3010

ARTIFICIAL INTELLIGENCE

Masashi Shimbo

2019-05-17

Agenda

- Adversarial search
- Minmax search
- $\alpha\beta$ pruning
- Heuristic evaluation function

Adversarial search

So far, we have concerned with single-agent search

This lecture is concerned with search when another, adversarial, agent is present

…in particular, **two-player games**

Type of two-player game we consider:

“zero-sum”

one player’s gain (loss) is the other’s loss (gain)

“perfect information”

complete state of the game is observable

chess, go, othello, backgammon, …

cf. poker is an imperfect information game

—we do not know the opponents’ hand

alternate move

player and opponent make moves alternately

Our goal is to design a game-playing agent

Given a state of a game,

the agent must be able to tell a “good” move from a “bad” move

—but what makes a move “good” or “bad”?

What is a "good" move? Here is our definition

A good move is the one that eventually brings the player some **payoff**:

- In some games, winning/losing only counts
- In other games, there maybe different levels of winning/losing points (e.g., backgammon)

Such difference is abstracted away by defining "payoff" as a certain numeric value ($+1$ for winning, -1 for losing, 0 for tie, etc.)

Game playing as search

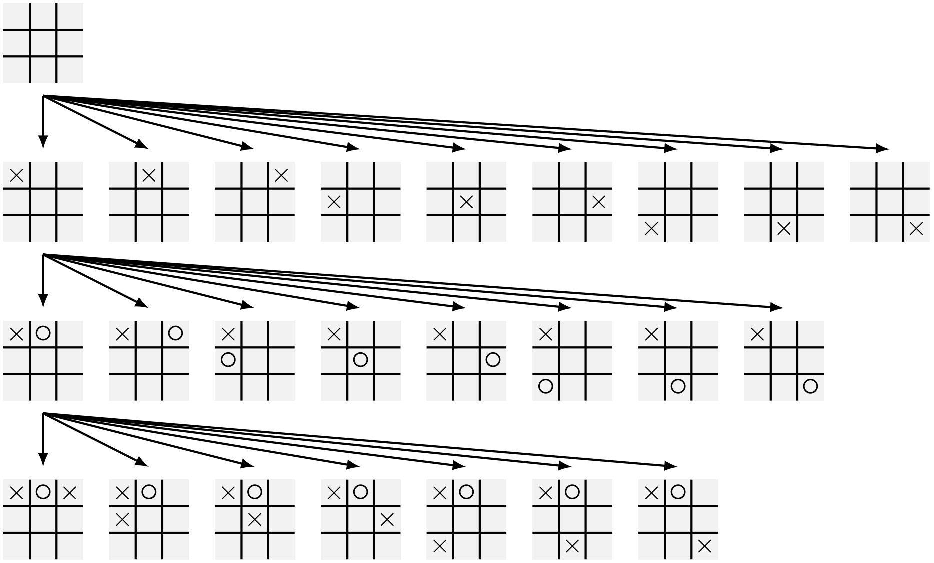
To find a good move, we “look ahead”

“What if I make this move? What will the opponent’s next would be? And what move should I make after she makes that move?”

Or, we “search” in the **game tree** from the current state

game tree = possible progression of game drawn as a tree (state space)

(Partial) game tree for Tic-Tac-Toe



Game playing as search

Initial state s

current board position

Successor function $\mathrm{Succ}(v)$

defines legal moves at state v and its outcome

Terminal test $\mathrm{IsTerminal}(v)$

tells whether state v is a terminal (game is over, win or lose)

Payoff (utility) function $\mathrm{Payoff}(t)$

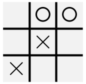
numeric outcome of the game at terminal state t

Note: Payoff is given only at terminal states.
There is no payoff for non-goal states.

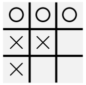
Example of states, terminal states, payoff

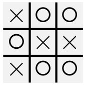
Tic-tac-toe

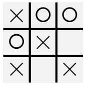
current state



terminal states







payoff -1 0 $+1$

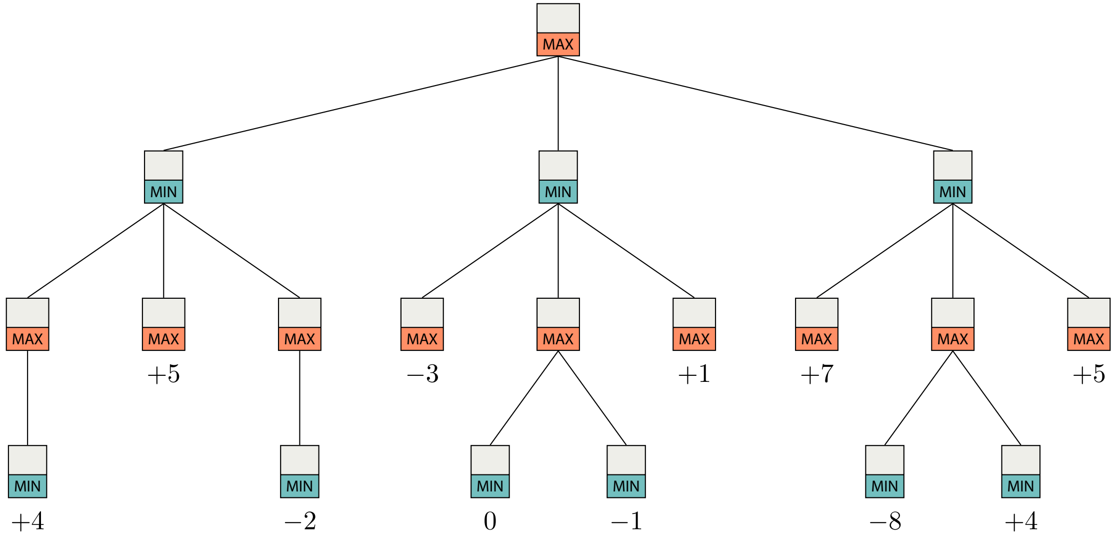
Objective of game tree search

Find which move to make at the current state (=root of the game tree) so that we can have a good amount of payoff

MAX Let’s call our agent (player) “Max”

MIN And let’s call the opponent “Minnie”

