# CMSC 471: Games

KMA Solaiman

ksolaima@purdue.edu

Slides courtesy Tim Finin and Frank Ferarro. Some material adopted from notes by Andreas Geyer-Schulz and Chuck Dyer,
Some materials adopted from slides by Dan Klein and Pieter Abbeel at UC Berkeley [http://ai.berkeley.edu]

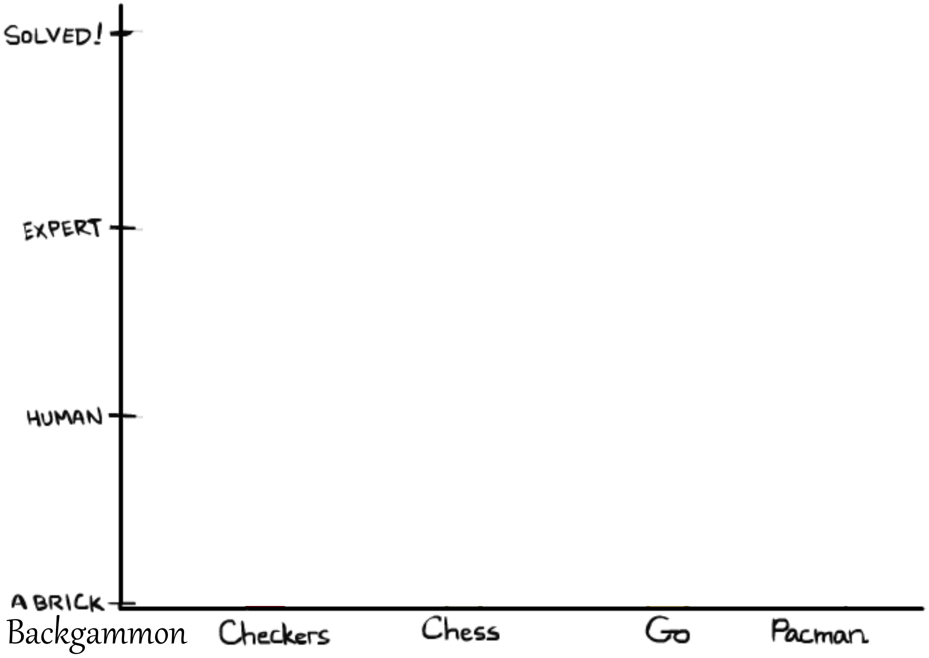
# Overview

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

# Why study games?

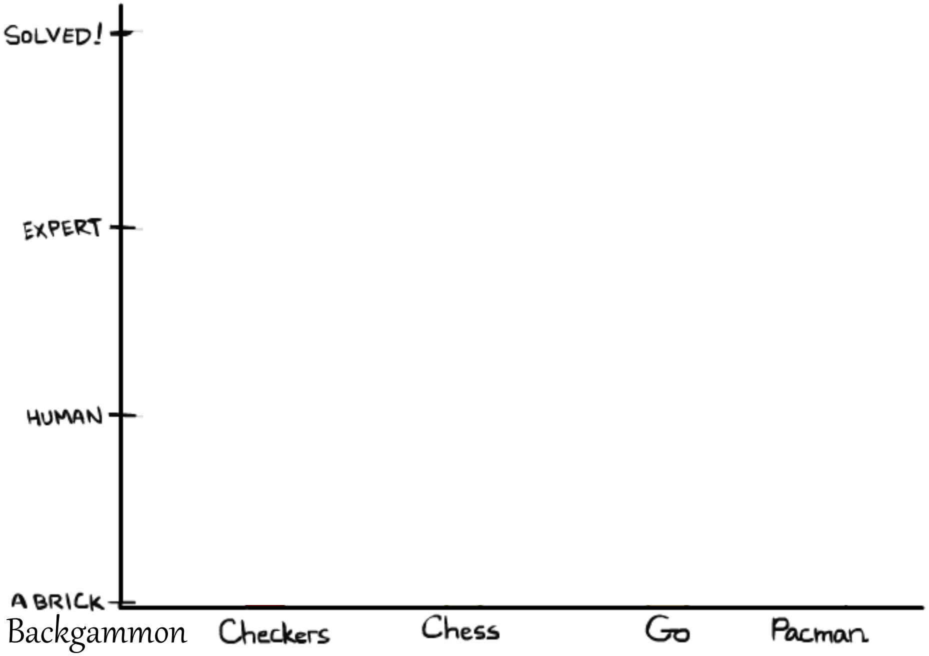
- Interesting, hard problems requiring minimal "initial structure"
- Clear criteria for success
- Study problems involving {hostile, adversarial, competing} agents and uncertainty of interacting with the natural world
- People have used them to assess their intelligence
- Fun, good, easy to understand,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

# Game Playing State-of-the-Art



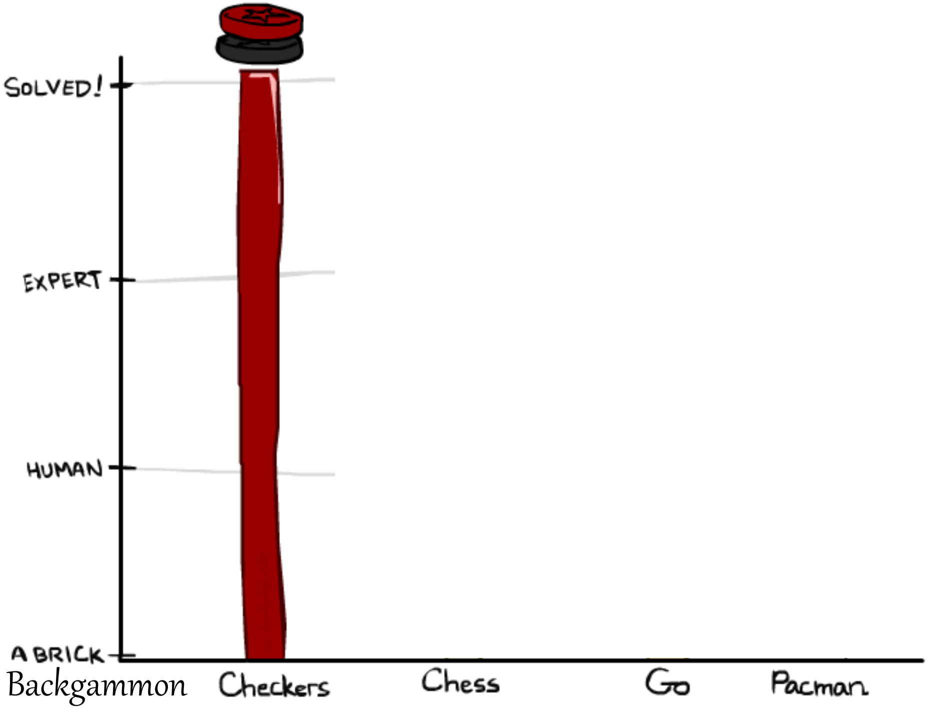
# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!



# Game Playing State-of-the-Art

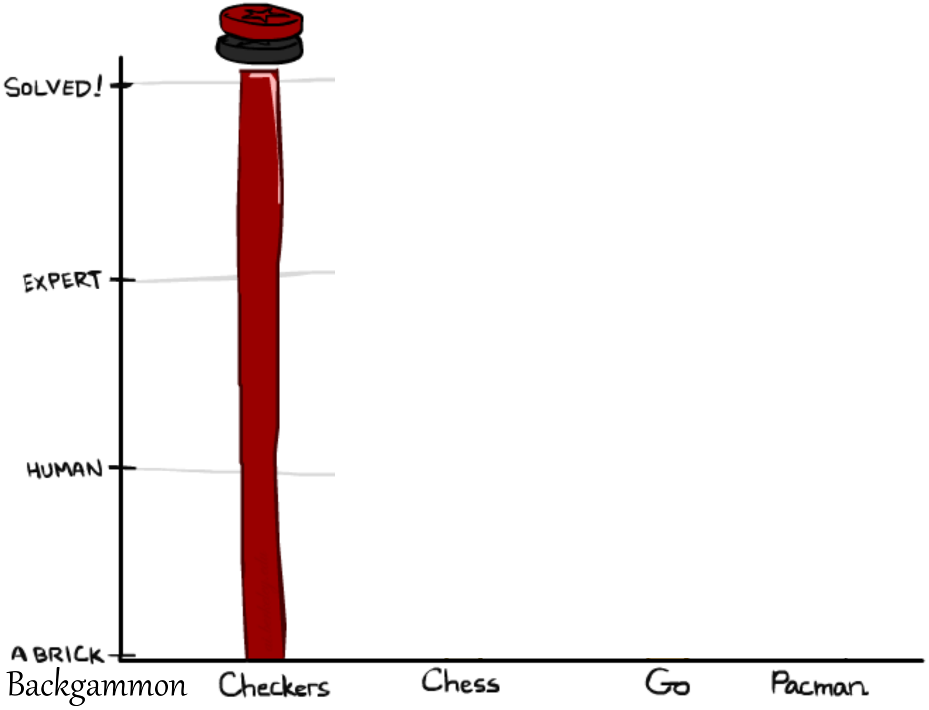
- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!



# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

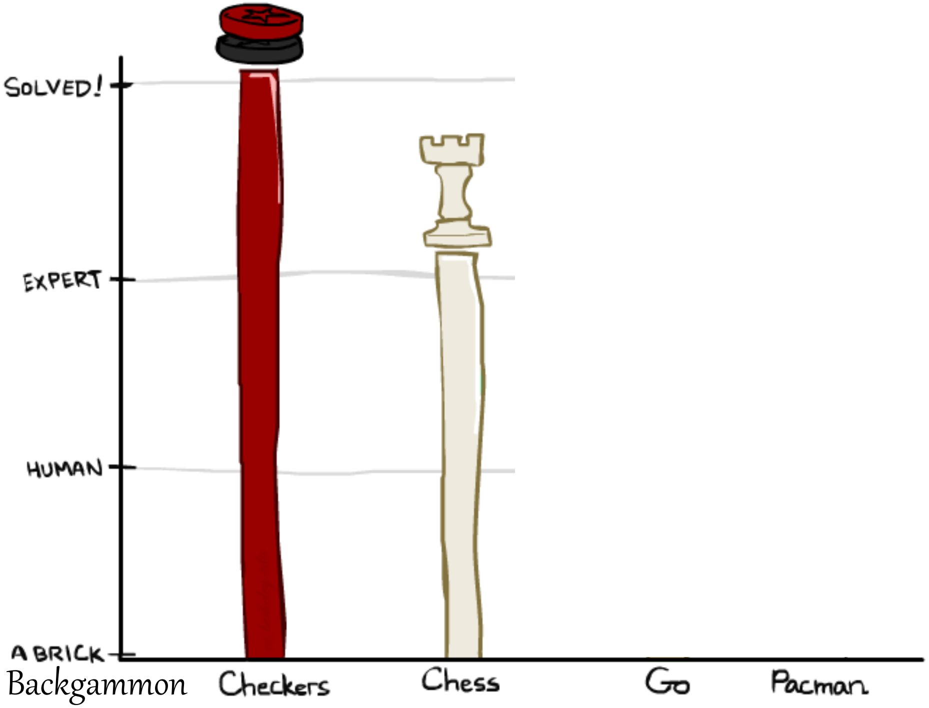
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.



# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

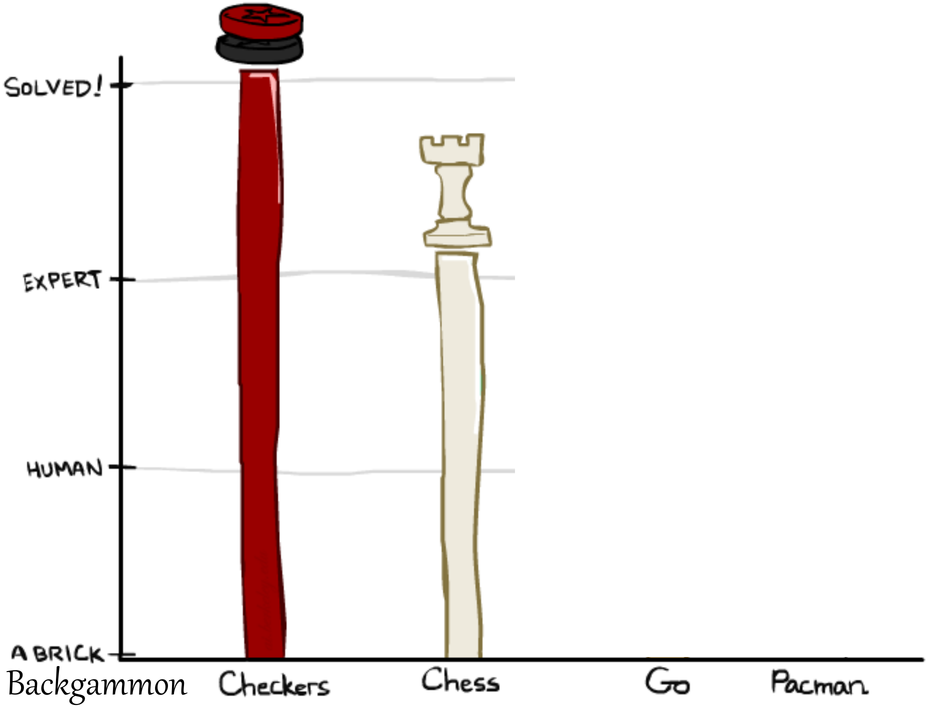


# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

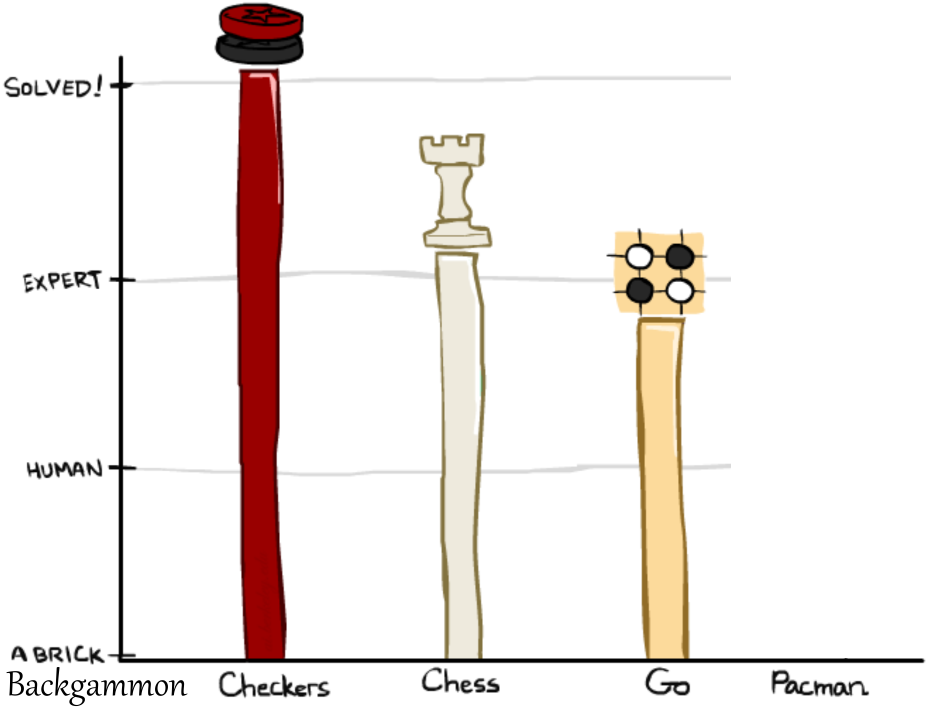


# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

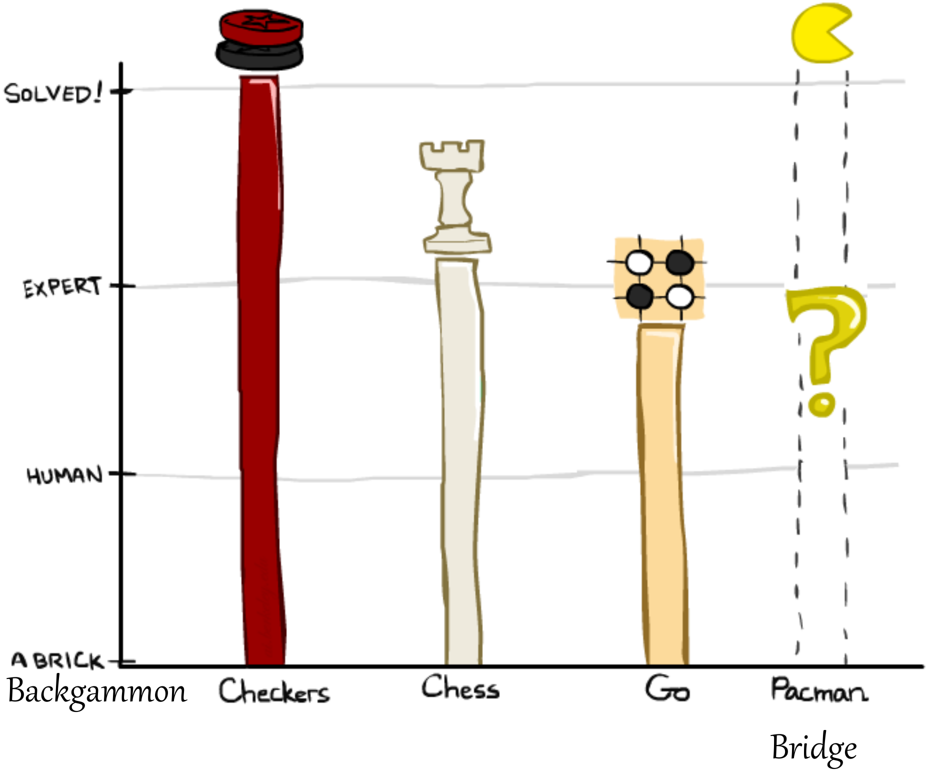
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.



# Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Go:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.



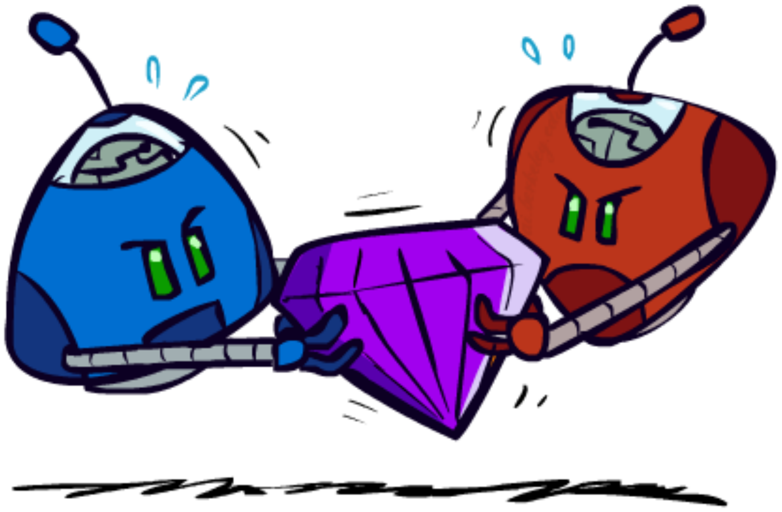
# Classical vs. Statistical/Neural Approaches

- 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

# Adversarial Games



# Types of Games

- **1-person, 2-person** game, with **alternating moves, or more players**
- **Zero-sum**: one player's loss is the other's gain **/ not**
- **Perfect information**: both players have access to complete information about state of game.  No information hidden from either player.
- **Chance** (e.g., using dice) **vs No chance** involved

# Types of Games

- **2-person** game, with **alternating moves**
- **Zero-sum**: one player's loss is the other's gain
  - A **zero-sum game** is 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 0+1, 1+0, or 1⁄2 + 1⁄2.
- **Perfect information**: both players've access to complete information about state of game (chess, checkers). No information hidden from either player (poker).
- **No chance** involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- But not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

# Can we use …

Want algorithms for calculating **a strategy (policy)** which recommends a move from each state

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

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

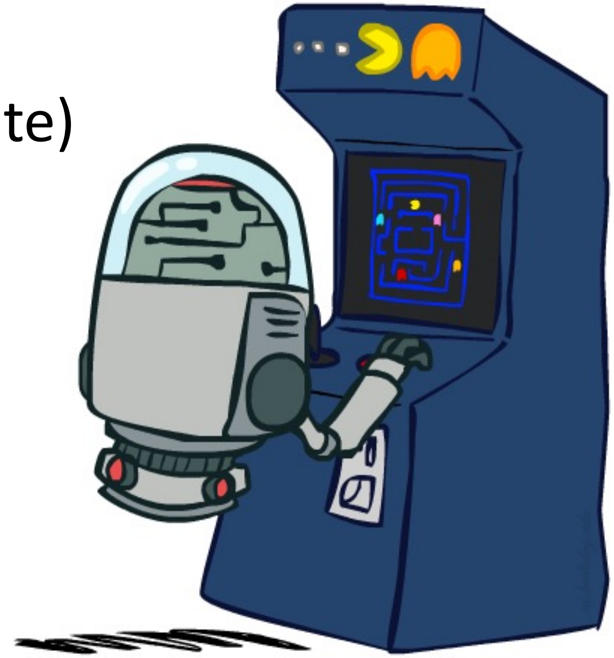
# How to play a game

- 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

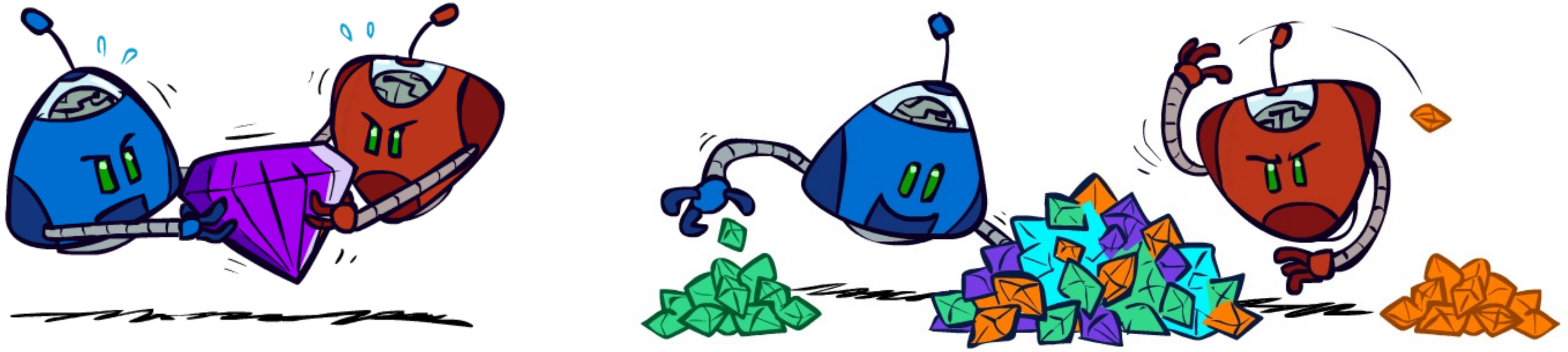
# Deterministic Games

- 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: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

- Solution for a player is a policy: S $\rightarrow$ A



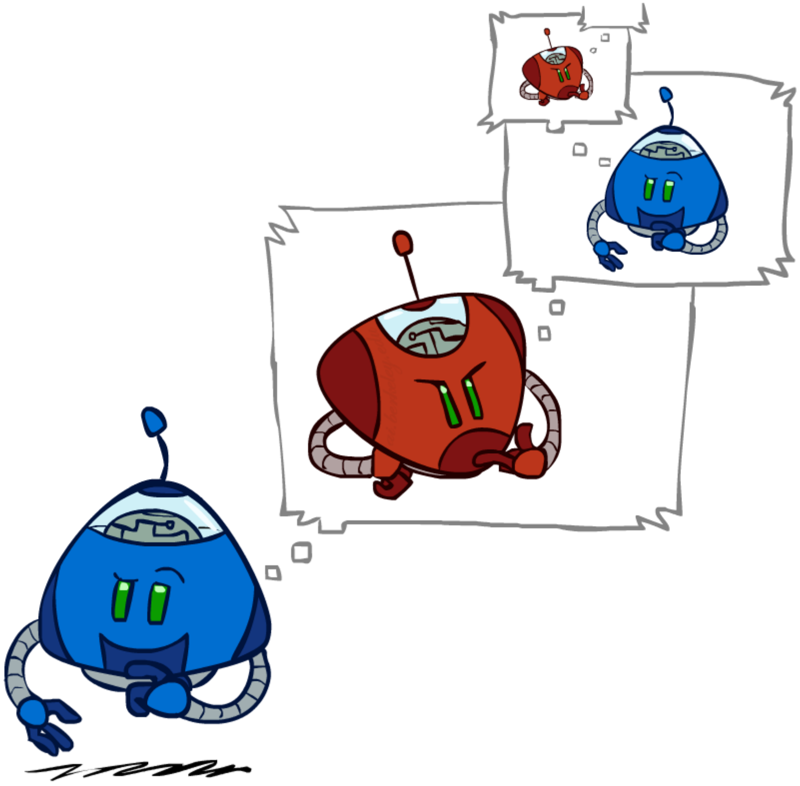
# Zero-Sum Games

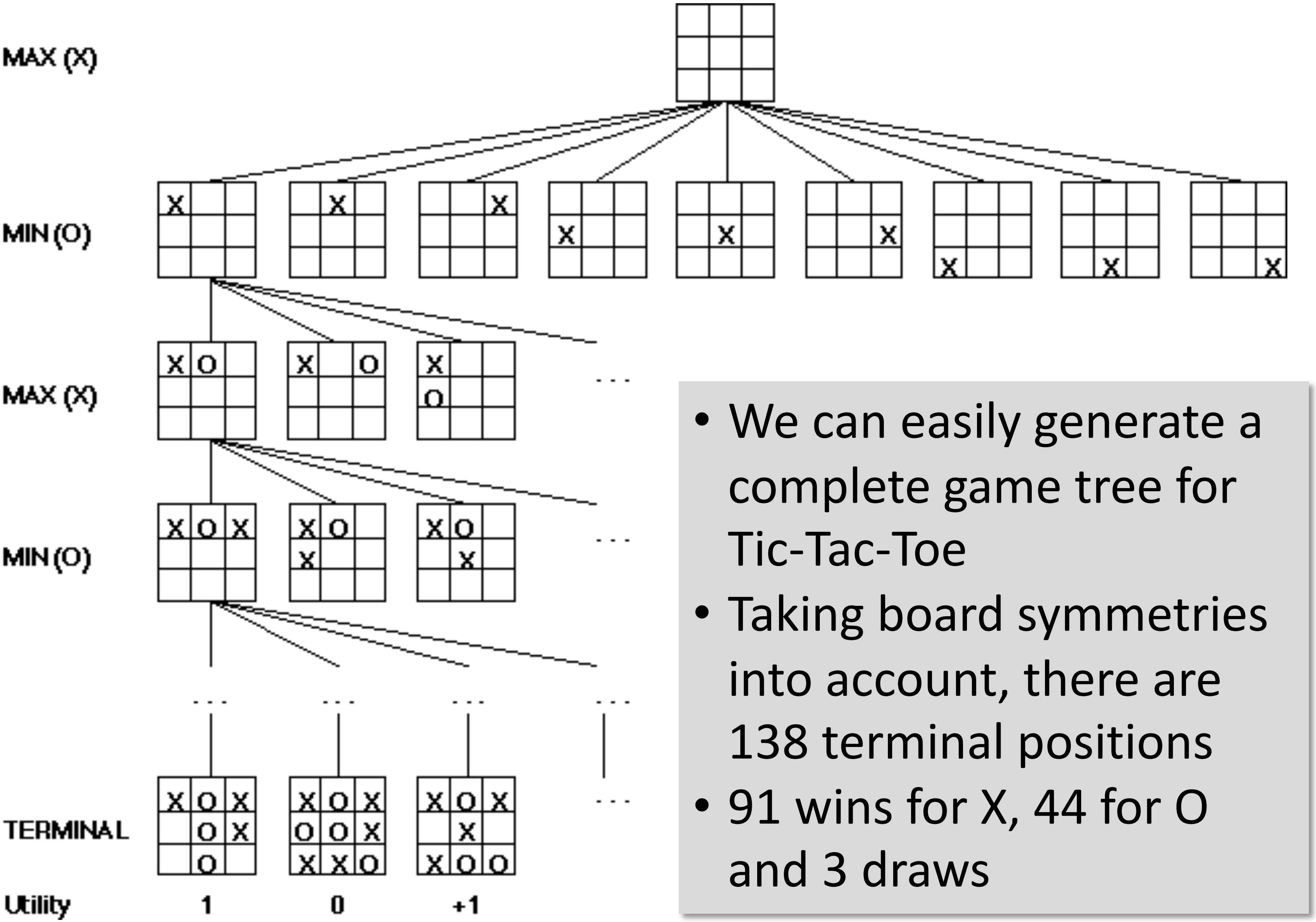


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - **Lets us think of a single value that one maximizes and the other minimizes**
  - Adversarial, pure competition

- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

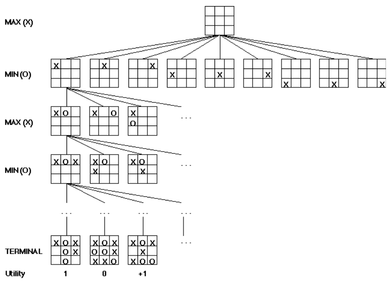
# Adversarial Search





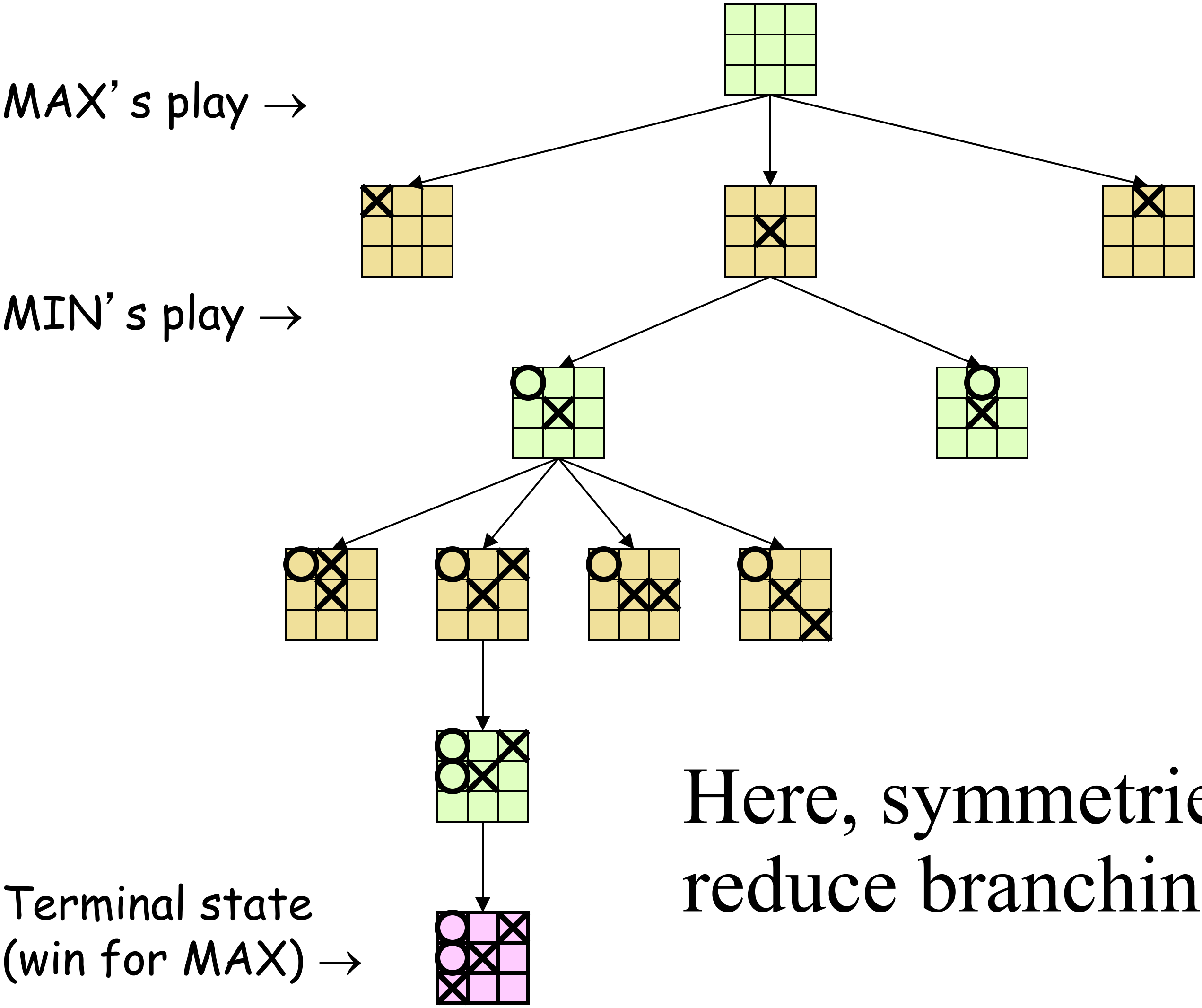
- We can easily generate a complete game tree for Tic-Tac-Toe
- Taking board symmetries into account, there are 138 terminal positions
- 91 wins for X, 44 for O and 3 draws

# Game trees



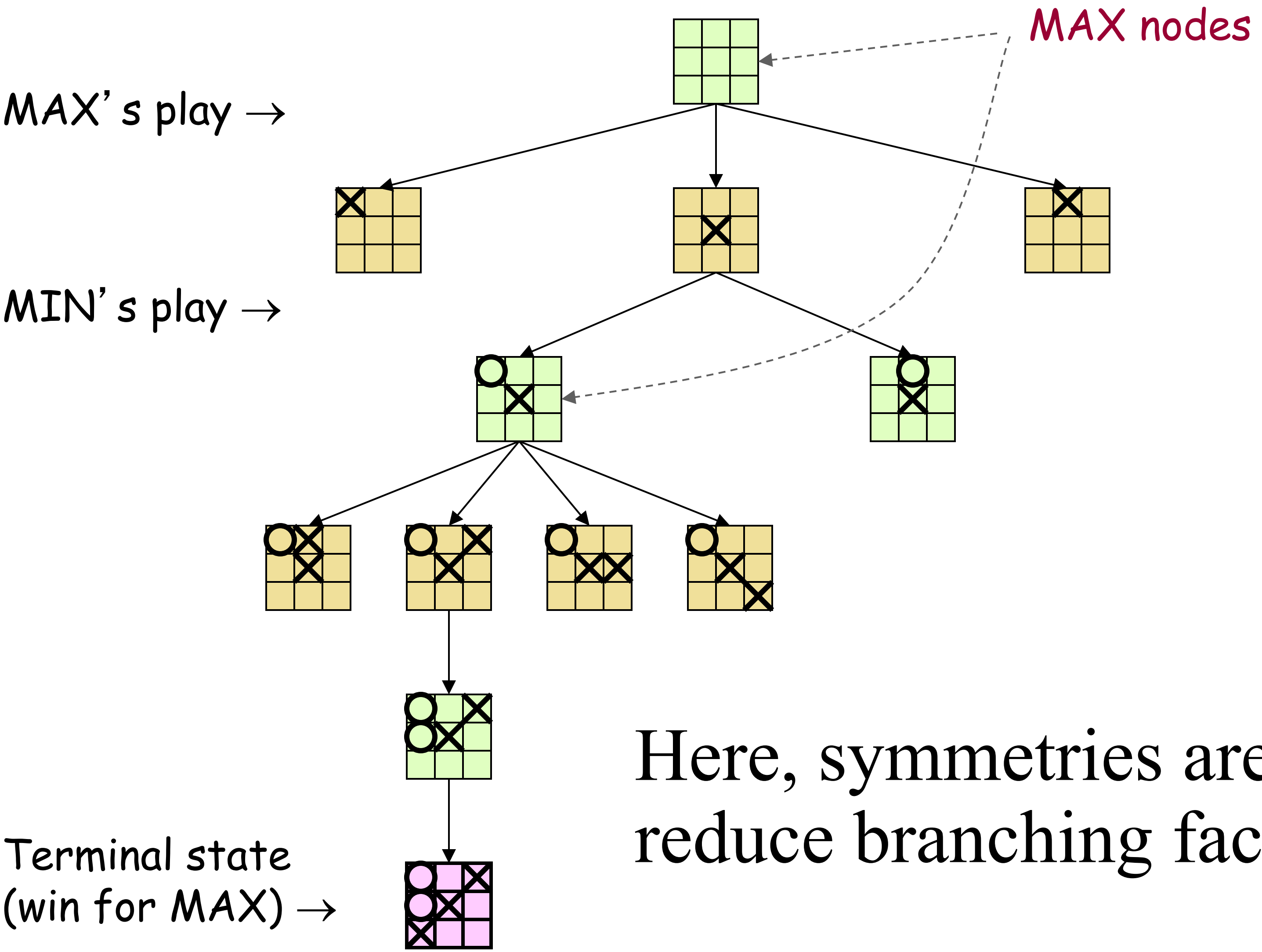
- Problem spaces for typical games are trees
- Root node is current board configuration; player must decide best single move to make next
- **Static evaluator function** rates board position **f(board):**real, > 0 for me; < 0 for opponent
- Arcs represent possible legal moves for a player
- If **my turn** 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



