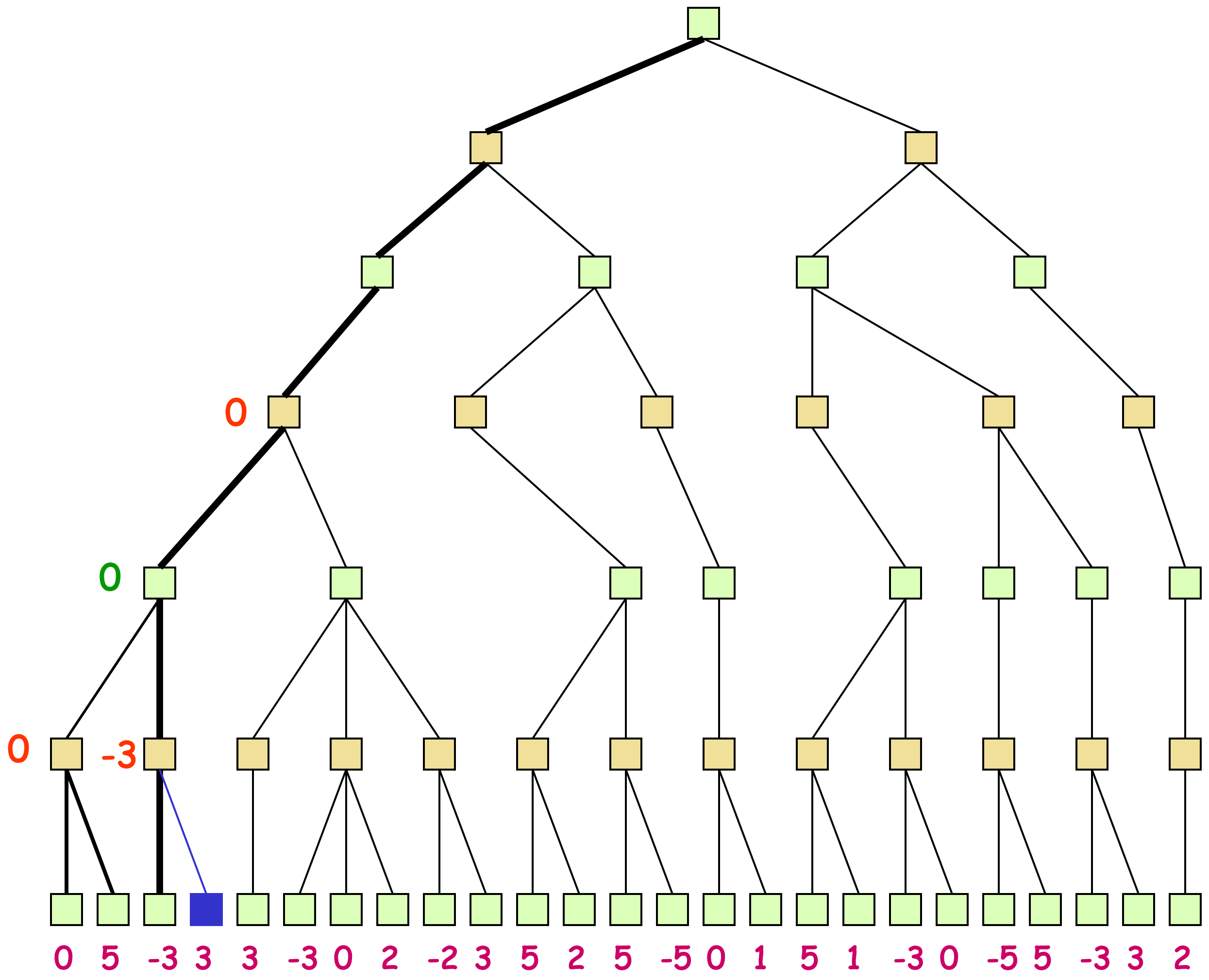


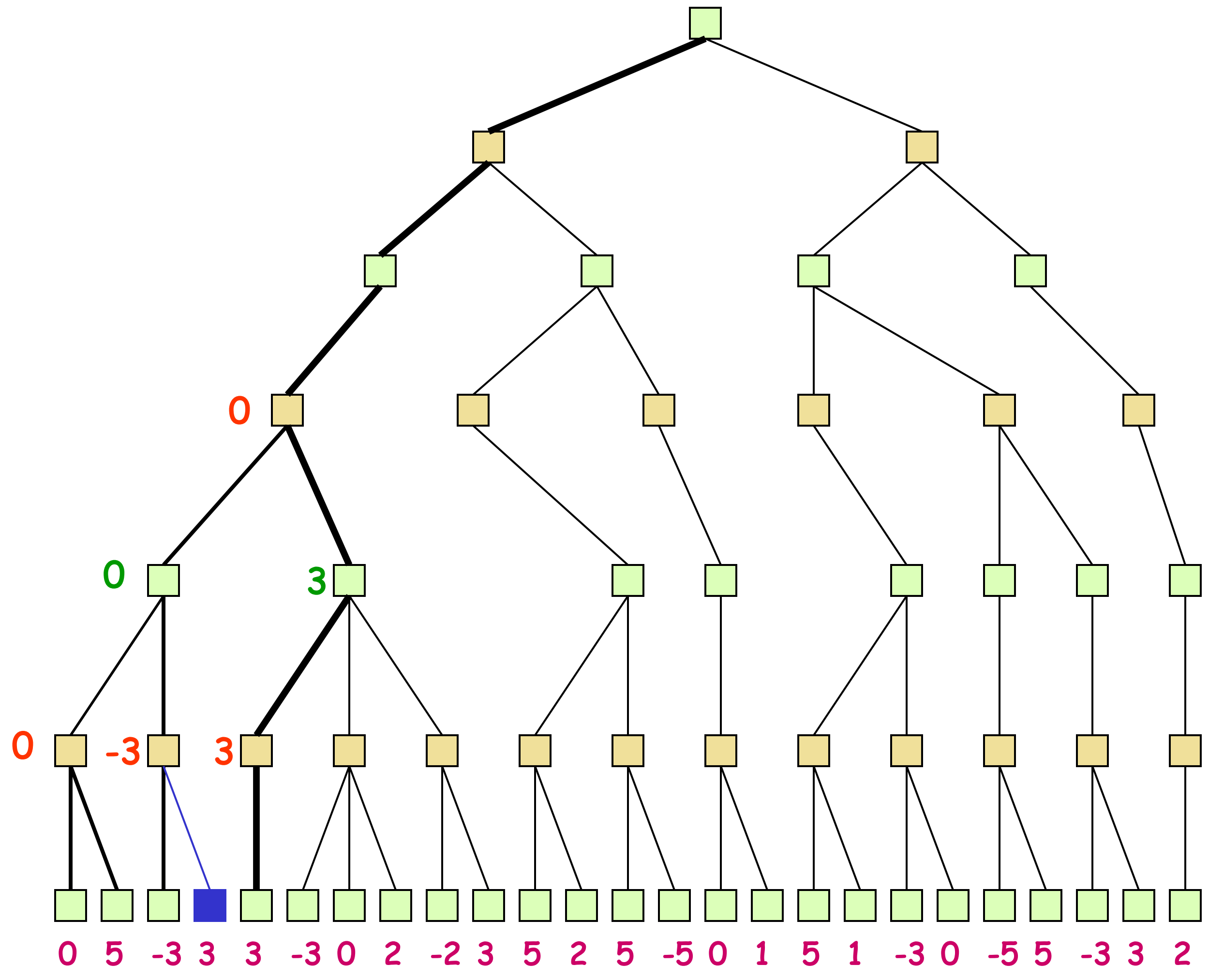

0
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

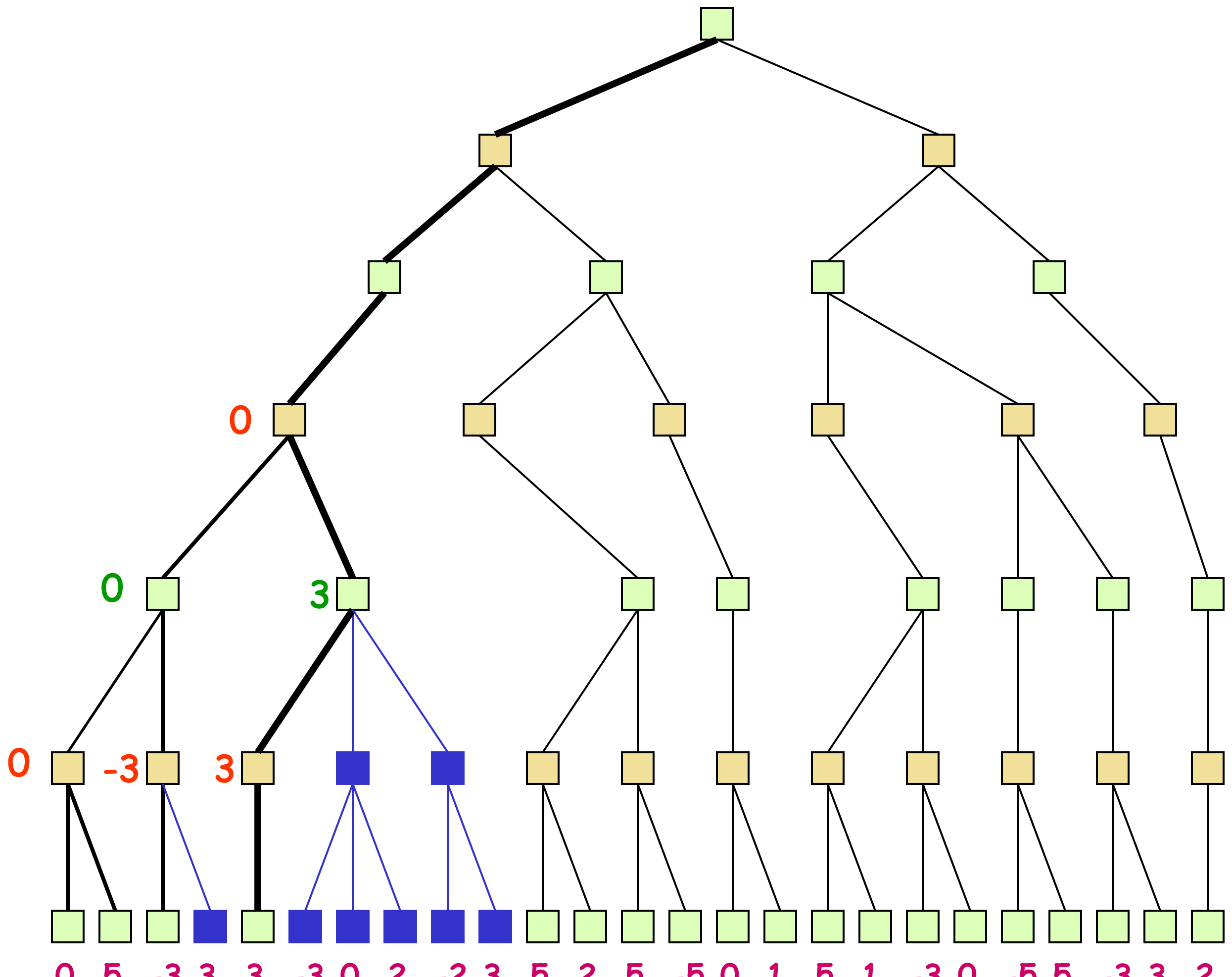

0
0
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

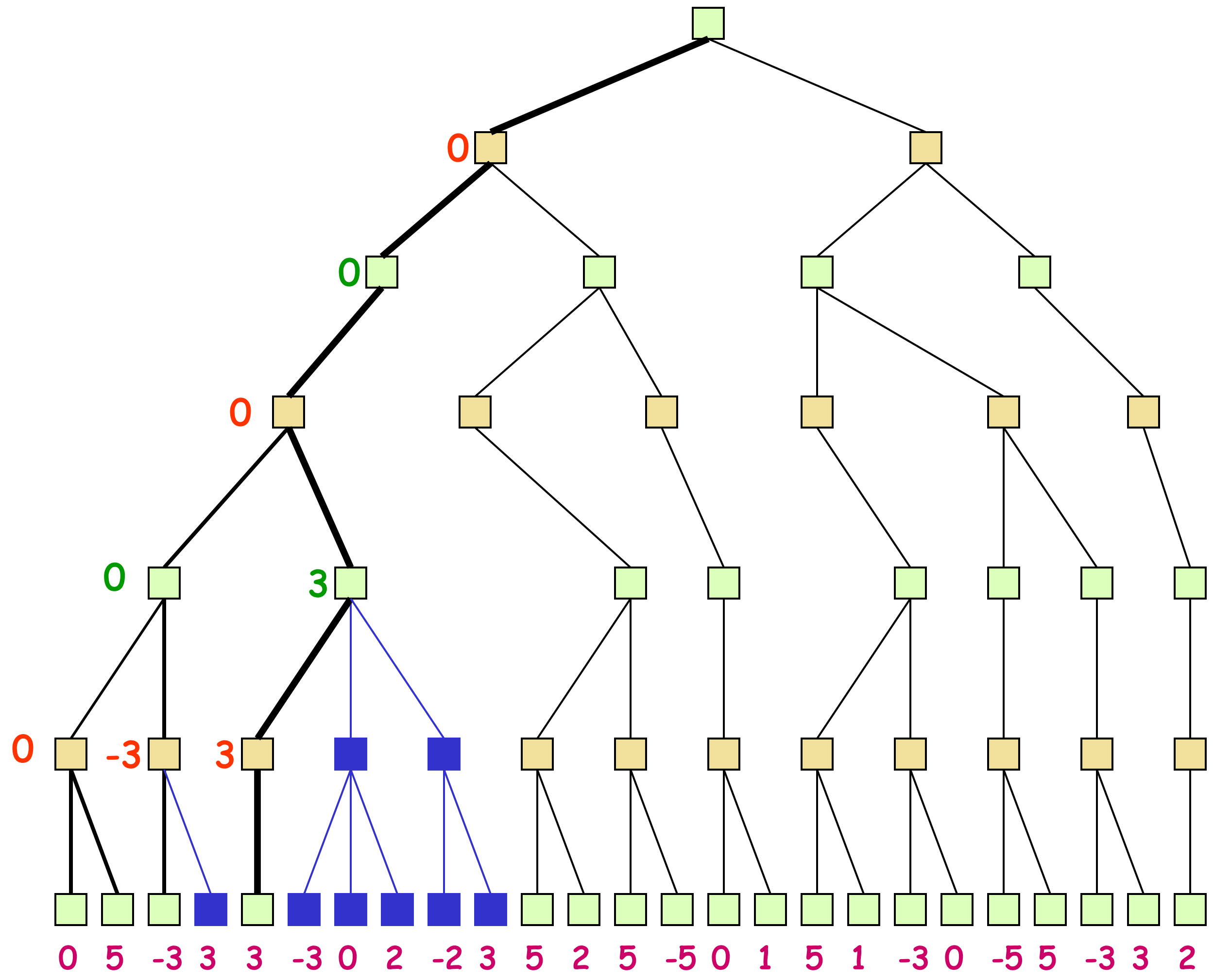

