Adversarial Search
Aka Games

Chapter 6

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

Overview

- Game playing
  - State of the art and resources
  - Framework
- Game trees
  - Minimax
  - Alpha-beta pruning
  - Adding randomness

Why study games?

- Interesting, hard problems that require minimal "initial structure"
- Clear criteria for success
- A way to study problems involving {hostile, adversarial, competing} agents and the uncertainty of interacting with the natural world
- People have used them to asses their intelligence
- Fun, good, easy to understand, PR potential
- Games often define very large search spaces
  - chess 35^{100} nodes in search tree, 10^{40} legal states

State of the art

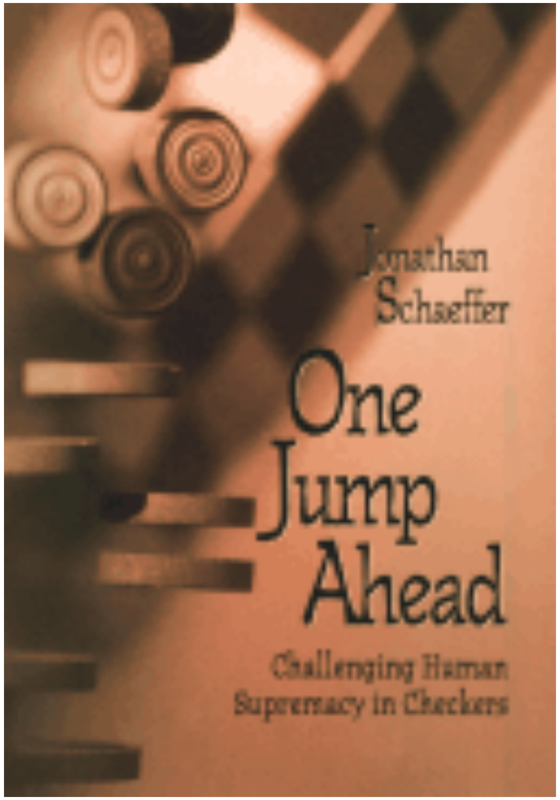
- **Chess**:
  - Deep Blue beat Gary Kasparov in 1997
  - Garry Kasparov vs. Deep Junior (Feb 2003): tie!
  - Kasparov vs. X3D Fritz (November 2003): tie!
- **Checkers**: Chinook is the world champion
- **Checkers:** has been solved exactly – it's a draw!
- **Go**: Computers starting to achieve expert level
- **Bridge**: Expert computer players exist, but no world champions yet
- **Poker:** Poki regularly beats human experts
- Check out the U. Alberta Games Group

Chinook

- Chinook is the World Man-Machine Checkers Champion, developed by researchers at the University of Alberta
- It earned this title by competing in human tournaments, winning the right to play for the (human) world championship, and eventually defeating the best players in the world
- Play Chinook online
- One Jump Ahead: Challenging Human Supremacy in Checkers, Jonathan Schaeffer, 1998
- See Checkers Is Solved, J. Schaeffer, et al., Science, v317, n5844, pp1518-22, AAAS, 2007.

The board set for play

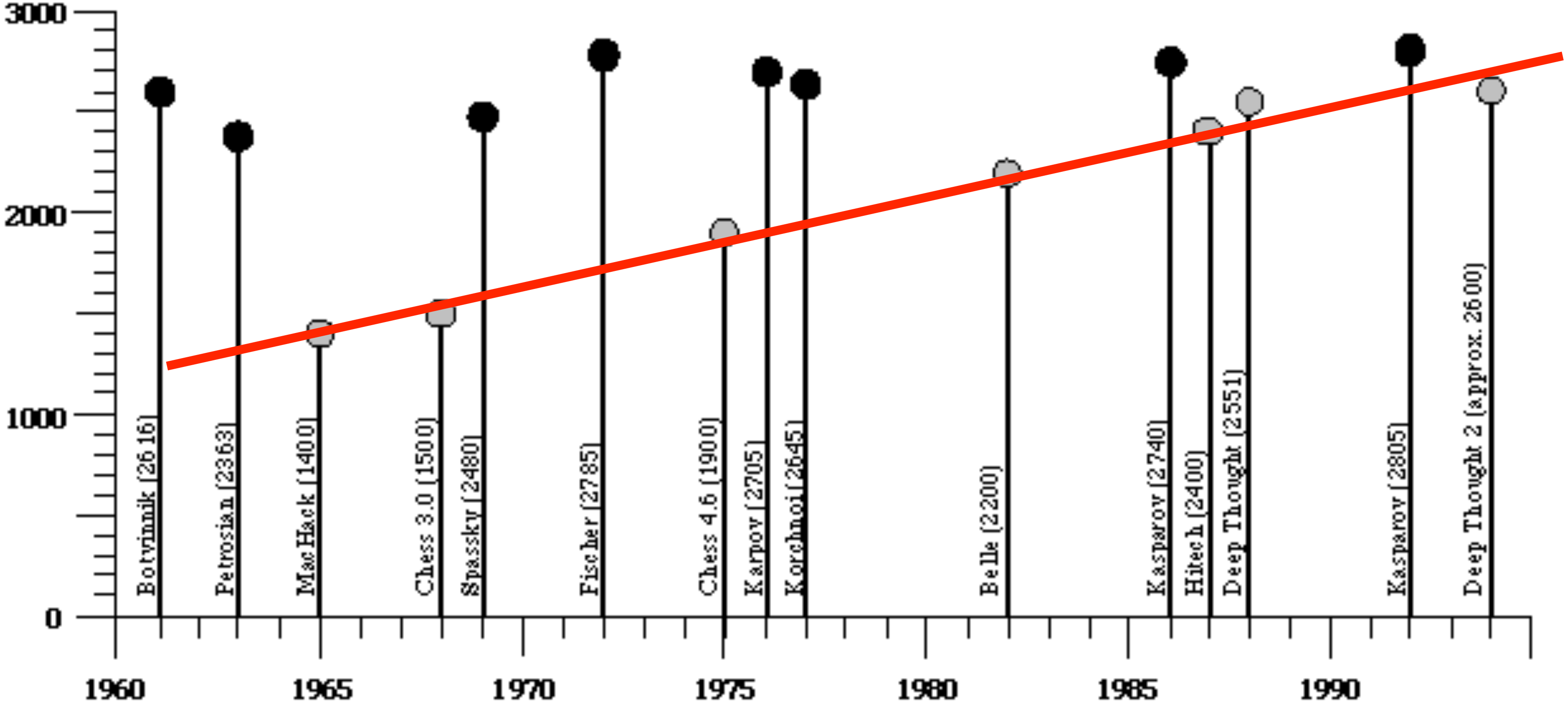
Red to play



Chess early days

- **1948**: Norbert Wiener's *Cybernetics* describes how a chess program could be developed using a depth-limited minimax search with an evaluation function
- **1950**: Claude Shannon publishes Programming a Computer for Playing Chess
- **1951**: Alan Turing develops on paper the first program capable of playing a full game of chess
- **1962**: Kotok and McCarthy (MIT) develop first program to play credibly
- **1967**: Mac Hack Six, by Richard Greenblatt et al. (MIT) defeats a person in regular tournament play

Ratings of human & computer chess champions



1997

Location: http://www.chess.ibm.com/home/html/b.html

may 11th game 6 : may 11 @ 3:00PM EDT | 19:00 GMT kasparov 2.5 deep blue 2.5

Deep Blue Wins 3.5 to 2.5

With a dramatic victory in Game 6, Deep Blue won its six-game rematch with Champion Garry Kasparov

Commentary
George Plimpton on chess, Kasparov, and the limitations of computers
Read the article

Commentary
Vishwanathan Anand on the legacy of Kasparov vs. Deep Blue
Read the article

Club Kasparov
Visit the virtual home of the world's greatest chess player.

Guest essays
Thoughts on chess, computers, and what it all means
Read the essays...

Community
During the rematch, more than 20,000 people from 120 countries joined the community to talk about the match.

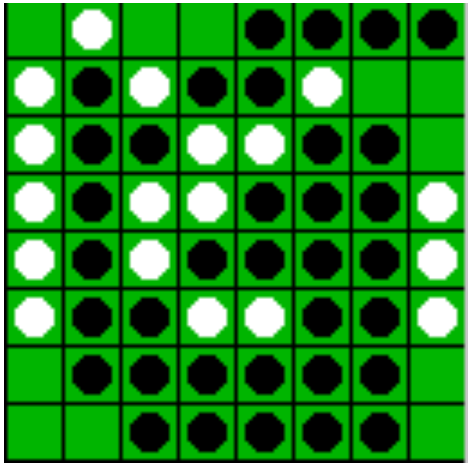
Clips from the rematch
Video footage from the games
Highlights from the games

Document: Done

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

Othello: Murakami vs. Logistello

Takeshi Murakami
World Othello Champion



open sourced

- 1997: The Logistello software crushed Murakami, 6 to 0
- Humans can not win against it
- Othello, with 10^{28} states, is still not solved

Go: Goemate vs. a young player

Name: Chen Zhixing
Profession: Retired
Computer skills:
    self-taught programmer
Author of Goemate (arguably the
    best Go program available today)

Gave Goemate a 9 stone handicap and still easily beat the program, thereby winning $15,000

Go: Goemate vs. ??

Name: Chen Zhixing
Profession: Retired
Computer skills:

Go has too high a branching factor for existing search techniques

Current and future software must rely on huge databases and pattern-recognition techniques

thereby winning $15,000

How can we do it?

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?

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 non-negative 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

- Alan Turing’s function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white’s pieces and b(n) = sum of black’s
  - Traditional piece values are -- Pawn:1; Knight, bishop: 3; Rook: 5; Queen: 9

Evaluation function examples

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

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

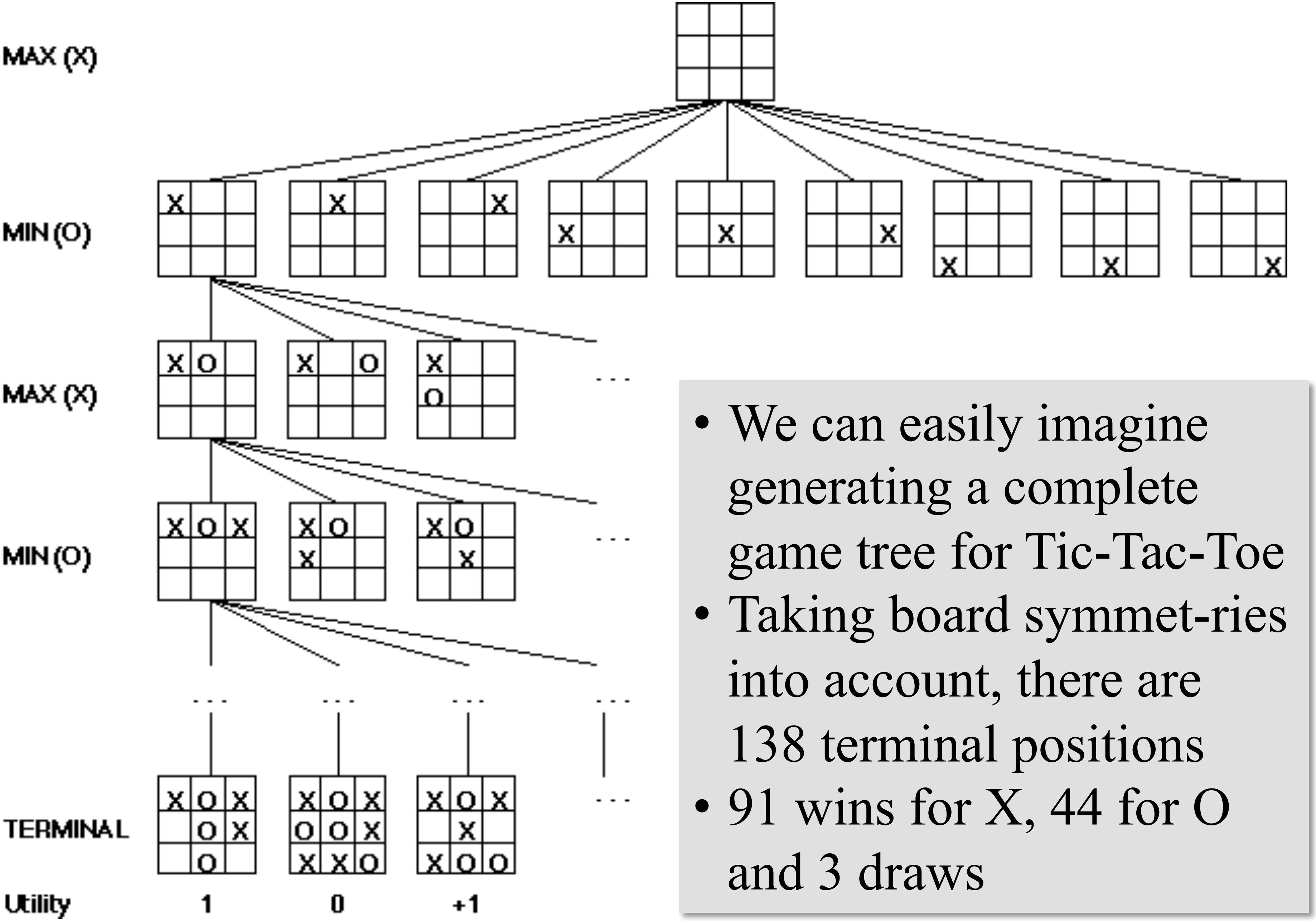
- Example features for chess are piece count, piece values, piece placement, squares controlled, etc.
- IBM's chess program Deep Blue had >8K features in its evaluation function

That’s not how people play

- People use *look ahead*

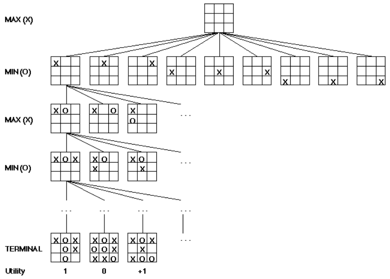
  i.e., enumerate actions, consider opponent’s possible responses, REPEAT

- Producing a complete **game tree** is only possible for simple games
- So, generate a partial game tree for some number of plys
  - Move = each player takes a turn
  - Ply = one player’s turn
- What do we do with the game tree?



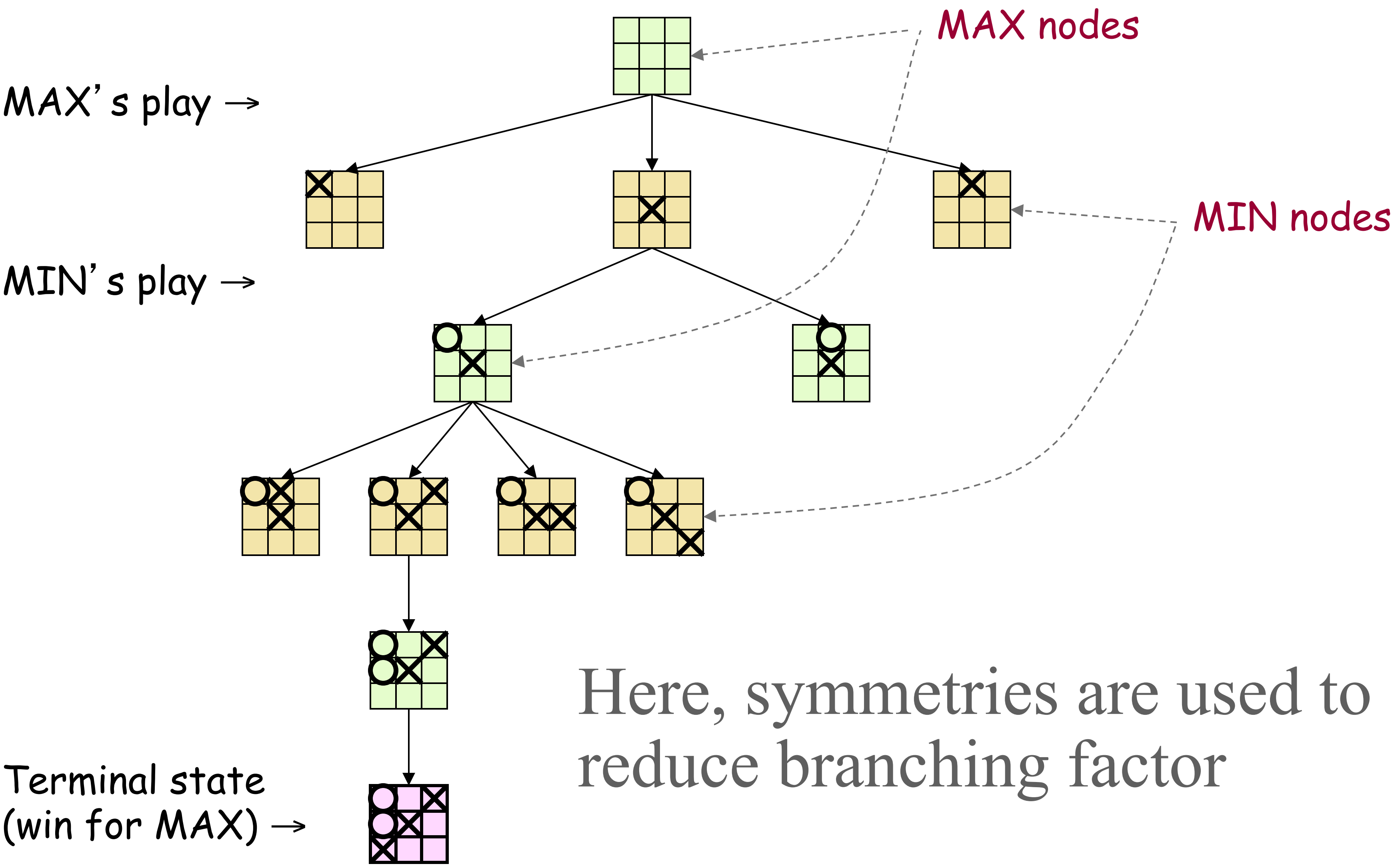
- We can easily imagine generating a complete game tree for Tic-Tac-Toe
- Taking board symmet-ries into account, there are 138 terminal positions
- 91 wins for X, 44 for O and 3 draws