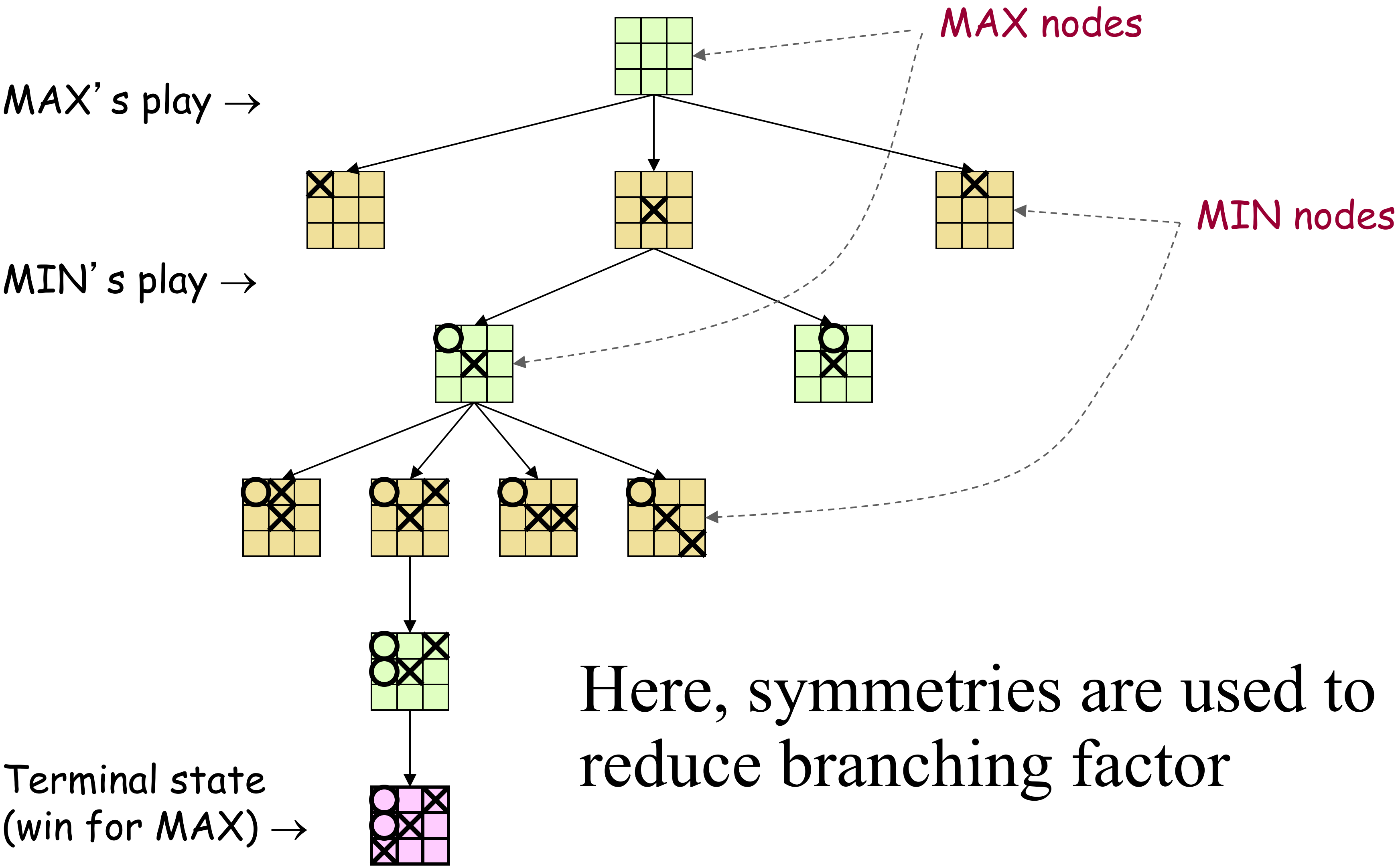
Here, symmetries are used to reduce branching factor

# Game Tree for Tic-Tac-Toe



Here, symmetries are used to reduce branching factor

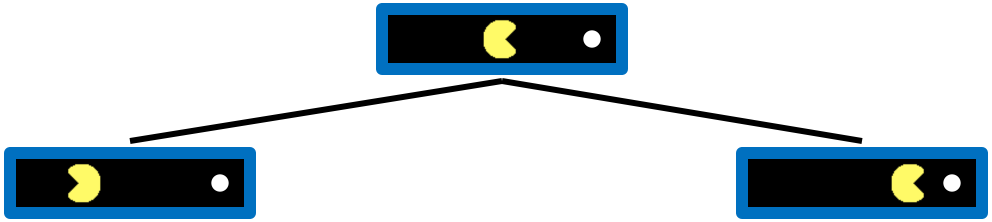
# Game Tree for Tic-Tac-Toe



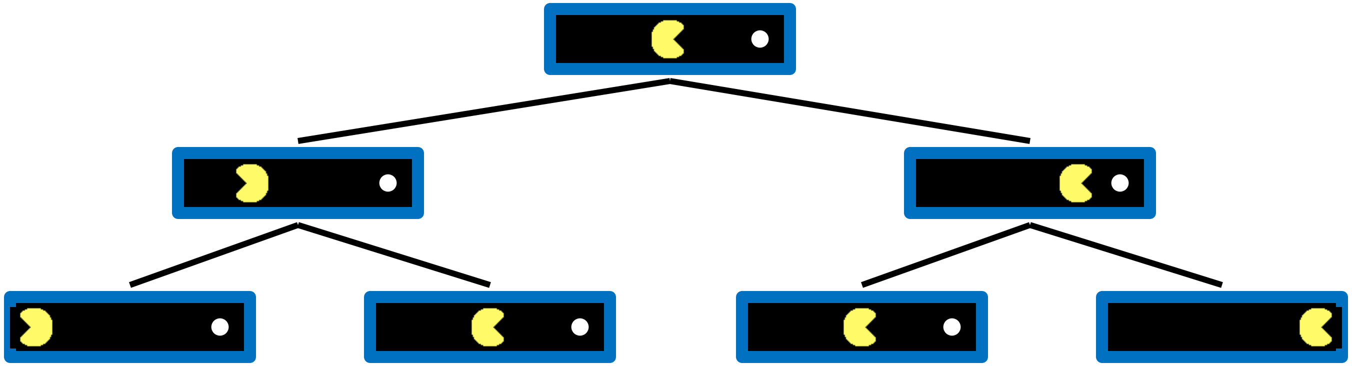
Here, symmetries are used to reduce branching factor

# Single-Agent Trees

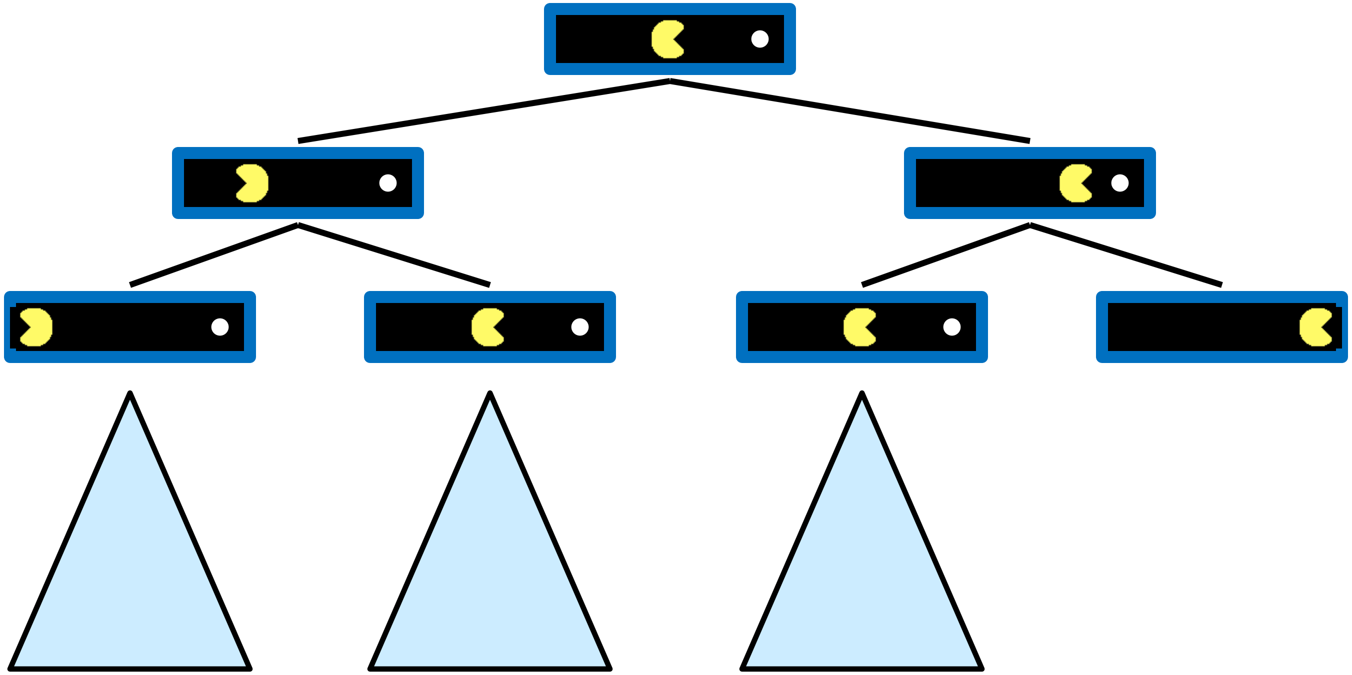
# Single-Agent Trees



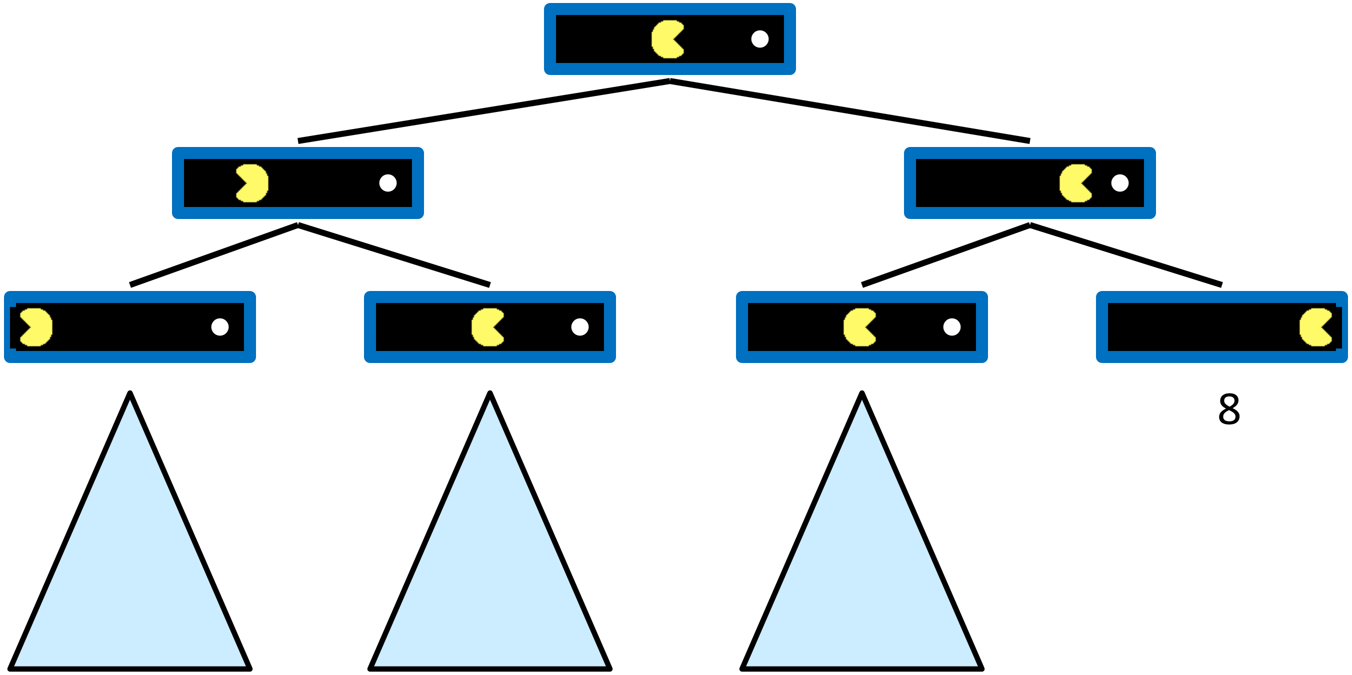
# Single-Agent Trees



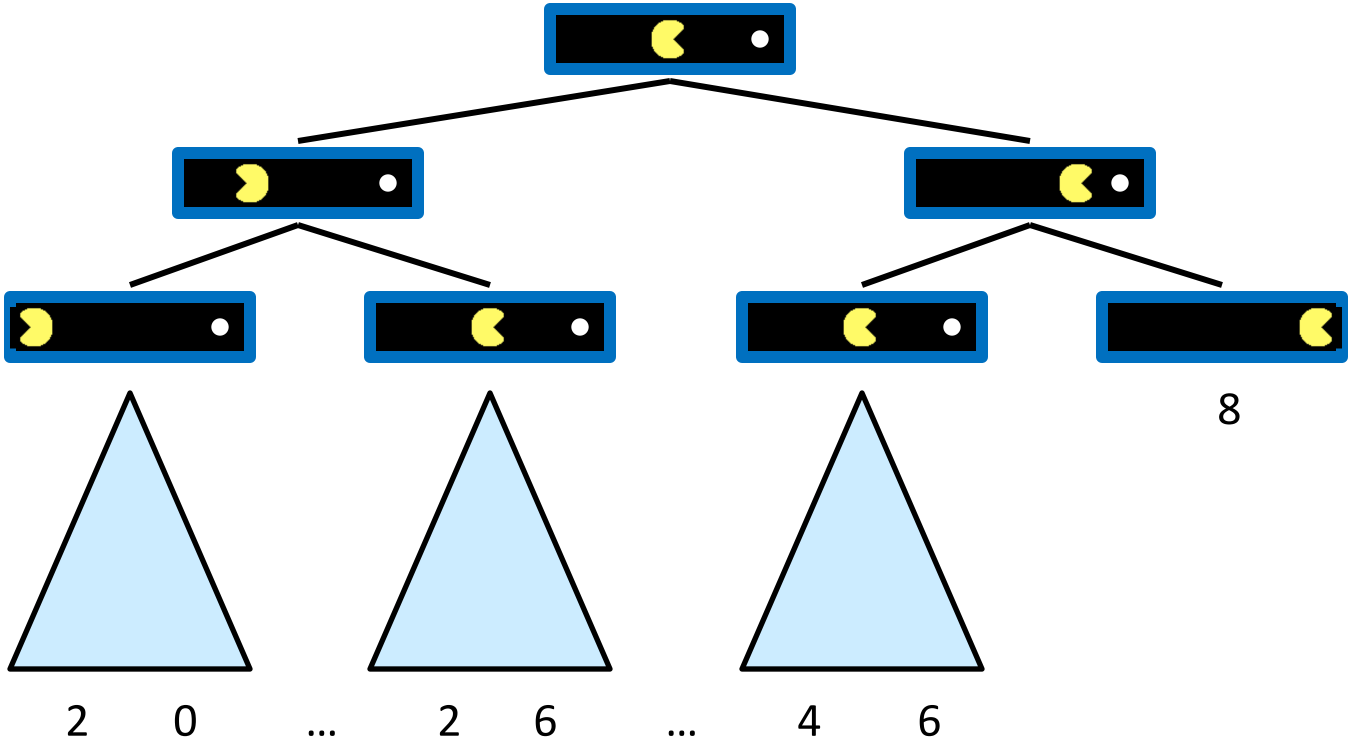
# Single-Agent Trees



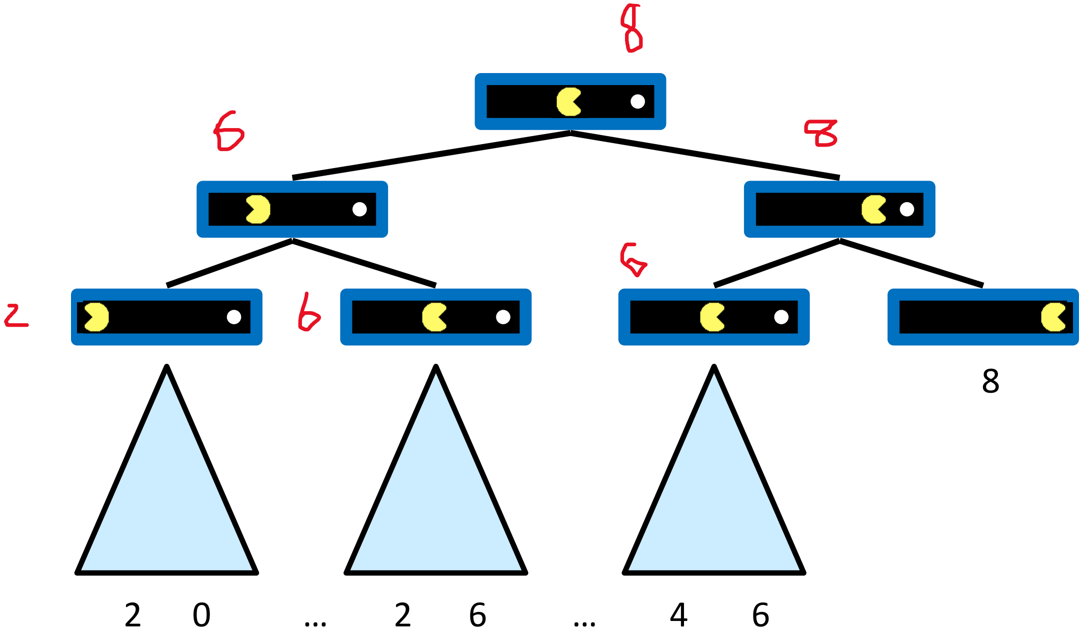
# Single-Agent Trees



# Single-Agent Trees

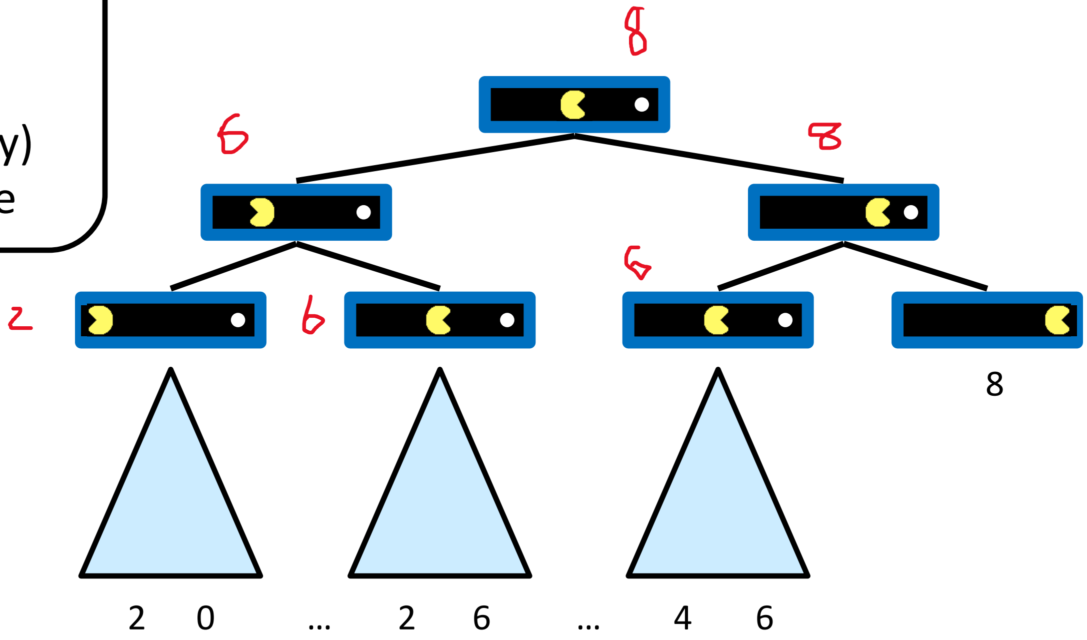


# Value of a State



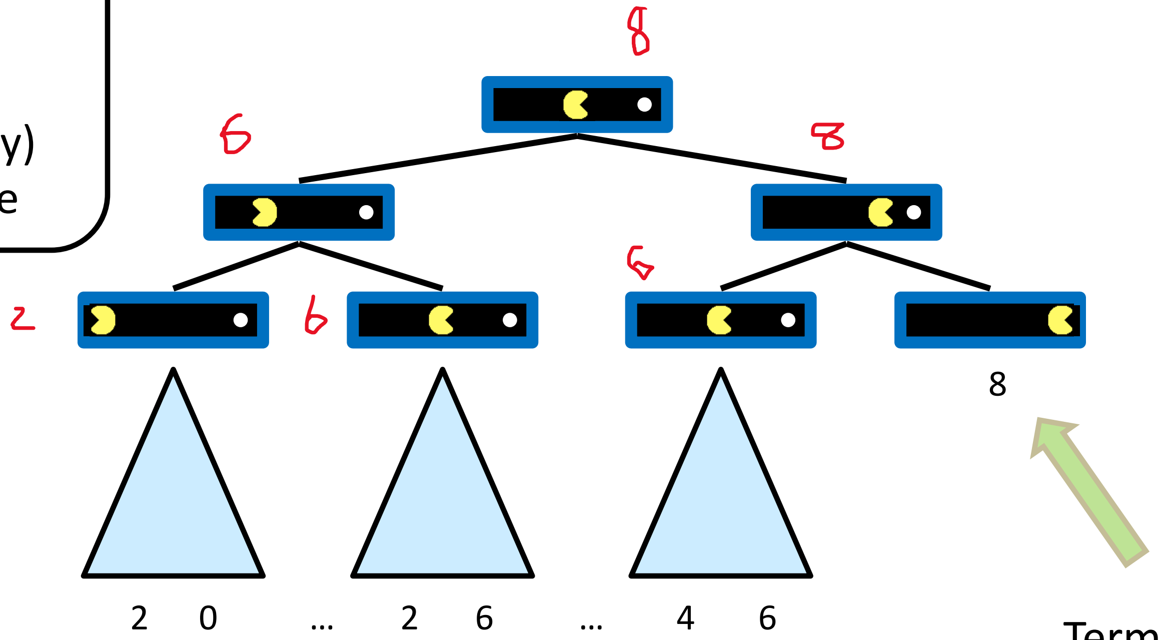
# Value of a State

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



# Value of a State

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

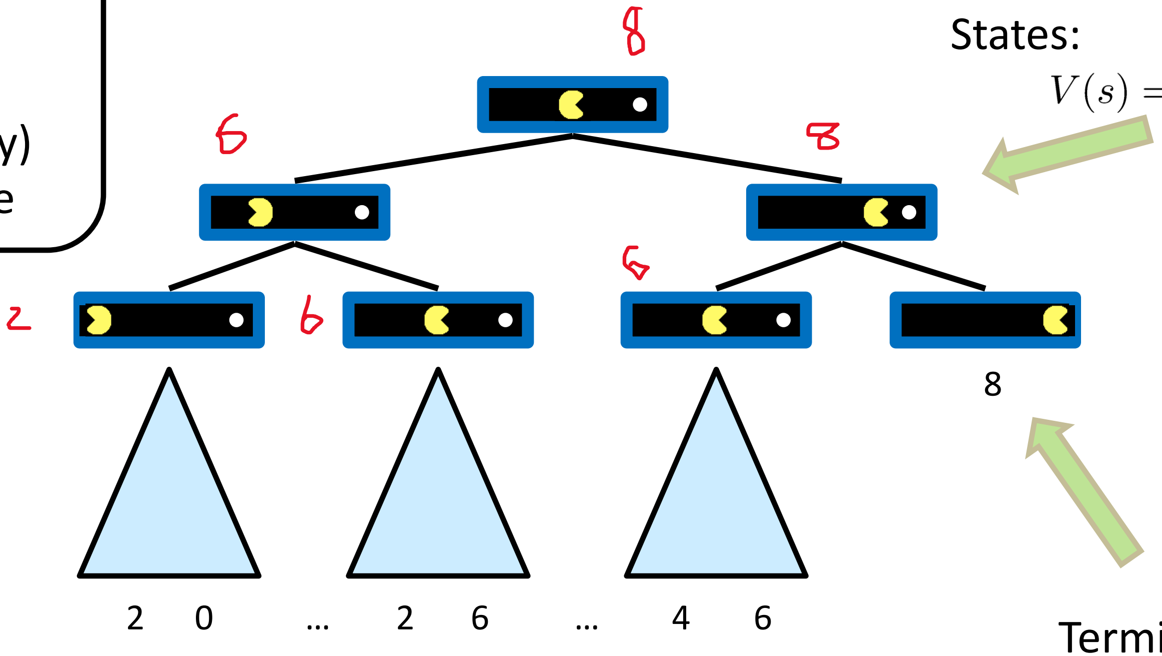


Terminal States:

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

# Value of a State

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



Non-Terminal
States:

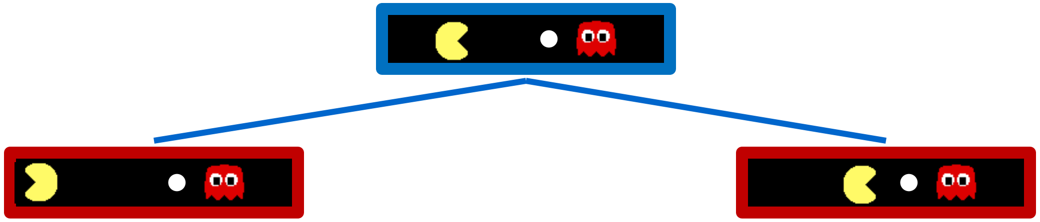
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

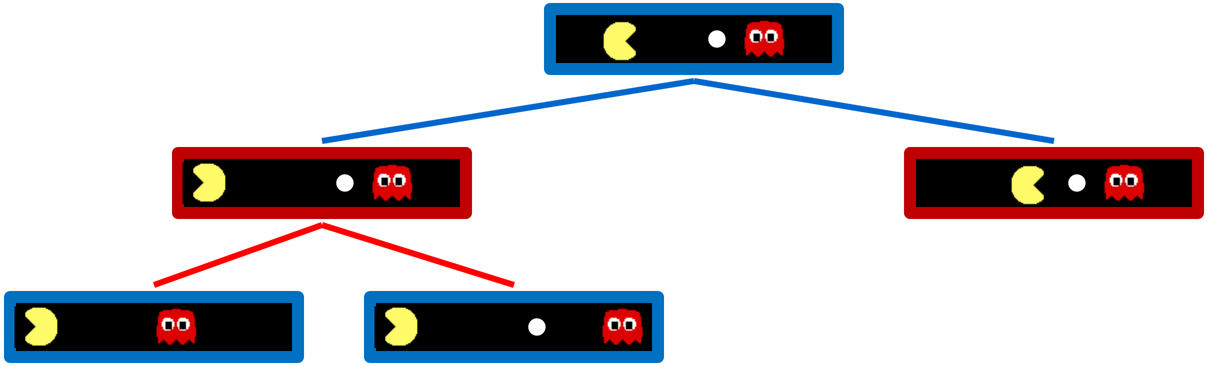
$$V(s) = \text{known}$$

# Adversarial Game Trees

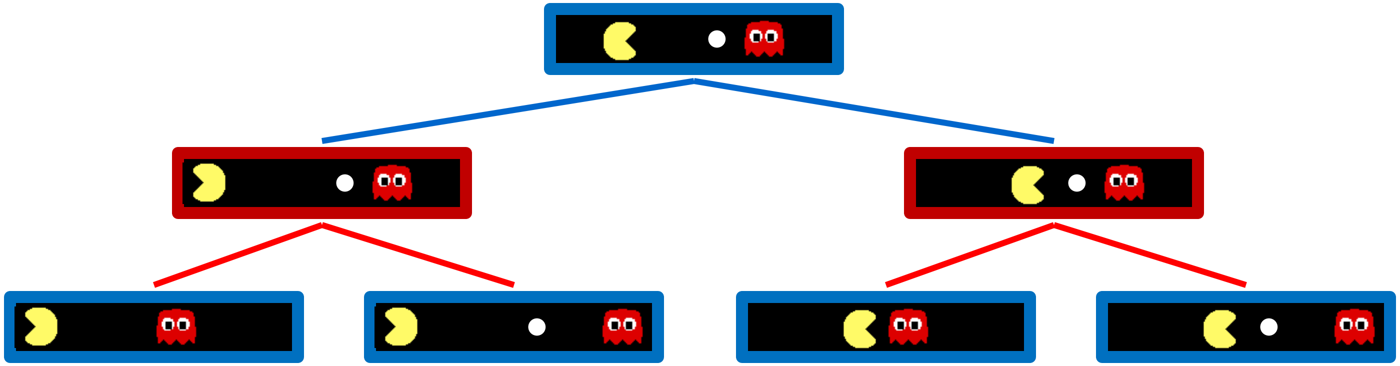
# Adversarial Game Trees



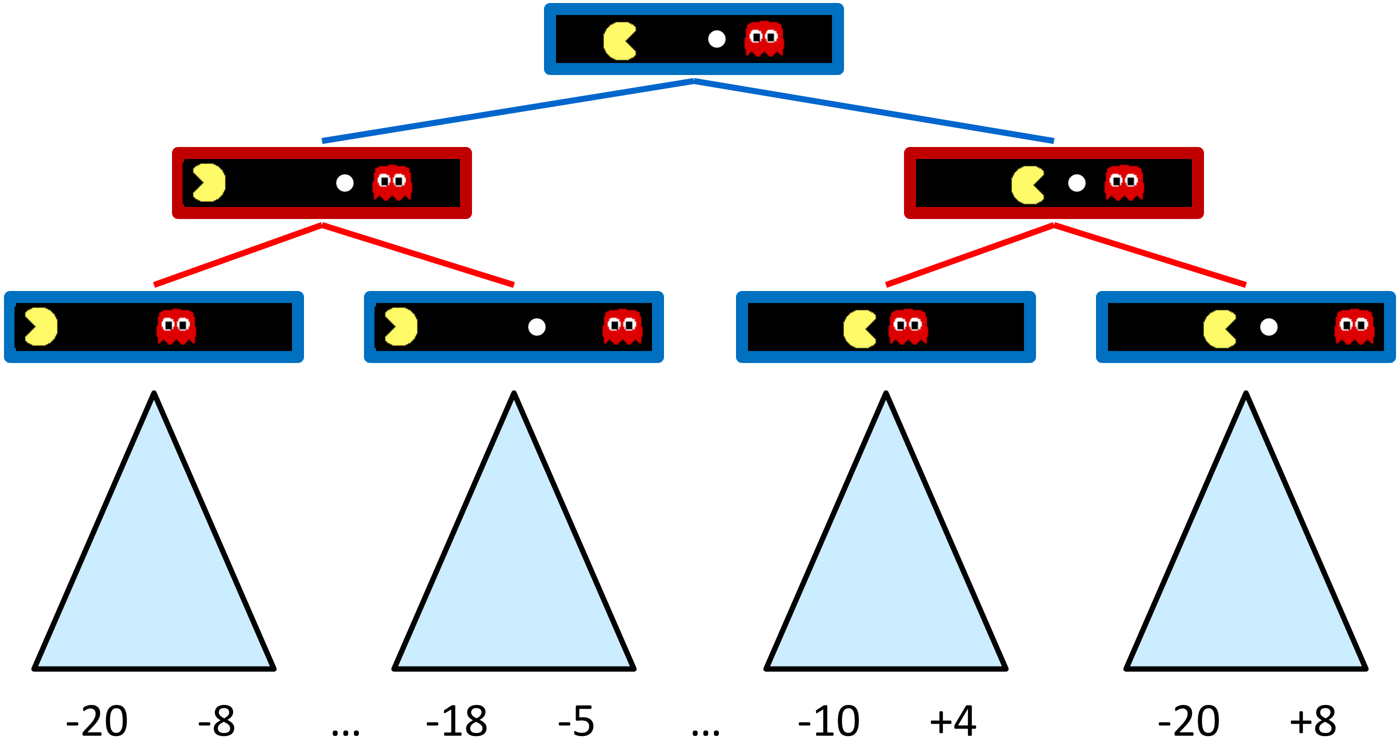
# Adversarial Game Trees



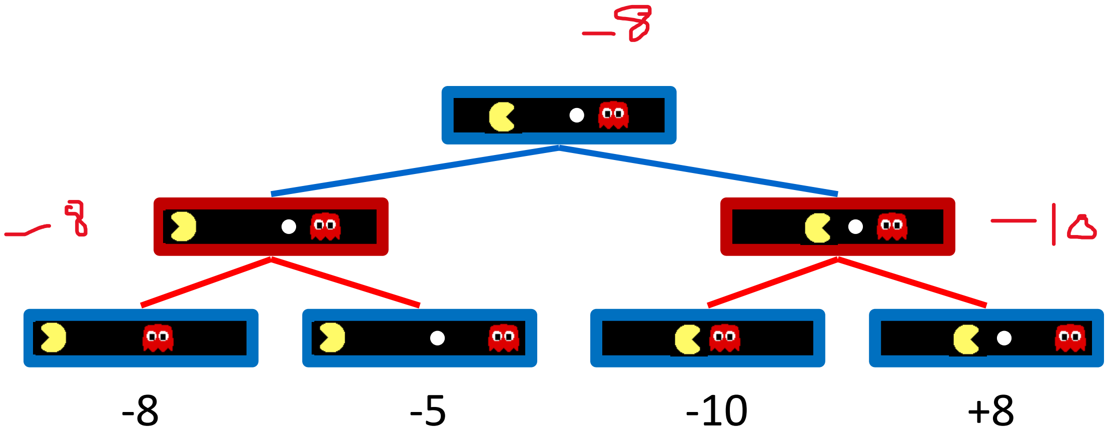
# Adversarial Game Trees



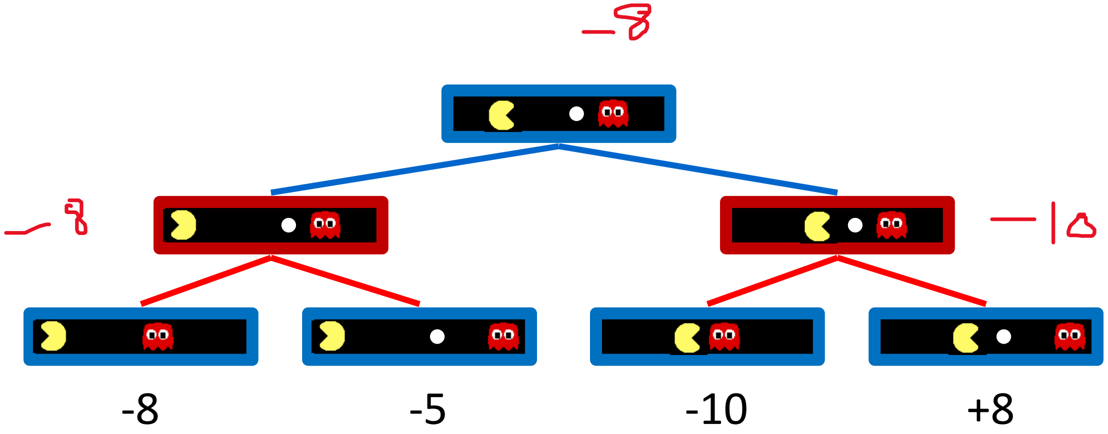
# Adversarial Game Trees



# Minimax Values



# Minimax Values



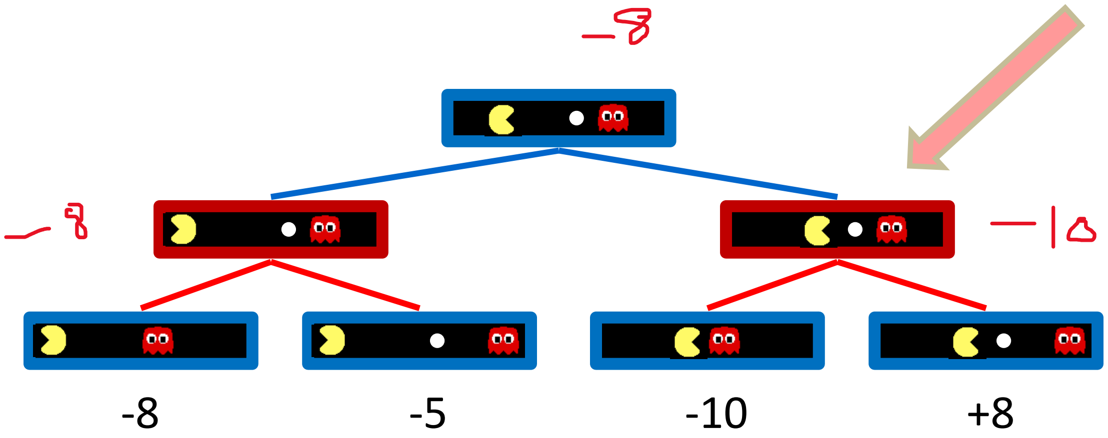
Terminal States:

