Alpha-Beta Pruning & MCTS

Outline

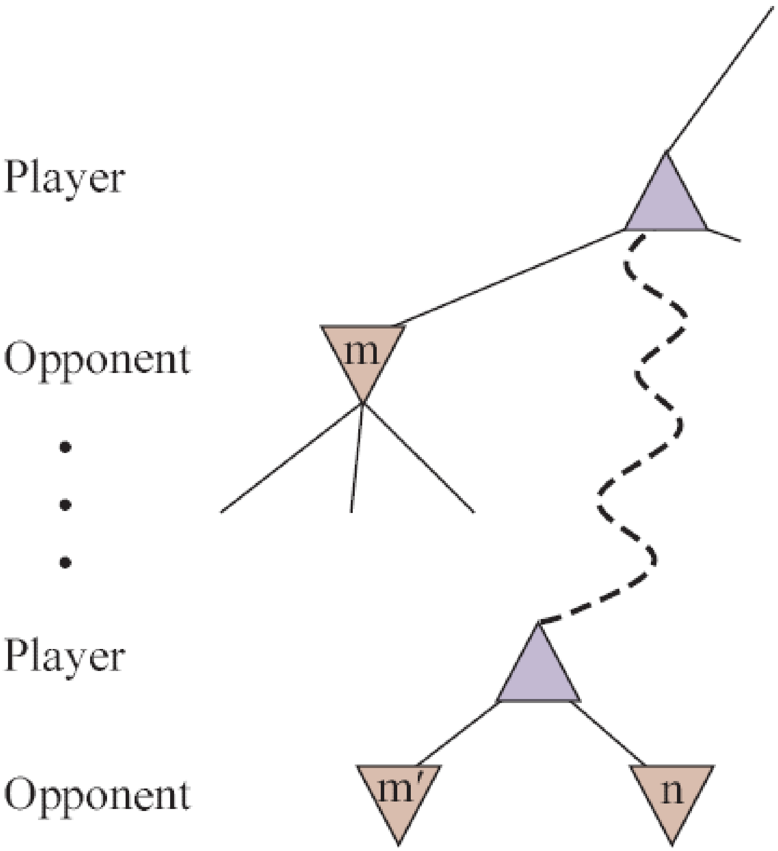
I. Alpha-beta pruning algorithm

II. Heuristic alpha-beta tree approach

III. Monte Carlo tree search (MCTS)

* Figures/images are from the textbook site (or by the instructor). Otherwise, the source is cited unless such citation would make little sense due to the triviality of generating the image.

I. Node Pruning



The player will not move to node n if it has a better choice

- either at the same level (e.g., m')
- or at any node (e.g., m) higher up in the tree.

Prune n once we have found enough about it to reach the above conclusion.

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

$\alpha =$ the highest-value (i.e., the best choice) so far along a path for MAX.

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

$\alpha =$ the highest-value (i.e., the best choice) so far along a path for MAX.

eventual value $\geq \alpha$ "at least"

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

$\alpha =$ the highest-value (i.e., the best choice) so far along a path for MAX.

eventual value $\geq \alpha$ "at least"

$\beta =$ the lowest-value (i.e., the best choice) so far along a path for MIN.

eventual value $\leq \beta$ "at most"

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

$\alpha =$ the highest-value (i.e., the best choice) so far along a path for MAX.

eventual value $\geq \alpha$ "at least"

$\beta =$ the lowest-value (i.e., the best choice) so far along a path for MIN.

eventual value $\leq \beta$ "at most"

- Update the values of α and β as the search goes along.
- Prune the remaining branches at a MIN node with current value $\leq \alpha$ or at a MAX node with current value $\geq \beta$.

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

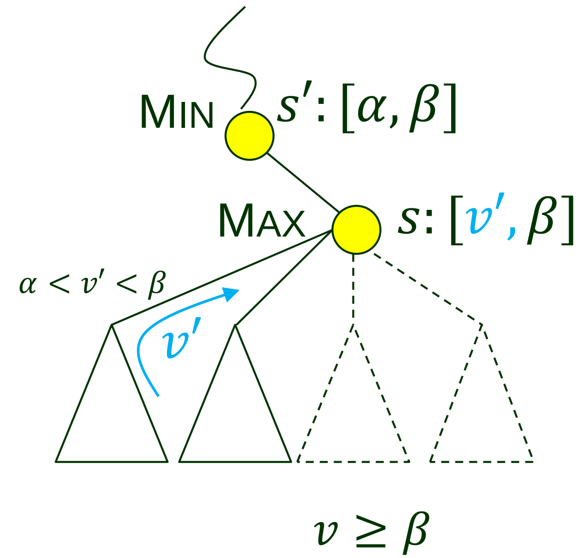
function ALPHA-BETA-SEARCH(*state*) **returns** an action
v ←MAX-VALUE(*state*, $-\infty, +\infty$)
return the action in ACTIONS(*state*) with value v

function MAX-VALUE (*state*,α, β) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow -\infty$
for each a in ACTIONS(*state*) **do**
v ←MAX(v, MIN-VALUE(RESULT(s, a),α, β))
if $v \geq \beta$ **then return** v
α ←MAX(α, v)
return v

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow +\infty$
for each a in ACTIONS(*state*) **do**
v ←MIN(v, MAX-VALUE(RESULT(s, a),α, β))
if $v \leq \alpha$ **then return** v
β ←MIN(β, v)
return v

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  v ←MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

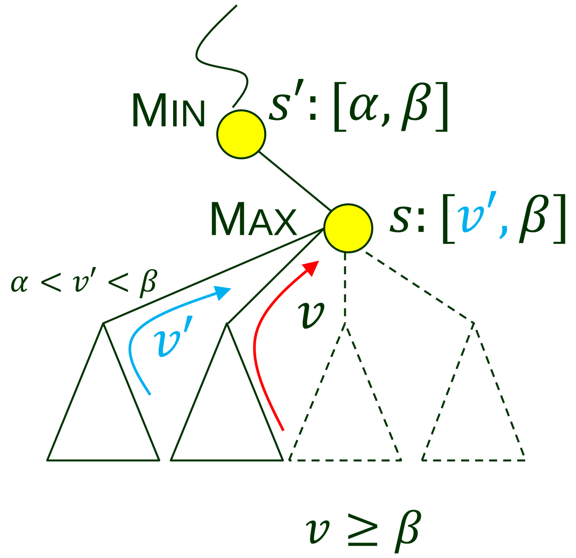


function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    v ←MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v
    α ←MAX(α, v) // new α to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    v ←MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    β ←MIN(β, v)
  return v