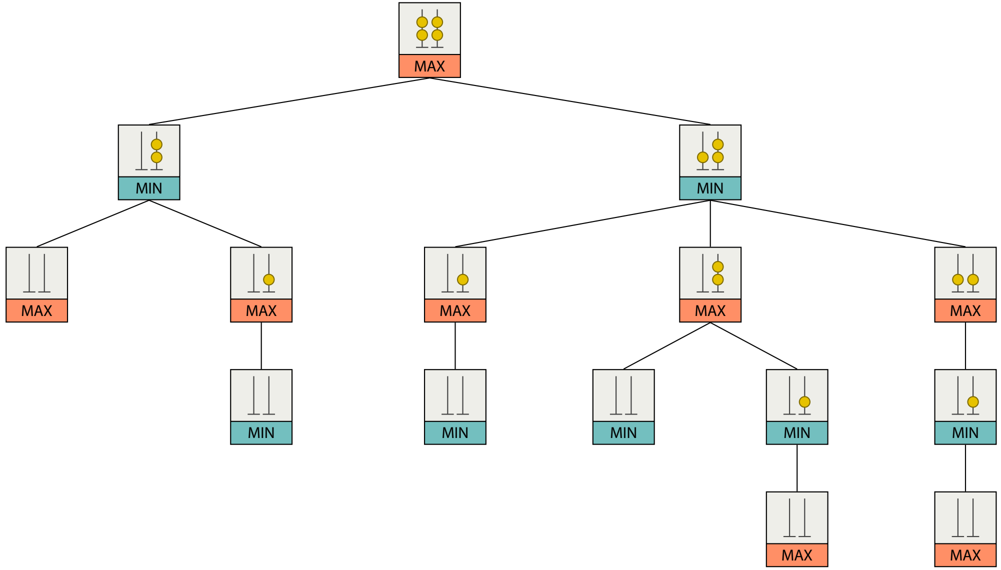
Game tree


MAX
MIN
MIN
MIN
MAX
MAX
MAX
MAX
MAX
MAX
MAX
MAX
MAX
+5
−3
+1
+7
+5
MIN
MIN
MIN
MIN
MIN
MIN
+4
−2
0
−1
−8
+4

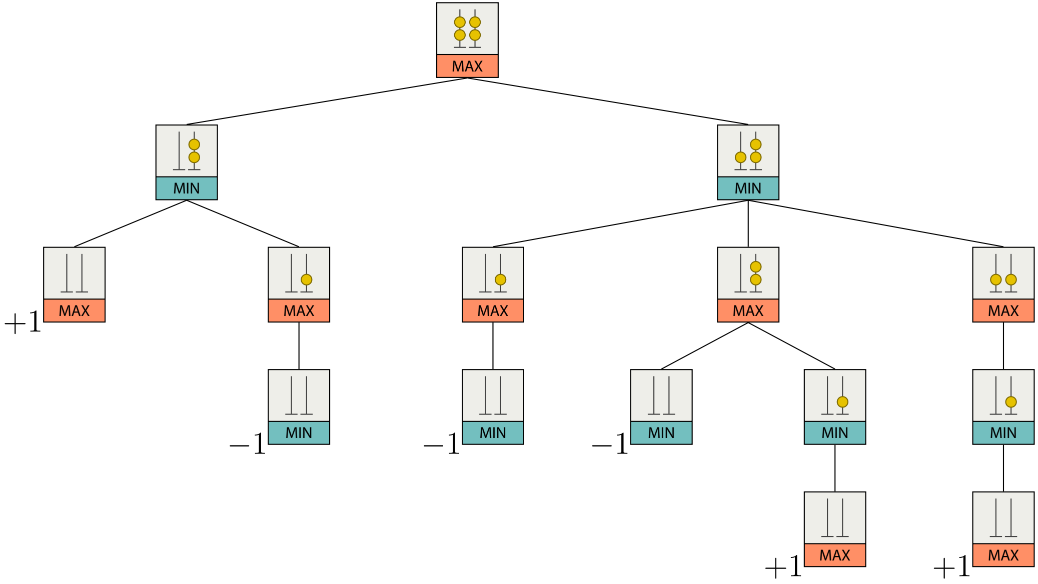
Game of Nim

- Two-player zero-sum perfect information game
- Each player makes a move alternately
- There are several heaps of coins
  (possibly each heap has a different number of coins)
- On each turn, a player takes one or more coins from a heap
- The player who takes the last coin **loses**

Game tree for Nim with 2 heaps of 2 coins



Game tree for Nim with 2 heaps of 2 coins



Difference from single-agent search

- ► No cost incurred by taking a move (=action)
- ► Instead payoff ($\simeq$ reward/cost/...) is given at terminal states
- ► We don't know which move the opponents will make after our move!

Then, what do you want to find with the game tree?

➥ We want to find the "best" move to make at the current (root) node

But that depends on the opponent's move. What kind of opponent do you assume?

➥ We assume she's the best player (i.e., worst for us)

Minimax search: Basic Idea

We are interested in the **“minimax”** strategy:

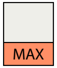
Assume the opponent (Minnie) will make the best moves to **mini**mize the payoff to the player (Max)

- What is the **maxi**mum payoff possible for Max?
  (= “minimax” value)
- What is the best immediate move to get the maximum payoff? (= “minimax” move/successor)

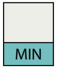
Max node/min node

Every node can be classified into either max node or min node

- The state at which the player makes a move is called **max node** (because at this node we choose an action (move) that maximizes the payoff)
- The state at which the opponent makes a move is called **min node** (because at this node we assume the opponent chooses an action (move) that minimizes the payoff)



Max node is exactly where our agent “Max” makes a move



Min node is exactly where the opponent “Minnie” makes a move

Minimax value of a node: Definition

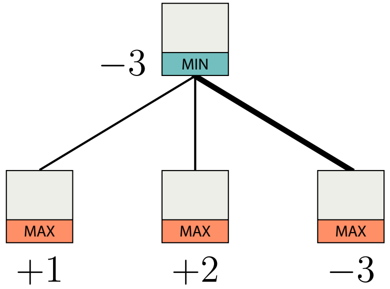
- Minimax value of a terminal node t is the $\mathrm{Payoff}(t)$
- Minimax value of a non-terminal Max node (MAX) is the **maximum** among the minimax value of its successors
- Minimax value of a non-terminal Min node (MIN) is the **minimum** among the minimax value of its successors

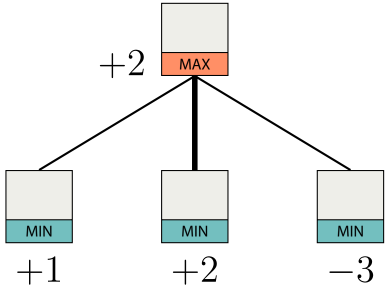
Interpretation:

Minimax value at a node defines the “best” possible payoff each player can obtain, provided that both players make optimal moves in the future

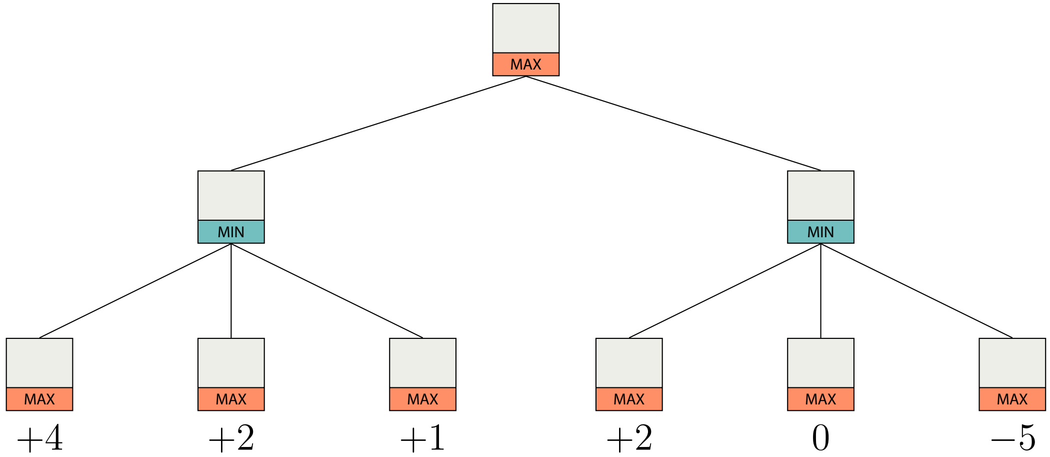
Computing minimax values

By definition, minimax values are computed **bottom-up** (first at terminal nodes , then their parents, …)