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

# Minimax Values

States Under Opponent's Control:

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



Terminal States:

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

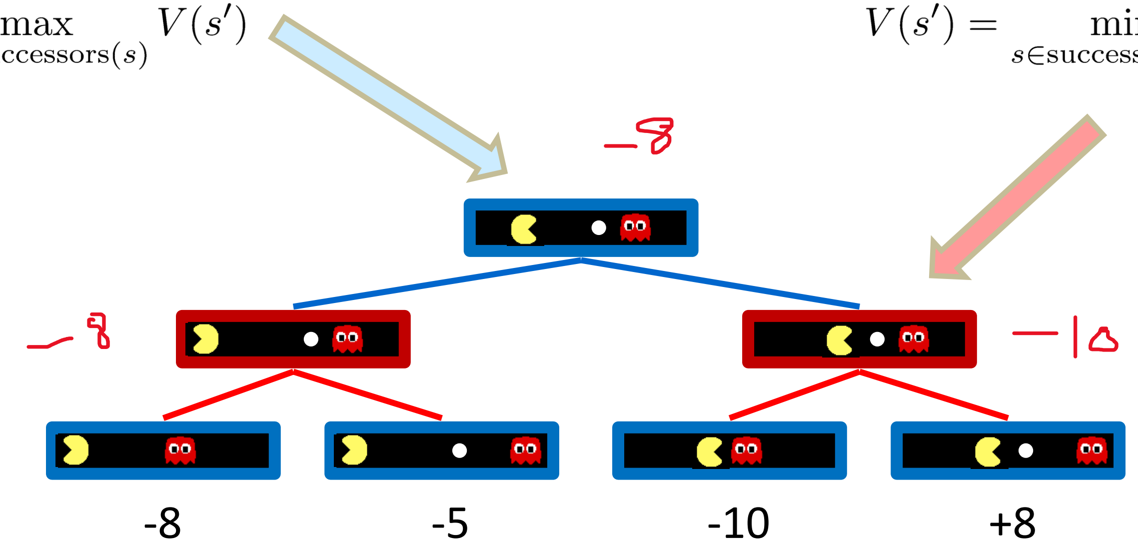
# Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

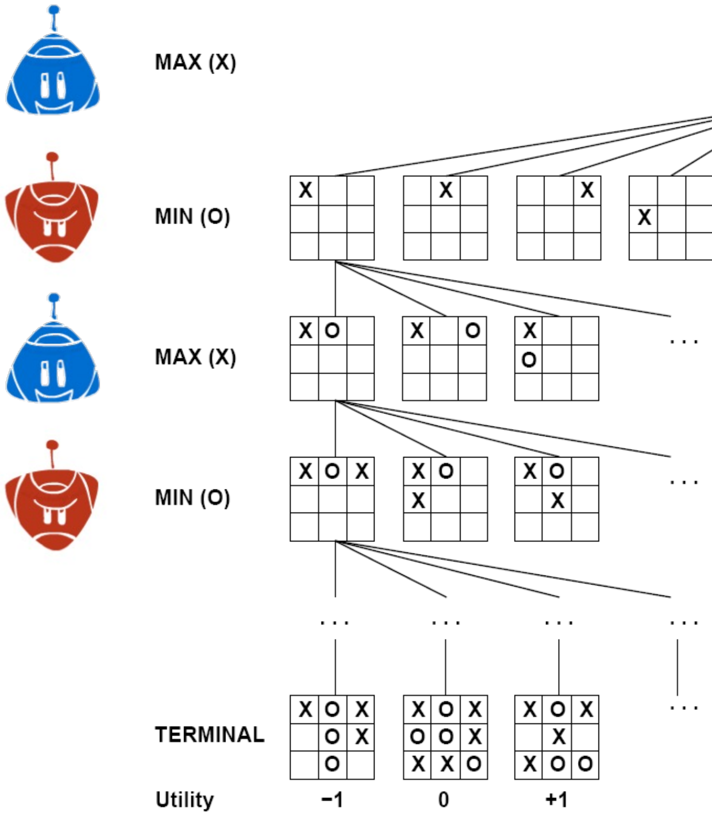
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

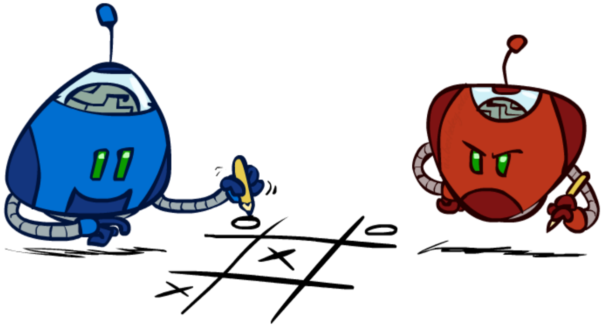


Terminal States:

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

# Tic-Tac-Toe Game Tree

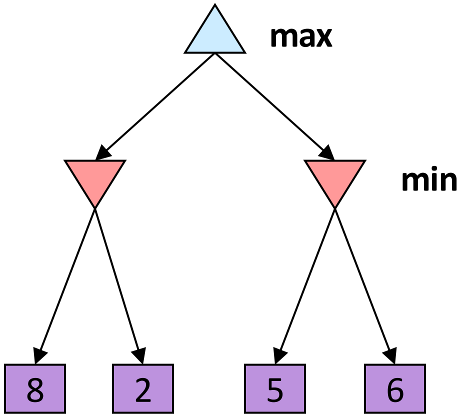




# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

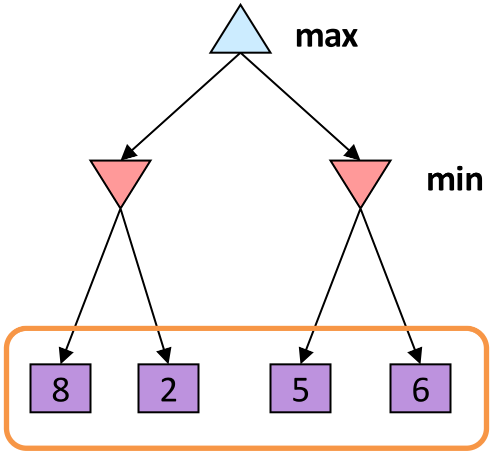
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

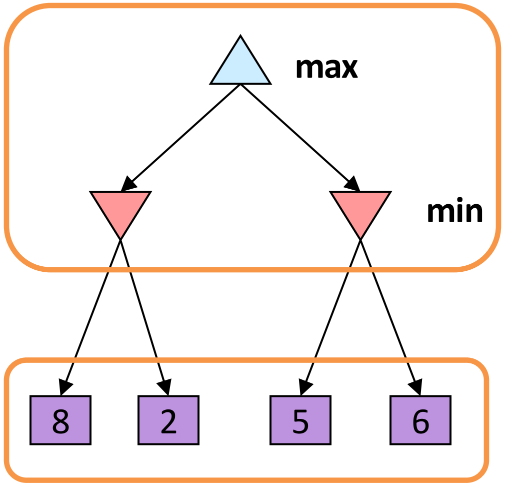


**Terminal values: part of the game**

# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

**Minimax values: computed recursively**



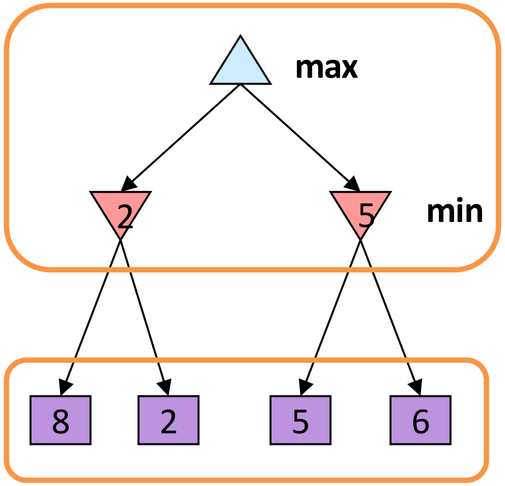
**Terminal values: part of the game**

# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

**Minimax values: computed recursively**



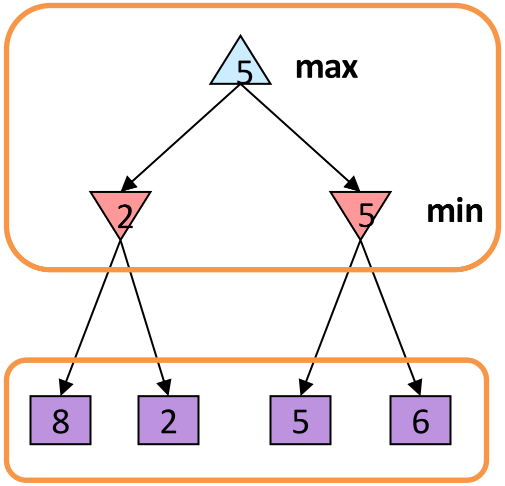
**Terminal values: part of the game**

# Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

**Minimax values: computed recursively**



**Terminal values: part of the game**

# Minimax Algorithm

1. Create MAX node with current board configuration
2. *Expand nodes to some **depth** (a.k.a. **plys**) of **lookahead** in game*
3. Apply evaluation function at each **leaf** node
4. ***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
5. Choose move to child node whose backed-up value determined value at root

# Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

# Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

# Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

# Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

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

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

# Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

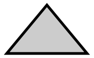
# Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

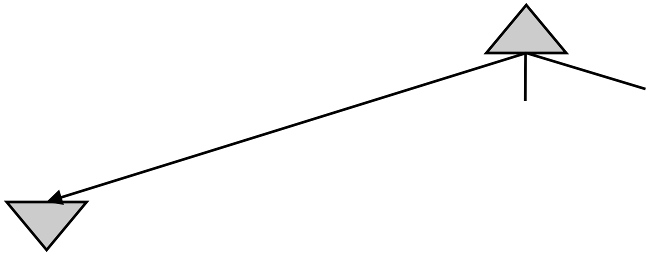
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

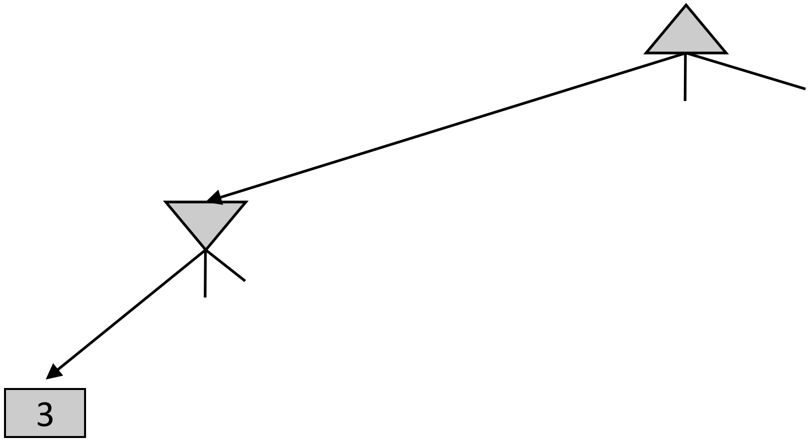
# Minimax Example



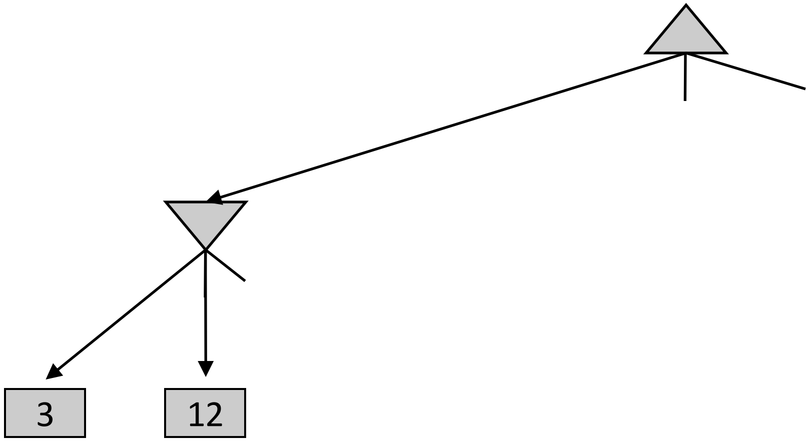
# Minimax Example



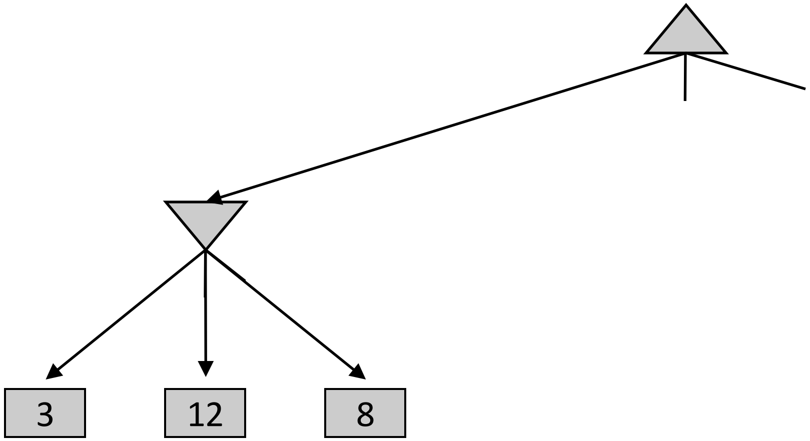
# Minimax Example



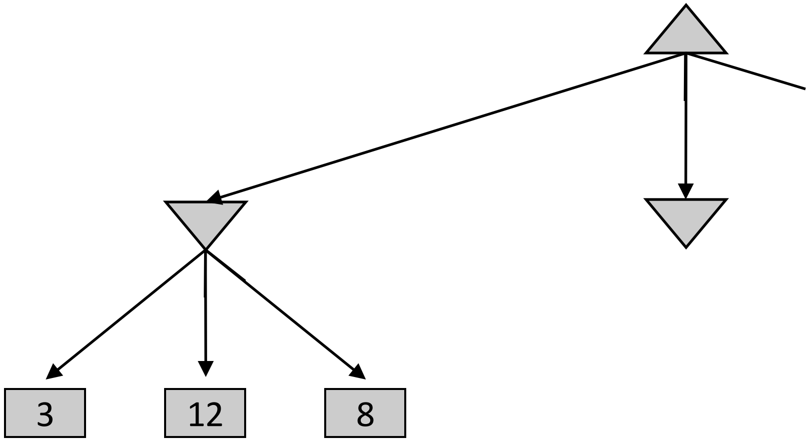
# Minimax Example



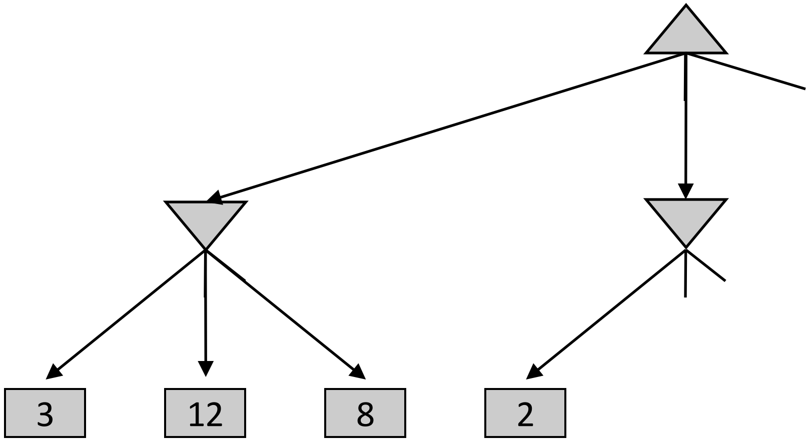
# Minimax Example



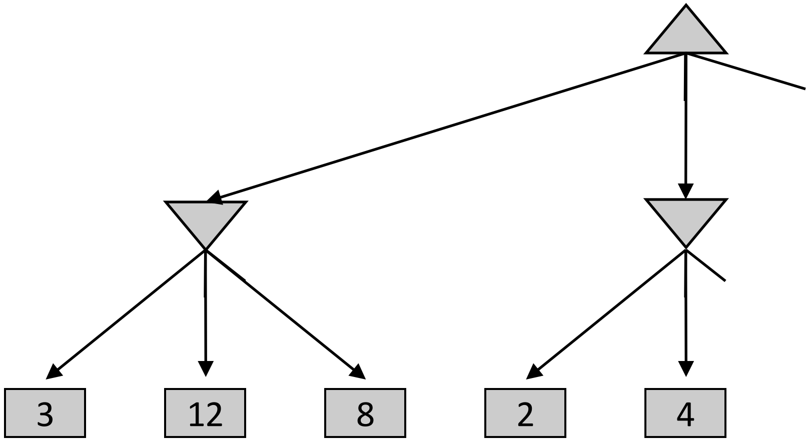
# Minimax Example



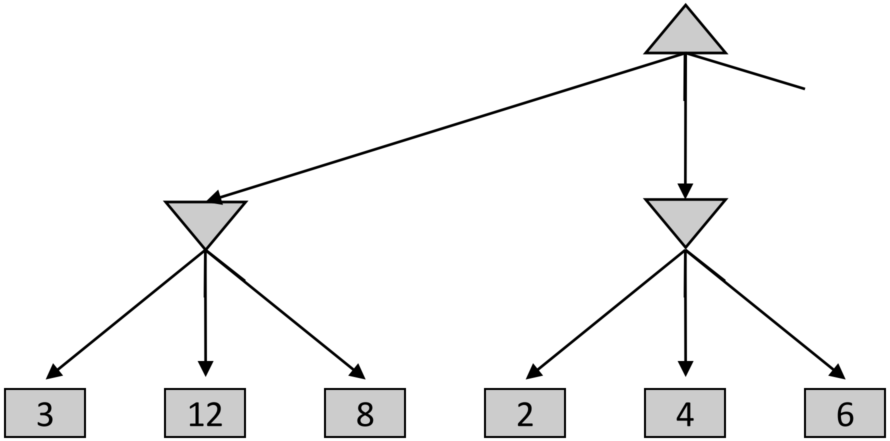
# Minimax Example



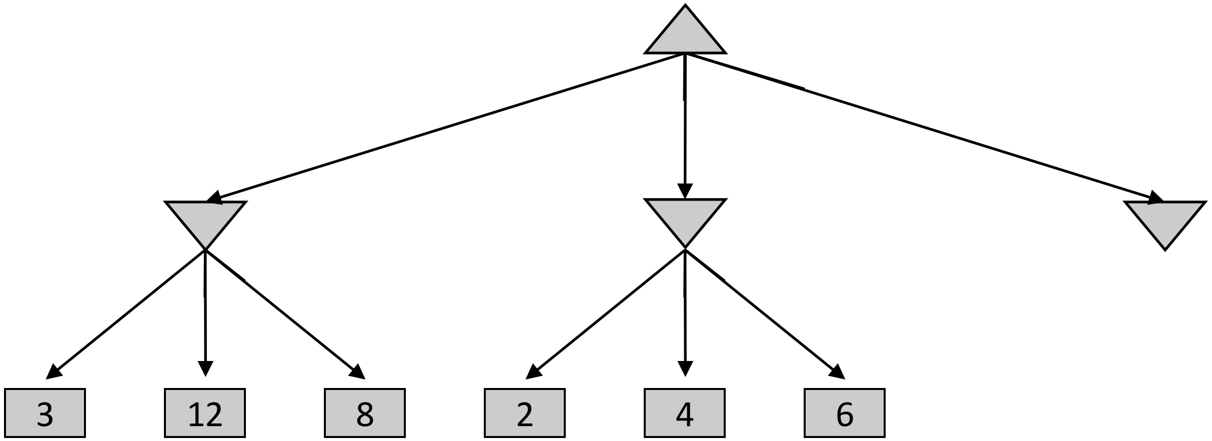
# Minimax Example



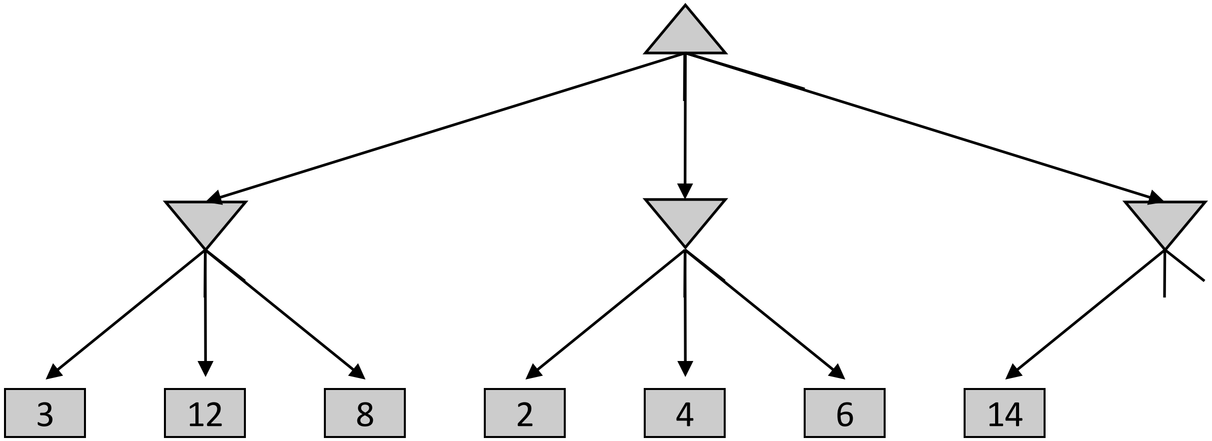
# Minimax Example



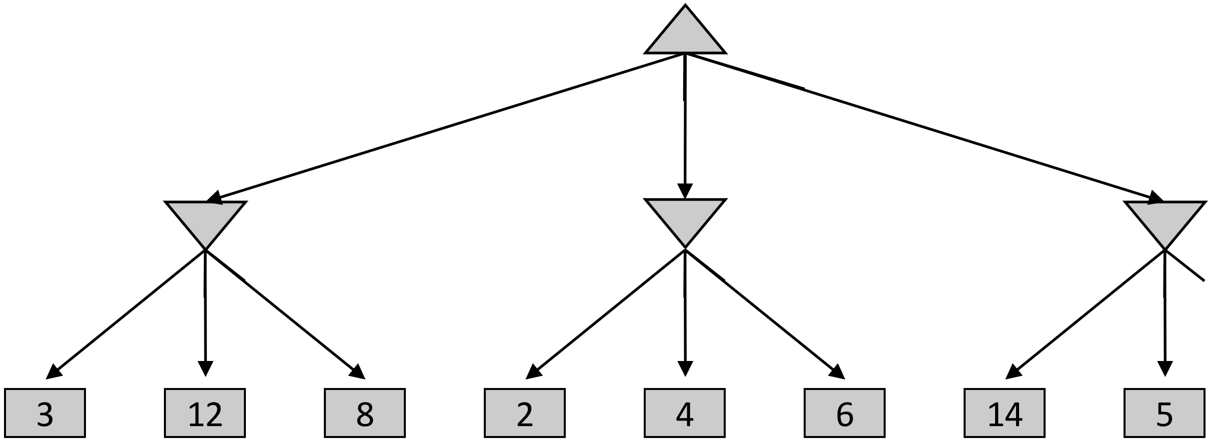
# Minimax Example



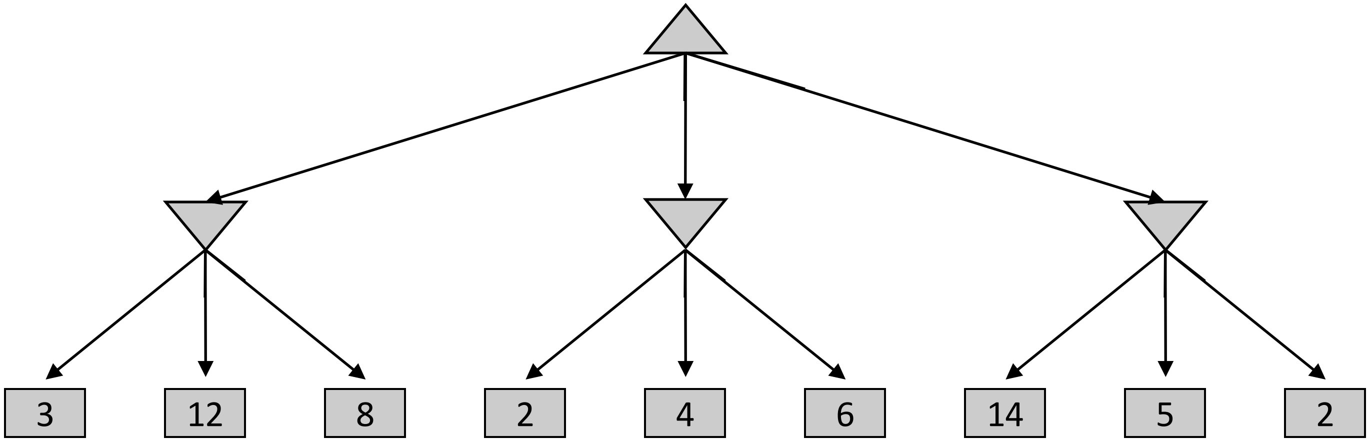
# Minimax Example



# Minimax Example

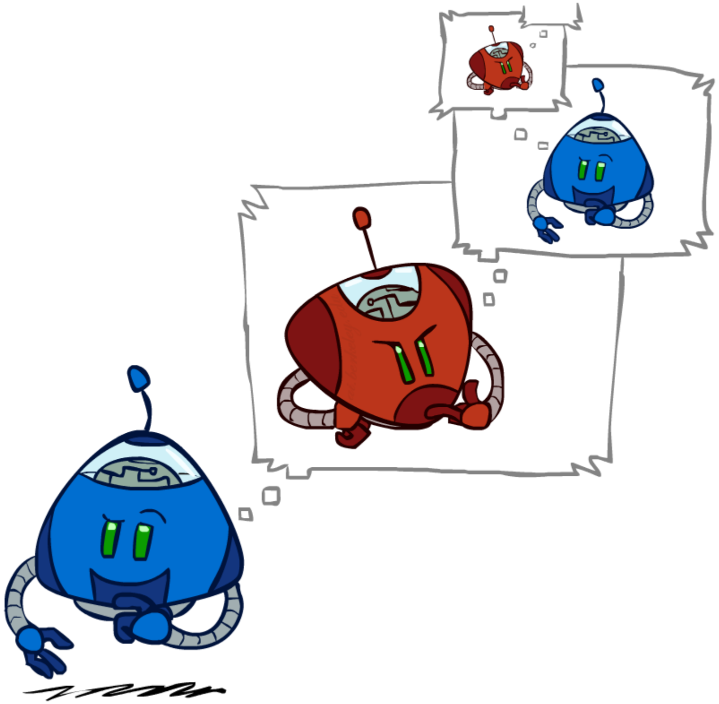


# Minimax Example

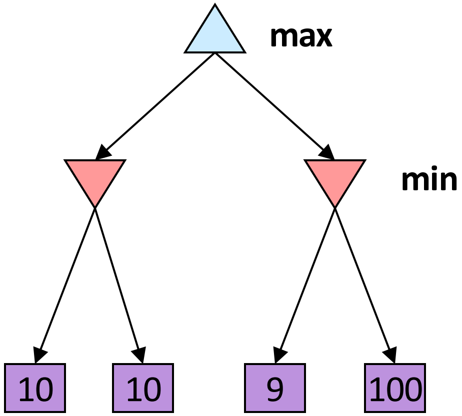


# Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

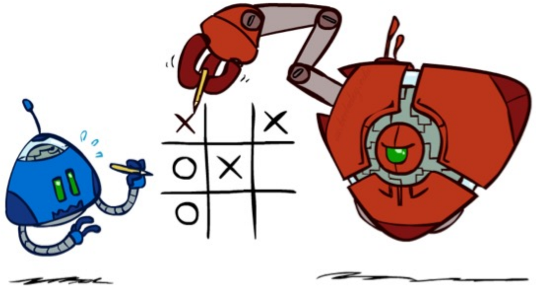


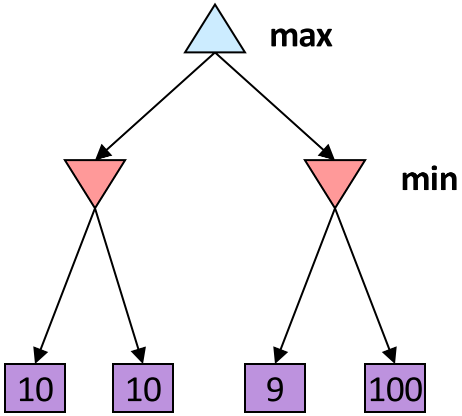
# Minimax Properties



Optimal against a perfect player. Otherwise?

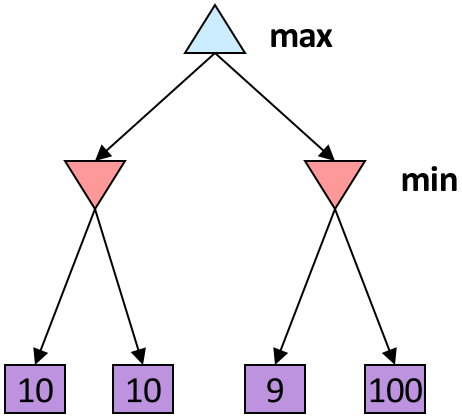
# Minimax Properties

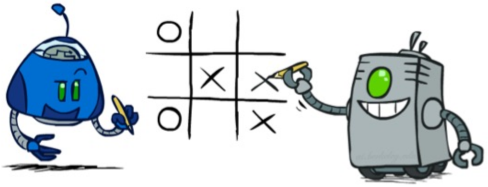




Optimal against a perfect player. Otherwise?

# Minimax Properties



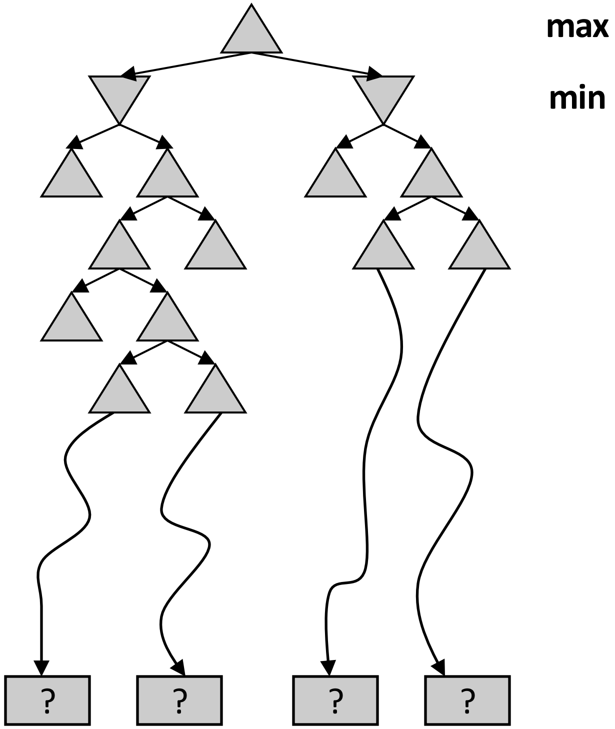


Optimal against a perfect player. Otherwise?

# Resource Limits

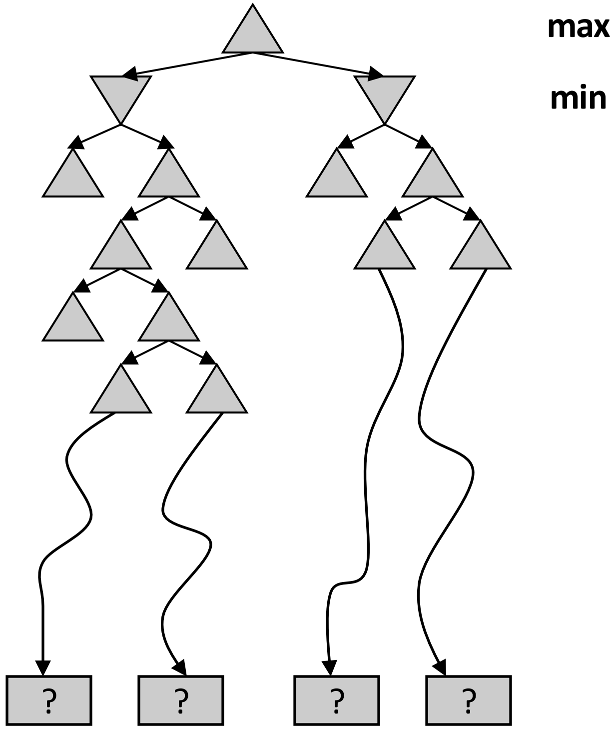
# Resource Limits

- Problem: In realistic games, cannot search to leaves!



# Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



# 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 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 that 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 chess features 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

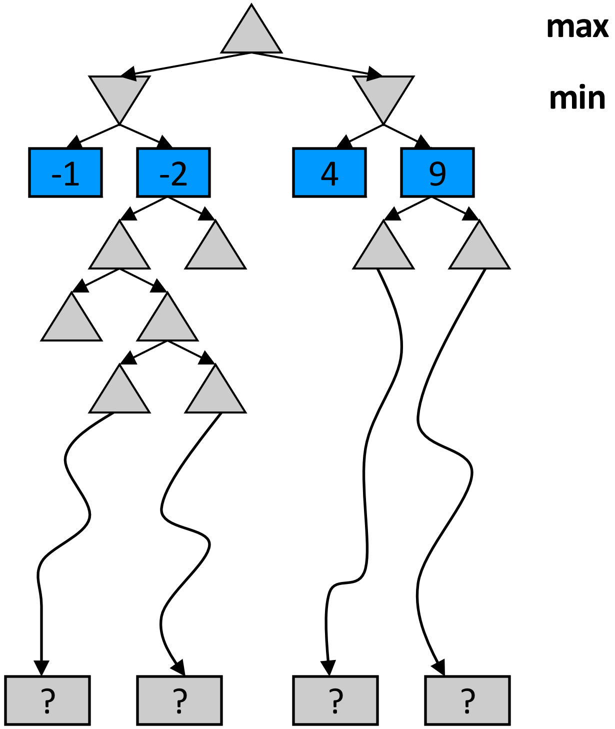
# But, that’s not how people play

- People also 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?

