Adversarial Search (aka 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} 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

Chess early days

- **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 game
- **1958**: 1st 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 tournament play

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 4 top poker players in 3-week challenge in casino
- **20?? Bridge**: Expert bridge programs exist, but no world champions yet
- Check out the U. Alberta Games Group

How can we do it?

Classical vs. Statistical approach

- We'll look first at the classical approach used from the 1940s to 2010
- Then at newer statistical approached of which AlphaGo is an example
- These 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) 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

- 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** used to evaluate the "goodness" of a game position

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

- Zero-sum assumption permits single function to describe goodness of board for both players
  - **f(n) >> 0**: position n good for me; bad for you
  - **f(n) << 0**: position n bad for me; 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

- **For Tic-Tac-Toe**

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

  Where 3length is complete row, column or diagonal and an open one has 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 piece values: pawn:1; knight:3; bishop:3; rook:5; queen:9

Evaluation function examples

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

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

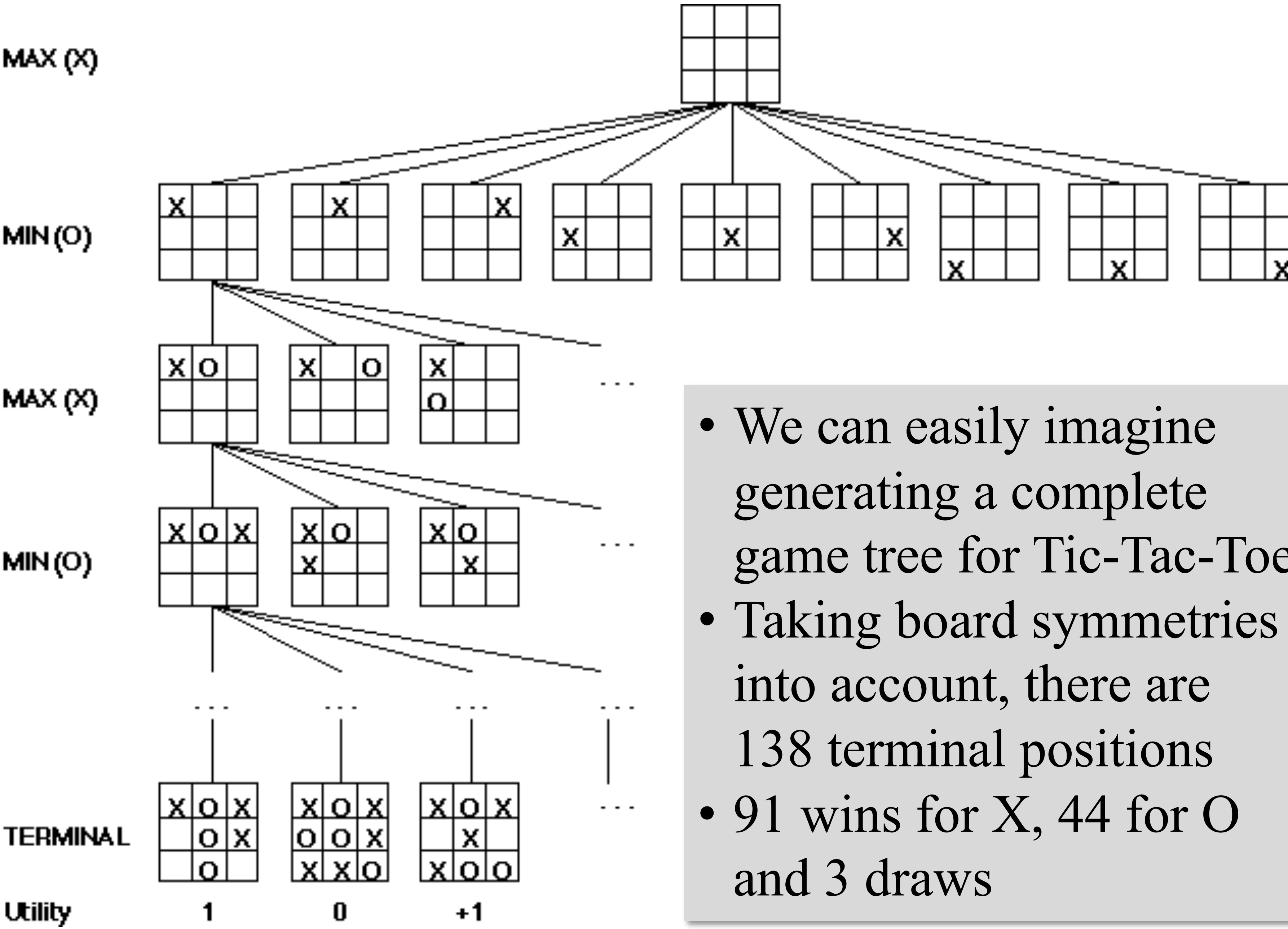
- Example features for chess are piece count, piece values, piece placement, squares controlled, etc.

- IBM's chess program Deep Blue (circa 1996) had >8K features in its evaluation function

But, that's not how people play

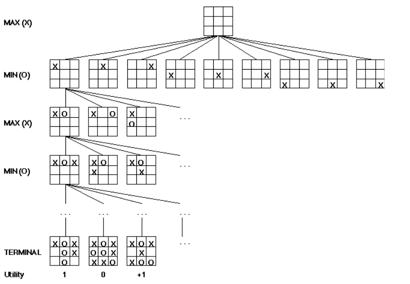
- People use *look ahead*

  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?



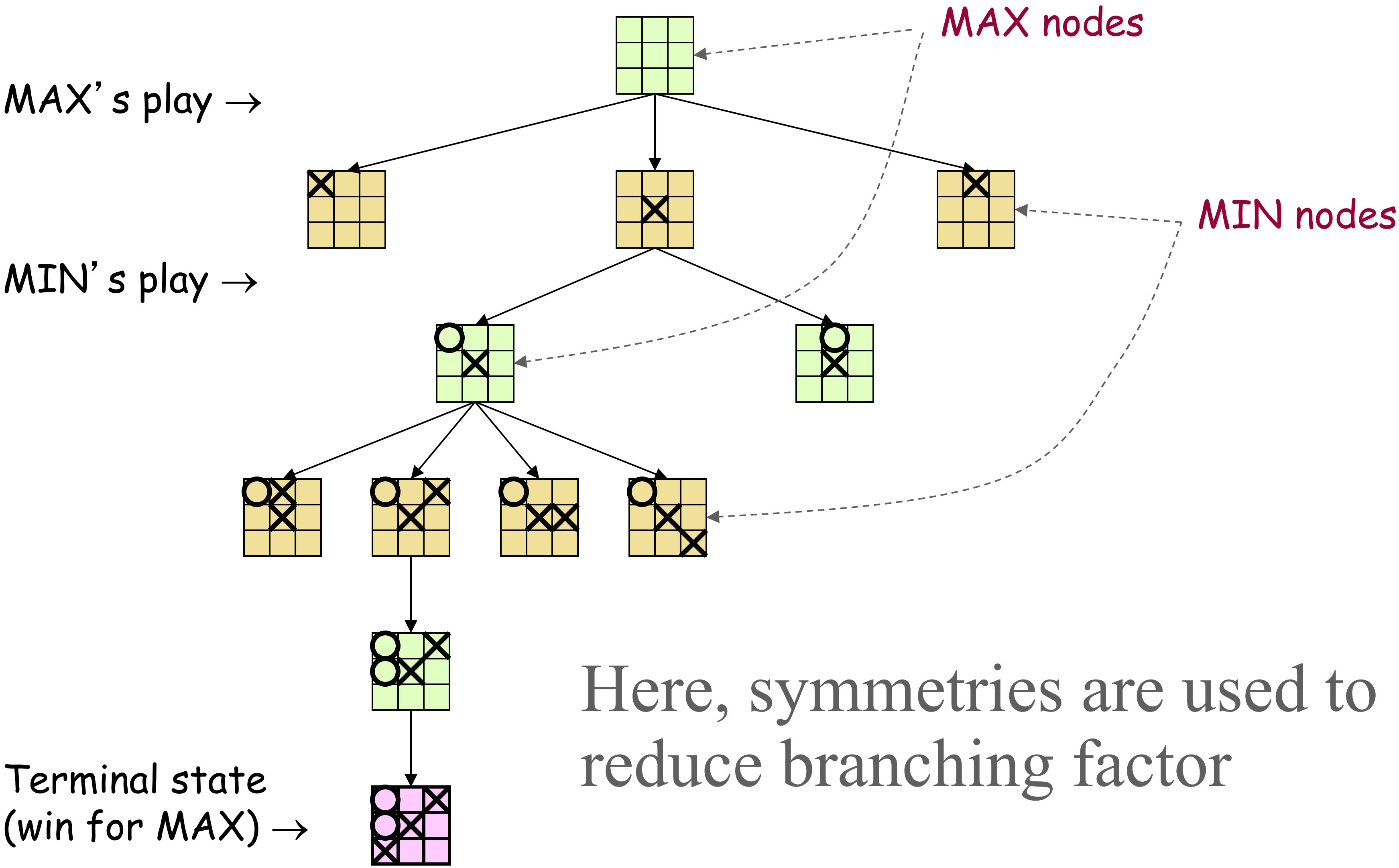
- We can easily imagine generating a complete game tree for Tic-Tac-Toe
- Taking board symmetries into account, there are 138 terminal positions
- 91 wins for X, 44 for O and 3 draws

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, >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 branching factor

Minimax procedure

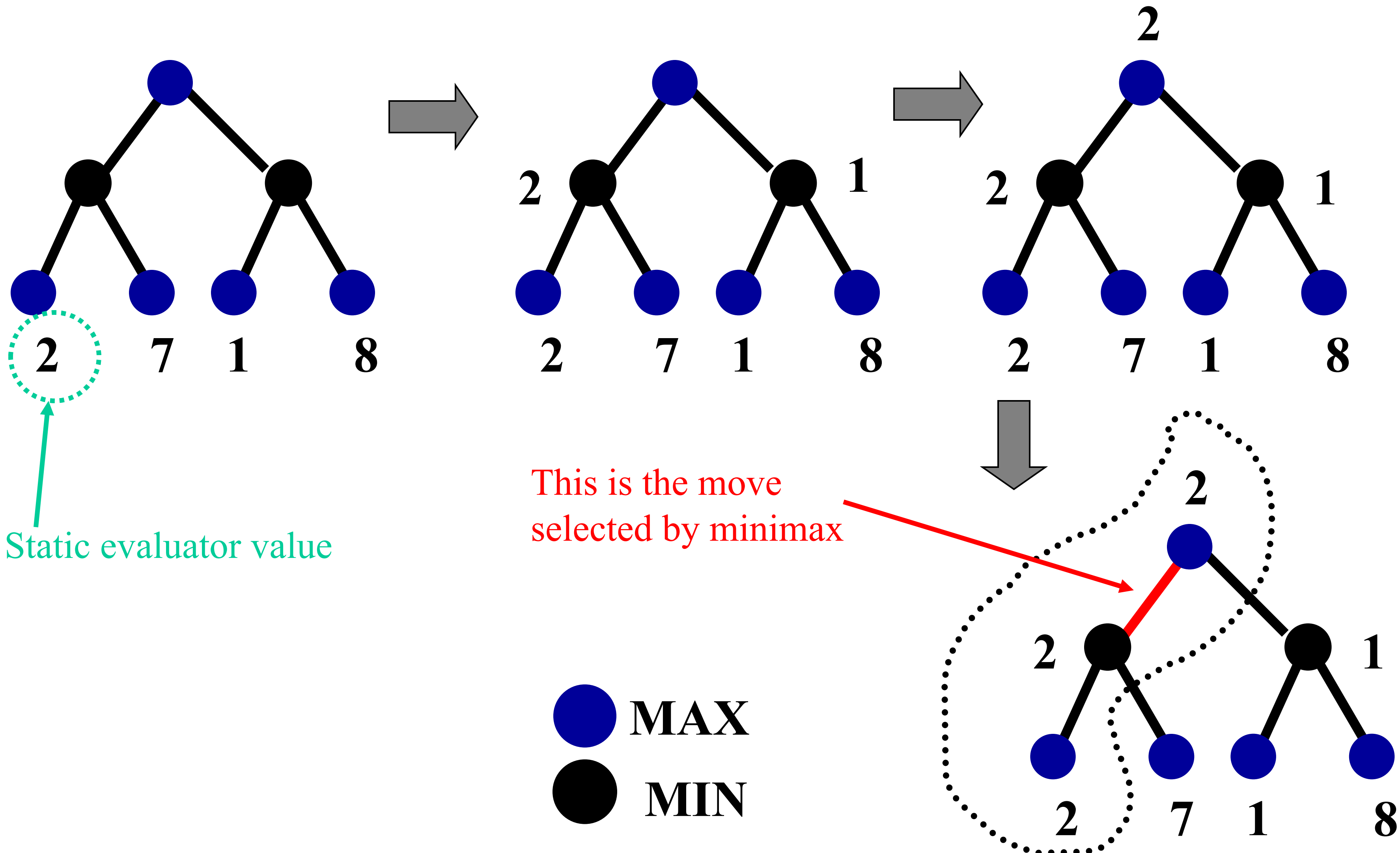
- Create MAX node with current board configuration
- Expand nodes to some **depth** (a.k.a. **plys**) of lookahead in game
- Apply evaluation function at each leaf node
- *Back up* values for each non-leaf node until value is computed for the root node
  - At MIN nodes: value is **minimum** of children's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

Minimax theorem

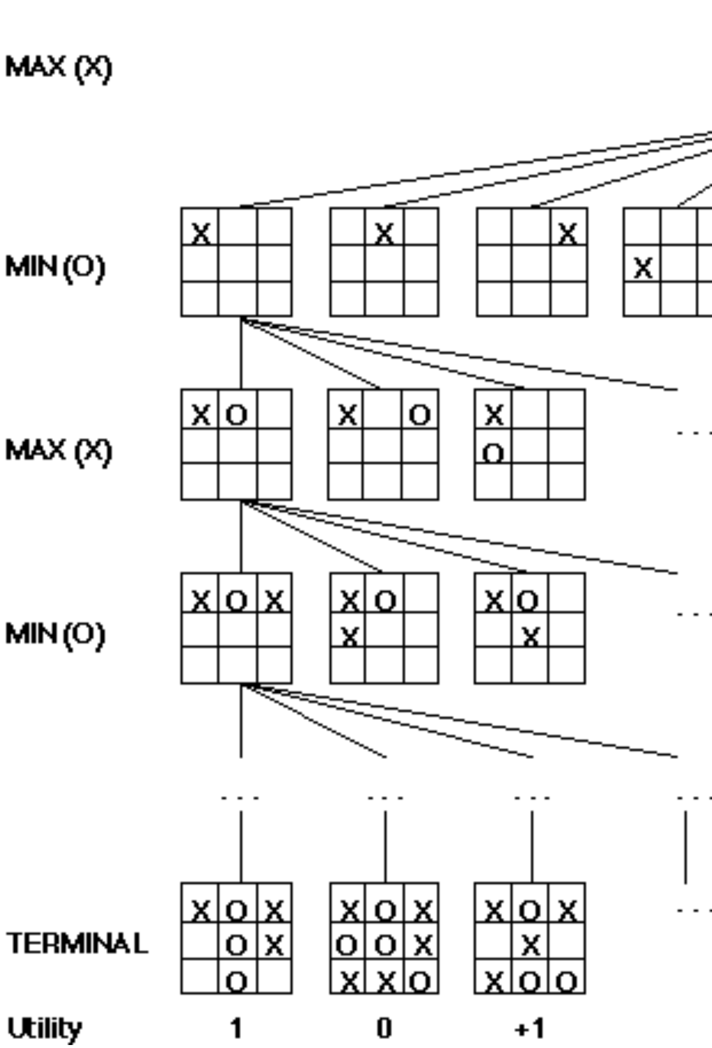
- Intuition: assume your opponent is at least as smart as you and play accordingly
  - If she's not, you can only do better!
- Von Neumann, J: *Zur Theorie der Gesellschafts-spiele* Math. Annalen. **100** (1928) 295-320

  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

Minimax Algorithm



Partial Game Tree for Tic-Tac-Toe



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

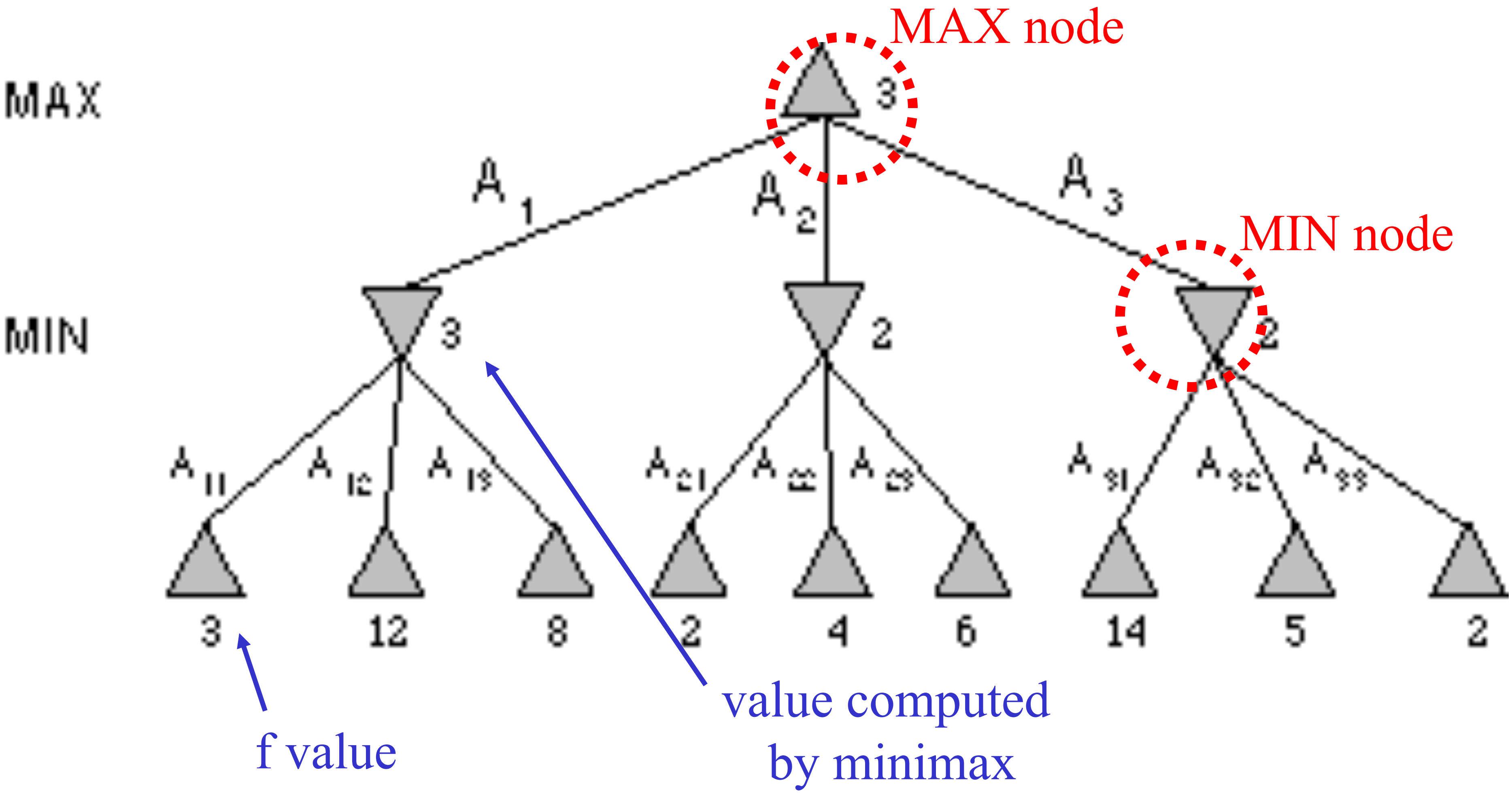
f(n)=-1 if position a win for O

f(n)=0 if position a draw

Why use backed-up values?