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
$v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
return the action in ACTIONS(*state*) with value v



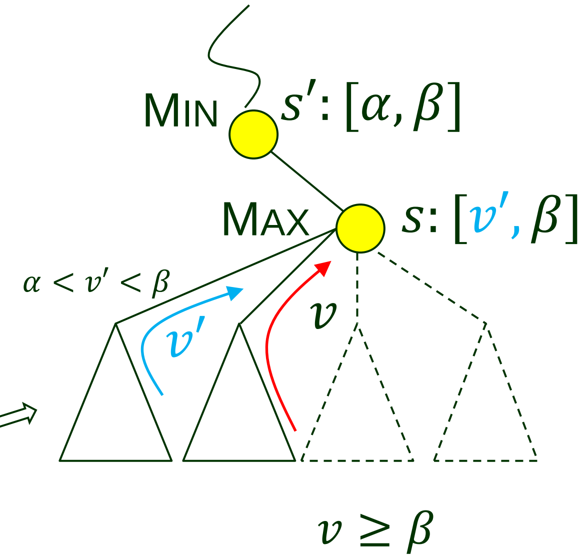
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow -\infty$
for each a in ACTIONS(*state*) **do**
$v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a),α, β))
if $v \geq \beta$ **then return** v
$\alpha \leftarrow$ MAX(α, v) // new α to be passed on to the rest MIN-VALUE
return v // calls within the for loop.

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow +\infty$
for each a in ACTIONS(*state*) **do**
$v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a),α, β))
if $v \leq \alpha$ **then return** v
$\beta \leftarrow$ MIN(β, v)
return v

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

function MAX-VALUE (*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if $v \geq \beta$ **then return** v // pruning
    $\alpha \leftarrow$ MAX(α, v) // new α to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.

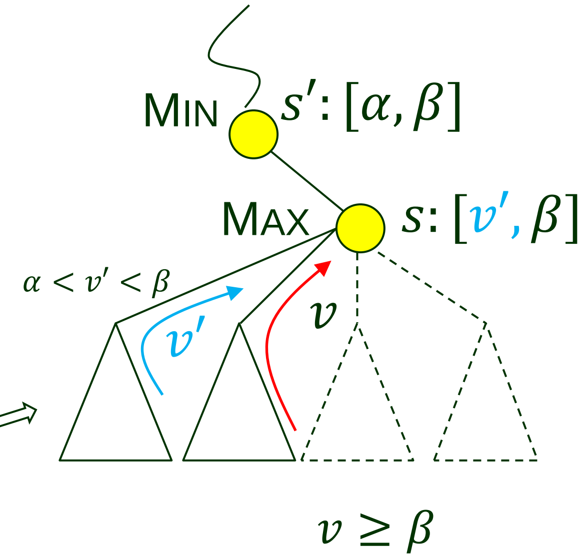


function MIN-VALUE (*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning
    $\alpha \leftarrow$ MAX(α, v) // new α to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.

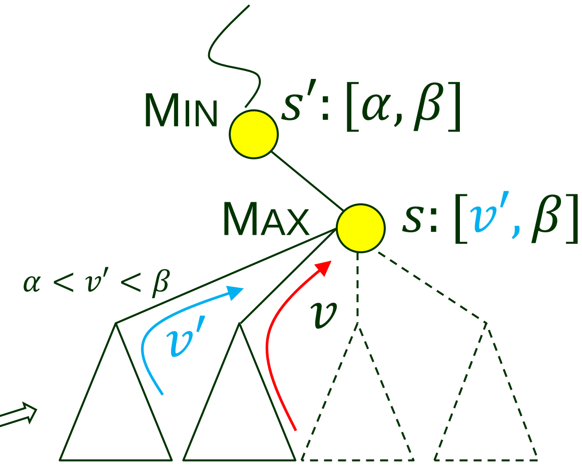


function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Search Algorithm (3rd Edition of Textbook)

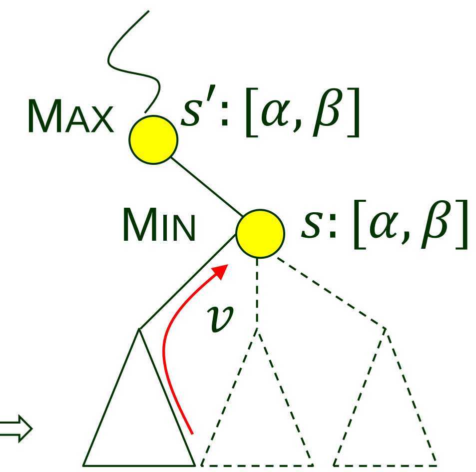
function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if $v \geq \beta$ **then return** v // pruning
    $\alpha \leftarrow$ MAX(α, v) // new α to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.



$v \geq \beta$

function MIN-VALUE (*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if $v \leq \alpha$ **then return** v // pruning
    $\beta \leftarrow$ MIN(β, v)
  return v

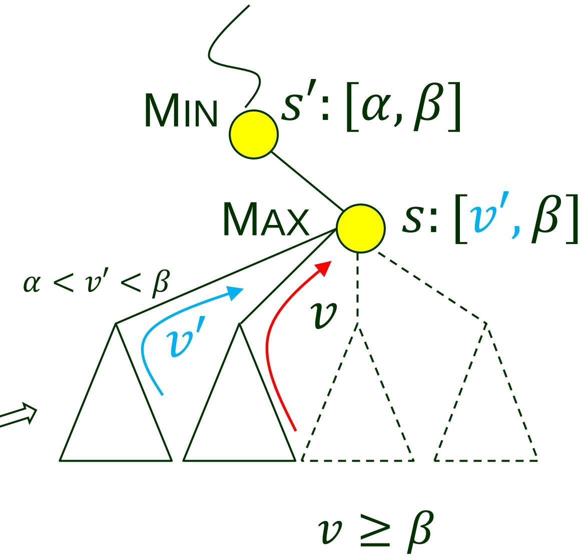


$v \leq \alpha$

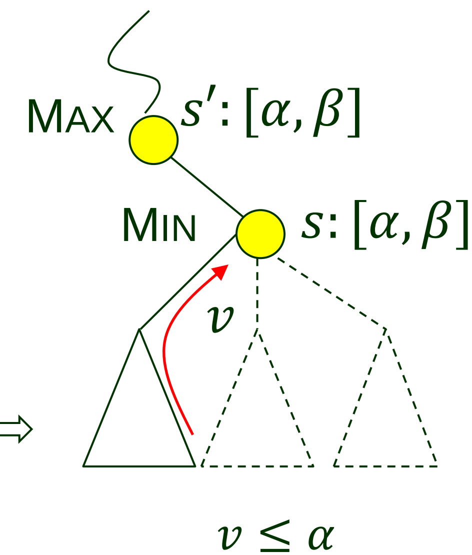
Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function Alpha-Beta-Search(*state*) **returns** an action
  $v \leftarrow$ Max-Value(*state*, $-\infty, +\infty$)
  return the action in Actions(*state*) with value v