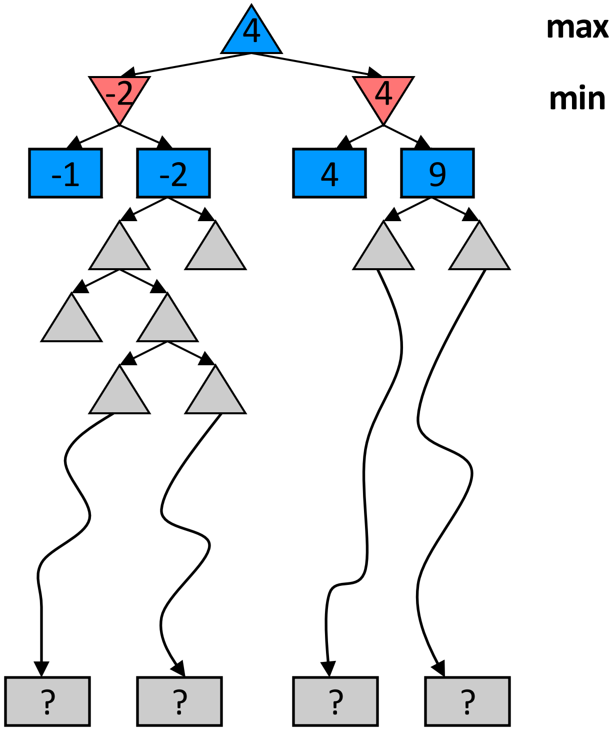
# Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



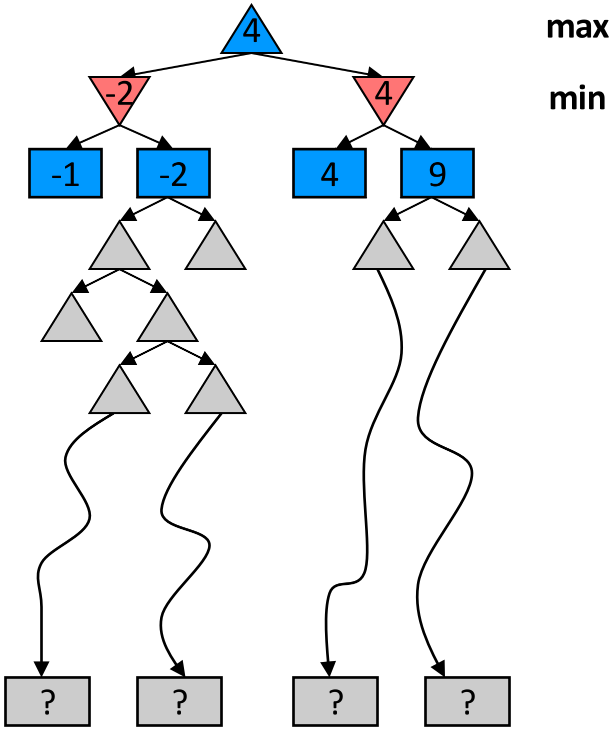
# Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



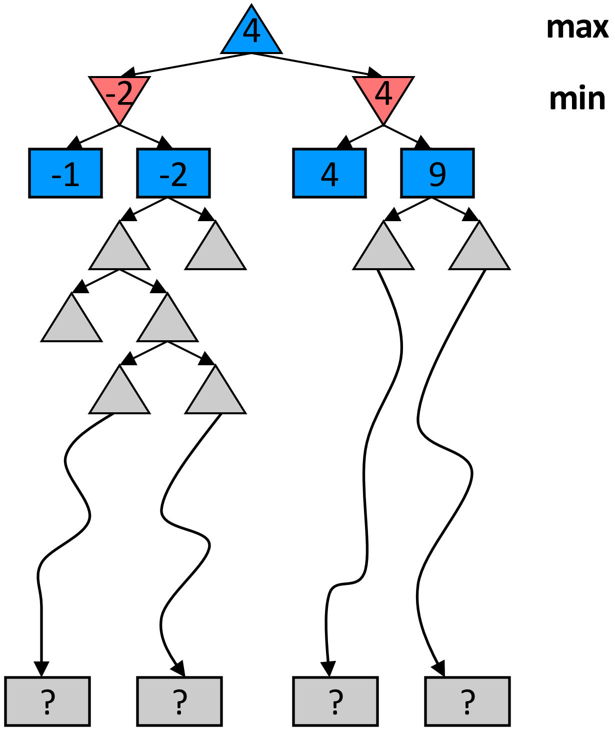
# Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha$-$\beta$ reaches about depth 8 – decent chess program



# Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha$-$\beta$ reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



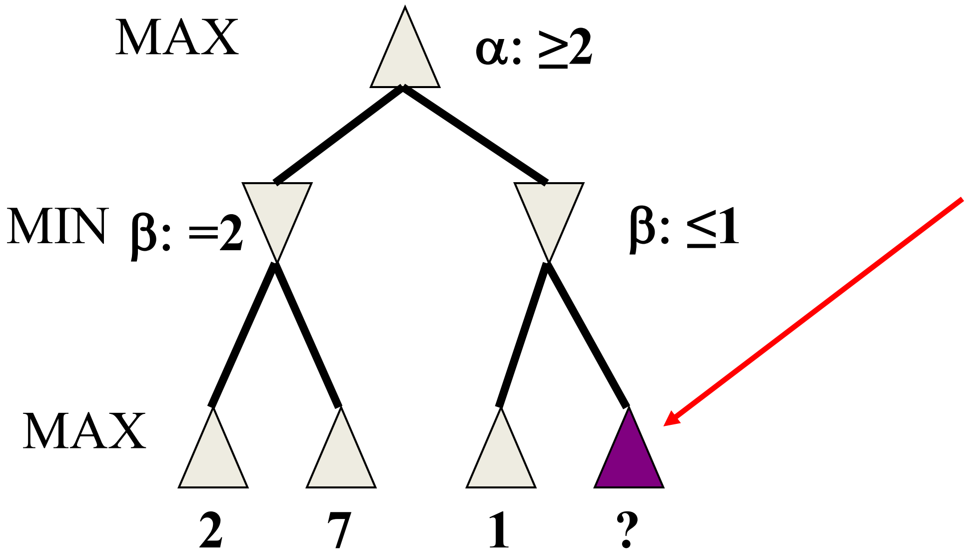
# Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

# 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 (MIT)

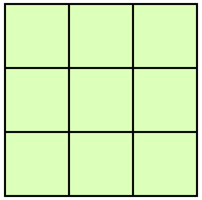


- We don't need to compute the value at this node
- No matter what it is, it can't affect value of the root node

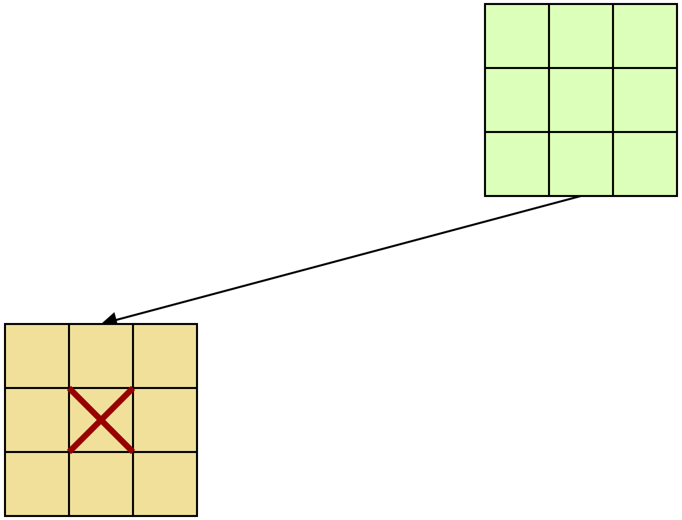
# Alpha-beta pruning

- 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 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 anceastor i of N

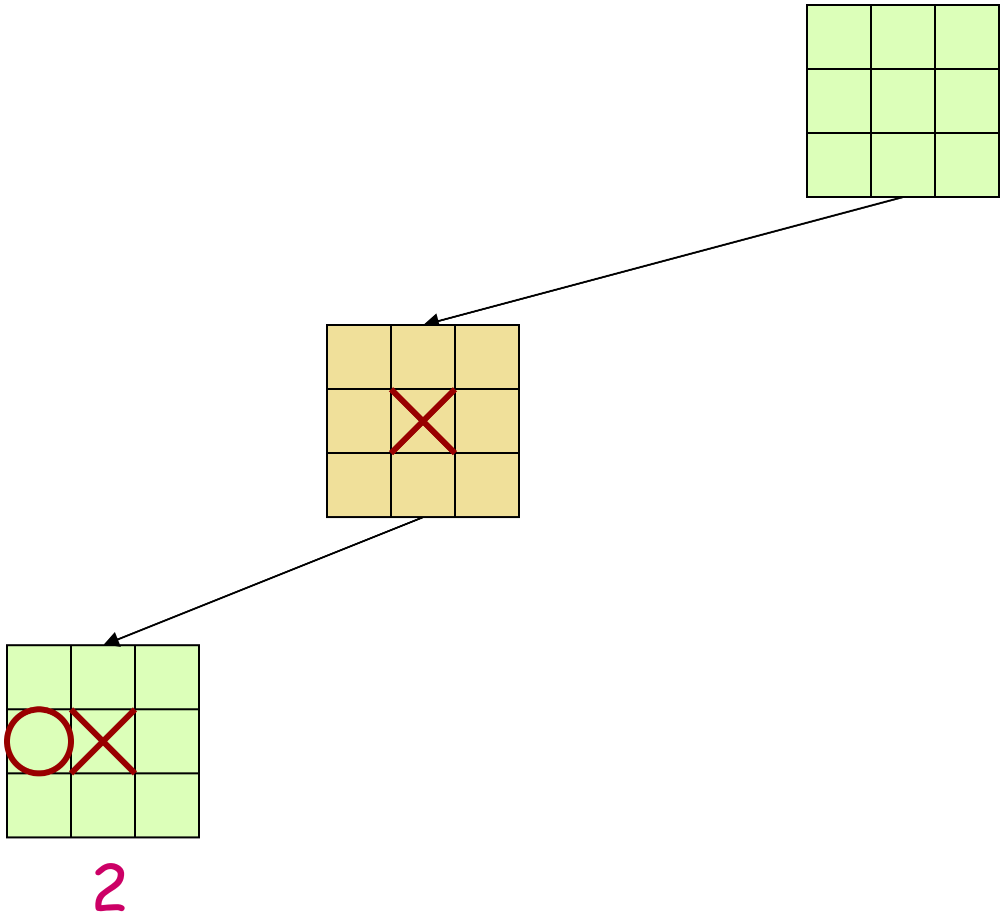
# Alpha-Beta Tic-Tac-Toe Example



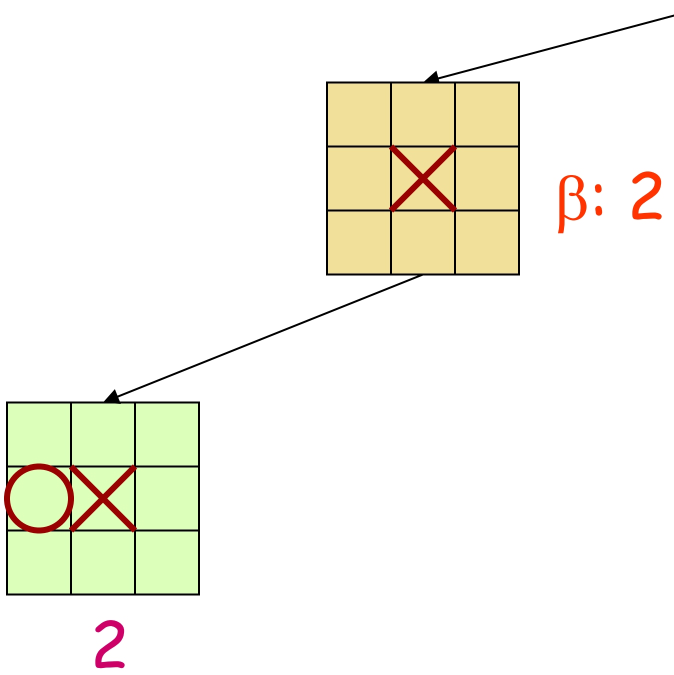
# Alpha-Beta Tic-Tac-Toe Example



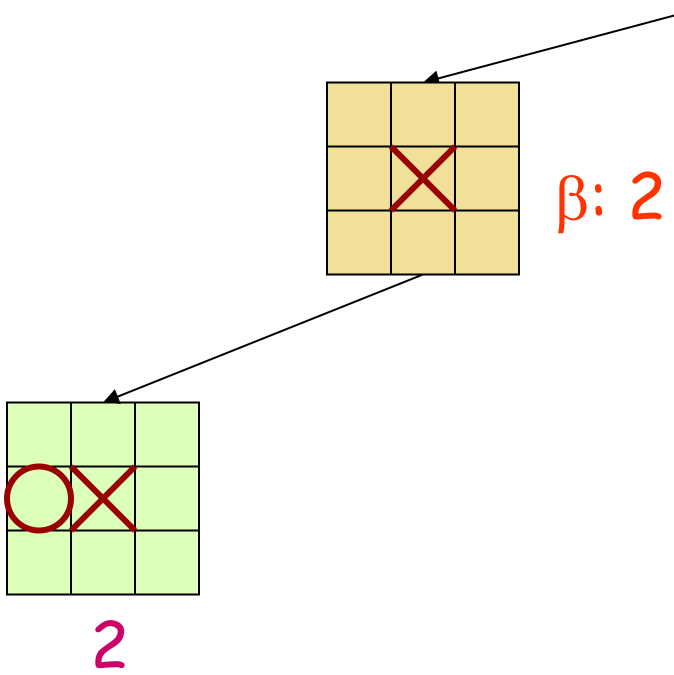
# Alpha-Beta Tic-Tac-Toe Example



# Alpha-Beta Tic-Tac-Toe Example

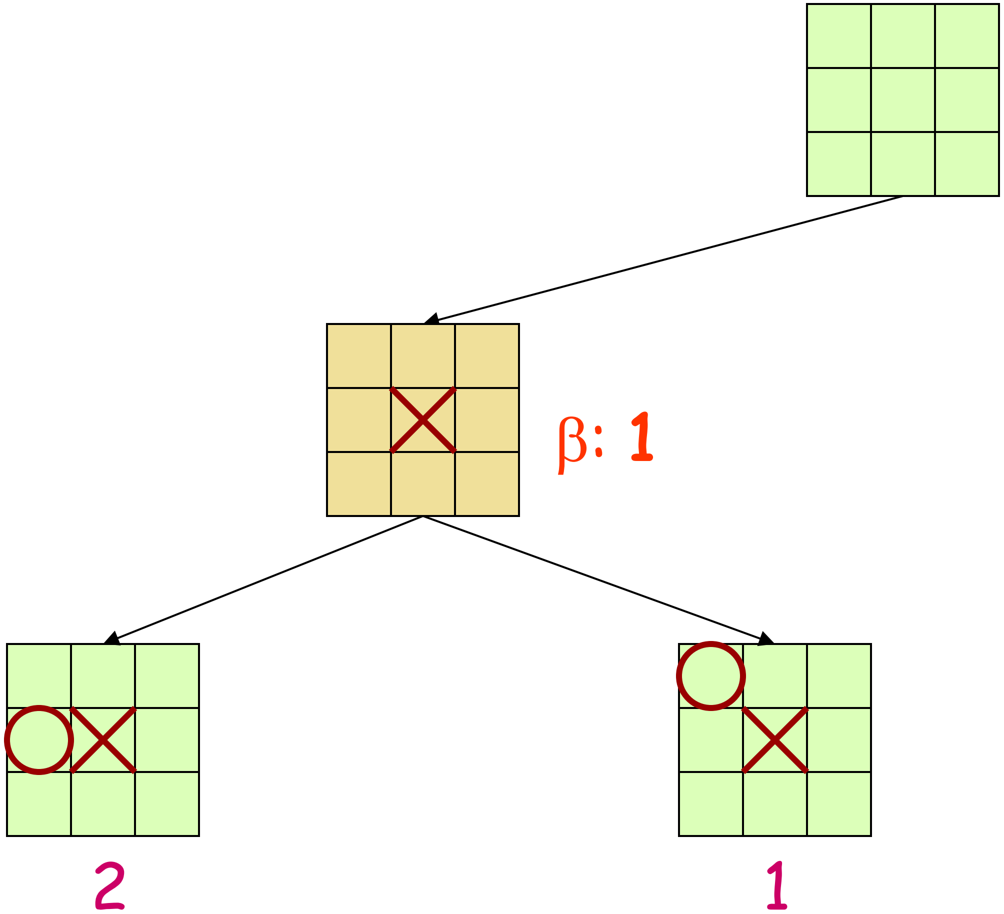


# Alpha-Beta Tic-Tac-Toe Example

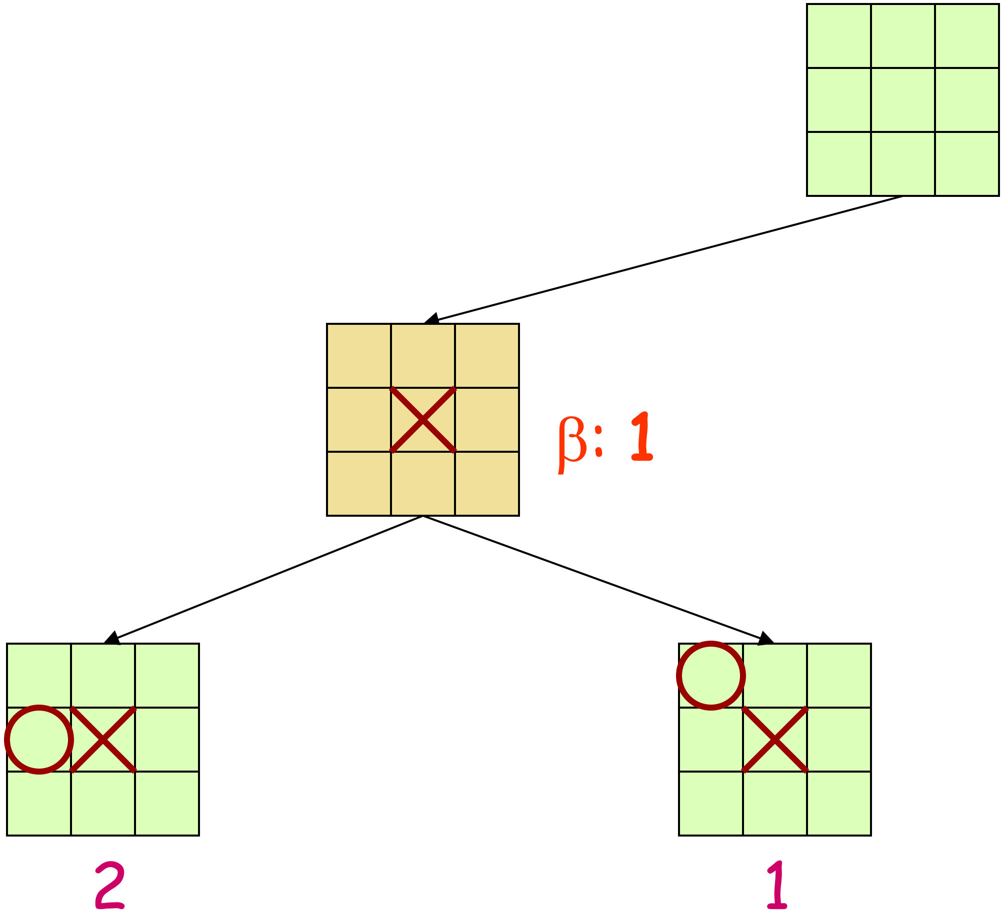


Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

# Alpha-Beta Tic-Tac-Toe Example

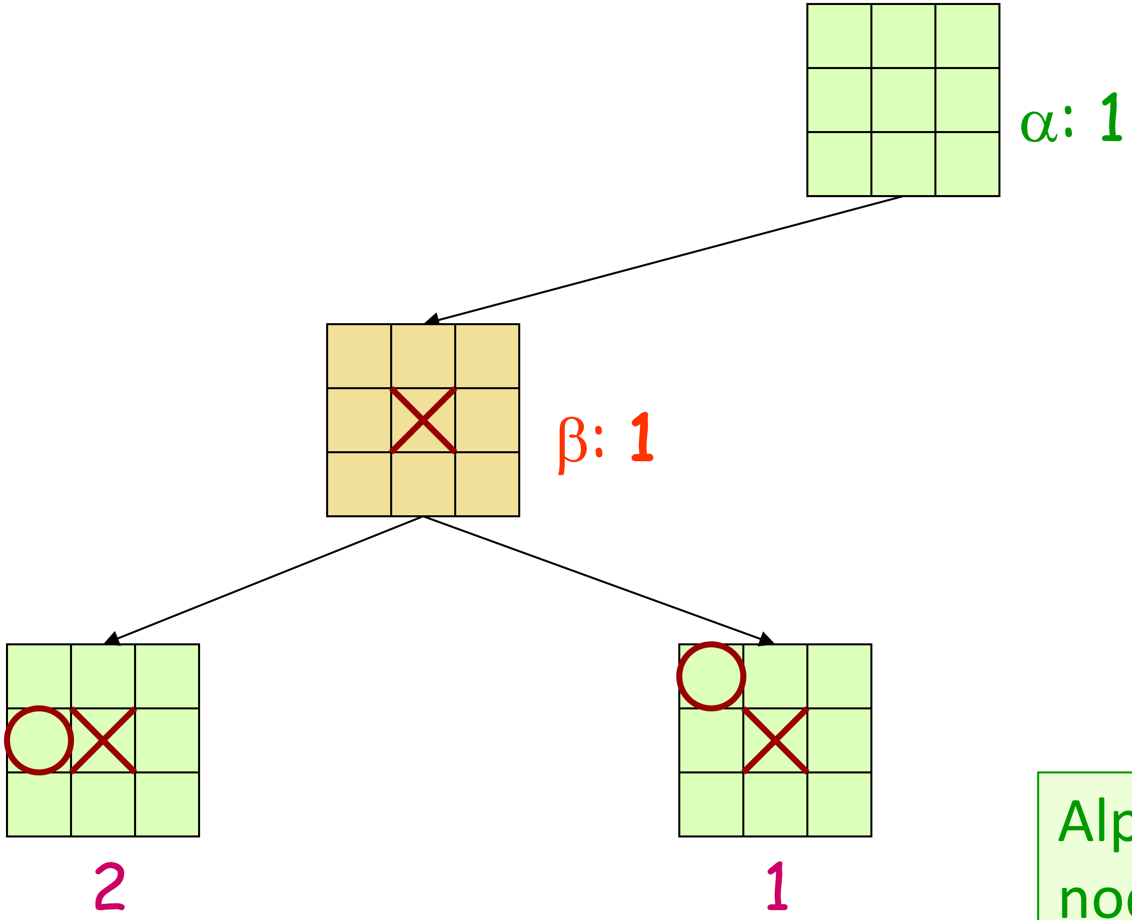


# Alpha-Beta Tic-Tac-Toe Example



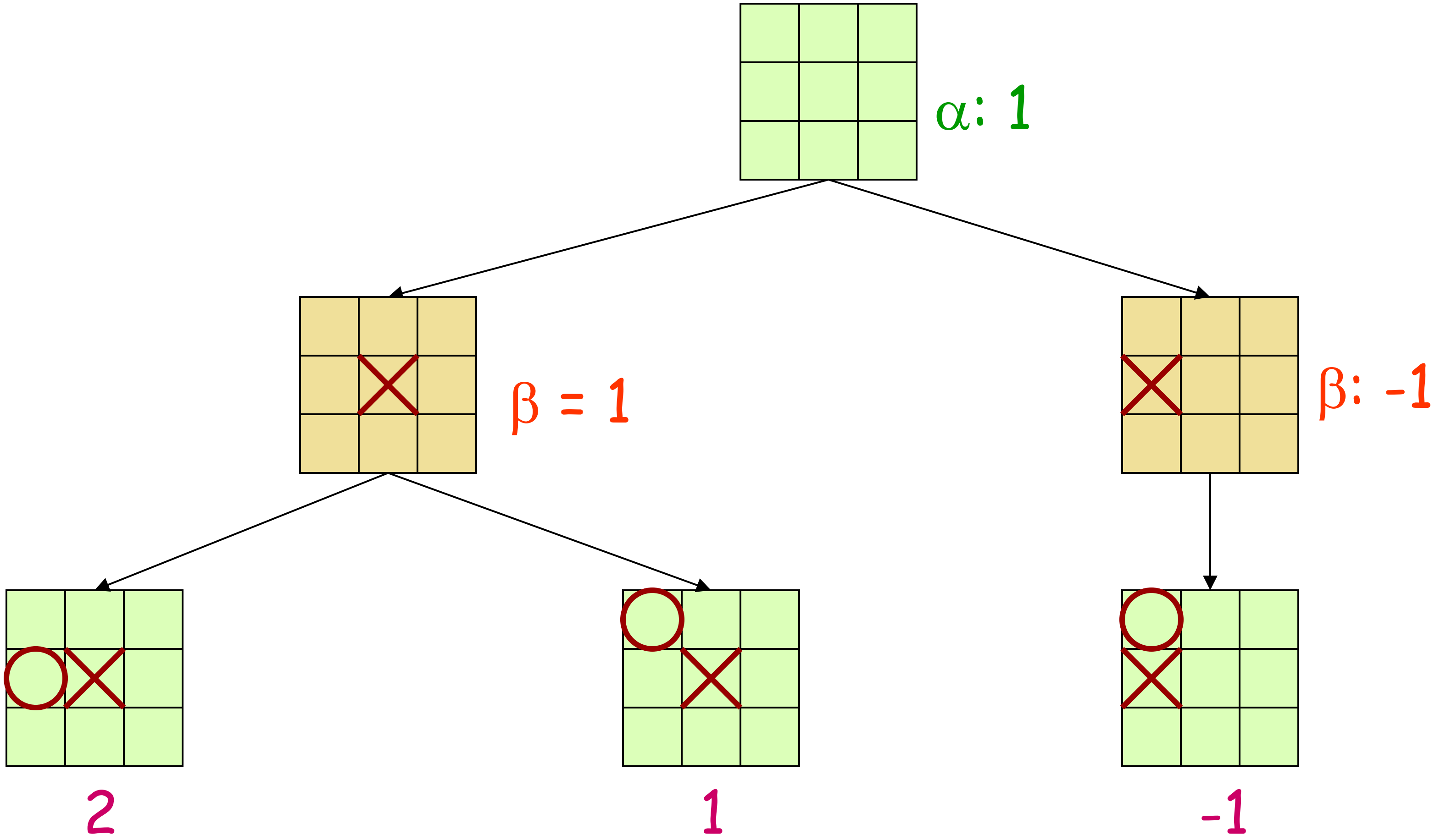
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

# Alpha-Beta Tic-Tac-Toe Example

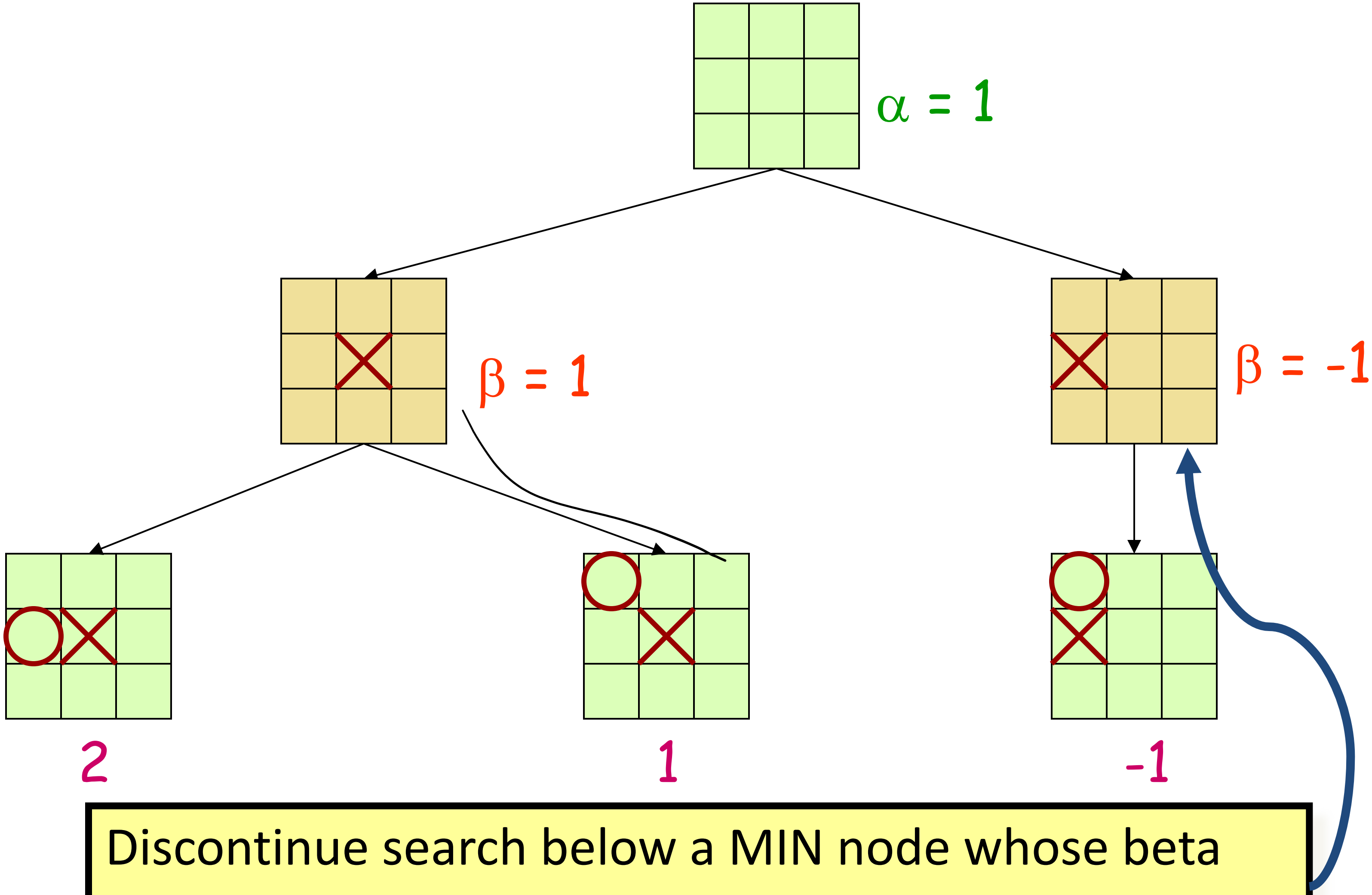


Alpha value of MAX node is **lower** bound on final backed-up value; it can never decrease

# 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

# Alpha-Beta Implementation

$\alpha$: MAX’s best option on path to root

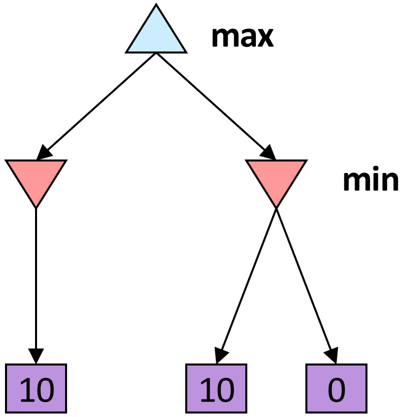
$\beta$: MIN’s best option on path to root

```
def max-value(state, $\alpha$, $\beta$):
    initialize v = $-\infty$
    for each successor of state:
        v = max(v, value(successor, $\alpha$, $\beta$))
        if v $\geq \beta$ return v
        $\alpha$ = max($\alpha$, v)
    return v
```

```
def min-value(state , $\alpha$, $\beta$):
    initialize v = $+\infty$
    for each successor of state:
        v = min(v, value(successor, $\alpha$, $\beta$))
        if v $\leq \alpha$ return v
        $\beta$ = min($\beta$, v)
    return v
```

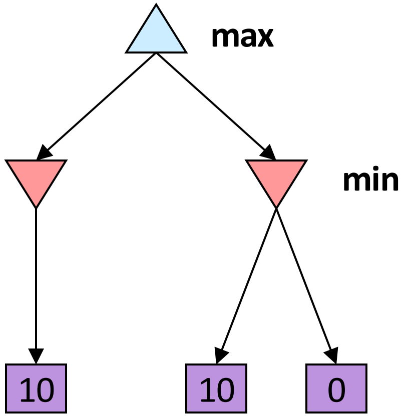
# Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection

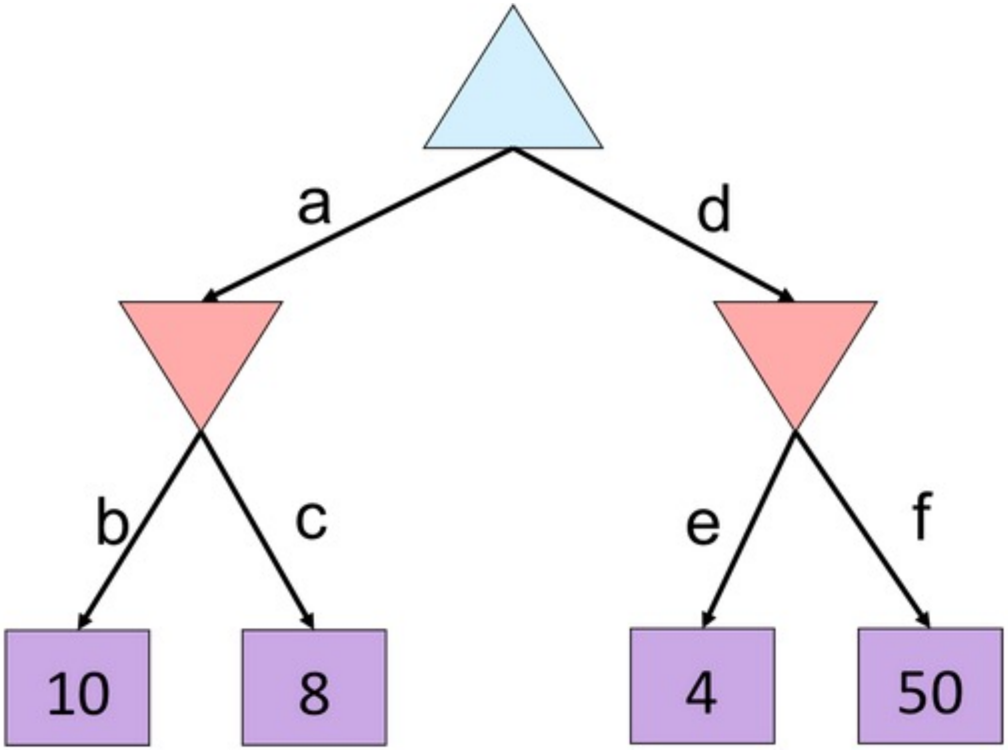


# Alpha-Beta Pruning Properties

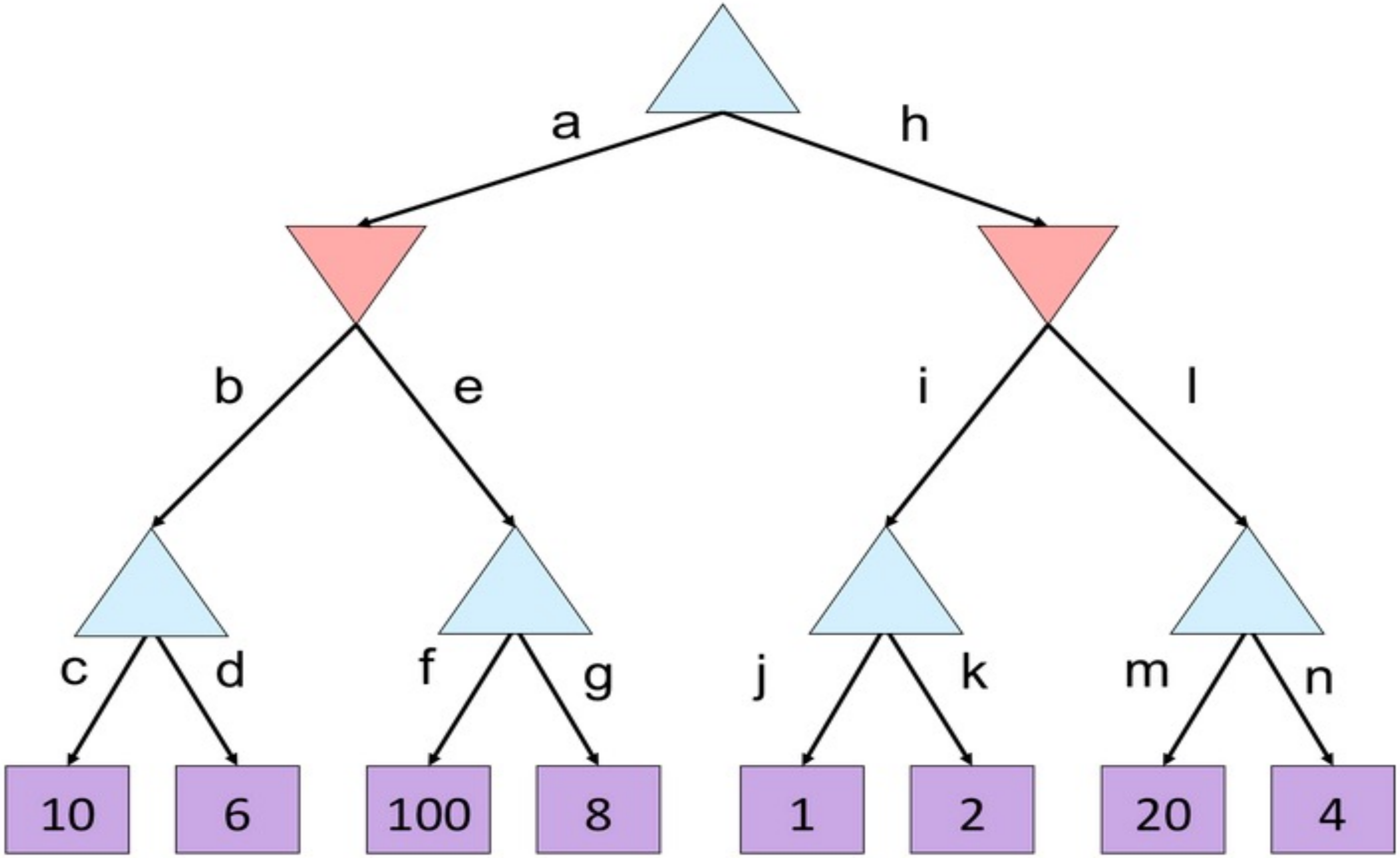
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - **Time complexity drops to O($b^{m/2}$)**
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