- **Intuition:** if evaluation function 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 lookup horizon **h** because time to choose move is limited

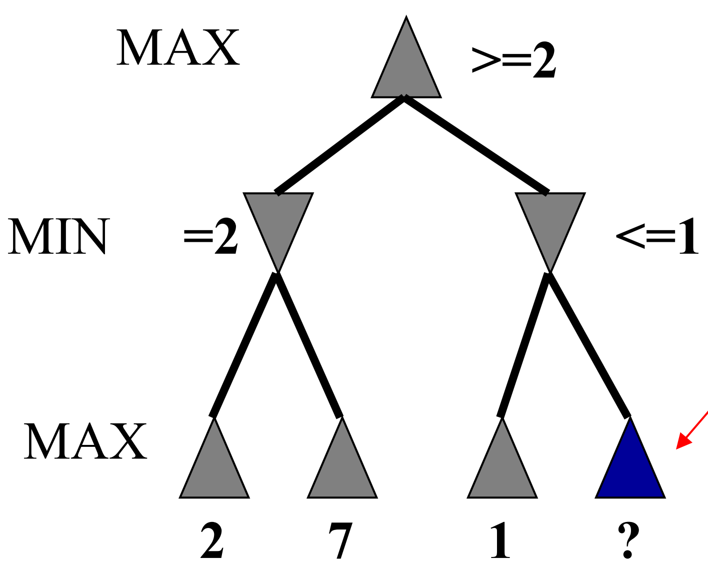
Minimax Tree



Is that all there is to simple games?

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



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

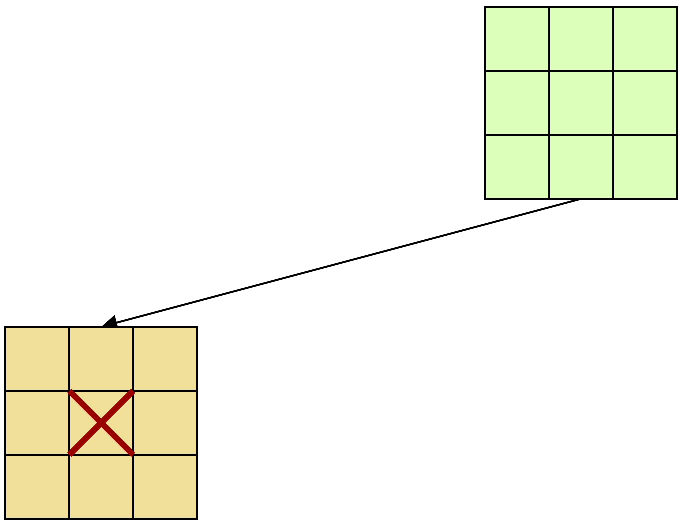
Alpha-beta pruning

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

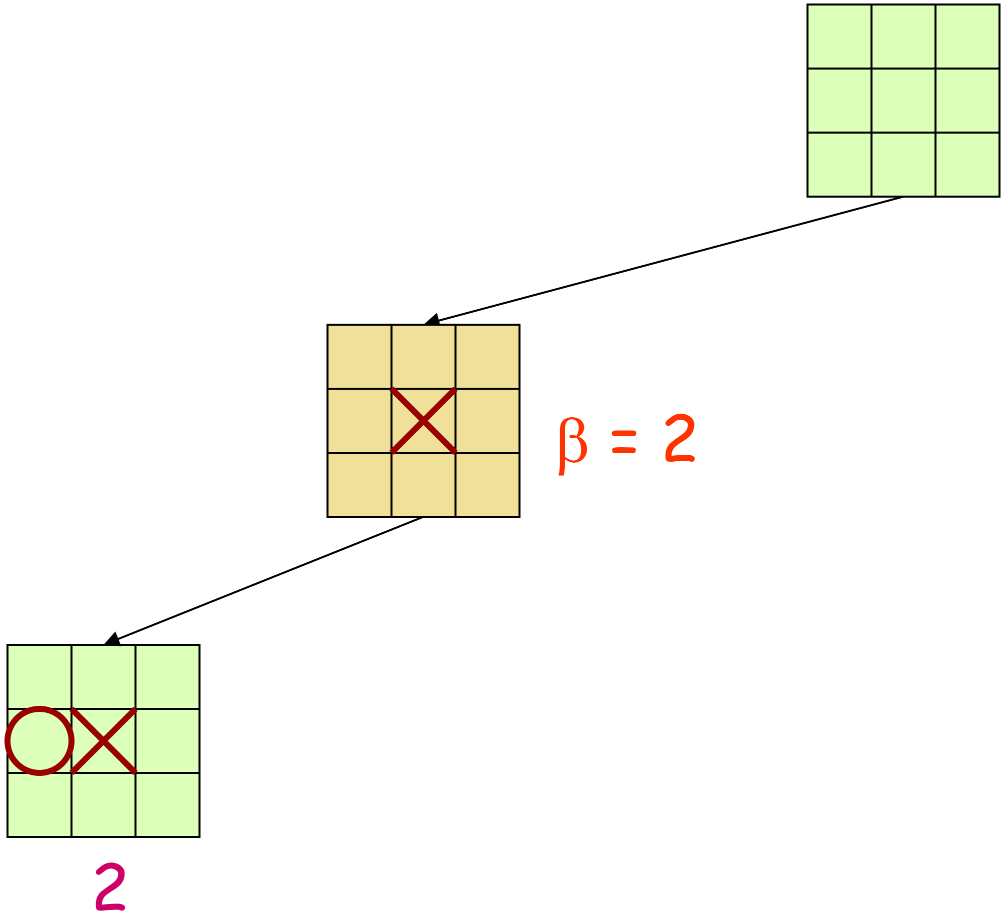
  Alpha values start at -∞ and only increase
- At **MIN** node n, **beta(n)** = min value found so far

  Beta values start at +∞ and only decrease
- **Beta cutoff**: stop search below MAX node N (i.e., don't examine more children) 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

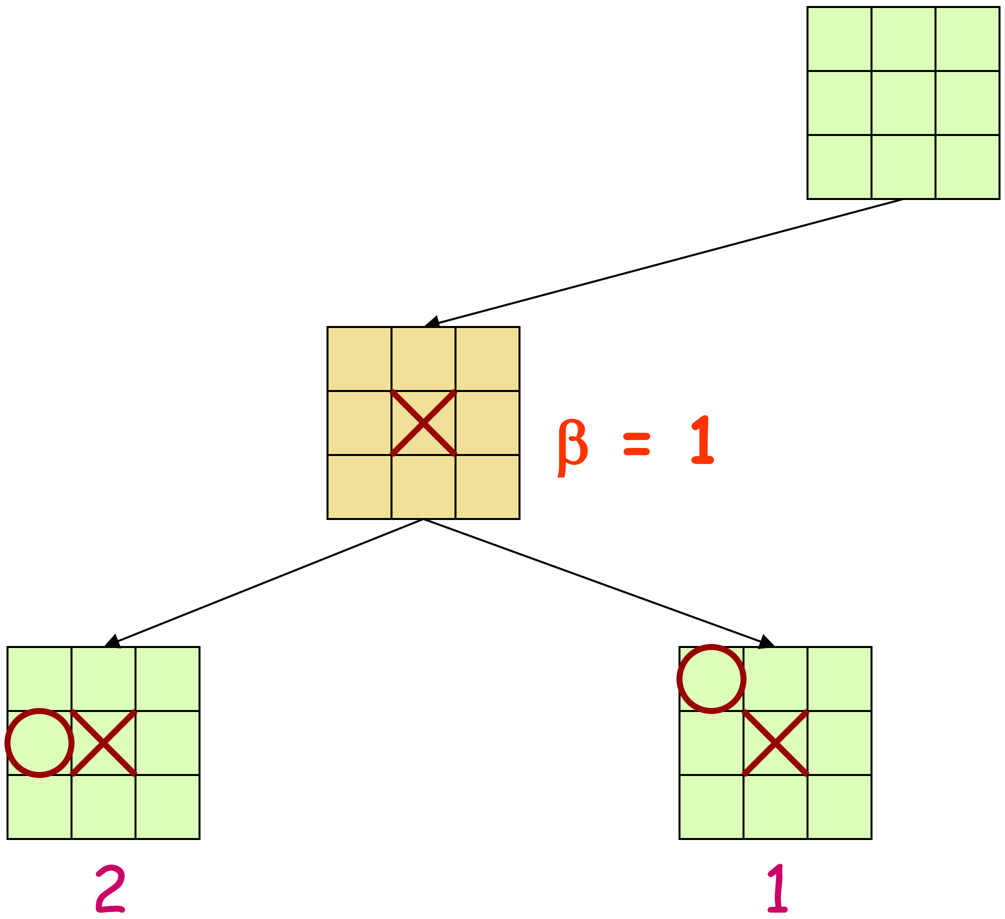


Alpha-Beta Tic-Tac-Toe Example



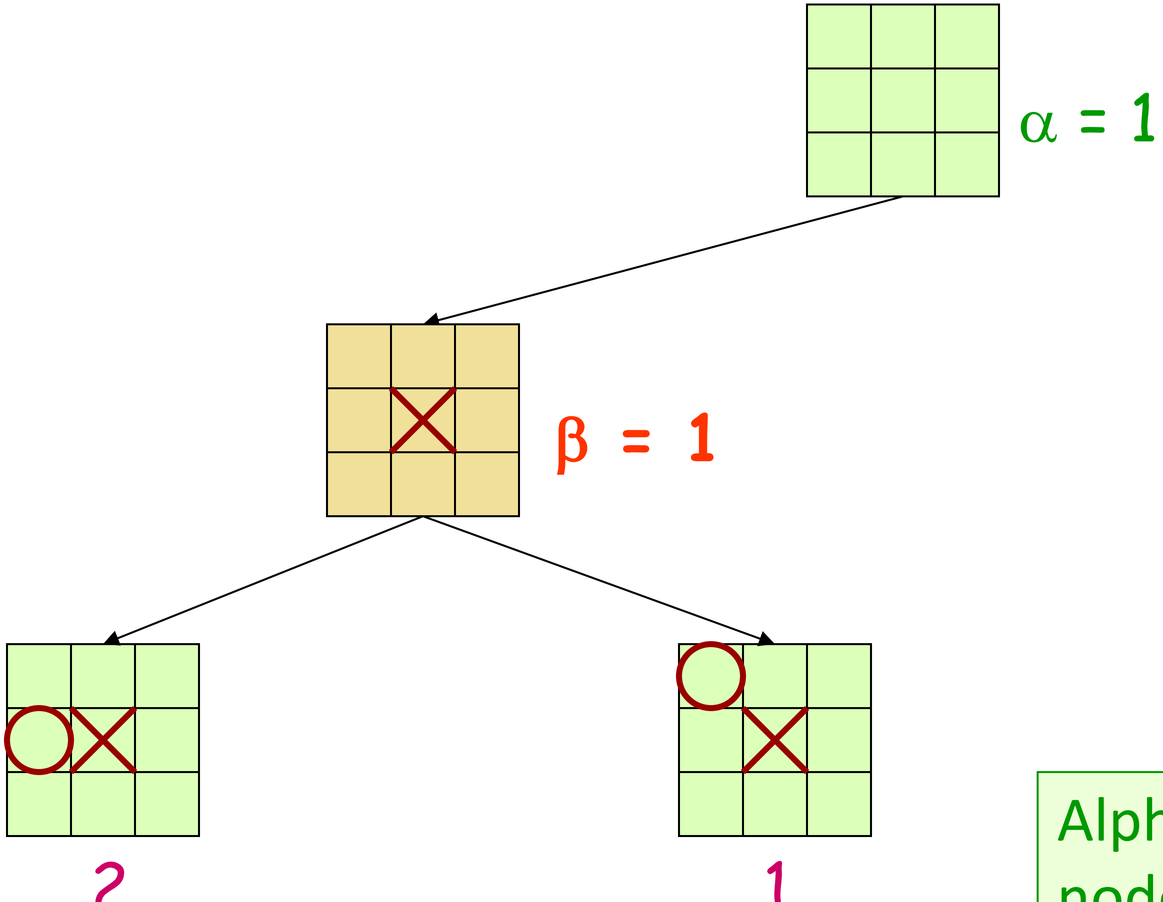
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example



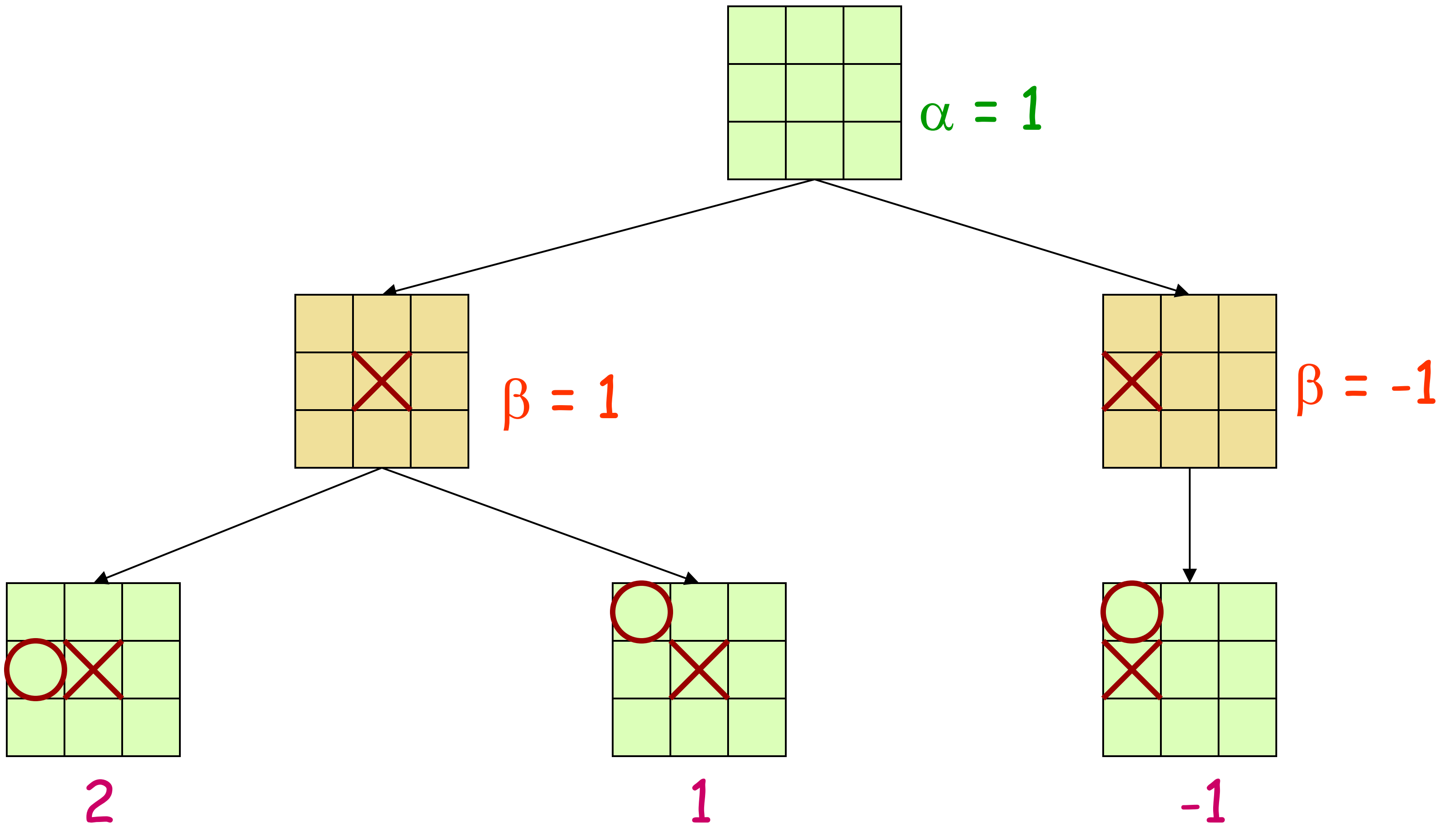
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

