Minimax and Alpha-Beta Pruning

Chapter 2 · Algorithmic Game Theory

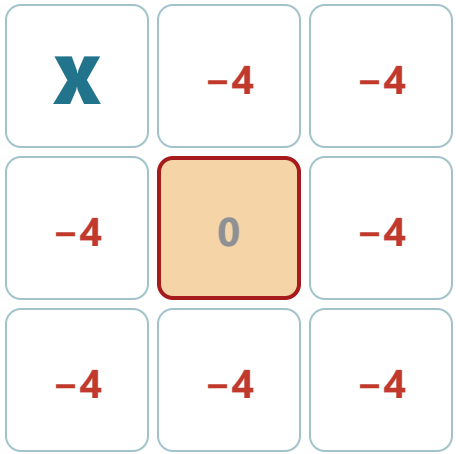
Game trees | Minimax | Nim | Minimax in tic-tac-toe | Alpha-beta pruning | Move ordering | Connect Four | Depth-limited search

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/agt/slides/Chapter_02_Minimax_and_Alpha_Beta_Pruning.html

INTRODUCTION

Warm-up Game: Tic-Tac-Toe vs the AI



New game: you start | New game: AI starts | AI move

Example: X is in the top-left corner. The AI (O) evaluated every empty square with the full tree; numbers are values from the AI's view (+ win, 0 draw, − loss). Choice: **the center square (0)**. The center square guarantees a draw; with perfect play by both sides the game is drawn. Every other square loses. (59,705 nodes)

Click a square to place the human player's mark. Before moving, the AI writes in every empty square what it would get at the end of the game by choosing it (+ win, 0 draw, − loss); the orange square is its choice. The **AI move** button makes the move.

Think about it: how does the AI know these numbers? After a corner opening, why does every reply except the center lose?

INTRODUCTION

Learning Outcomes

Game trees

Draw positions as nodes and moves as edges; recognize MAX and MIN levels, leaves and utilities; estimate tree size as b^d.

Minimax

Propagate values from the leaves to the root by hand; state the move chosen at the root and why; trace the code line by line.

Nim and tic-tac-toe

Show that minimax agrees with P/N positions and the XOR rule in Nim, and why tic-tac-toe needs a depth penalty.

Alpha-beta pruning

Define α and β, track the window [α, β] at each node, find the branches pruned when $\alpha \geq \beta$, and explain why the result is unchanged.

Move ordering

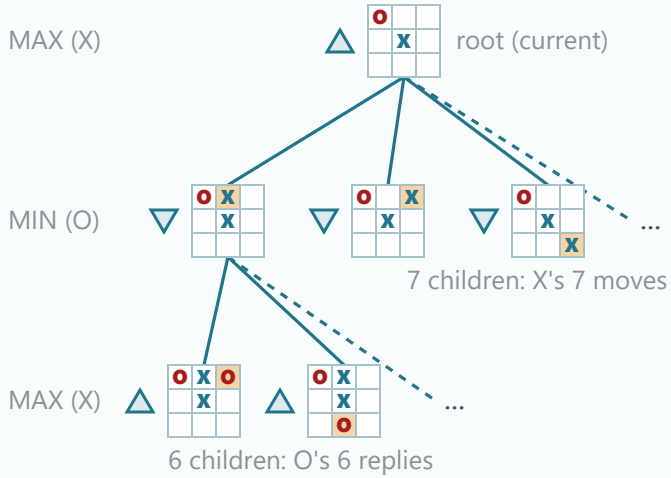
Compare the number of leaves evaluated under the best and worst orderings ($b^{d/2}$ vs b^d).

Depth-limited search

Write an evaluation function for Connect Four; explain the horizon effect, iterative deepening and transposition tables.

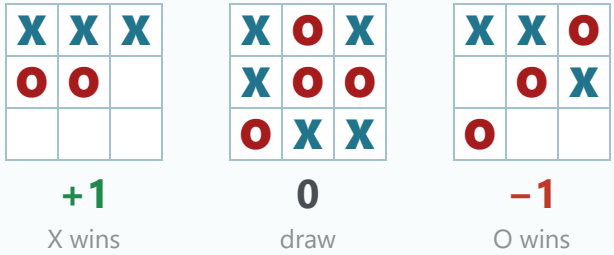
Game Trees: Positions and Moves

- A **node** is a position (board and player to move); an **edge** is a move. The **root** is the current position.
- Two players alternate, there is no chance, and both see the board: a **perfect-information**, **zero-sum** game.
- The player to move at the root is **MAX** (▲) and wants a large value. The opponent is **MIN** (▽) and wants a small one.
- Levels alternate between MAX and MIN; going down one level is one ply (one move by one player).
- The same position can be reached by different move orders; it then appears as several nodes in the tree.



Leaves and Utilities

- A **leaf** (terminal position): the game is won or the board is full; it has no children.
- Every leaf has a **utility**. In tic-tac-toe, from MAX's (X's) view: **+1** if X wins, **0** for a draw, **−1** if O wins.
- The game is zero-sum: O's utility is minus X's ($u_O = -u_X$), so one number per node is enough.
- The values of inner nodes are not known yet; minimax computes them from the leaves.



Utility is defined only at the leaves. A judgment like "the center is a good square" becomes a number only once the tree is searched down to the leaves.

Branching Factor and Depth

- **Branching factor** b: the number of legal moves in a position (average or maximum). **Depth** d: the length of the game in plies.
- If every node has b children, there are b^d leaves at depth d: **tree size ≈ b^d**. One more ply multiplies the work by b.

Game	b	d (plies)	Number of games or positions
Tic-tac-toe	≤ 9	≤ 9	9! = 362,880 upper bound; actually 255,168 games, 549,946 nodes
Connect Four	≤ 7	≤ 42	4,531,985,219,092 ≈ $4.5 \cdot 10^{12}$ legal positions (Tromp)
Chess	≈ 35	≈ 80	$35^{80} \approx 10^{123}$; Shannon's (1950) rough estimate ≈ 10^{120} games
Go (19×19)	≈ 250	≈ 150	≈ $2.1 \cdot 10^{170}$ legal positions (Tromp, 2016)

The full tic-tac-toe tree is searched in under a second. Even Connect Four needs clever methods for a full search; in chess and Go the search stops at some depth.

The Minimax Value

The **minimax value** of a position s is the utility MAX gets at the end of the game if both sides play optimally:

```
minimax(s) =
  utility(s)                        if s is terminal
  max  minimax(result(s, a))        if MAX to move
   a
  min  minimax(result(s, a))        if MIN to move
   a
```

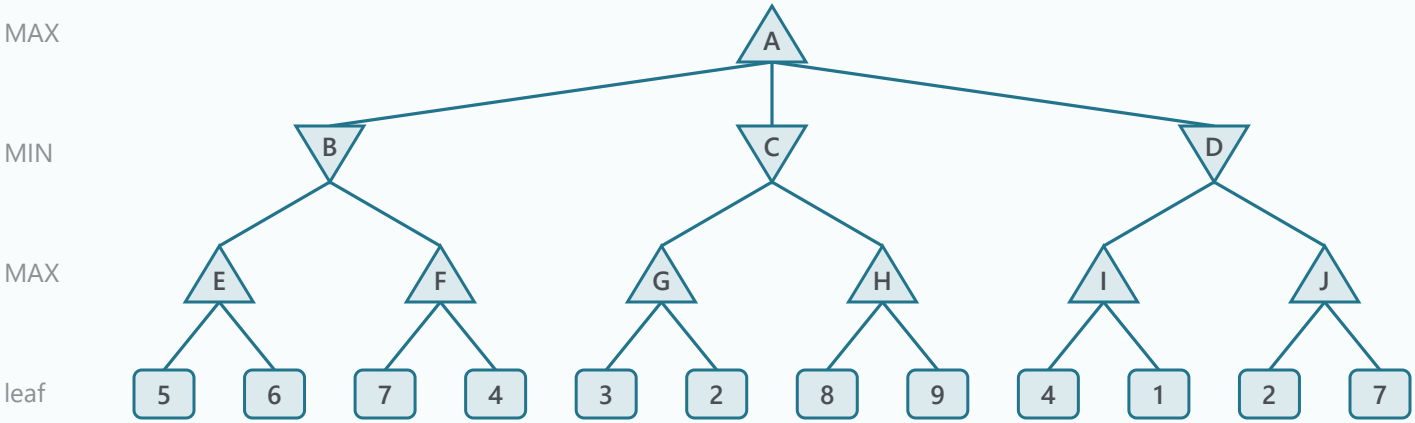
- The definition is recursive: the search goes **depth-first**, and values **rise** from the leaves to the root.
- At the root, MAX picks the move leading to the child with the largest value.

Cost

- Time: each node once → $O(b^d)$.
- Space: only the path from the root to the current node plus sibling information → $O(b \cdot d)$.
- Against a perfect opponent the minimax move is optimal; if the opponent errs, MAX gets at least the minimax value.

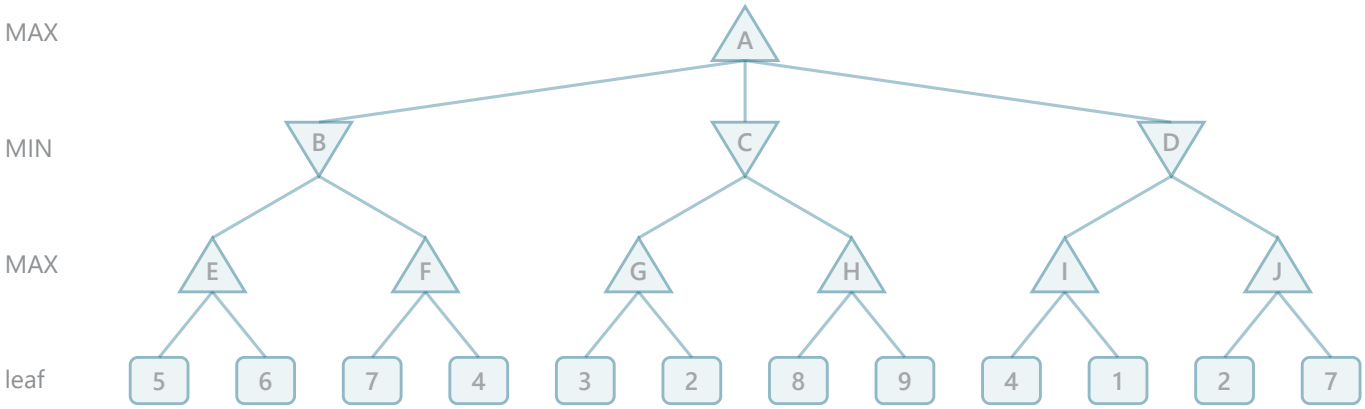
What Is the Root's Value?