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

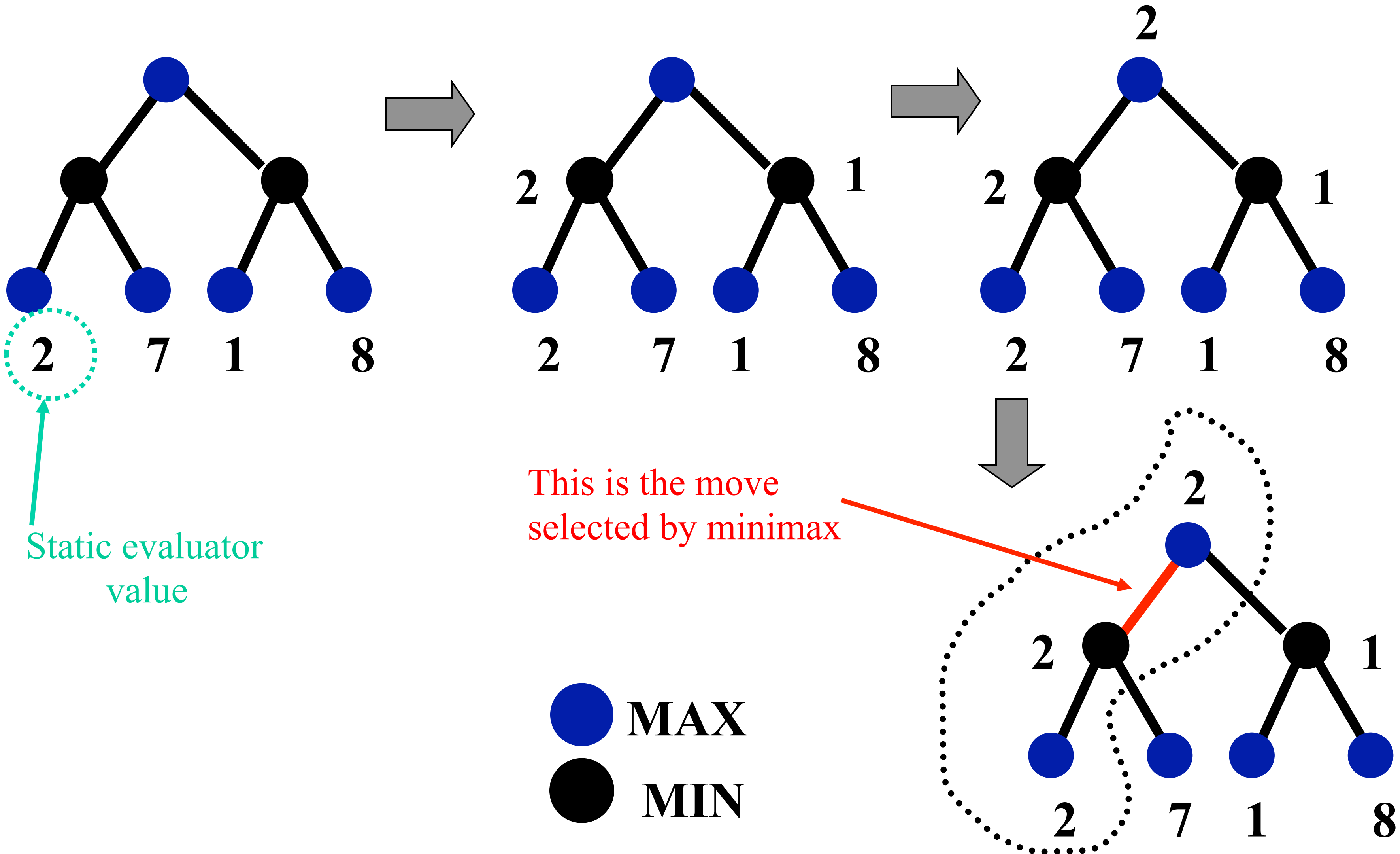
- Create MAX node with current board configuration
- Expand nodes to some **depth** (a.k.a. **ply**) of lookahead in game
- Apply evaluation function at each leaf node
- *Back up* values for each non-leaf node until value is computed for the root node
  - At MIN nodes, backed-up value is **minimum** of values associated with its children.
  - At MAX nodes, backed-up value is **maximum** of values associated with its children.
- Pick operator associated with child node whose backed-up value determined value at the 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 Gesellschaftsspiele* 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, the best payoff possible for player 1 is V, and (b) given player 1's strategy, the best payoff possible for player 2 is –V.
- You can think of this as:
  - Minimizing your maximum possible loss
  - Maximizing your minimum possible gain

Minimax Algorithm



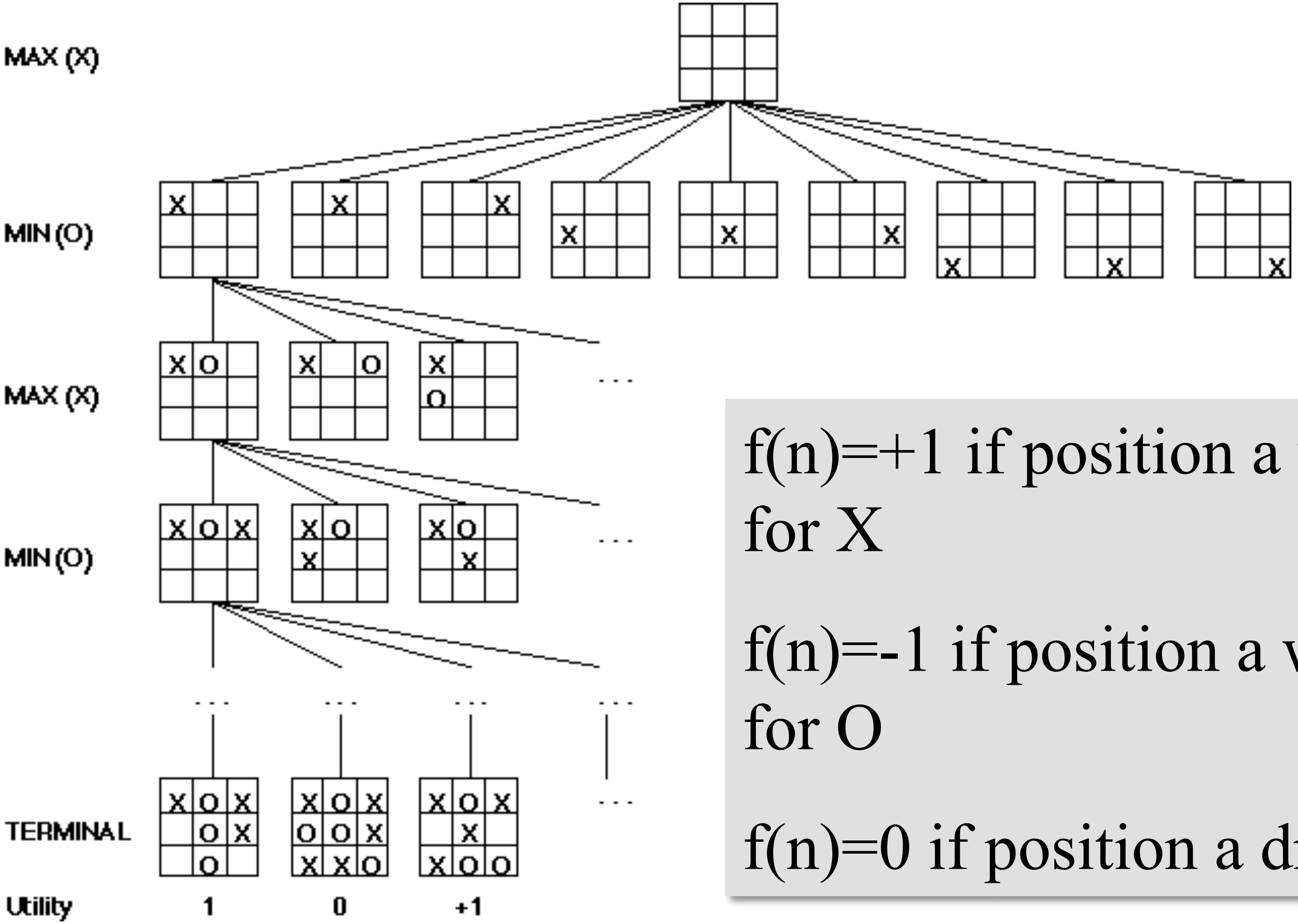
Static evaluator value

This is the move selected by minimax

● MAX

● MIN

Partial Game Tree for Tic-Tac-Toe



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

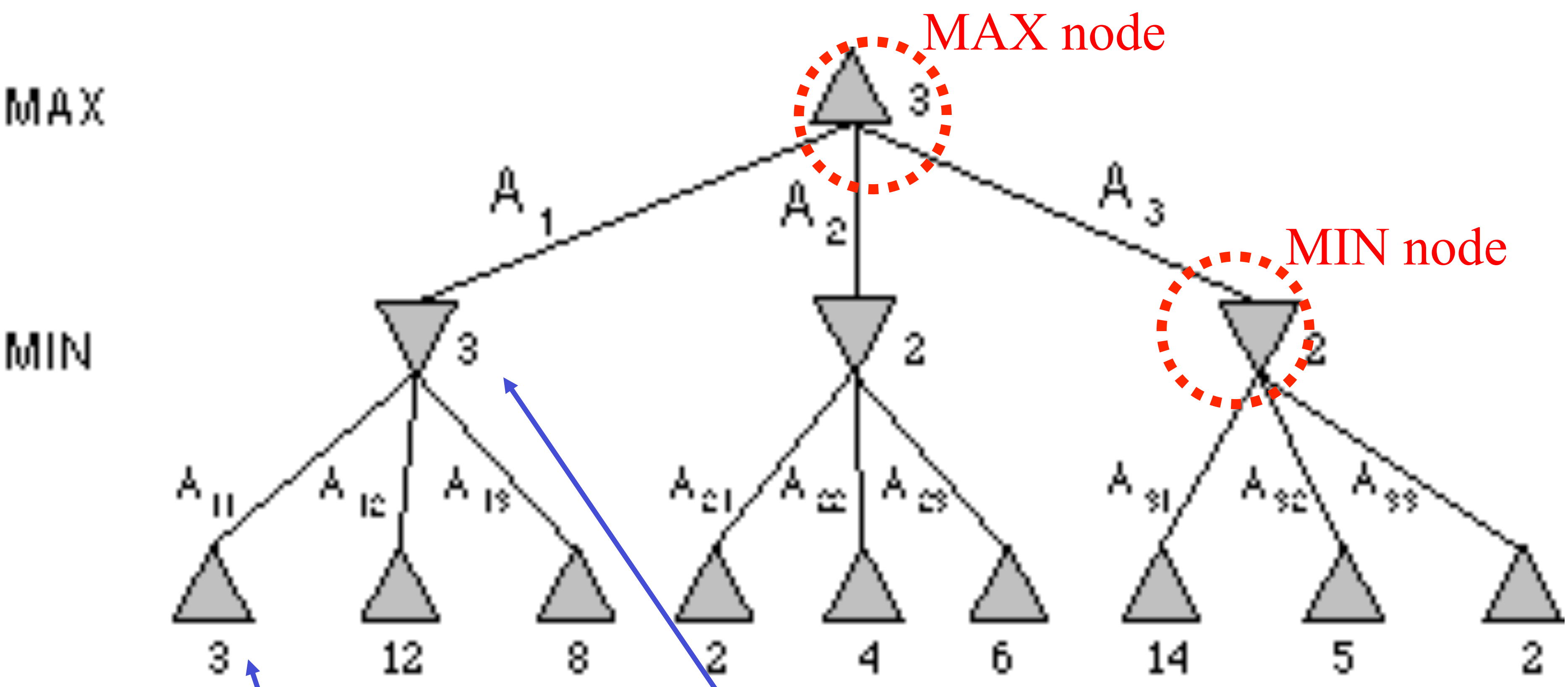
f(n)=-1 if position a win for O

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 that MAX can reach at depth **h** if MIN plays well
  - "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))
- We use a lookup horizon **h** because time to compute a move is limited

Minimax Tree



MAX node

MIN node

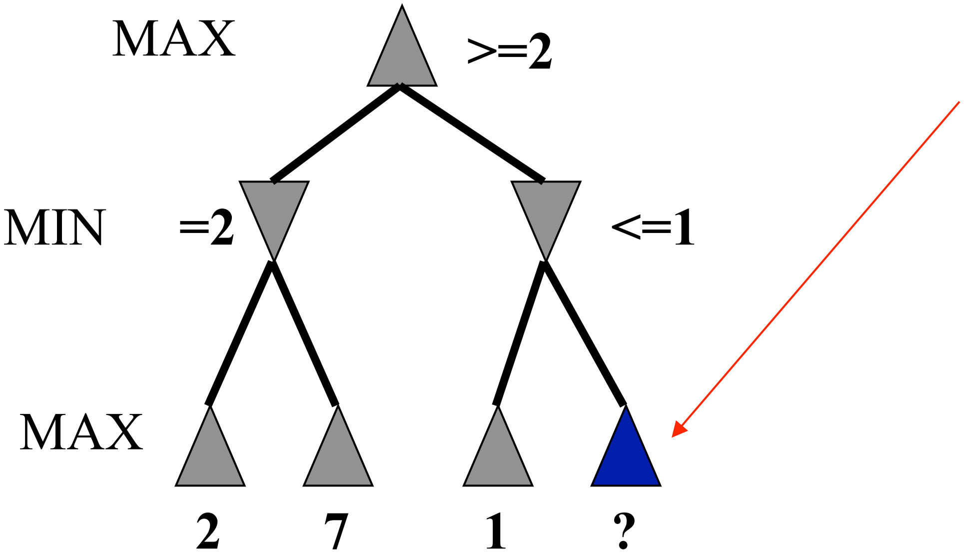
f value

value computed
by minimax

Is that all there is to simple games?

Alpha-beta pruning

- Improve on 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

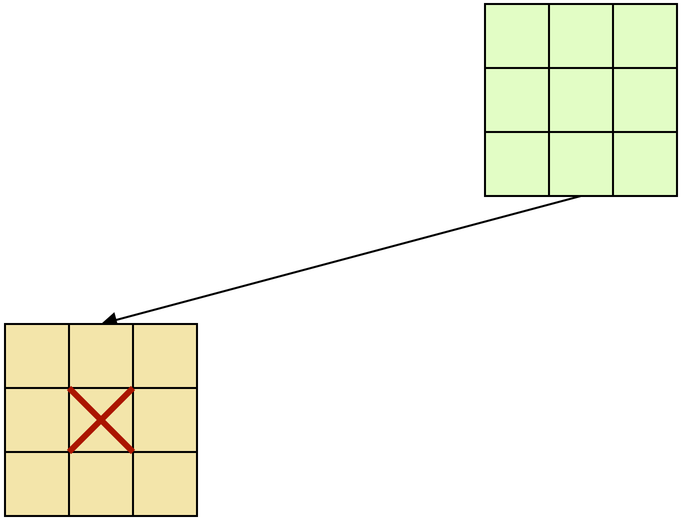


- We don't need to compute the value at this node
- No matter what it is, it can't affect value of the root node

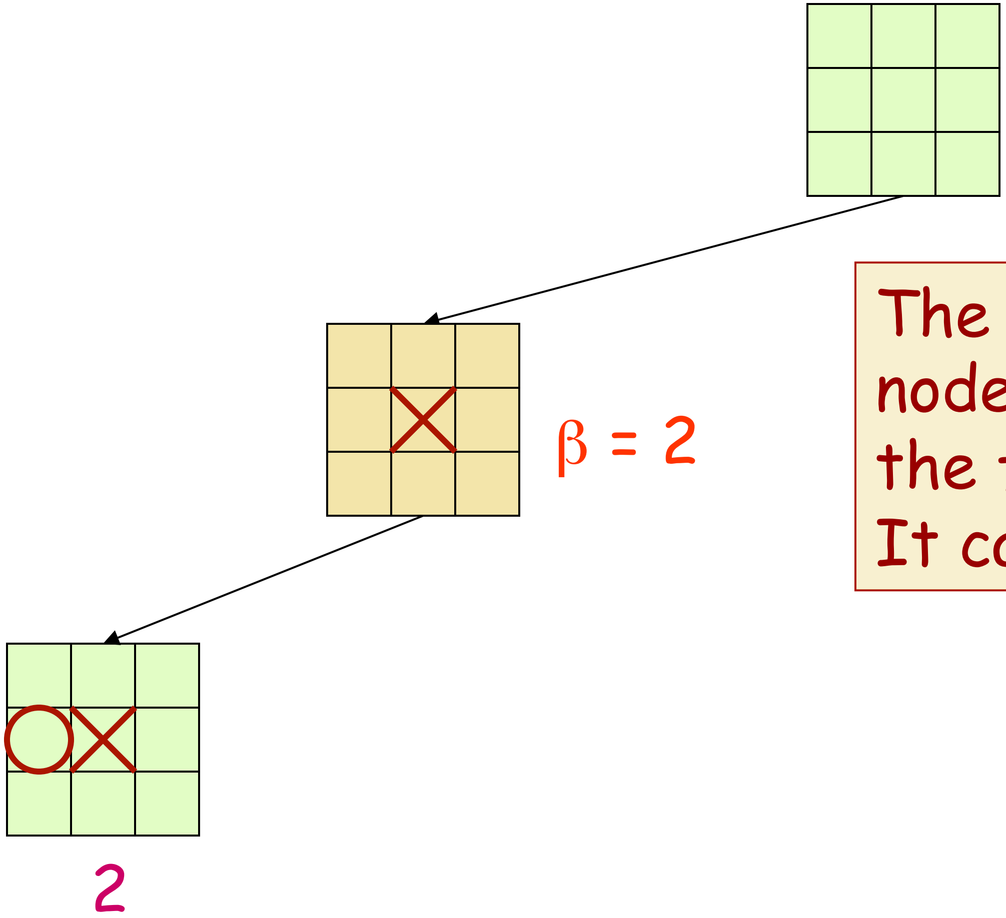
Alpha-beta pruning

- Traverse search tree in depth-first order
- At **MAX** node n, **alpha(n)** = max value found so far
- At **MIN** node n, **beta(n)** = min value found so far
  - Alpha values start at -∞ and only increase, while beta values start at +∞ and only decrease
- **Beta cutoff**: Given MAX node n, cut off search below n (i.e., don't examine any more of n's children) if alpha(n) >= beta(i) for some MIN node ancestor i of n
- **Alpha cutoff:** stop searching below MIN node n if beta(n) <= alpha(i) for some MAX node ancestor i of n