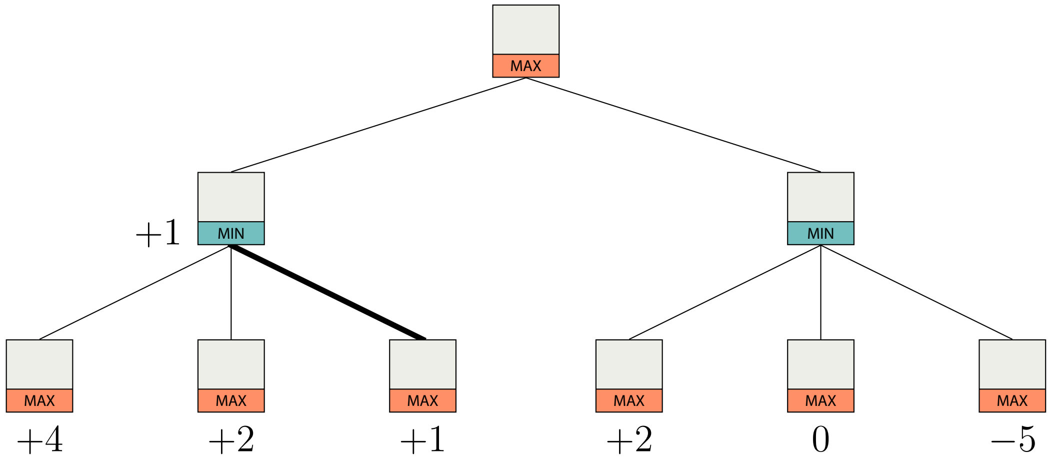




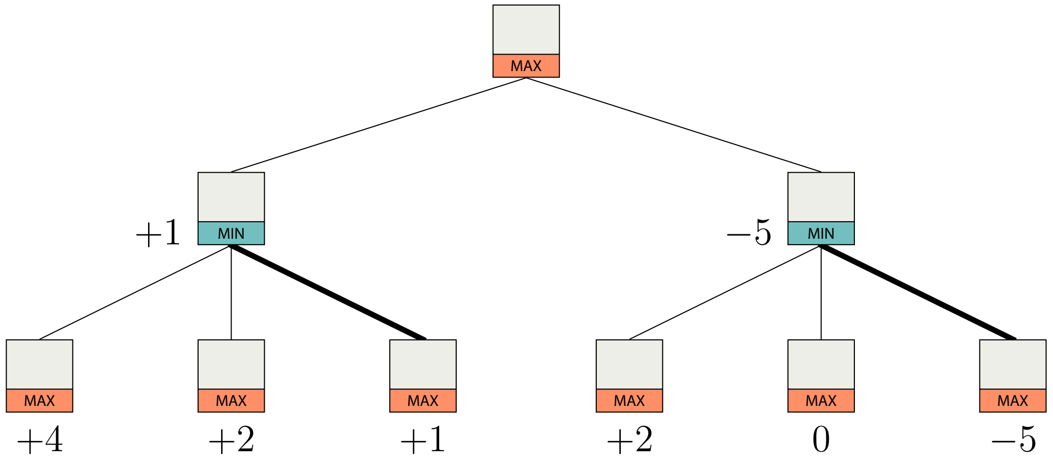
Computing minmax values: example



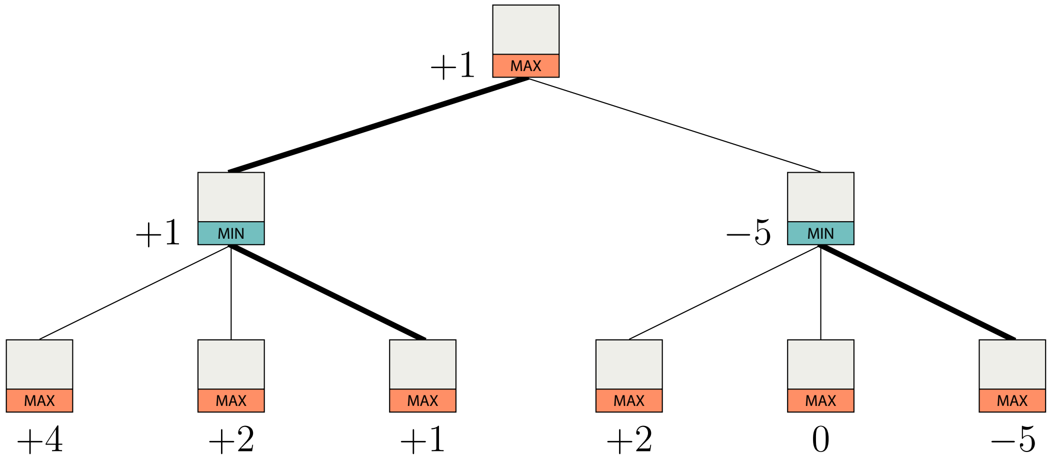
Computing minmax values: example



Computing minmax values: example



Computing minmax values: example



Minimax value and win/loss game

In win/loss game (payoff = $+1$ for win, -1 for loss),

- If the minimax value of the initial node $= +1$
  ➥ there is a winning move for the player (Max)
- If the value $= -1$,
  ➥ there is a winning move for the opponent (Minnie)

Minimax depth-first search

A variation of depth-first search (DFS)

- state space in game search is not necessarily a tree
- state space may be infinite for some games (chess, etc.)
- but DFS is often preferred because it is more compatible to how minimax values are computed (i.e., bottom-up)

DFS is preferable because

- payoff is only defined for terminal (leaf) nodes
- minimax value of a non-leaf node is computed by taking the minimum or maximum of the successor nodes

Run DFS until a leaf (terminal) is met, and then compute (“backup”) the minimax value of non-leaf nodes upon backtracking

Note that unlike single-agent DFS, we don’t stop searching even if a terminal node is met

➥ we continue until all nodes in the game tree is exhausted

```
function $\mathrm{MaxValue}(v)$
    if $\mathrm{IsTerminal}(v)$ then return $\mathrm{Payoff}(v)$
    $m \leftarrow -\infty$
    foreach $u \in \mathrm{Succ}(v)$ do
        $m \leftarrow \max(m, \mathrm{MinValue}(u))$
    return $m$
```

```
function $\mathrm{MinValue}(v)$
    if $\mathrm{IsTerminal}(v)$ then return $\mathrm{Payoff}(v)$
    $m \leftarrow +\infty$
    foreach $u \in \mathrm{Succ}(v)$ do
        $m \leftarrow \min(m, \mathrm{MaxValue}(u))$
    return $m$
```

function MiniMax(s)

- returns the minimax successor of the root node s
- nearly identical to MaxValue(s), but returns the minimax successor, not minimax value
- red lines indicate major changes from MaxValue

```
function MiniMax($s$)
    $u_{\text{max}} \leftarrow$ nil                      # $u_{\text{max}}$ maintains the best successor found
    $\alpha \leftarrow -\infty$
    foreach $u \in \text{Succ}(s)$ do
        $m \leftarrow \text{MinValue}(u)$
        if $m > \alpha$ then
            $\alpha \leftarrow m$
            $u_{\text{max}} \leftarrow u$
    return $u_{\text{max}}$                              # return the minimax successor, not value
```

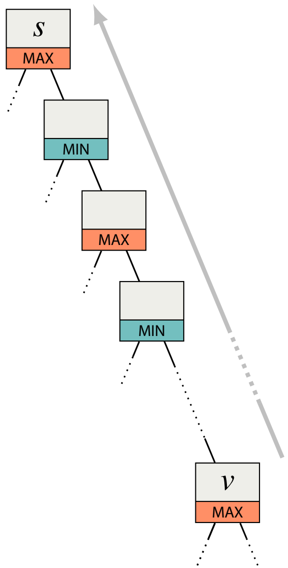
α-β pruning

A “branch-and-bound” technique for minimax search

Minimax DFS traverses the entire game tree to find the minimax value of the root node

Can we avoid traversing the entire tree

by not exploring subtrees that have no influence on the solution???

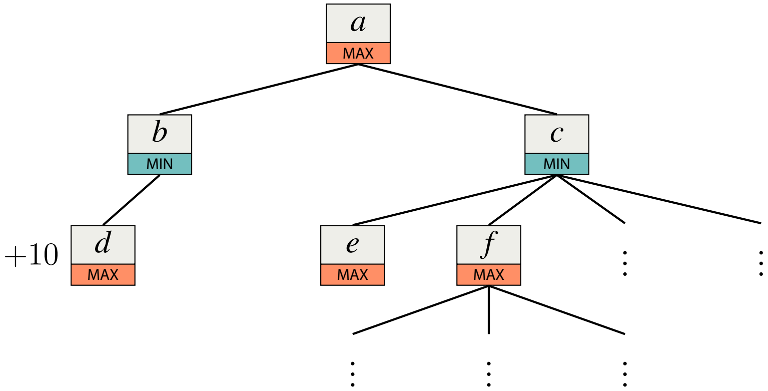


When does a non-root node's minimax value becomes that of the root node?