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

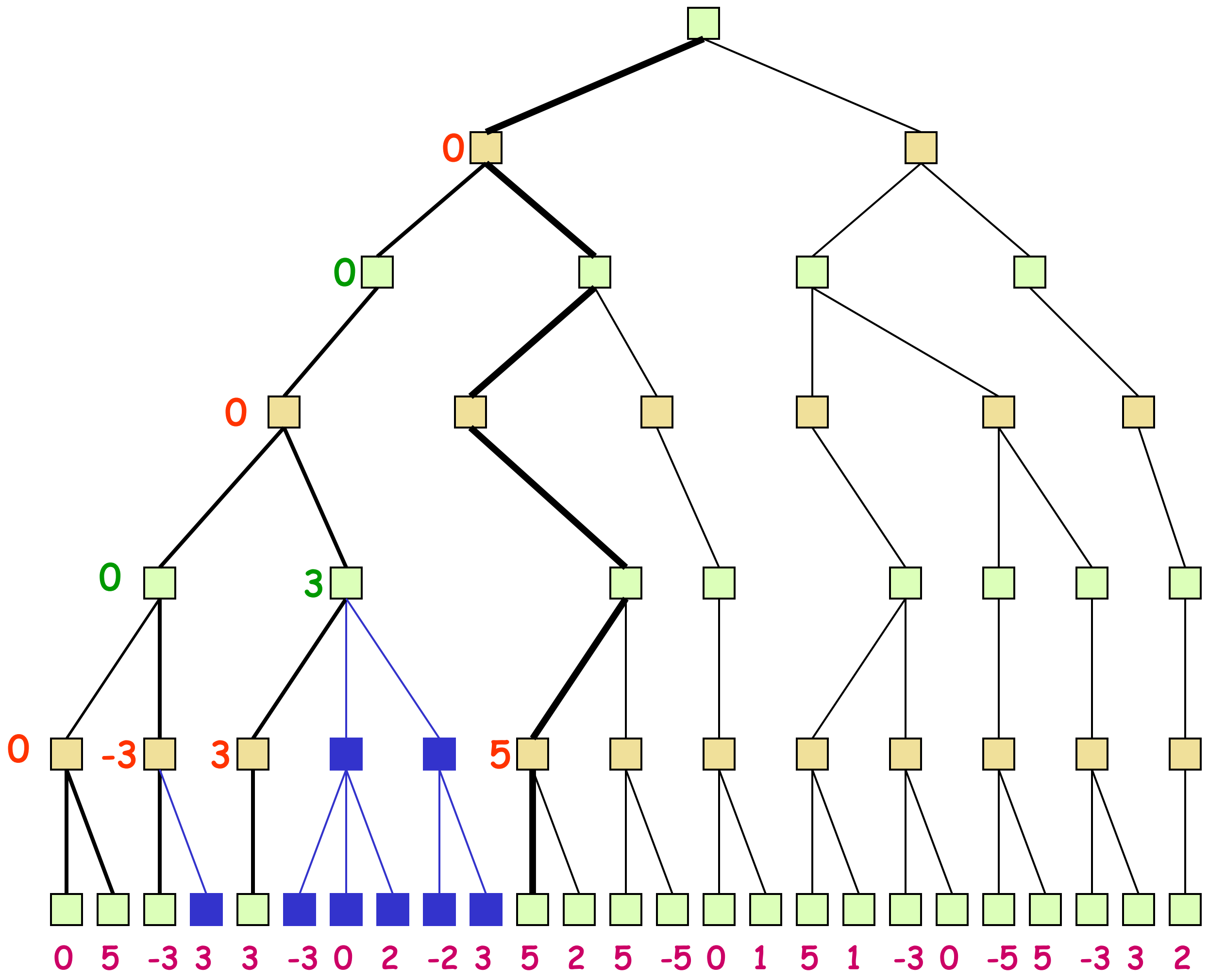

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

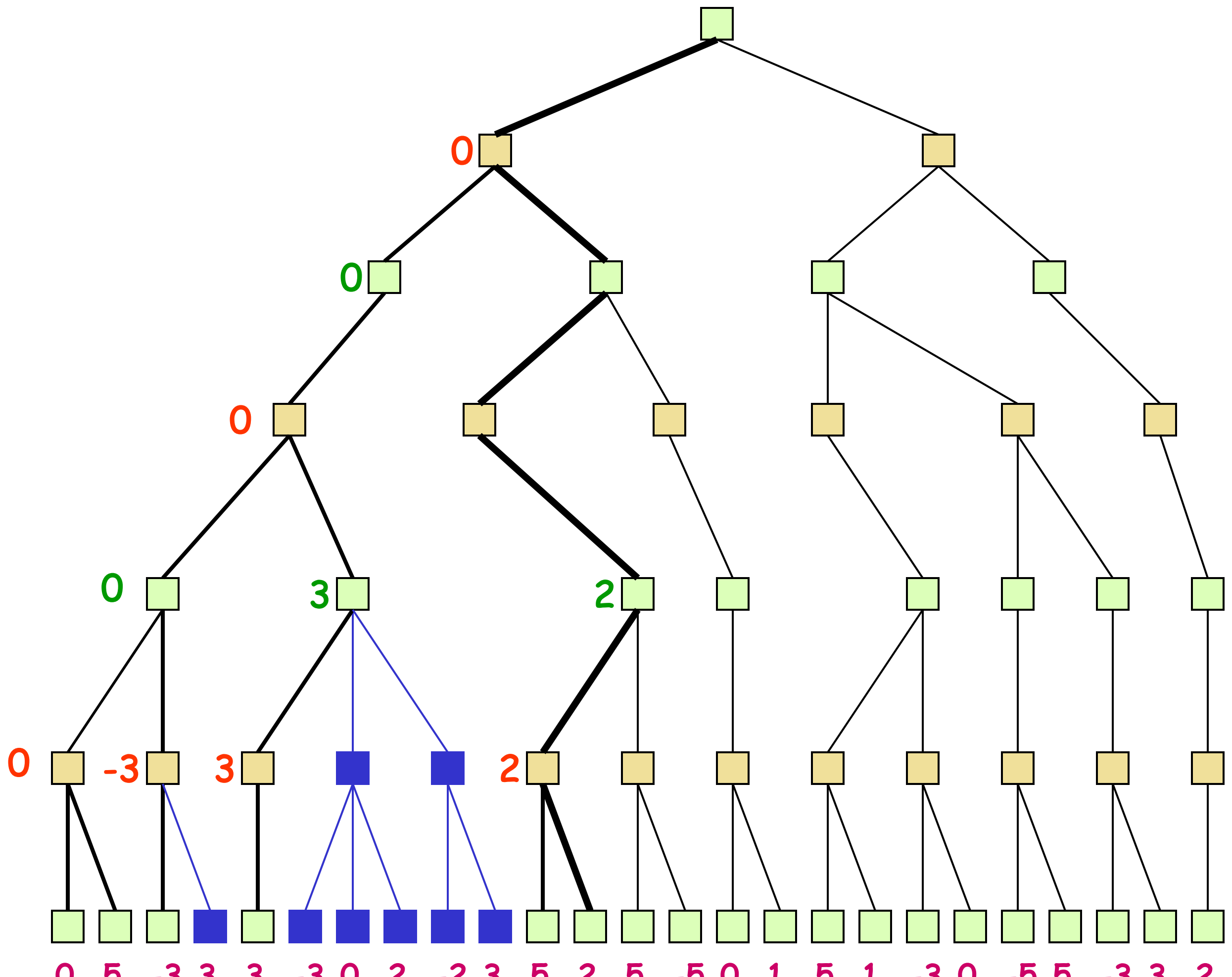

0
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