In this three-level tree the root is MAX, the second level MIN and the third MAX. What is the root's value, and which move does MAX choose?



Show answer

Minimax Step by Step



Start: the values sit at the leaves. The Step button starts a depth-first search; values rise from the leaves to the root.

Step **0 / 32** · Leaves evaluated: **0 / 12**

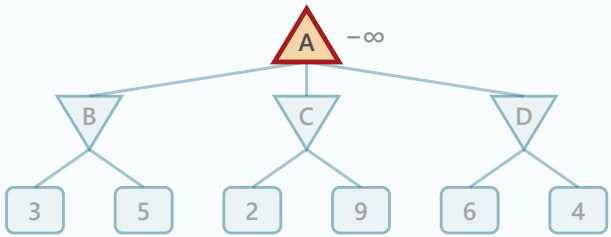
Back | Step | Play | Reset

Orange: node being processed · red frame: nodes on the stack · gray number: value not final yet · bold edge: the move chosen at the root.

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



First call: root A is a MAX node, best = -inf.

Step 1 / 14

VARIABLES

`n = "A"` `maxi = True` `best = -inf`

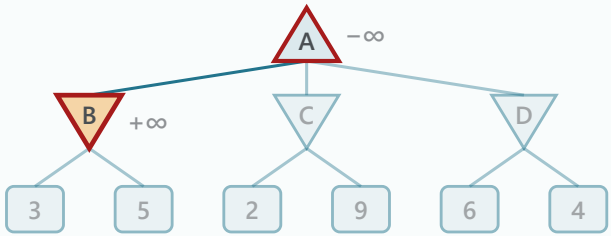
CALL STACK

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax("B") is called: MIN, best = inf.

Step 2 / 14

VARIABLES

`n = "B"` `maxi = False` `best = inf`

CALL STACK

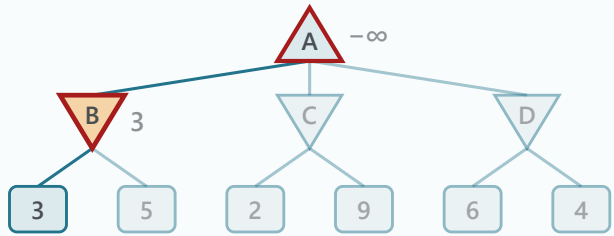
`minimax("B", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax(3) is a leaf: returns 3. With min, best = 3.

Step 3 / 14

VARIABLES

`n = "B"` `maxi = False` `c = 3` `v = 3` `best = 3`

CALL STACK

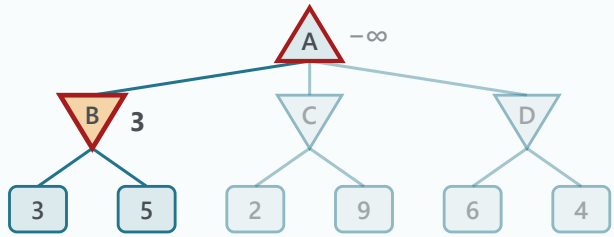
`minimax("B", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
▶    return best
15 print(minimax("A", True))
```

OUTPUT



minimax(5) is a leaf: returns 5. With min, best = 3; B is done and returns 3.

Step 4 / 14

VARIABLES

`n = "B"` `maxi = False` `c = 5` `v = 5` `best = 3`

CALL STACK

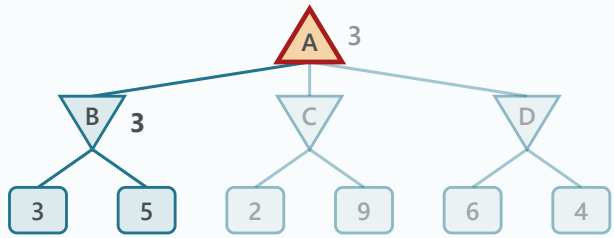
`minimax("B", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
▶        best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



The call on B returned 3. With max, best = 3.

Step 5 / 14

VARIABLES

`n = "A"` `maxi = True` `c = "B"` `v = 3` `best = 3`

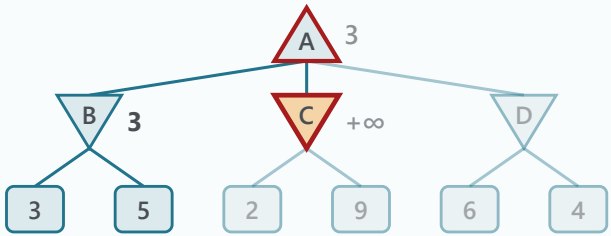
CALL STACK

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax("C") is called: MIN, best = inf.

Step 6 / 14

VARIABLES

`n = "C"` `maxi = False` `best = inf`

CALL STACK

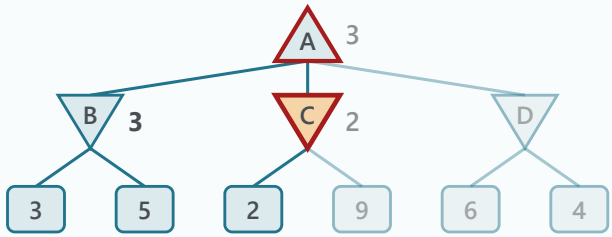
`minimax("C", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax(2) is a leaf: returns 2. With min, best = 2.

Step 7 / 14

VARIABLES

`n = "C"` `maxi = False` `c = 2` `v = 2` `best = 2`

CALL STACK

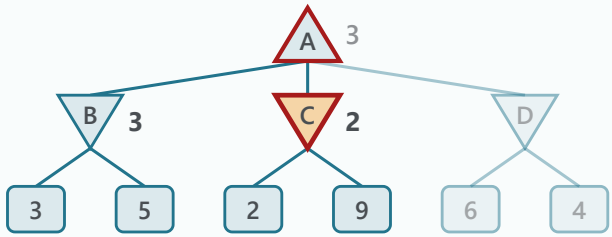
`minimax("C", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
▶    return best
15 print(minimax("A", True))
```

OUTPUT



minimax(9) is a leaf: returns 9. With min, best = 2; C is done and returns 2.

Step 8 / 14

VARIABLES

`n = "C"` `maxi = False` `c = 9` `v = 9` `best = 2`

CALL STACK

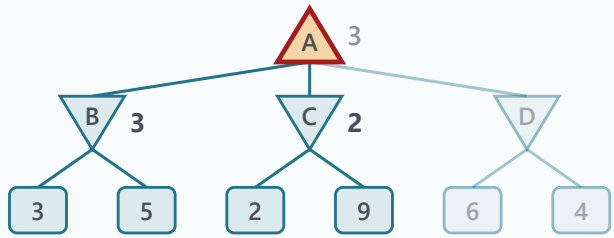
`minimax("C", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
▶        best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



The call on C returned 2. With max, best = 3.

Step 9 / 14

VARIABLES

`n = "A"` `maxi = True` `c = "C"` `v = 2` `best = 3`

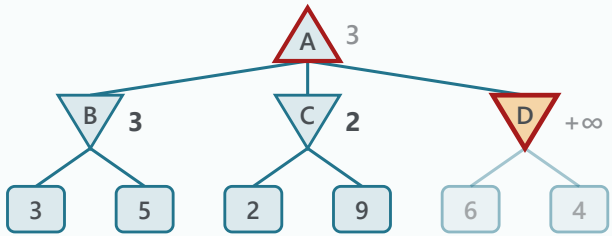
CALL STACK

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax("D") is called: MIN, best = inf.

Step 10 / 14

VARIABLES

`n = "D"` `maxi = False` `best = inf`

CALL STACK

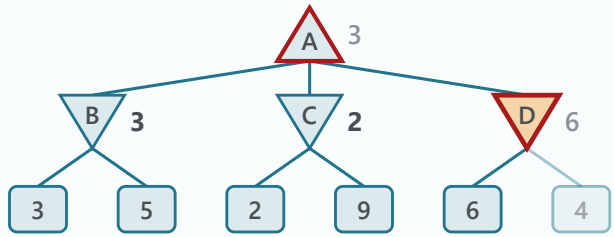
`minimax("D", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
14   return best
15 print(minimax("A", True))
```

OUTPUT



minimax(6) is a leaf: returns 6. With min, best = 6.

Step 11 / 14

VARIABLES

`n = "D"` `maxi = False` `c = 6` `v = 6` `best = 6`

CALL STACK

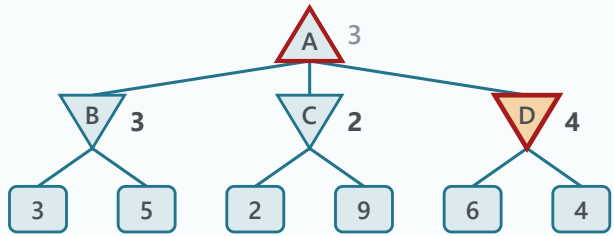
`minimax("D", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
▶        best = min(best, v)
▶    return best
15 print(minimax("A", True))
```

OUTPUT



minimax(4) is a leaf: returns 4. With min, best = 4; D is done and returns 4.