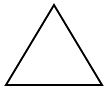
# Alpha-Beta Quiz



# Alpha-Beta Quiz 2



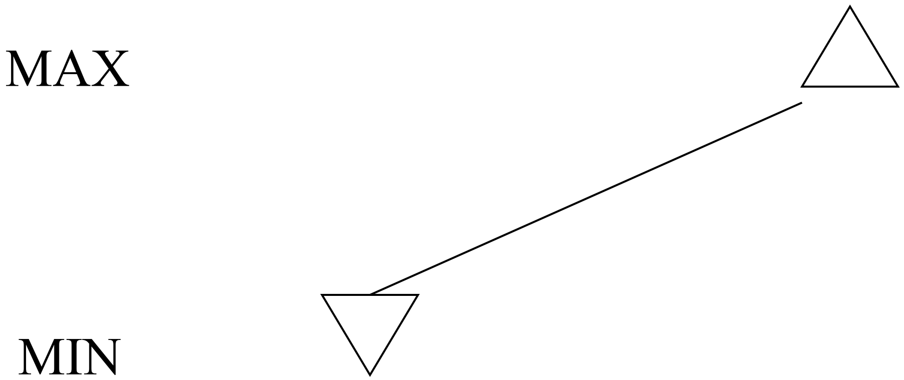
# Another alpha-beta example



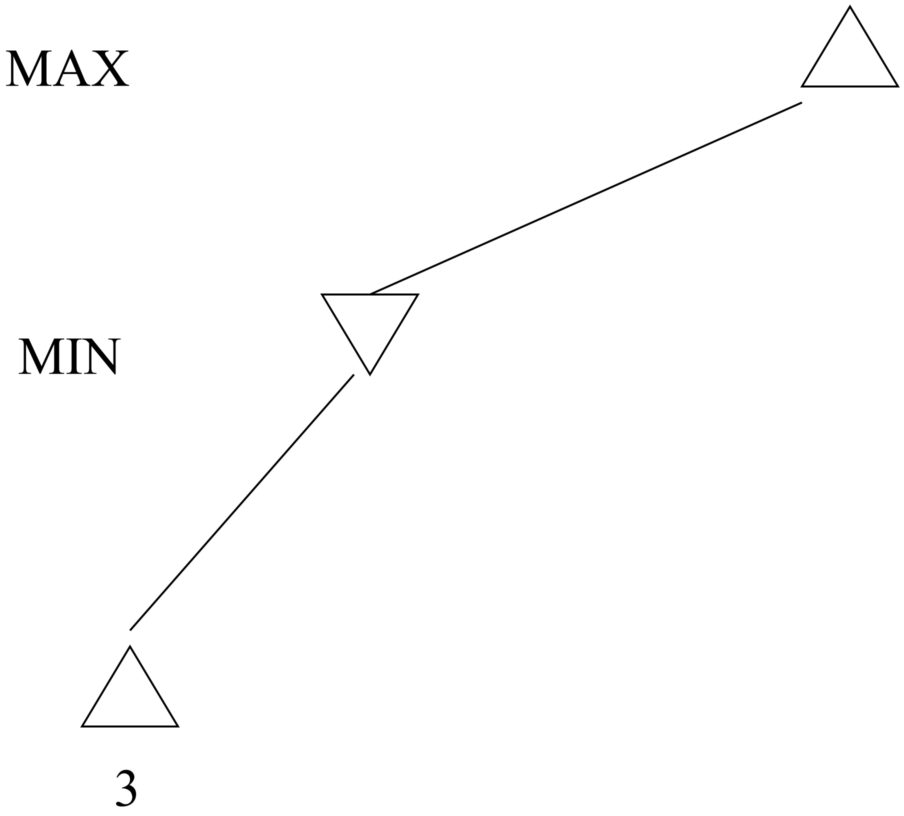
MAX

MIN

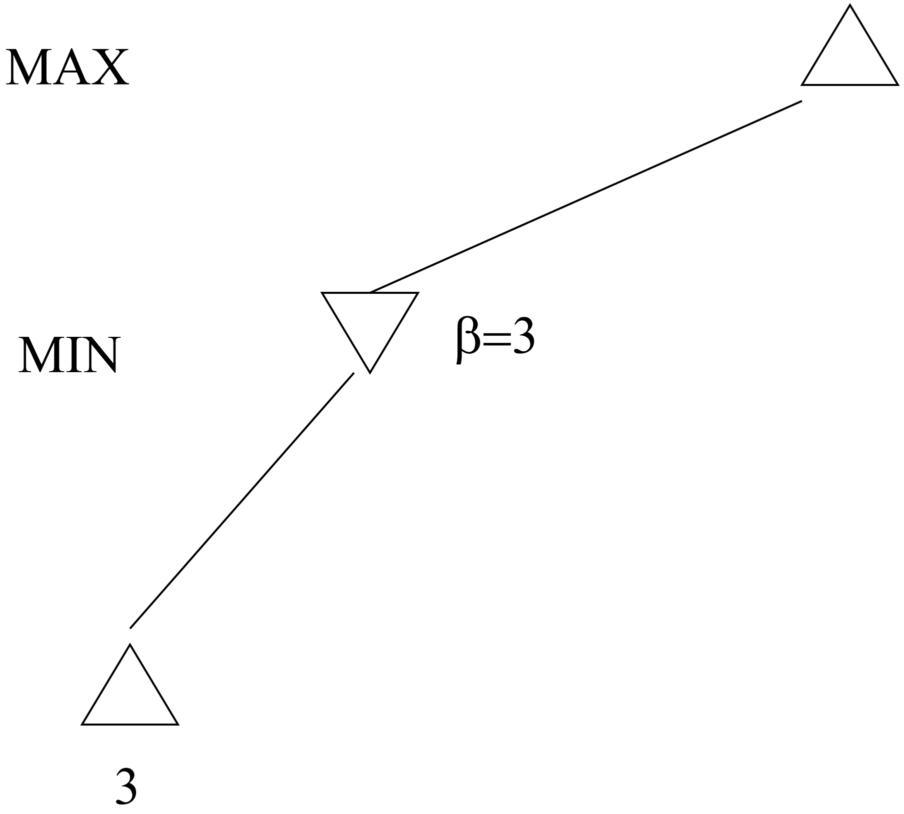
# Another alpha-beta example



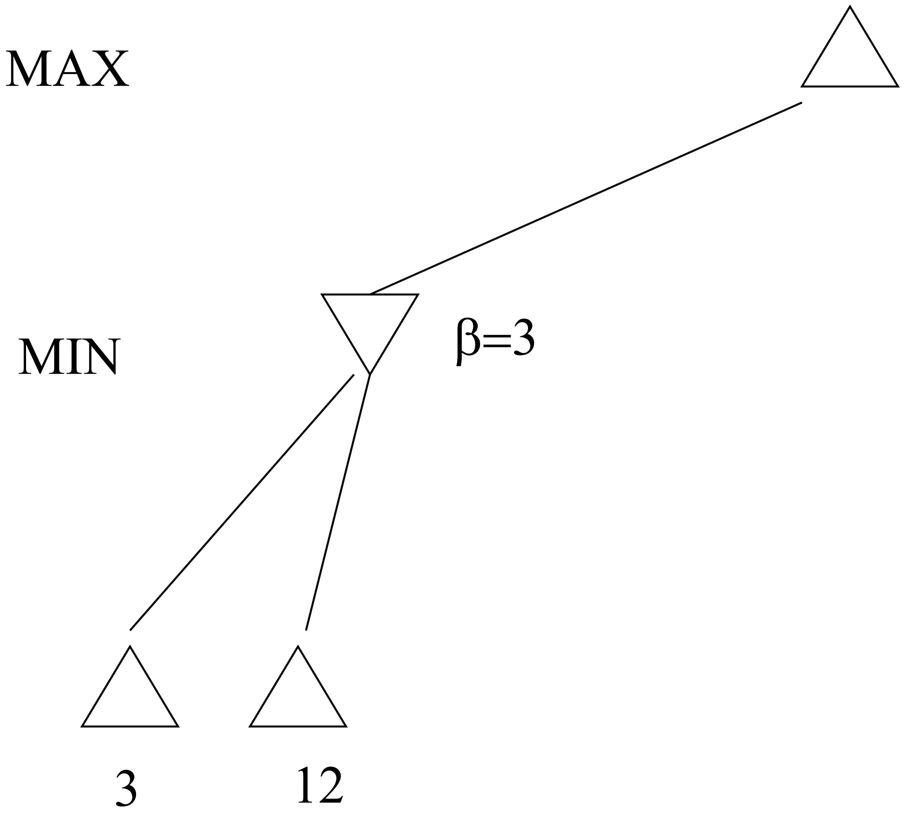
# Another alpha-beta example



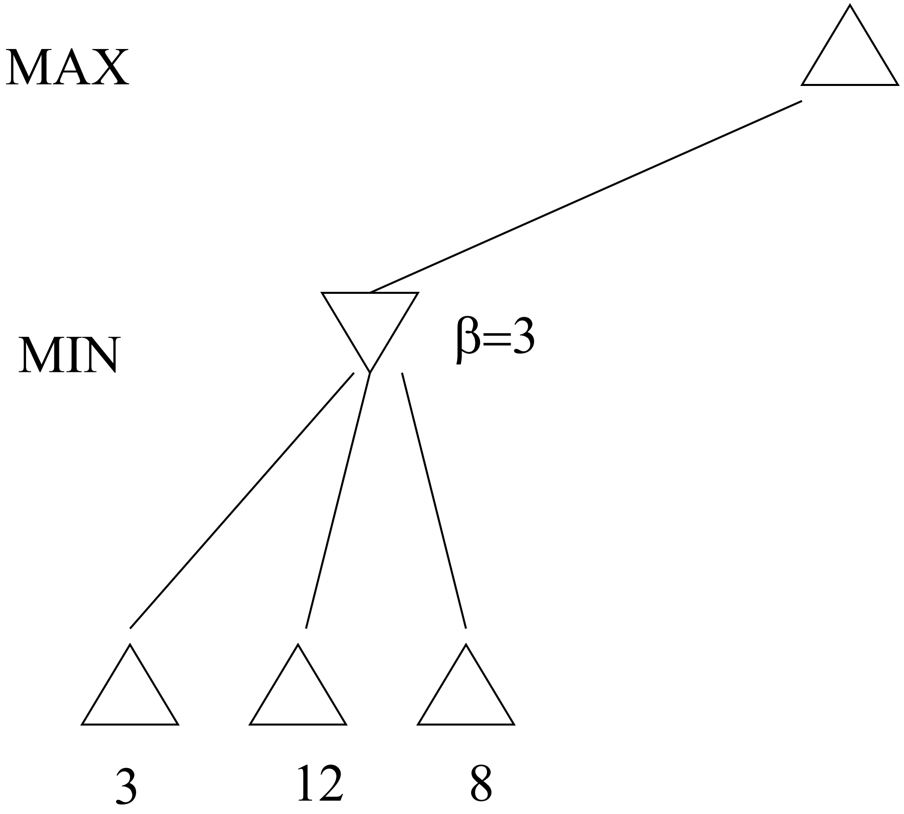
# Another alpha-beta example



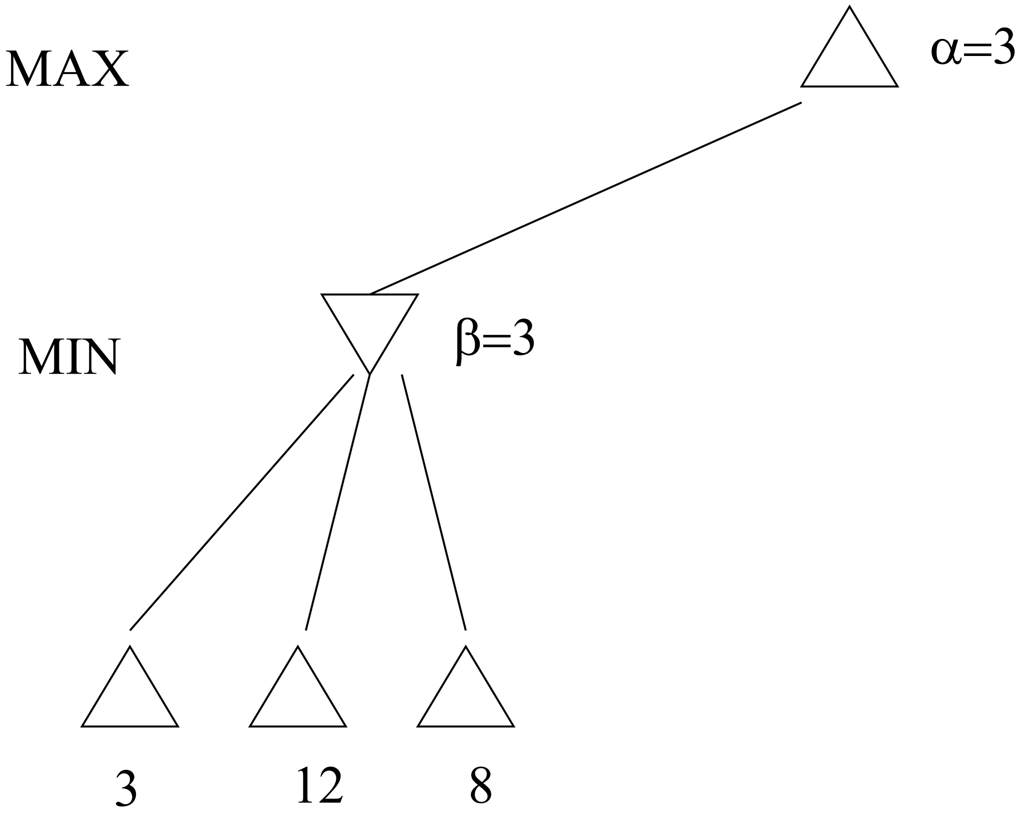
# Another alpha-beta example



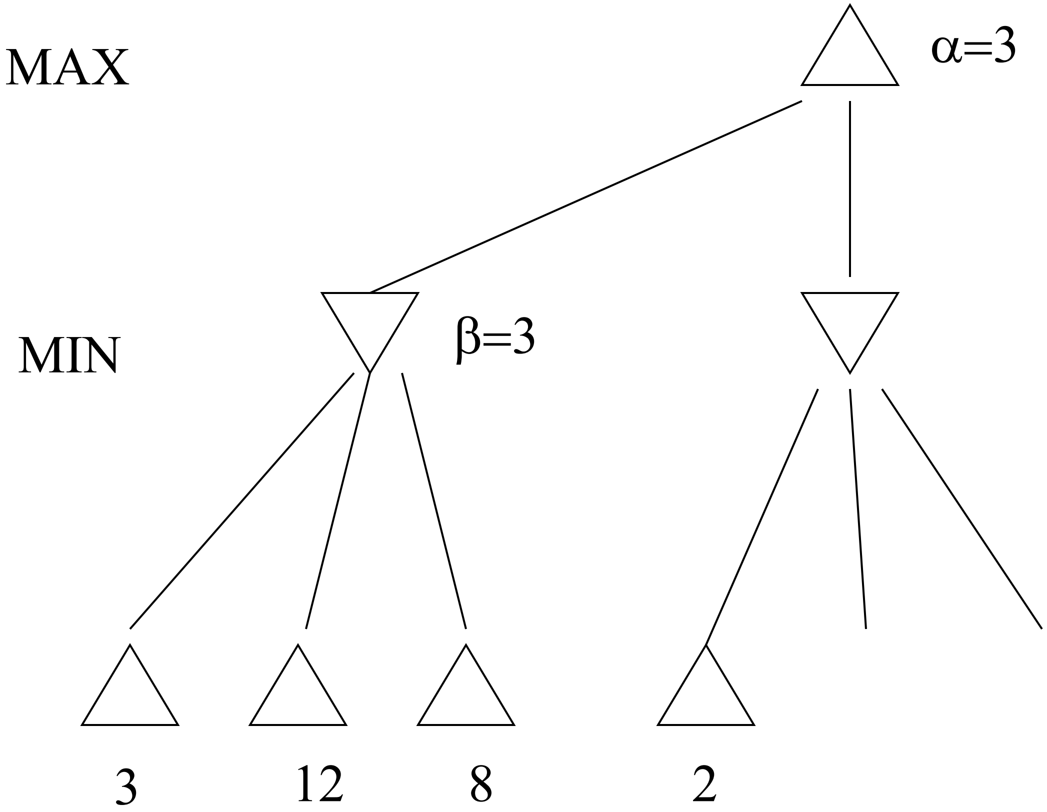
# Another alpha-beta example



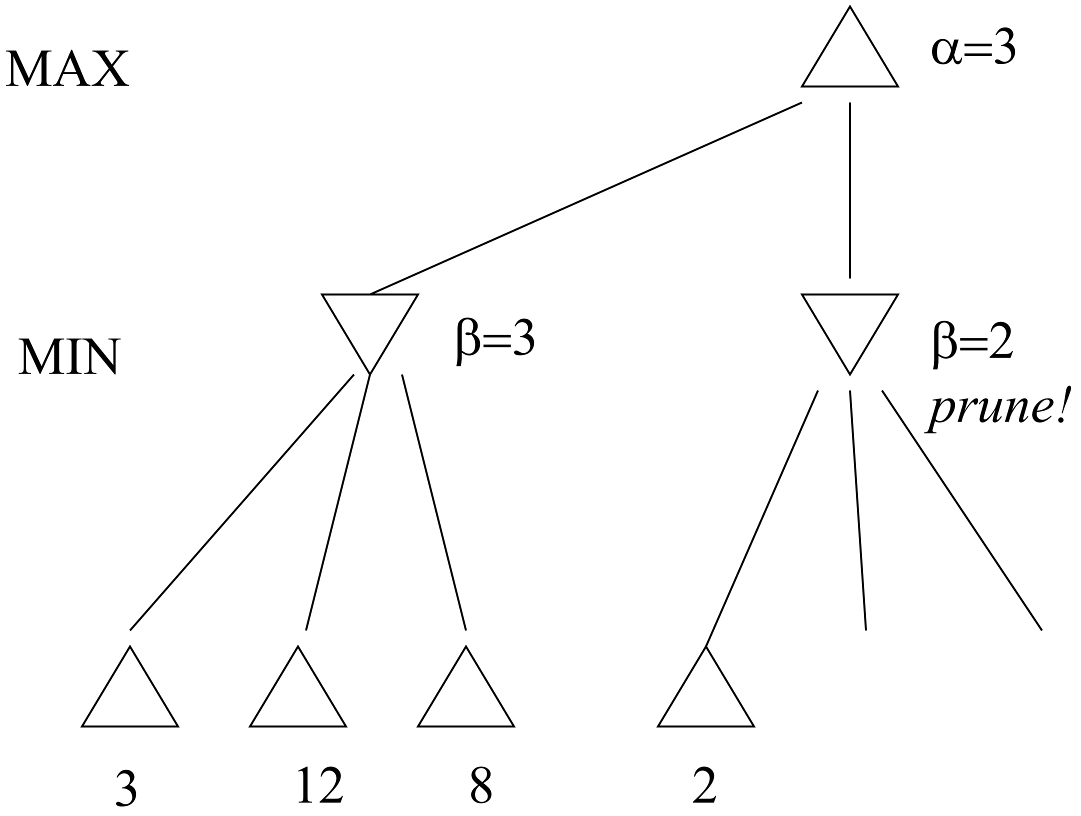
# Another alpha-beta example



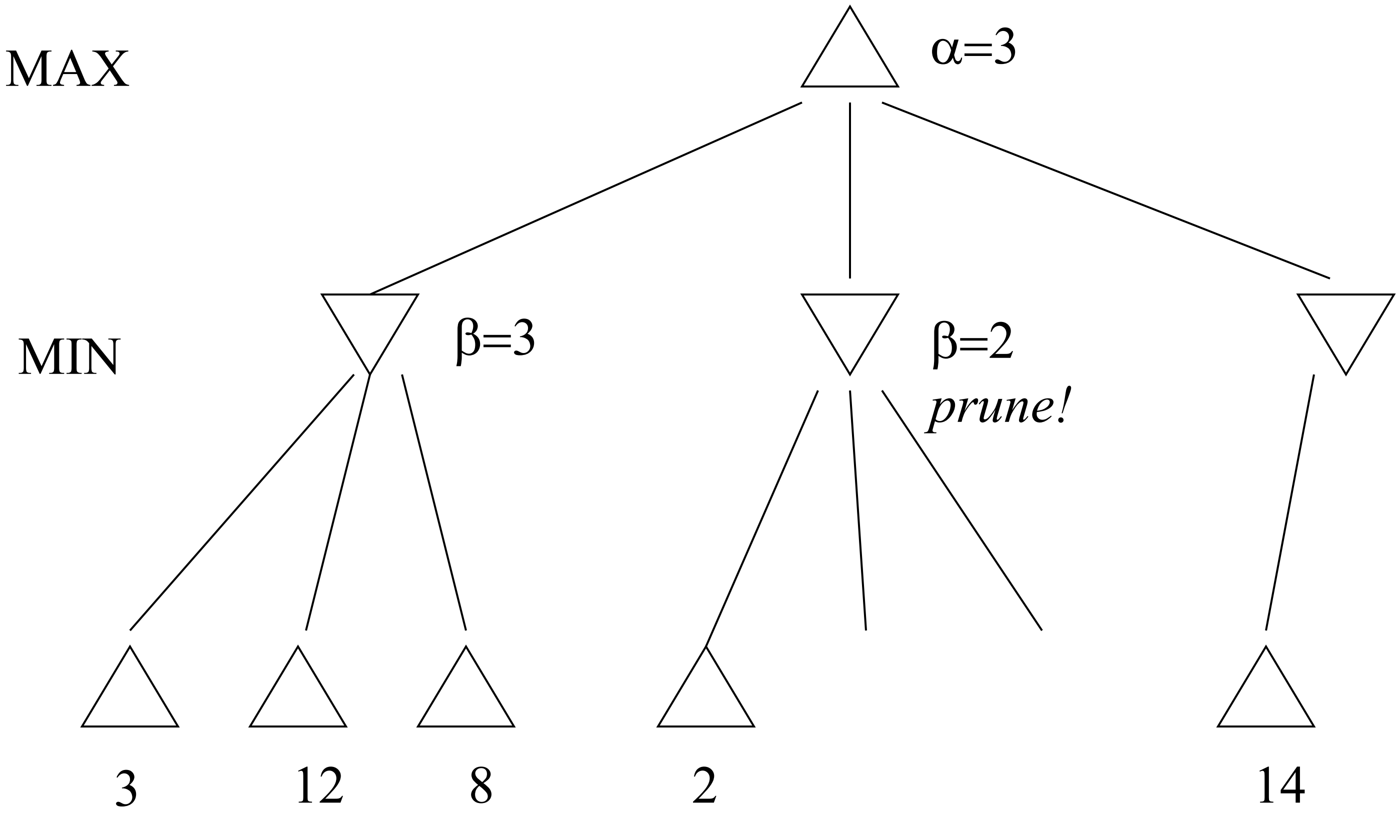
# Another alpha-beta example



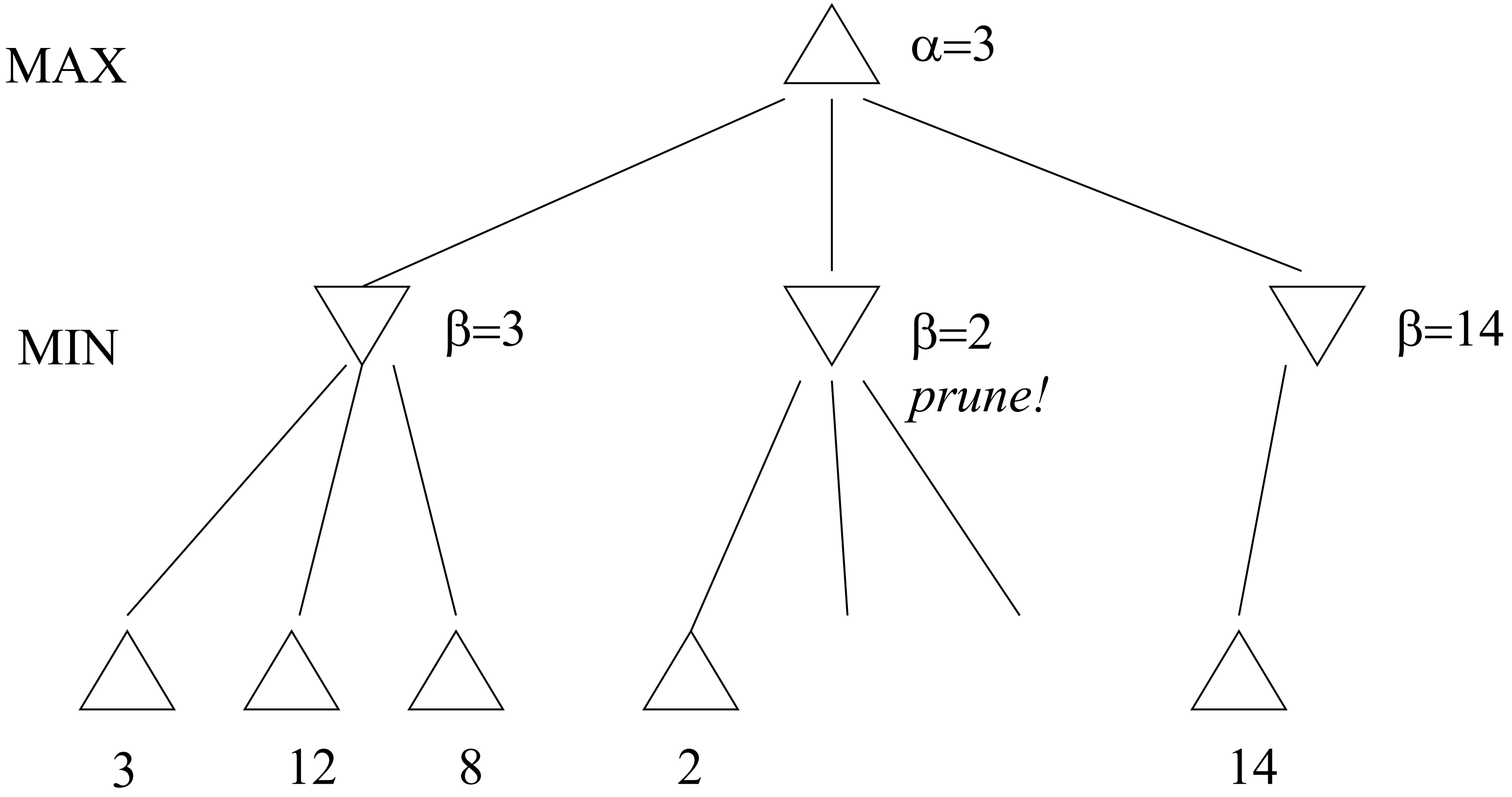
# Another alpha-beta example



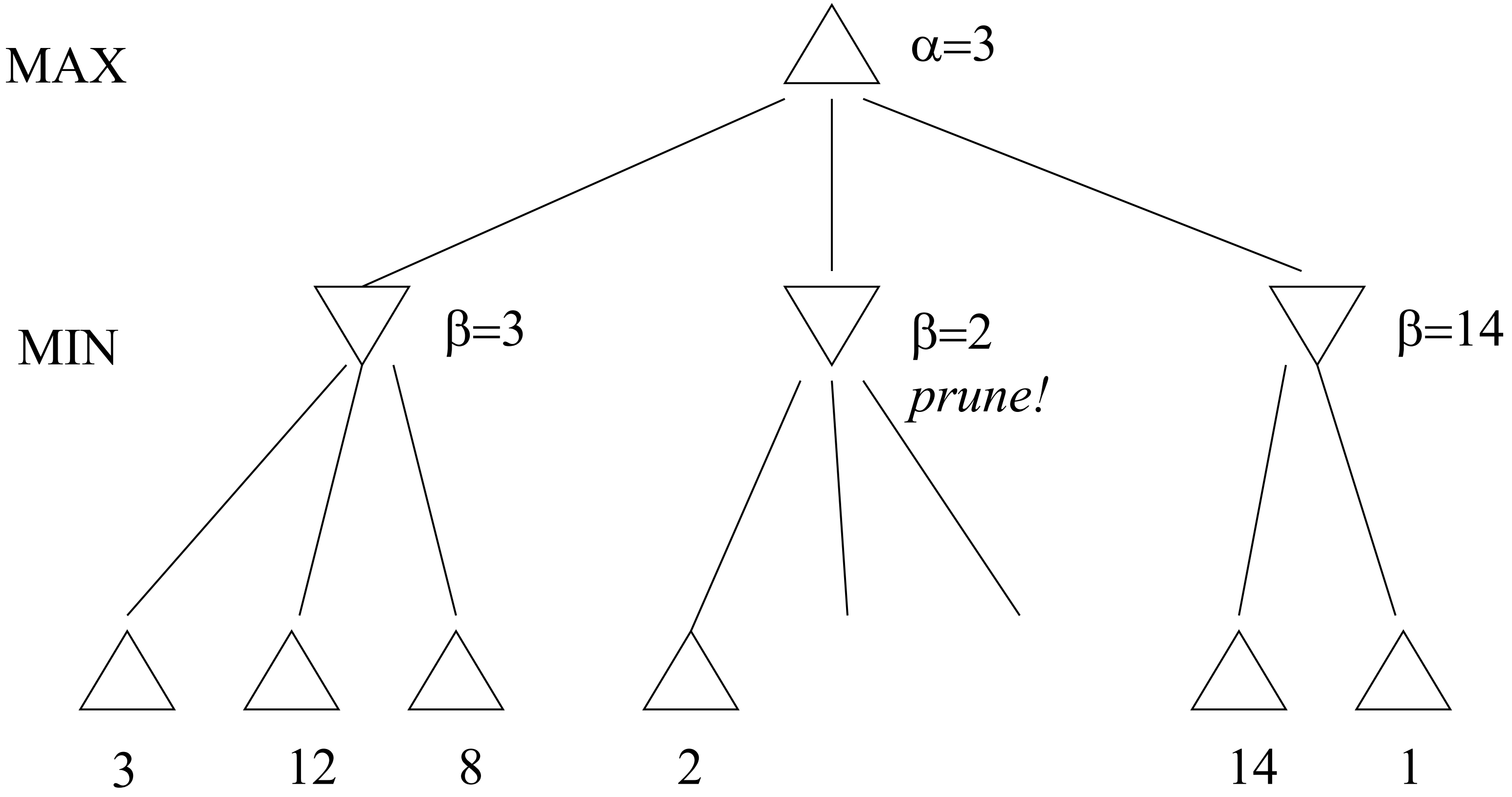
# Another alpha-beta example



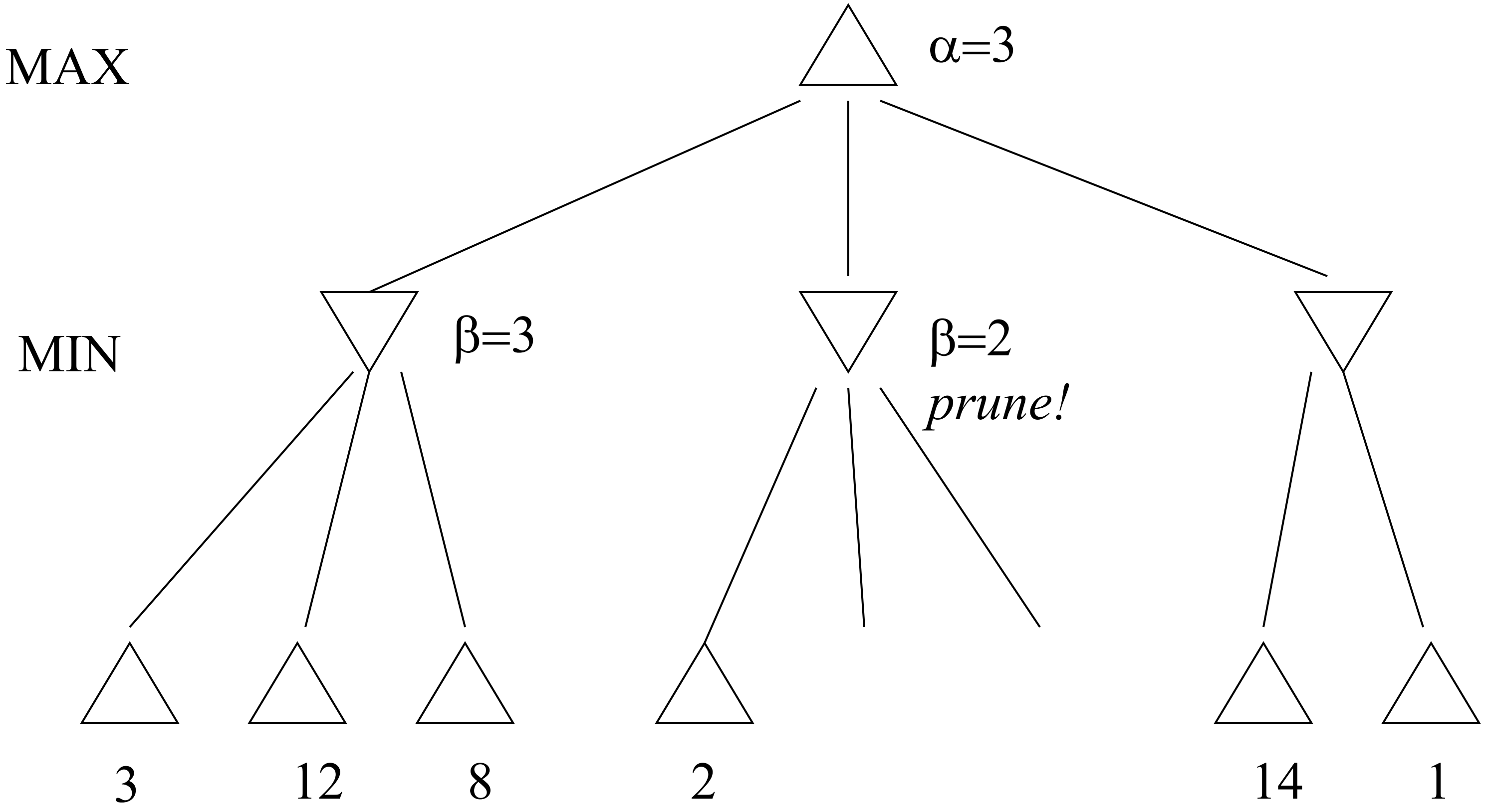
# Another alpha-beta example



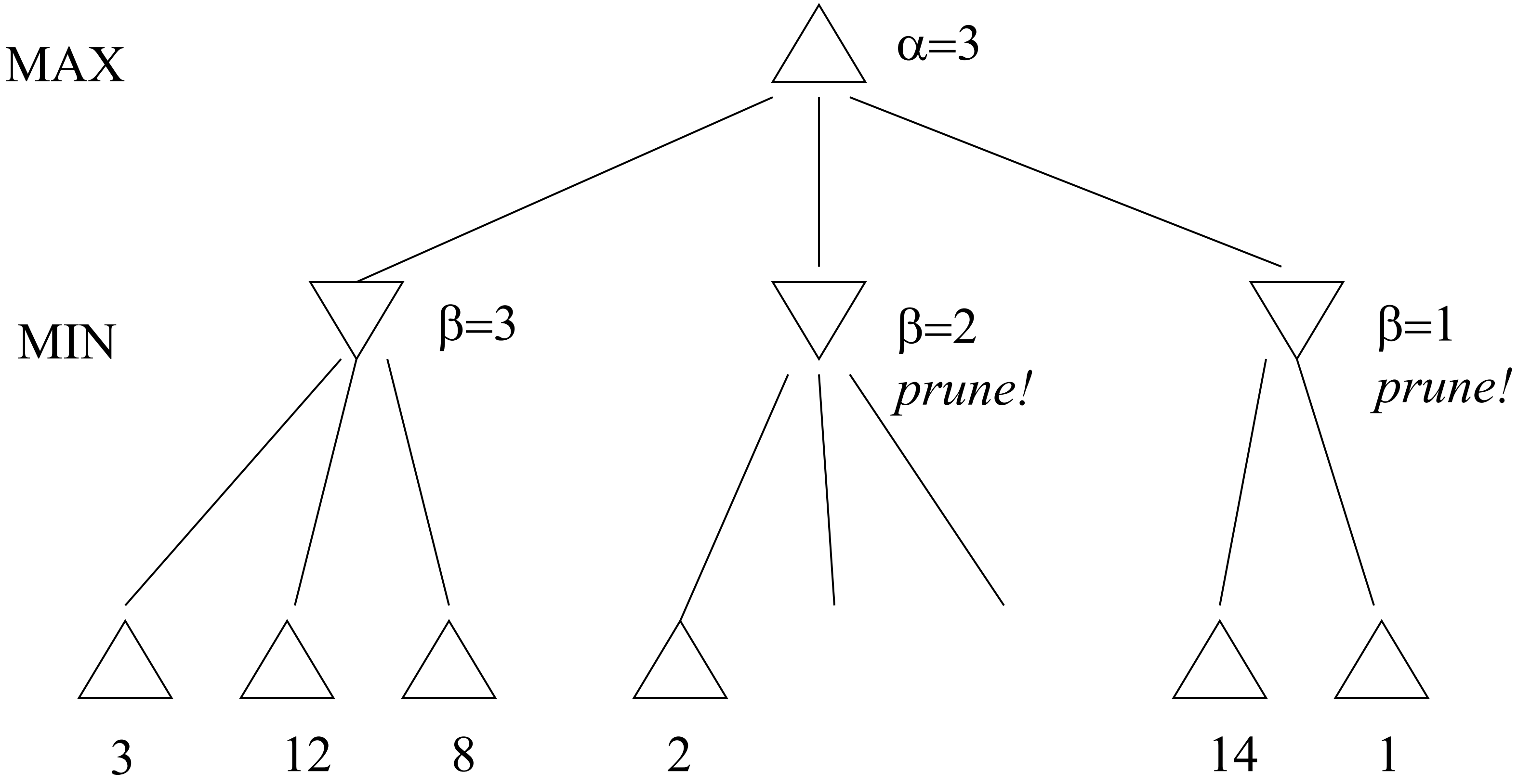