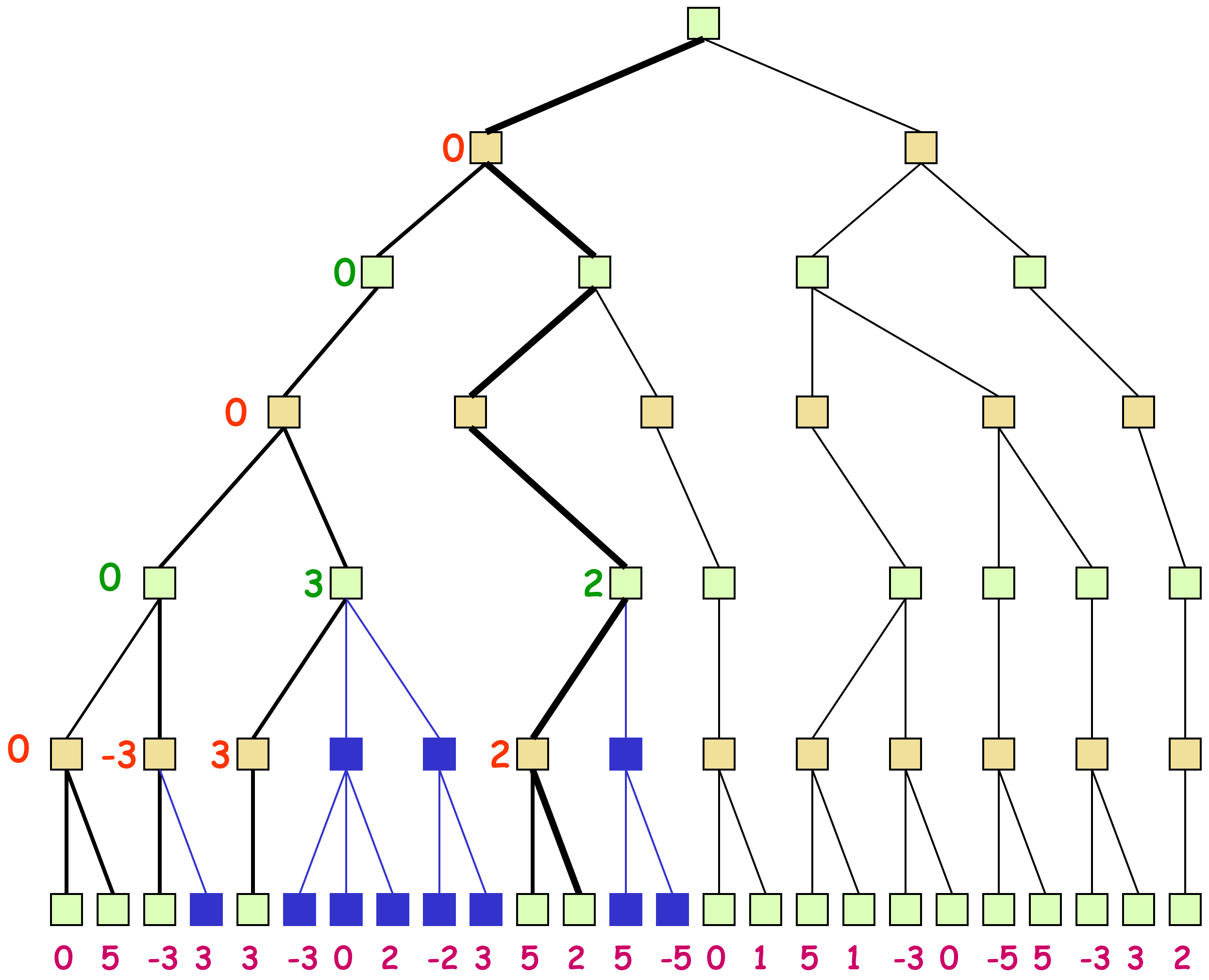

0
0
3
0
-3
3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

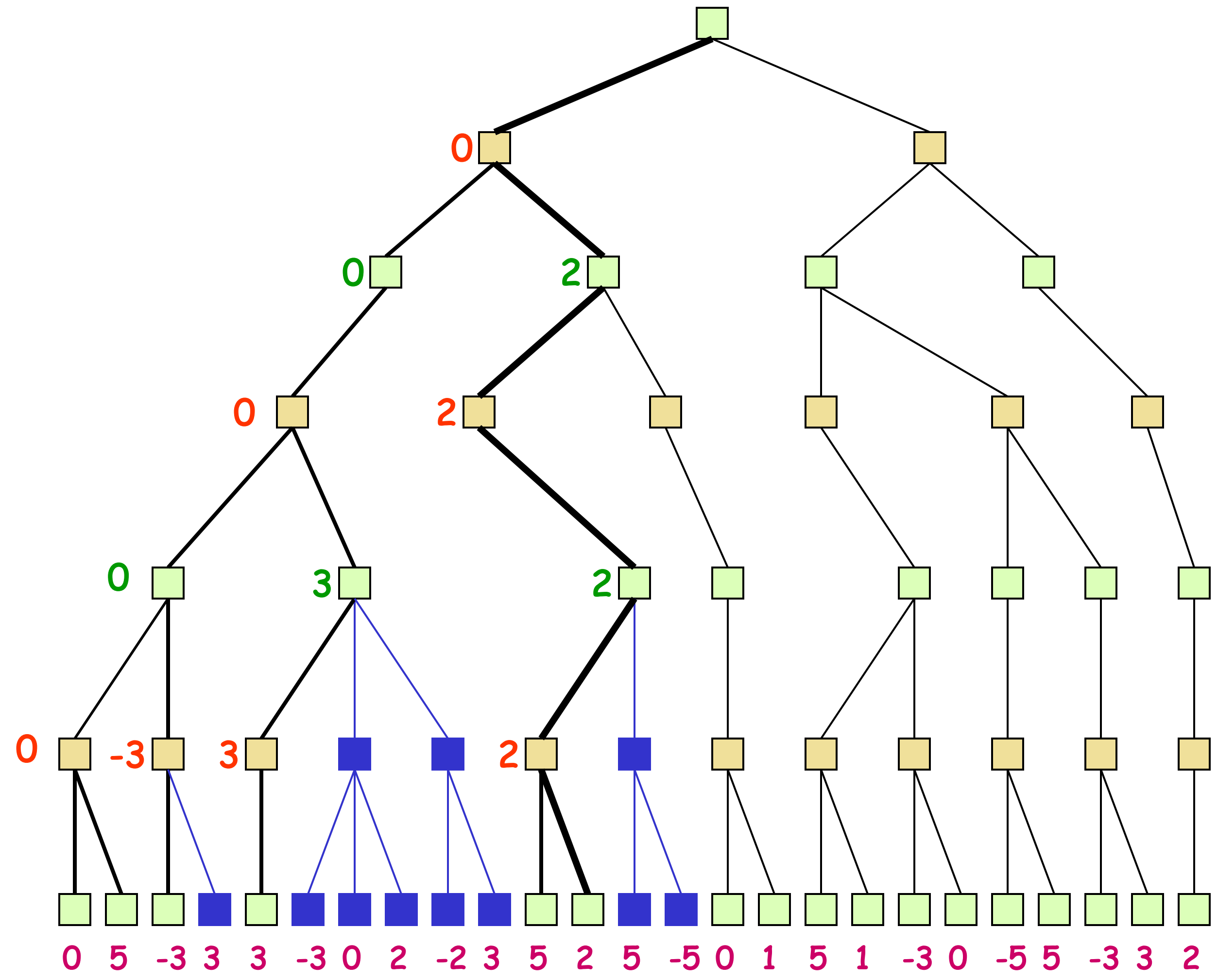

0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

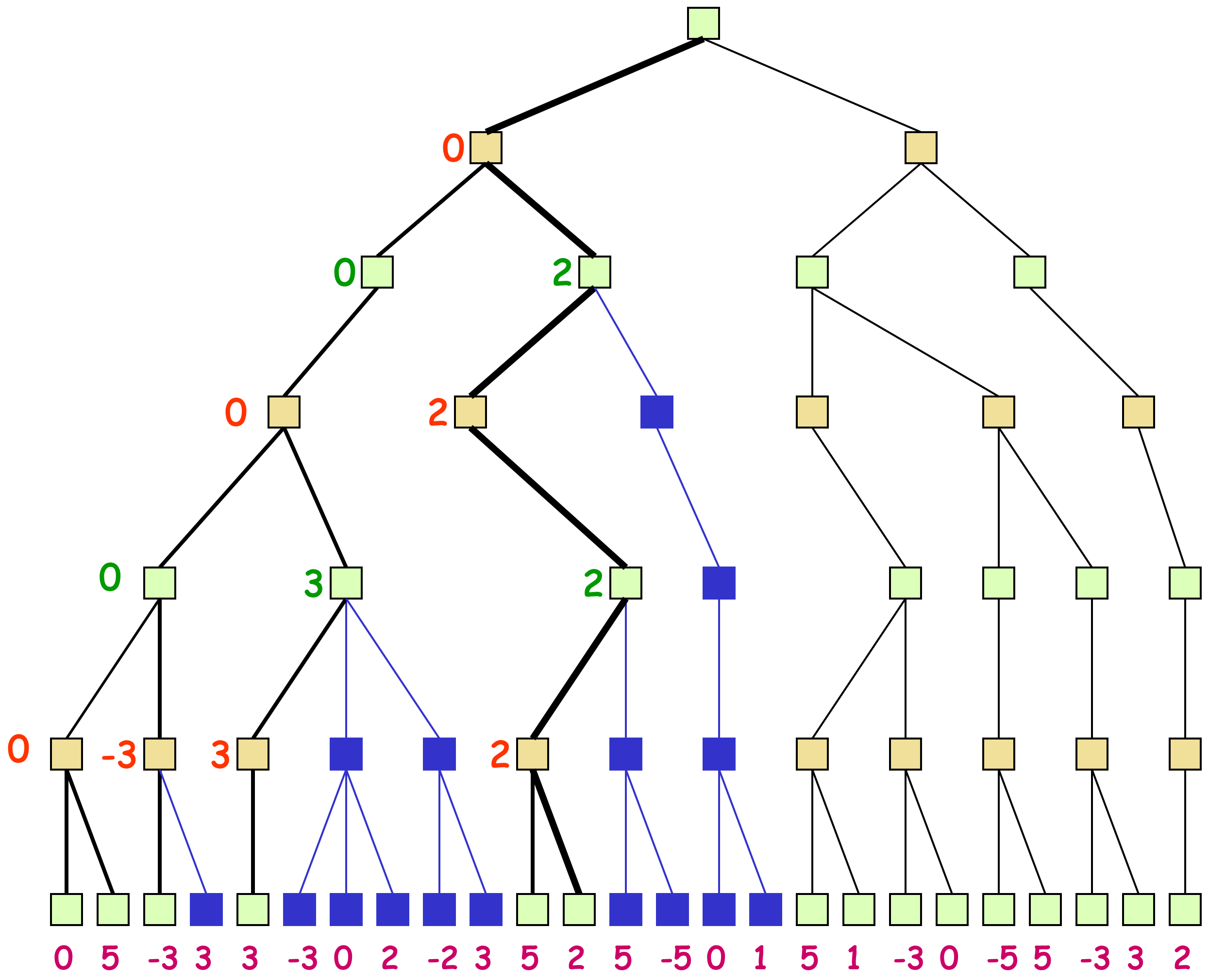

0
0