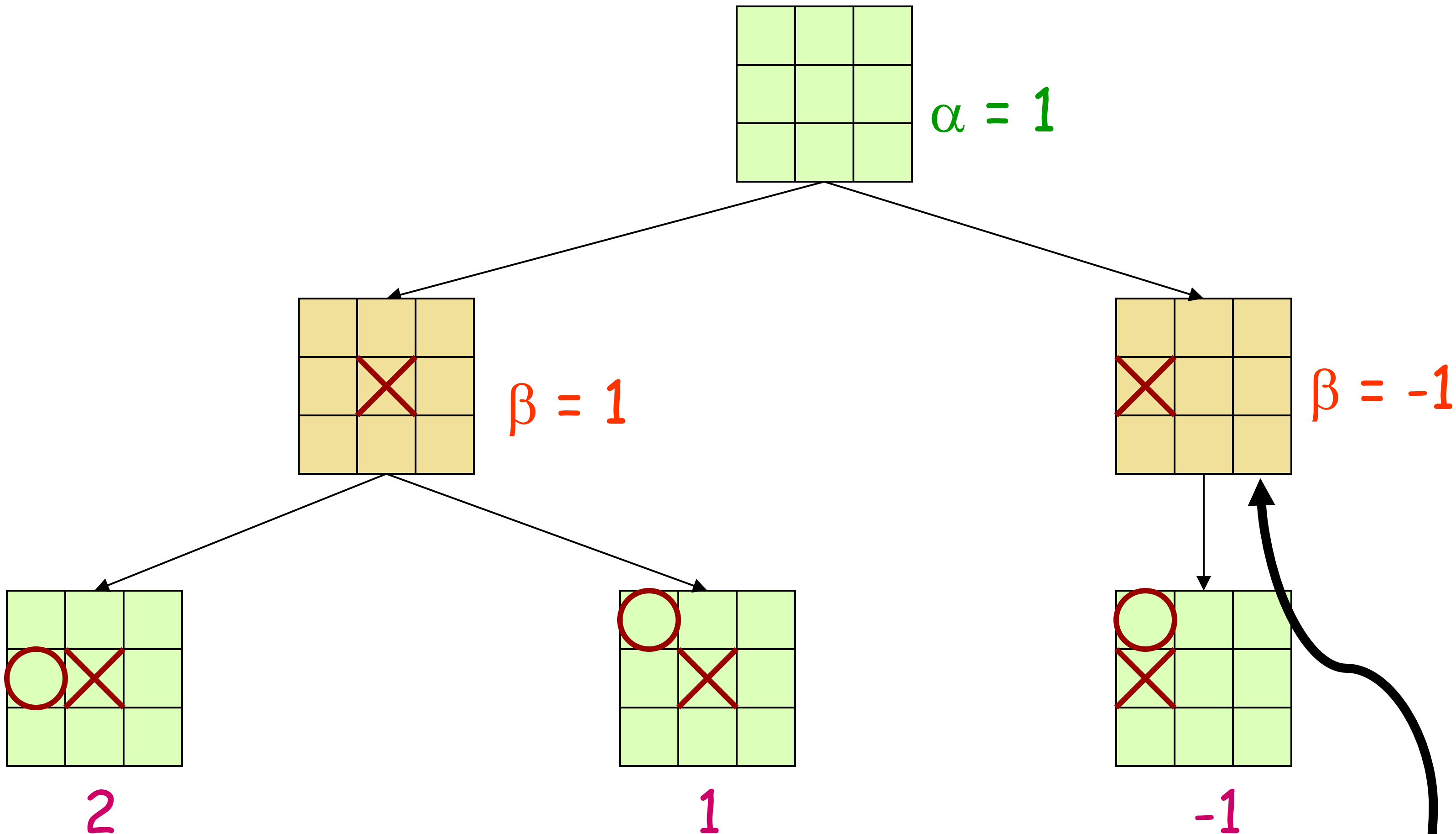


Alpha value of MAX node is **lower** bound on final backed-up value; it can never decrease

Alpha-Beta Tic-Tac-Toe Example

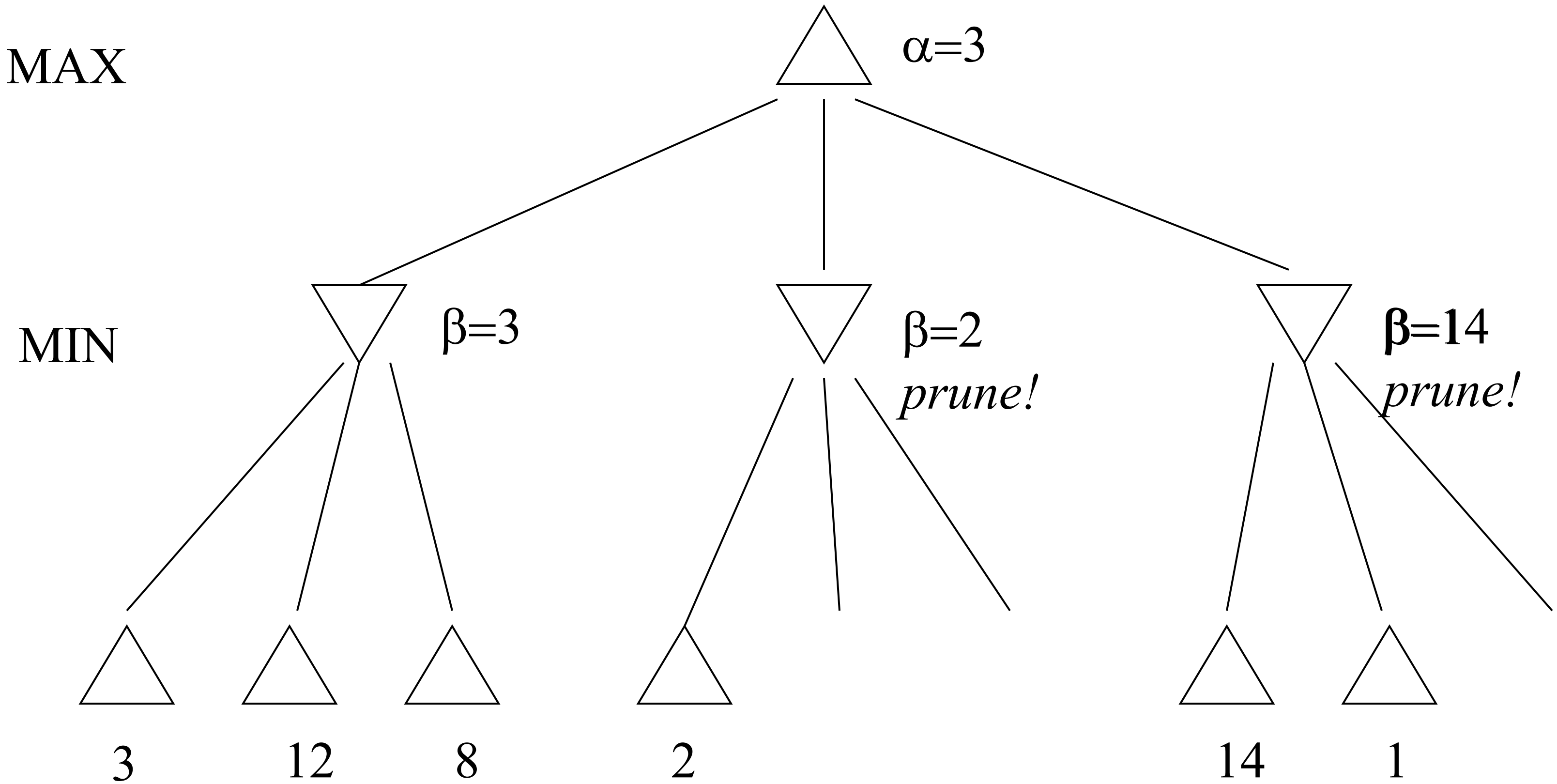


Alpha-Beta Tic-Tac-Toe Example

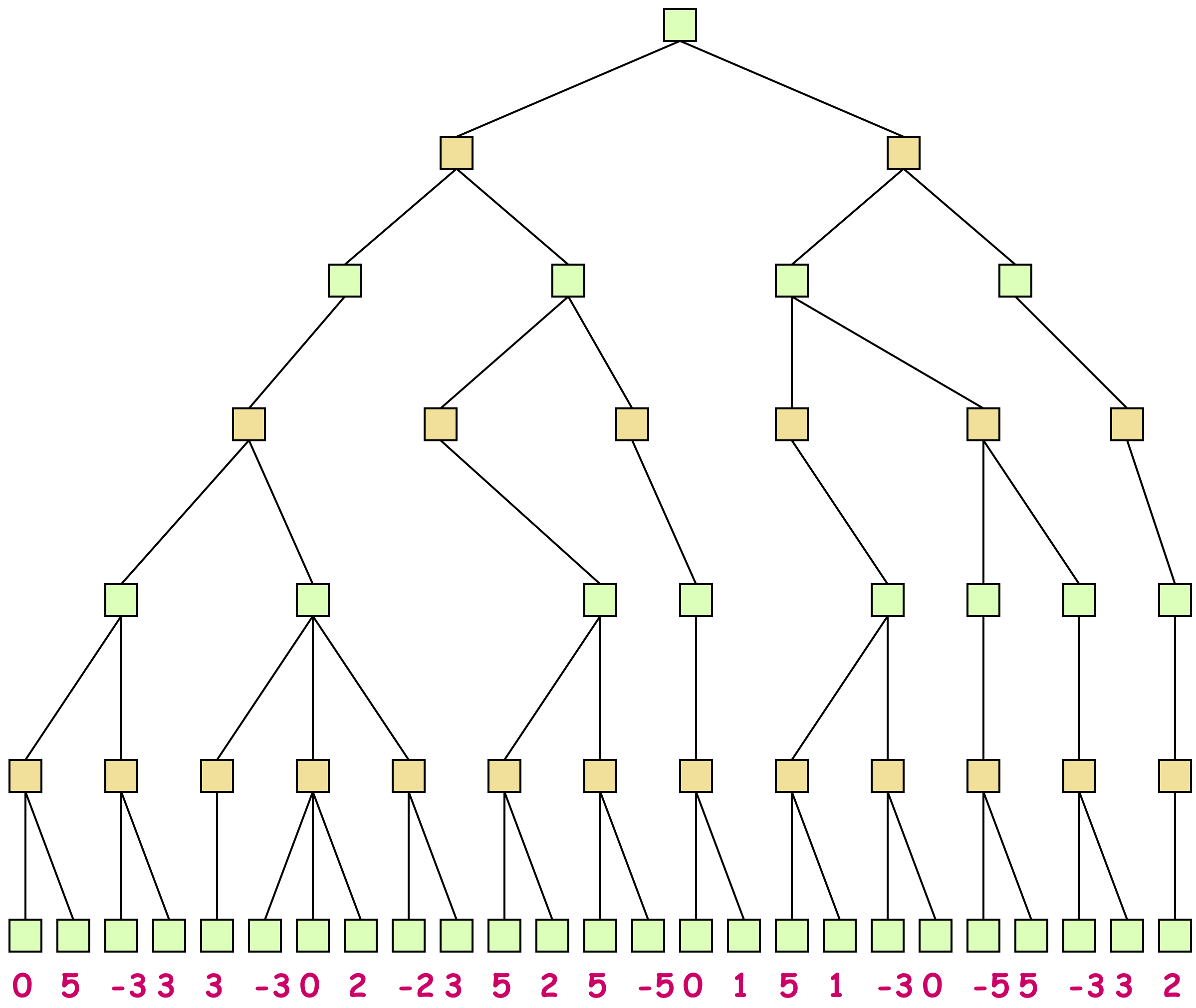


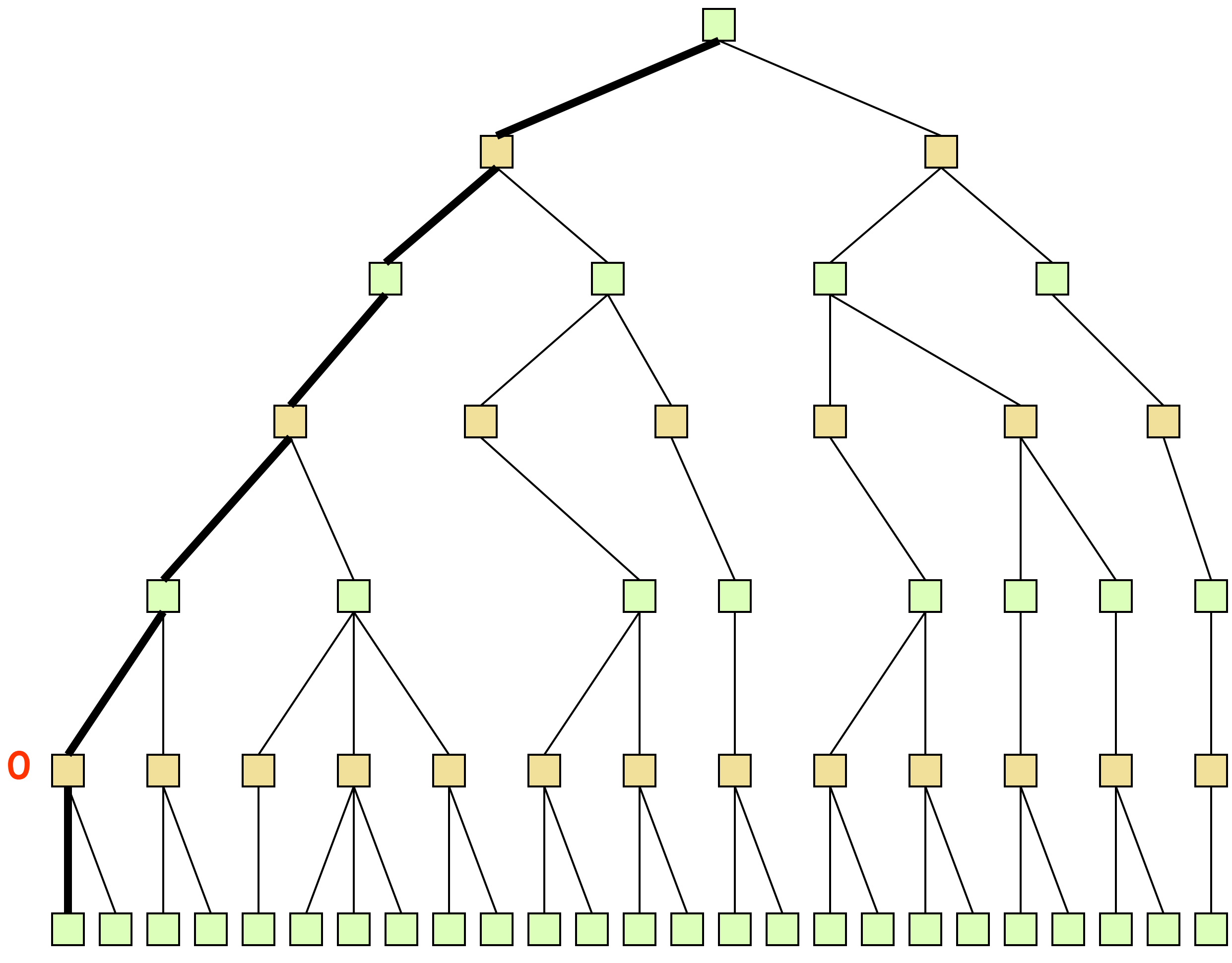
Discontinue search below a MIN node whose beta value ≤ alpha value of one of its MAX ancestors