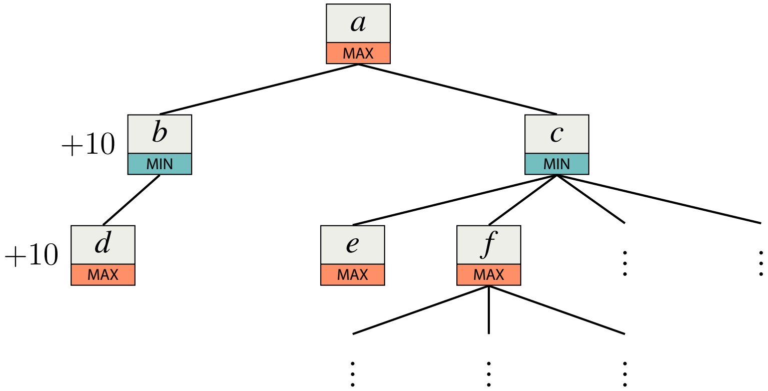
For node v's minimax value to be backed up to the root node, it has to be

- greater than or equal to the values of any children of Max nodes along the path from root, and
- less than or equal to the values of any children of Min nodes along the path

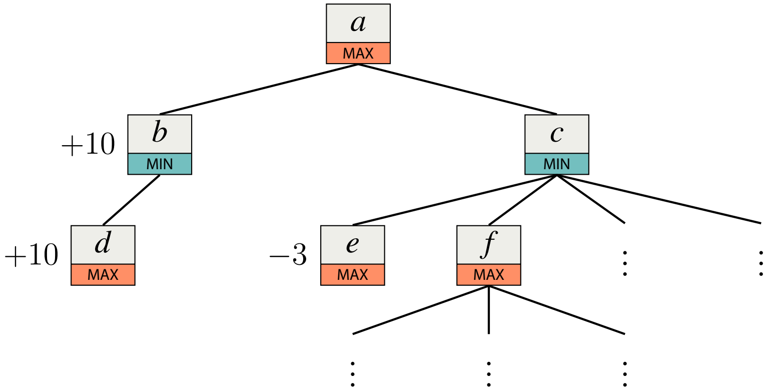
We can stop searching below v if we find v's minimax value cannot meet either of these conditions

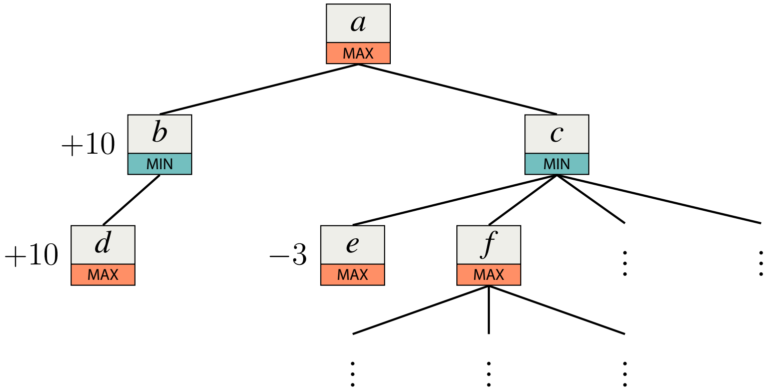


a
MAX
b
MIN
c
MIN
$+10$
d
MAX
e
MAX
f
MAX

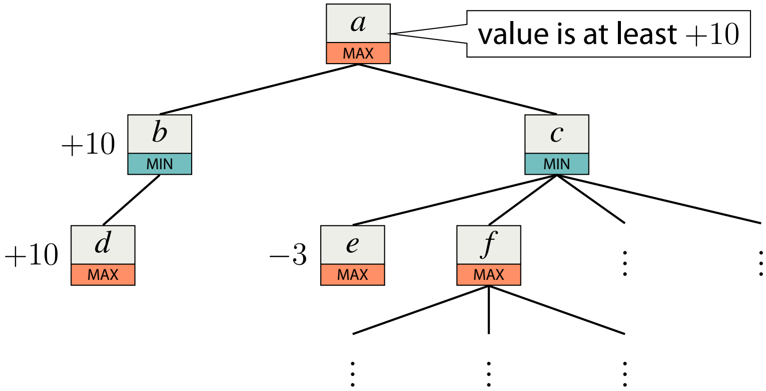


a
MAX
$+10$
b
MIN
c
MIN
$+10$
d
MAX
e
MAX
f
MAX

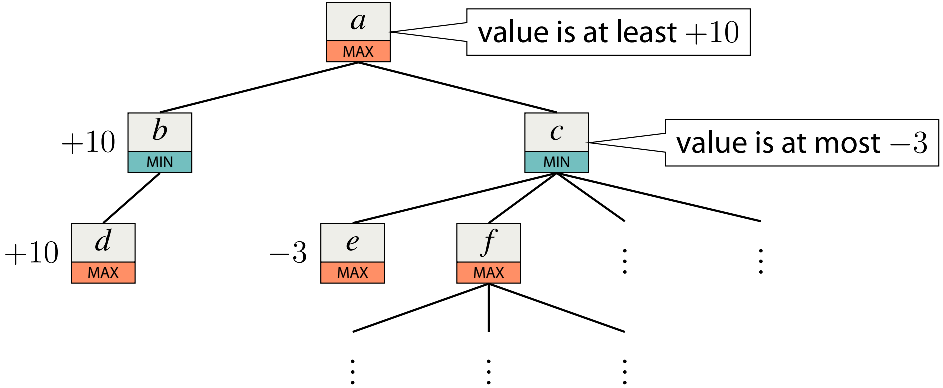

a
MAX
+10
b
MIN
c
MIN
+10
d
MAX
−3
e
MAX
f
MAX



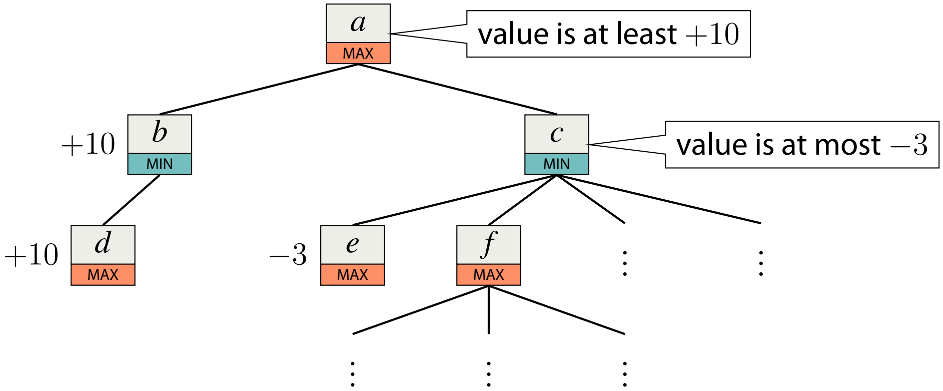
Next step would be to search node f and the subtree underneath —but do we really have to?



Next step would be to search node f and the subtree underneath —but do we really have to?



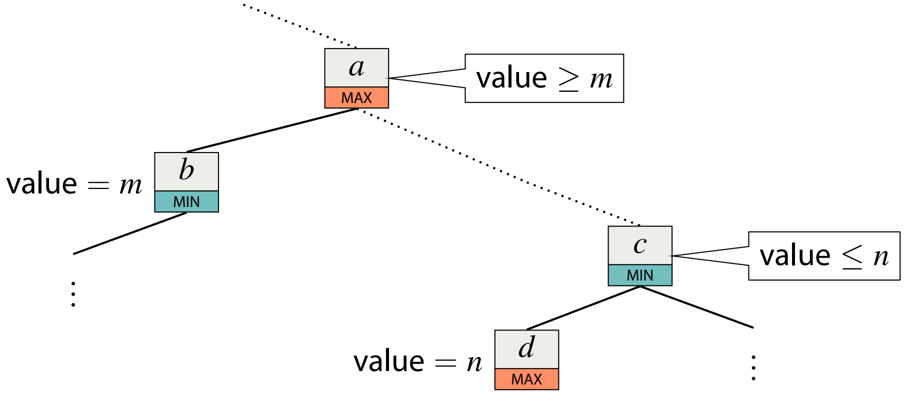
Next step would be to search node f and the subtree underneath —but do we really have to?



Next step would be to search node f and the subtree underneath
—but do we really have to?