0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

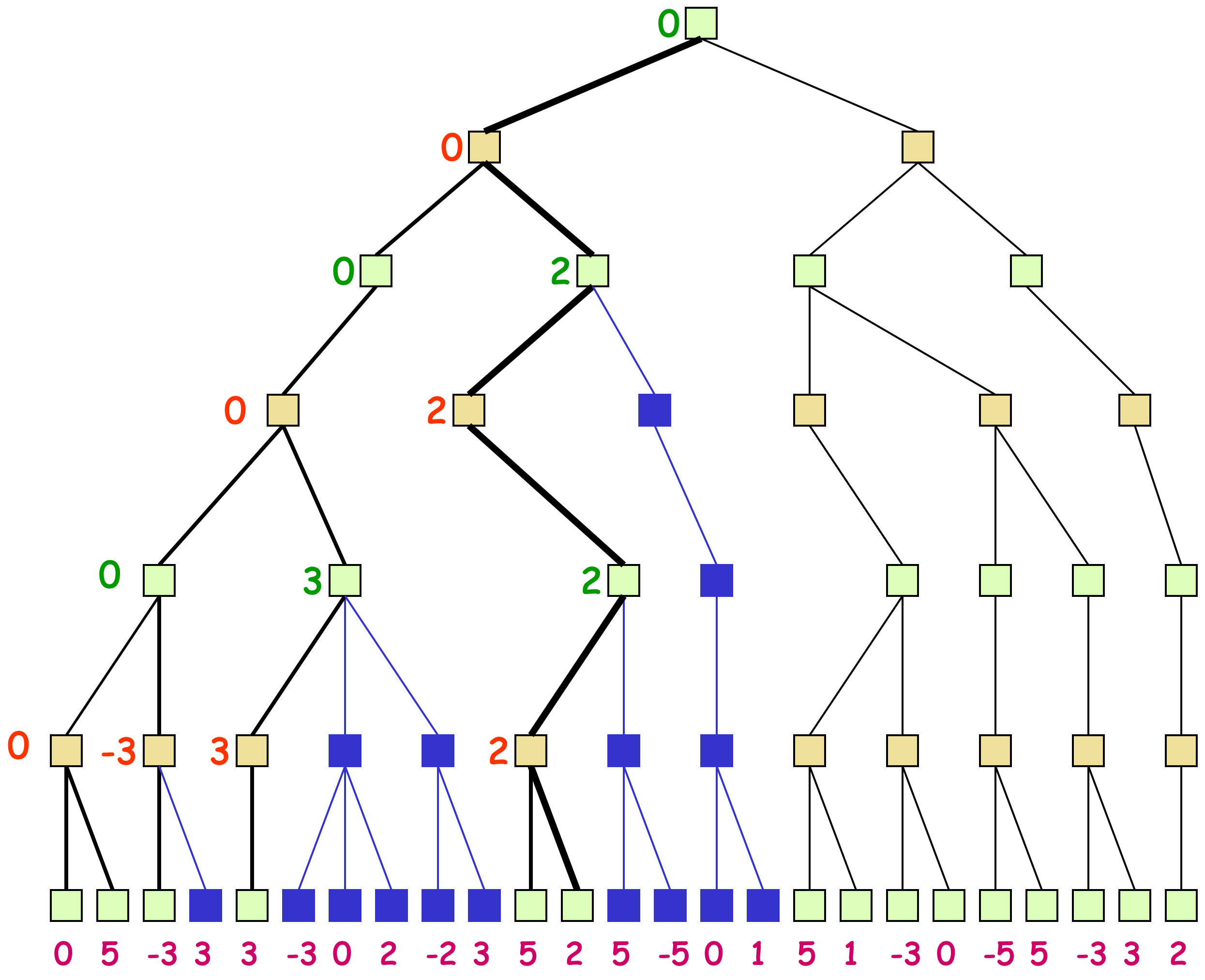

0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

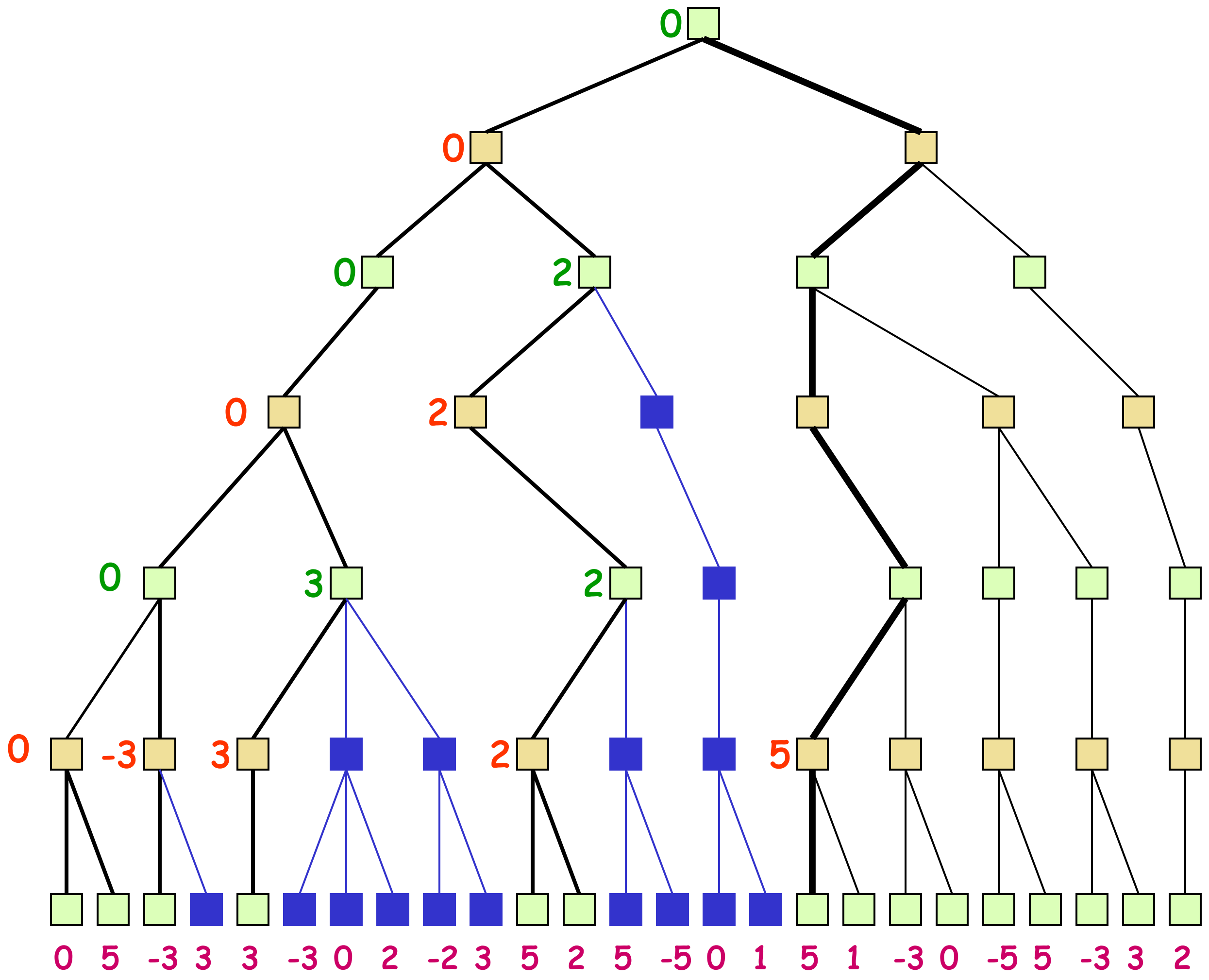

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

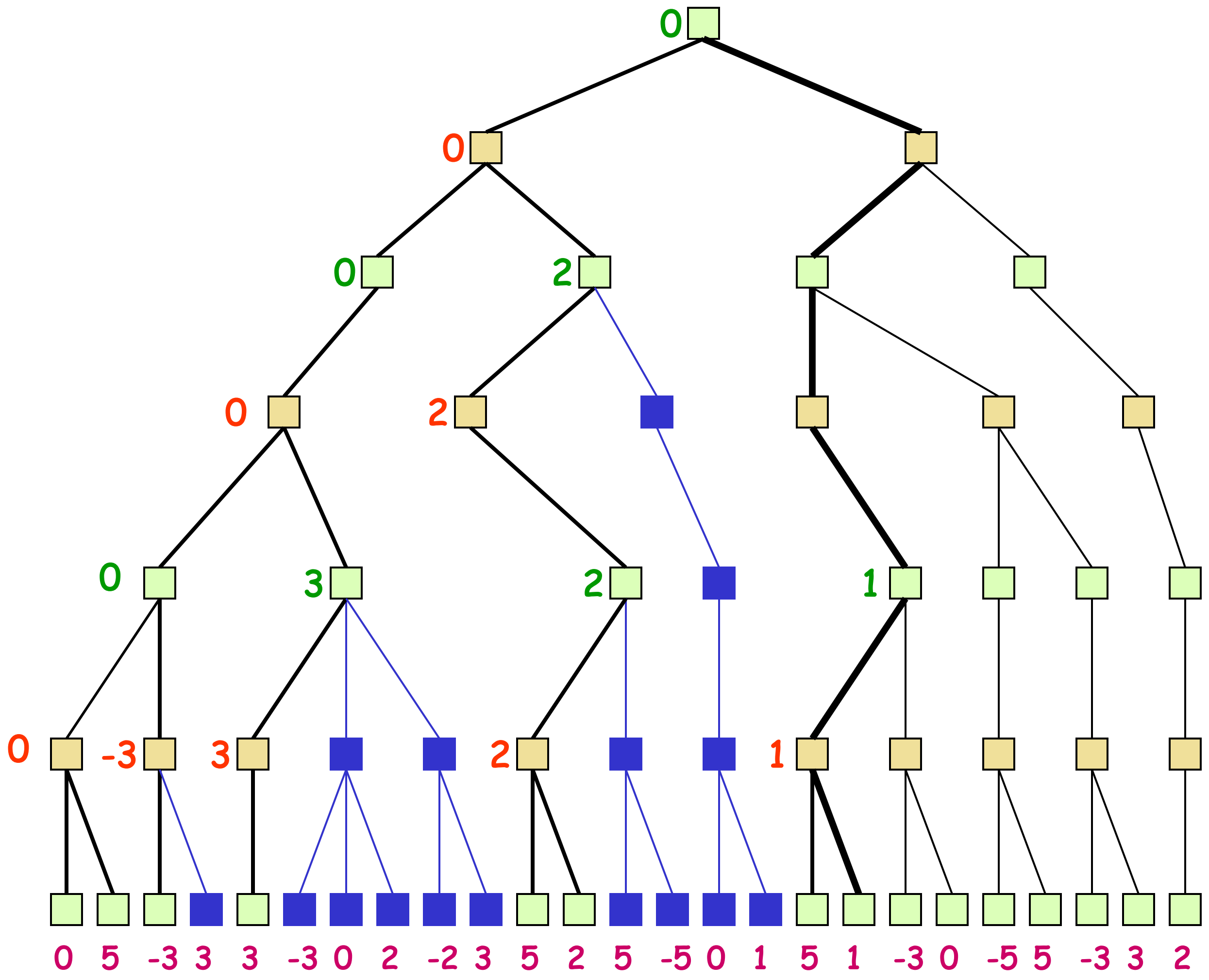