Step 12 / 14

VARIABLES

`n = "D"` `maxi = False` `c = 4` `v = 4` `best = 4`

CALL STACK

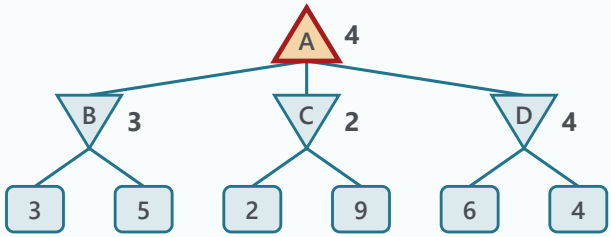
`minimax("D", False)`

`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = minimax(c, not maxi)
10     if maxi:
▶        best = max(best, v)
12     else:
13       best = min(best, v)
▶    return best
15 print(minimax("A", True))
```

OUTPUT



The call on D returned 4. With max, best = 4; A is done and returns 4.

Step 13 / 14

VARIABLES

`n = "A"` `maxi = True` `c = "D"` `v = 4` `best = 4`

CALL STACK

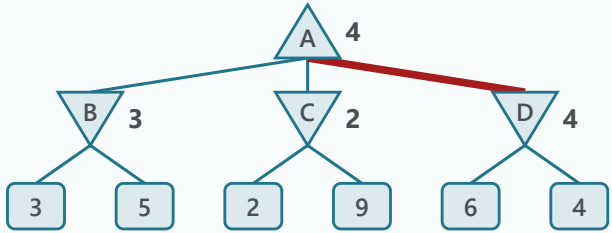
`minimax("A", True)`

Code Trace: The Minimax Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def minimax(n, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
9      v = minimax(c, not maxi)
10     if maxi:
11       best = max(best, v)
12     else:
13       best = min(best, v)
14   return best
▶ print(minimax("A", True))
```

OUTPUT

```
4
```



The root's value is 4: A picks the move to D. All 6 leaves were read.

Step 14 / 14

VARIABLES

`result = 4`

CALL STACK

empty

Minimax, Backward Induction and Maximin

- Minimax is the two-player zero-sum special case of **backward induction**: in a general game the player to move at each node maximizes *its own* payoff; in a zero-sum game B's payoff is minus A's, so "B maximizes" = "B minimizes A's payoff".
- **Maximin**: the best value MAX can guarantee, $\max_s \min_t u(s, t)$. **Minimax**: the lowest value MIN can hold MAX to, $\min_t \max_s u(s, t)$.
- In finite games of perfect information the two are equal and pure strategies suffice (Zermelo, 1913; Kuhn, 1953). The minimax value of the tree is this common value.

The difference in simultaneous games

In Matching Pennies (A gets +1 on a match, −1 otherwise), with pure strategies maximin is −1 and minimax +1: the gap does not close.

von Neumann (1928): once mixed strategies are allowed, maximin = minimax in every finite two-player zero-sum game (the **minimax theorem**). In Matching Pennies the common value is 0.

The Minimax Assumption: A Perfect Opponent

What does it guarantee?

Whatever the opponent does, MAX gets at least the minimax value. If the opponent errs, MAX may get more, never less.

What does it not guarantee?

It does not find the move earning the most against a weak opponent: of two equally valued moves, it cannot tell which one sets a trap.

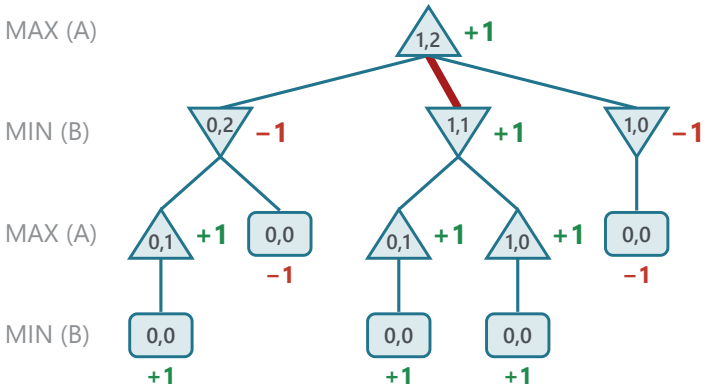
With chance

In dice games such as backgammon, **chance nodes** are added to the tree and averaged over (expectiminimax); pruning becomes harder.

In a two-player zero-sum game, assuming "my opponent plays its best move" yields a safe lower bound: the minimax move maximizes the worst-case payoff.

The Game Tree of Nim

Position: **(1, 1)** **(1, 2)** **(3)** **(1, 3)** **(2, 2)** **Hide values**



Root (1, 2): value **+1**, A to move wins → an **N position**. Winning move: **(1, 1)**. Nim-sum $1 \oplus 2 = 3 \neq 0$; the move making the XOR 0 is the same ✓.

The player to move takes at least one chip from a single heap; whoever takes the last chip wins. Nodes show heap sizes; +1: A (the first player) wins. Nim →

P and N Positions = Minimax Values

- **N position**: the next player (to move) wins. **P position**: the previous player (not the one to move) wins.
- Rules: a position with no moves is P; a position is N ⇔ at least one move leads to P; it is P ⇔ every move leads to N.
- If values are written from the *mover's* view (+1 win, −1 loss): `value(s) = max_a -value(result(s, a))`. This is minimax written as a single function (**negamax**).
- The rule "+1 if at least one child is −1" is exactly the definition of N: N ⇔ +1, P ⇔ −1.

Position	Value for the mover	Type
(0, 0)	−1	P
(1, 0)	+1	N
(1, 1)	−1	P
(2, 0)	+1	N
(1, 2)	+1	N
(2, 2)	−1	P
(1, 2, 3)	−1	P

Minimax and Bouton's XOR Rule

Bouton (1901): a position is P ⇔ the nim-sum (XOR) of the heaps is 0. The winning move makes the XOR 0. Compared with the move minimax finds on the full tree:

Position	XOR	Minimax's winning move	Move making the XOR 0	Full tree
(1, 2)	3	→ (1, 1)	→ (1, 1) ✓	12 nodes
(1, 3)	2	→ (1, 1)	→ (1, 1) ✓	28 nodes
(2, 3)	1	→ (2, 2)	→ (2, 2) ✓	86 nodes
(1, 2, 3)	0	none: a P position	none ✓	447 nodes
(3, 4, 5)	2	→ (1, 4, 5)	→ (1, 4, 5) ✓	1,038,768 nodes

Both methods find the same value and a winning move in every position; the difference is cost. The minimax tree grows exponentially with the number of chips; the XOR rule takes as many operations as there are heaps. When a game-specific theory exists, no search is needed; when it does not, minimax always works.

Value Scale and Depth Penalty in Tic-Tac-Toe

- On a ±1 scale a quick win and a slow win get the same value. In the position on the right, X wins *at once* with the bottom-left square and two moves later with the center: both +1.
- Ties go to the first square tried; if the center is tried first, the AI passes up the ready win and drags the game out. When losing, it may also pick the fastest loss.
- **Depth penalty**: with d the number of moves from the root,

```
X wins  +(10 − d)
O wins  −(10 − d)
draw     0
```

Quick wins and slow losses are preferred: +9 > +7, −6 > −8.

±1 scale: center chosen

X	O	−1
X	+1	−1
+1	O	−1

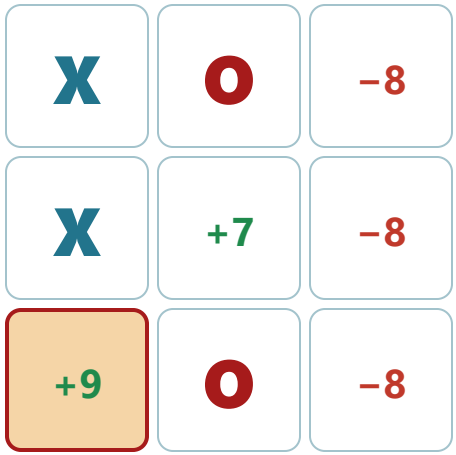
center, bottom-left: +1

depth penalty: bottom-left chosen

X	O	−8
X	+7	−8
+9	O	−8

+9 wins now, +7 two moves later

Tic-Tac-Toe Lab



To move: X (MAX) | Depth penalty: on

Play the best move

Empty board | Fork | Block | Penalty example

X to move (MAX, picks the largest): **the bottom-left square, +9**. The bottom-left square completes three in a row: an immediate win.

Minimax: **113** nodes (0.1 ms) · Alpha-beta: **32** nodes (0 ms) · root value in both **+9**

Clicking a square cycles empty → X → O → empty, so you can build any position. Numbers are minimax values from X's view: MAX (X) picks the largest, MIN (O) the smallest; the orange square is the choice. A reason is then given by rules, after the values.

Explaining the Move

X to move: wins

X	X	+9
O	O	0
−8	−8	−8

completes three

O to move: blocks

X	X	0
+8	O	+8
+8	+8	+8

only drawing square

X to move: forks

X	O	0
+7	X	0
+7	0	O

two threats

O to move: holds draw

X	+4	+4
+4	0	+4
+4	+4	+4

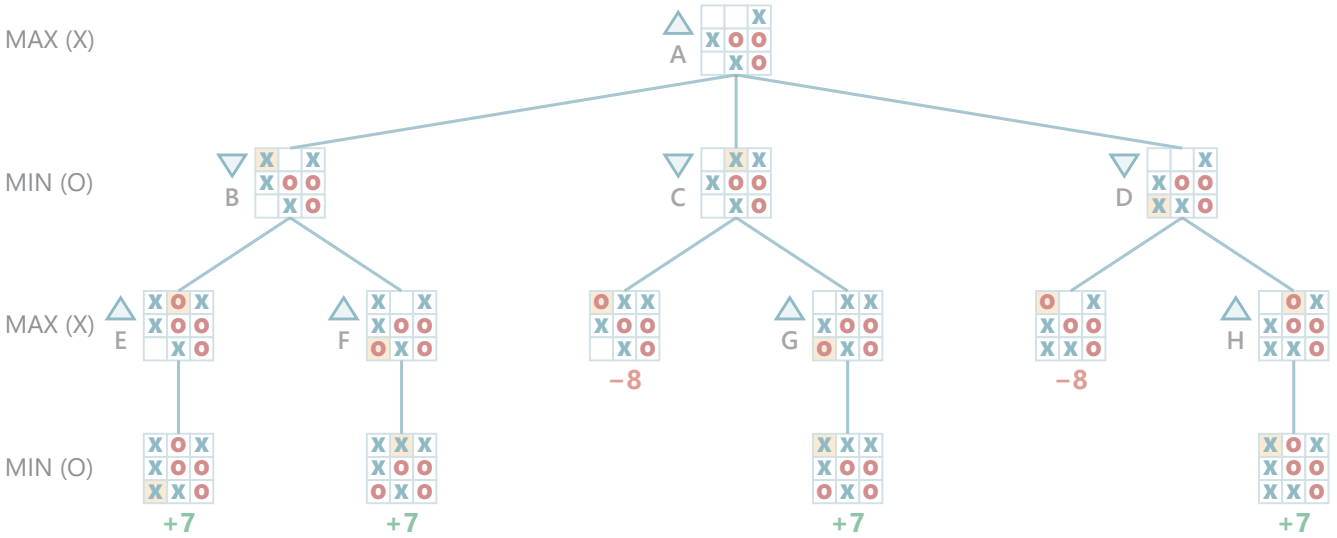
all others lose

- The values alone make the choice; the reason is found by asking, in order, what the chosen move achieves:
- 1) Does it complete three in a row? → **wins**. 2) Does it block the opponent's one-move threat? → **blocks**.
- 3) Does it create two threats at once? → **forks**. 4) Otherwise, the sign of the value: forces a win / holds a draw / delays the loss.