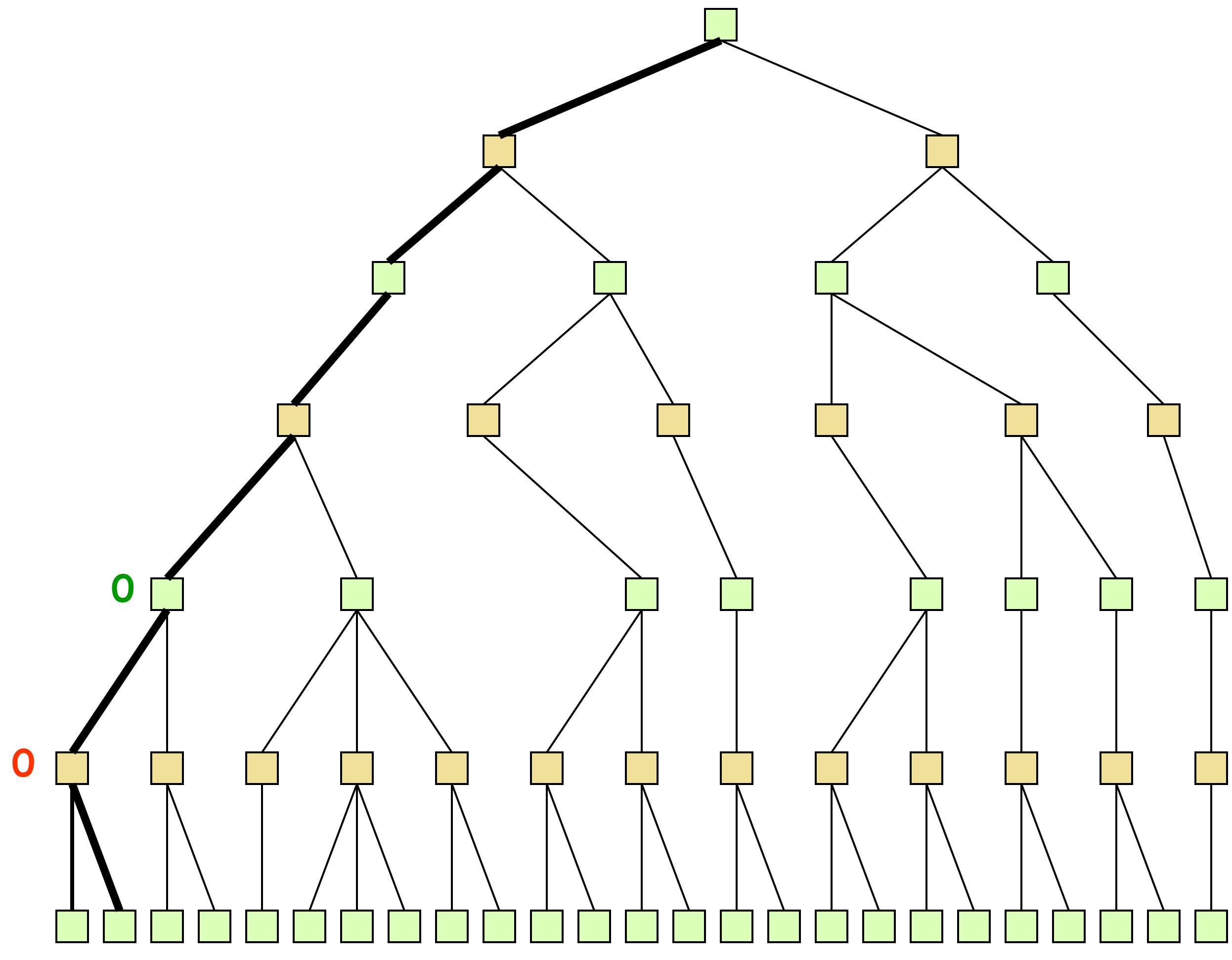
Another alpha-beta example



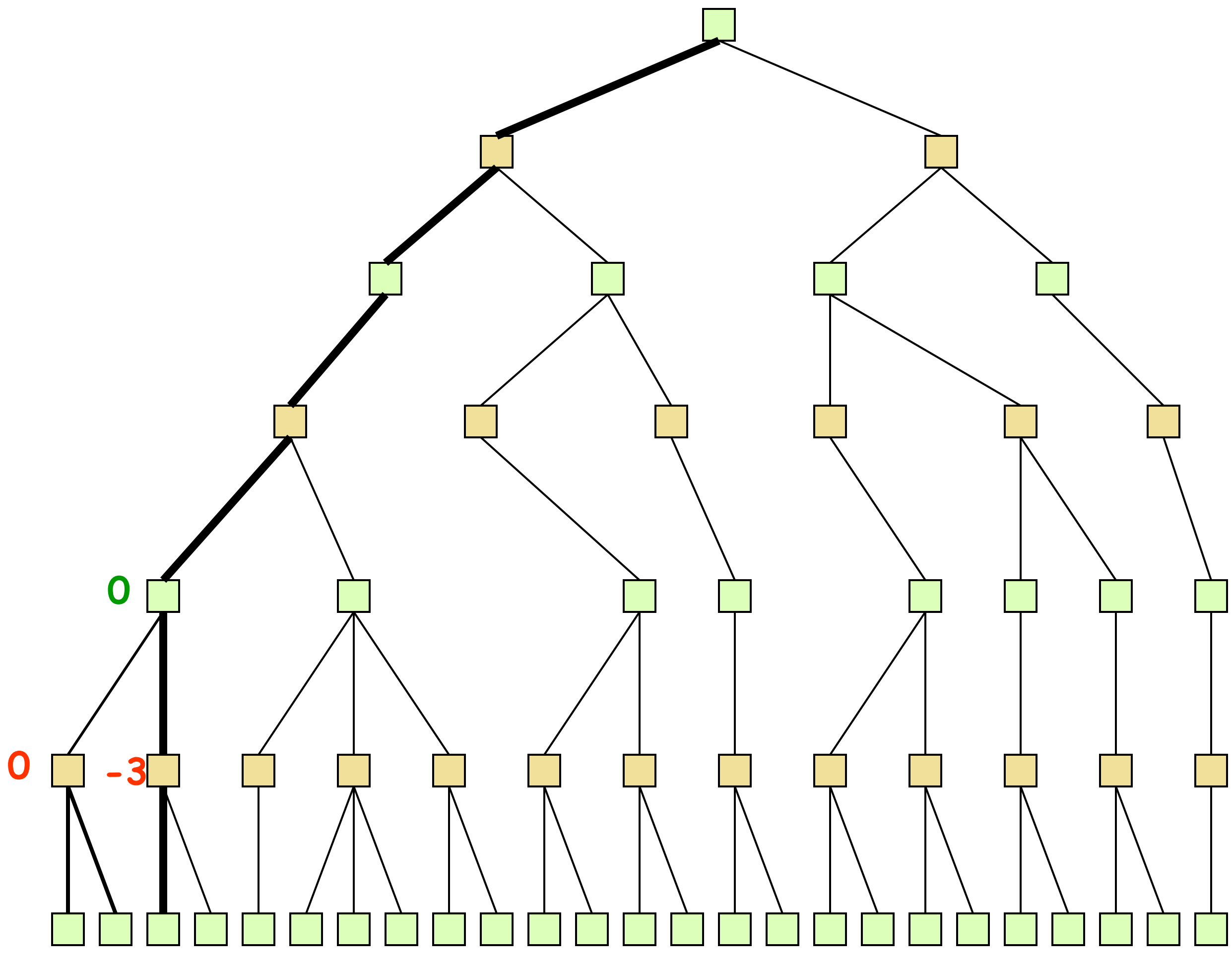
Alpha-Beta Tic-Tac-Toe Example 2



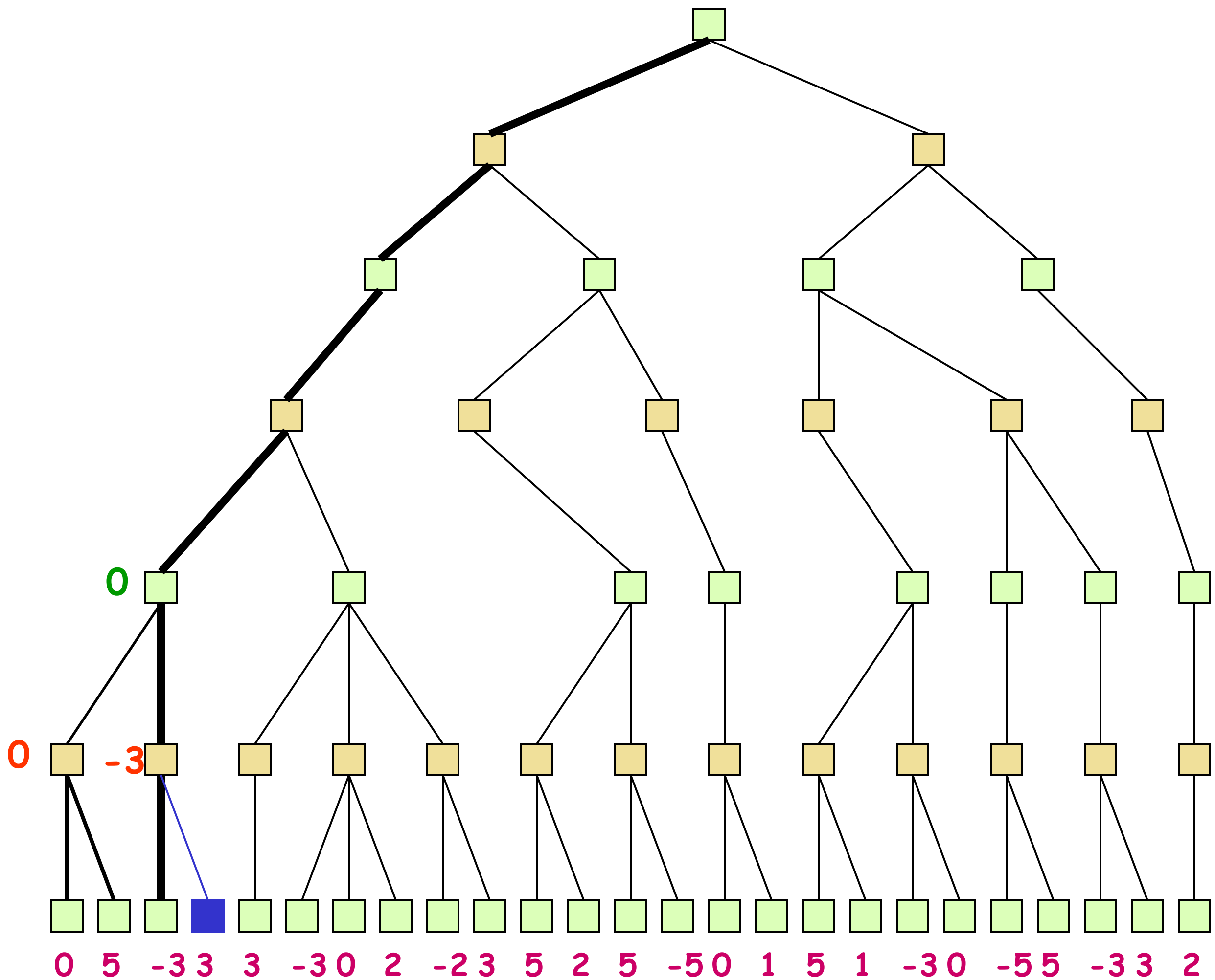

0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



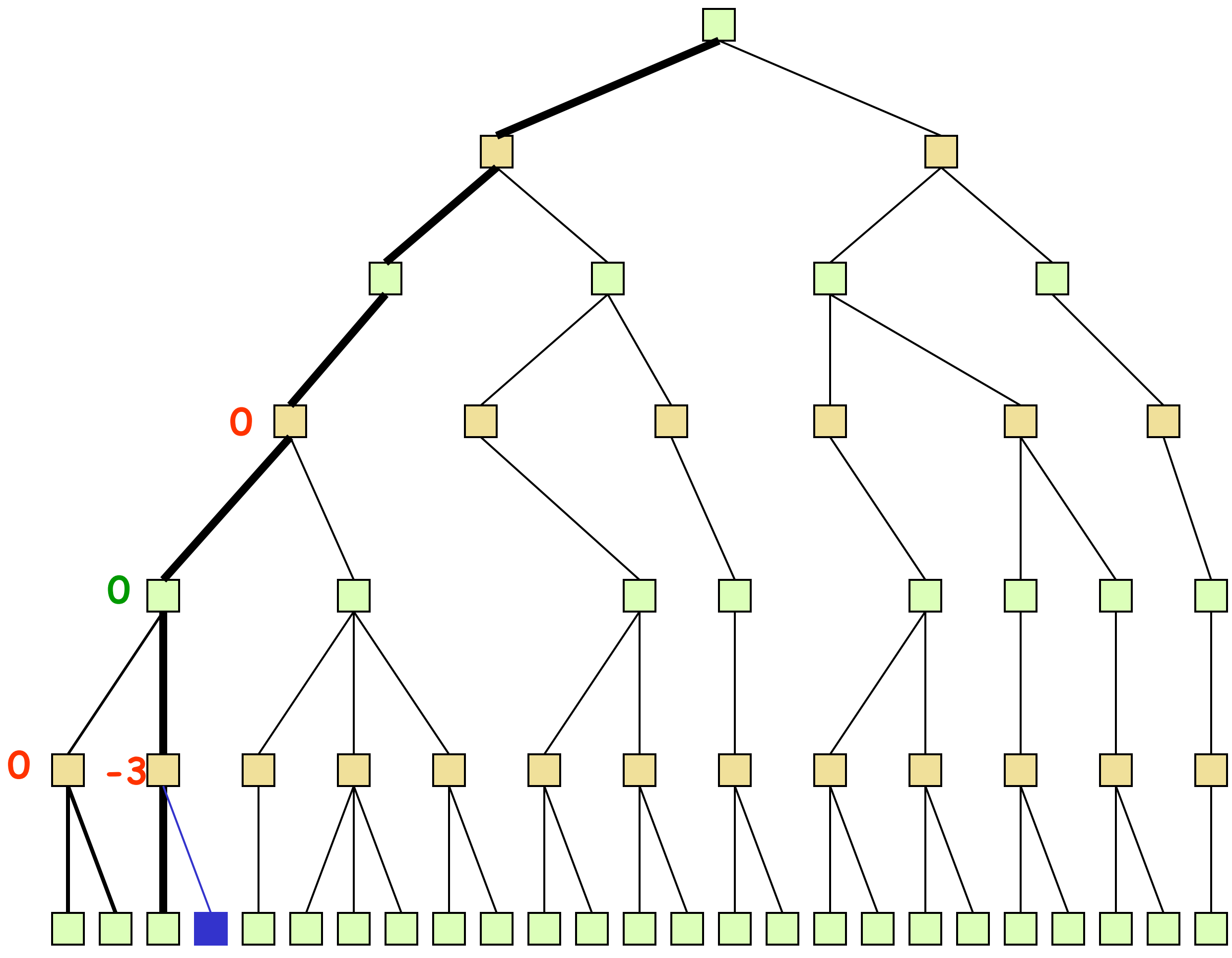
0
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



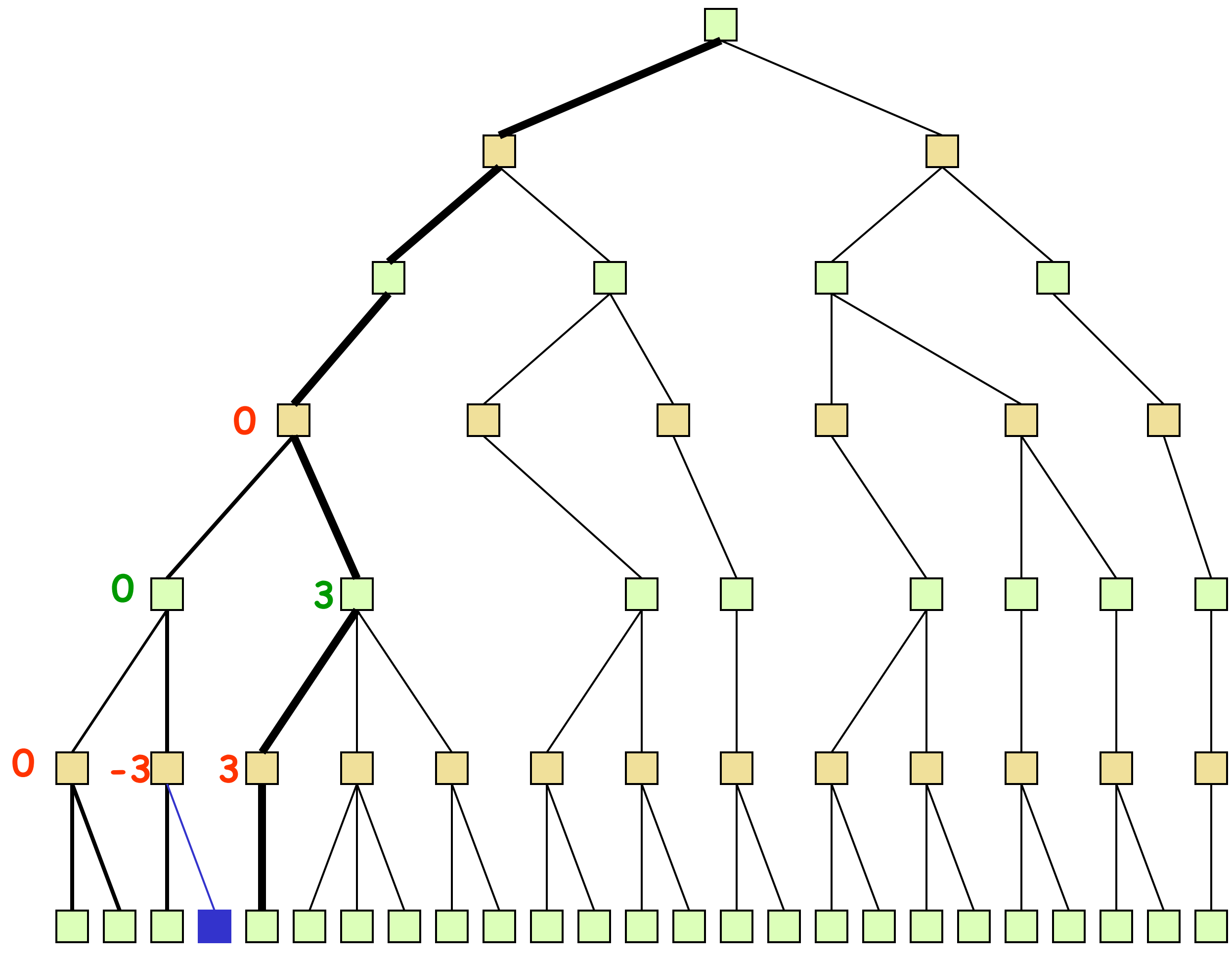
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



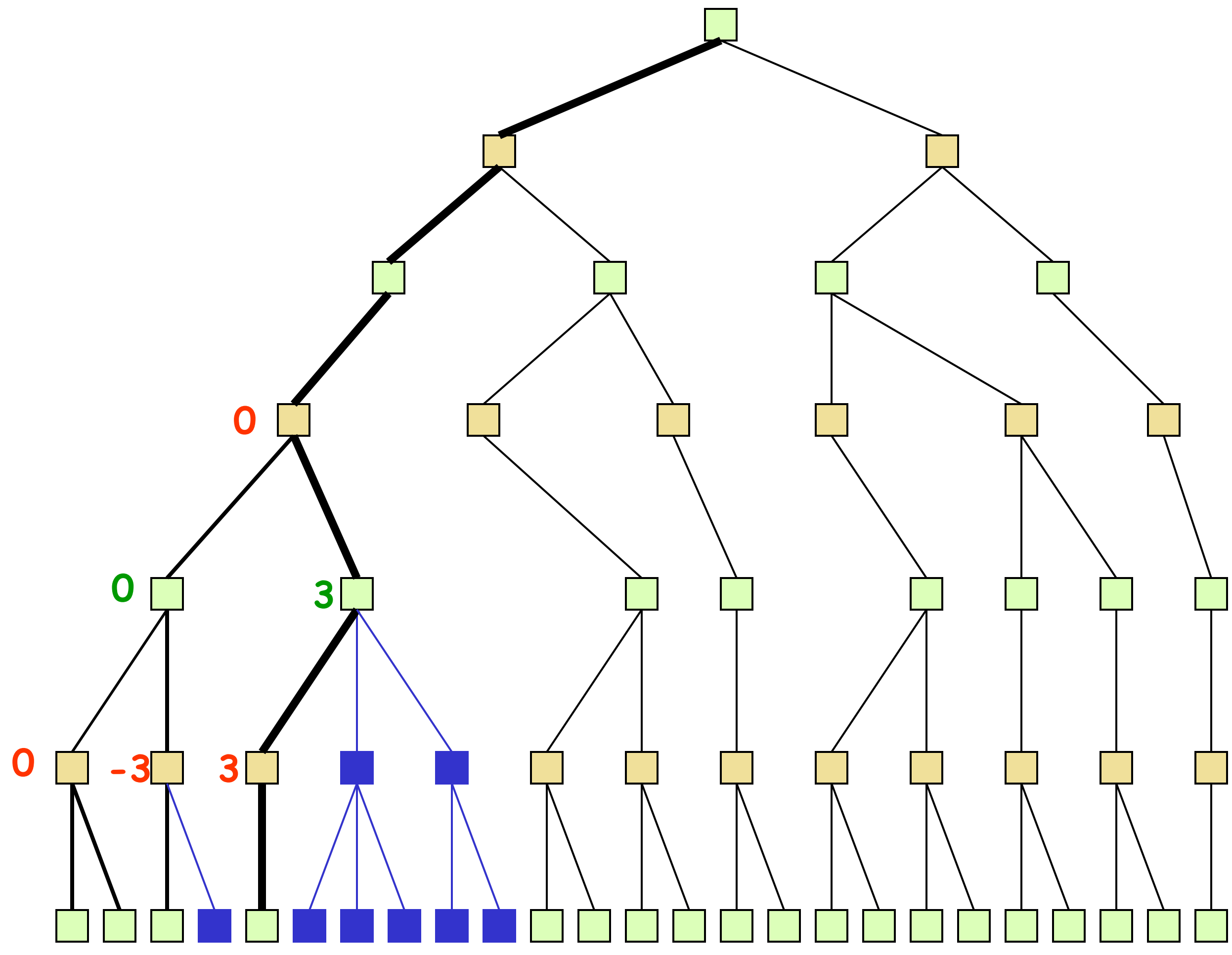
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