0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


0
0
2
0
2
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

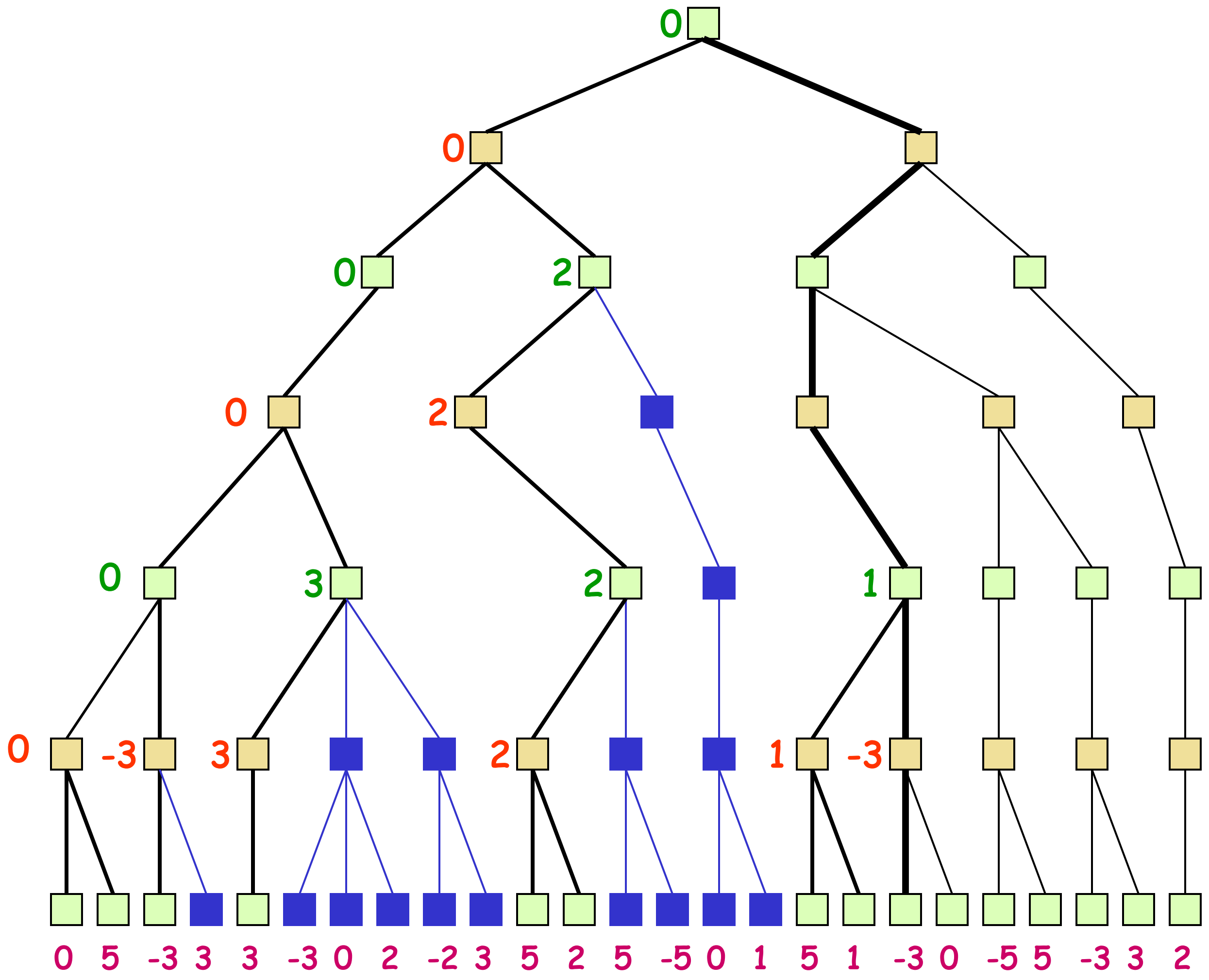

0
0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

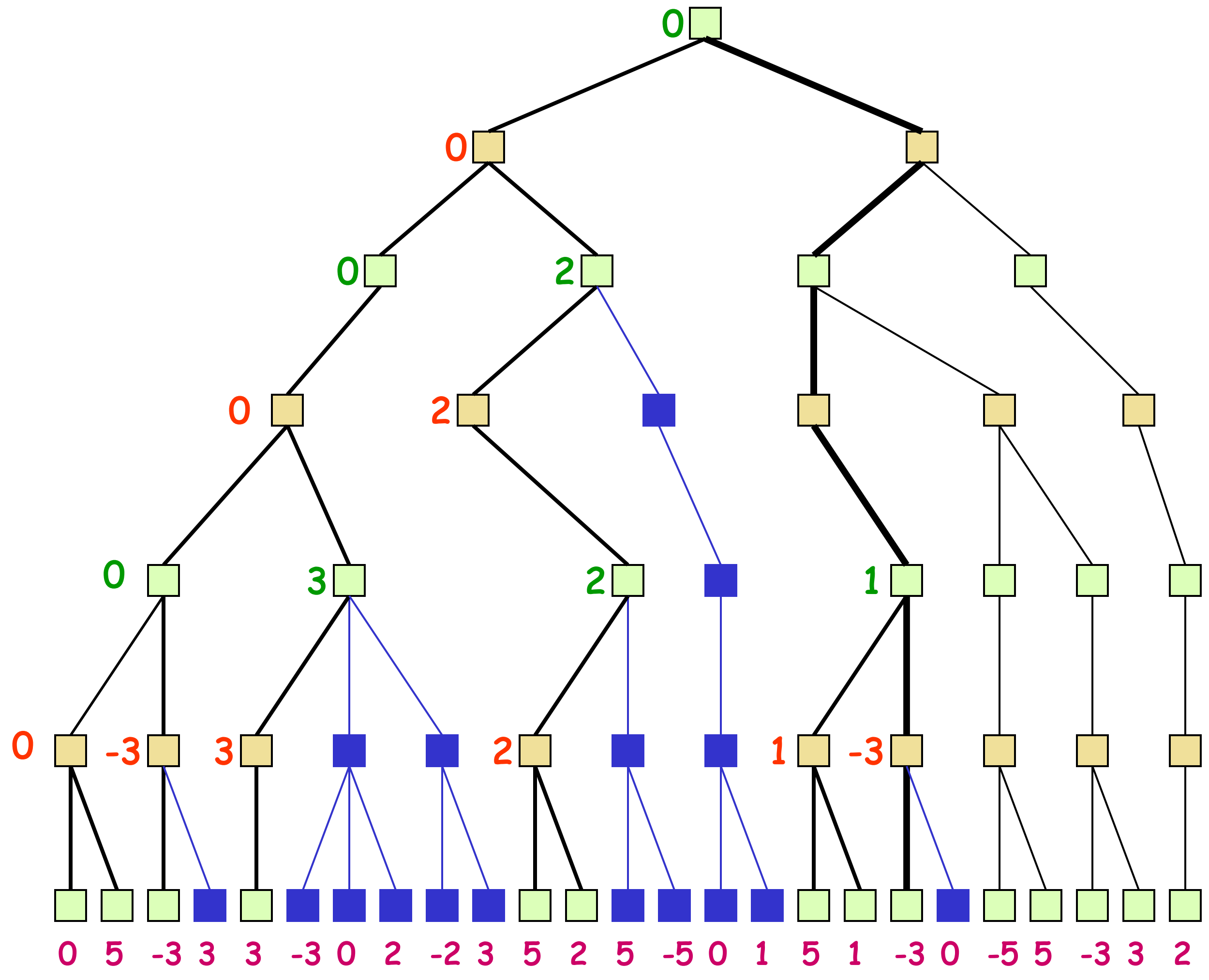