// no change of β value within Max-Value()
function Max-Value (*state*,α, β) **returns** *a utility value*
  if Terminal-Test(*state*) **then return** Utility(*state*)
  $v \leftarrow -\infty$
  for each a in Actions(*state*) **do**
    $v \leftarrow$ Max(v, Min-Value(Result(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning
    $\alpha \leftarrow$ Max(α, v) // new α to be passed on to the rest Min-Value
  return v // calls within the for loop.



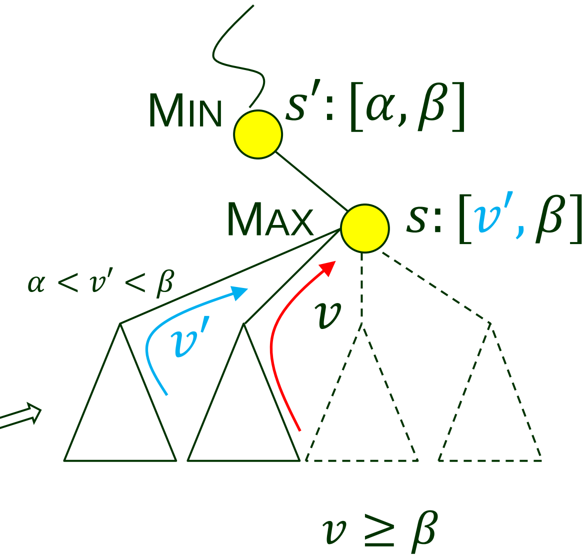
function Min-Value (*state*,α, β) **returns** *a utility value*
  if Terminal-Test(*state*) **then return** Utility(*state*)
  $v \leftarrow +\infty$
  for each a in Actions(*state*) **do**
    $v \leftarrow$ Min(v, Max-Value(Result(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning
    $\beta \leftarrow$ Min(β, v) // new β to be passed on to the rest Min-Value
  return v // calls within the for loop.



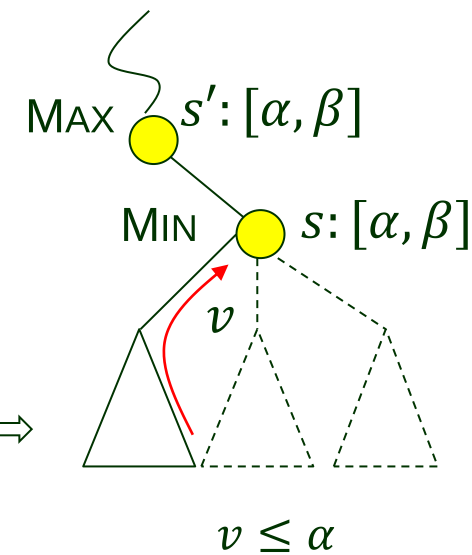
Alpha-Beta Search Algorithm (3[rd] Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

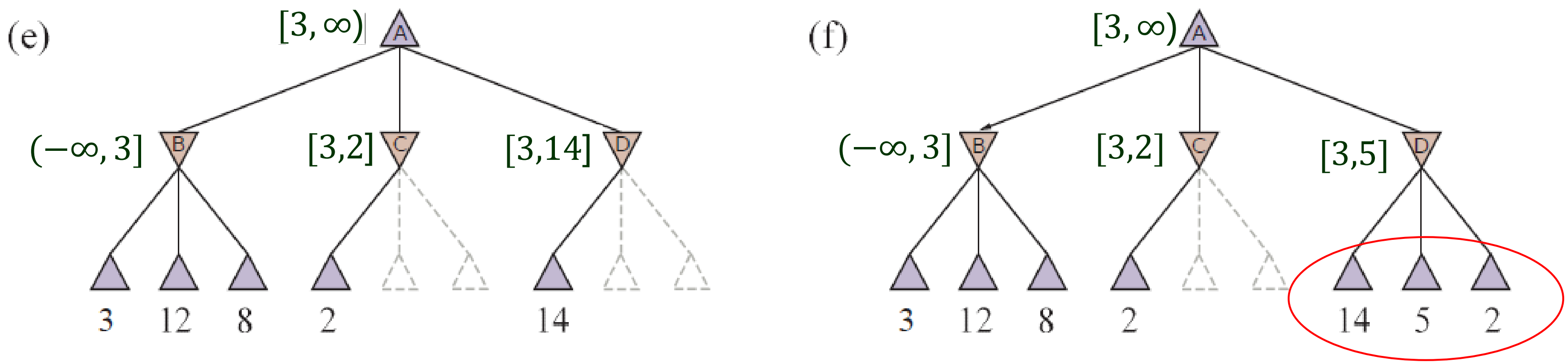
// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning
    $\alpha \leftarrow$ MAX(α, v) // new α to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.



// no change of α value within MIN-VALUE()
function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning
    $\beta \leftarrow$ MIN(β, v) // new β to be passed on to the rest MIN-VALUE
  return v // calls within the for loop.