Alpha-Beta Tic-Tac-Toe Example

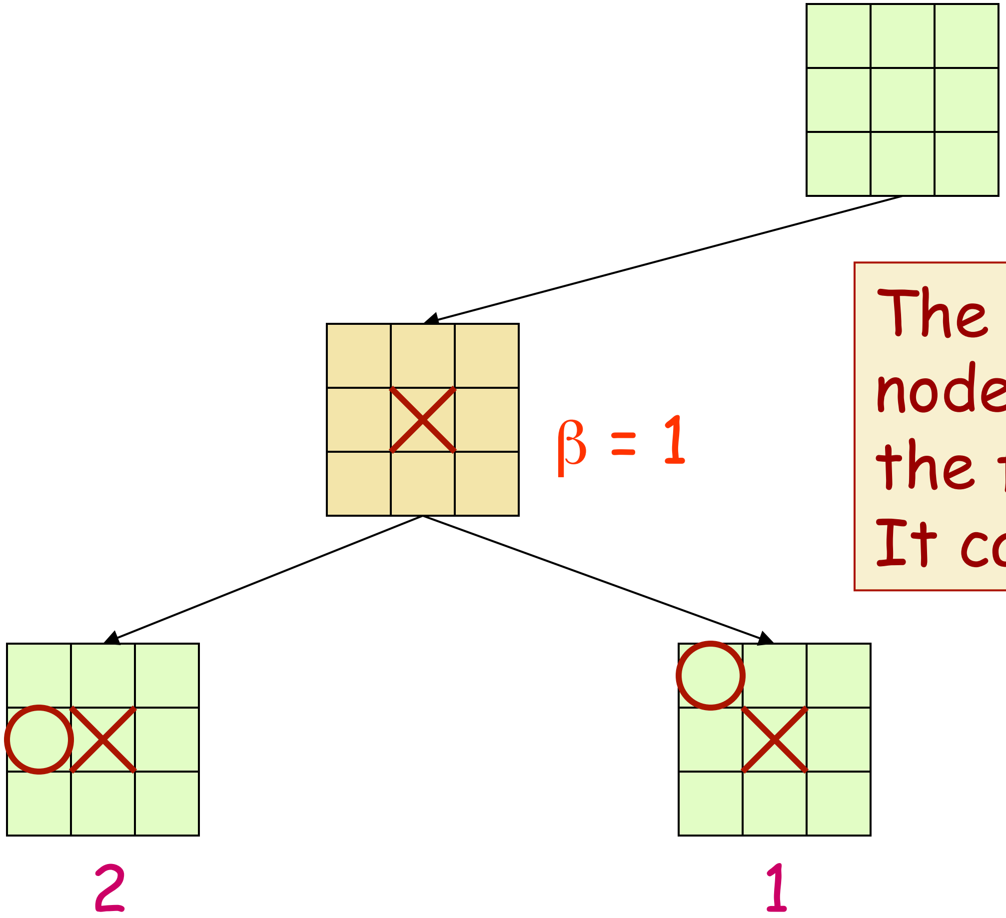


Alpha-Beta Tic-Tac-Toe Example



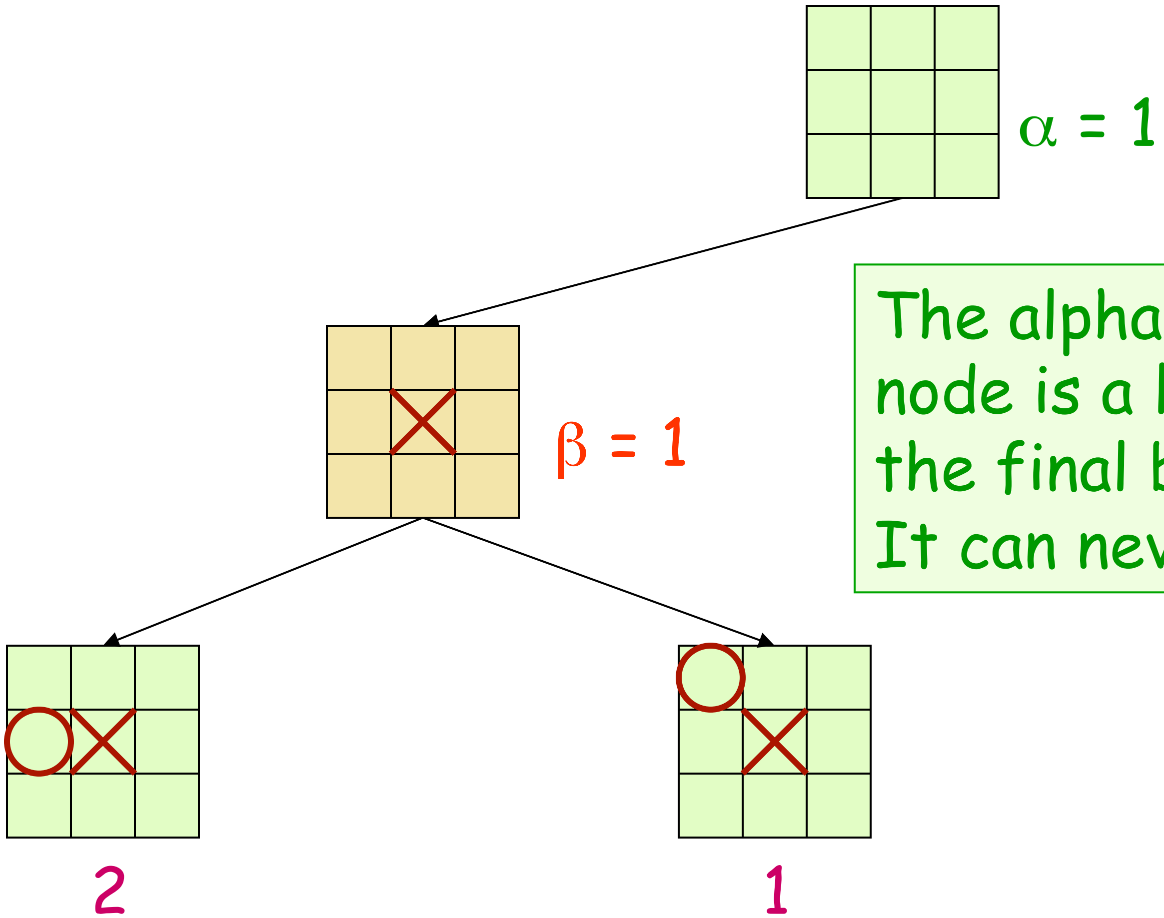
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example



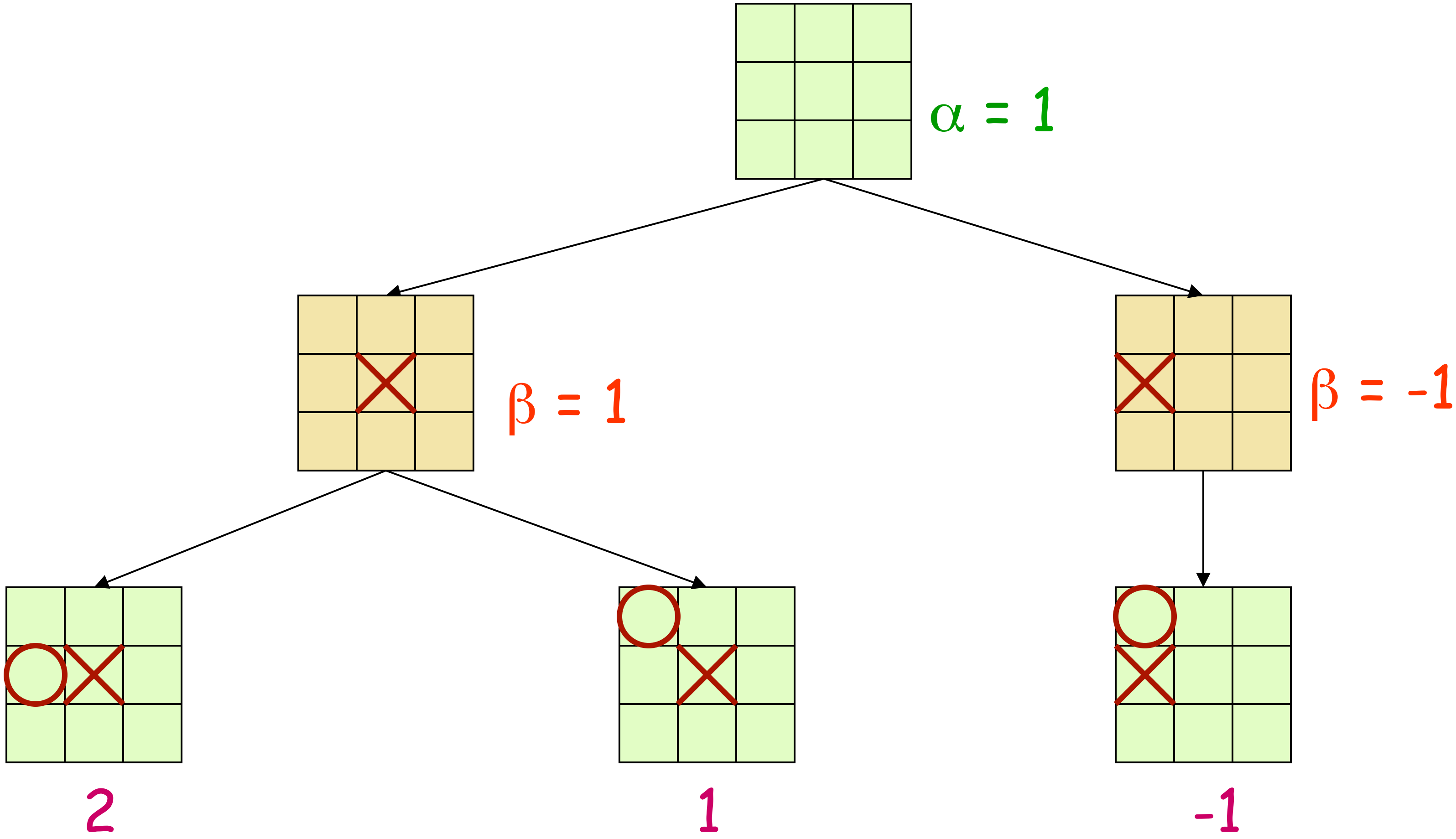
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example

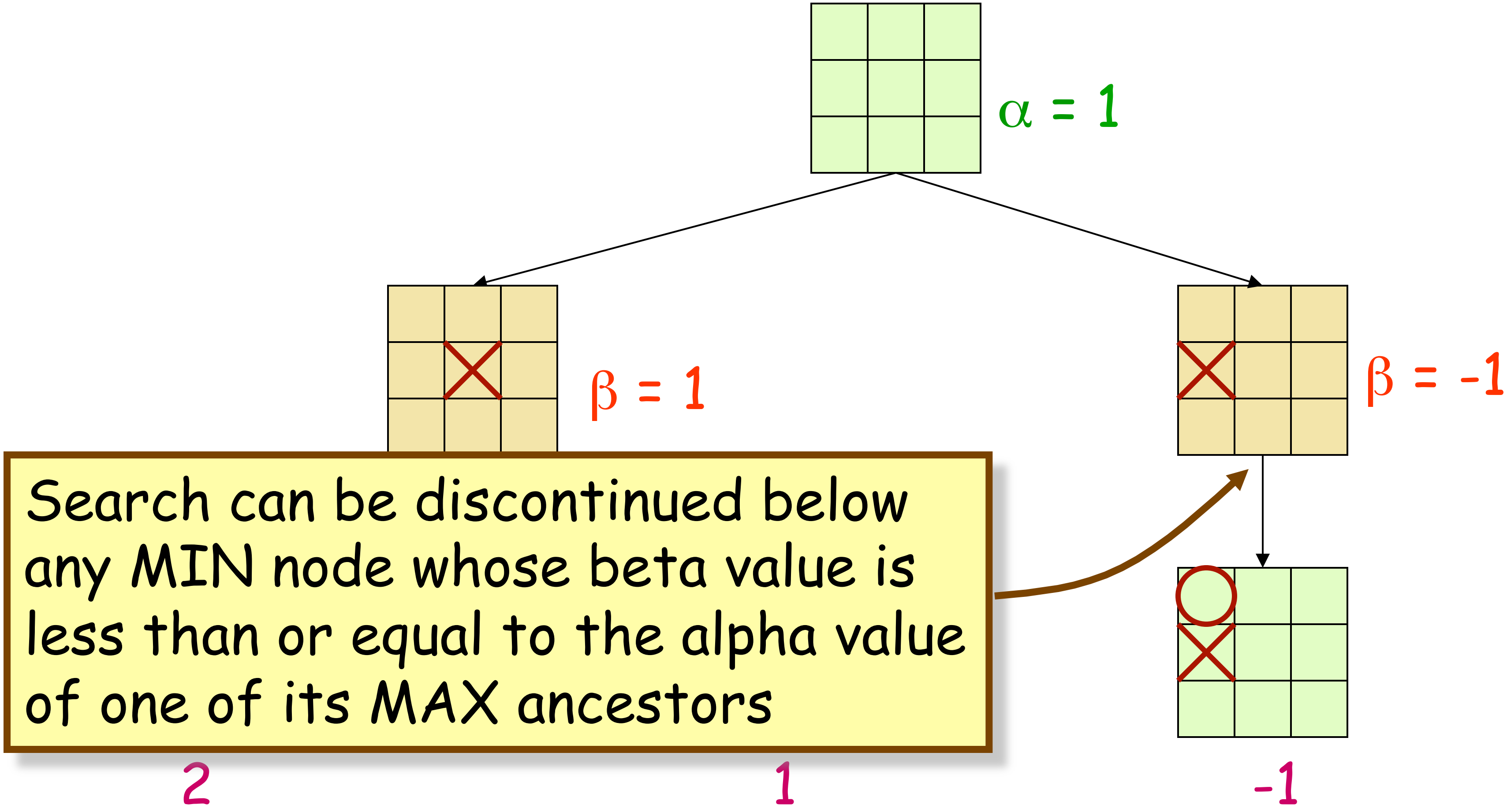


The alpha value of a MAX node is a lower bound on the final backed-up value. It can never decrease