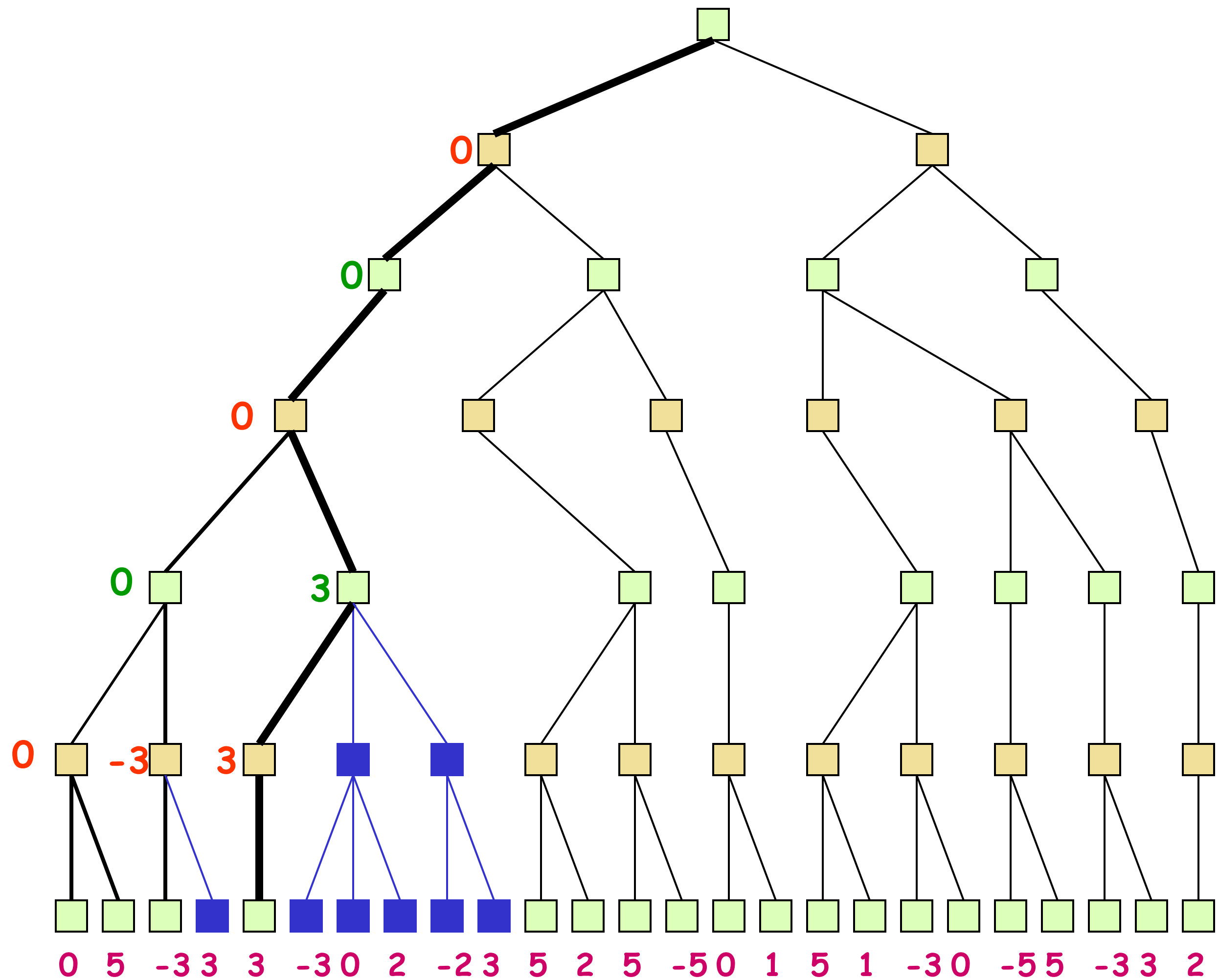


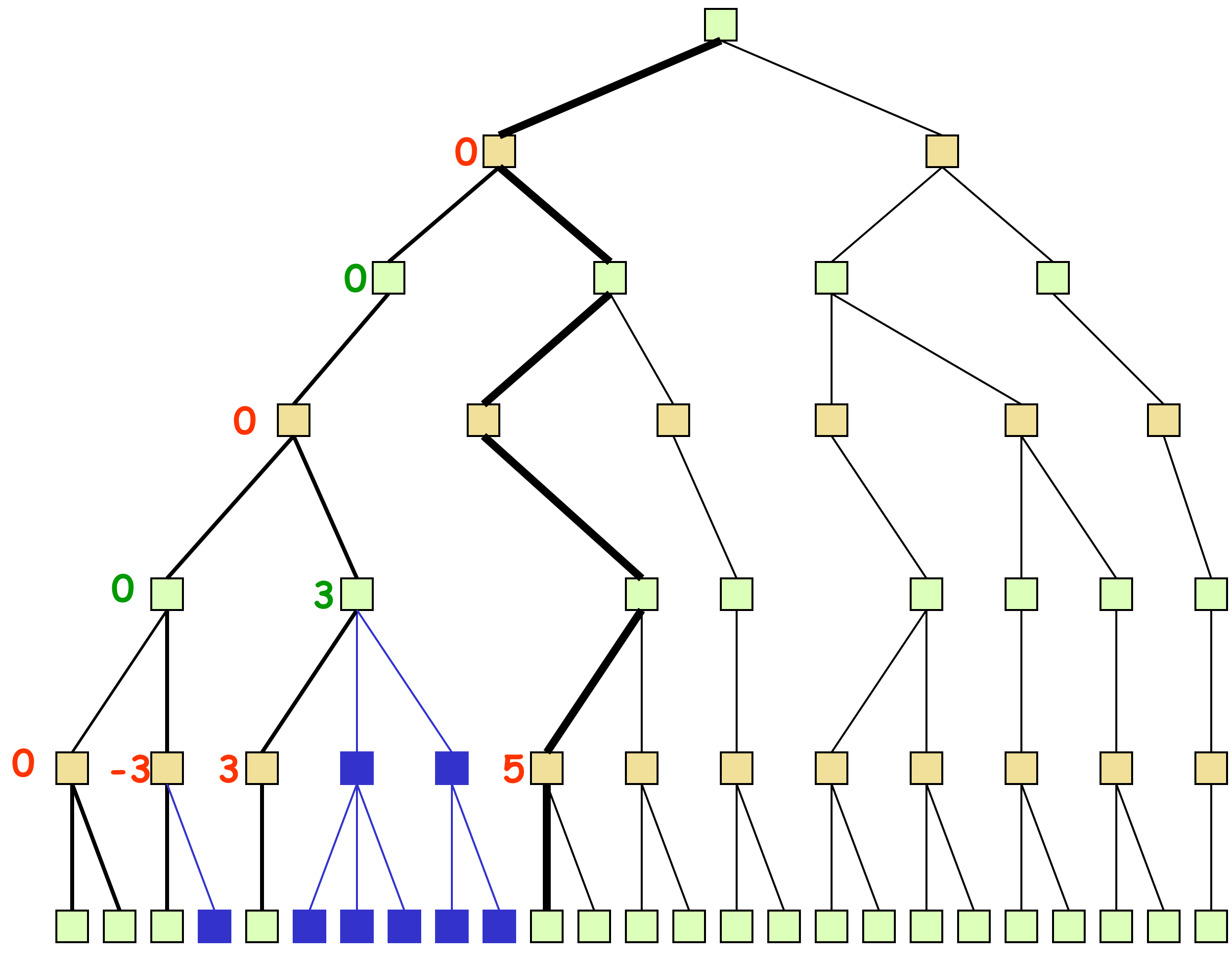
0
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