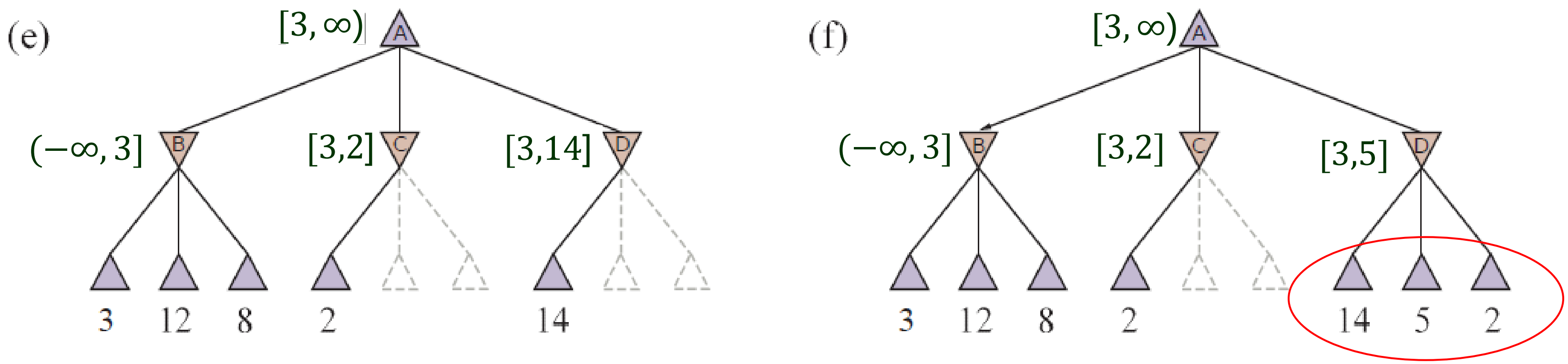


Move Ordering



Successors 14 and 5 would've been pruned had 2 been generated first.

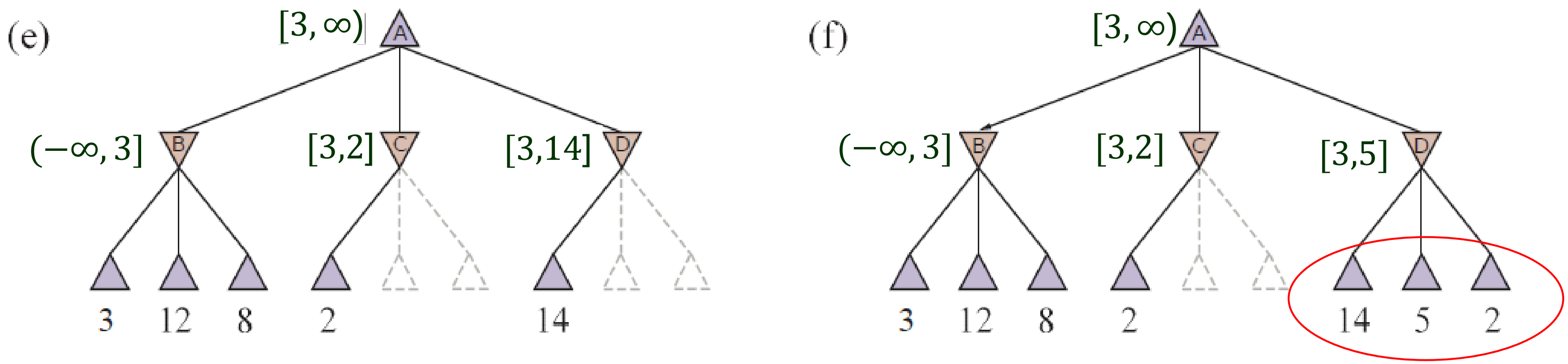
Move Ordering



Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.

Move Ordering

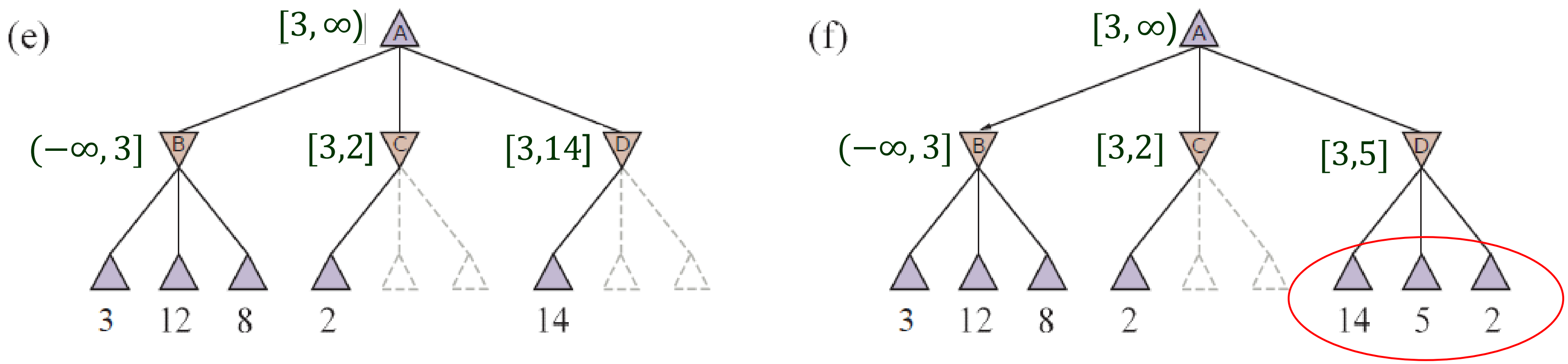


Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.

- "Perfect ordering" has effective branching factor $\sqrt{b}$, which limits examination to only $O(b^{m/2})$ nodes compared to $O(b^m)$ for minimax.

Move Ordering



Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.

- "Perfect ordering" has effective branching factor $\sqrt{b}$, which limits examination to only $O(b^{m/2})$ nodes compared to $O(b^m)$ for minimax.

- $O(b^{3m/4})$ nodes for random move ordering.

Two Strategies

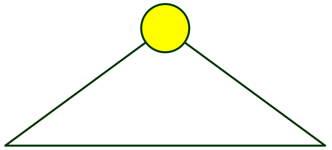
- For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

Two Strategies

- For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

Claude Shannon (1950) suggested two strategies:

- Type A (heuristic alpha-beta tree search) – chess
  - Considers all possible moves to a certain depth.
  - Use a heuristic function to estimate utilities of states at that depth.



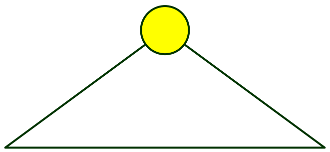
Explores wide & shallow portion of the search tree.

Two Strategies

- ♦ For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- ♠ Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

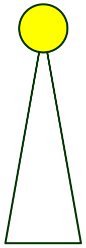
Claude Shannon (1950) suggested two strategies:

- Type A (heuristic alpha-beta tree search) – chess
  - ♣ Considers all possible moves to a certain depth.
  - ♣ Use a heuristic function to estimate utilities of states at that depth.