Backing up the minimax value of c upward will never exceed the value at node a
➥ no use in searching the remaining descendants of c

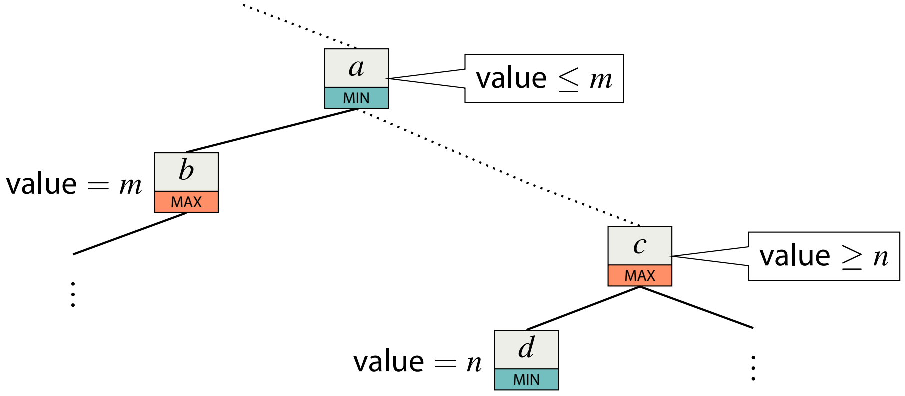
General case (i)



if $n \leq m$, searching the remaining descendants of c is worthless

$\because$ we know that value$(c) \leq n$ regardless of the search outcome, and it cannot exceed $m \leq$ value(a)

Case (ii)



if $m \leq n$, searching the remaining descendants of c is worthless

$\because$ we know that $n \leq$ value(c) regardless of the search outcome, and it cannot be smaller than value$(a) \leq m$

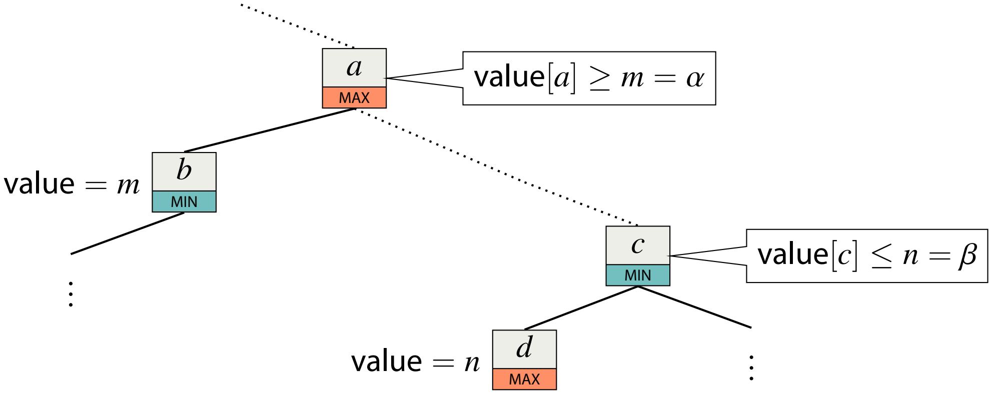
α and β values at node v

α best (higest) value found for Max nodes along the path from root to v
— value at v must exceed α to impact the value of the root node

β best (lowest) value found for Min nodes along the path from root to v
— value at v must be lower than β to impact the value of the root node

If lower bound (α) $\geq$ upper bound (β) at a node v, we can prune the search below v

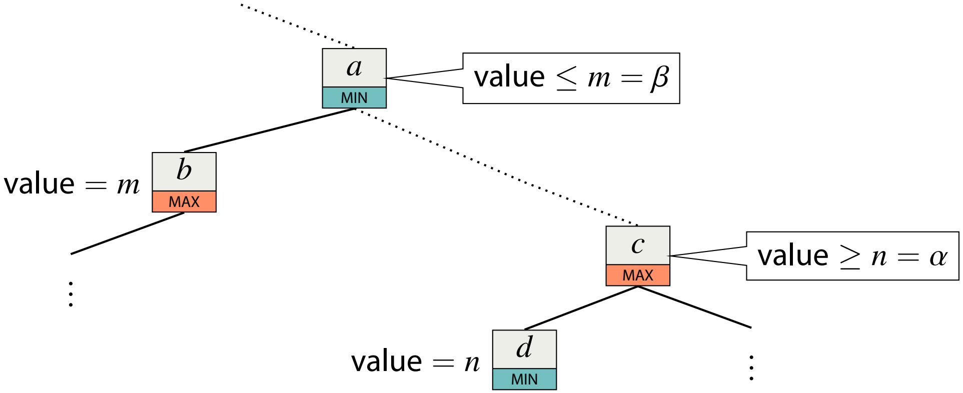
Case (i)


a
MAX
$\text{value}[a] \geq m = \alpha$
value $= m$
b
MIN
c
MIN
$\text{value}[c] \leq n = \beta$
value $= n$
d
MAX

if $n \leq m$, searching the remaining descendants of c is worthless

$\because$ we know that value$[c] \leq n$ regardless of the search outcome, and it cannot exceed $m \leq$ value$[a]$

Case (ii)



if $m \leq n$, searching the remaining descendants of c is worthless

$\because$ we know that $n \leq \text{value}(c)$ regardless of the search outcome, and it cannot be smaller than $\text{value}(a) \leq m$

```
function AlphaBetaMaxValue($v, \alpha, \beta$)
    if IsTerminal($v$) then return Payoff($v$)
    $m \leftarrow -\infty$
    foreach $u \in$ Succ($v$) do
        $m \leftarrow \max(m$, AlphaBetaMinValue$(u, \alpha, \beta))$
        $\alpha \leftarrow \max(\alpha, m)$
        if $\alpha \geq \beta$ then return $\alpha$
    return $m$
```

```
function AlphaBetaMinValue($v, \alpha, \beta$)
    if IsTerminal($v$) then return Payoff($v$)
    $m \leftarrow +\infty$
    foreach $u \in$ Succ($v$) do
        $m \leftarrow \min(m$, AlphaBetaMaxValue$(u, \alpha, \beta))$
        $\beta \leftarrow \min(\beta, m)$
        if $\alpha \geq \beta$ then return $\beta$
    return $m$
```

function AlphaBetaMiniMax(s)

- returns the minimax successor of the root node s
- nearly identical to MiniMax(s) for vanilla minimax search, but utilizes $\alpha\beta$-pruning
- Changes from MiniMax are indicated in red

```
function AlphaBetaMiniMax($s$)
    $u_{\text{max}} \leftarrow$ nil
    $\alpha \leftarrow -\infty$
    foreach $u \in \text{Succ}(s)$ do
        $m \leftarrow \text{AlphaBetaMinValue}(u, \alpha, +\infty)$
        if $m > \alpha$ then
            $u_{\text{max}} \leftarrow u$
            $\alpha \leftarrow m$
    return $u_{\text{max}}$                # return the minimax successor, not value
```

Maintaining α and β values

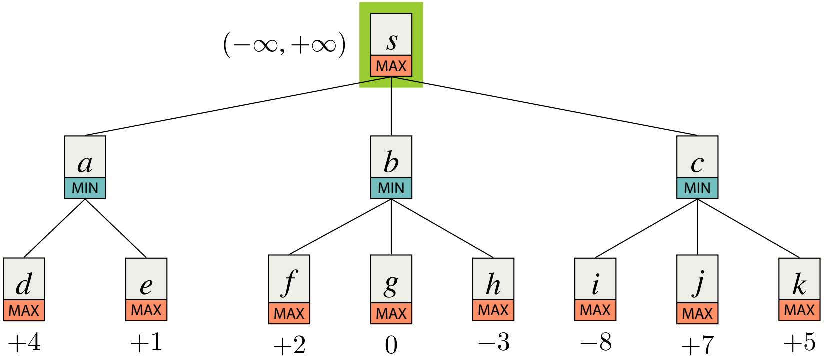
α best (higest) value found for Max nodes along the path from root to v

β best (lowest) value found for Min nodes along the path from root to v

- Push down α, β as we go deeper in the tree
- When a new terminal is found, return its payoff value to the parent node.
- Upon receiving a value from a child, the parent node
  - updates only α if it is a Max node
  - updates only β if it is a Min node