Instead of writing rules one by one

A good tic-tac-toe player can also be written with rules such as "win first, then block, then fork". Minimax knows none of these rules; it discovers them all by searching the tree. The same program works for Connect Four; the rules do not.

The Full Tree of a Mid-Game Position



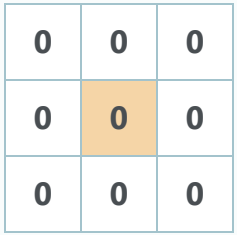
Start: X to move, three empty squares. Each node is a position; the orange square is the move leading to it. Values rise from the game ends to the root.

Step **0 / 22** · Leaves evaluated: **0 / 6**

Back | Step | Play | Reset

The Value of the Empty Board: a Draw

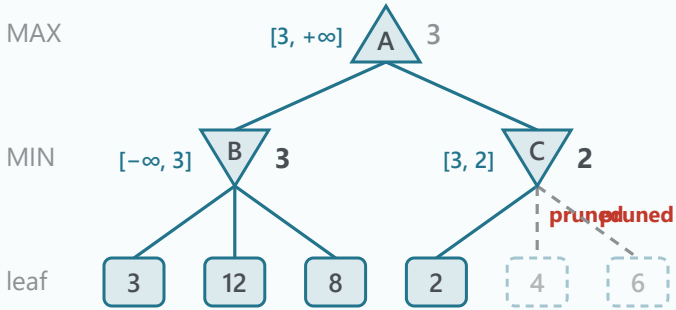
- A full minimax search from the empty board visits **549,946** nodes; **255,168** of them are leaves, i.e. distinct games.
- All nine of X's first moves have value **0**: no opening forces a win, and with perfect play by both sides the game is a **draw**.
- Since the values tie, the ordering decides; the AI in the lab breaks ties in the order center, corners, edges.
- O has the harder job: after a corner opening, O's only drawing reply is the center; each of the other seven replies loses.
- The same positions are searched again and again: there are only **5,478** distinct positions.



every first move is worth 0

Some Branches Need Not Be Read

- A (MAX) secured **3** from its first child B: B = min(3, 12, 8) = 3.
- C's first leaf is **2**. C is a MIN node, so its value is **at most 2**.
- With 3 in hand, A will never choose C, worth at most 2: whatever C's remaining leaves (4 and 6) are, the root's value does not change.
- These two leaves are never read: they are **pruned**. In chess a "leaf" is a deep subtree; pruning saves millions of nodes.



Alpha-beta pruning finds the **same root value** as minimax and (with the same move order) the same move, but skips branches that cannot affect the result.

α and β

α (alpha): MAX's lower bound

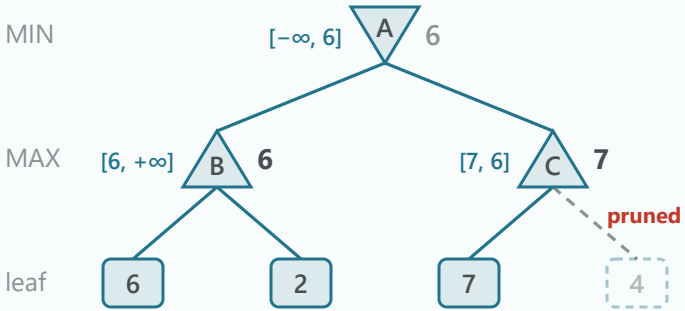
The best value MAX has *already secured* along the path from the root to this node. Starts at −∞. Grows only at MAX nodes: α = max(α, value).

β (beta): MIN's upper bound

The best (smallest) value MIN has *already secured* along the same path. Starts at +∞. Shrinks only at MIN nodes: β = min(β, value).

- Each node **takes the window [α, β] from its parent** and **passes it unchanged** to its children; only the value goes back to the parent.
- If a node's value falls outside this window it cannot affect the result: below α, MAX will not choose it; above β, MIN will not.
- Once the window is empty (**α ≥ β**), the node's remaining children need not be examined.

The Pruning Condition: α ≥ β



The MIN root secured 6 from B (β = 6). C is a MAX node; its first leaf is 7 → α = 7 ≥ β = 6. C gives at least 7, so MIN will not choose C: leaf 4 is pruned.

	At a MIN node	At a MAX node
Updated	β = min(β, value)	α = max(α, value)
Pruning	when β ≤ α	when α ≥ β
Meaning	this node gives at most β; the MAX above already has α	this node gives at least α; the MIN above already has β
Returned	an upper bound (true value ≤ β)	a lower bound (true value ≥ α)

Both cases are the same condition: α ≥ β. Some books call the cutoff at a MIN node an α-cutoff and the one at a MAX node a β-cutoff.

Why Pruning Does Not Change the Result

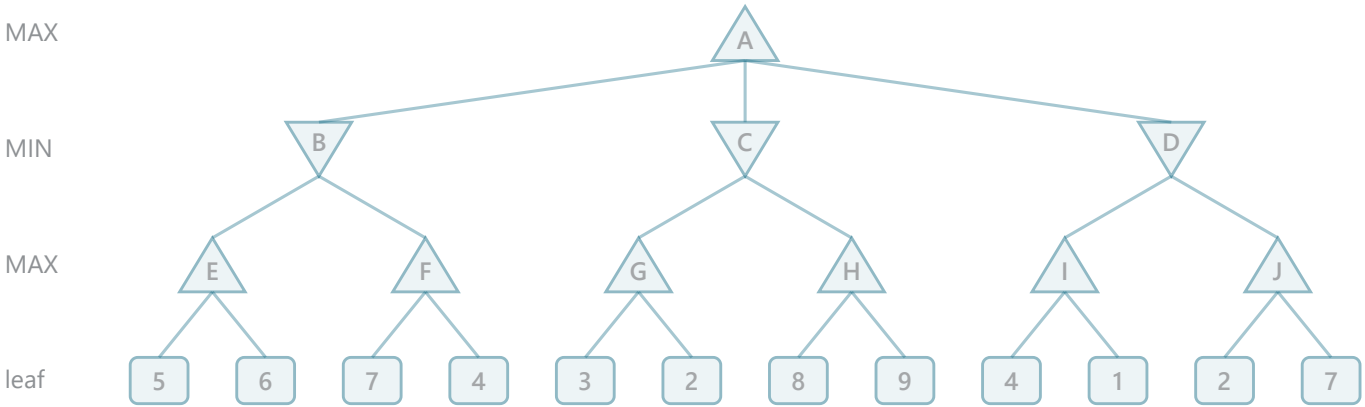
- Called with the window [α, β], alpha-beta guarantees (v the true minimax value, r the returned value):

if $\alpha < v < \beta$	r = v (exact value)
if $v \leq \alpha$	$r \leq \alpha$ (only known to be "bad enough")
if $v \geq \beta$	$r \geq \beta$ (only known to be "good enough")

- The root is called with $(-\infty, +\infty)$; v always lies inside, so the root's value is **exact**.
- Whatever the true value of a pruned branch, it cannot change the parent's max or min: for MAX a value $\leq \alpha$ cannot exceed α, and for MIN a value $\geq \beta$ cannot drop below β.
- Numbers returned at inner nodes may be bounds; the move chosen at the root is still minimax's move.

ALPHA-BETA PRUNING

Alpha-Beta Step by Step



Start: at root A, α = −∞, β = +∞. The Step button starts a depth-first search; pruned branches are dashed.

Step **0 / 26** · Leaves evaluated: **0 / 12**

Back | Step | Play | Reset

[α, β] window to the left of a node, its value to the right · dashed line and faded node: pruned branch · counter: leaves evaluated.

Minimax and Alpha-Beta: The Same Trees

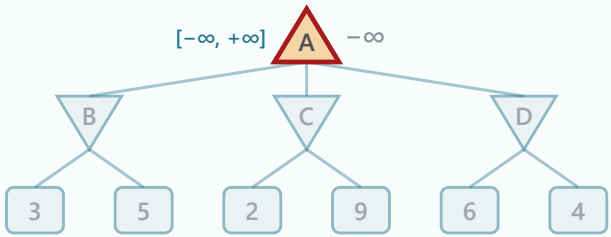
Tree	Minimax	Alpha-beta	Root value	Pruned, or ordering
Tree in the stepper (12 leaves)	12 leaves	7 leaves	6 and 6	F's 4; all of H and J
Tree in the code trace (6 leaves)	6 leaves	5 leaves	4 and 4	C's 9
Tree to solve by hand (8 leaves)	8 leaves	5 leaves	7 and 7	E's 1; all of G
Tic-tac-toe, empty board (full search)	549,946 nodes	7,865 nodes	0 and 0	center, corners, edges order
Connect Four, start, depth 7	960,793 nodes	5,086 nodes	+8 and +8	center-out order

In every row the root value and the chosen move agree; only the number of leaves read or nodes visited changes. The savings grow with the tree: a few leaves in the small trees, 70× in tic-tac-toe, 189× in Connect Four.

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶7   best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
14     if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



First call: root A is a MAX node, best = -inf, a = -inf, b = inf.

Step 1 / 14

VARIABLES

`n = "A"` `a = -inf` `b = inf` `maxi = True` `best = -inf`

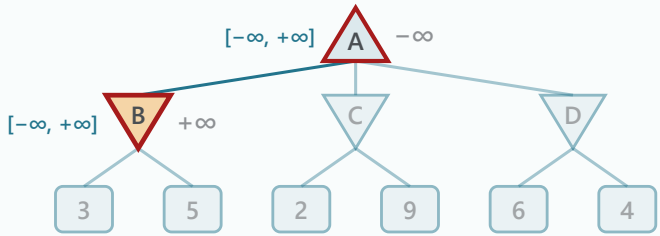
CALL STACK

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
14     if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta("B") is called: MIN, best = inf; a and b come from the caller: [-inf, inf].