Alpha-Beta Tic-Tac-Toe Example

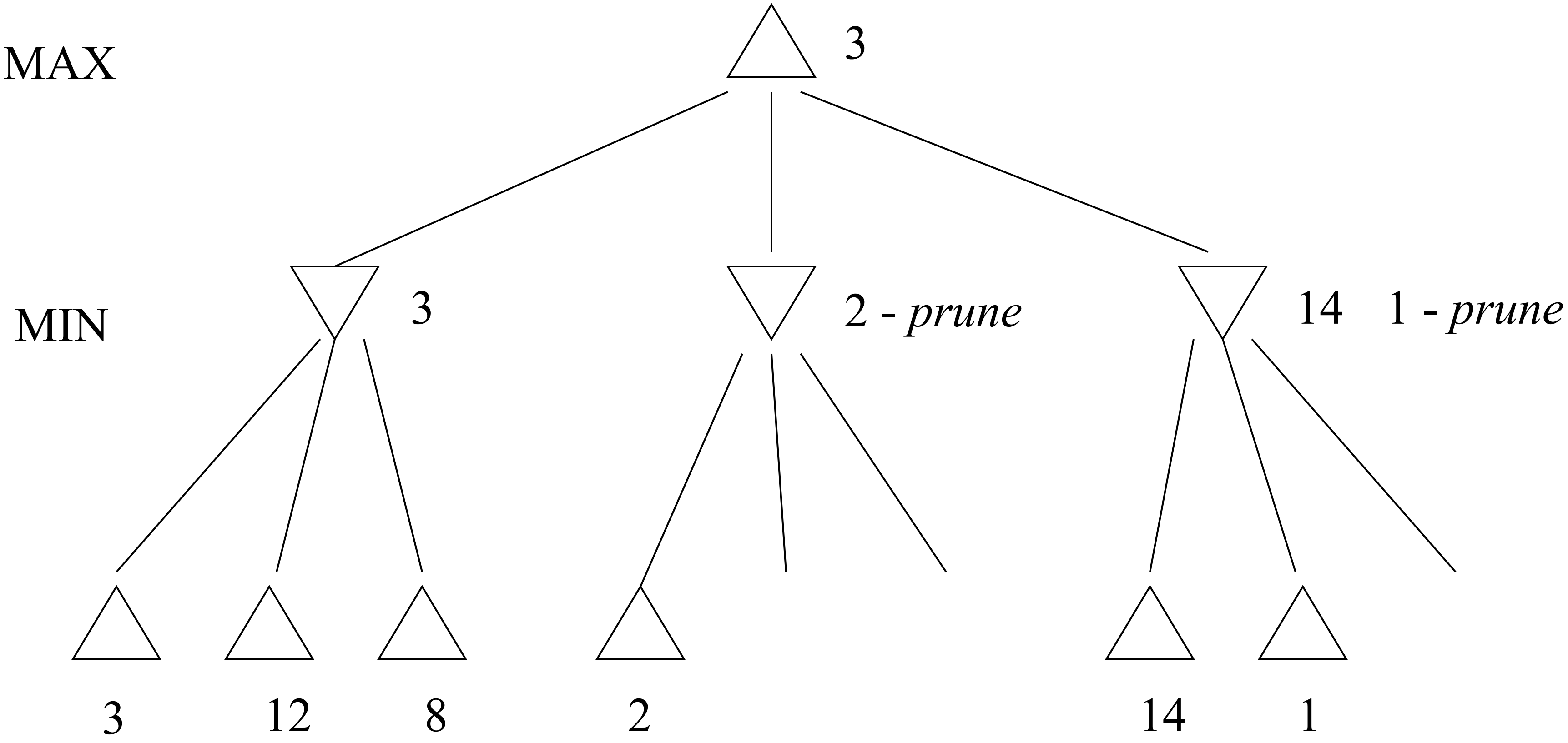


Alpha-Beta Tic-Tac-Toe Example

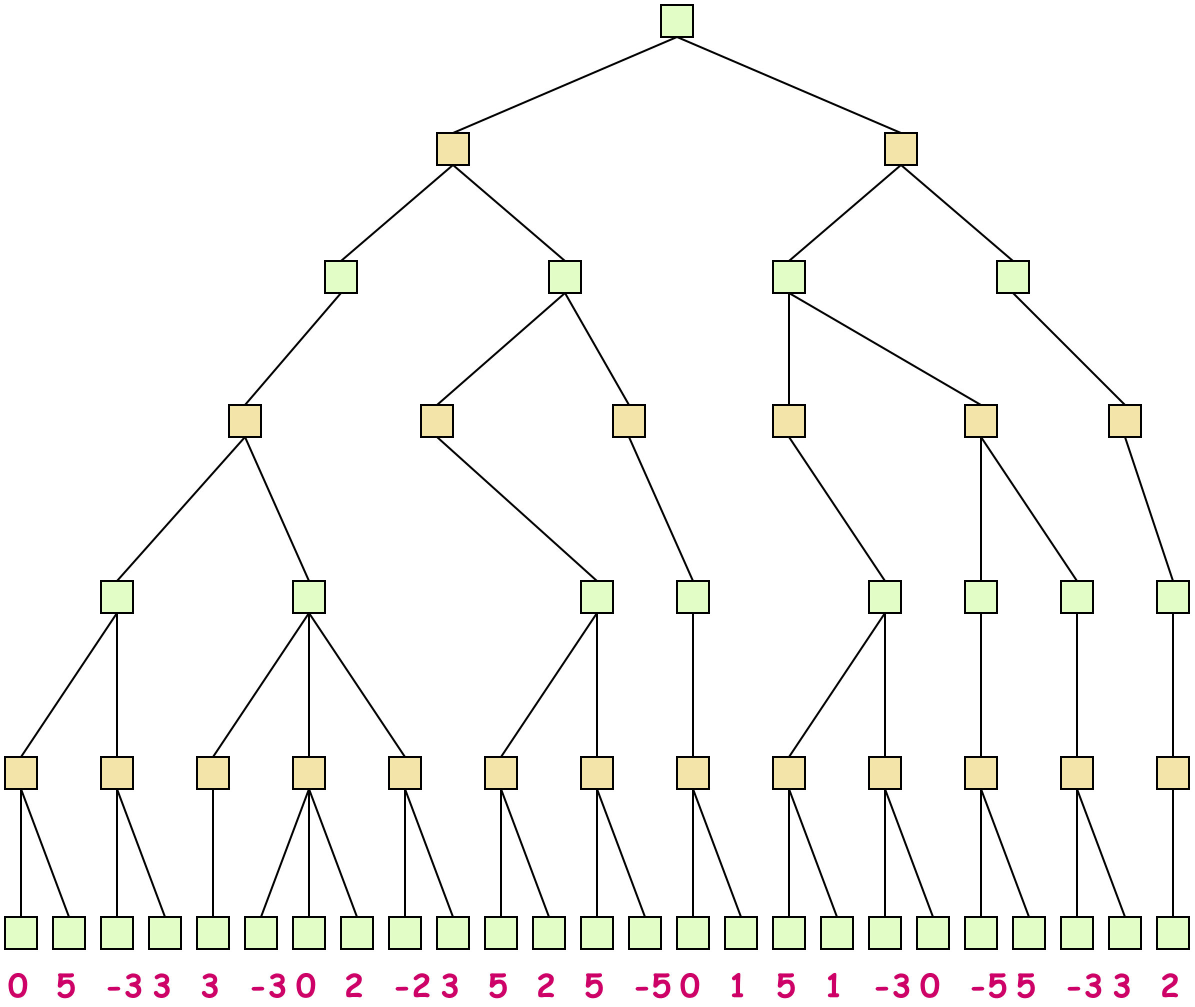


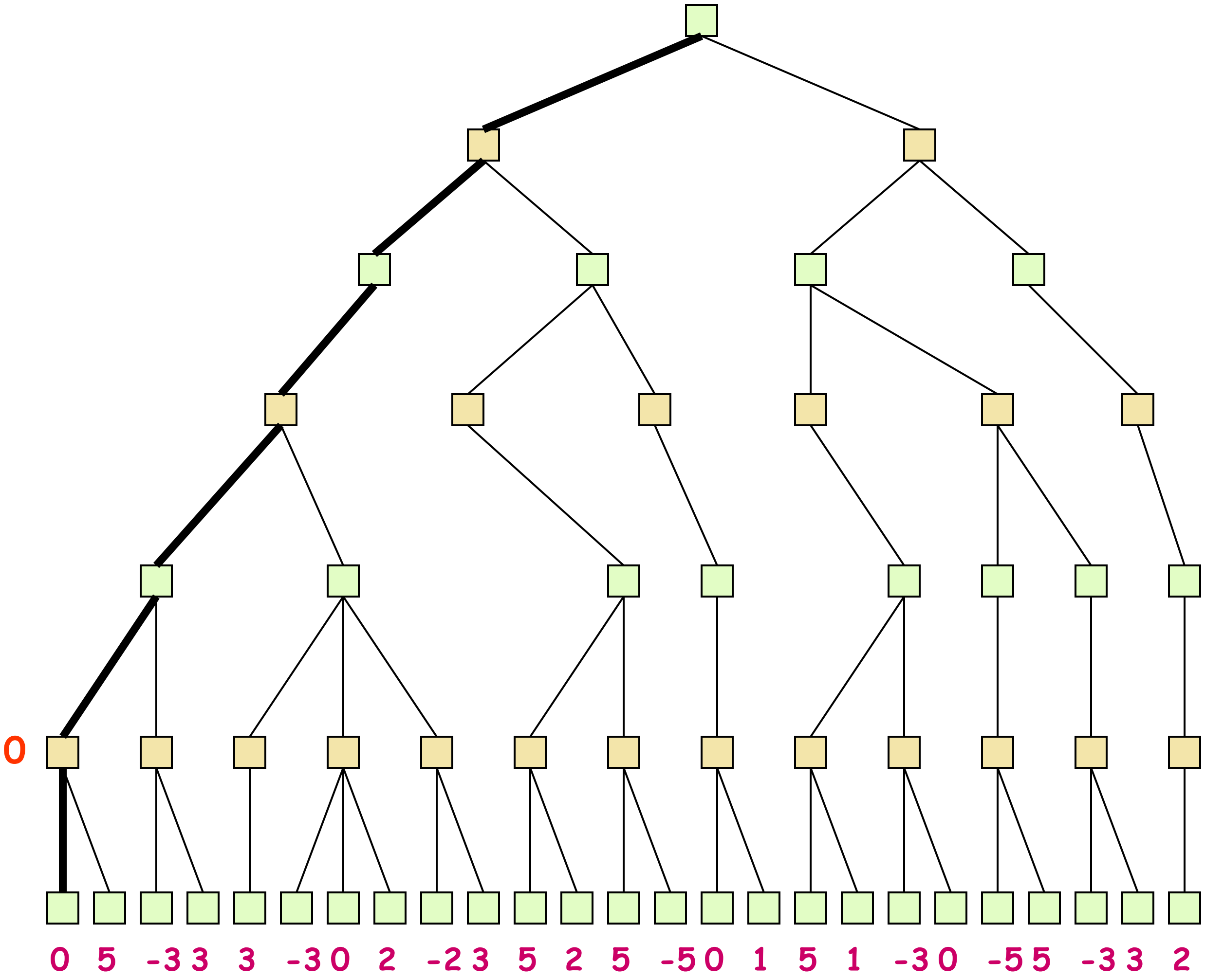
Search can be discontinued below any MIN node whose beta value is less than or equal to the alpha value of one of its MAX ancestors