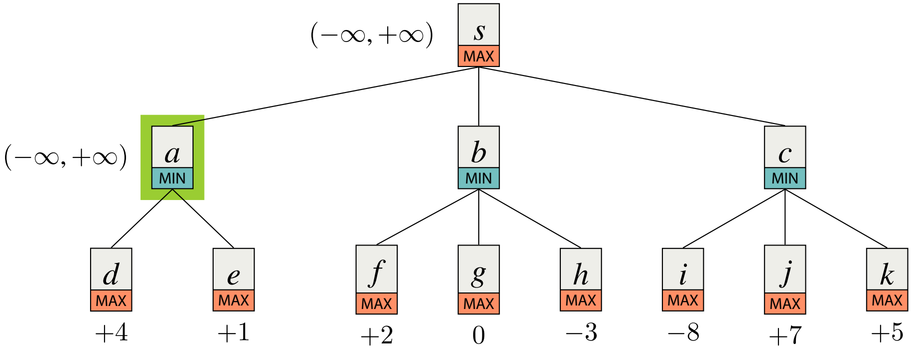
α-β pruning: sample run

numbers next to a node represent its (α, β) values



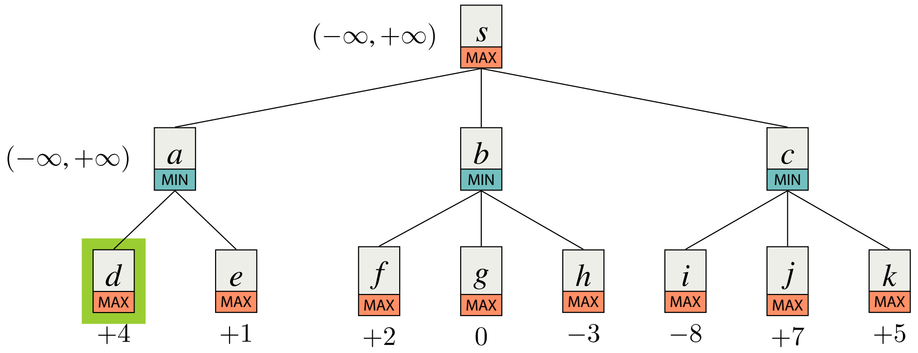
α-β pruning: sample run

numbers next to a node represent its (α, β) values



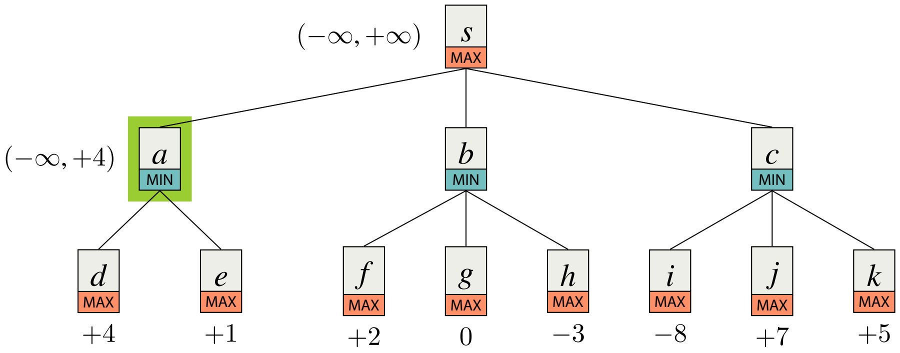
α-β pruning: sample run

numbers next to a node represent its (α, β) values



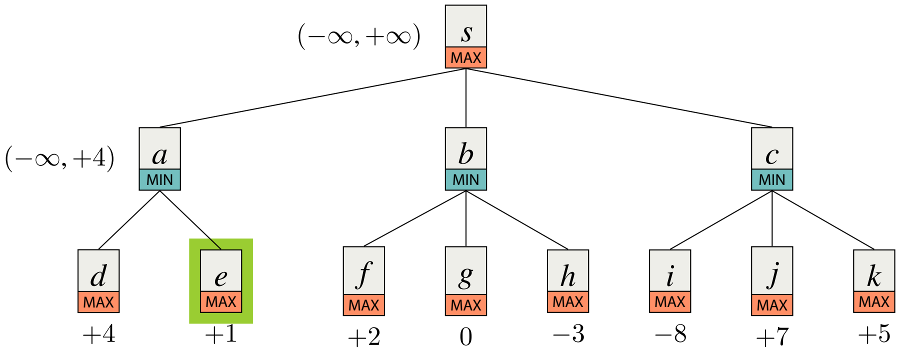
α-β pruning: sample run

numbers next to a node represent its (α, β) values



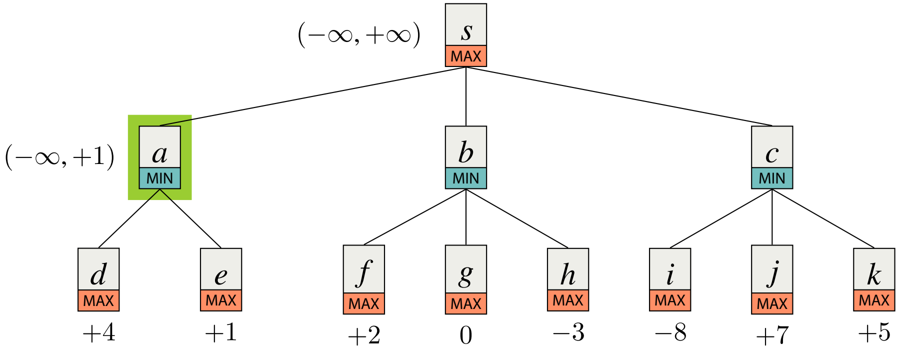
α-β pruning: sample run

numbers next to a node represent its (α, β) values



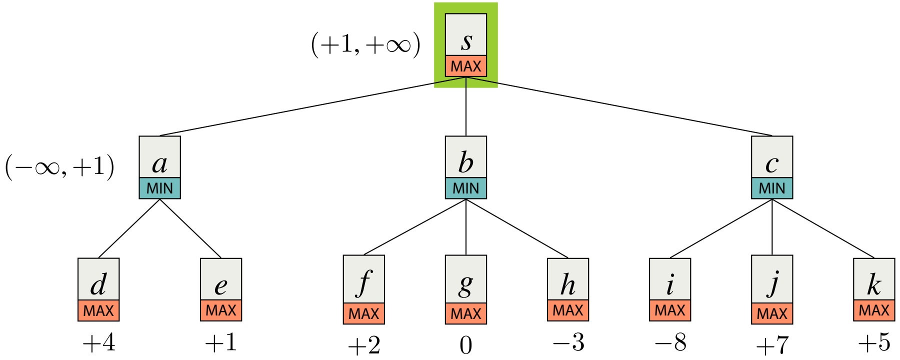
α-β pruning: sample run

numbers next to a node represent its (α, β) values



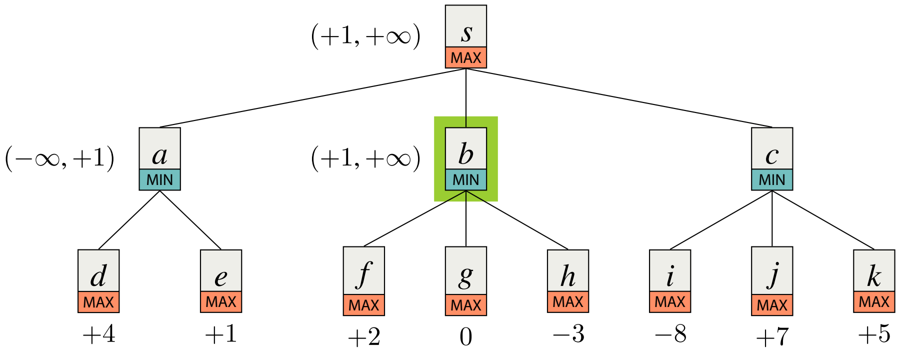
α-β pruning: sample run

numbers next to a node represent its (α, β) values



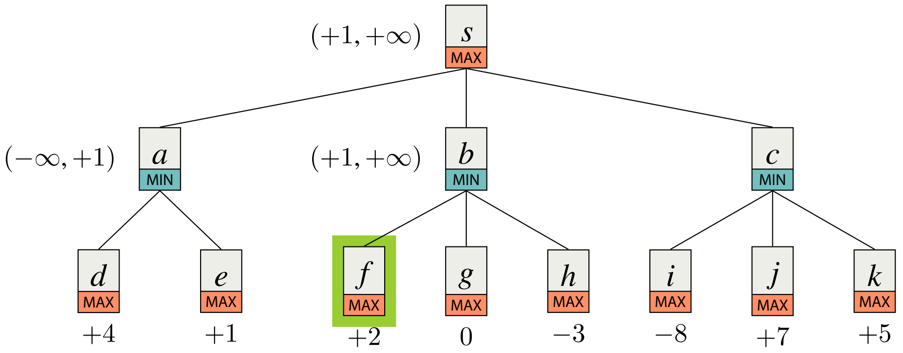
α-β pruning: sample run

numbers next to a node represent its (α, β) values



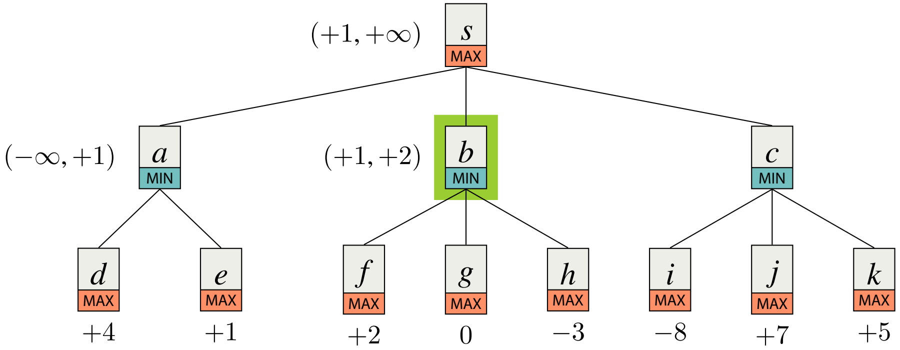
α-β pruning: sample run