Step 2 / 14

VARIABLES

`n = "B"` `a = -inf` `b = inf` `maxi = False` `best = inf`

CALL STACK

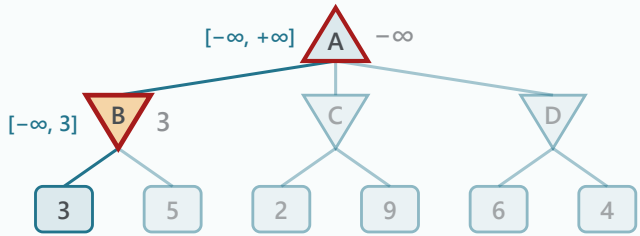
`alphabeta("B", -inf, inf, False)`

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
▶        best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta(3) is a leaf: returns 3. With min, best = 3, b = 3; a < b, continue.

Step 3 / 14

VARIABLES

`n = "B"` `a = -inf` `b = 3` `maxi = False` `c = 3` `v = 3` `best = 3`

CALL STACK

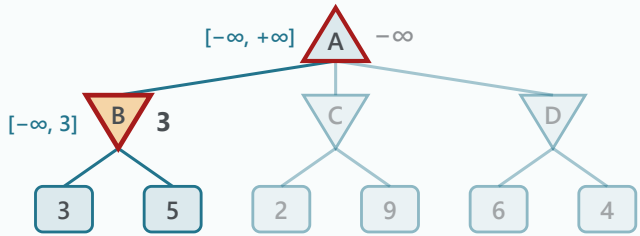
```
alphabeta("B", -inf, inf, False)
alphabeta("A", -inf, inf, True)
```

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):          # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
▶        best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
▶    return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT





alphabeta(5) is a leaf: returns 5. With min, best = 3, b = 3; a < b, continue; B is done and returns 3.

Step 4 / 14

VARIABLES

`n = "B"` `a = -inf` `b = 3` `maxi = False` `c = 5` `v = 5` `best = 3`

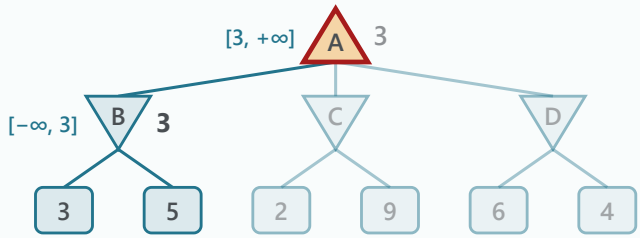
CALL STACK

```
alphabeta("B", -inf, inf, False)
alphabeta("A", -inf, inf, True)
```

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
▶        best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



The call on B returned 3. With max, best = 3, a = 3; a < b, continue.

Step 5 / 14

VARIABLES

`n = "A"` `a = 3` `b = inf` `maxi = True`
`c = "B"` `v = 3` `best = 3`

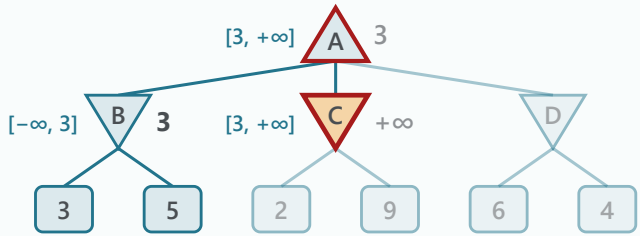
CALL STACK

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
14     if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta("C") is called: MIN, best = inf; a and b come from the caller: [3, inf].

Step 6 / 14

VARIABLES

`n = "C"` `a = 3` `b = inf` `maxi = False` `best = inf`

CALL STACK

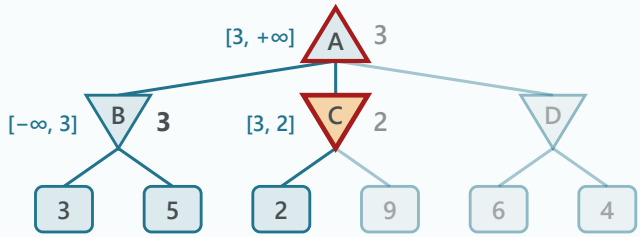
`alphabeta("C", 3, inf, False)`

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
▶        best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta(2) is a leaf: returns 2. With min, best = 2, b = 2; a ≥ b.

Step 7 / 14

VARIABLES

`n = "C"` `a = 3` `b = 2` `maxi = False` `c = 2` `v = 2` `best = 2`

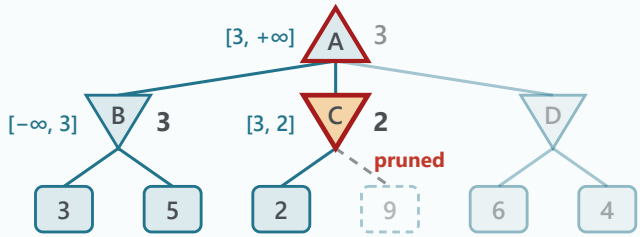
CALL STACK

```
alphabeta("C", 3, inf, False)
alphabeta("A", -inf, inf, True)
```

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
▶      if a >= b:
▶        break                   # prune
▶    return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



a = 3 ≥ b = 2: break. C gives at most 2 and A already found 3; leaf 9 is never read.

Step 8 / 14

VARIABLES

`n = "C"` `a = 3` `b = 2` `maxi = False` `c = 9` `best = 2`

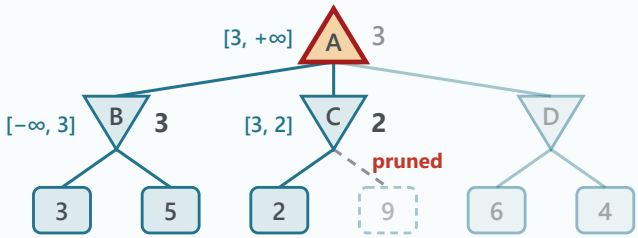
CALL STACK

```
alphabeta("C", 3, inf, False)
alphabeta("A", -inf, inf, True)
```

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
▶        best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



The call on C returned 2. With max, best = 3, a = 3; a < b, continue.

Step 9 / 14

VARIABLES

`n = "A"` `a = 3` `b = inf` `maxi = True`
`c = "C"` `v = 2` `best = 3`

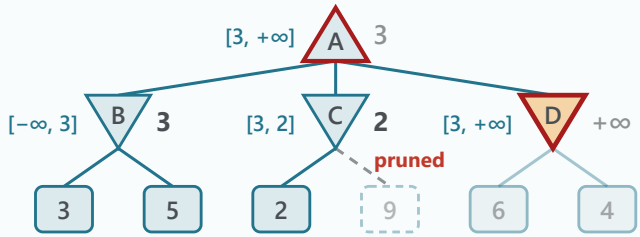
CALL STACK

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
▶    best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
14     if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta("D") is called: MIN, best = inf; a and b come from the caller: [3, inf].

Step 10 / 14

VARIABLES

`n = "D"` `a = 3` `b = inf` `maxi = False` `best = inf`

CALL STACK

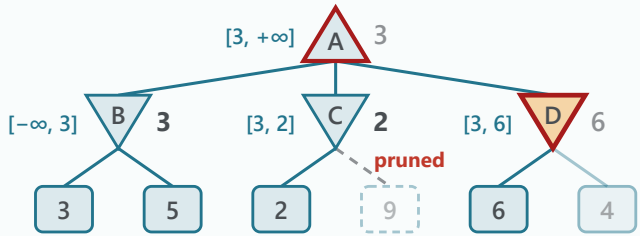
`alphabeta("D", 3, inf, False)`

`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
▶        best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
16   return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT



alphabeta(6) is a leaf: returns 6. With min, best = 6, b = 6; a < b, continue.

Step 11 / 14

VARIABLES

`n = "D"` `a = 3` `b = 6` `maxi = False` `c = 6` `v = 6` `best = 6`

CALL STACK

`alphabeta("D", 3, inf, False)`

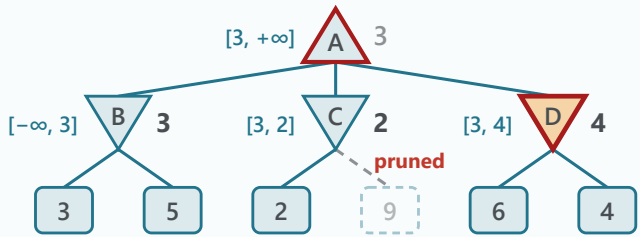
`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
▶        best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
▶    return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT





alphabeta(4) is a leaf: returns 4. With min, best = 4, b = 4; a < b, continue; D is done and returns 4.

Step 12 / 14

VARIABLES

`n = "D"` `a = 3` `b = 4` `maxi = False` `c = 4` `v = 4` `best = 4`

CALL STACK

`alphabeta("D", 3, inf, False)`

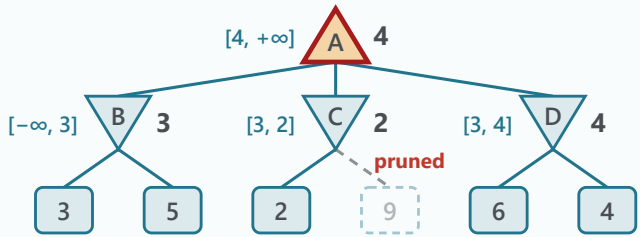
`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
▶      v = alphabeta(c, a, b, not maxi)
10     if maxi:
▶        best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
▶      if a >= b:
15       break                     # prune
▶    return best
17 print(alphabeta("A", -inf, inf, True))
```

OUTPUT





The call on D returned 4. With max, best = 4, a = 4; a < b, continue; A is done and returns 4.

Step 13 / 14

VARIABLES

`n = "A"` `a = 4` `b = inf` `maxi = True` `c = "D"` `v = 4` `best = 4`

CALL STACK

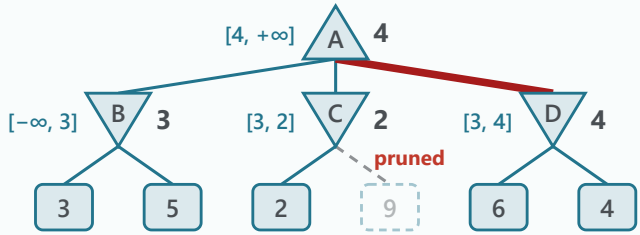
`alphabeta("A", -inf, inf, True)`

Code Trace: The Alpha-Beta Function