Explores wide & shallow portion of the search tree.

- Type B (Monte Carlo tree search) – Go
  - ♣ Ignore moves that look bad.
  - ♣ Follow promising lines "as far as possible".



Explores deep but narrow portion of the search tree.

II. Heuristic Alpha-Beta Tree Search

- Cut off the search early by applying a heuristic evaluation function.
- Replace UTILITY with EVAL, which estimates a state's utility.

II. Heuristic Alpha-Beta Tree Search

- Cut off the search early by applying a heuristic evaluation function.
- Replace UTILITY with EVAL, which estimates a state's utility.

$$\text{H-MINIMAX}(s,d) =$$

$$\begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Evaluation Functions

$\text{EVAL}(s, p)$ returns an estimate of the expected utility s to player p.

- $\text{EVAL}(s, p) = \text{UTILITY}(s, p)$ if s is terminal;
- $\text{UTILITY}(loss, p) \leq \text{EVAL}(s, p) \leq \text{UTILITY}(win, p)$ if s is nonterminal.

Evaluation Functions

EVAL(s, p) returns an estimate of the expected utility s to player p.

- EVAL(s, p) = UTILITY(s, p) if s is terminal;
- UTILITY$(loss, p) \leq$ EVAL$(s, p) \leq$ UTILITY(win, p) if s is nonterminal.

Criteria:

- No excessive computation time.
- Strong correlation with actual chances of winning.

Eval Function: State Categorization

- Calculate various features of the state (e.g., #pawns, #queens in chess).

Eval Function: State Categorization

- Calculate various features of the state (e.g., #pawns, #queens in chess).
- Define categories (equivalent classes) of states (e.g., all two-pawn vs one-pawn endgames).
  - Each category may contain states leading to wins, draws, and losses.
  - Nevertheless, all such states have the same feature values.

Eval Function: State Categorization

- Calculate various features of the state (e.g., #pawns, #queens in chess).
- Define categories (equivalent classes) of states (e.g., all two-pawn vs one-pawn endgames).
  - Each category may contain states leading to wins, draws, and losses.
  - Nevertheless, all such states have the same feature values.
- Determine an expected value for each category.

Eval Function: State Categorization

- Calculate various features of the state (e.g., #pawns, #queens in chess).
- Define categories (equivalent classes) of states (e.g., all two-pawn vs one-pawn endgames).
  - Each category may contain states leading to wins, draws, and losses.
  - Nevertheless, all such states have the same feature values.
- Determine an expected value for each category.

e.g., two-pawns vs. one-pawn

$1 \leftarrow$	wins	82%
$0 \leftarrow$	losses	2%
$0.5 \leftarrow$	draws	16%

$$(0.82 \times 1) + (0.02 \times 0) + (0.16 \times 0.5) = 0.9$$

Eval Function: State Categorization

- Calculate various features of the state (e.g., #pawns, #queens in chess).
- Define categories (equivalent classes) of states (e.g., all two-pawn vs one-pawn endgames).
  - Each category may contain states leading to wins, draws, and losses.
  - Nevertheless, all such states have the same feature values.
- Determine an expected value for each category.

e.g., two-pawns vs. one-pawn

$1 \hookleftarrow$	wins	82%
$0 \hookleftarrow$	losses	2%
$0.5 \hookleftarrow$	draws	16%

$$(0.82 \times 1) + (0.02 \times 0) + (0.16 \times 0.5) = 0.9$$

- Too many categories and too much dependence on experience.

Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\textsc{Eval}(s) = \underset{\substack{\uparrow \\ \text{weight}}}{w_1} \underset{\substack{\uparrow \\ \text{e.g., \#pawns in chess}}}{f_1}(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

weight e.g., #pawns in chess

- Strongly correlated with the chance of winning.

Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

weight e.g., #pawns in chess

- Strongly correlated with the chance of winning.
- But not necessarily linearly correlated.

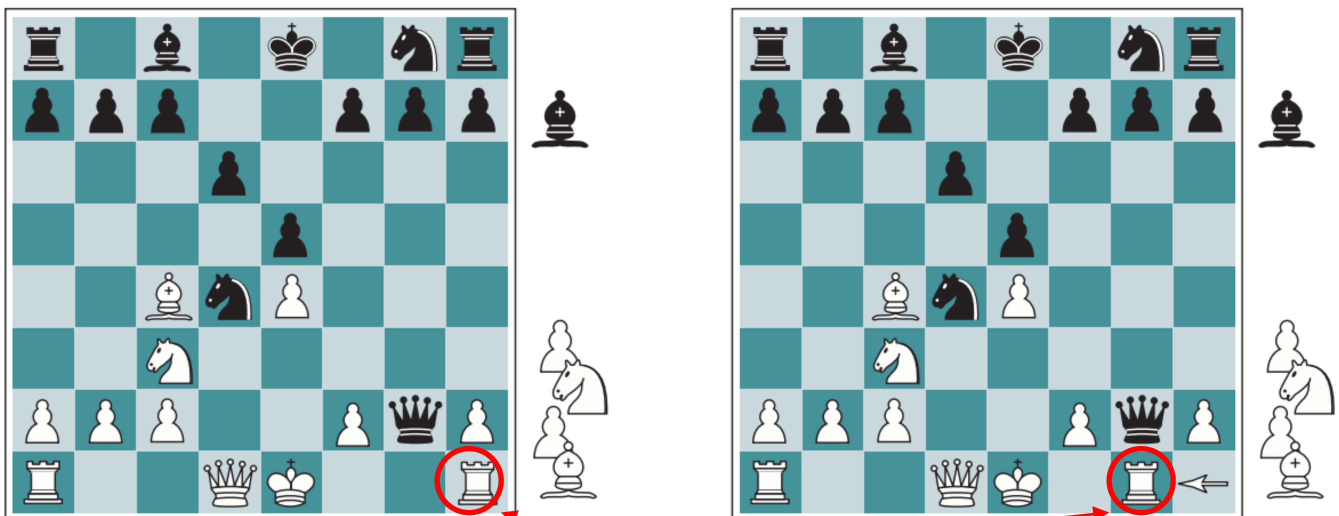
Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

weight   e.g., #pawns in chess

- Strongly correlated with the chance of winning.
- But not necessarily linearly correlated.