numbers next to a node represent its (α, β) values



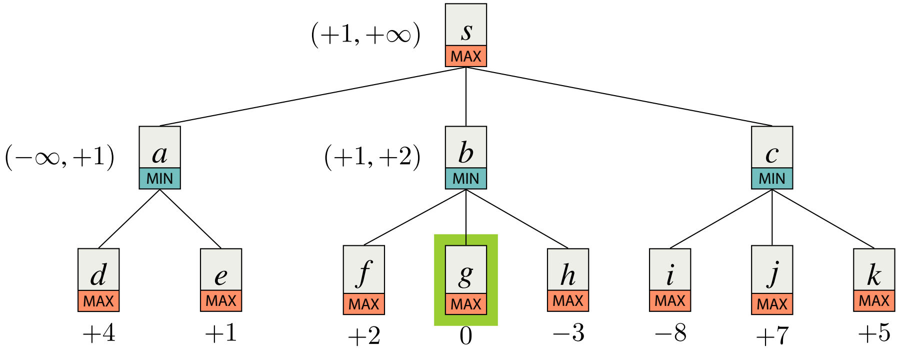
α-β pruning: sample run

numbers next to a node represent its (α, β) values



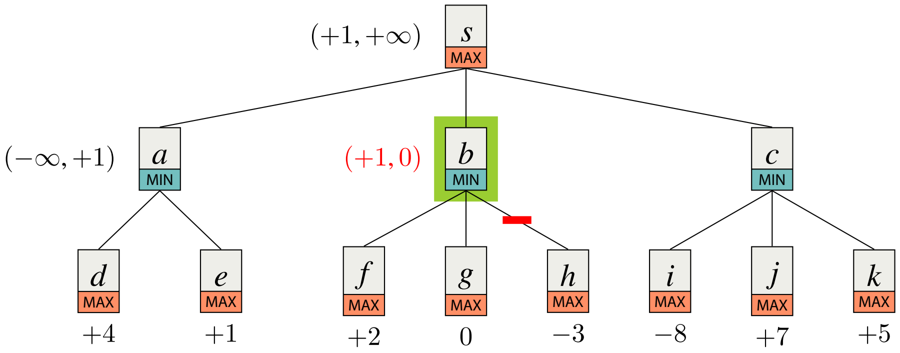
α-β pruning: sample run

numbers next to a node represent its (α, β) values



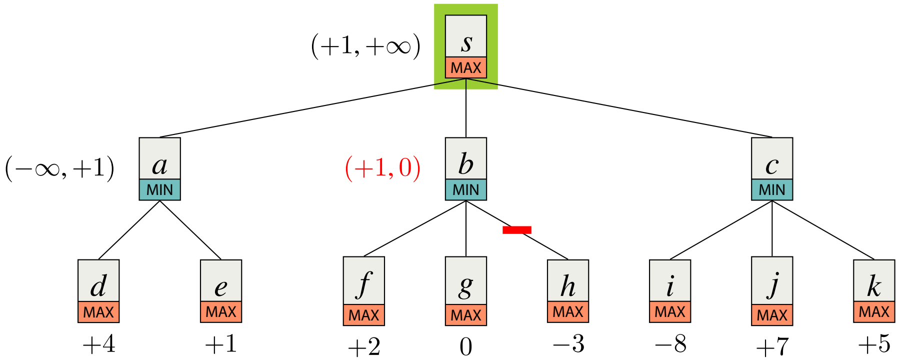
α-β pruning: sample run

numbers next to a node represent its (α, β) values



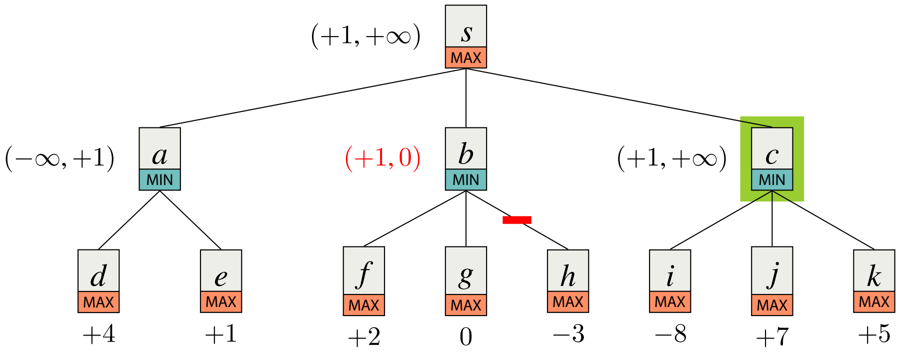
α-β pruning: sample run

numbers next to a node represent its (α, β) values



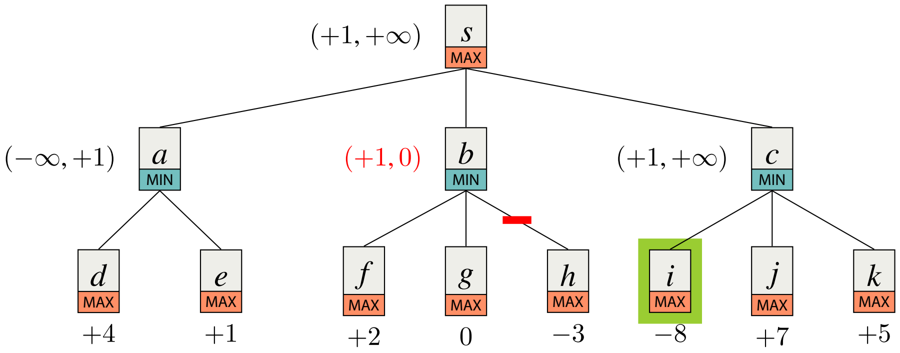
α-β pruning: sample run

numbers next to a node represent its (α, β) values



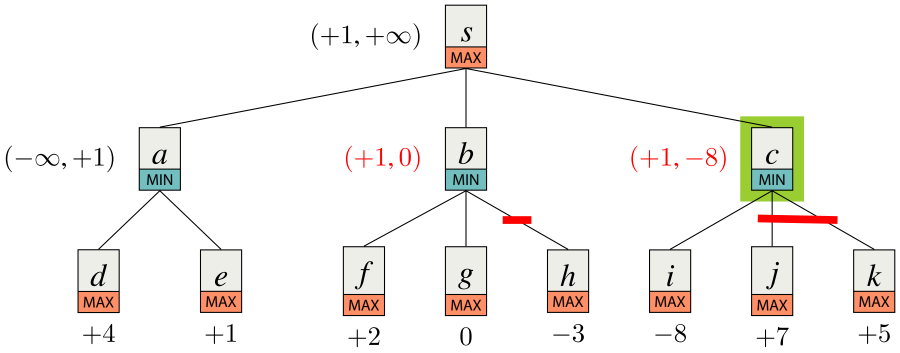
α-β pruning: sample run

numbers next to a node represent its (α, β) values



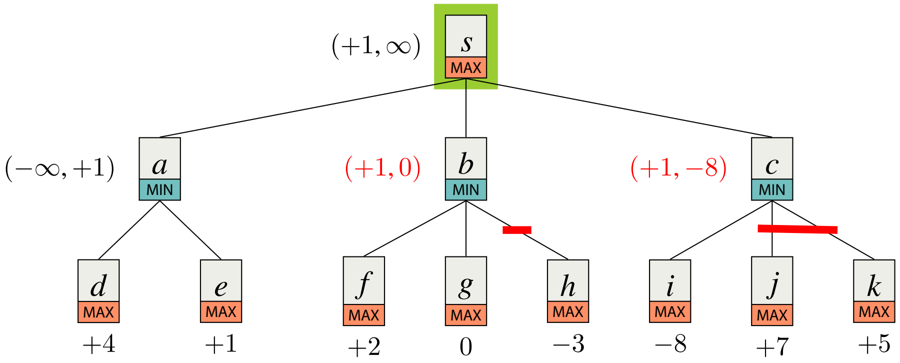
α-β pruning: sample run

numbers next to a node represent its (α, β) values



α-β pruning: sample run

numbers next to a node represent its (α, β) values



How effective is $\alpha\beta$ pruning?