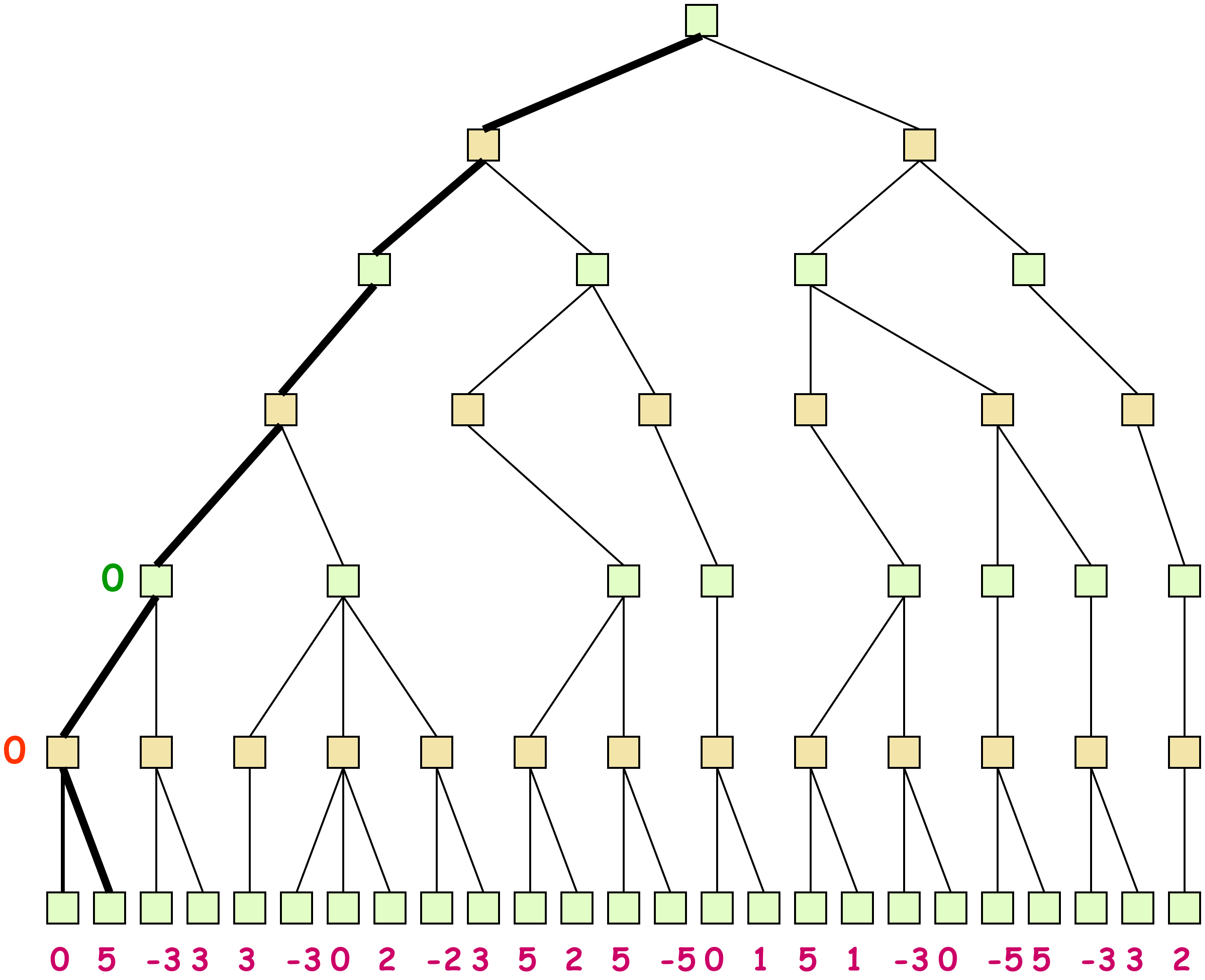
Alpha-beta general example

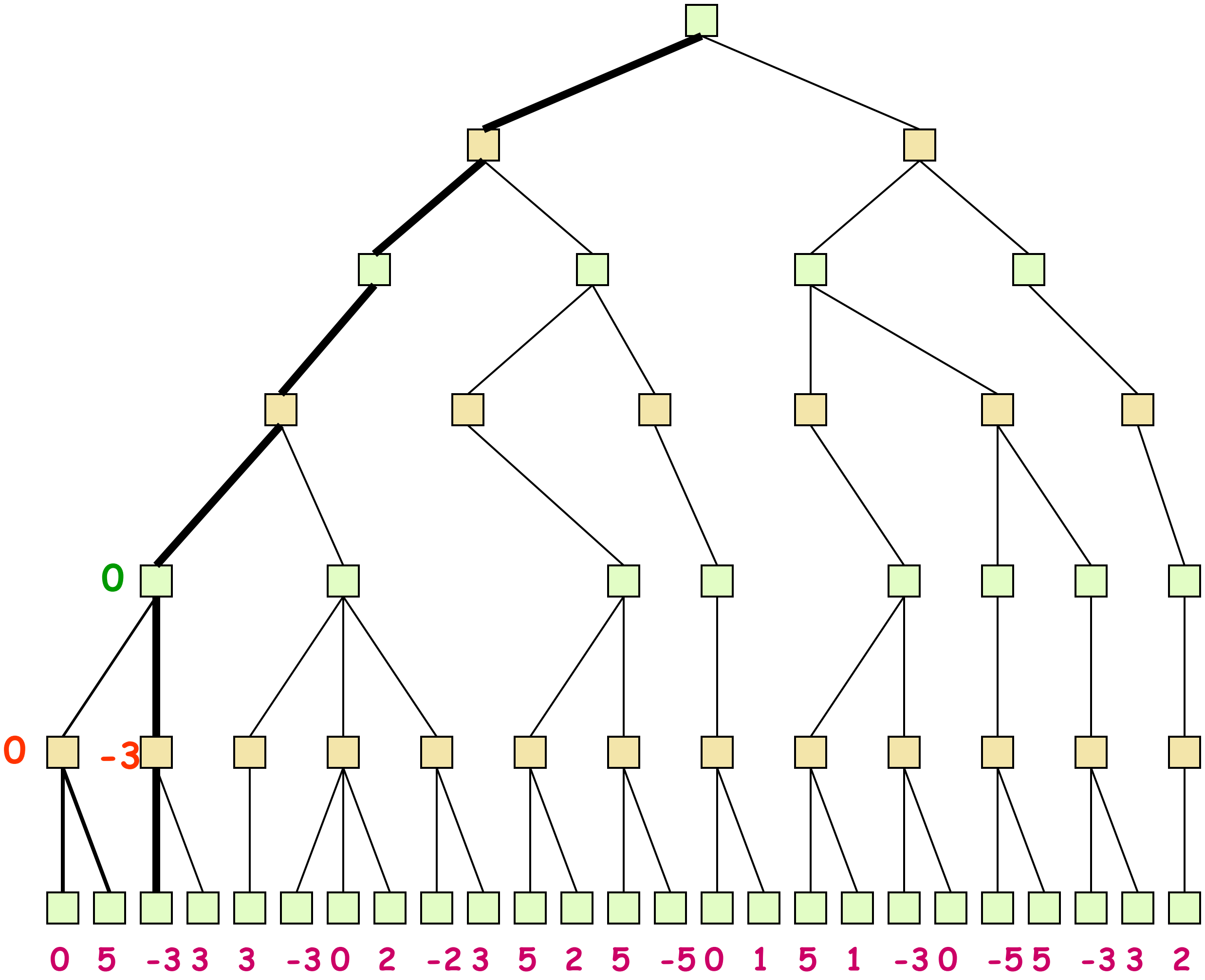


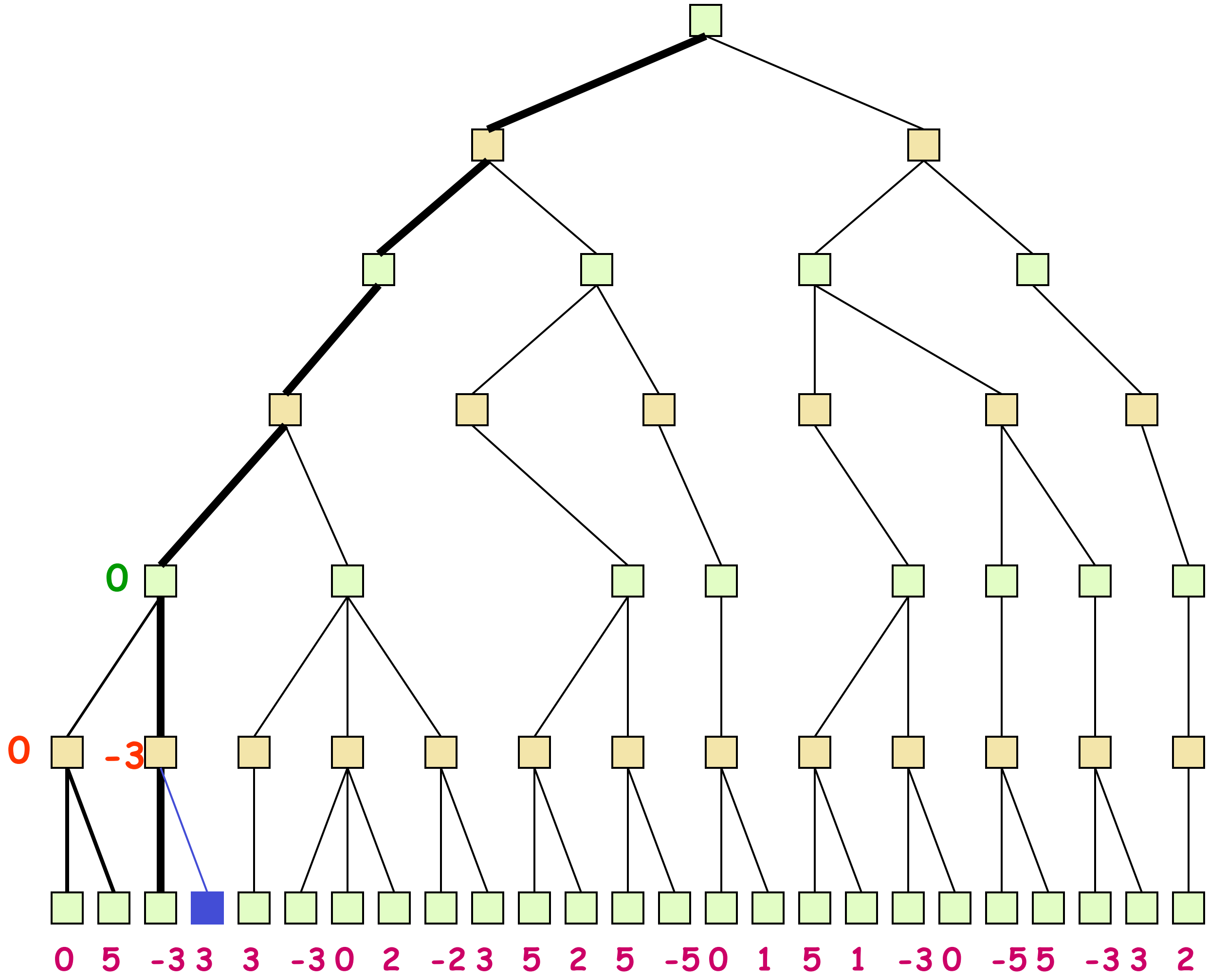
Alpha-Beta Tic-Tac-Toe Example 2

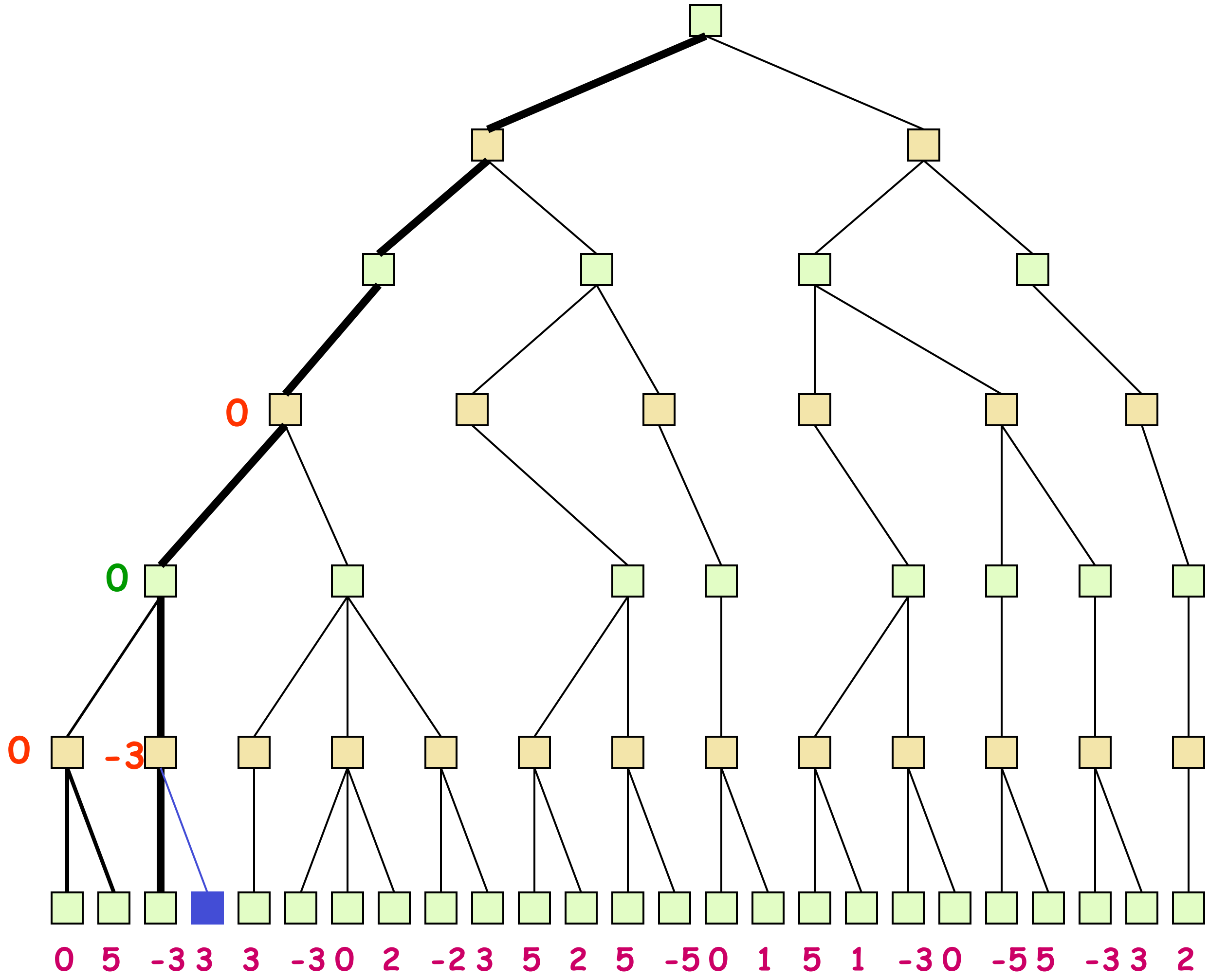


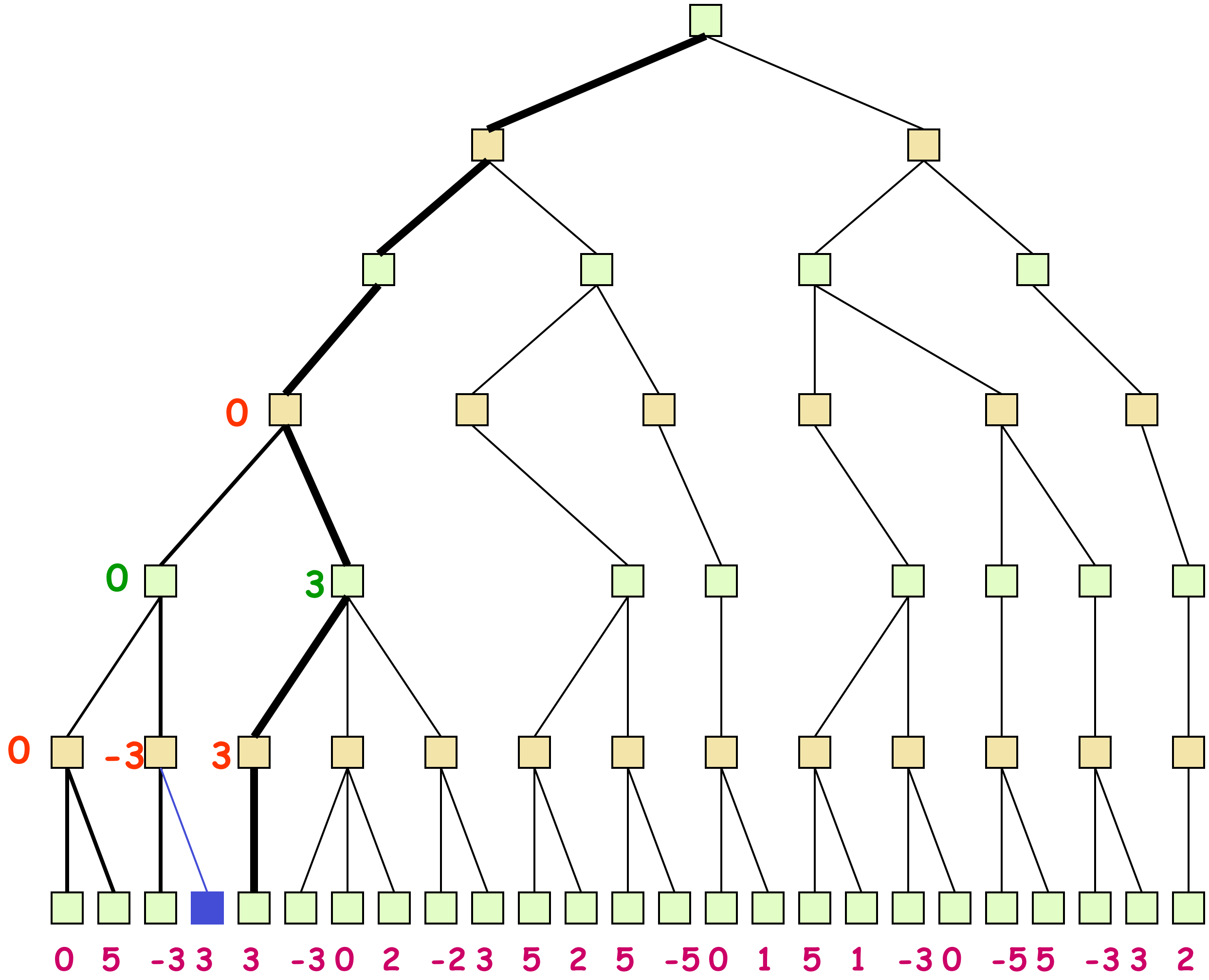

0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

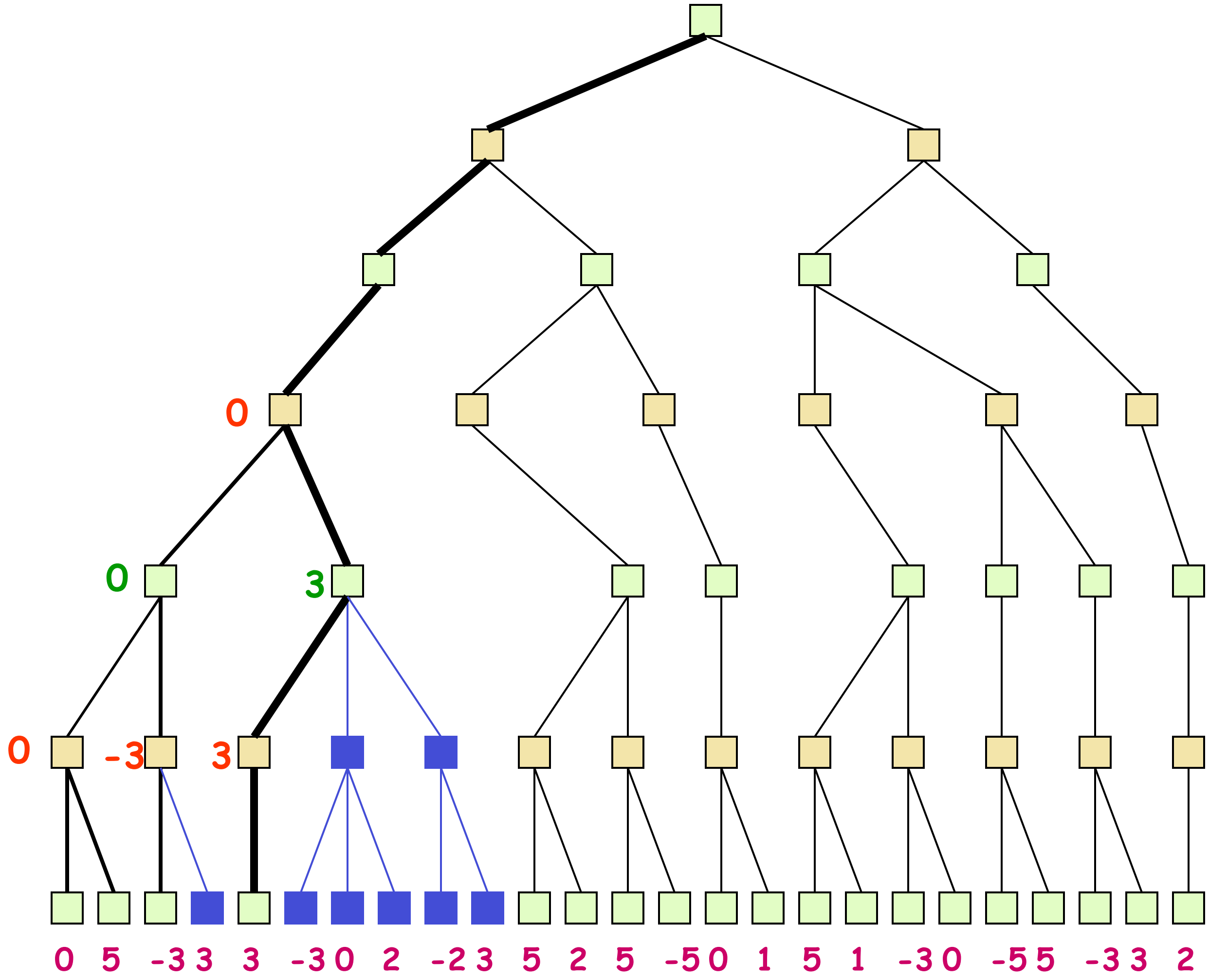

0
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

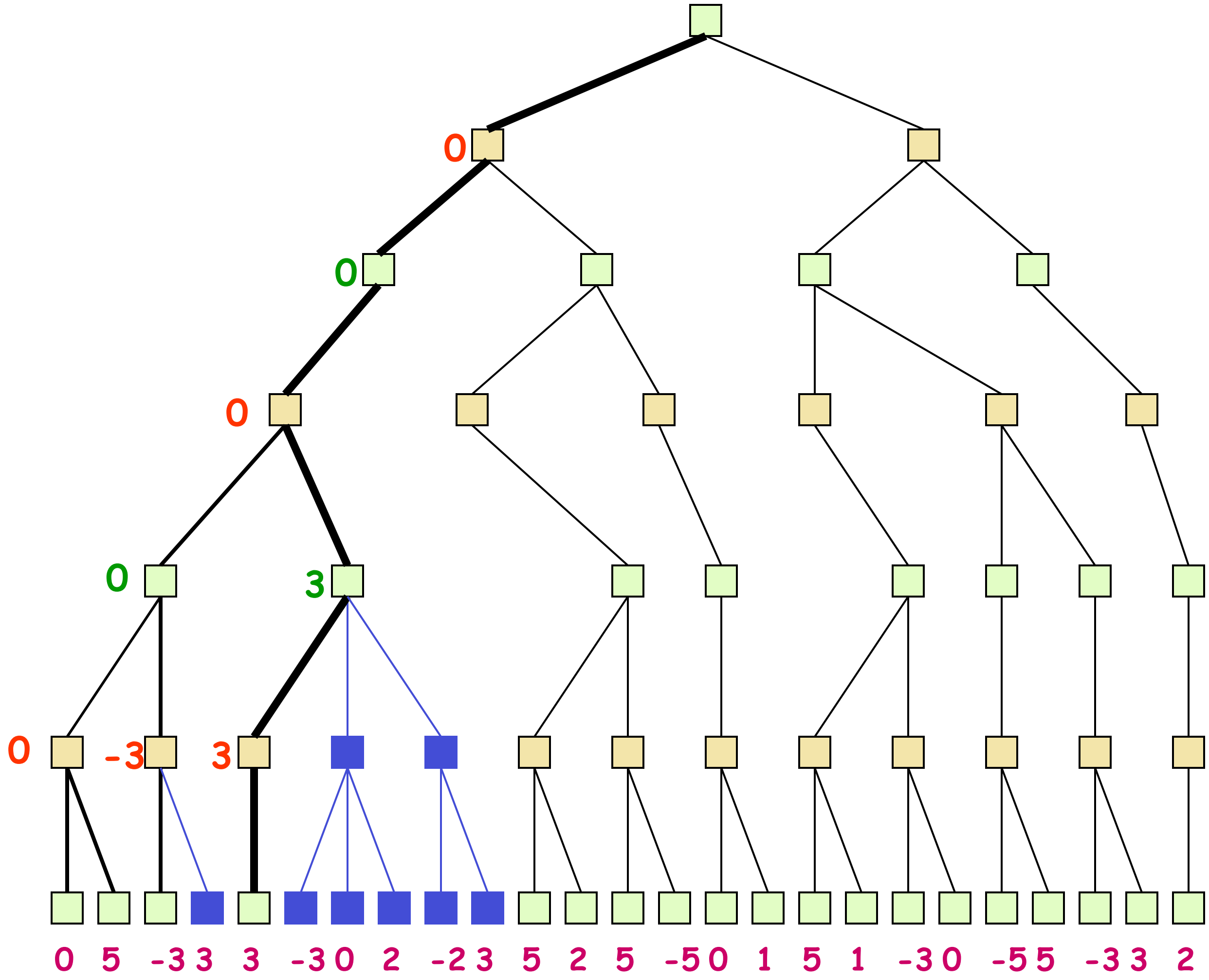

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

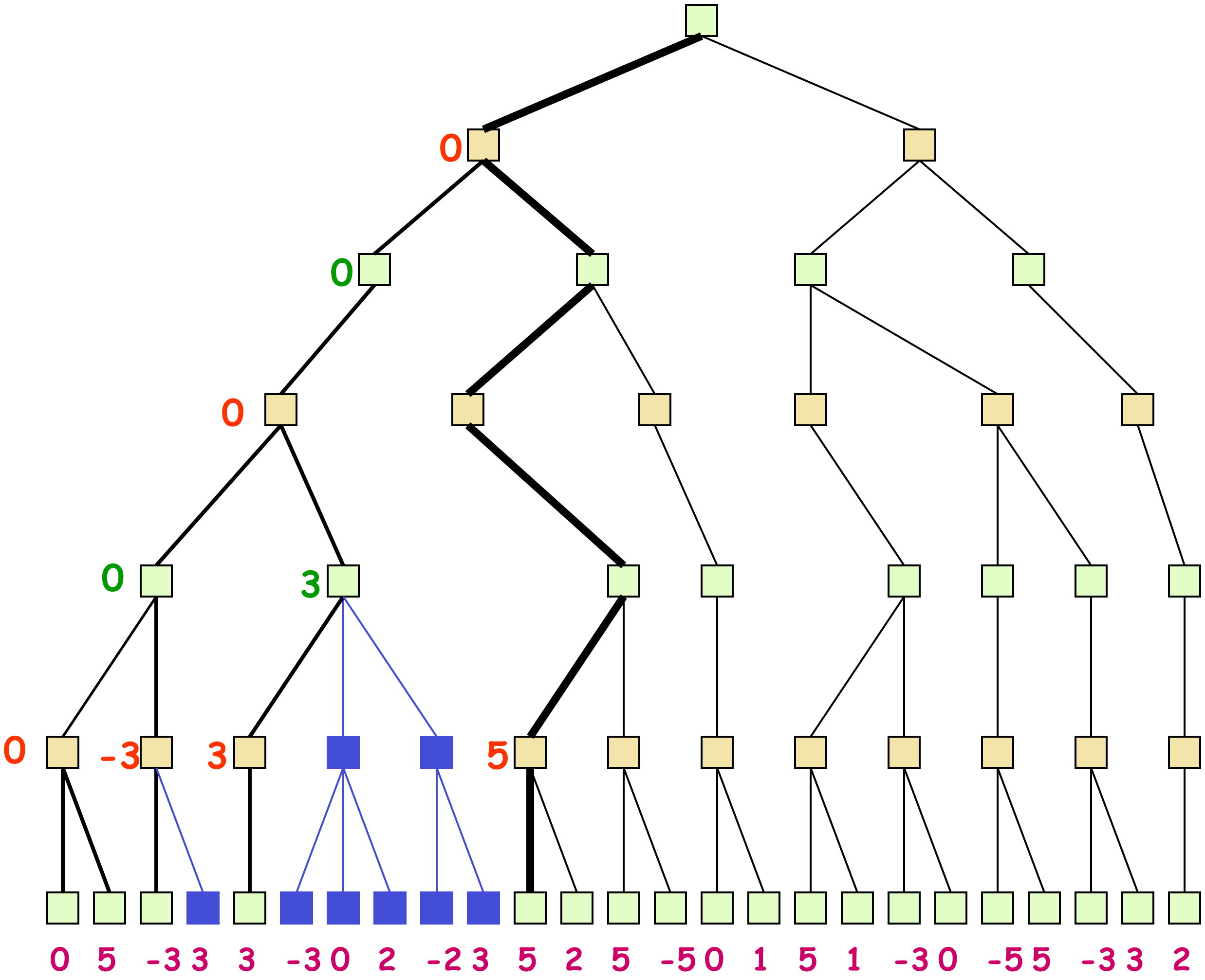

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

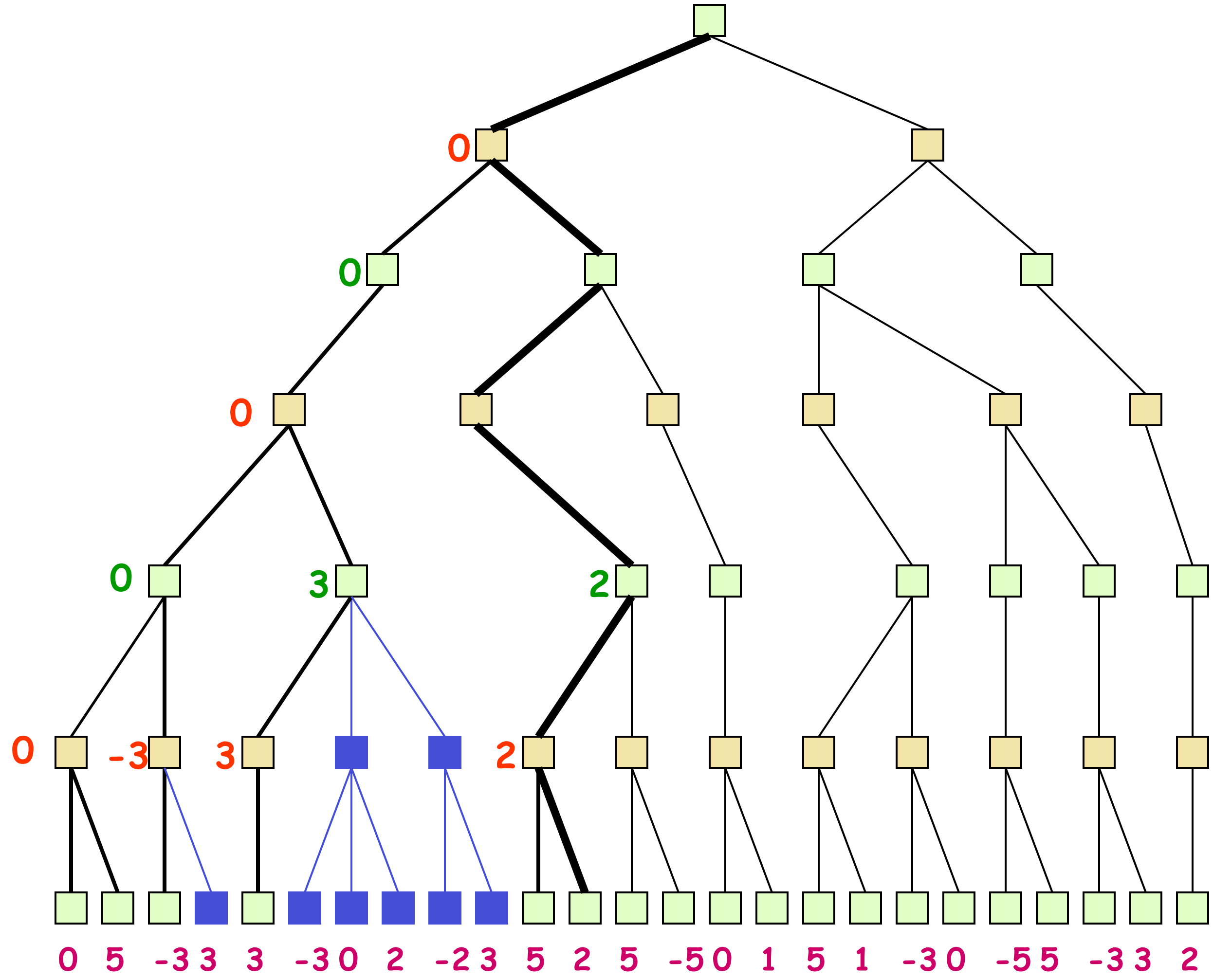

0
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