(a) White to move   Only difference   (b) White to move

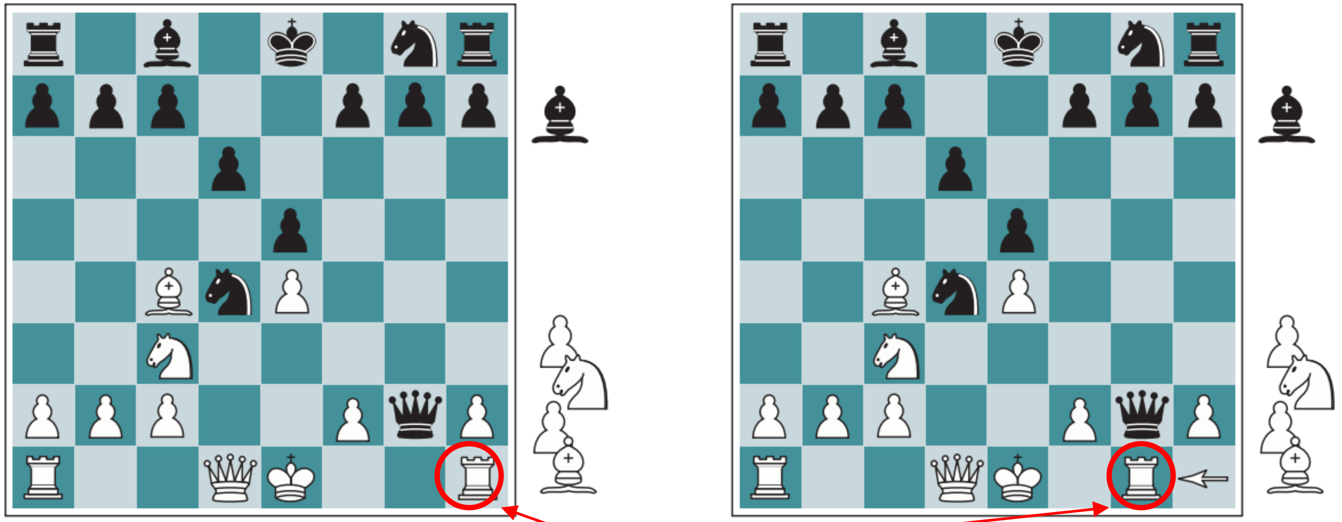
Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

weight e.g., #pawns in chess

- Strongly correlated with the chance of winning.
- But not necessarily linearly correlated.



(a) White to move Only difference (b) White to move

Black should win because of an advantage (1 knight & 2 pawns)

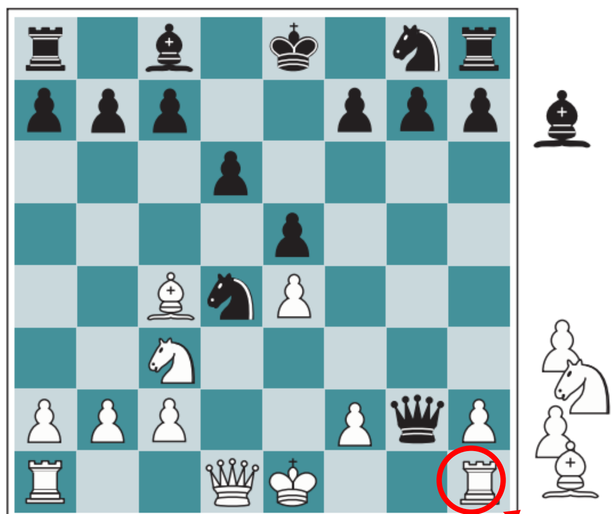
Eval Function: Feature Combination

Compute separate numerical contributions from each feature and combine them.

$$\boxed{\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

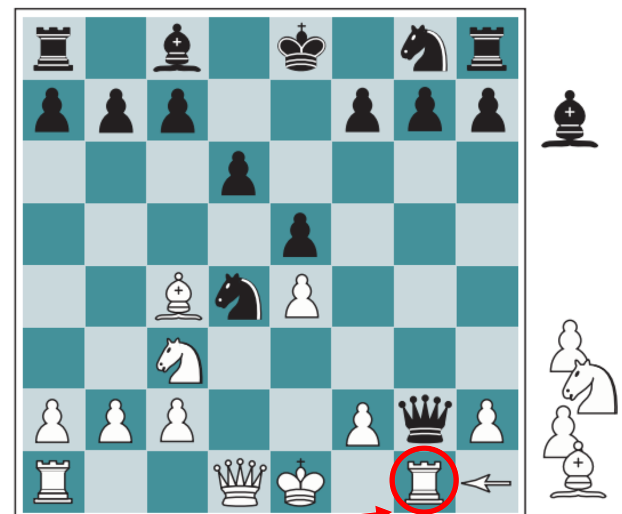
weight e.g., #pawns in chess

- Strongly correlated with the chance of winning.
- But not necessarily linearly correlated.



(a) White to move

Only difference



(b) White to move

Black should win because of an advantage (1 knight & 2 pawns)

White should win because its rook will capture the queen.

Eval Function (cont'd)

$$\boxed{\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)}$$

♠ Assumes independent feature contributions.

♦ Use a nonlinear feature combination.

e.g., two bishops might be worth more than twice the value of a single bishop.

Cutting off Search

if $game.\cancel{\text{IS-TERMINAL}}(state)$ **then return** $game.\cancel{\text{UTILITY}}(state, player), null$

IS-CUTOFF EVAL

Cutting off Search

if $game.\cancel{\text{IS-TERMINAL}}(state)$ **then return** $game.\cancel{\text{UTILITY}}(state, player), null$

IS-CUTOFF EVAL

Some strategies:

- Set a fixed depth limit d to control the amount of search.

  IS-CUTOFF returns true if depth $> d$.

Cutting off Search

if $game.\cancel{\text{IS-TERMINAL}}(state)$ **then return** $game.\cancel{\text{UTILITY}}(state, player), null$

IS-CUTOFF EVAL

Some strategies:

- Set a fixed depth limit d to control the amount of search.

  IS-CUTOFF returns true if depth $> d$.