The number of pruned nodes depends on the order of nodes traversed

In a tree with branching factor b and depth n (= $O(b^n)$ nodes),

- In the best case, only $O(b^{n/2})$ nodes need to be examined
  - Depth of trees that can be searched in a fixed amount of time is doubled (since $b^{2n/2} = b^n$)
  - Equivalently, branching factor is effectively reduced to b to $\sqrt{b}$ of minimax search (since $b^{n/2} = \sqrt{b^n}$)
- In the worst case, no nodes are pruned

Need to make imperfect decisions

Minimax procedure (with or without α-β pruning) assumes that the agent has enough time to exhaustively search the entire tree

But this is not usually the case (consider Chess, Go, etc.) the agent must make decision in a limited amount of time.

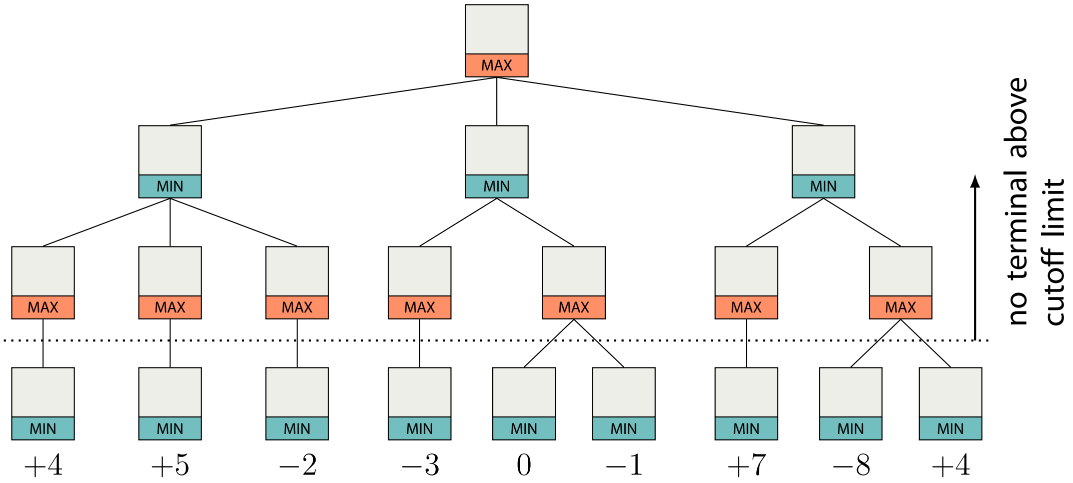
Use depth-limited search/iterative deepening

You could do depth-limited search (or iterative deepening)

But what shall we do when we don’t find any terminal node within the given amount of time?

—recall that Payoff is only defined over terminal state, and thus we cannot decide which move is better or worse

Suppose cutoff depth = 2



Then we cannot decide on which move is better at root node

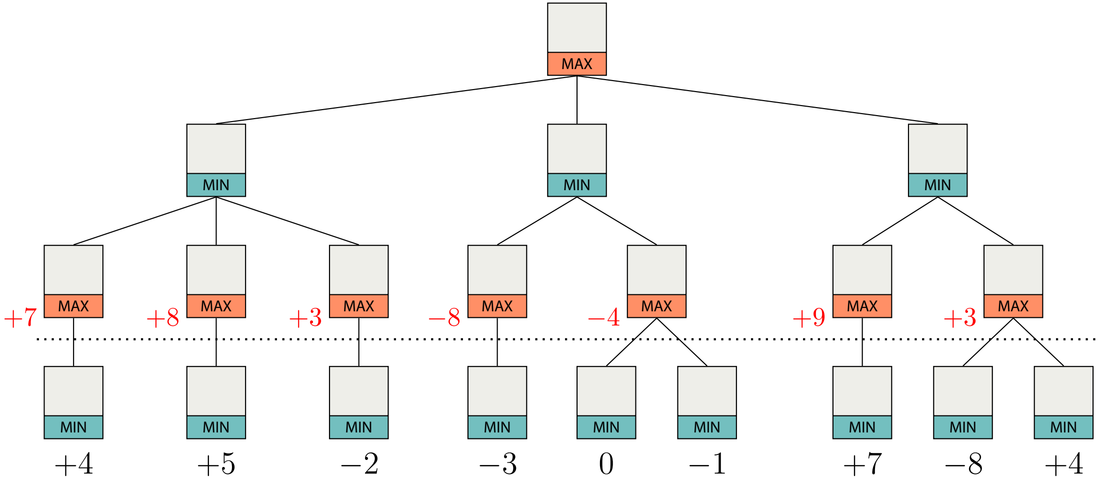
Heuristic evaluation function

We assume each node v has a (heuristic) estimate that gives the figure of merit of each state

$$\mathrm{HeuristicEval}(v)$$

If a fixed bound (depth, etc.) is reached at node v, regard v as if it were a terminal node, using $\mathrm{HeuristicEval}(v)$ as $\mathrm{Payoff}(v)$

We now have heuristic estimates for non-terminal nodes



Treat cutoff nodes as if it were terminals, using their heuristic estimates as payoff at the nodes

Changes needed for introducing depth bound and heuristic function

- Maintain depth d of nodes as they are examined
- $\text{CutoffTest}(v, d)$ returns true if current depth d $=$cutoff depth
- Change
  - $\text{IsTerminal}(v) \longrightarrow \text{CutoffTest}(v, d)$
  - $\text{Payoff}(v) \longrightarrow \text{HeuristicEval}(v)$

Weighted linear function of features

Conventionally, heuristic functions are defined as a weighted linear function of **features**

Convert state v as a vector of features

$$f(v) = [f_1(v), f_2(v), \ldots, f_m(v)]$$

where $f_i(v)$ is the ith feature extractor

For example, in Chess, features might include

$$\begin{aligned} f_1(v) &= (\text{number of white pawns on the board}) \\ f_2(v) &= (\text{number of black pawns on the board}) \\ \vdots &= \quad \vdots \end{aligned}$$

and many diverse pieces of information that are considered relevant to the evaluation of chess positions

Weighted linear function of features (continued)

Then define heuristic function as a linear function of these features

$$\begin{aligned}\text{HeuristicEval}(v) &= w_1 f_1(v) + w_2 f_2(v) + \cdots + w_m f_m(v) \\ &= \sum_{i=1}^{m} w_i f_i(v)\end{aligned}$$

where $w_1, \ldots, w_m$ are the “weights” measuring the relative importance of features

Unfortunately, in game tree search, we do not have a useful class of heuristic evaluation function that gives a performance guarantee, in a way that admissible heuristic functions guarantees A* its admissibility

Moreover, making a good heuristic evaluation function is not easy

Summary

- A game can be formulated as a search problem in game tree
- State space in game (game tree) is defined by the initial state (representing the current setup in the game), legal moves (actions) in each state, and terminal test, and payoff (utility) function

Summary (continued)

- Minimax value is the best payoff a player can receive when the opponent moves optimally (to reduce thee player's payoff)
- Minimax DFS algorithm can find the minimax value/successor at the initial state:
- Alpha-beta pruning can also compute the same minimax value/successor, but prunes subtrees that are found to be irrelevant

Summary (continued)

- It is impractical to assume the whole game tree is searchable (even with alpha-beta)
- Use depth-limited search/iterative deepening
- At the cutoff node, use heuristic evaluation function (defined for non-terminal nodes) to compute its figure of merit, and use it as if it were the payoff value (defined for terminal node only) at that node