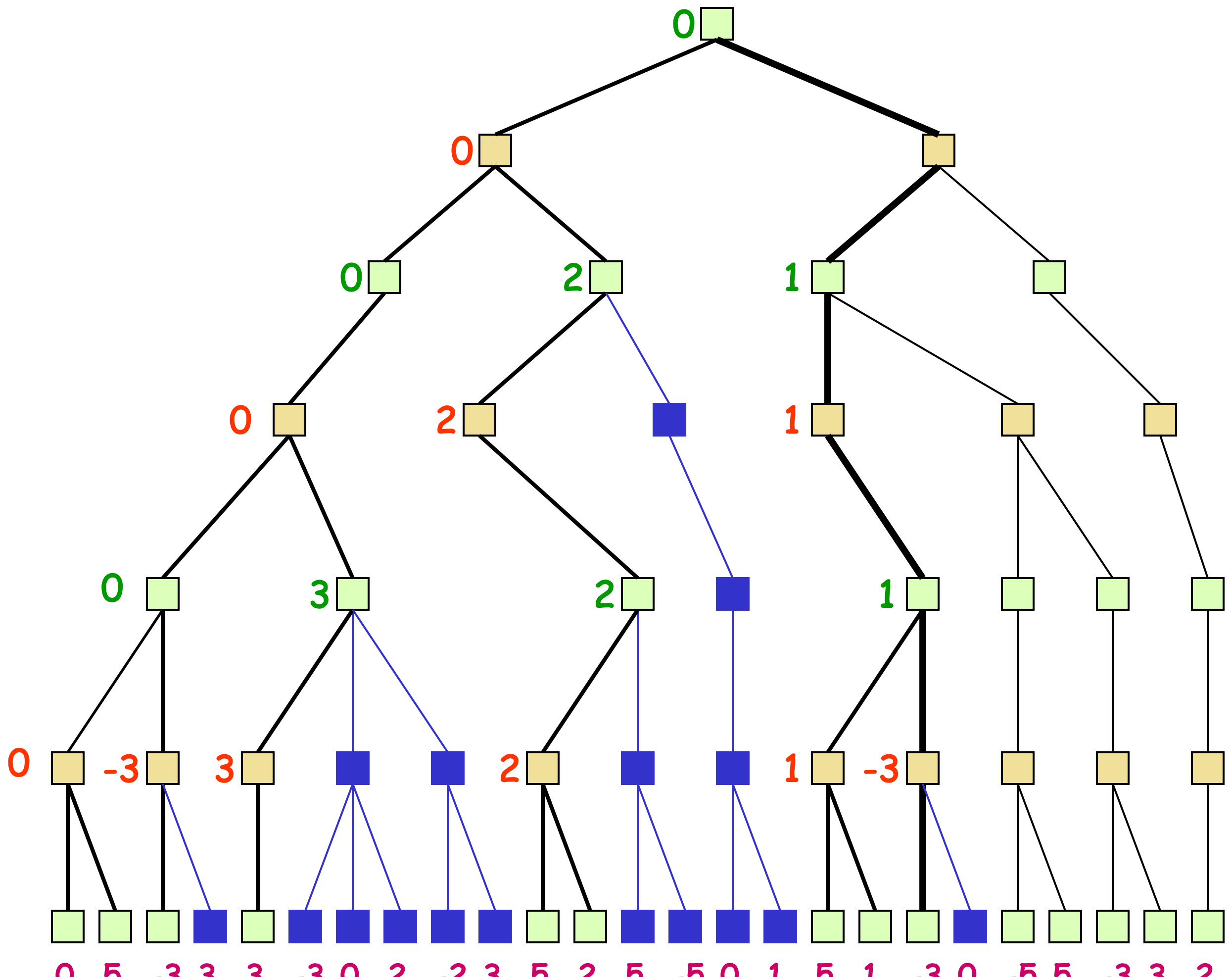
0
0
0
2
0
2
0
3
2
0
-3
3
2
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

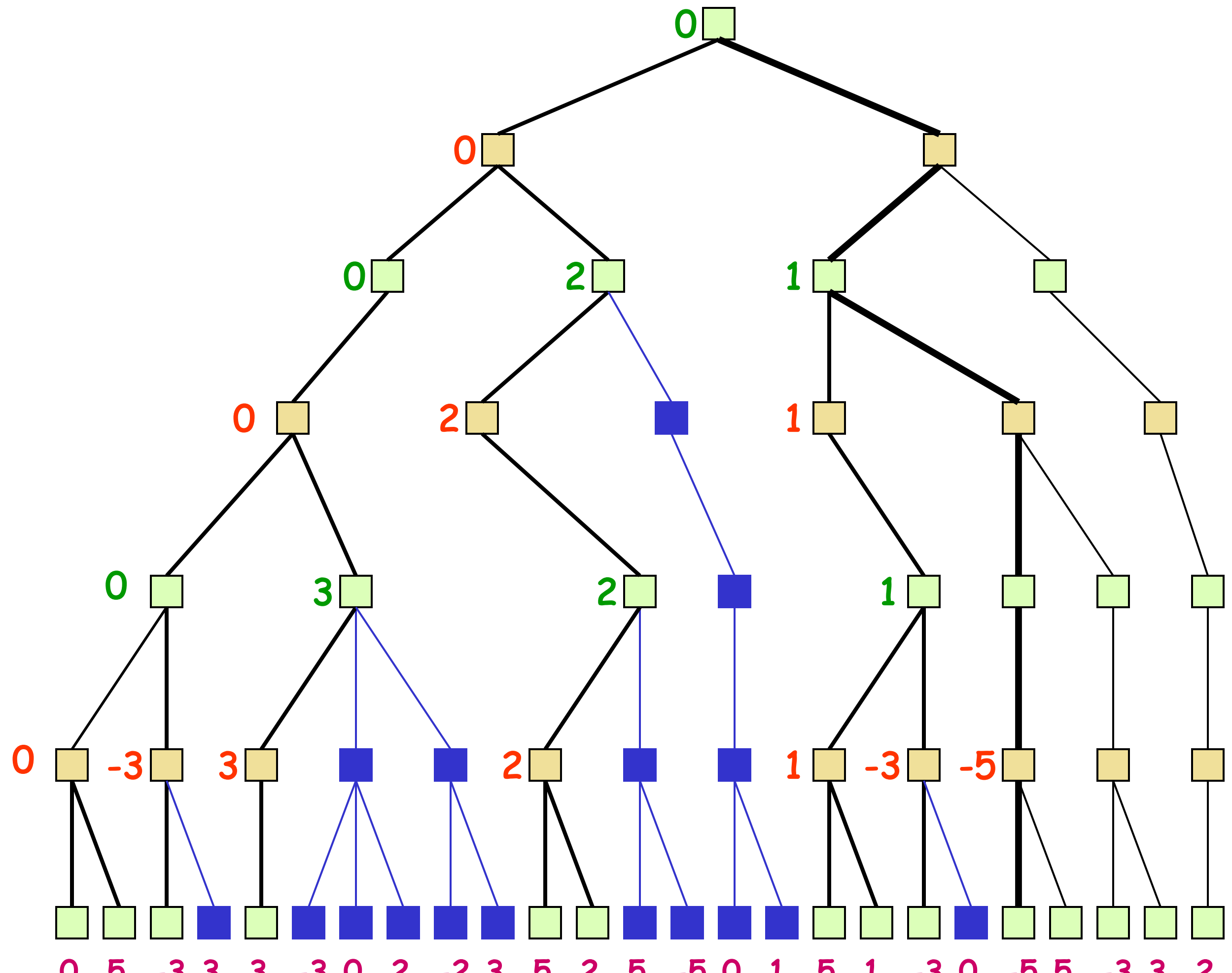


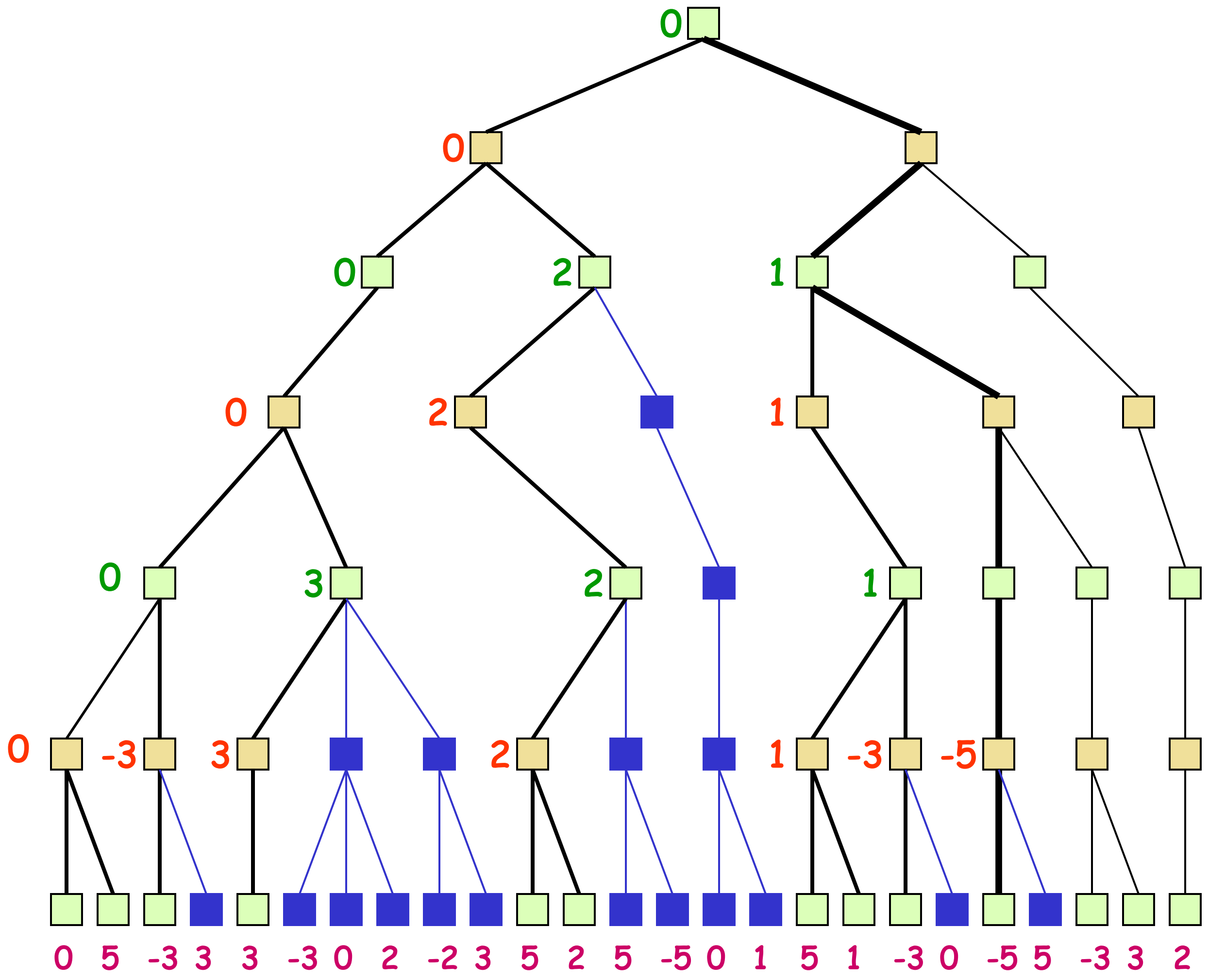
0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