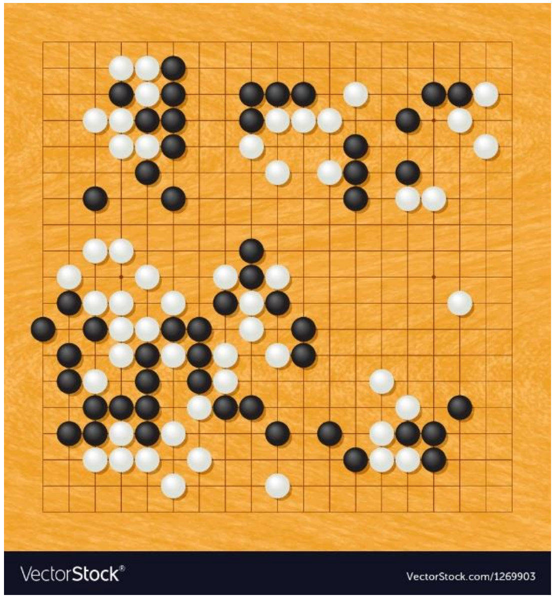
- Apply iterative deepening:

  When time runs out, returns the move selected by the deepest completed search.

Real-Time Decisions

- Minimax with alpha-beta pruning.
- Extensively tuned evaluation function.
- Pruning heuristics.
- A transposition table of repeated states and evaluations.
  - To avoid re-searching the game tree below that state.
- A large database of optimal opening and endgame moves.
  - Table lookup instead of search.
  - Chess endgames with up to 7 pieces solved.
- Minimax unsuccessful in Go.

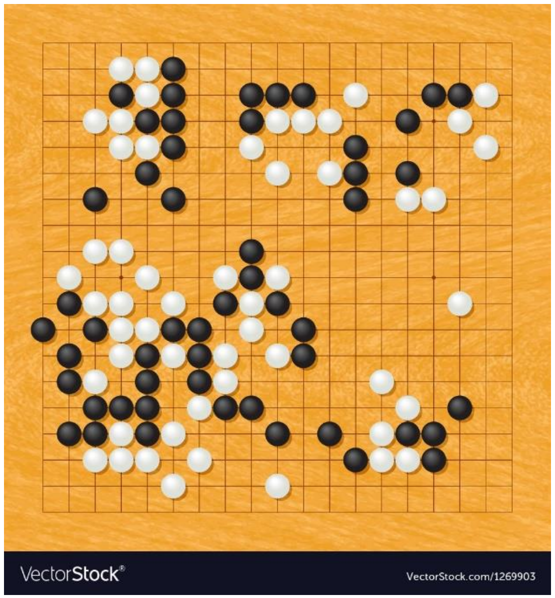
III. Monte Carlo Tree Search



Two weaknesses of heuristic alpha-beta search on Go:

♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

III. Monte Carlo Tree Search

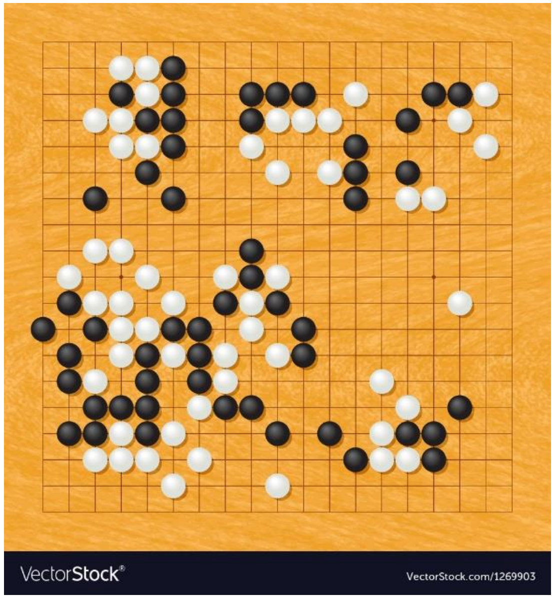


Two weaknesses of heuristic alpha-beta search on Go:

- ♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

- ♠ It is difficult to define a good evaluation function.

III. Monte Carlo Tree Search



Two weaknesses of heuristic alpha-beta search on Go:

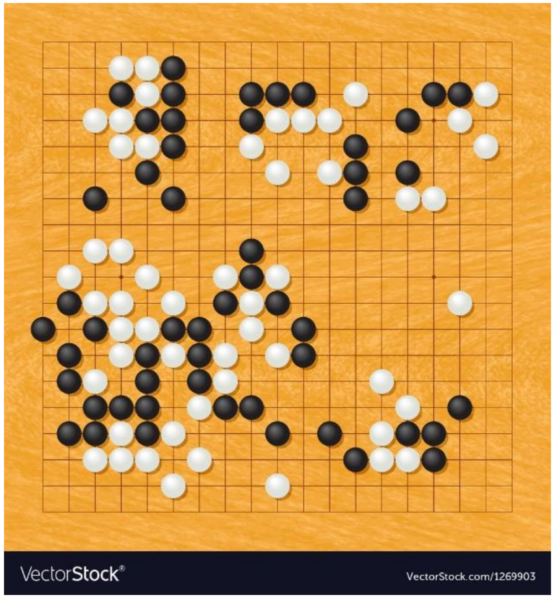
♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

♠ It is difficult to define a good evaluation function.

- #pieces is not a strong indicator.

Score = #vacant intersection points inside own territory
+ #stones captured from the opponent

III. Monte Carlo Tree Search



Two weaknesses of heuristic alpha-beta search on Go:

♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

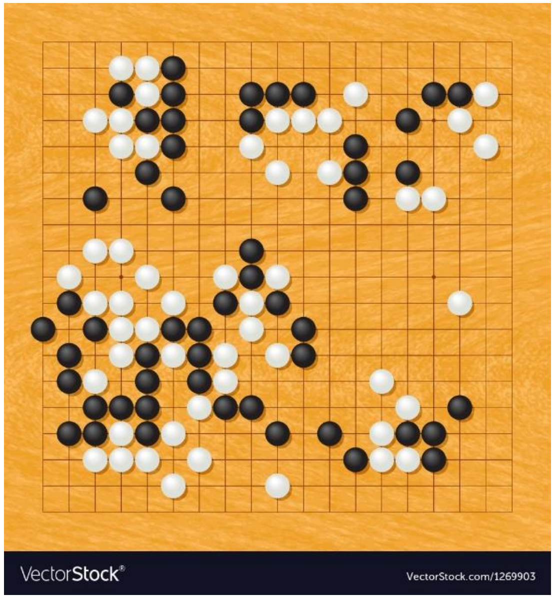
♠ It is difficult to define a good evaluation function.

- #pieces is not a strong indicator.

Score = #vacant intersection points inside own territory
+ #stones captured from the opponent

- Most positions are changing until the endgame.