# Another alpha-beta example



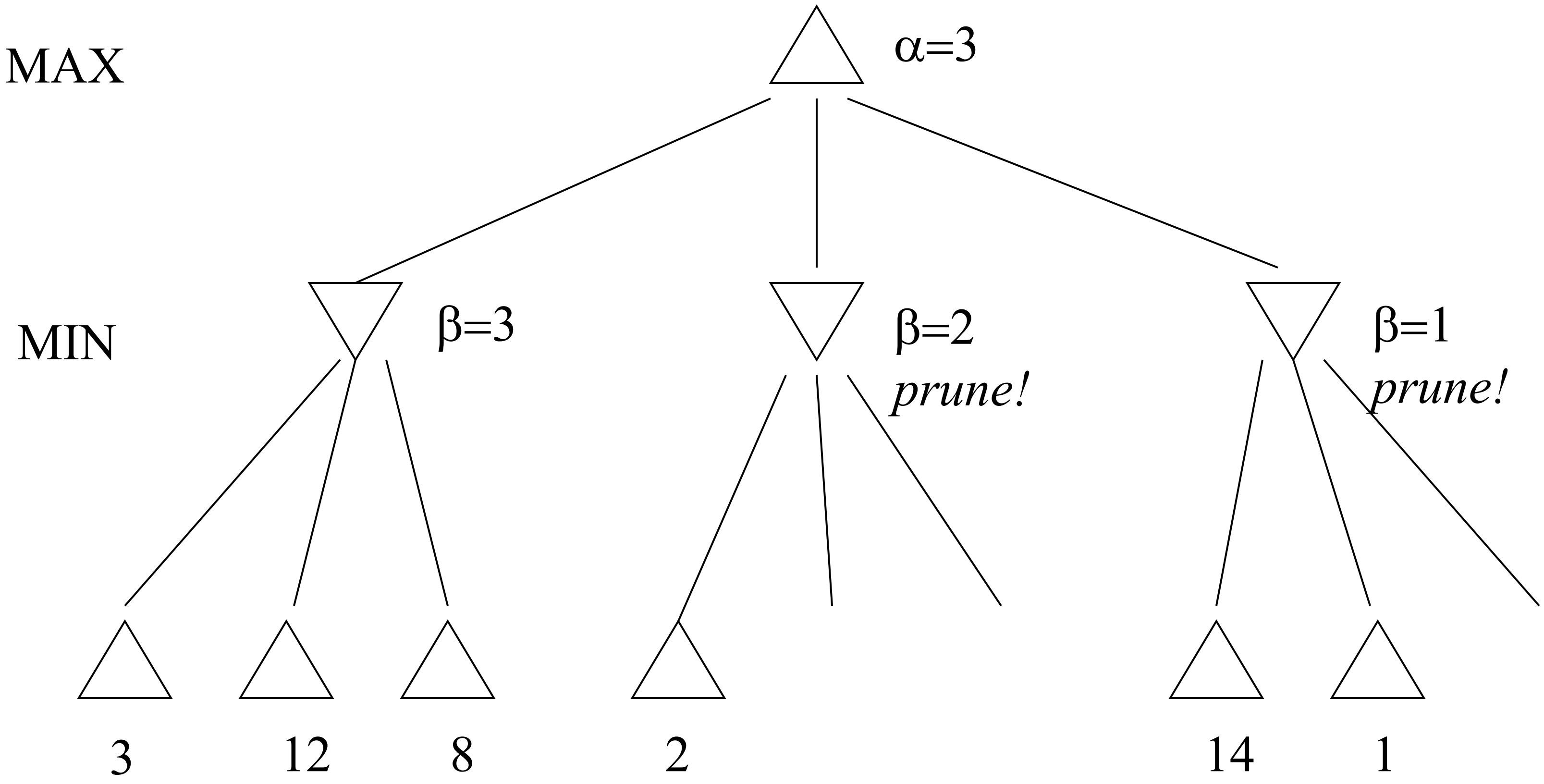
# Another alpha-beta example



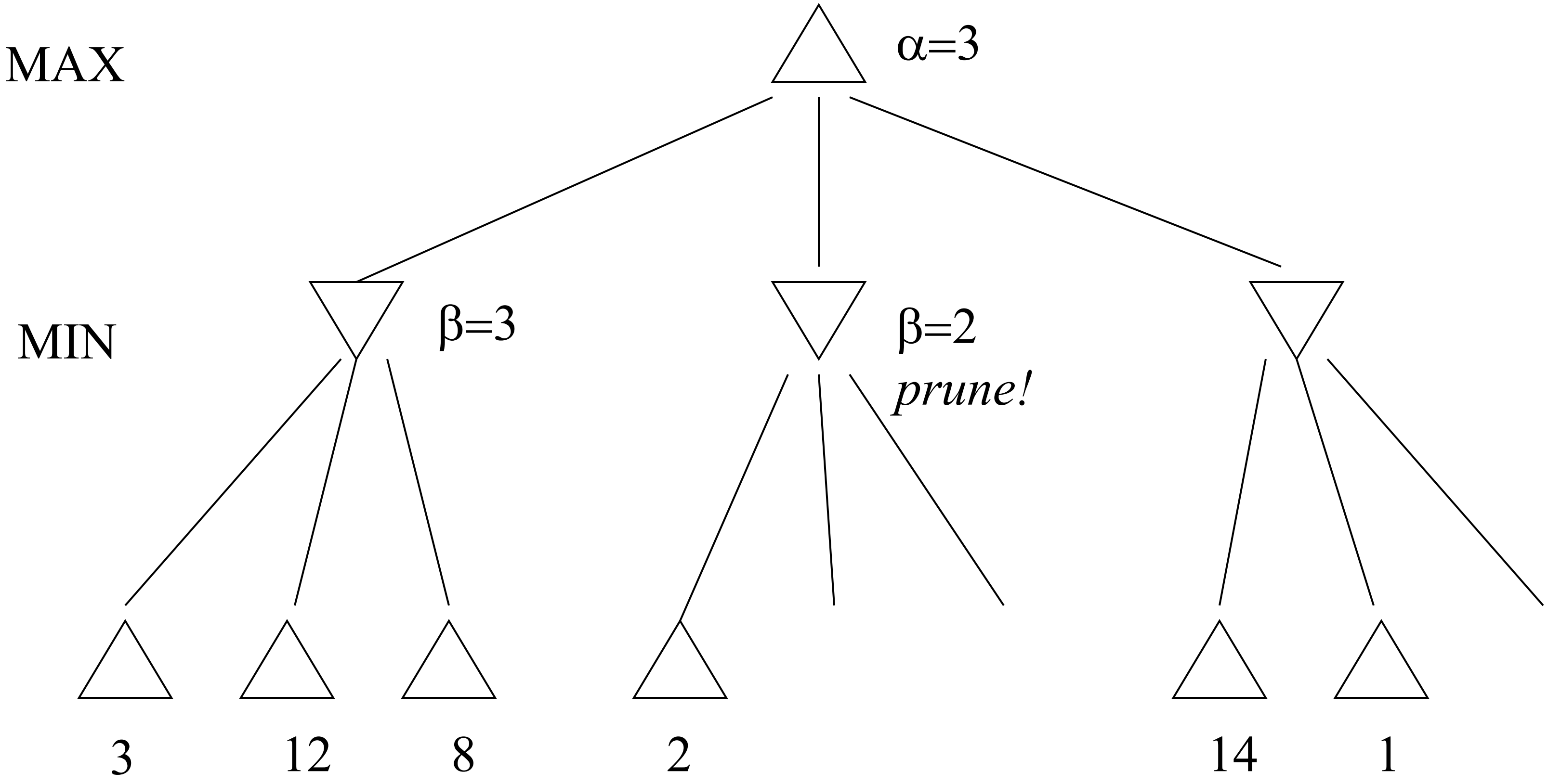
# Another alpha-beta example



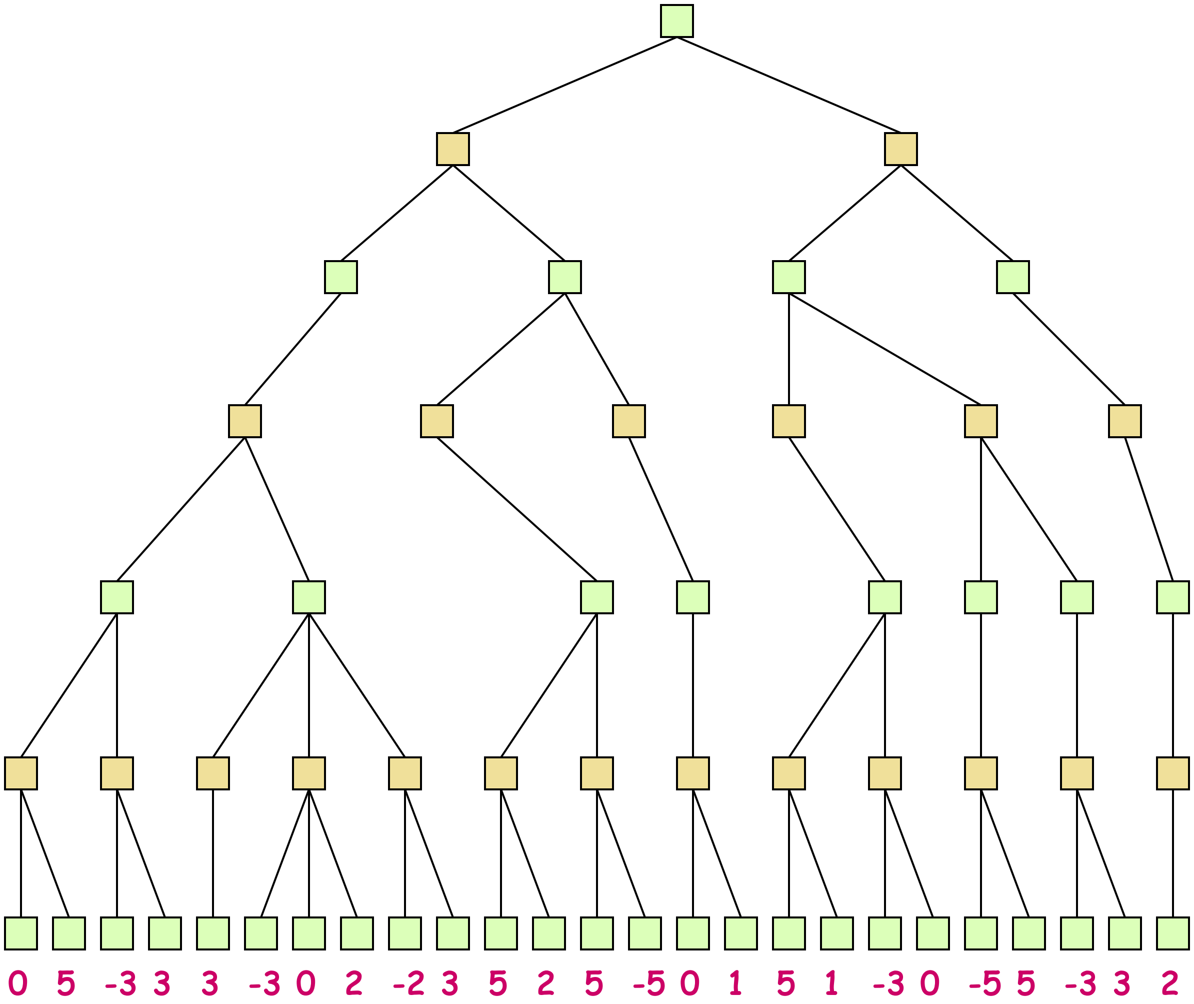
# Another alpha-beta example

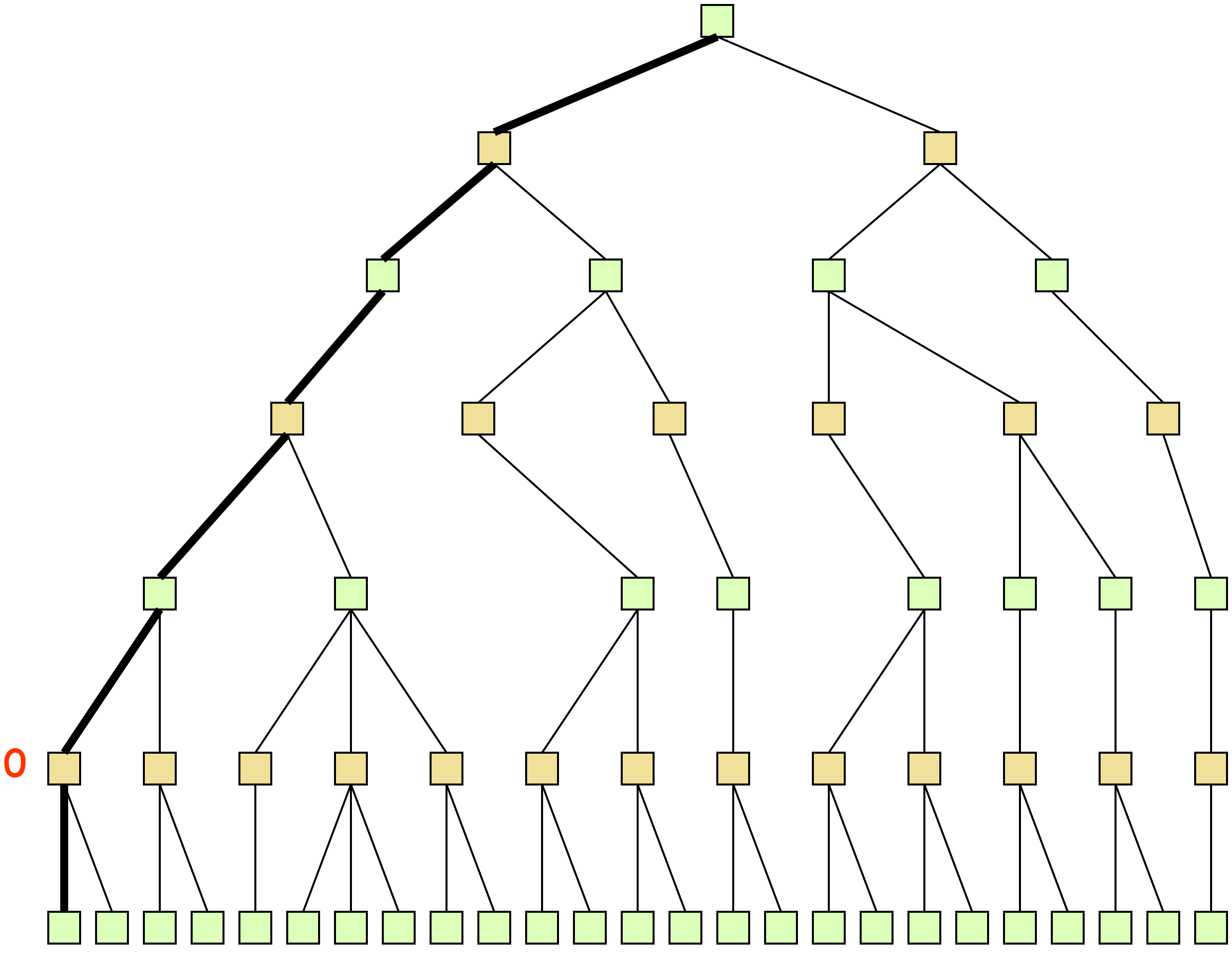


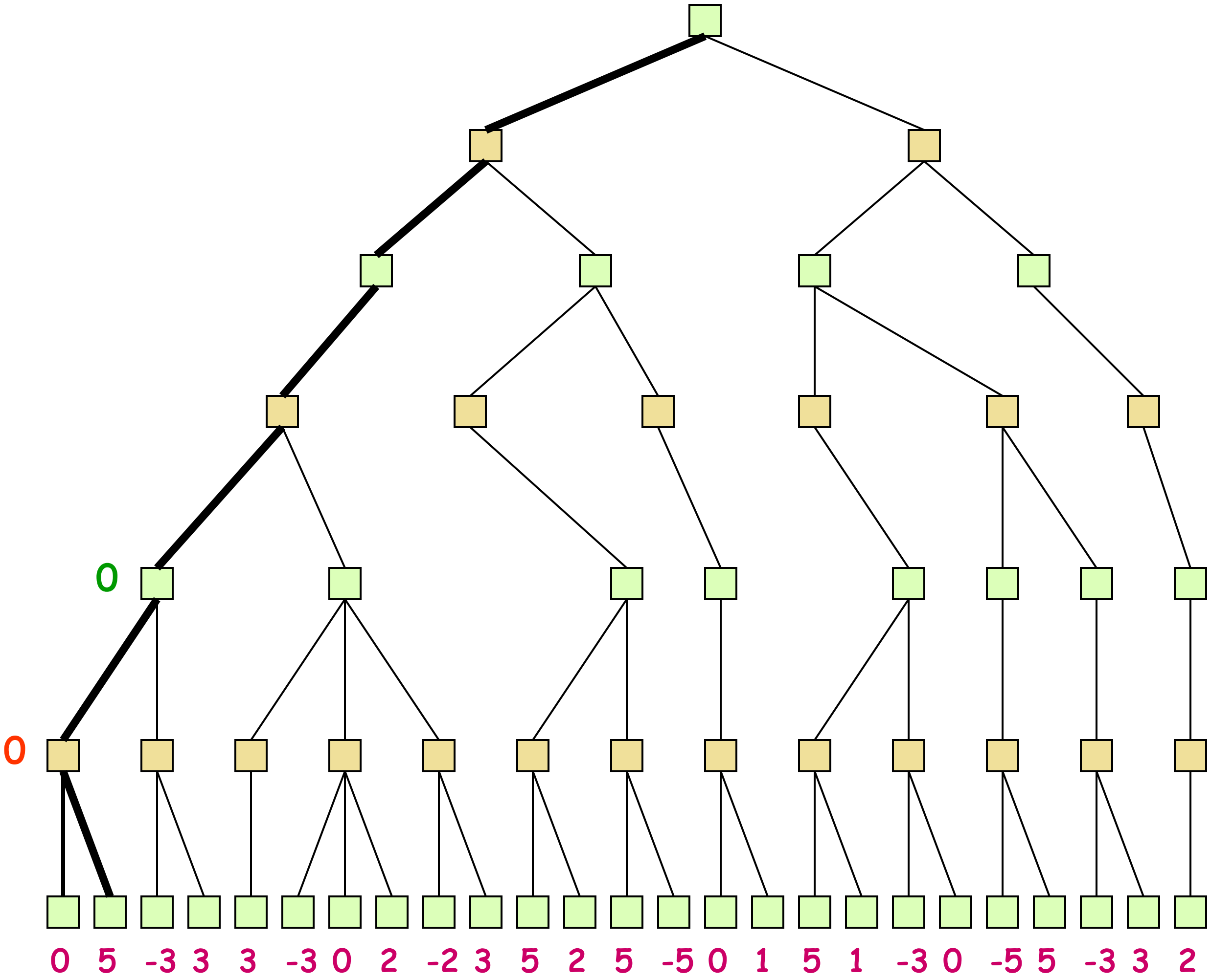
# Another alpha-beta example

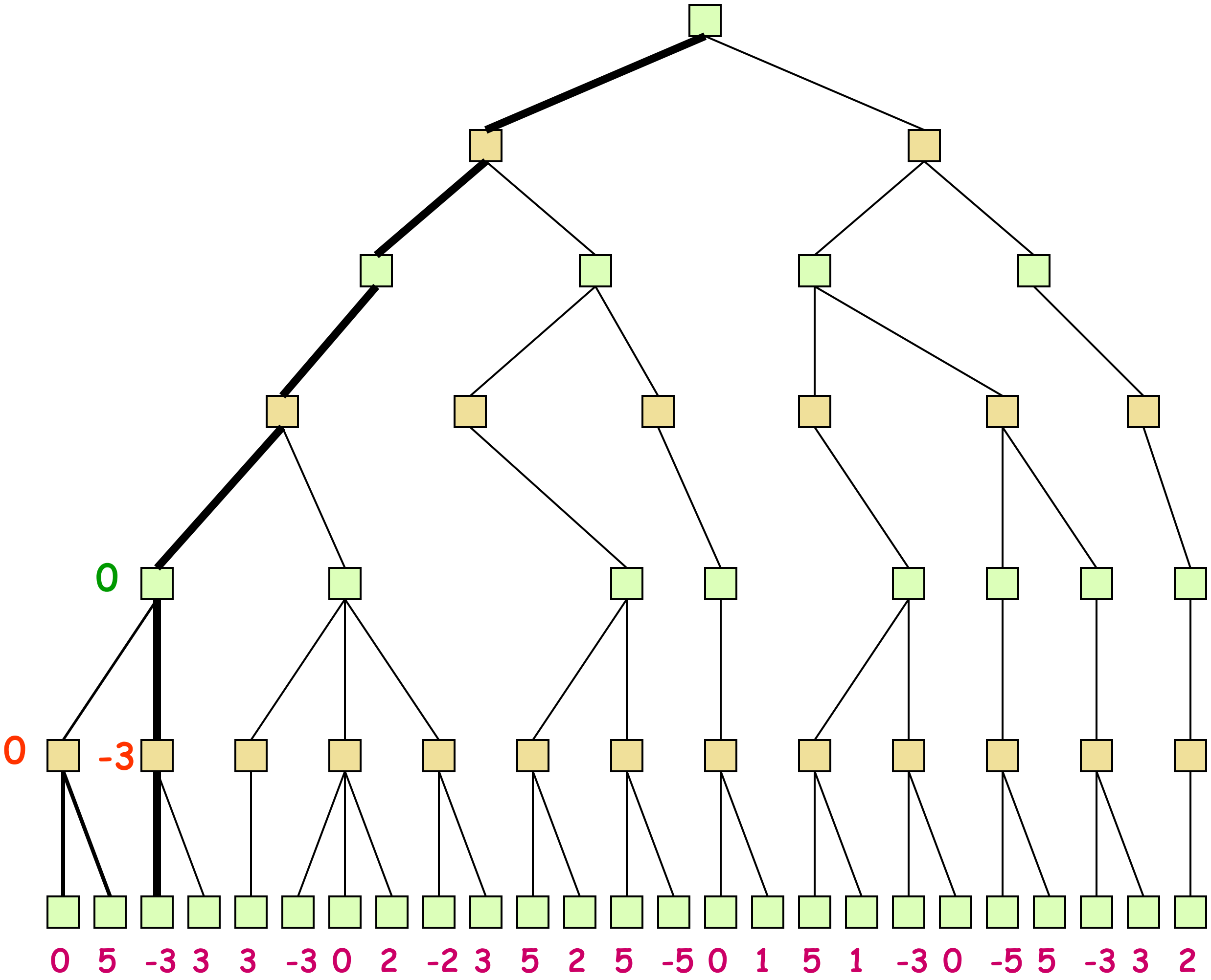


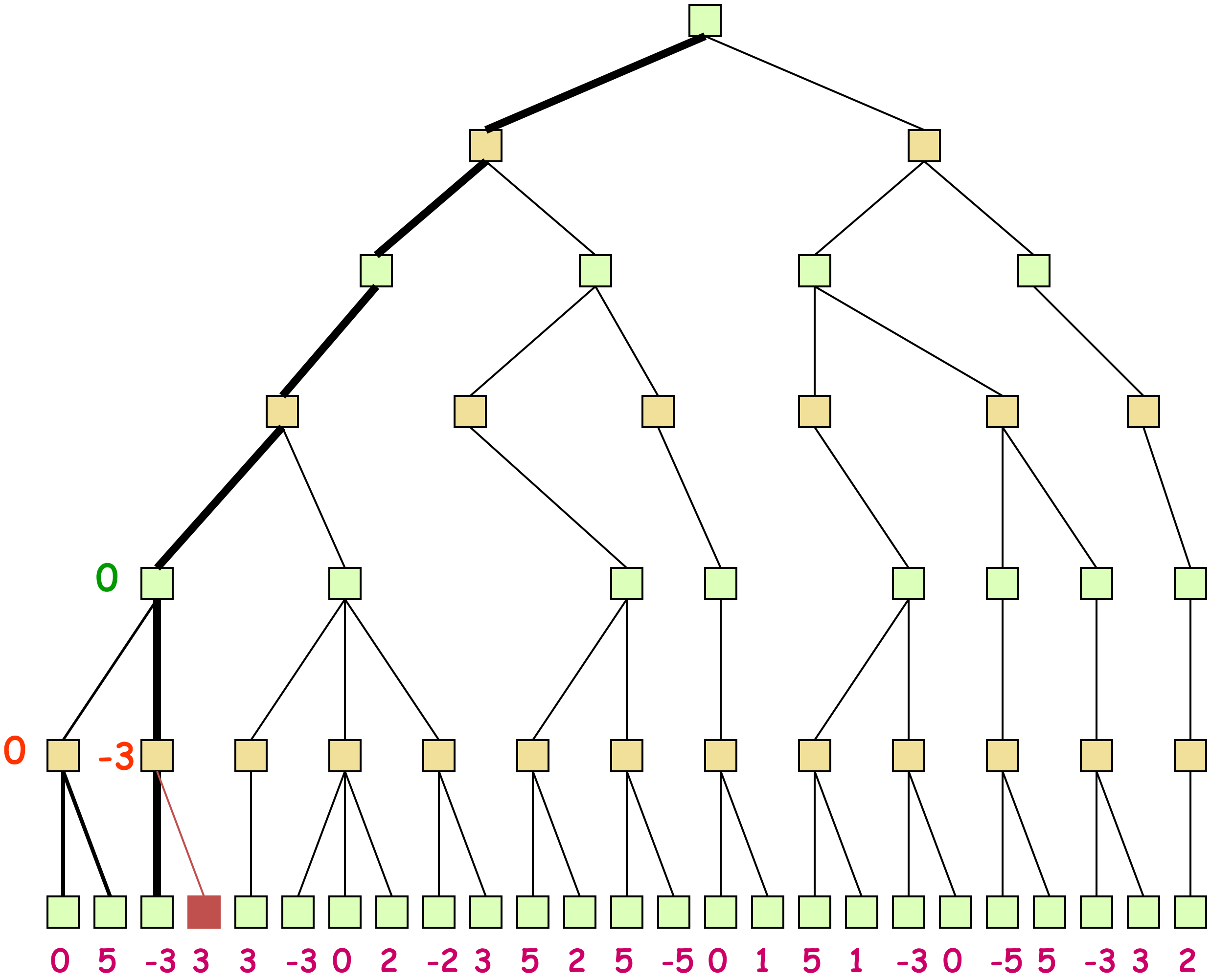
# Alpha-Beta Tic-Tac-Toe Example 2

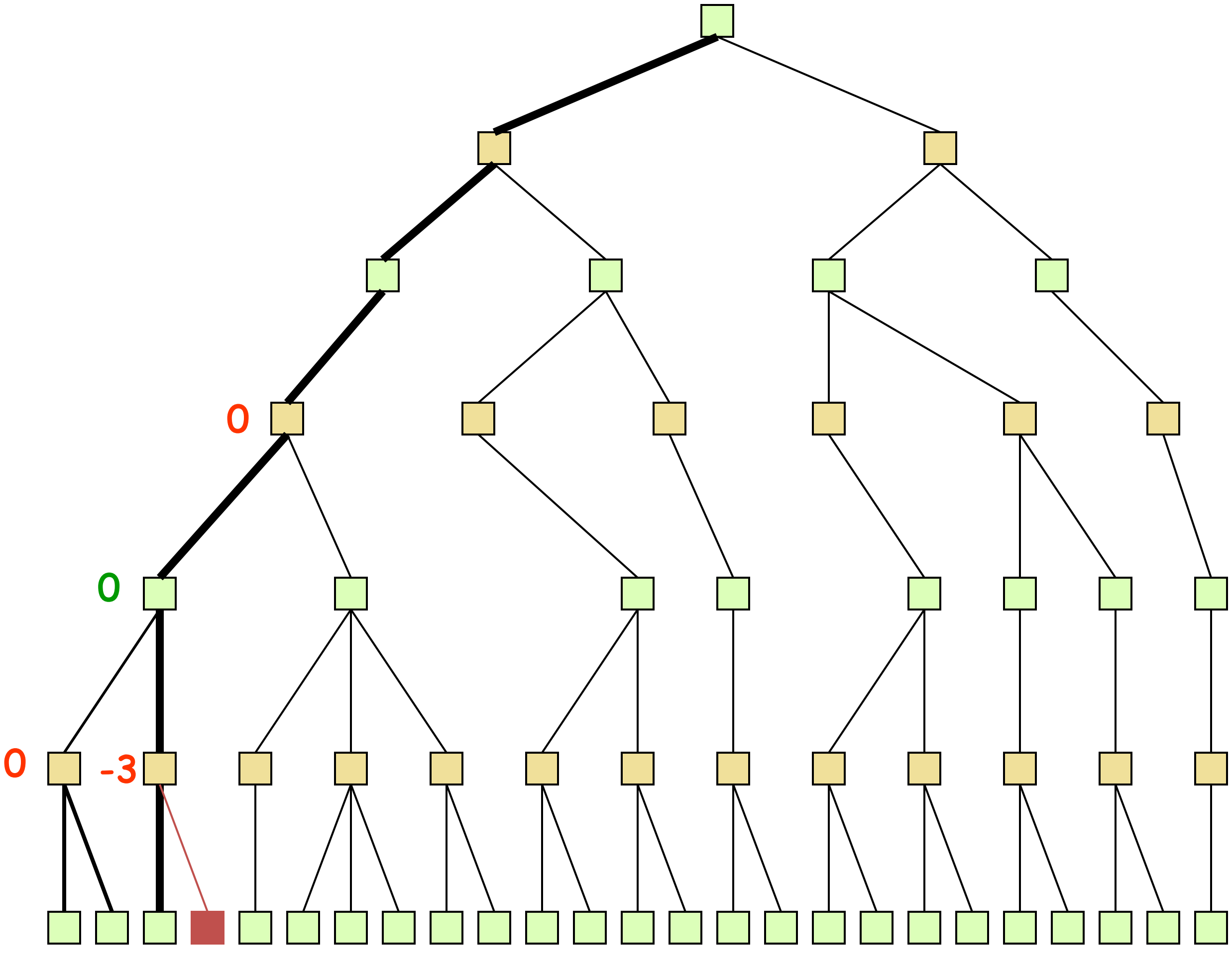


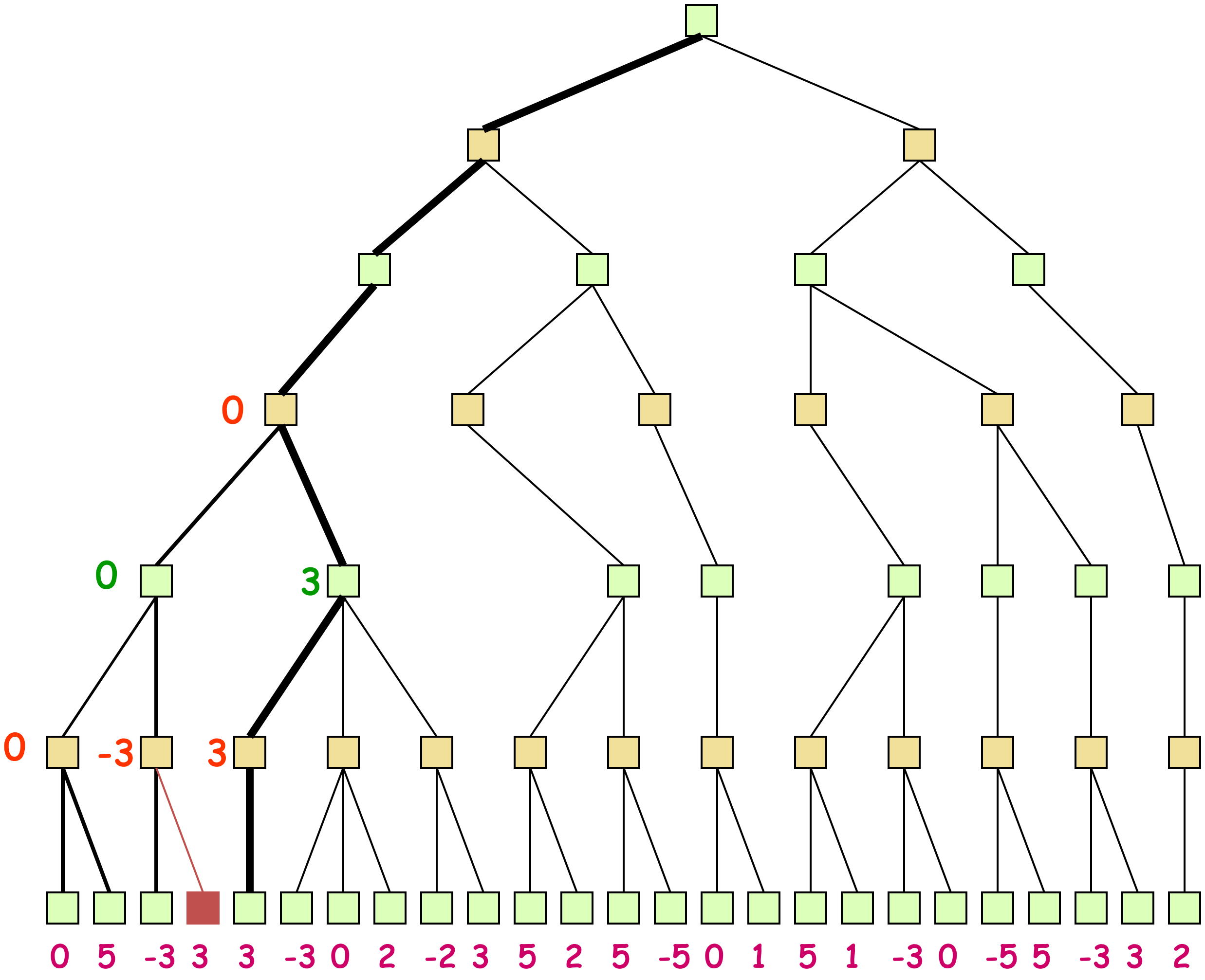

0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

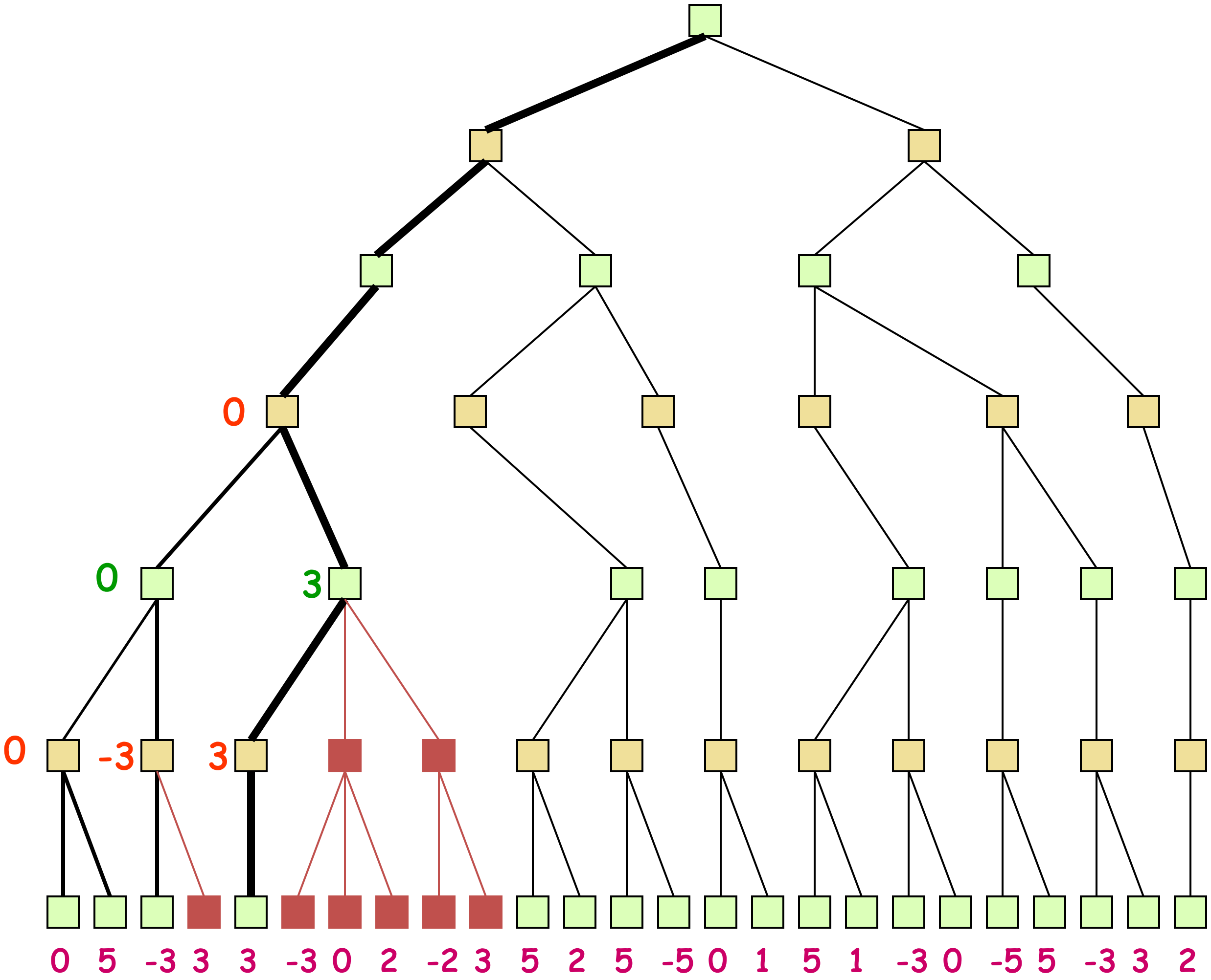

0
0
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

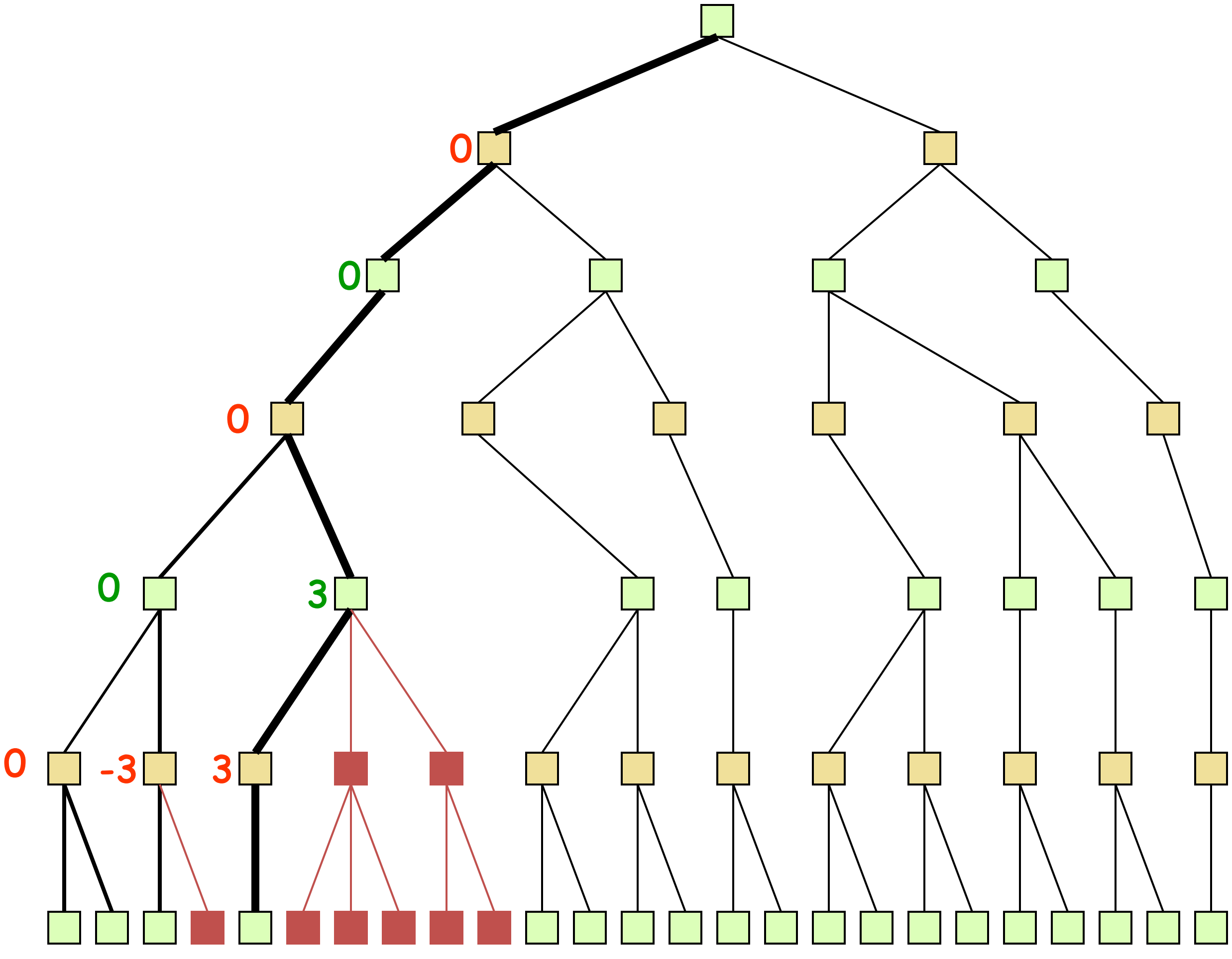

0
0
-3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

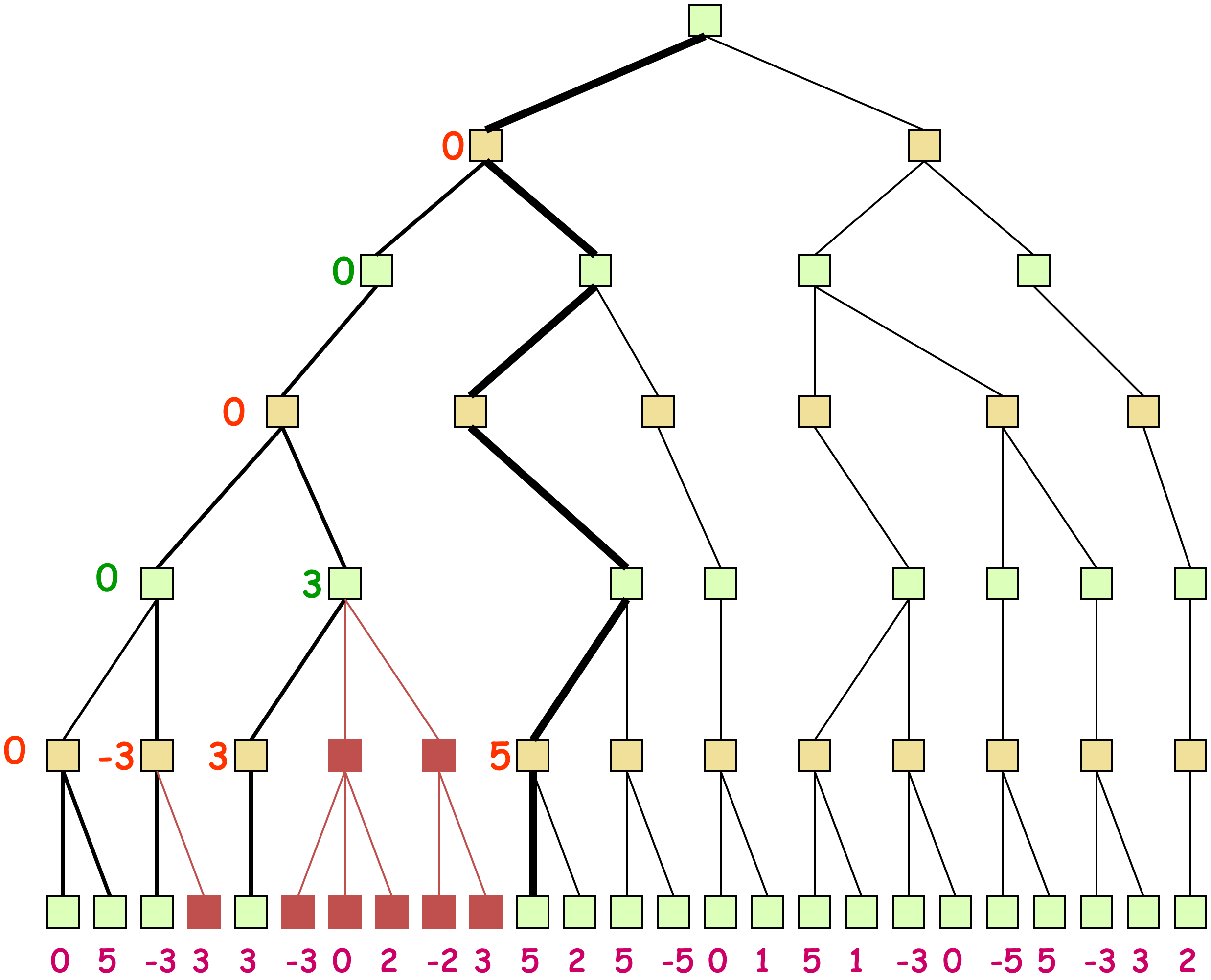

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

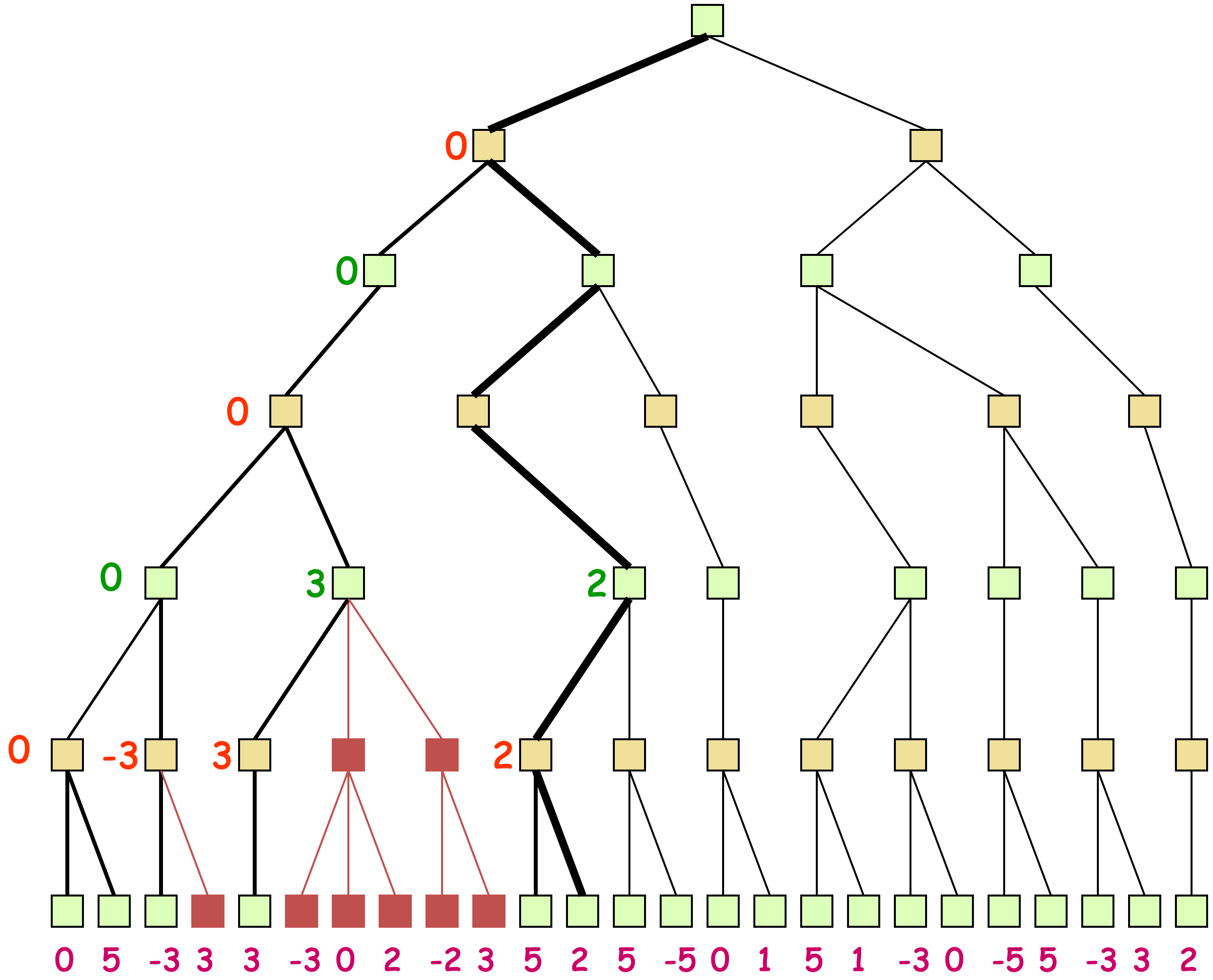

0