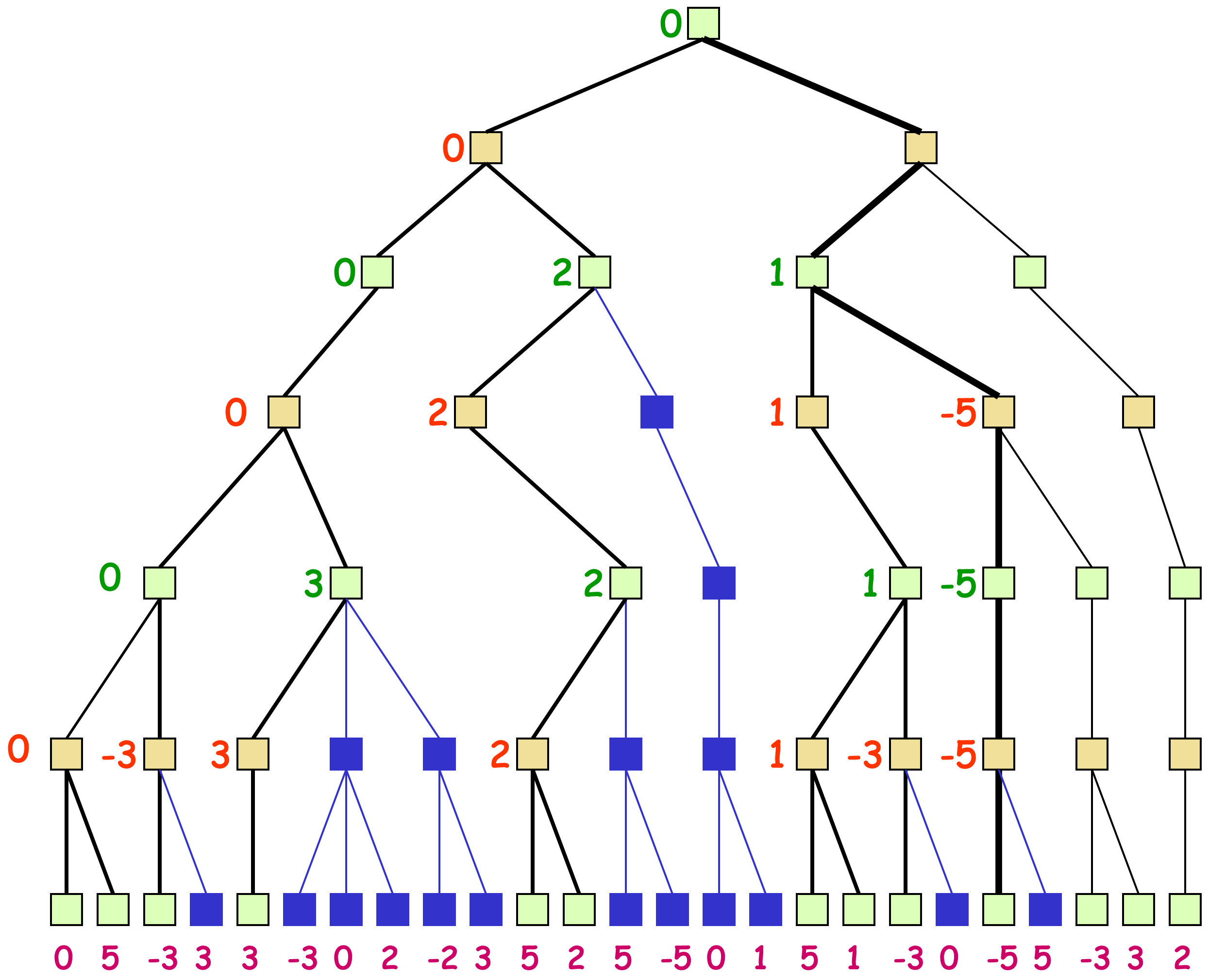

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

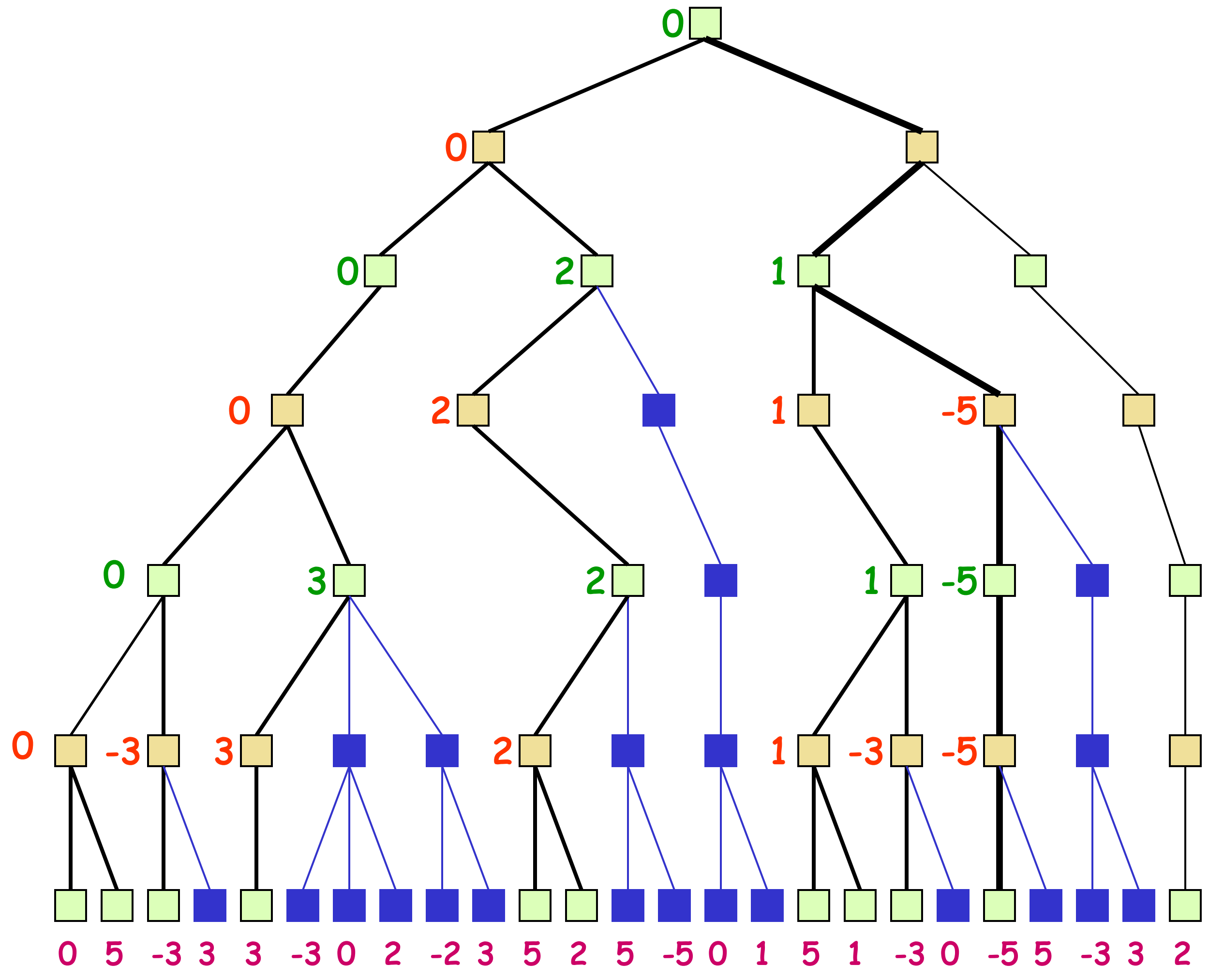

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

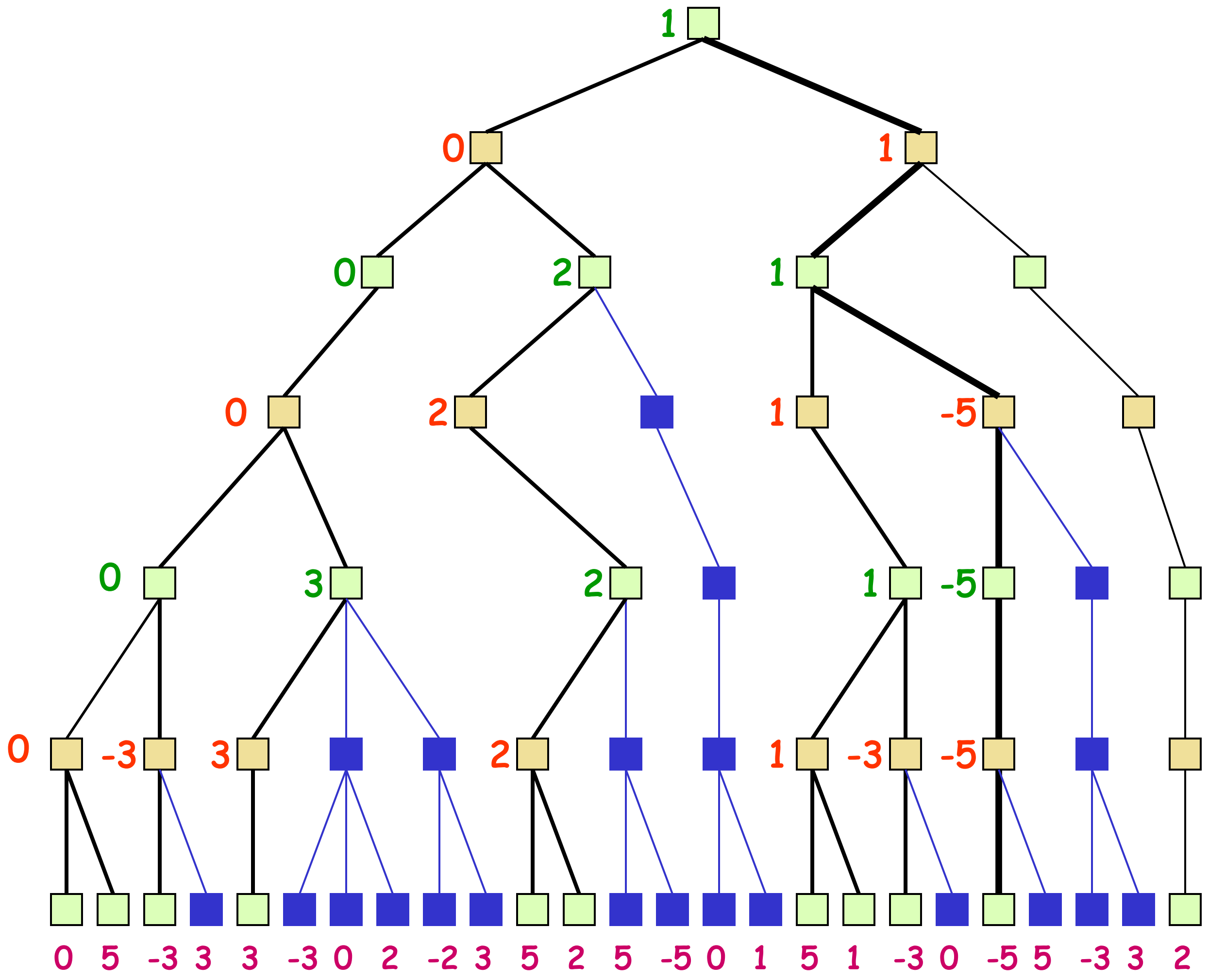

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

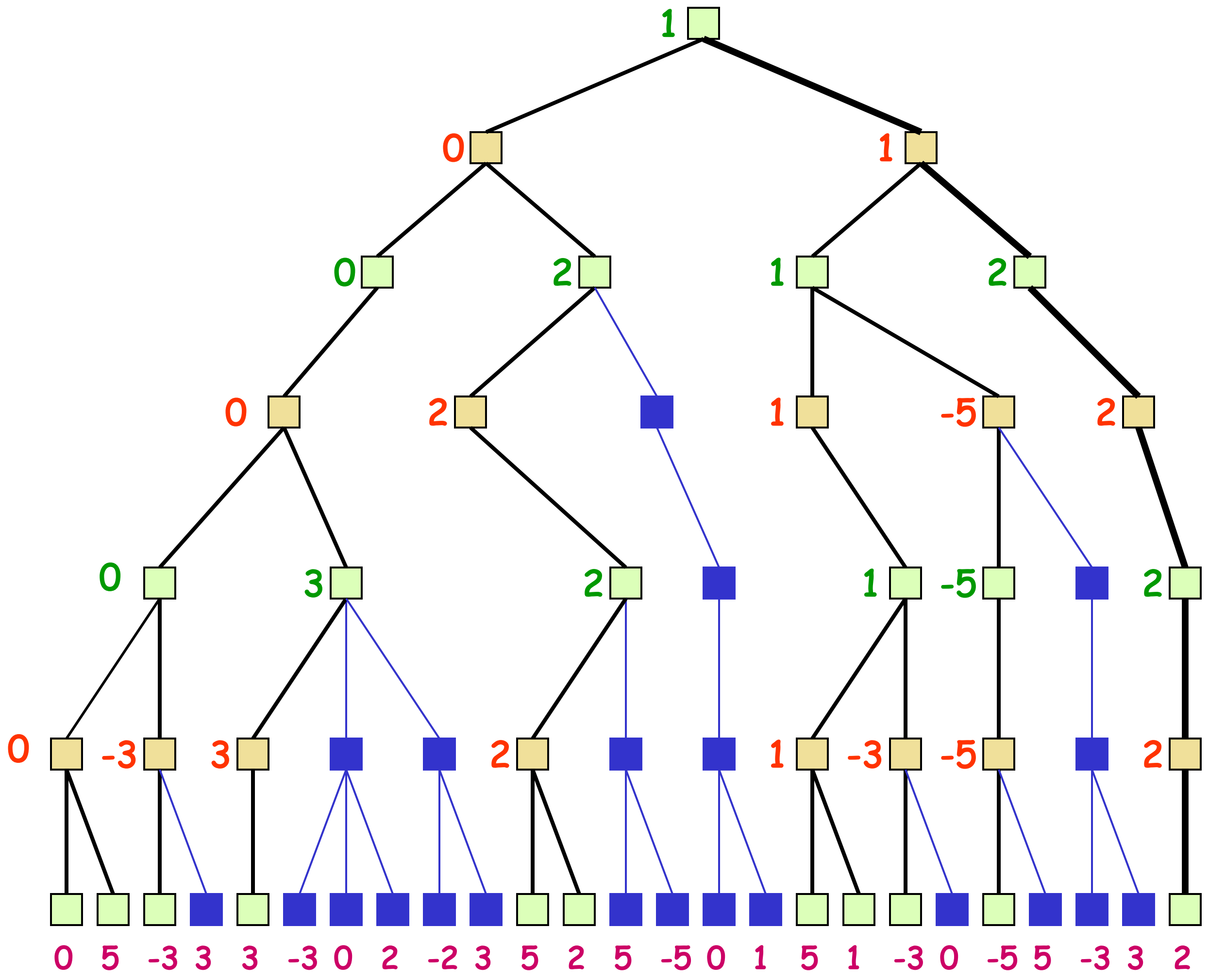

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

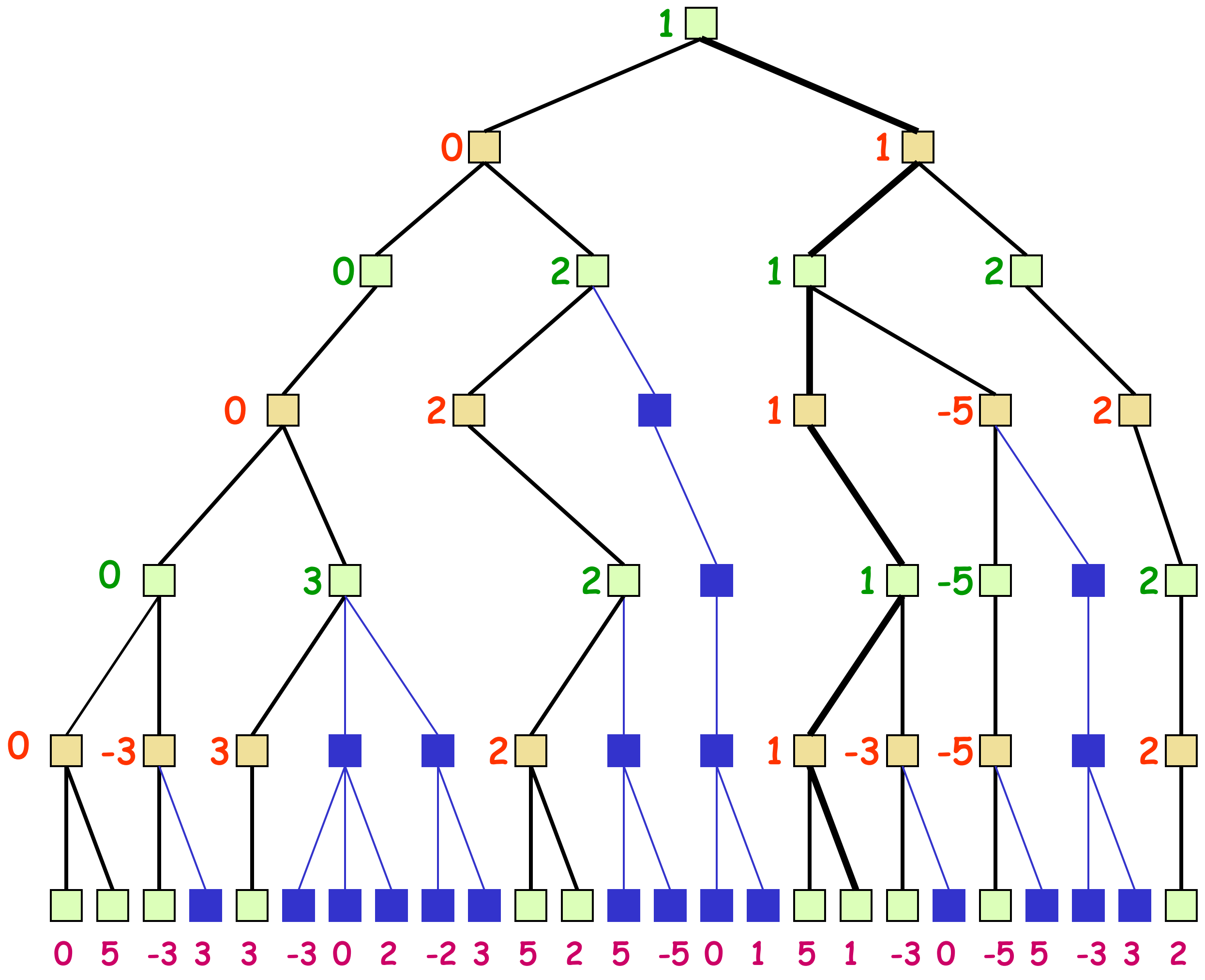

0
0
0
2
1
0
2
1
-5
0
3
2
1
-5
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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


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


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

With alpha-beta we avoided computing a static evaluation metric for 14 of the 25 leaf nodes



Alpha-beta algorithm

```
function MAX-VALUE (state, α, β)
;; α = best MAX so far; β = best MIN
if TERMINAL-TEST (state) then return
 UTILITY(state)
v := -∞
for each s in SUCCESSORS (state) do
    v := MAX (v, MIN-VALUE (s, α, β))
    if v >= β then return v
    α := MAX (α, v)
end
return v

function MIN-VALUE (state, α, β)
if TERMINAL-TEST (state) then return
 UTILITY(state)
v := ∞
for each s in SUCCESSORS (state) do
    v := MIN (v, MAX-VALUE (s, α, β))
    if v <= α then return v
    β := MIN (β, v)
end
return v
```

Effectiveness of alpha-beta

- Alpha-beta guaranteed to compute same value for root node as minimax, but with less computation