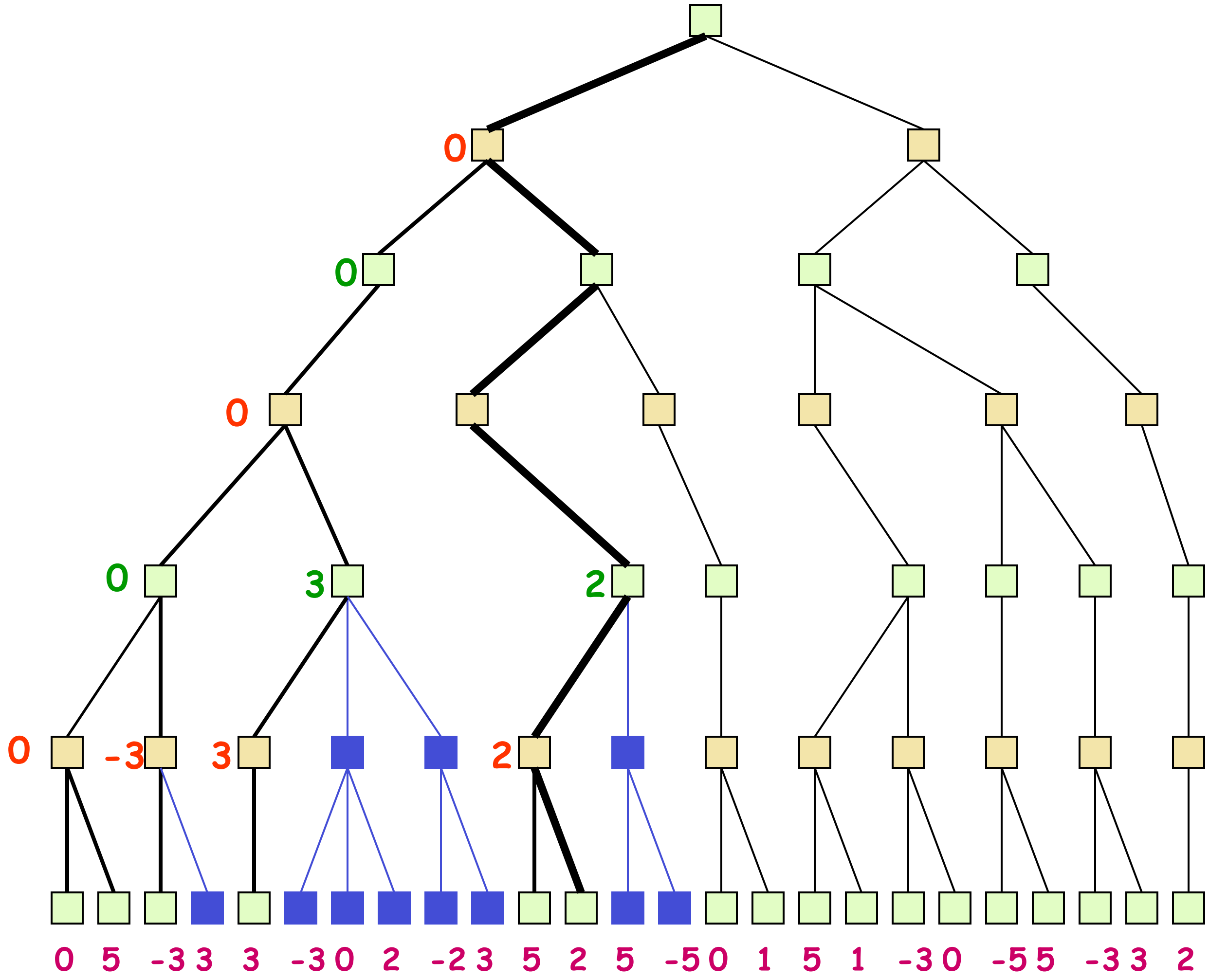

0
0
3
0
-3
3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

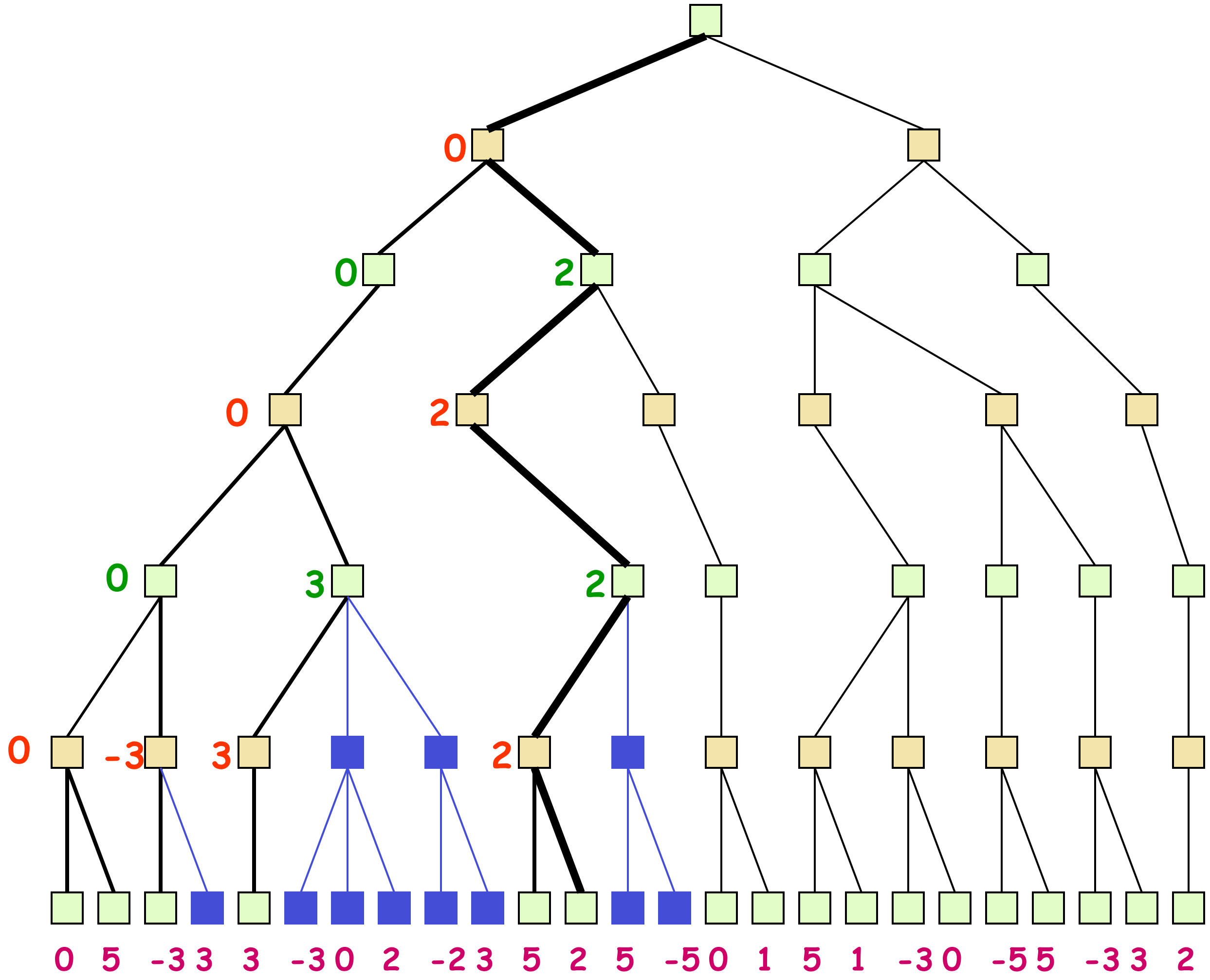

0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

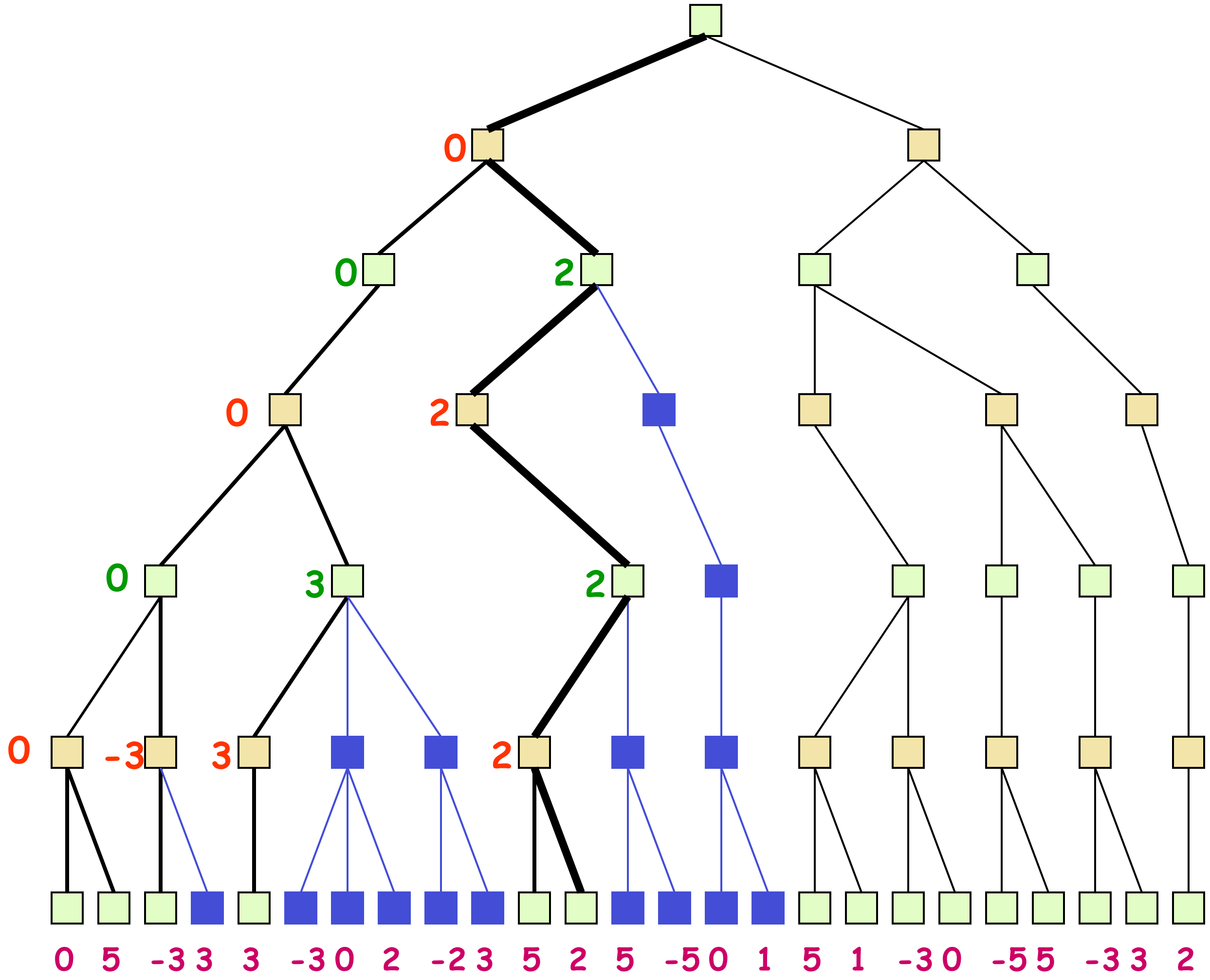

0
0
0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

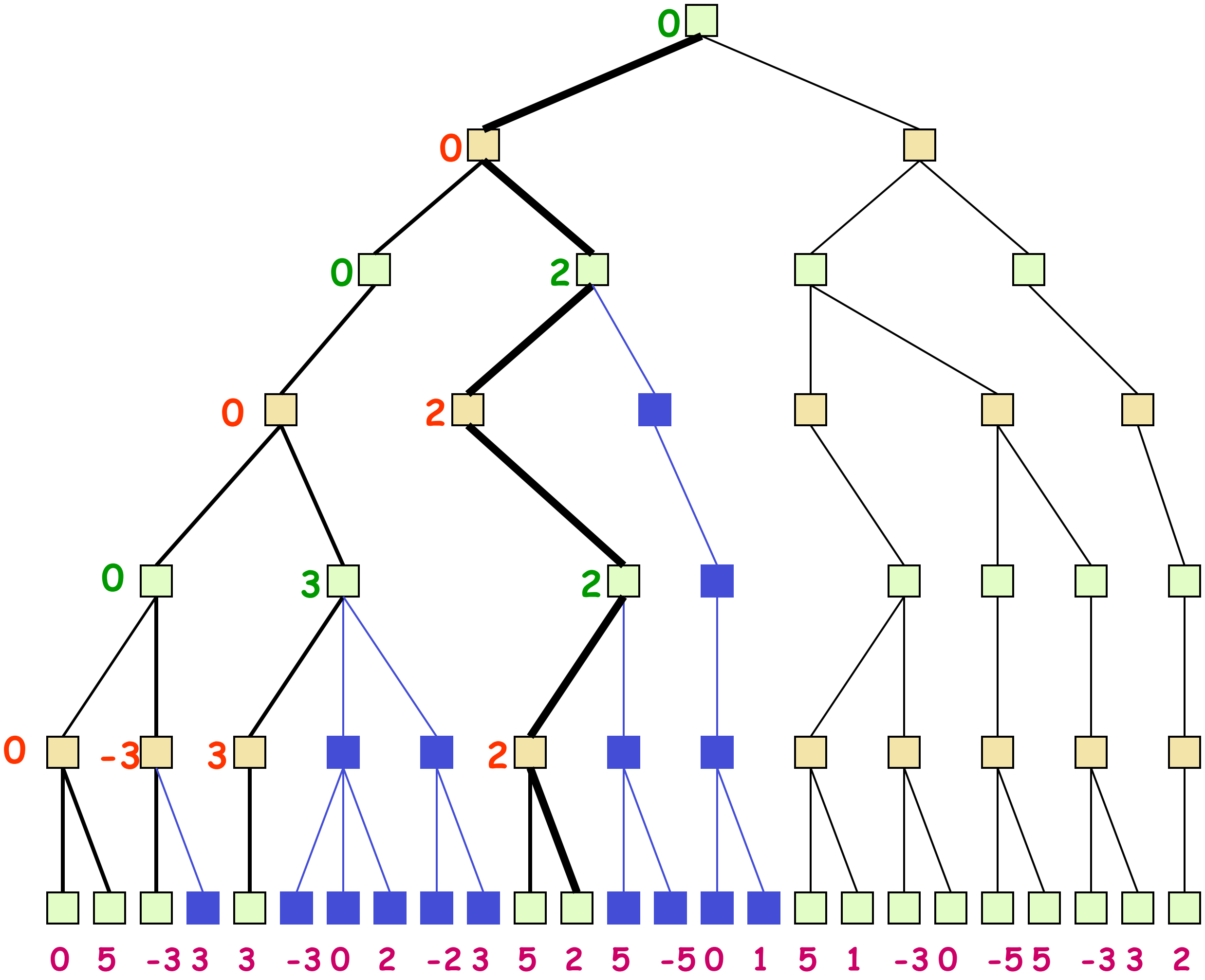

0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

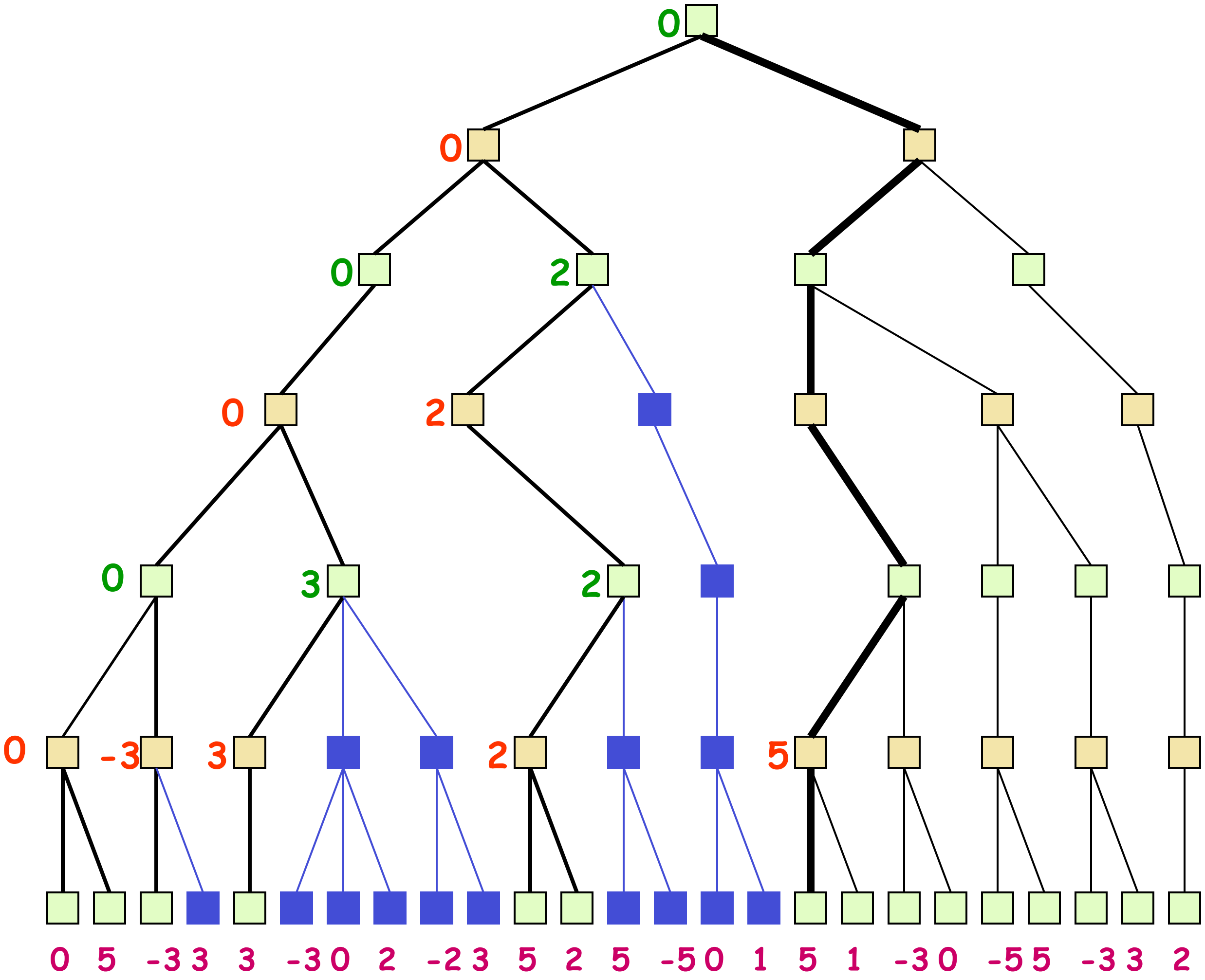

0
0
0
0
3
2
0
-3
3
2
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

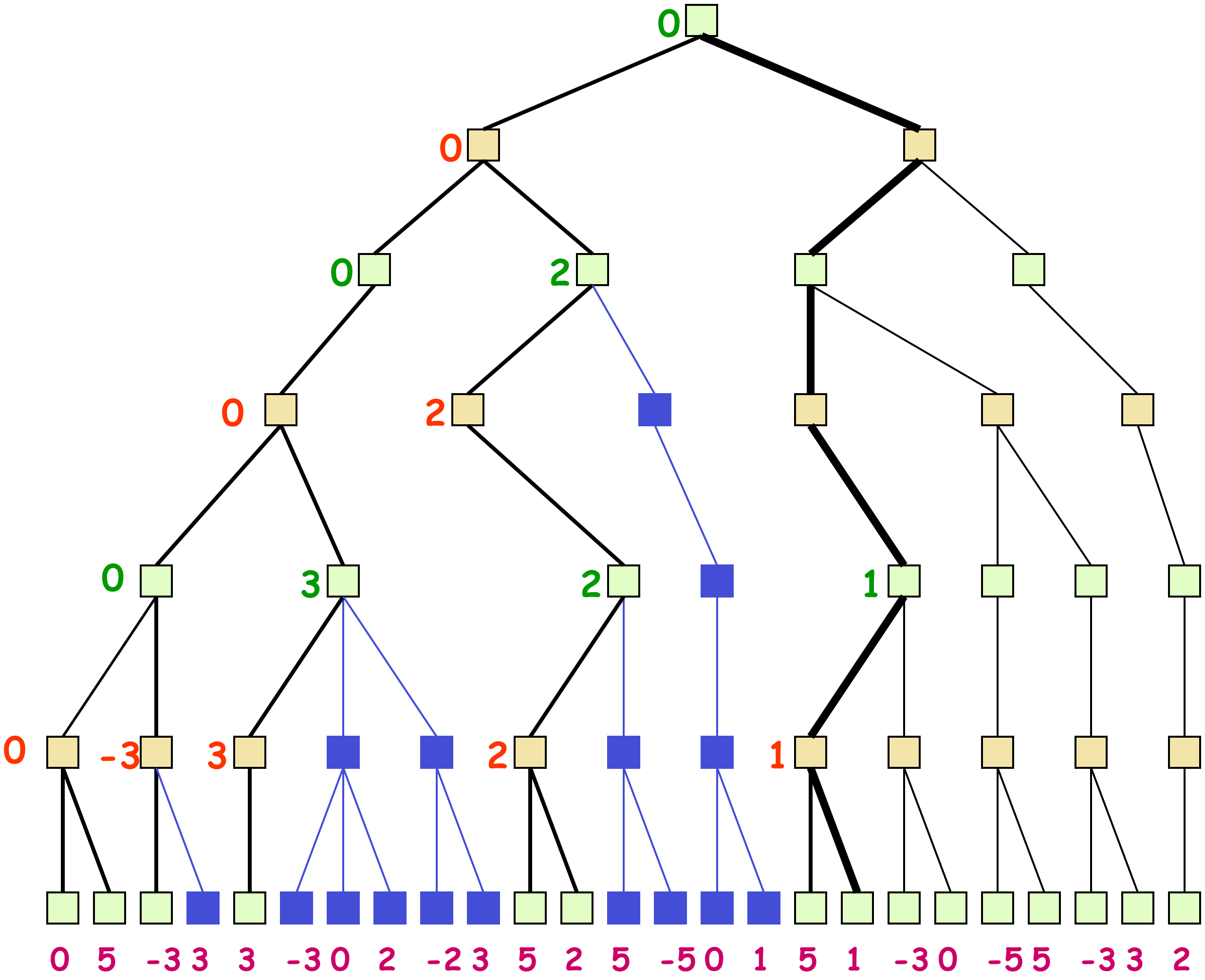

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

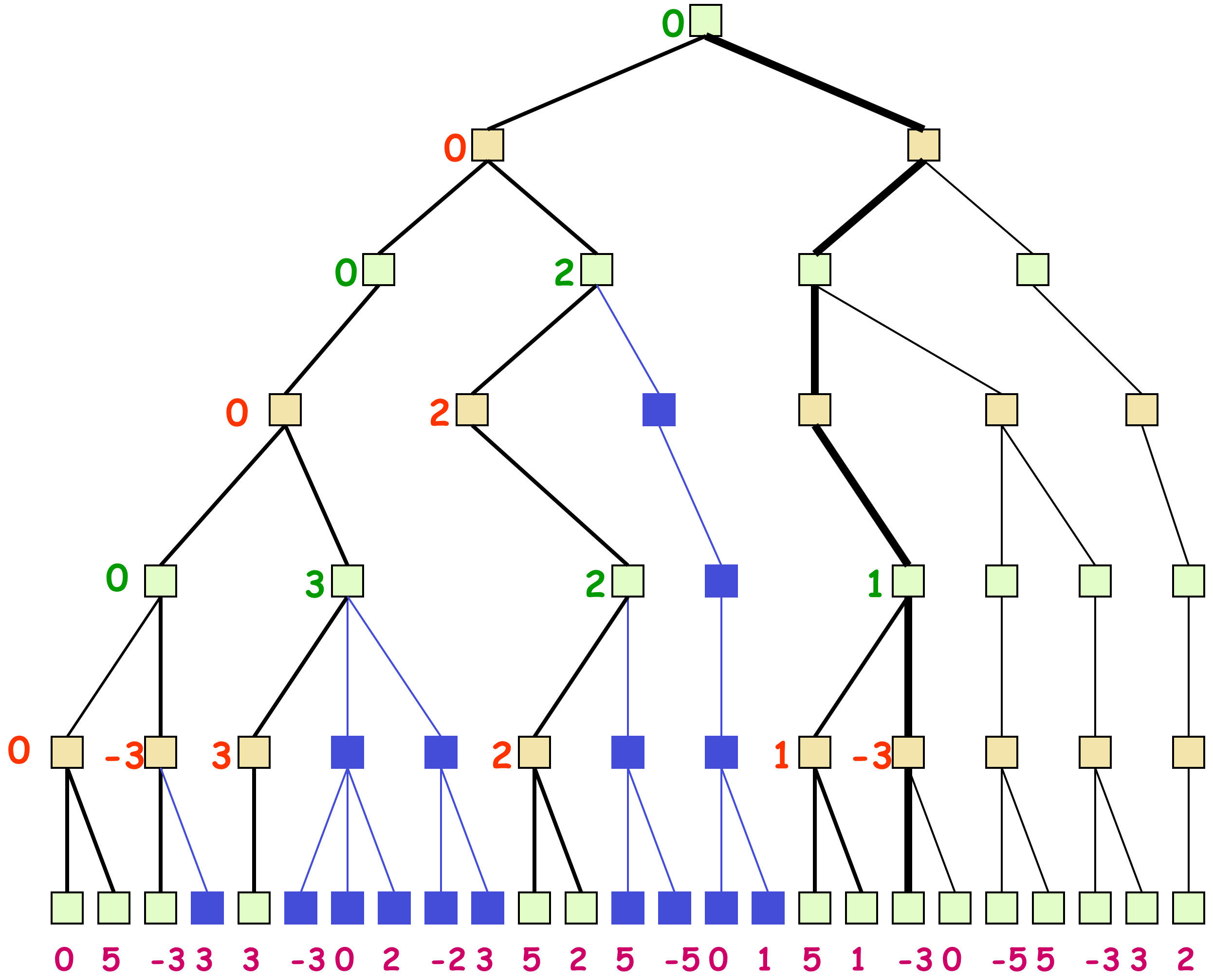

0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