```
1  from math import inf
2  T = {"A": ["B", "C", "D"], "B": [3, 5],
3       "C": [2, 9], "D": [6, 4]}
4  def alphabeta(n, a, b, maxi):
5    if isinstance(n, int):        # leaf
6      return n
7    best = -inf if maxi else inf
8    for c in T[n]:
9      v = alphabeta(c, a, b, not maxi)
10     if maxi:
11       best = max(best, v); a = max(a, best)
12     else:
13       best = min(best, v); b = min(b, best)
14     if a >= b:
15       break                     # prune
16   return best
▶ print(alphabeta("A", -inf, inf, True))
```

OUTPUT

```
4
```



The root's value is 4: A picks the move to D. 5 of the 6 leaves were read; leaf 9 was pruned.

Step 14 / 14

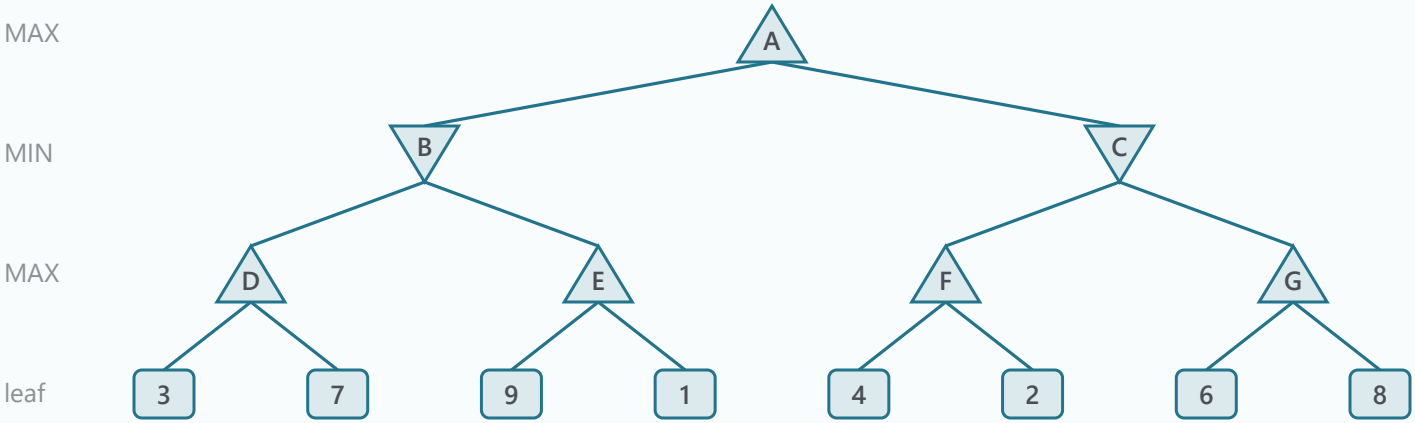
VARIABLES

`result = 4`

CALL STACK

empty

Alpha-Beta by Hand: Question



1. The root's value

What is the root's minimax value? Which move does MAX choose?

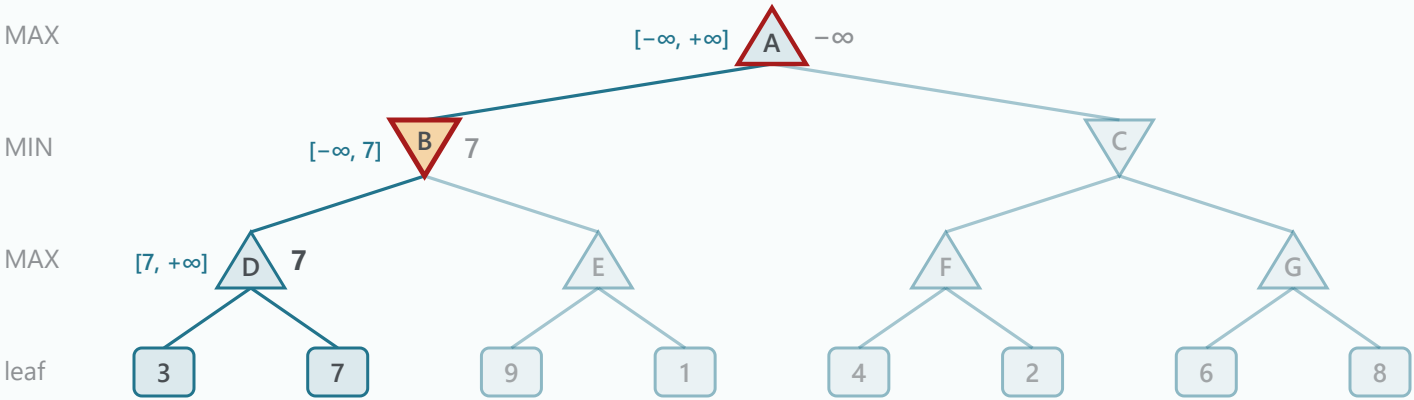
2. Windows

Searching children left to right, which window [α, β] reaches each inner node?

3. Pruned leaves

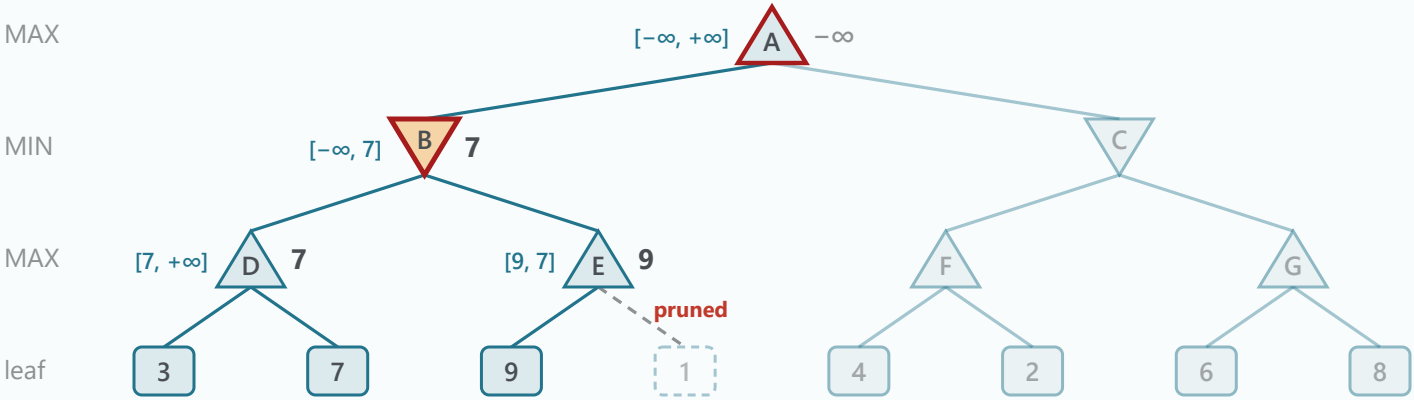
Which leaves are never read? How many leaves are evaluated?

Alpha-Beta by Hand: Solution



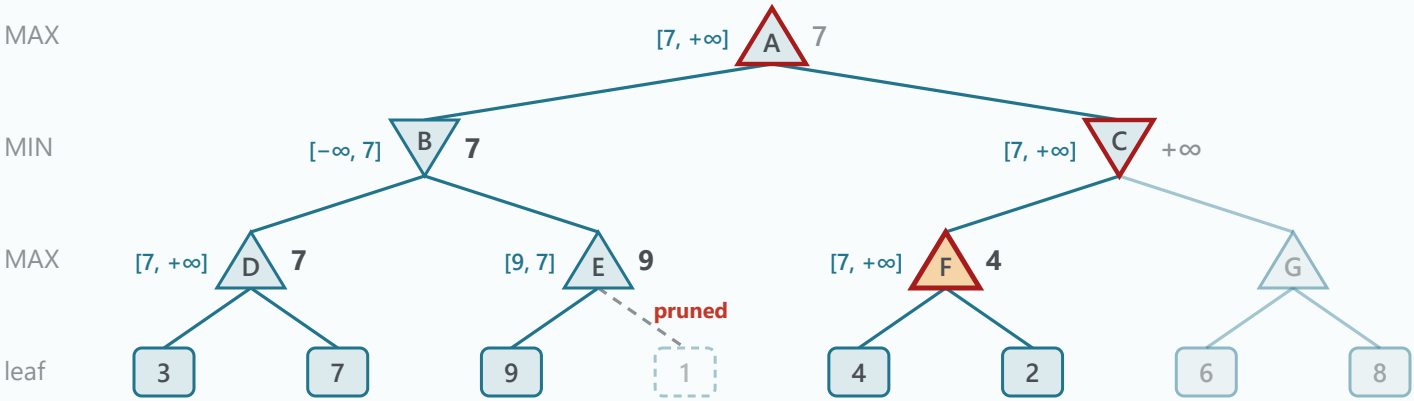
B receives [−∞, +∞]. D (MAX) reads 3 and 7: D = 7. B (MIN) sets β = 7.

Alpha-Beta by Hand: Solution



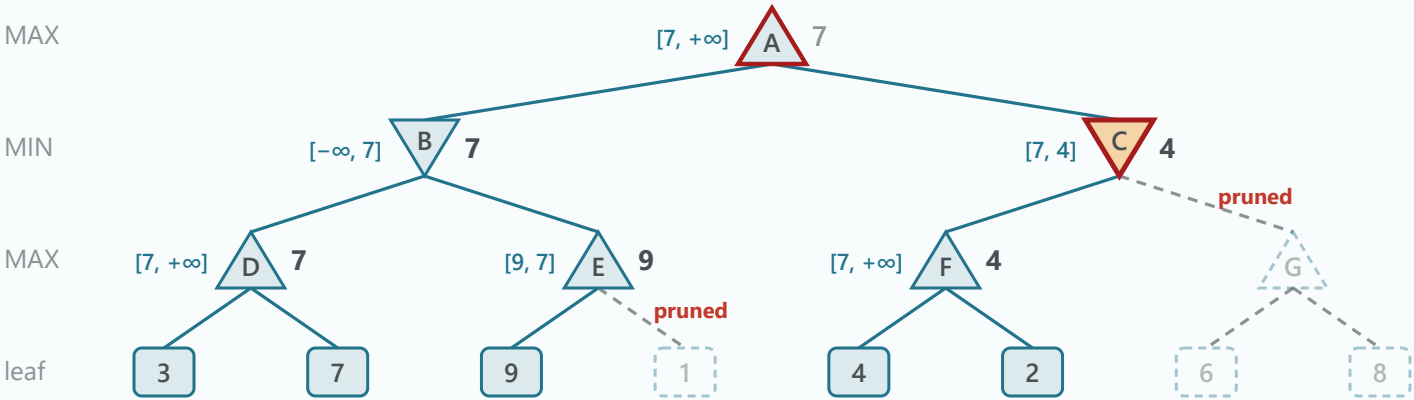
E receives [−∞, 7]. First leaf 9 → α = 9 ≥ β = 7: E gives at least 9, B will not choose it; leaf 1 is pruned. B = min(7, 9) = 7.

Alpha-Beta by Hand: Solution



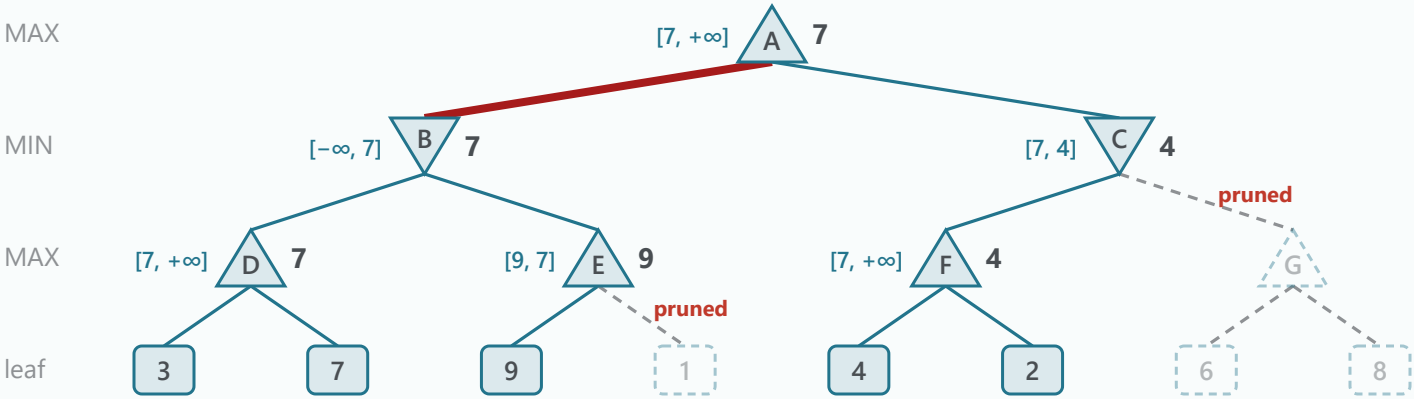
At the root α = 7. C receives [7, +∞], and so does F. F (MAX) reads 4 and 2: F = 4 (α stays 7).

Alpha-Beta by Hand: Solution



At C, β = min(+∞, 4) = 4 ≤ α = 7: C gives at most 4, the root will not choose it. All of G (6 and 8) is pruned.

Alpha-Beta by Hand: Solution



Root = max(7, 4) = 7: MAX picks B. 5 of the 8 leaves were read and 3 pruned. Same result as minimax.

Move Ordering

- How much alpha-beta prunes depends on the **order** in which children are tried; the root value does not.
- **Best ordering** (largest child first at MAX, smallest first at MIN): $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \approx b^{d/2}$ leaves evaluated (Knuth and Moore, 1975).
- **Worst ordering**: no pruning at all; all b^d leaves are read.
- Result: about **twice the depth** in the same time; the effective branching factor drops from b to $\sqrt{b}$.
- Where does a good ordering come from? Game knowledge (the center column in Connect Four, center and corners in tic-tac-toe, captures first in chess) and the best move of the previous search.

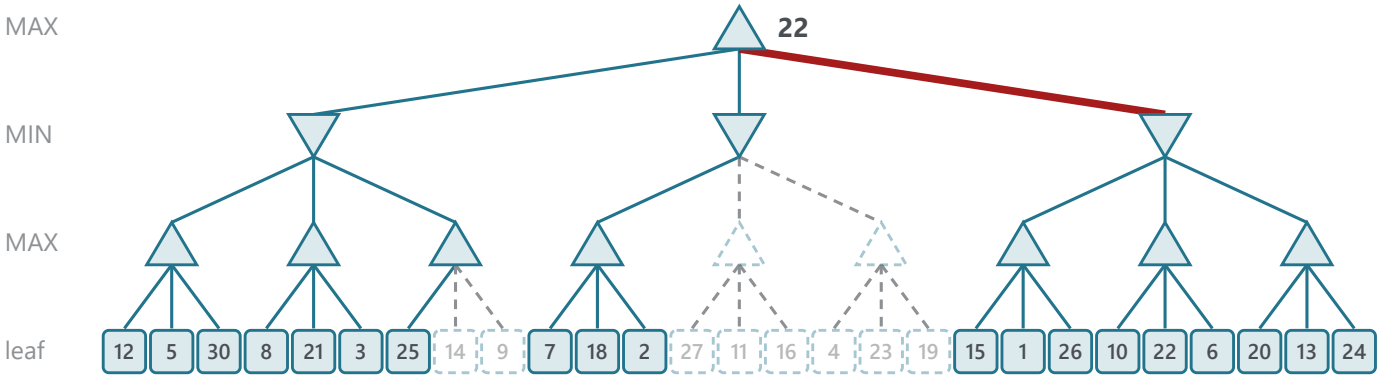
Leaves read for b = 35 (the chess average).

d	35^d	Best order
2	1,225	69
4	1,500,625	2,449
6	1,838,265,625	85,749
8	$\approx 2.25 \cdot 10^{12}$	3,001,249

MOVE ORDERING

Ordering Lab

Original order | Shuffle | Best order | Worst order | New values



Original order: leaves evaluated **19 / 27**, root value **22**. Theory: $3^2 + 3^1 - 1 = 11$ with the best order, 27 leaves for plain minimax.

Depth 3, branching 3: 27 leaves. The buttons change only the order of the children (the tree stays the same); alpha-beta reruns from scratch each time. Faded leaves and dashed edges are pruned branches. The best order reads 11 leaves; the worst order prunes nothing.

Alpha-Beta in Tic-Tac-Toe: Node Counts

Full search from the empty board	Nodes visited	Time	Root value
Minimax (order irrelevant)	549,946	34 ms	0
Alpha-beta, squares in order 0–8 (left to right, top to bottom)	20,866	2.9 ms	0
Alpha-beta, center first, then corners, then edges	7,865	0.7 ms	0
Minimax with a transposition table (each position once)	5,478	6.4 ms	0

Recompute

- The numbers are computed in this browser when the page opens, using the depth-penalty scale. On the ±1 scale alpha-beta visits 18,297 and 7,275 nodes: pruning also depends on the value scale.
- Good ordering cuts the node count by 2.7×; compared with minimax the saving is about 70×.
- A transposition table never searches the same position twice; with alpha-beta it also records whether the stored value is exact or a bound.

Connect Four: Why Not a Full Search?

- 7 columns, 6 rows; a disc drops to the lowest empty cell of the chosen column. The first to line up four discs horizontally, vertically or diagonally wins.
- The branching factor is at most 7, and a game lasts at most 42 plies. There are **4,531,985,219,092** ≈ $4.5 \cdot 10^{12}$ legal positions (John Tromp).
- The game was solved independently in 1988 by **James D. Allen** and **Victor Allis**: the first player can force a win by starting in the middle column.
- The solution was found not by brute force but with game-specific rules and very strong search. No browser can do this in a second.

What do we do?

The search is **cut off at a fixed depth**. Positions there are not terminal, so their utilities are unknown; instead an **evaluation function** estimates how good a position is.

From the start position, even depth 7 means 960,793 nodes for minimax.

Depth-Limited Search