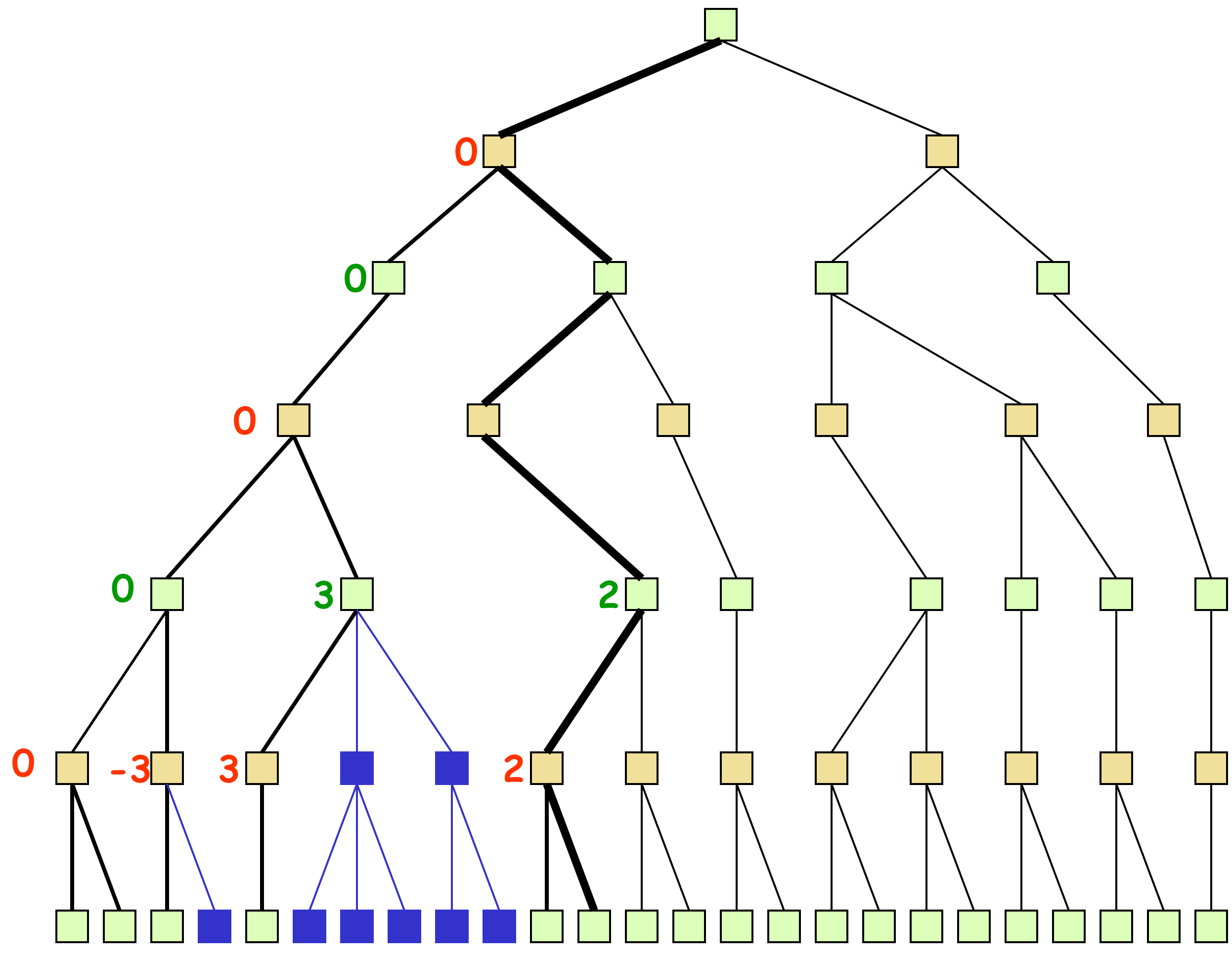


0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

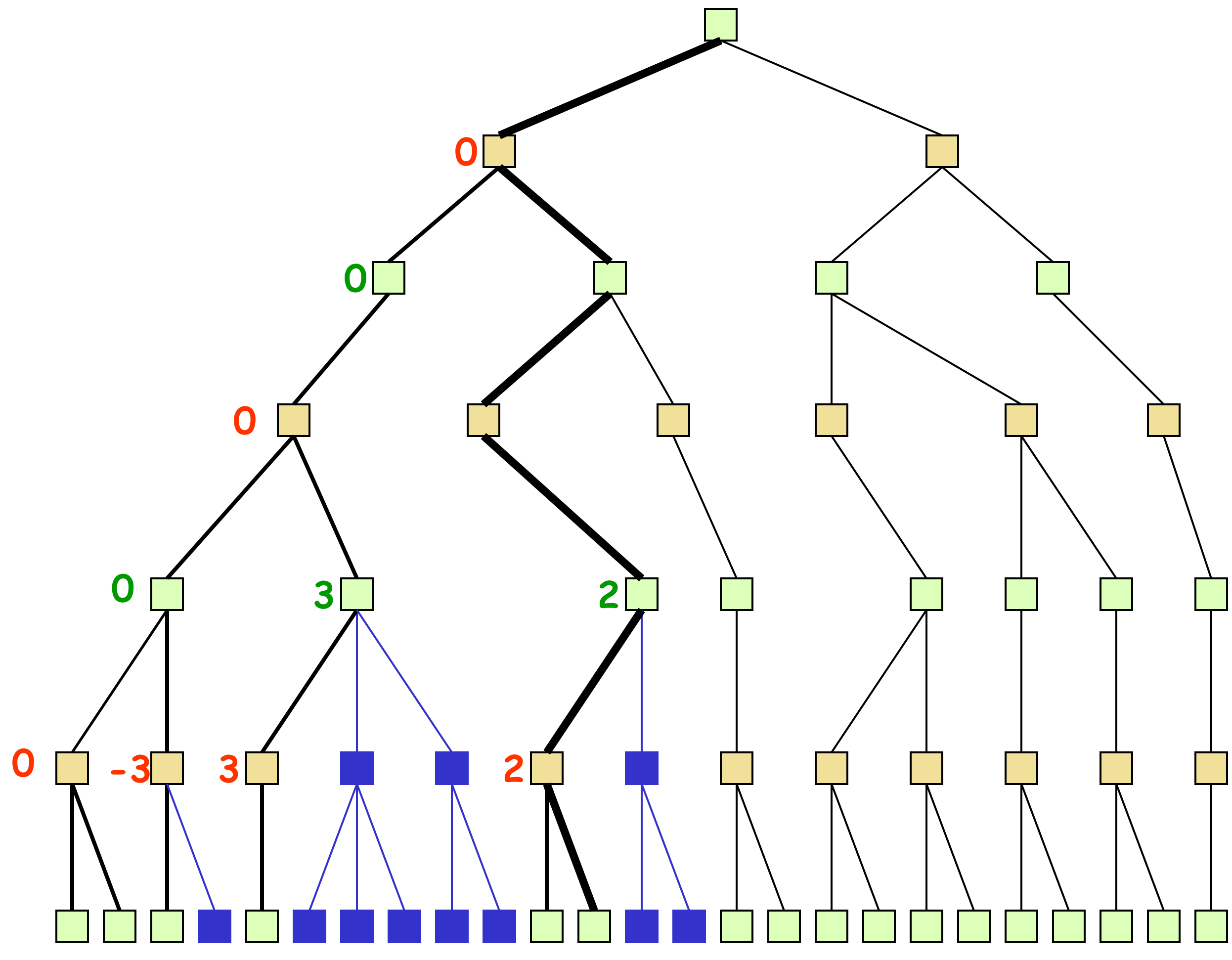

0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

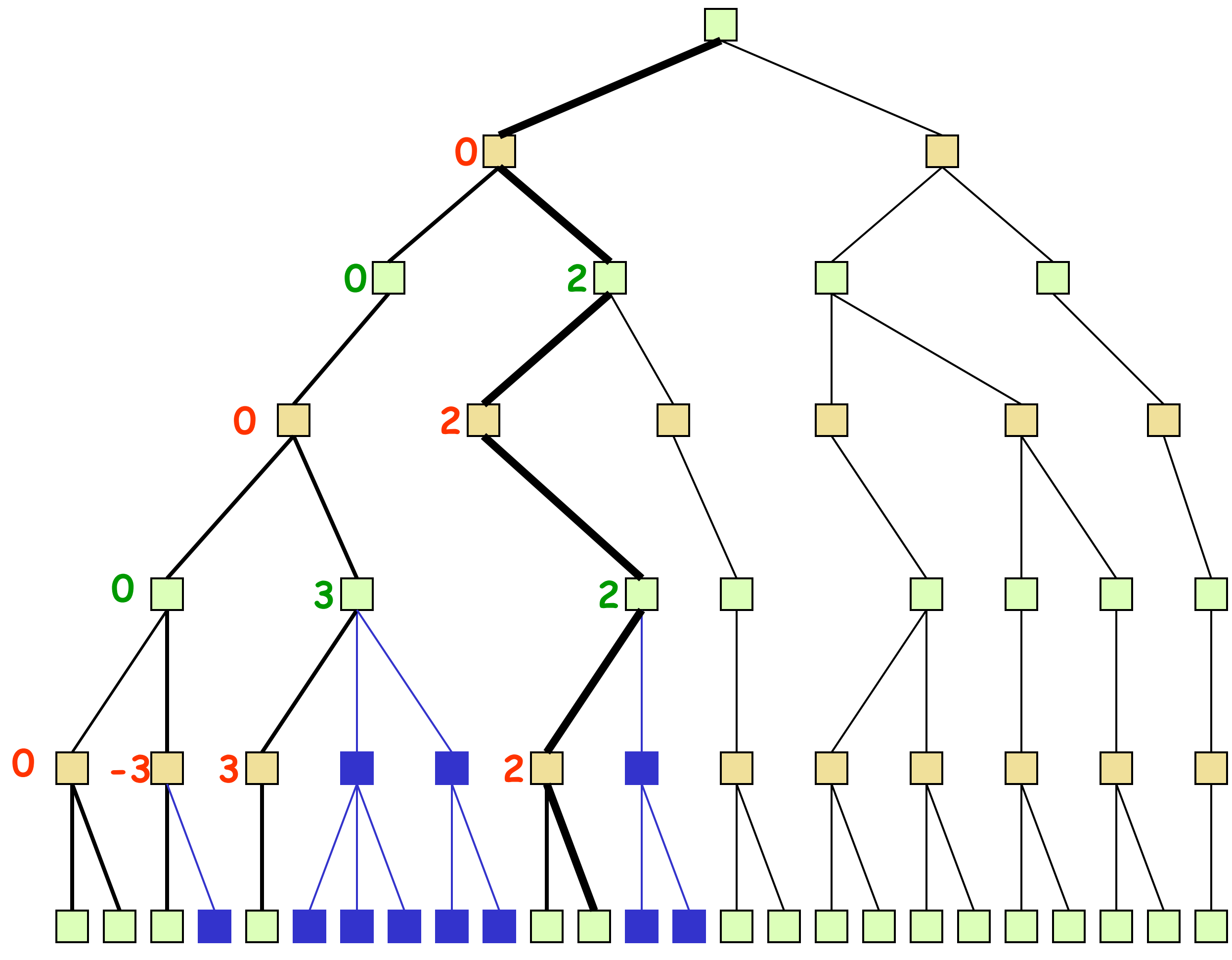

0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



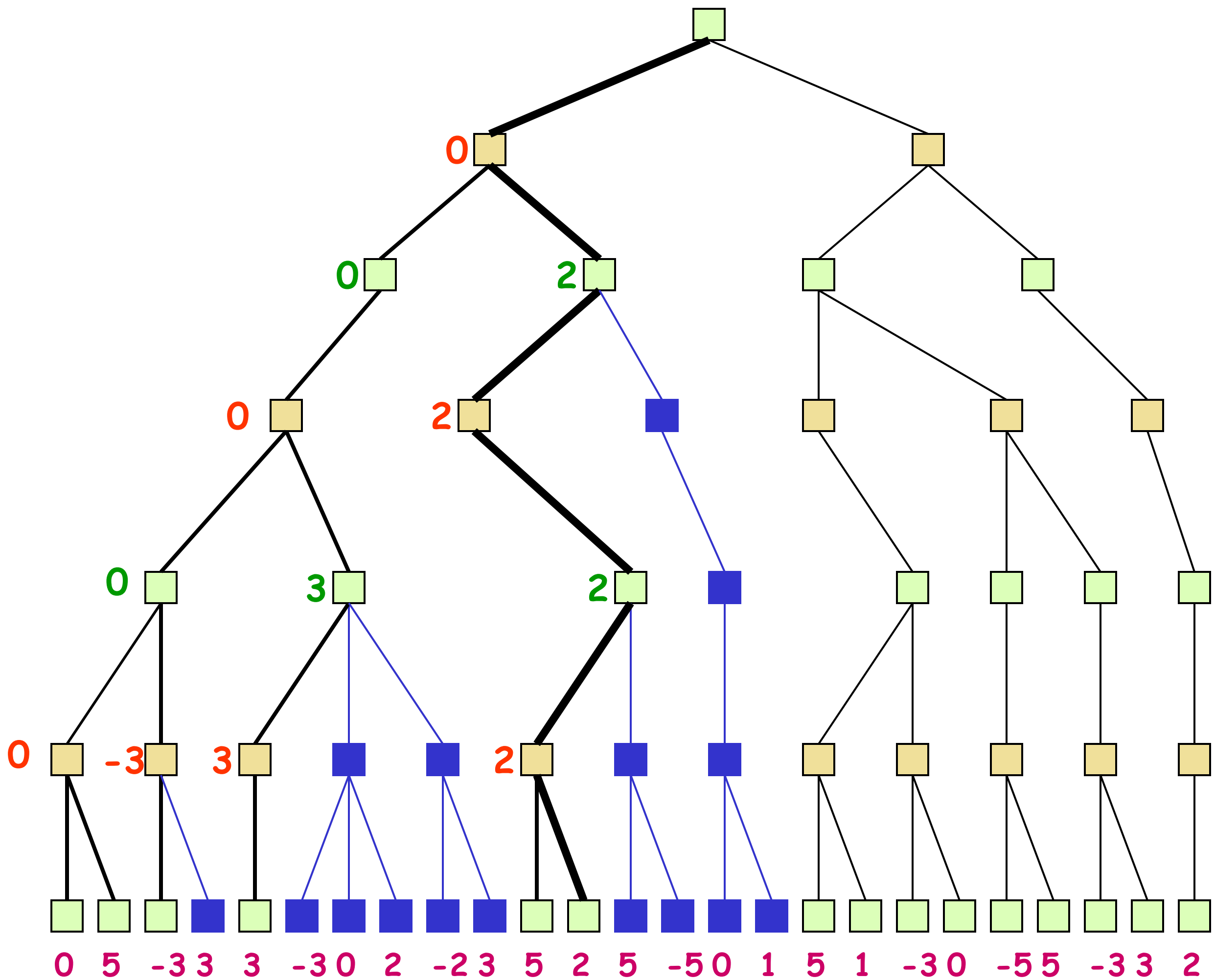
0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



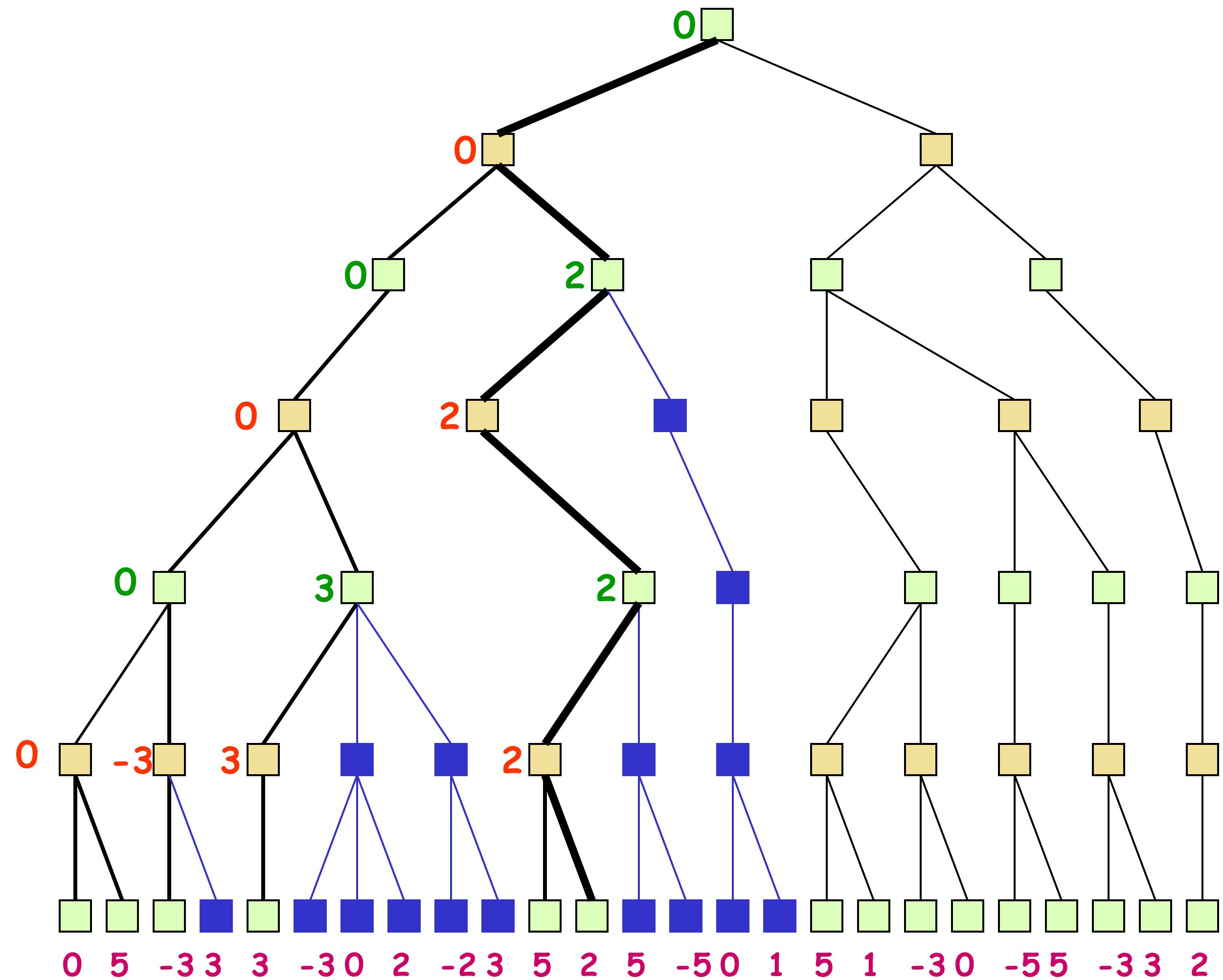
0
0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

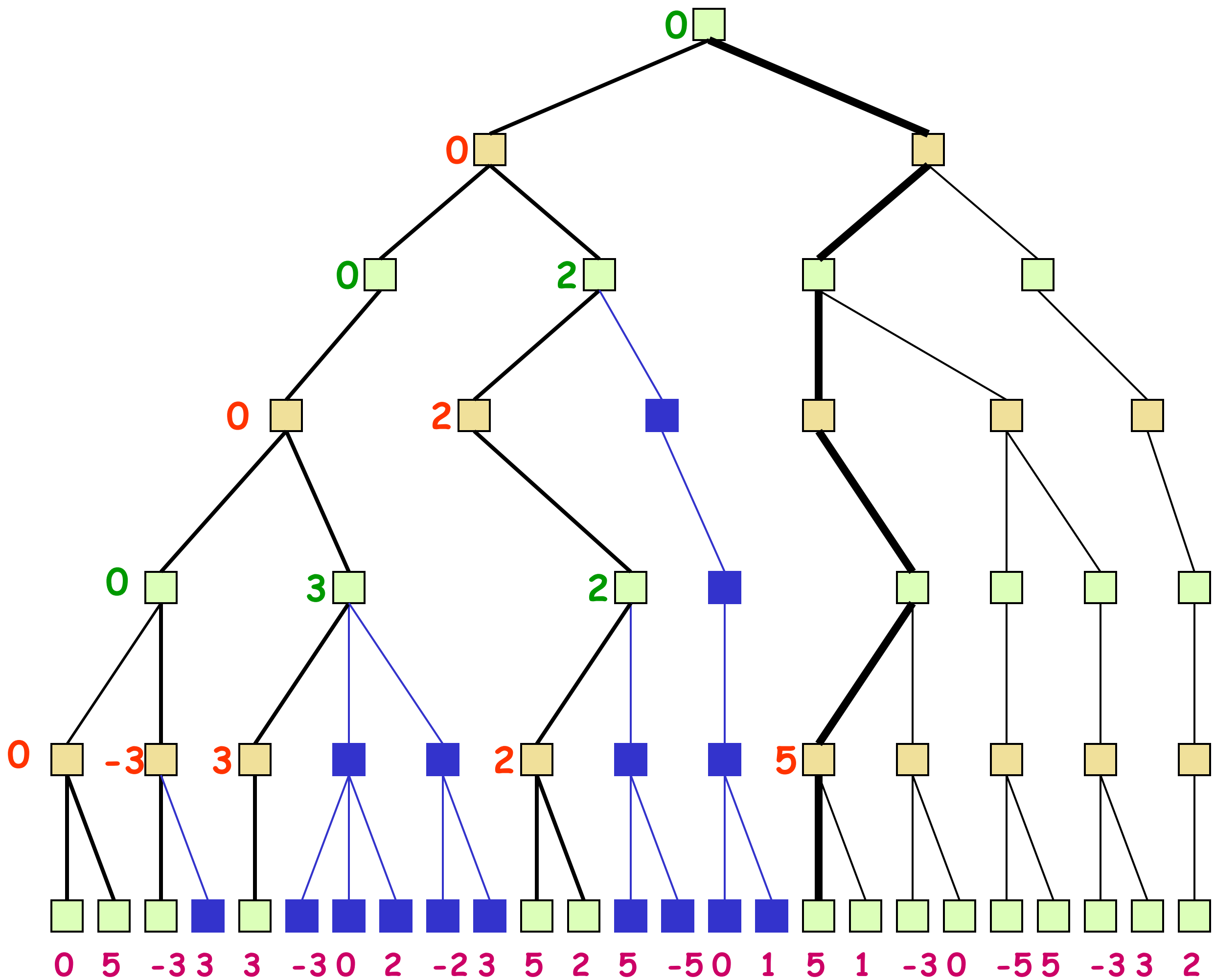


0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

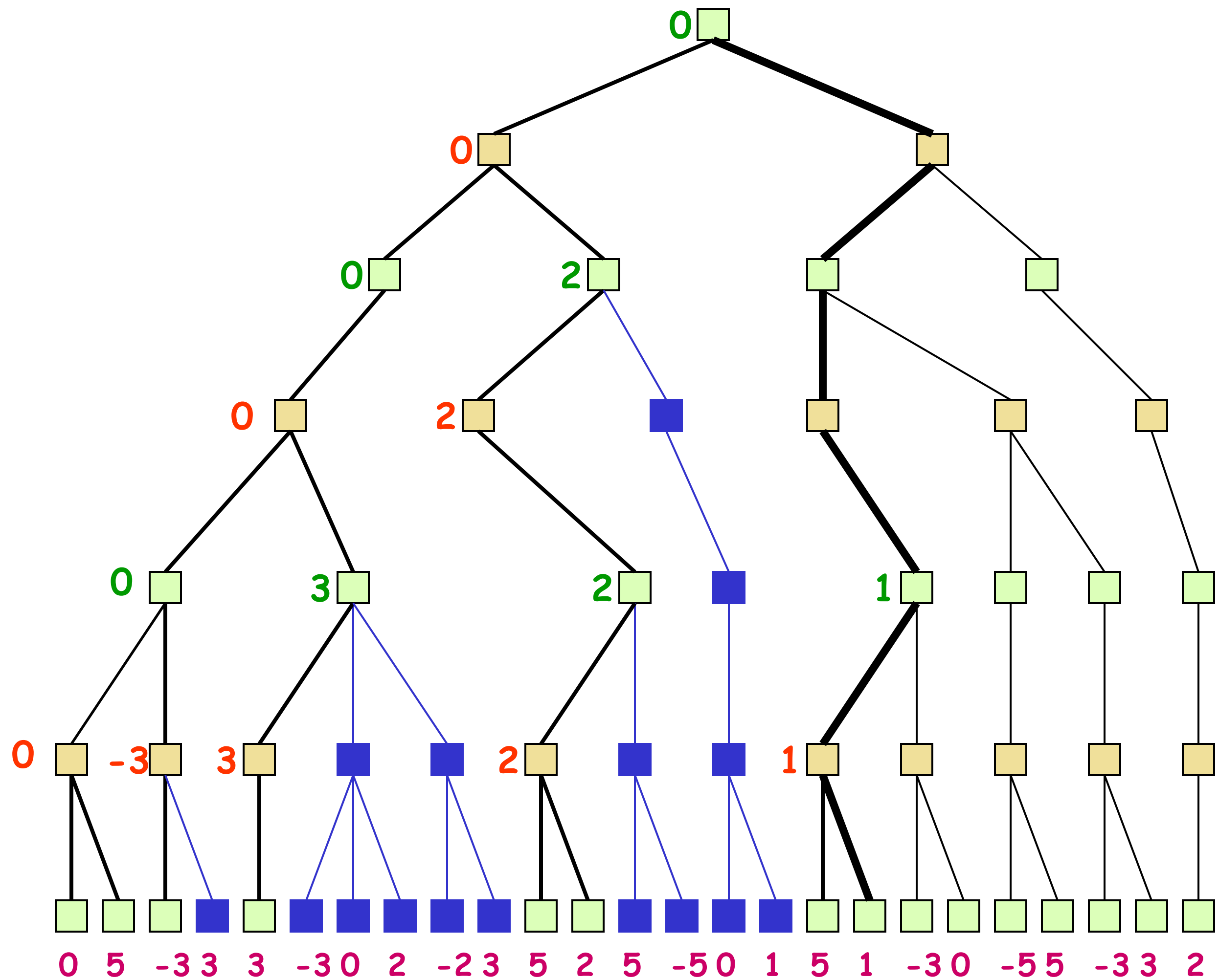


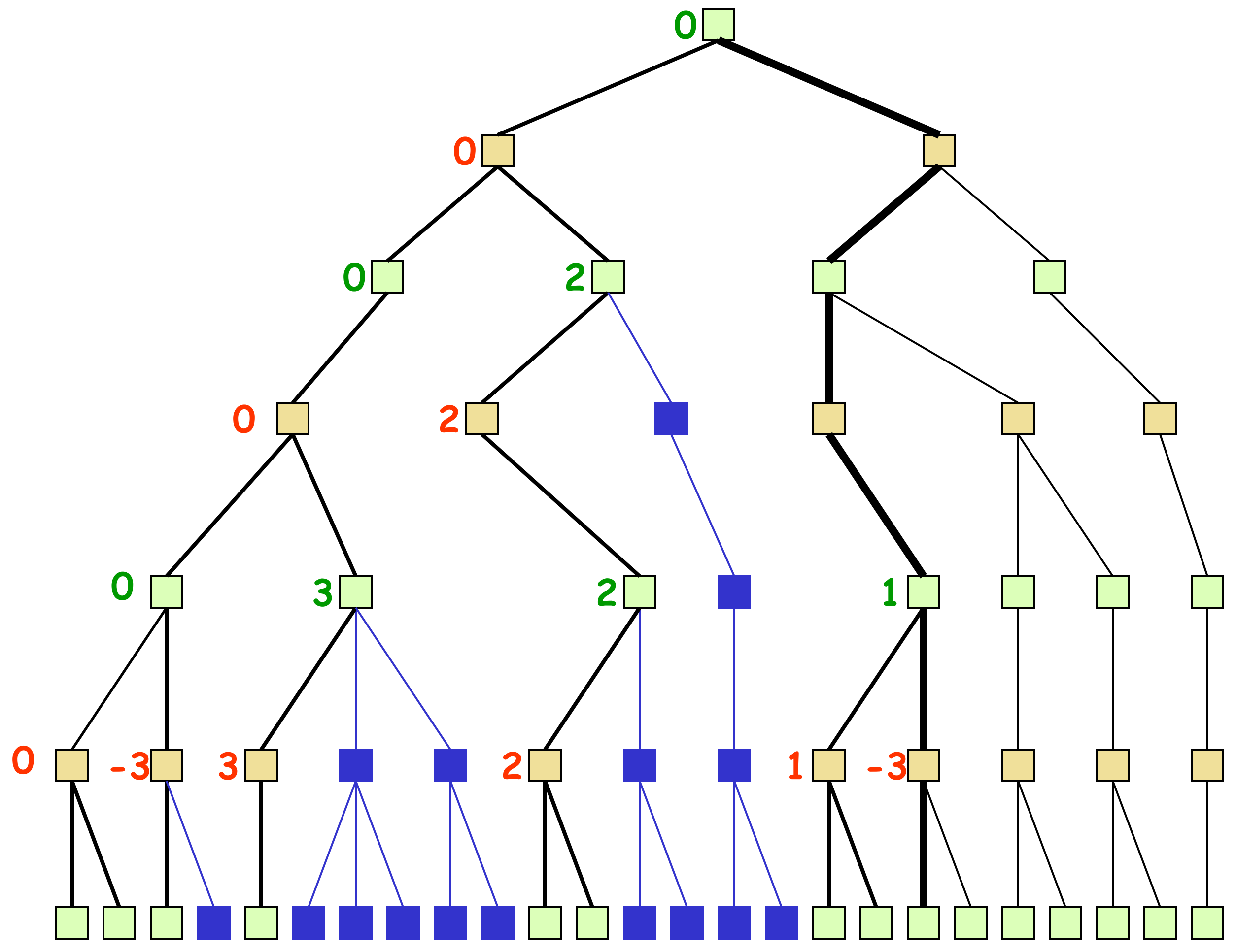
0
0 2
0 2
0 3 2
0 -3 3 2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