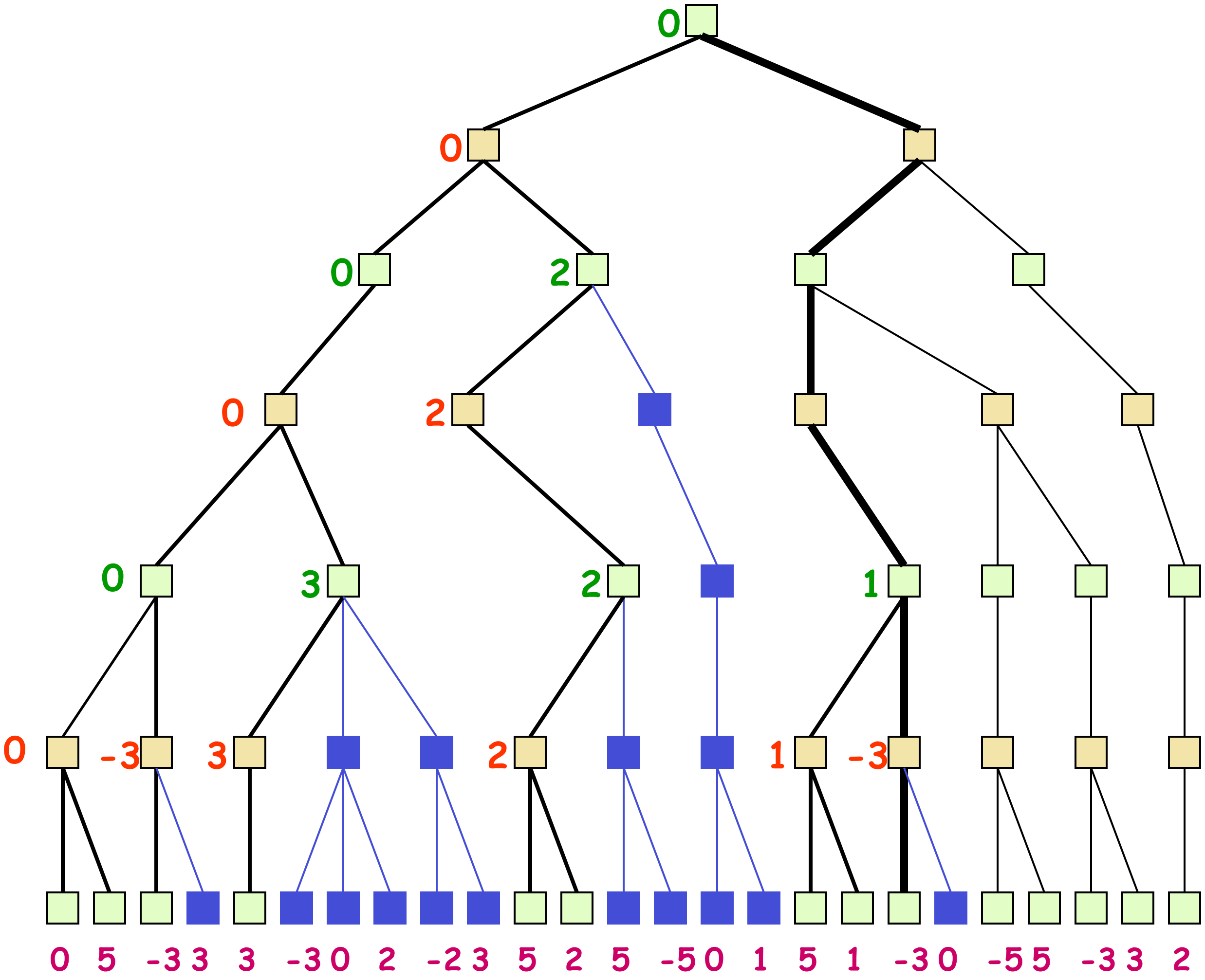

0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

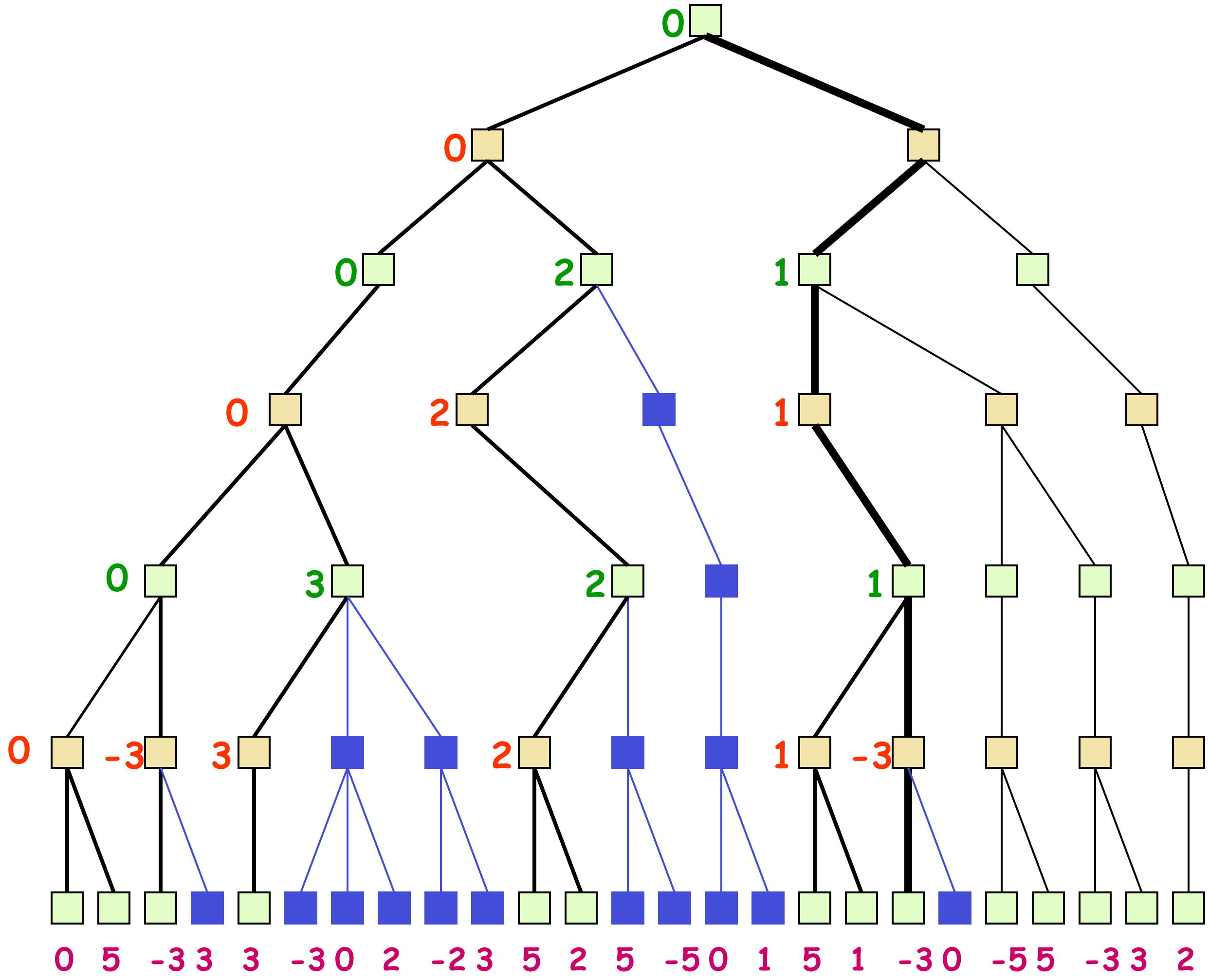

0
0
0
2
0
2
0
3
2
0
-3
3
2
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

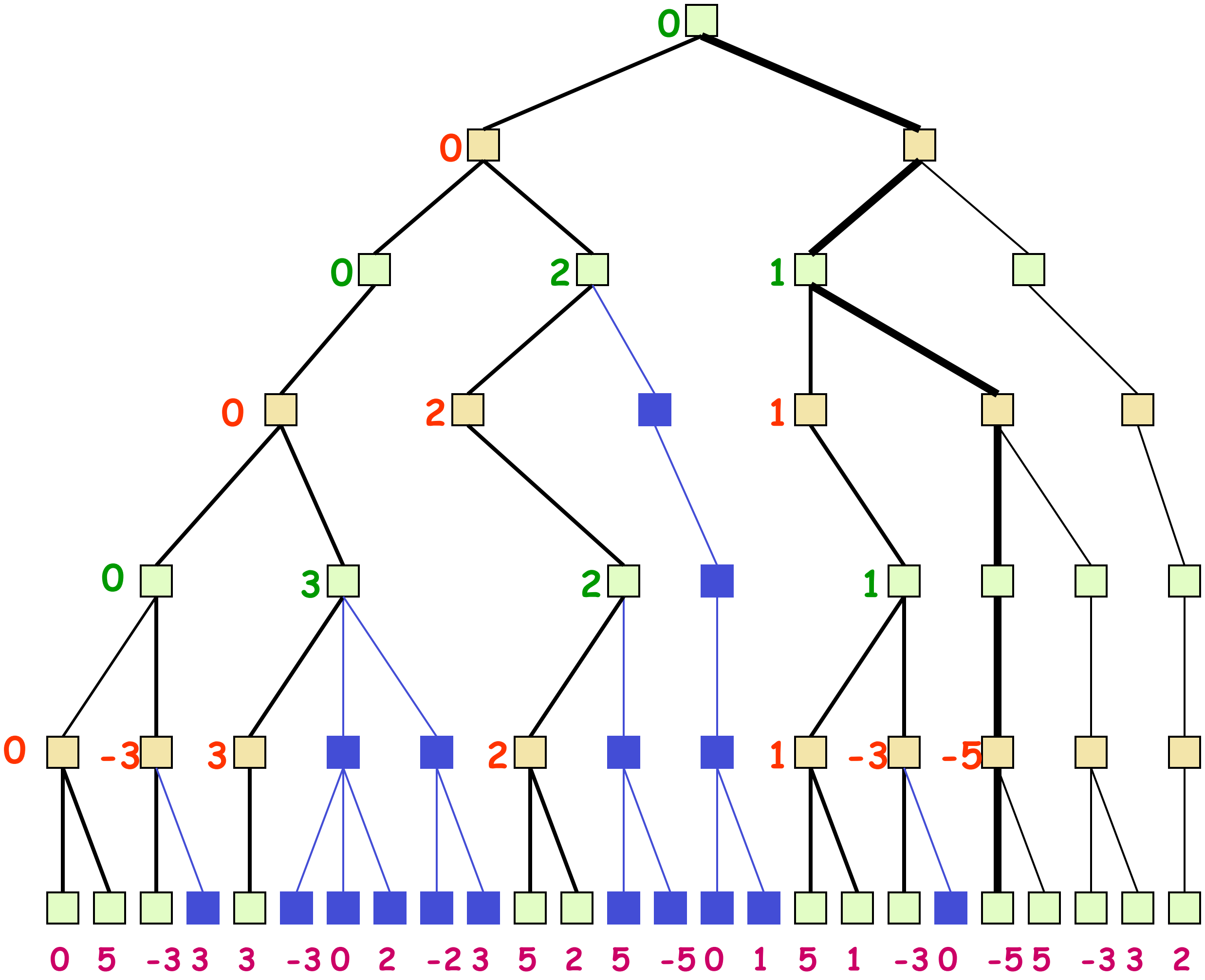

0
0
0
2
0
2
0
3
2
0
-3
3
2
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

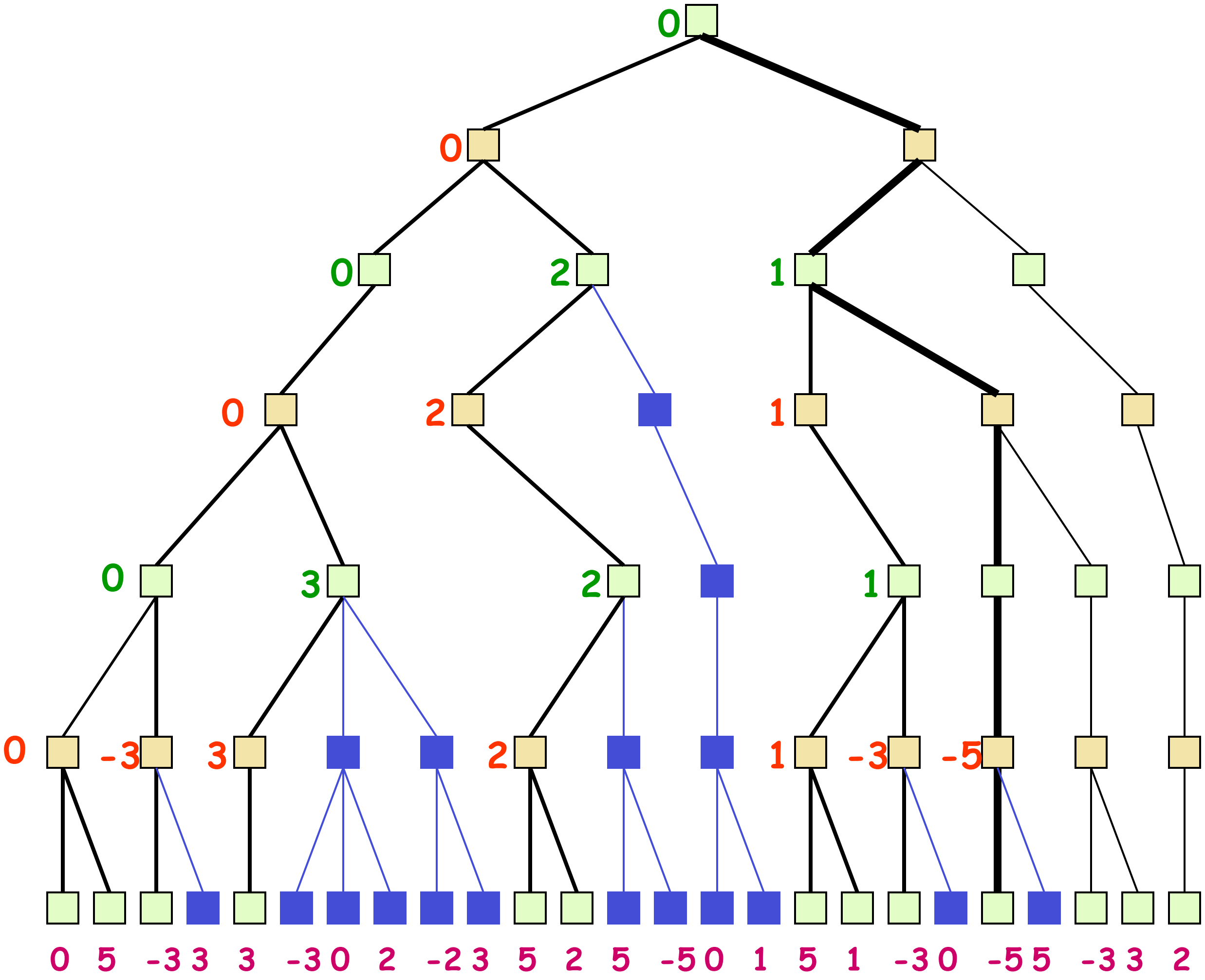

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

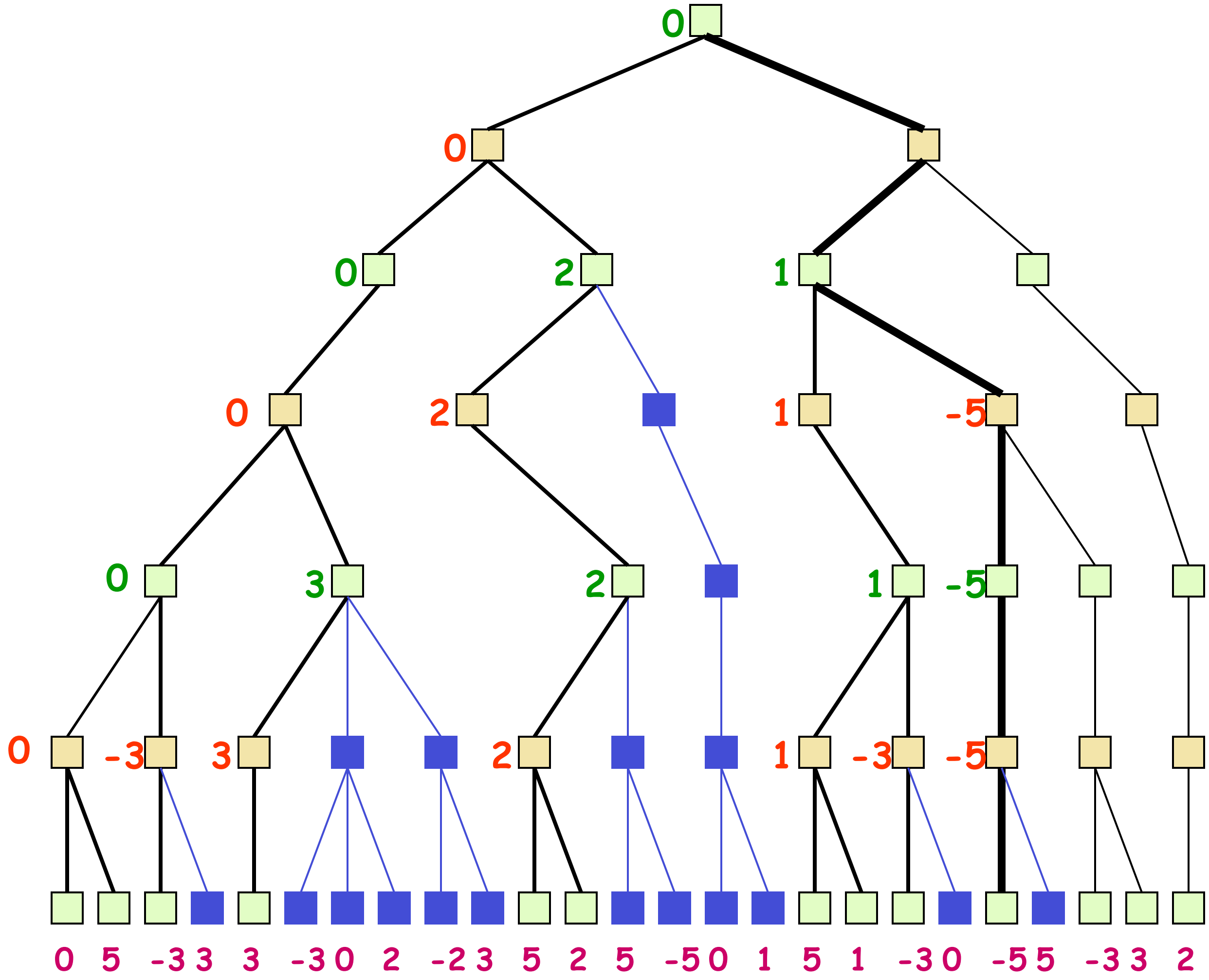

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

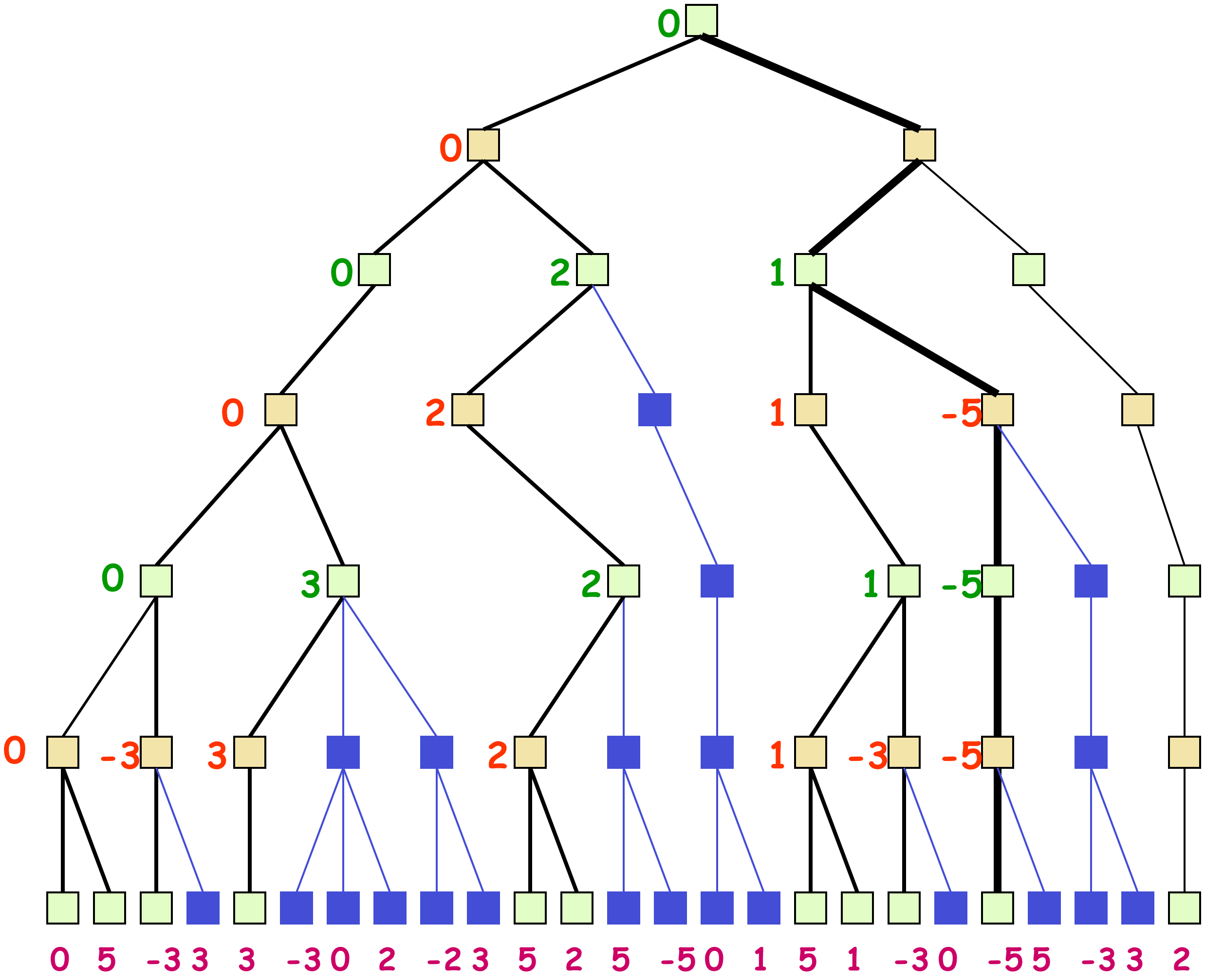

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

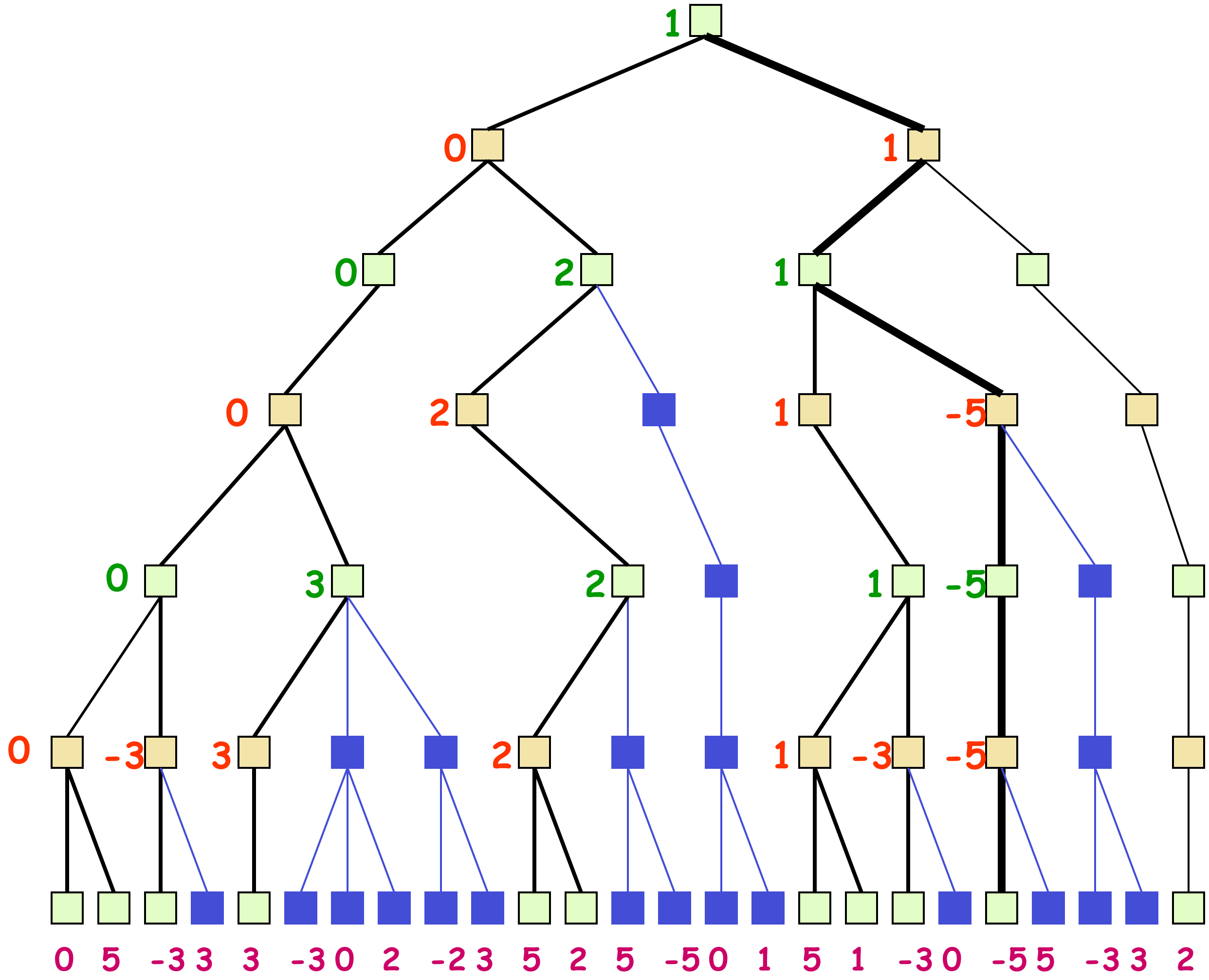