```
H(s, d) =
  utility(s)          if s is terminal
  E(s)                if d = 0
  max  H(result(s, a), d − 1)  MAX
   a
  min  H(result(s, a), d − 1)  MIN
   a
```

Utility in the lab: +(1000 − k) if Red (MAX) wins on ply k, −(1000 − k) if Blue wins, 0 if the board fills up. Numbers near 1000 appear on the column buttons as "win" or "loss".

- **Cutoff test**: stop descending if the game is over or the depth limit is reached.
- **E(s)** is an estimate, not an exact value: large if the position looks good for MAX, small if it looks bad.
- Terminal values must lie above or below every value of E: wins are around ±1000, E is at most a few hundred.
- The depth penalty applies here too: quick wins and slow losses are preferred.
- Alpha-beta works unchanged; at the cutoff, E values are used like leaves.

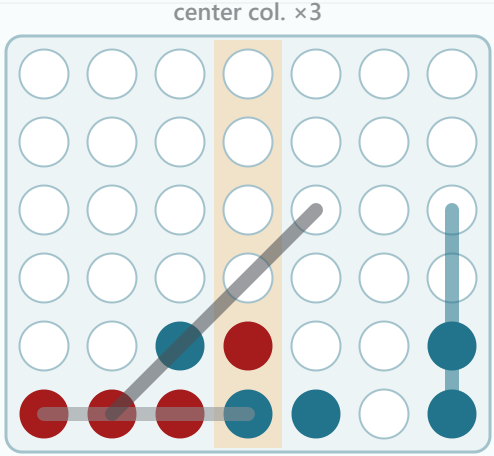
The Evaluation Function

The board has **69 windows** of four adjacent cells: 24 horizontal, 21 vertical, 24 diagonal. Each window scores:

Discs in the window	p(w)
3 red + 1 empty	+5
2 red + 2 empty	+2
3 blue + 1 empty	−5
2 blue + 2 empty	−2
both colors, one disc, or empty	0

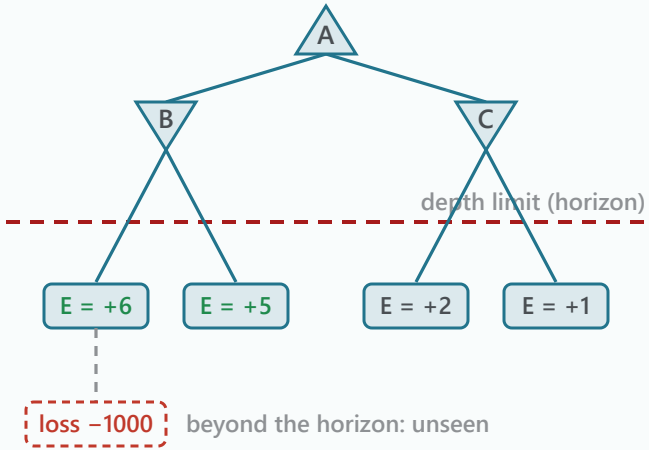
$$E(s) = \Sigma\ p(w) + 3 \cdot (R_{mid} - B_{mid})$$

R_{mid}, B_{mid}: numbers of red and blue discs in the center column, which belongs to the most windows.



The Horizon Effect

- Since the search stops at the depth limit, it **cannot see** events just beyond it: this is the **horizon effect**.
- In the figure the position below B looks best with E = +6; yet one move later an unavoidable loss follows.
- A sneakier form: pointless moves that push an unavoidable loss beyond the horizon (useless checks in chess). The loss does not go away; it just becomes invisible.
- Remedies: search deeper, keep searching in volatile positions (**quiescence search**), extend one more ply when there are threats.
- In Connect Four, an AI playing at depth 1 or 2 cannot see double threats and can be beaten.



The Search in the Lab

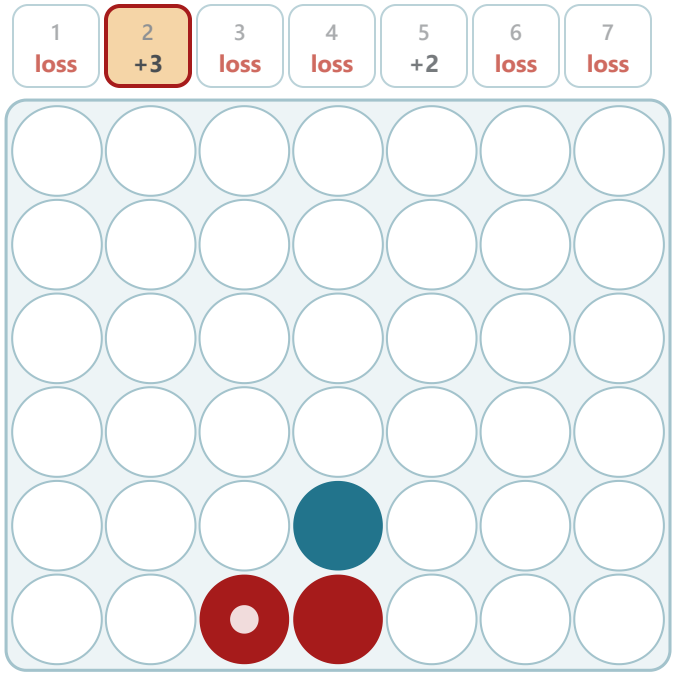
```
from math import inf
def alphabeta(s, d, a, b, maxi):
  if s.over():                  # game over
    return s.utility()            # ±(1000 − k) or 0
  if d == 0:
    return E(s)                 # evaluation
  best = -inf if maxi else inf
  for m in ordered(s.moves()):
    v = alphabeta(s.play(m), d-1, a, b, not maxi)
    if maxi:
      best = max(best, v); a = max(a, best)
    else:
      best = min(best, v); b = min(b, best)
    if a >= b:
      break
  return best
```