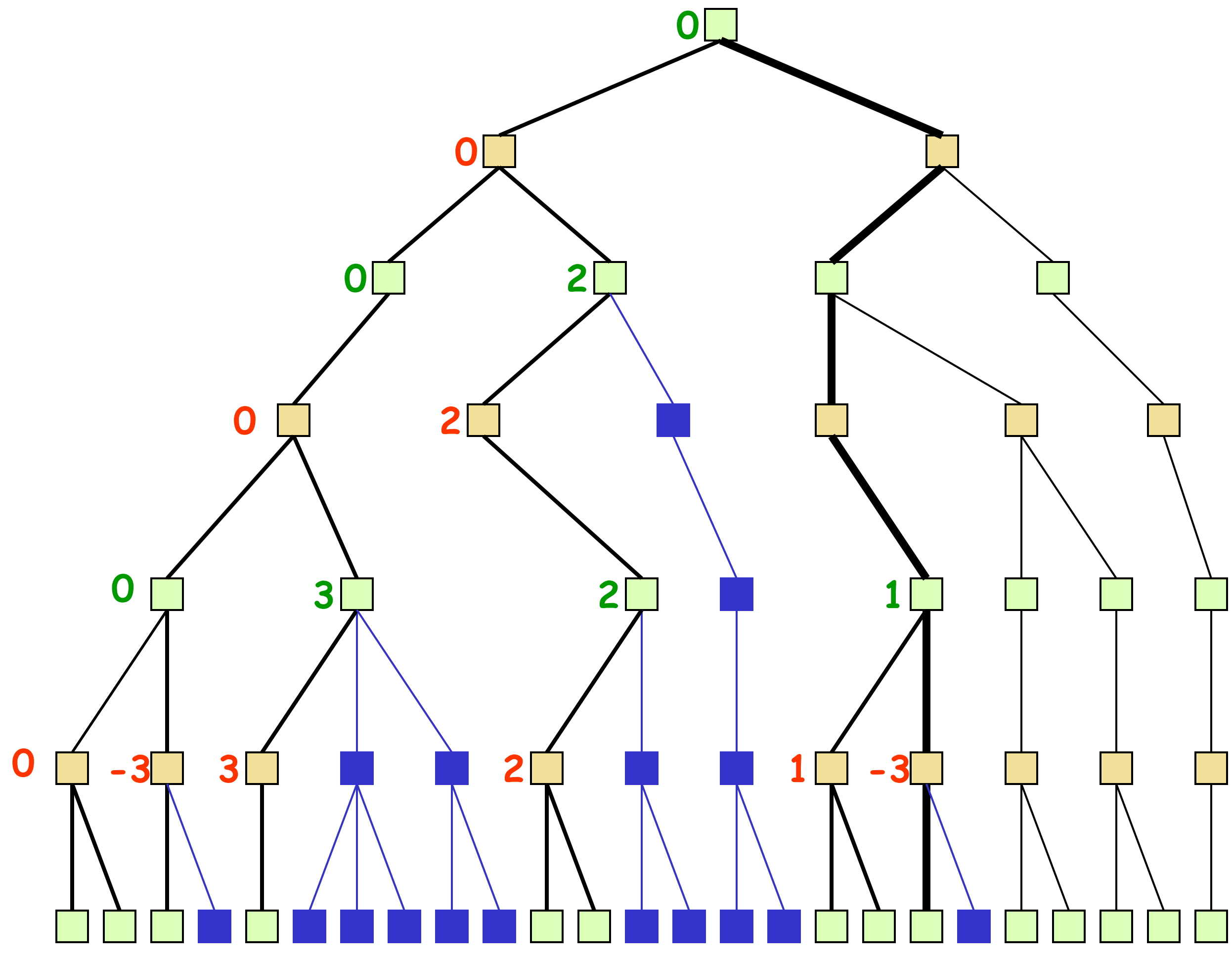

0
0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

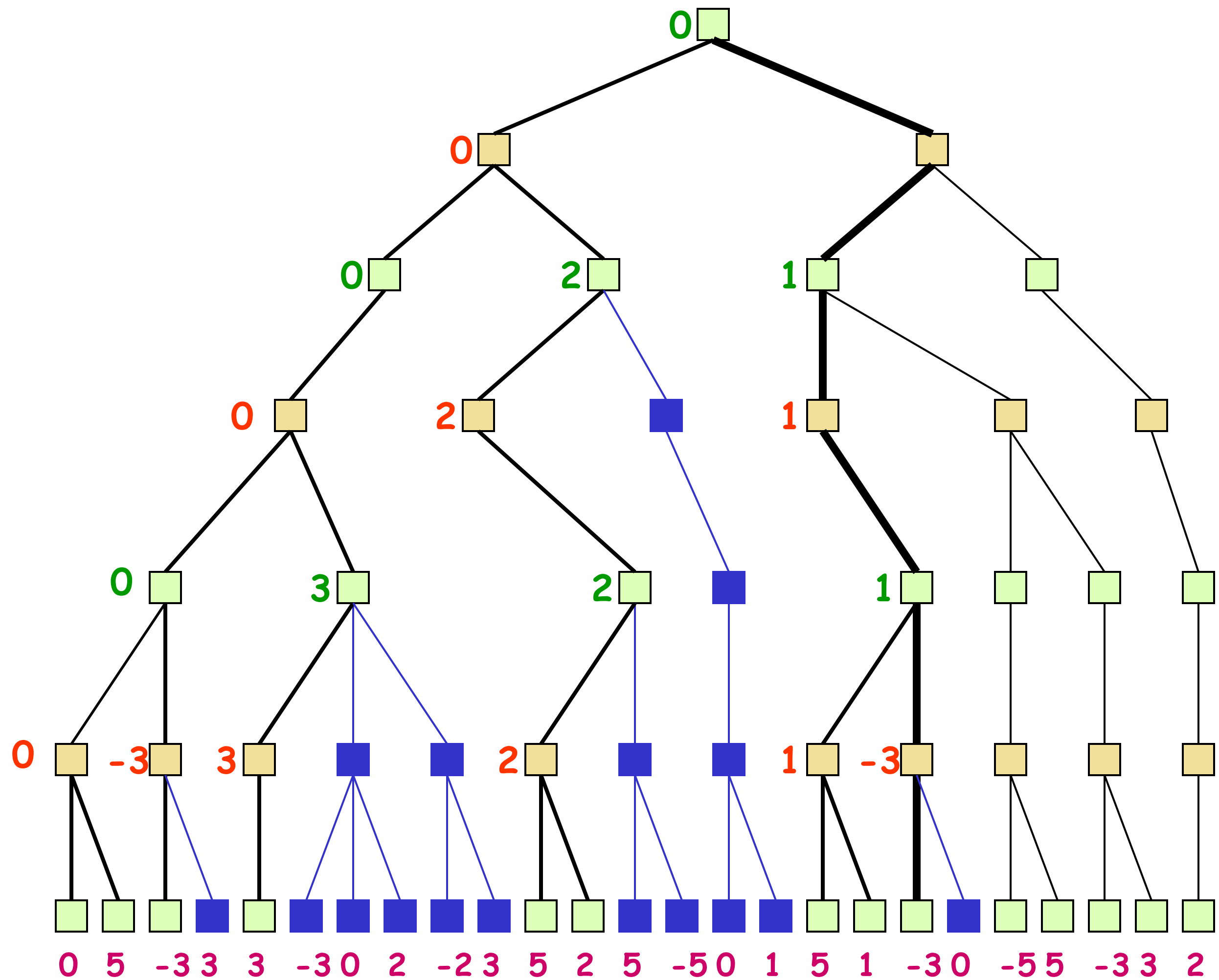


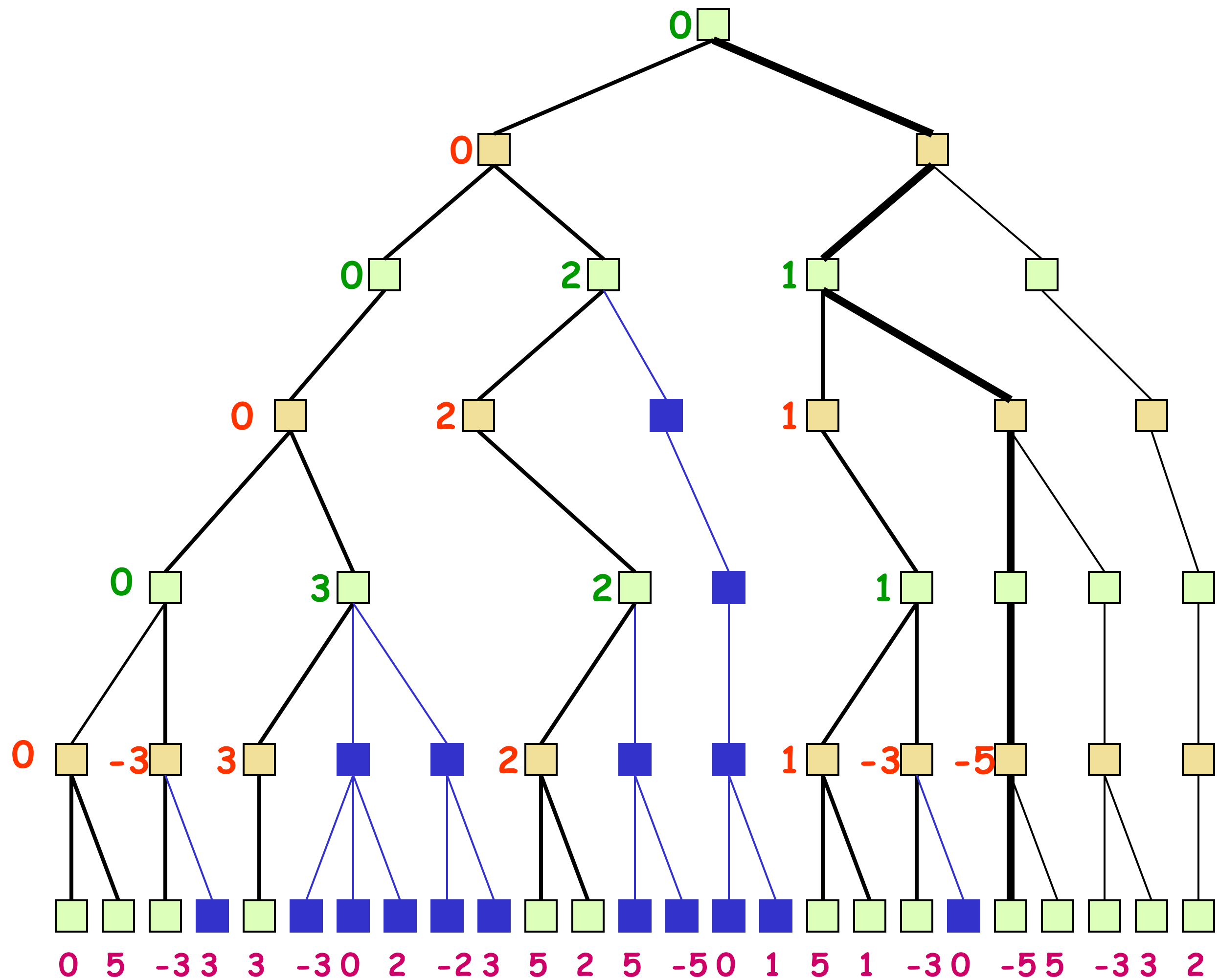
0
0
0 2
0 2
0 3 2
0 -3 3 2 5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