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

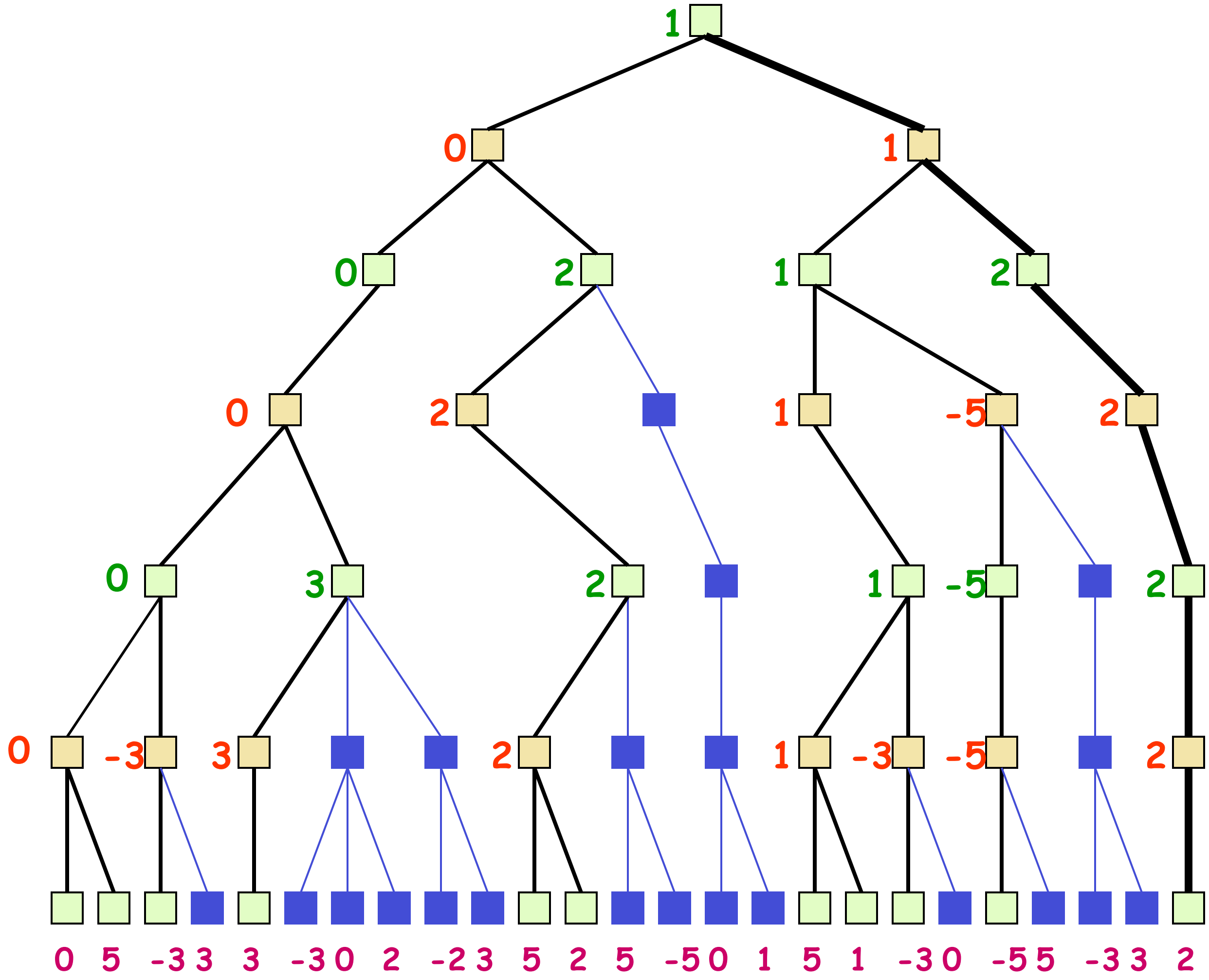

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

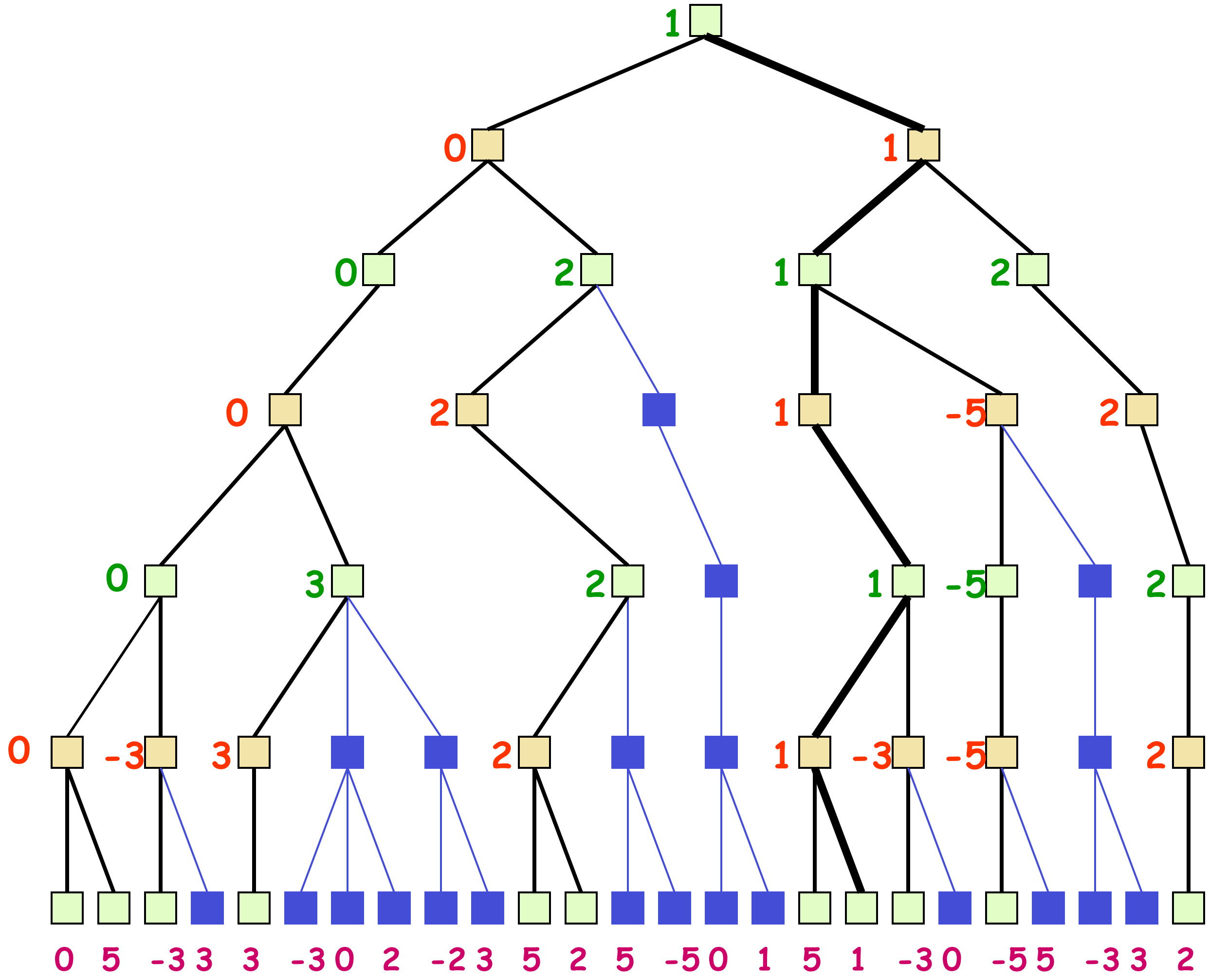

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
-5
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2


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


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

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 ≤ 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 Deep Blue'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 a shallow analysis of tree; result must surely be worse than if player had moved. Can be used to recognize moves that should be explored fully.