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

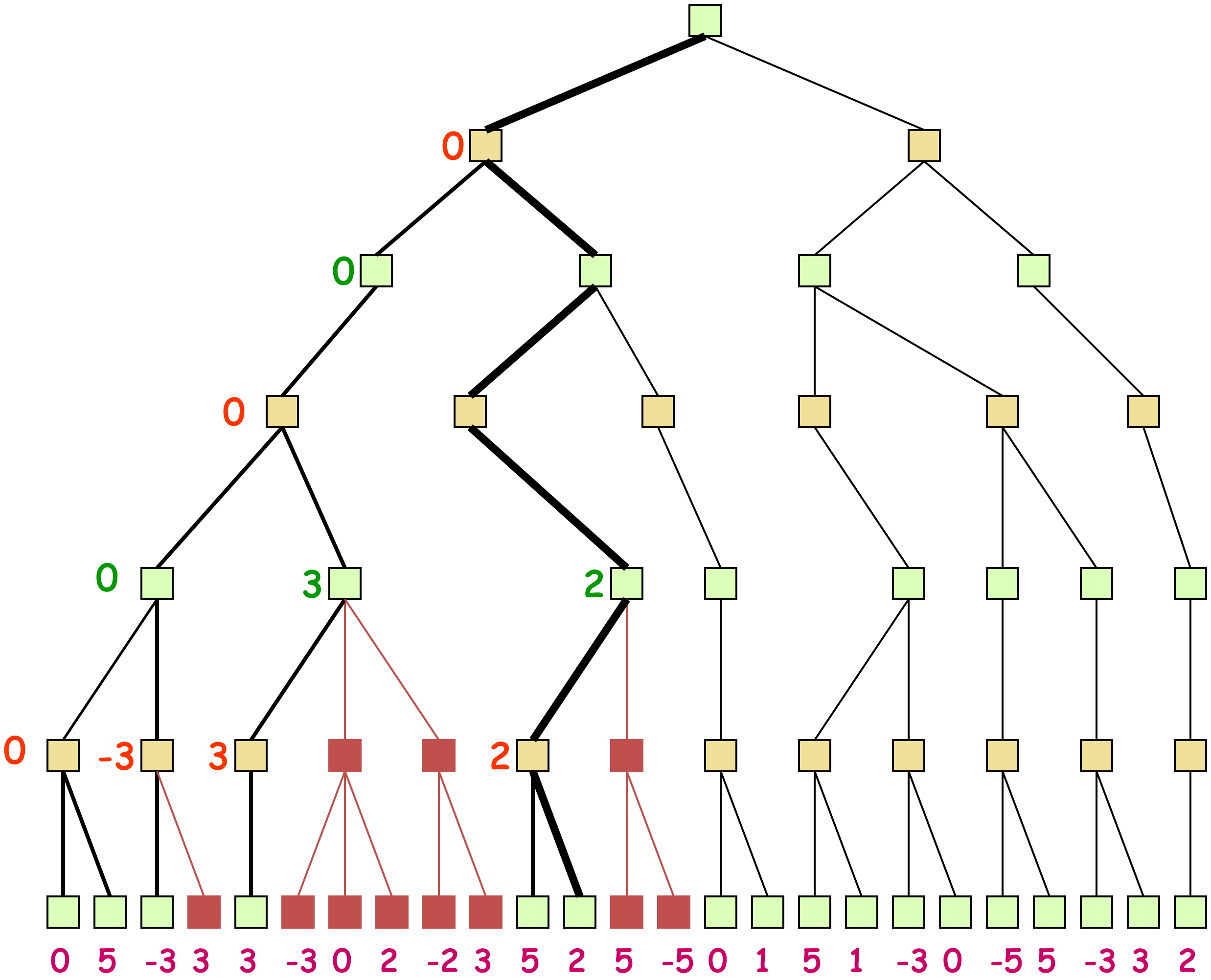

0
0
3
0
-3
3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

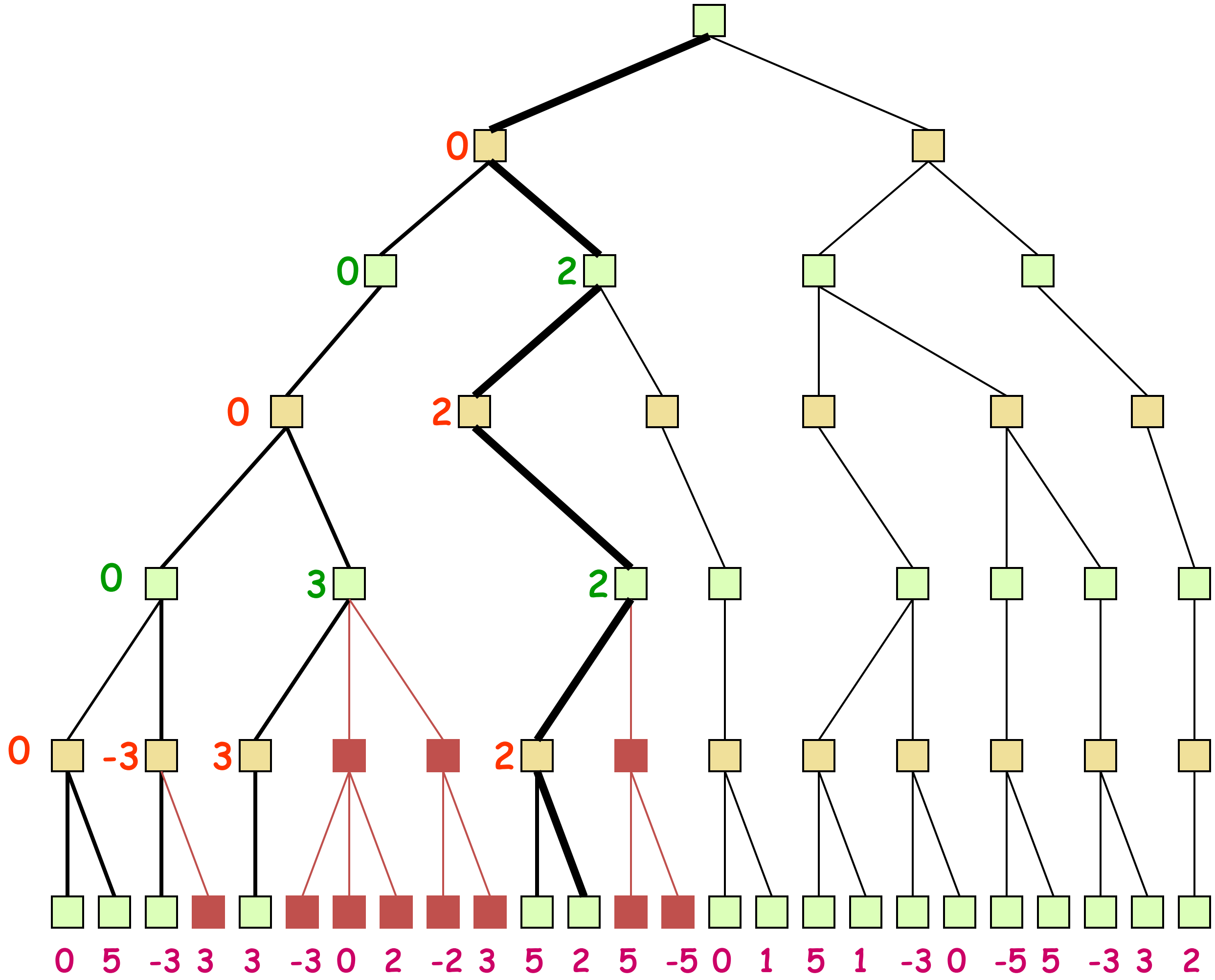

0
0
3
0
-3
3
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

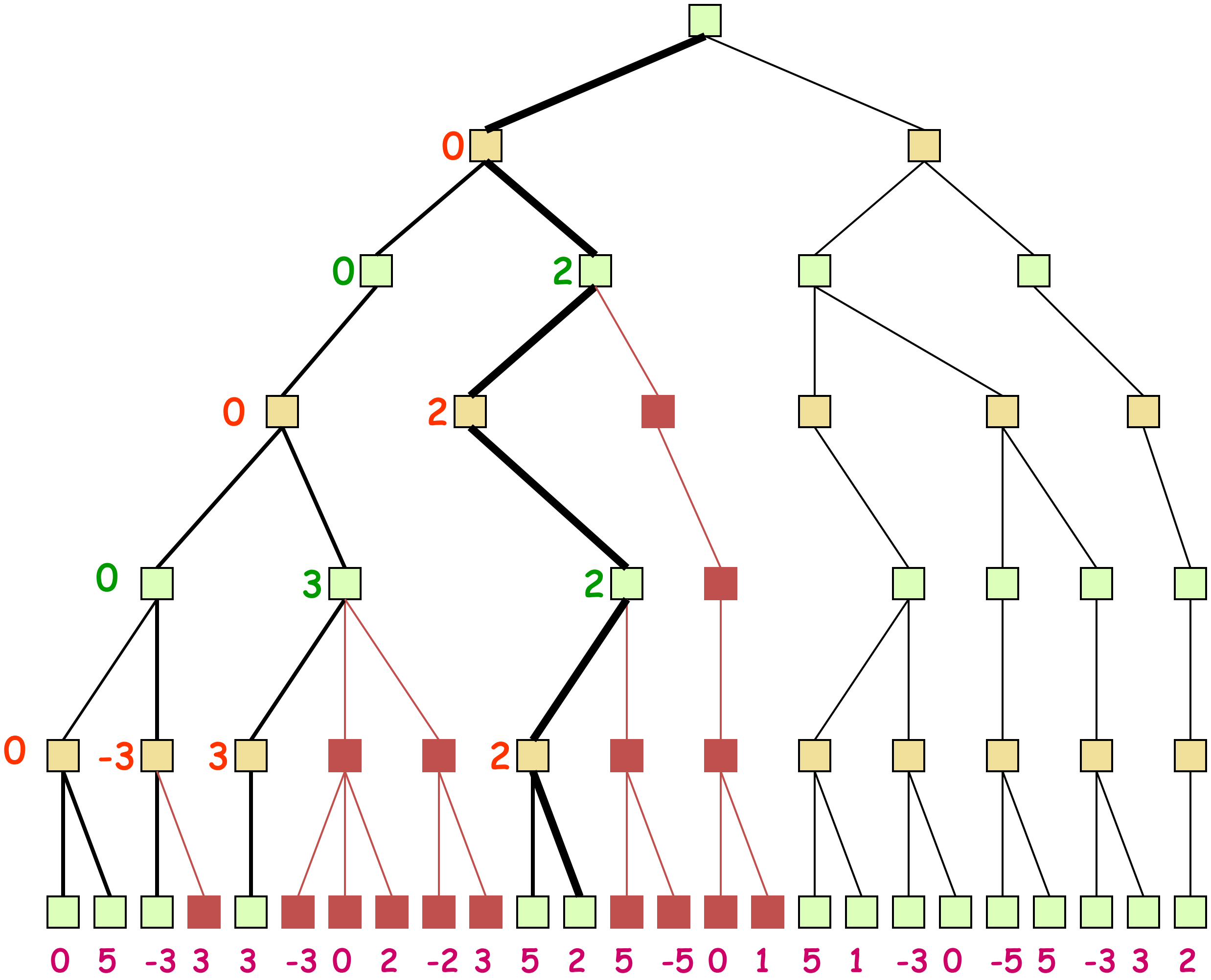

0
0
0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

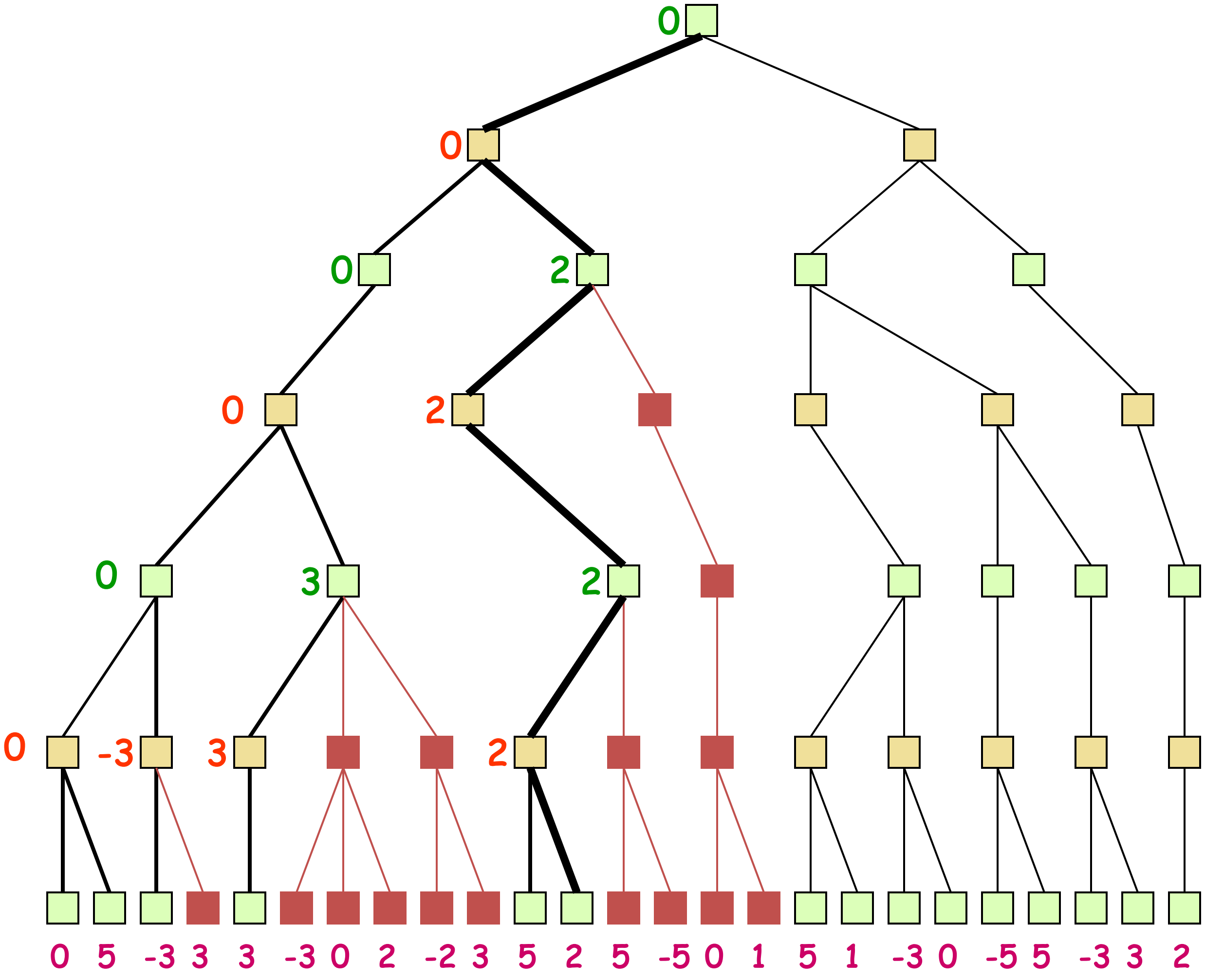

0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

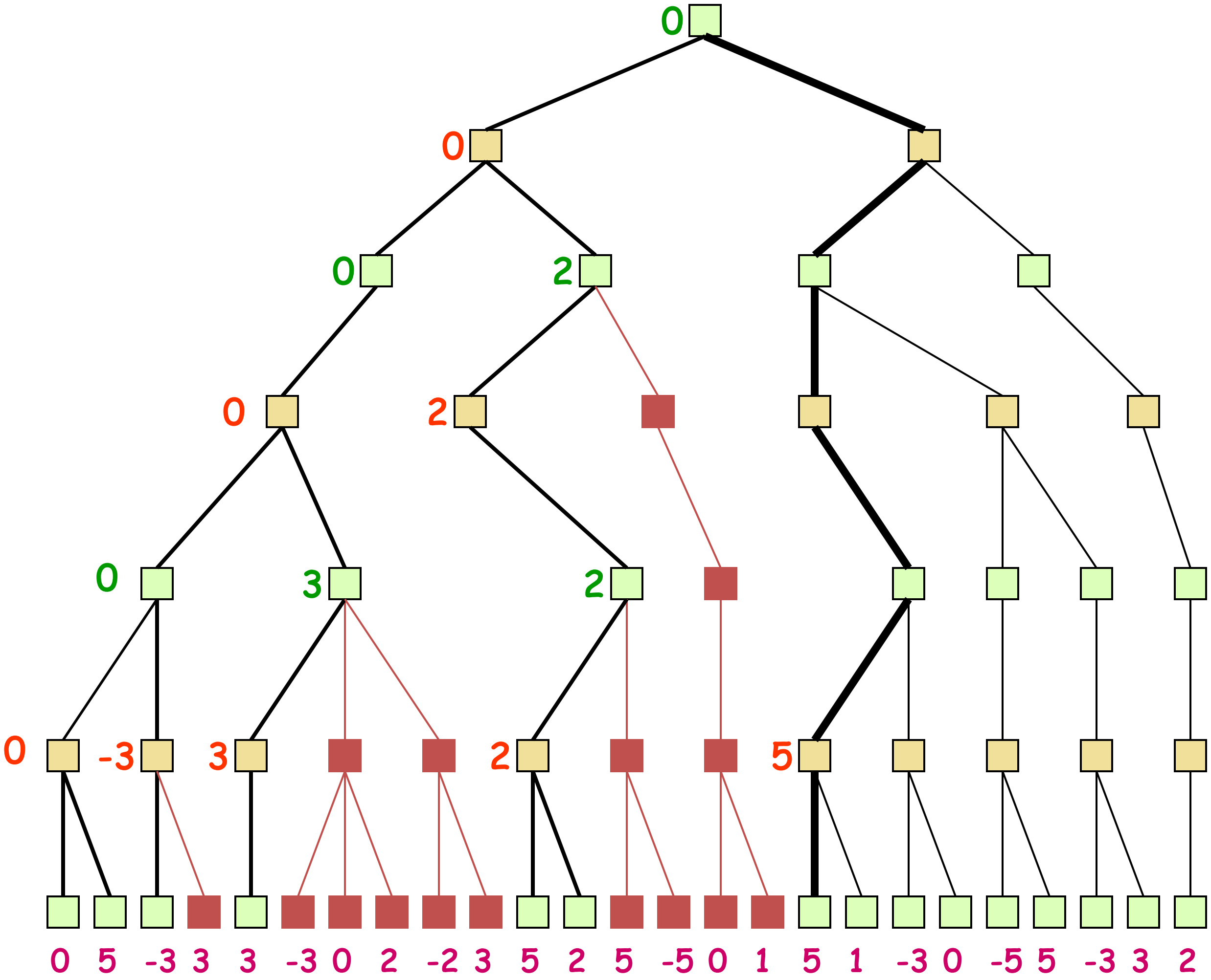

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

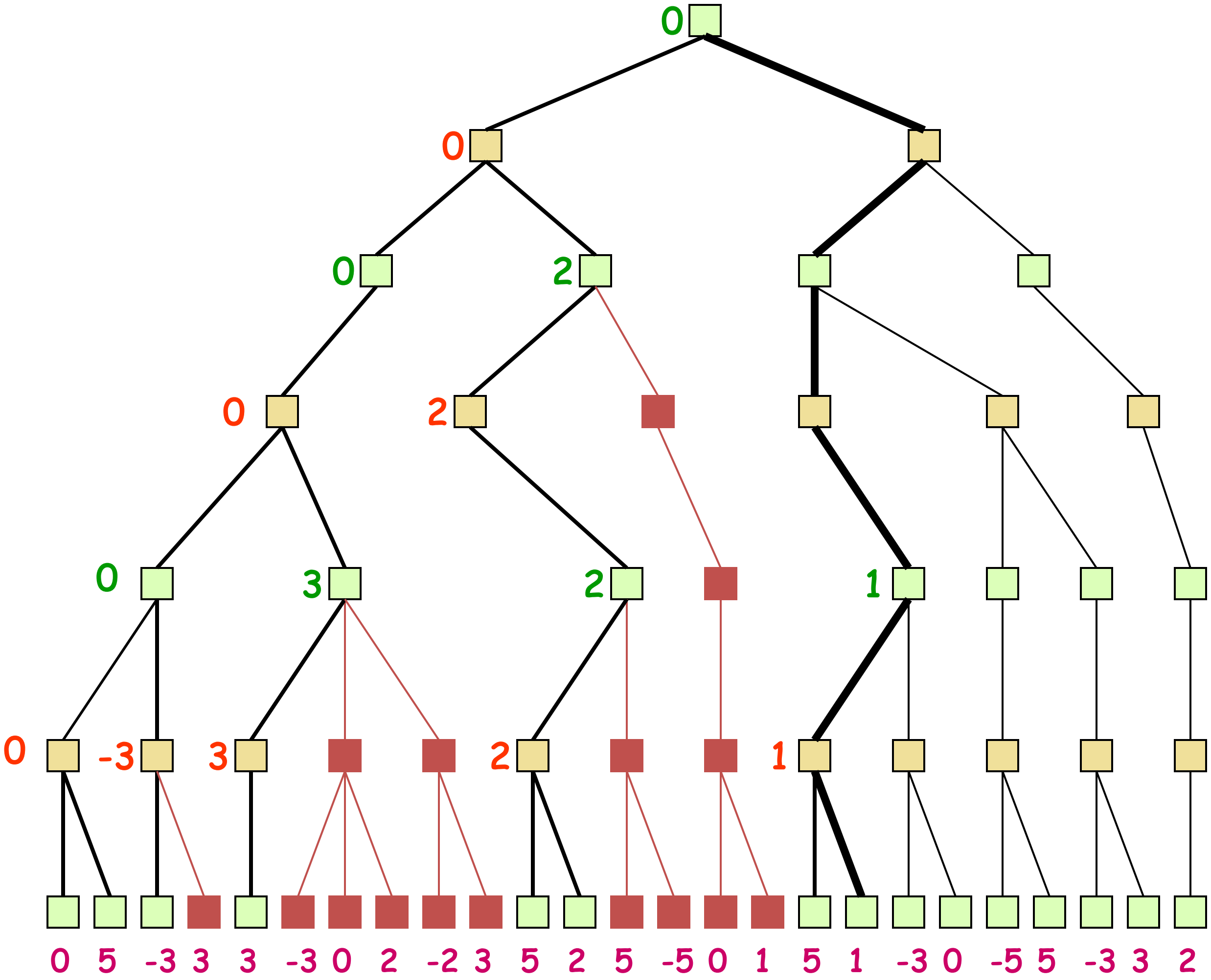

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

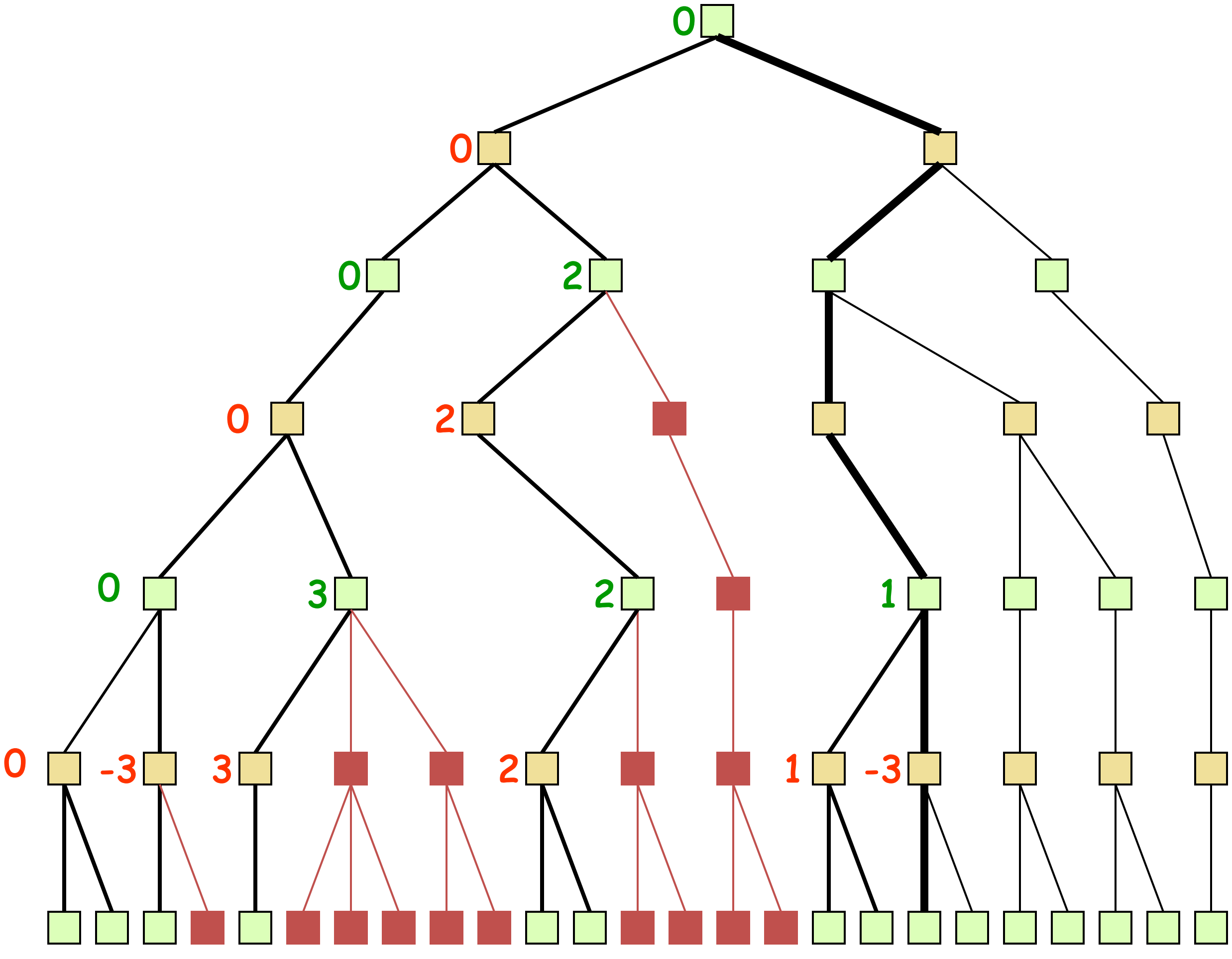

0
0
2
0
2
0
3
2
0
-3
3
2
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

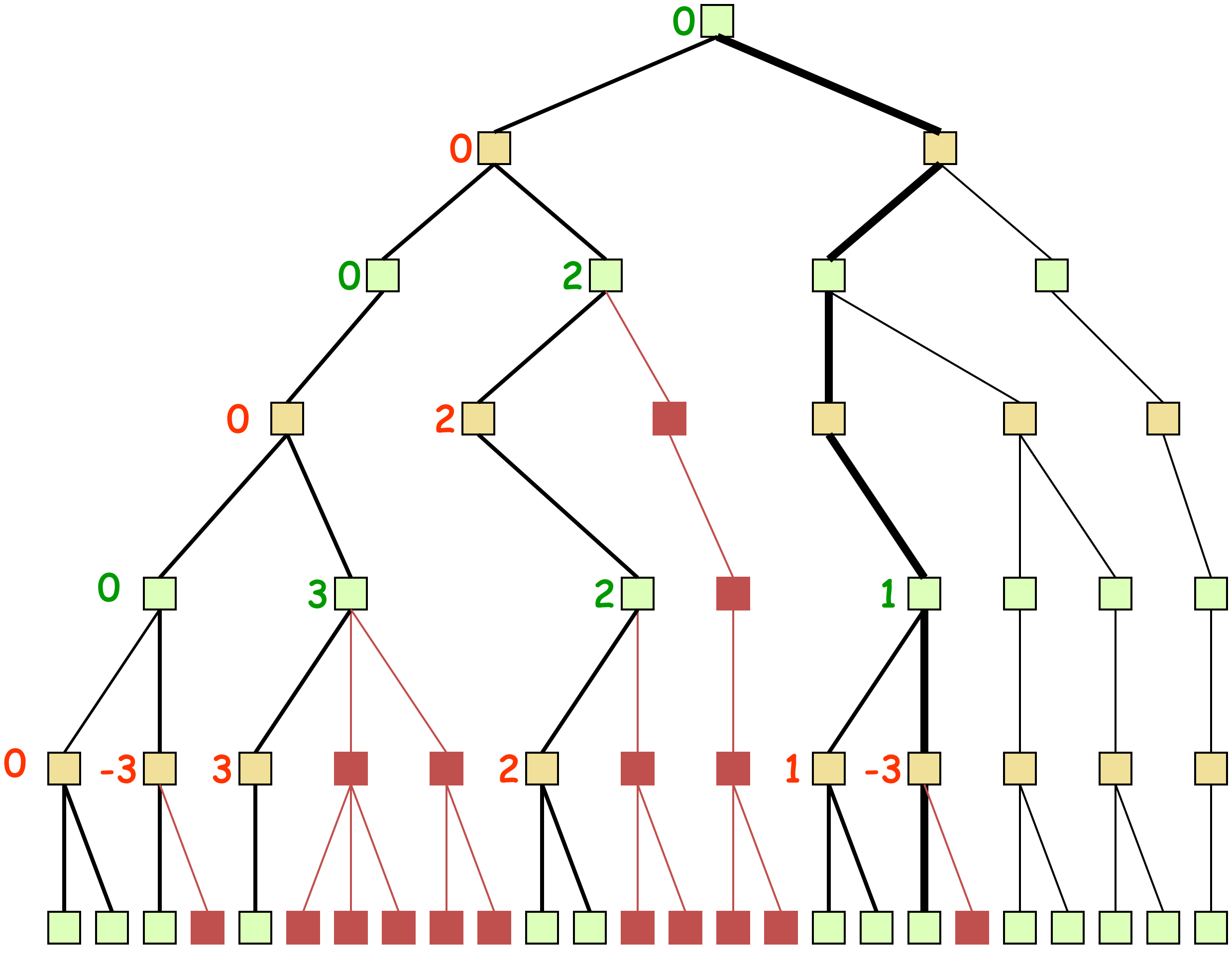

0
0
2
0
2
0
3
2
0
-3
3
2
0
5
-3
3
3
-3
0
2
-2
3
5
2
5
-5
0
1
5
1
-3
0
-5
5
-3
3
2

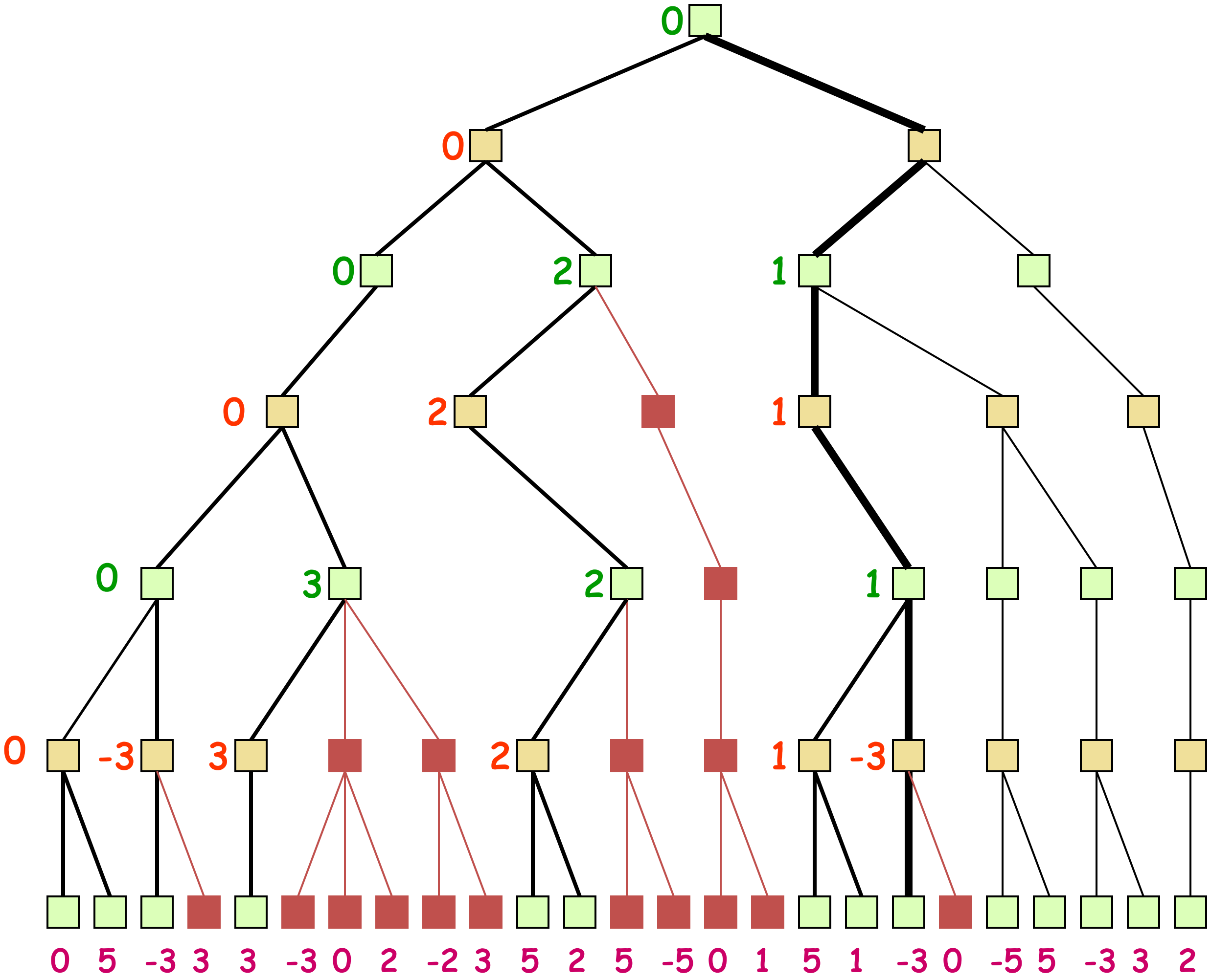

0
0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