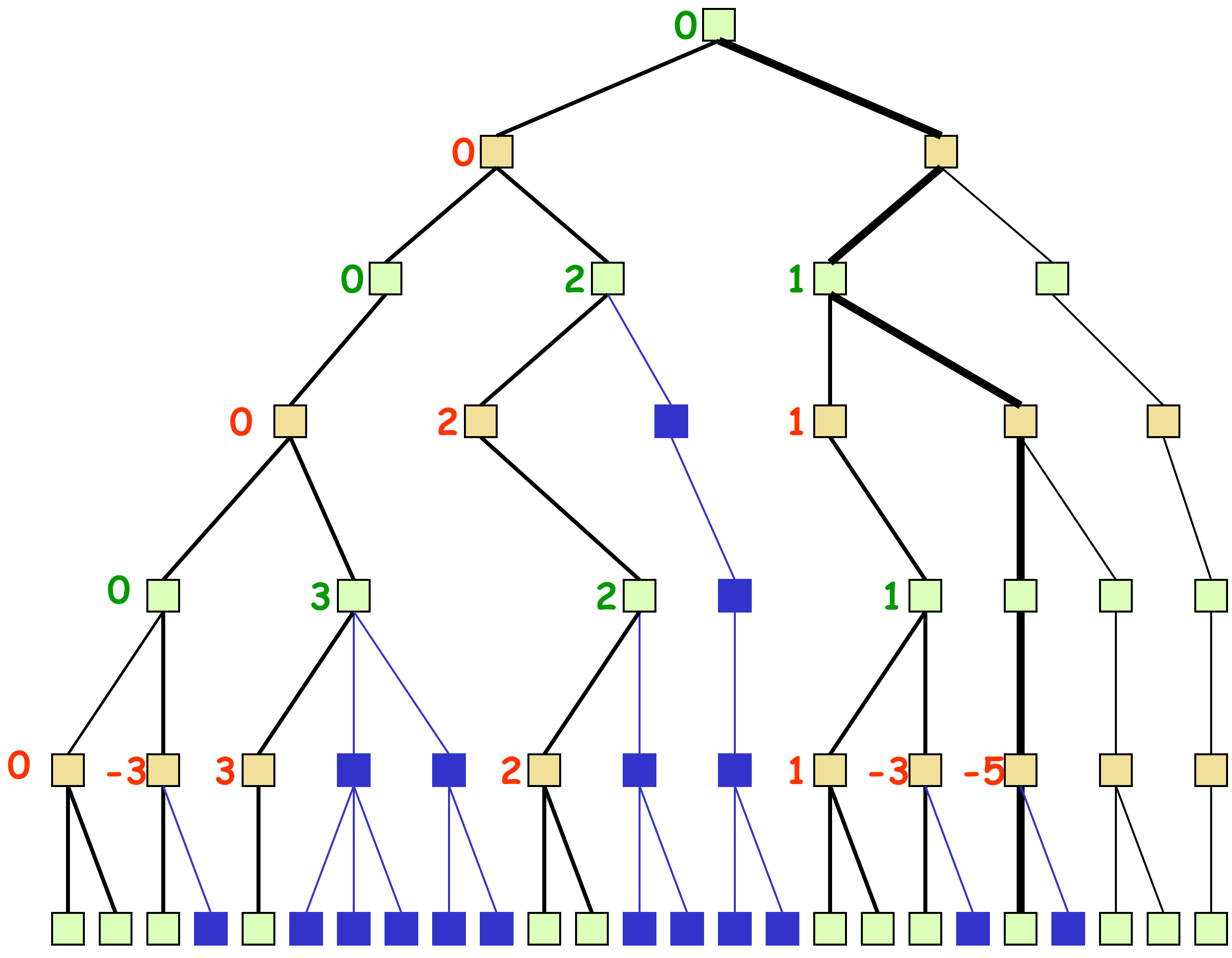

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

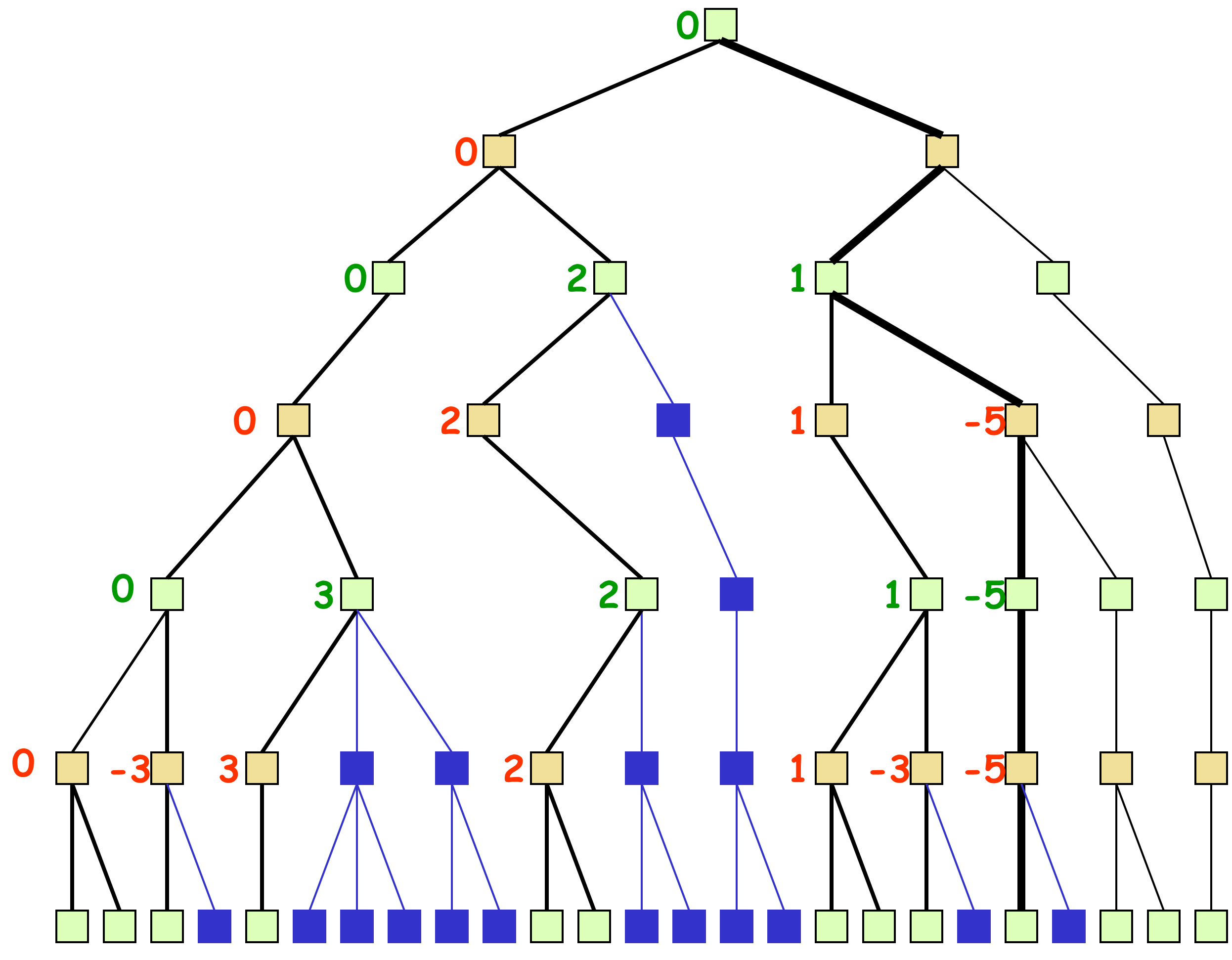

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



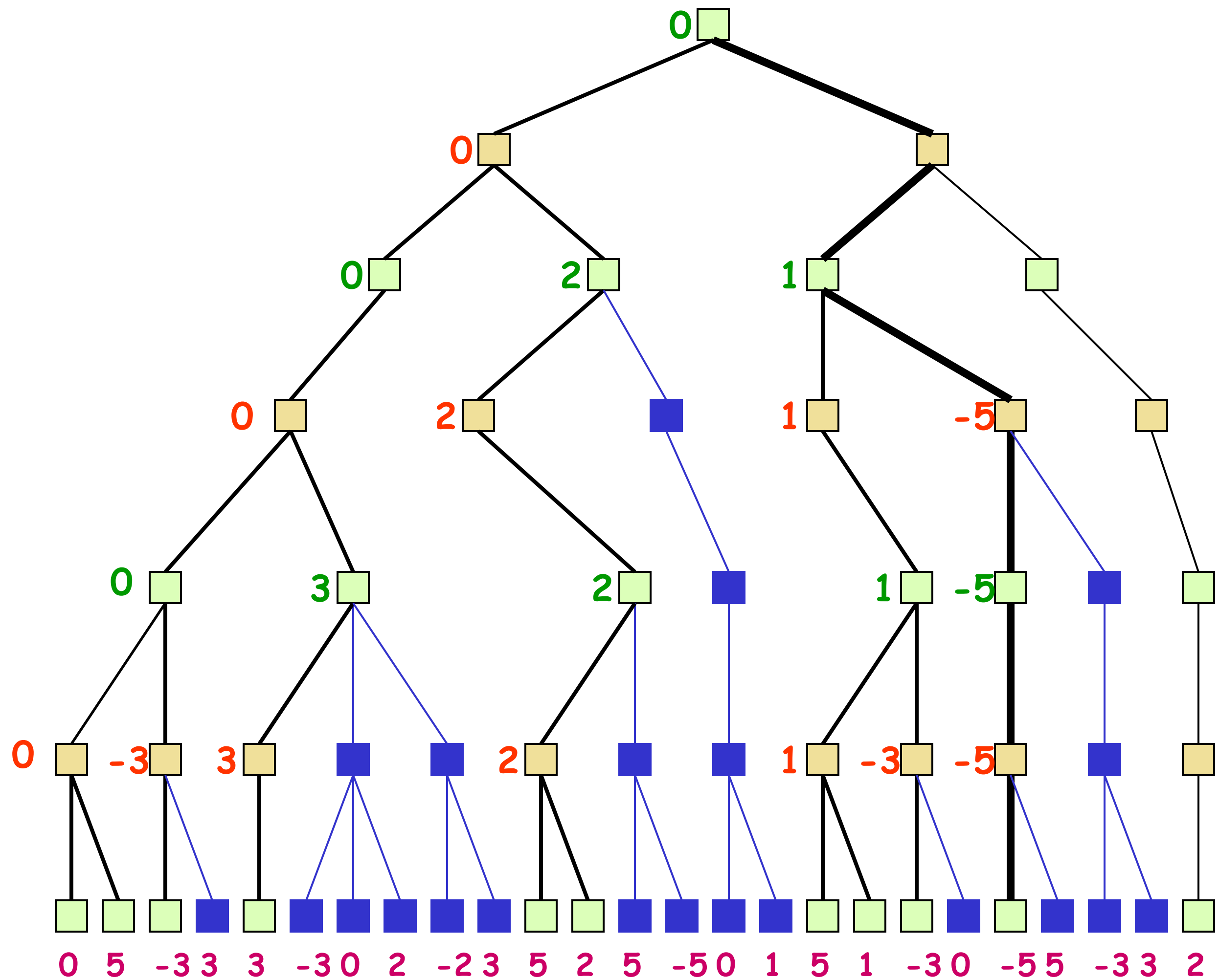
0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

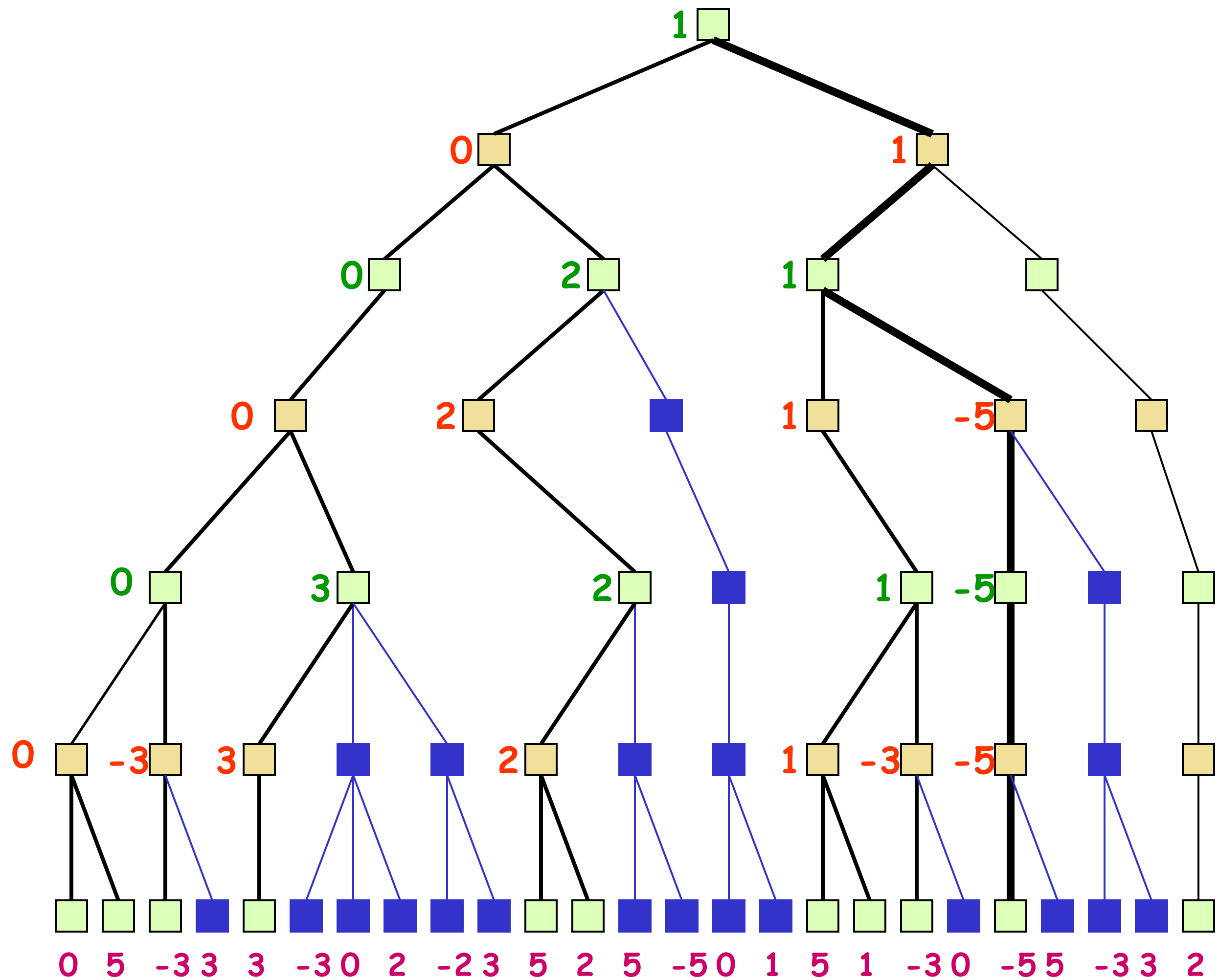


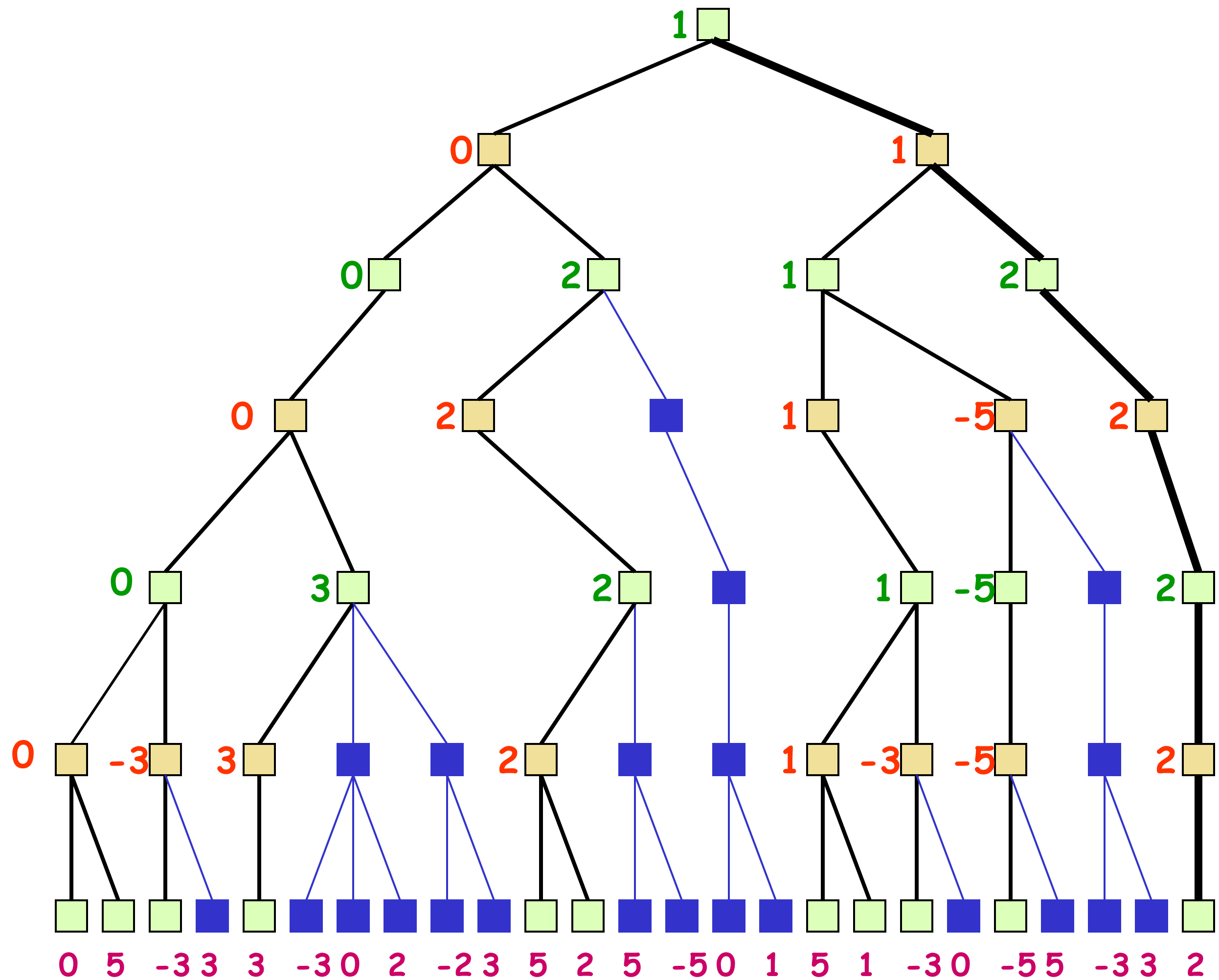
0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2



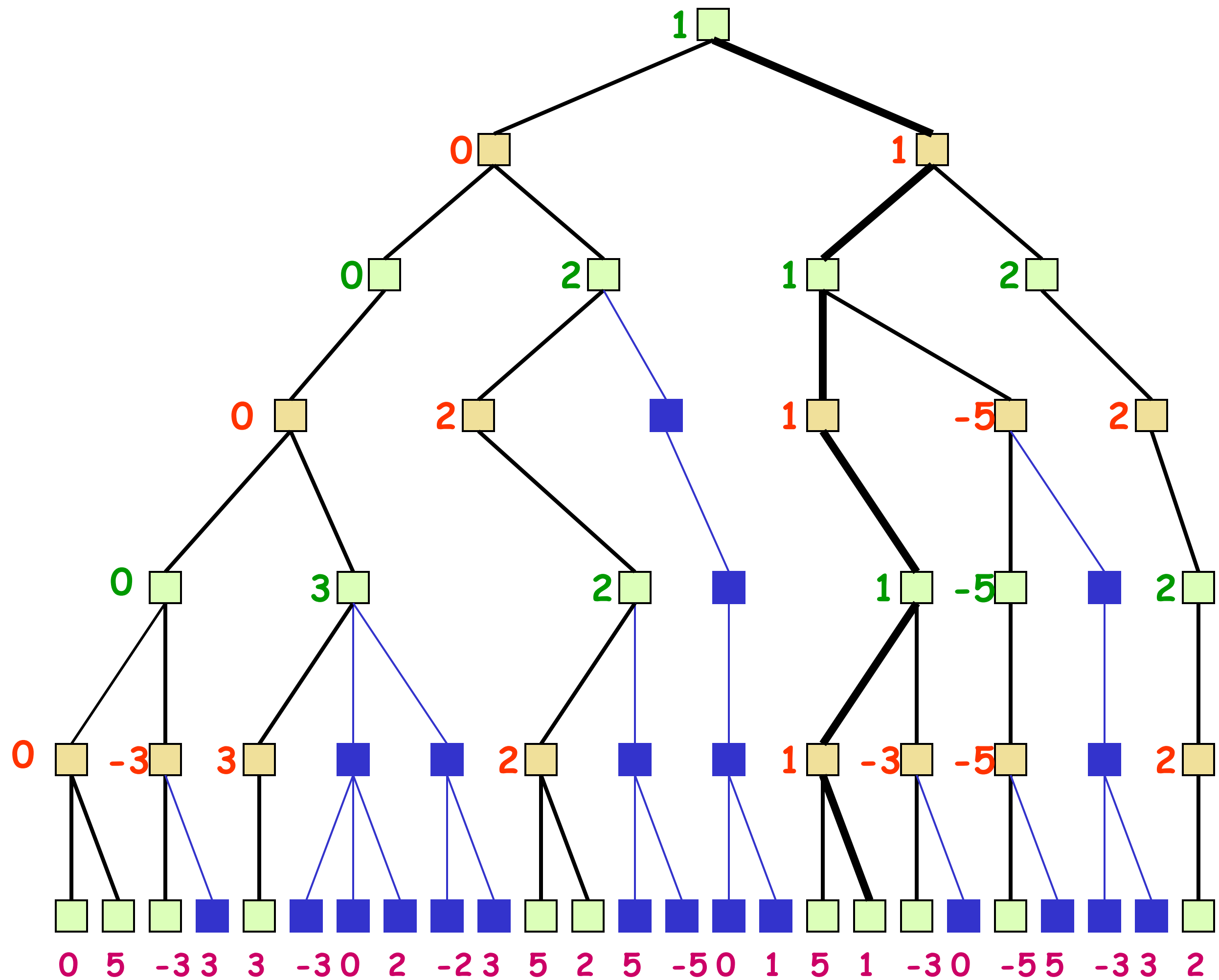
0
0
0 2 1
0 2 1 -5
0 3 2 1 -5
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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



Effectiveness of alpha-beta

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

Other Improvements

- **Adaptive horizon + iterative deepening**
- **Extended search**: retain k>1 best paths (not just one) extend tree at greater depth below their leaf nodes to help dealing 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
- **Null-move** search: assume player forfeits move; do shallow analysis of tree; result must surely be worse than if player had moved. Can recognize moves that should be explored fully.