```
def best_move(s, d):      # root: MAX
  a, best = -inf, None
  for m in ordered(s.moves()):
    v = alphabeta(s.play(m), d-1,
                  a, inf, False)
    if v > a:
      a, best = v, m
  return best
```

- It is game-independent: only `over`, `utility`, `moves`, `play` and `E` are game-specific.
- At the root, α is carried across the root's children: after the first child, the rest are searched with a narrower window.
- To show each column's score, the lab also runs plain minimax and compares the two results.

CONNECT FOUR

Connect Four Lab



New game: you start | AI starts | AI move

– | Depth 5 | + | Order: center out

Example position: Red played columns 4 and 3, Blue played column 4. The AI's choice: **column 2** (+3). No forced win or loss within the depth limit; the column whose horizon positions score best on the evaluation function (open twos and threes, center column) was chosen. 5 columns lead to a loss within the depth limit.

Minimax: **18,347** nodes, 5.4 ms · Alpha-beta: **1,391** nodes, 0.4 ms · same value, same column ✓

Click a column button to drop the human player's disc. The numbers on the buttons are each column's minimax score from the AI's view; the orange column is the choice. Depth 1–7. Connect Four →

The Effect of Center-Out Ordering

Depth	Minimax	Alpha-beta (left to right)	Alpha-beta (center out)	Root value
1	8	8	8	+3
2	57	33	21	0
3	400	200	76	+6
4	2,801	966	175	0
5	19,608	4,340	627	+6
6	137,257	16,409	1,285	0
7	960,793	67,549	5,086	+8

- Nodes visited searching from the empty board; the numbers are recomputed in this browser when the page opens.
- Minimax grows like 7^d. At depth 7 alpha-beta visits 14× fewer nodes with left-to-right ordering and 189× fewer with **center-out** ordering (4, 3, 5, 2, 6, 1, 7).
- The root value is the same for all three methods: ordering changes only the cost. The center column belongs to the most fours, so good moves are usually central; trying them first narrows α and β early.

Iterative Deepening and Transposition Tables

Iterative deepening

- Search to depth 1, then 2, 3, … until time runs out; then play the move of the last completed search.
- Little work is repeated: the last level outweighs all the others combined (for b = 7 the total is about 7/6 times the work).
- The previous search's best move is tried first: move ordering improves automatically.

Transposition table

- The same position is reached by different move orders (transposition). Search it once and store the result in a hash table.
- Entry: value, search depth, whether the value is exact or a lower/upper bound, best move.
- In tic-tac-toe, 5,478 positions instead of 549,946 nodes. Chess programs map a position to a hash key with Zobrist's (1970) method.

The skeleton of strong game programs: iterative deepening + alpha-beta + good move ordering + transposition table + a good evaluation function.

WRAP-UP

Common Mistakes

Passing α and β the wrong way

The window goes down unchanged from parent to child; only the value comes back up. An α or β updated in a child is not carried to the parent. MAX updates only α, MIN only β.

Counting a pruned branch

Pruned leaves are not read; feeding them into a min or max when solving by hand is wrong. The number returned by a cut-off node is a bound, not an exact value.

Viewpoint and sign errors

E(s) must always be written from the same player's view (here Red/MAX). If the AI moves second it picks the smallest; with negamax the sign flips at every level.

Forgetting the depth penalty

If all wins get the same value, the AI postpones a ready win, and when losing it may pick the fastest loss. Utility must also reflect how many moves away the win is.

WRAP-UP

Game-Tree Search in the Real World

Deep Blue (1997)

IBM's computer beat world champion Kasparov 3½–2½ in a six-game match: parallel alpha-beta on custom chess chips, about 200 million positions per second.

Chess engines

Programs such as Stockfish still use alpha-beta-style search: iterative deepening, transposition tables, strong pruning heuristics; since 2020, neural-network evaluation (NNUE).

Go and MCTS

In Go $b \approx 250$ and a good $E(s)$ is very hard to write. Monte Carlo tree search (Coulom, 2006; UCT: Kocsis and Szepesvári, 2006) evaluates with random playouts. AlphaGo combined MCTS with deep neural networks and beat Lee Sedol 4–1 in March 2016.

The core idea is unchanged: look ahead assuming the opponent plays best. What changed is how much of the tree is searched and how the leaves are evaluated.

WRAP-UP

Summary

Game trees

Nodes are positions, edges moves; MAX ▲ maximizes, MIN ▽ minimizes. Size ≈ b^d.

Minimax

Values rise from leaves to root: max at MAX levels, min at MIN levels. The best move against a perfect opponent; $O(b^d)$ time.

Nim and tic-tac-toe

In Nim, minimax = P/N = the XOR rule. Tic-tac-toe is worth 0; a depth penalty favors quick wins and slow losses.

Alpha-beta

The window [α, β] comes down from above; when α ≥ β the remaining children are pruned. The root value and move are unchanged.

Move ordering

≈ $b^{d/2}$ leaves with the best ordering, b^d with the worst. Center-out ordering saves 189× in Connect Four.

Depth limit

An evaluation function estimates; the horizon effect, iterative deepening and transposition tables.

References

- S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Pearson, 2020 (Chapter 5: Adversarial Search and Games).
- D. E. Knuth, R. W. Moore, “An analysis of alpha-beta pruning”, *Artificial Intelligence* 6(4), 293–326, 1975.
- C. E. Shannon, “Programming a computer for playing chess”, *Philosophical Magazine* 41(314), 1950.
- J. von Neumann, “Zur Theorie der Gesellschaftsspiele”, *Mathematische Annalen* 100, 1928 · H. W. Kuhn, “Extensive games and the problem of information”, *Contributions to the Theory of Games* II, 1953.
- C. L. Bouton, “Nim, a game with a complete mathematical theory”, *Annals of Mathematics* 3, 1901–1902.
- L. V. Allis, *A Knowledge-Based Approach of Connect-Four*, master's thesis, Vrije Universiteit Amsterdam, 1988 · J. Tromp, **John's Connect Four Playground** (position count).
- J. Tromp, G. Farnebäck, “Combinatorics of Go”, *Computers and Games*, 2006.
- J. Tromp, “The Number of Legal Go Positions”, *Computers and Games*, 2016.
- A. L. Zobrist, “A new hashing method with application for game playing”, Tech. Rep. 88, University of Wisconsin, 1970.
- M. Campbell, A. J. Hoane, F. Hsu, “Deep Blue”, *Artificial Intelligence* 134, 57–83, 2002.
- R. Coulom, “Efficient selectivity and backup operators in Monte-Carlo tree search”, *Computers and Games*, 2006 · L. Kocsis, C. Szepesvári, “Bandit based Monte-Carlo planning”, *ECML*, 2006.
- D. Silver et al., "Mastering the game of Go with deep neural networks and tree search", *Nature* 529, 484–489, 2016.

The End

Chapter 2: Minimax and Alpha-Beta Pruning

Previous: Chapter 2: Games and Representations
Next: Chapter 3: Nash Equilibrium