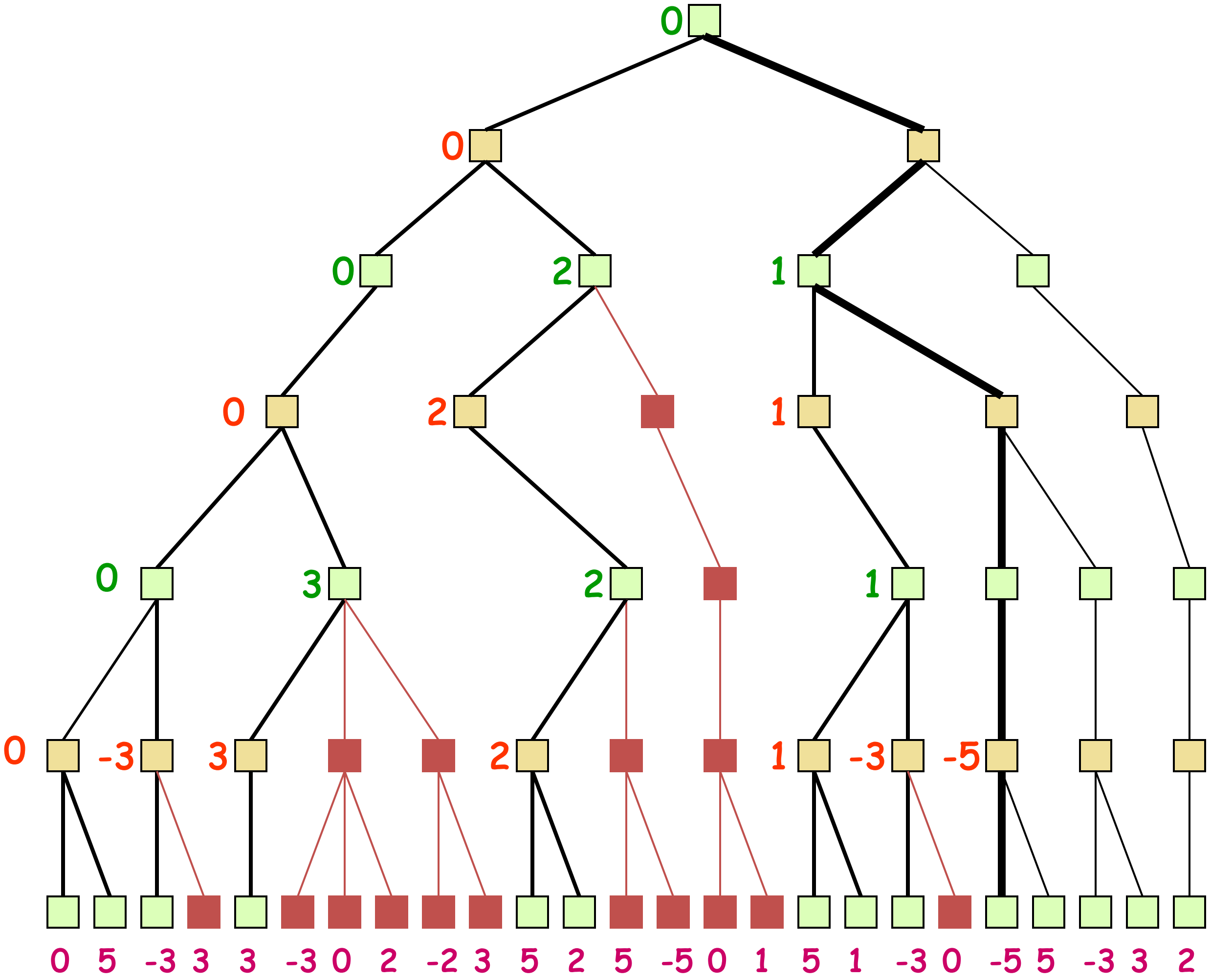

0
0
0
2
0
2
0
3
2
0
-3
3
2
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

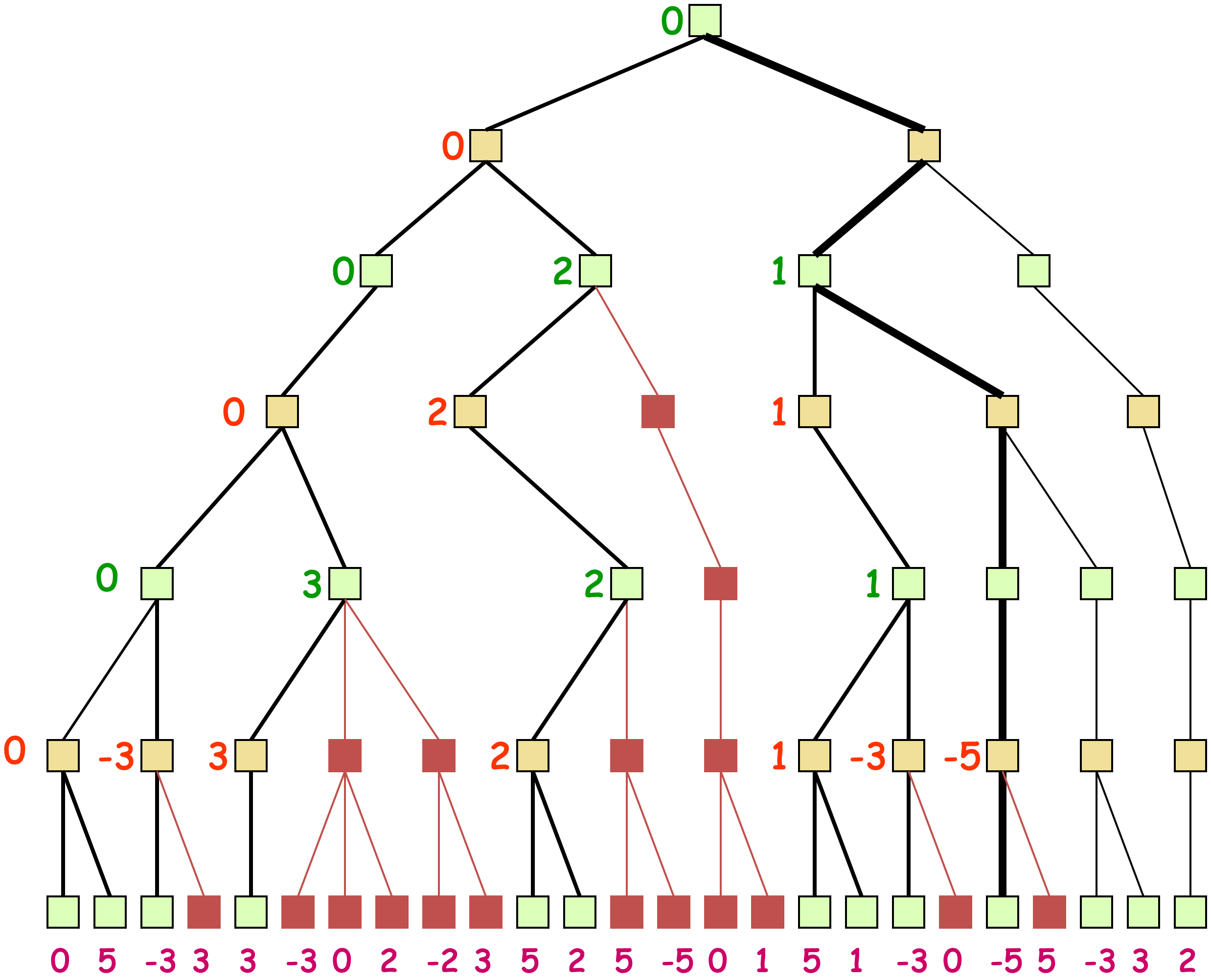

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

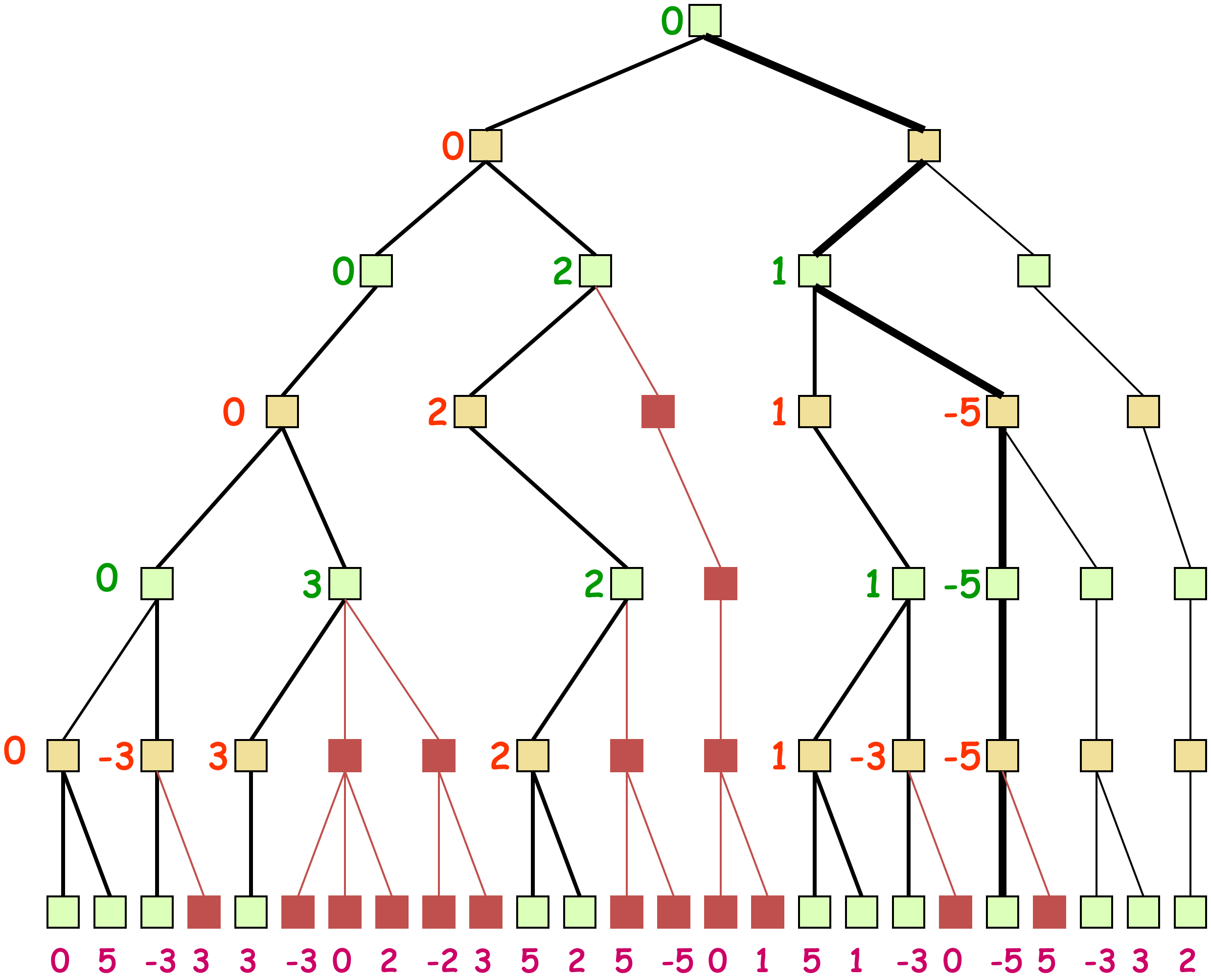

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

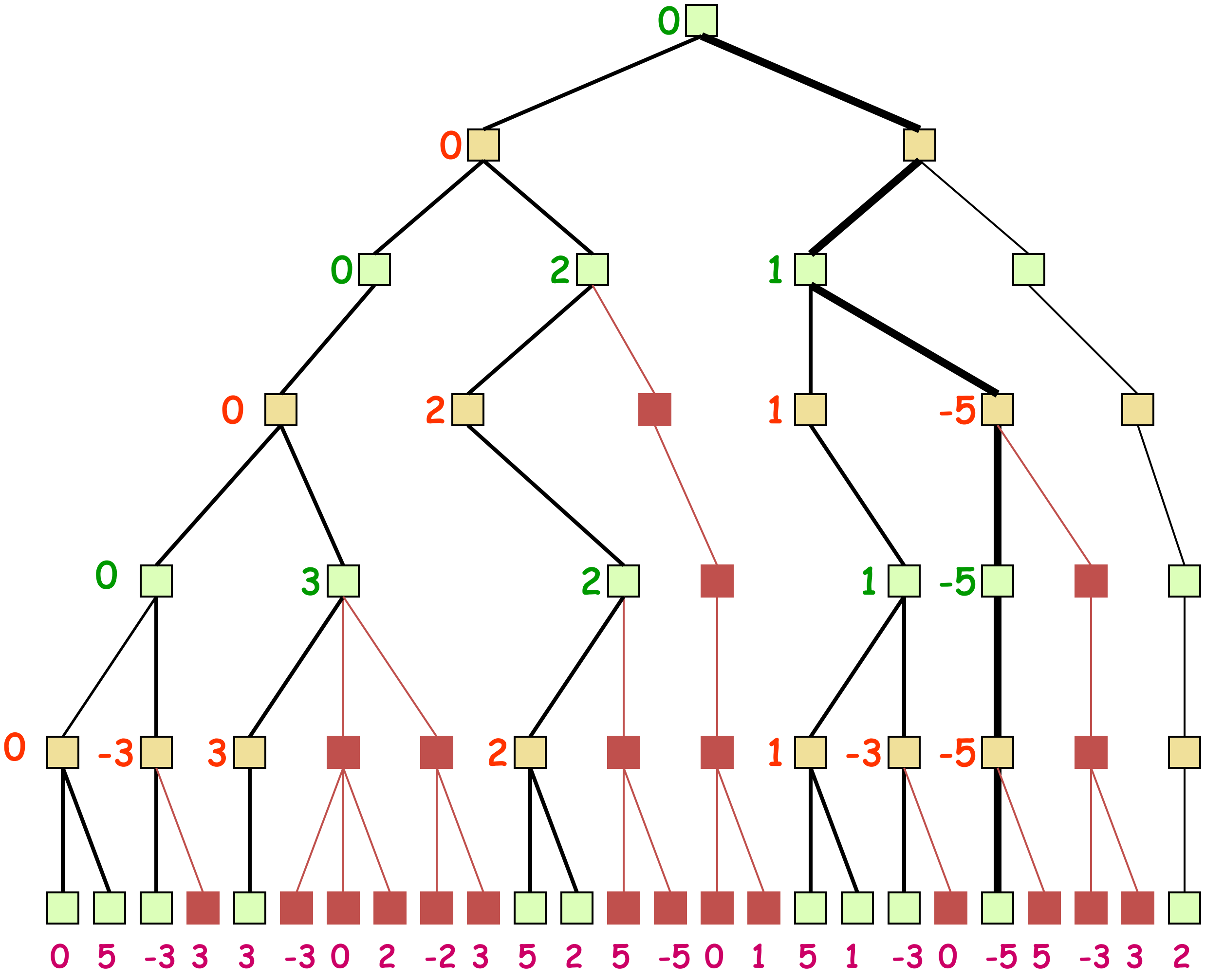

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

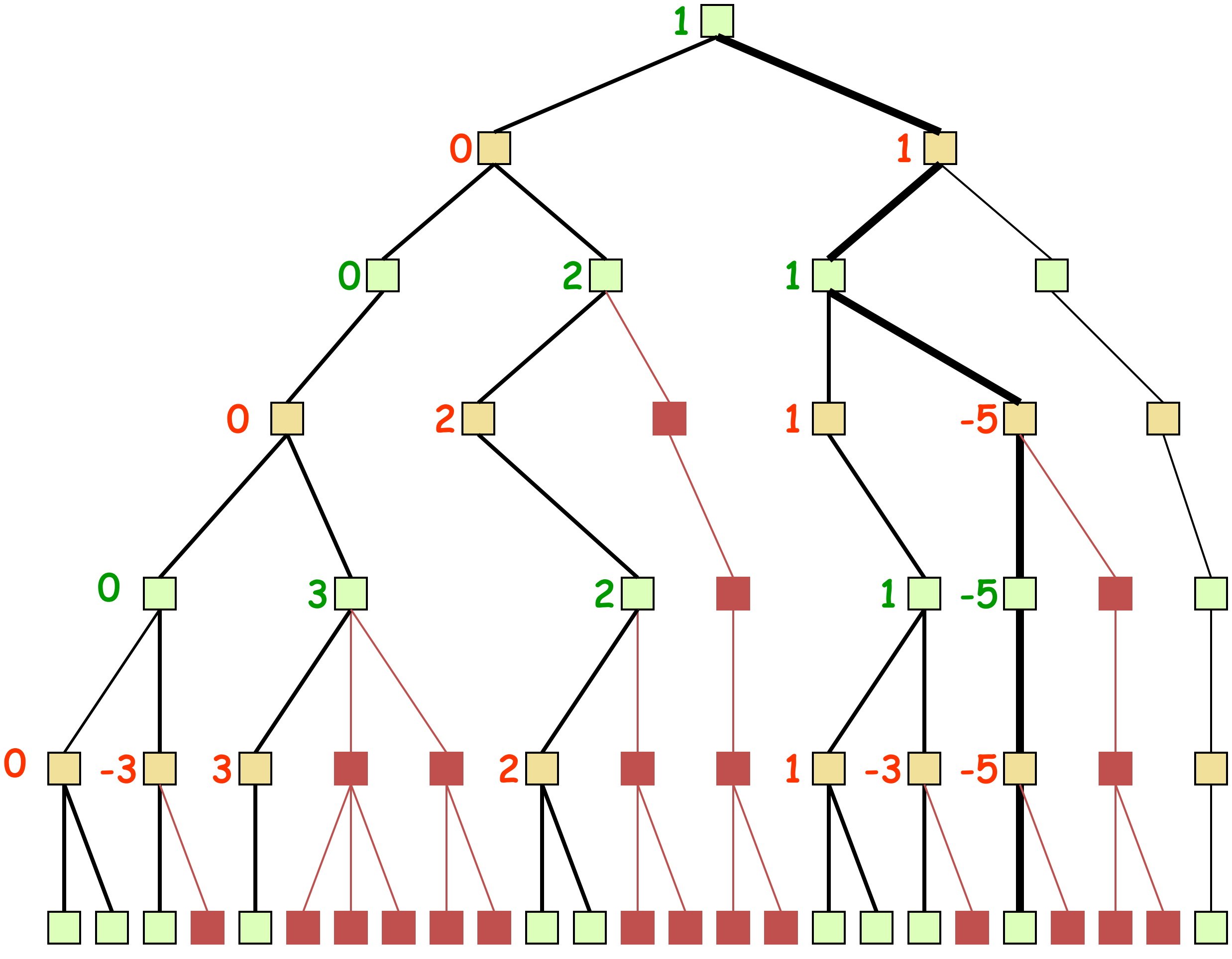

0
0
0
2
1
0
2
1
0
3
2
1
0
-3
3
2
1
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

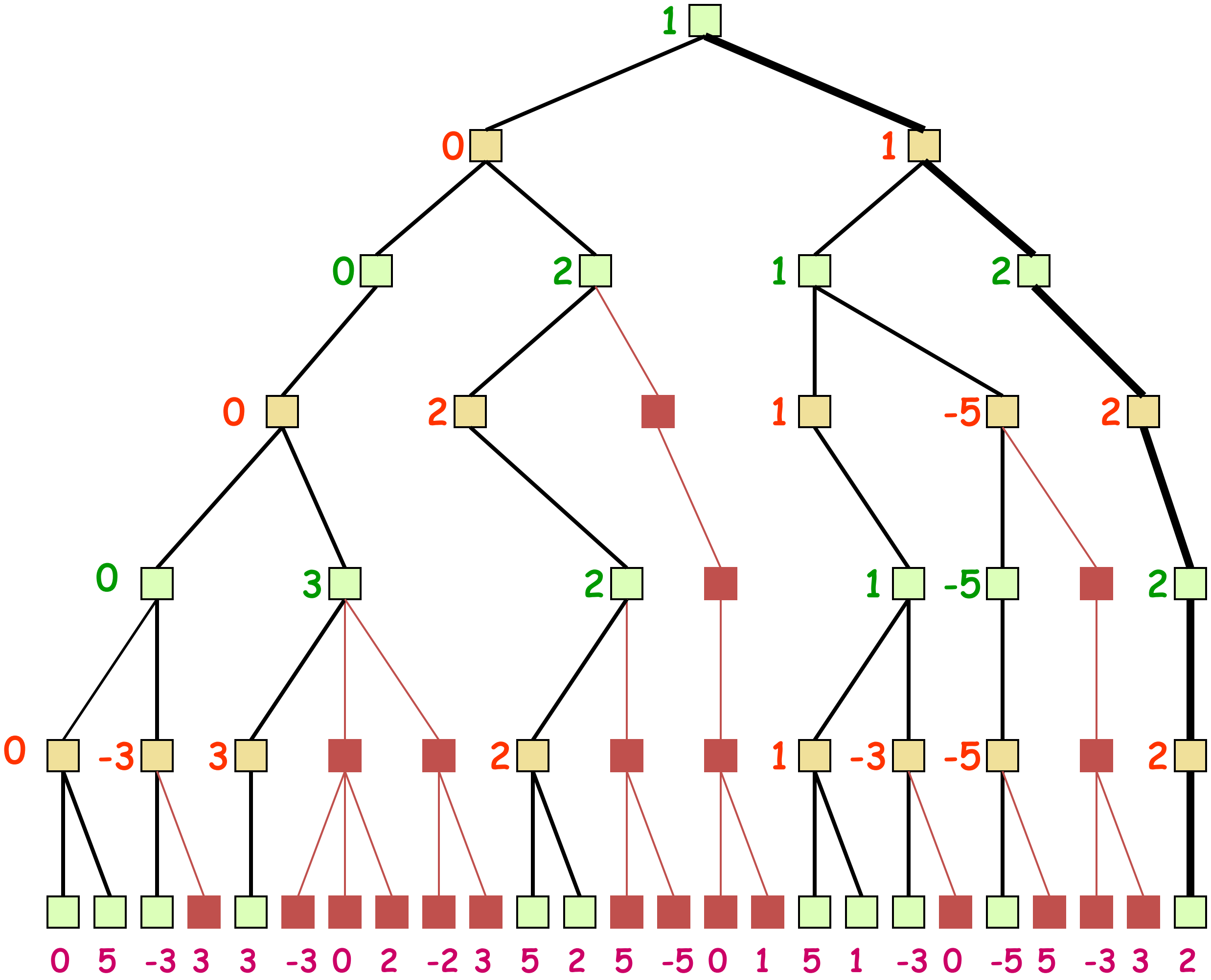

0
0
0 2 1
0 2 1
0 3 2 1
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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

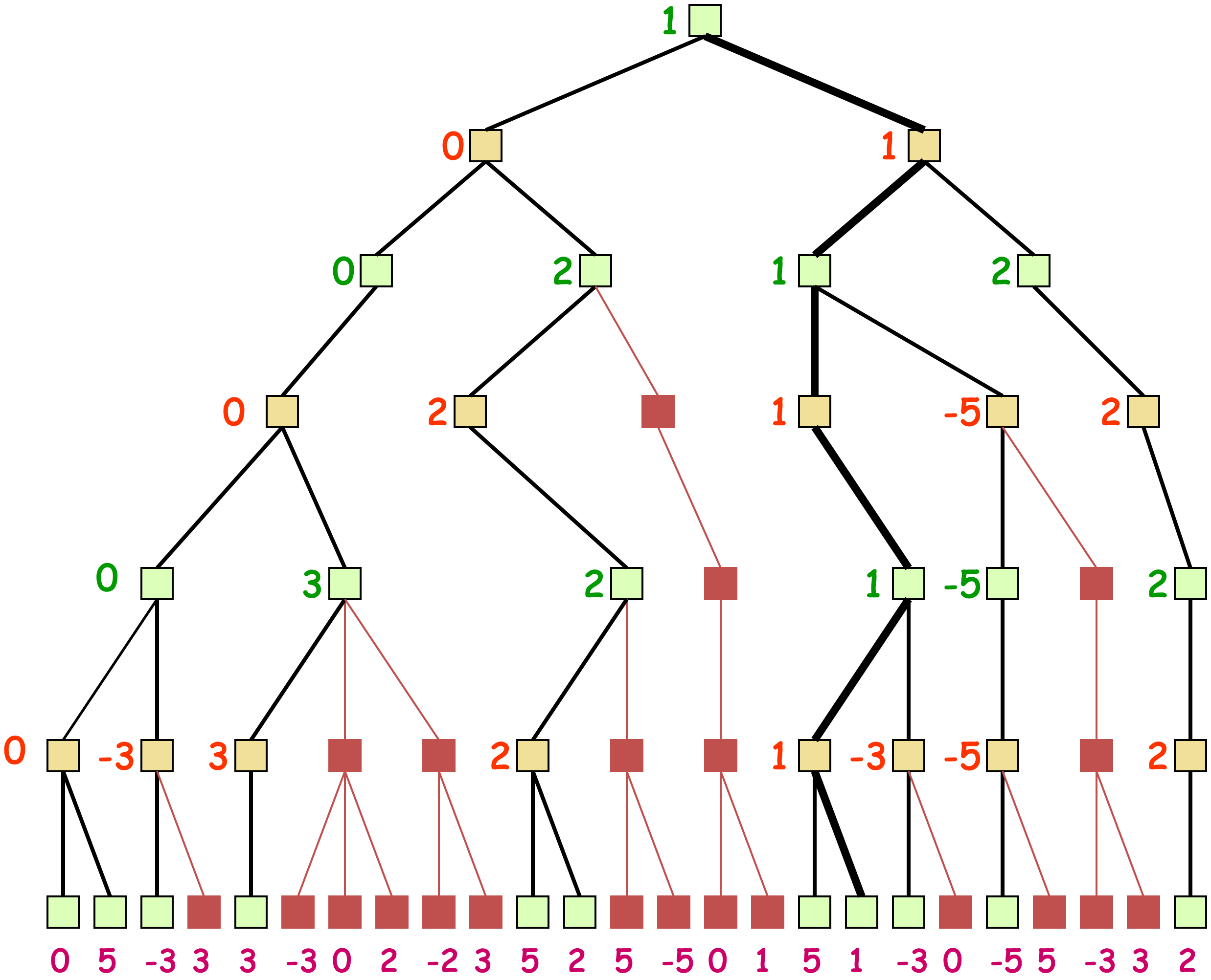

0
0
0
2
1
0
2
1
-5
0
3
2
1
-5
0
-3
3
2
1
-3
-5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


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



# Many 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