- **Worst case:** no pruning, examine $\mathbf{b^d}$ leaf nodes, where nodes have b children and d-ply search is done
- **Best case:** examine only $\mathbf{(2b)^{d/2}}$ leaf nodes
  - You can search twice as deep as minimax!
  - **Occurs if each player's best move is 1st alternative**
- In Deep Blue, alpha-beta pruning reduced *effective branching factor* from ~35 to ~6

Effective branching factor

- Complexity of search problems often dominated by the branching factor of the graph or tree
- That number is in the exponent of the problem size
- Ignoring some node successors, helps
- Effective branching factor is number of successors generated by a "typical" node for a given search problem

Many other improvements

- **Adaptive horizon + iterative deepening**
- **Extended search**: retain k>1 best paths (not just 1) and extend tree at greater depth below their leaf nodes to help deal with "horizon effect"
- **Singular extension**: If move is obviously better than others in node at horizon h, expand it
- Use **transposition tables** to deal with repeated states

Simple Games Summary

- Simple 2-player, zero-sum, deterministic, perfect information games are popular and let us explore adversarial search
- Use a **static evaluator** and **look ahead** to choose move
- Computing static evaluator uses most computing
- **Minimax** makes best choice for next move
- **Alpha-beta** gives same answer, but typically requires much less work