III. Monte Carlo Tree Search



Two weaknesses of heuristic alpha-beta search on Go:

♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

♠ It is difficult to define a good evaluation function.

- #pieces is not a strong indicator.

Score = #vacant intersection points inside own territory
+ #stones captured from the opponent

- Most positions are changing until the endgame.

Modern Go programs use Monte Carlo tree search (MCTS) instead of alpha-beta search.

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

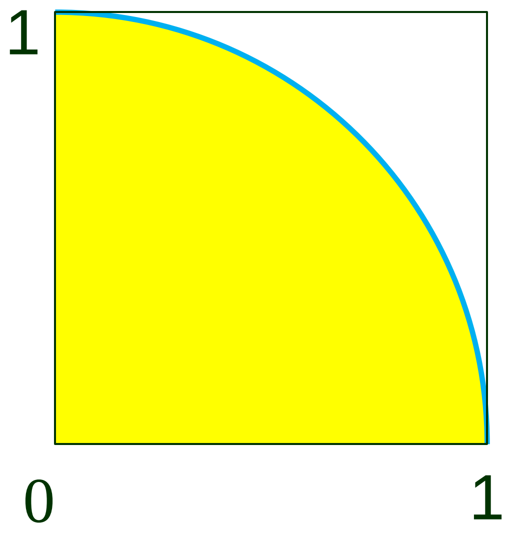
Monte Carlo Method

Idea: Use random sampling to evaluate a function.

How to estimate π?

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

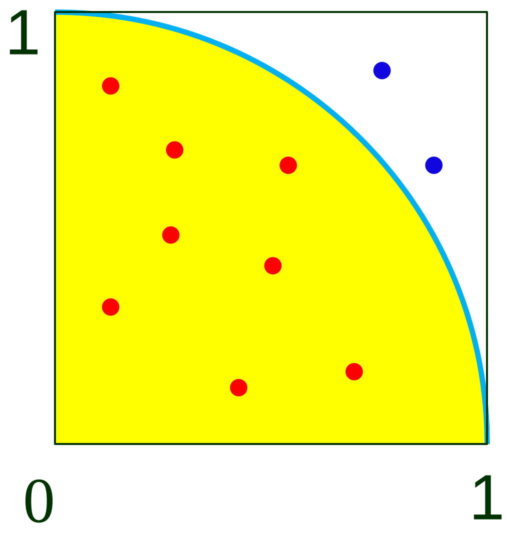


How to estimate π?

- Inscribe a quadrant within a unit square.

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

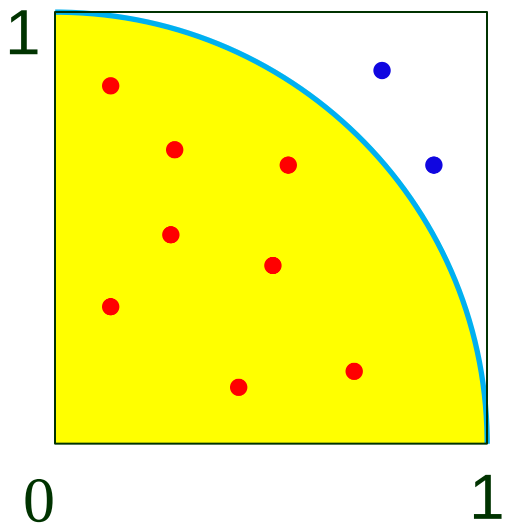


How to estimate π?

- Inscribe a quadrant within a unit square.
- Generate n random points (x, y) inside the square, with x, y uniformly distributed in $[0, 1]$.

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

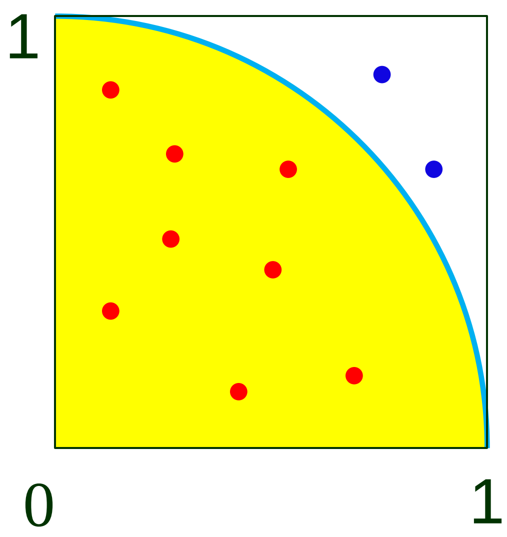


How to estimate π?

- Inscribe a quadrant within a unit square.
- Generate n random points (x, y) inside the square, with x, y uniformly distributed in $[0, 1]$.
- Let m be the number of points inside the quadrant, $x^2 + y^2 \leq 1$.

Monte Carlo Method

Idea: Use random sampling to evaluate a function.



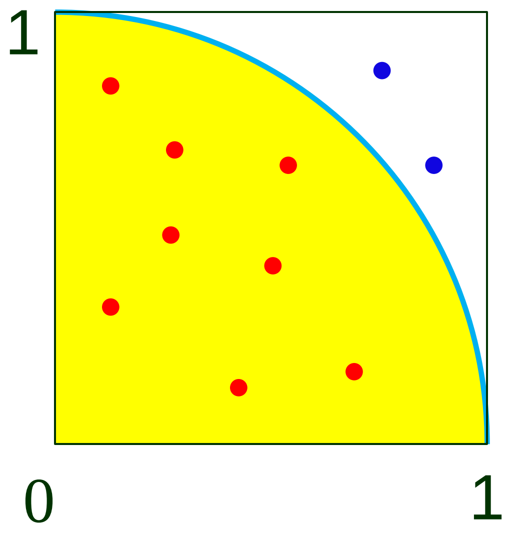
How to estimate π?

- Inscribe a quadrant within a unit square.
- Generate n random points (x, y) inside the square, with x, y uniformly distributed in $[0, 1]$.
- Let m be the number of points inside the quadrant, $x^2 + y^2 \leq 1$.
- We have

$$\frac{m}{n} \approx \frac{quadrant\ area}{square\ area} = \frac{\pi/4}{1}$$

Monte Carlo Method

Idea: Use random sampling to evaluate a function.



How to estimate π?

- Inscribe a quadrant within a unit square.
- Generate n random points (x, y) inside the square, with x, y uniformly distributed in $[0, 1]$.
- Let m be the number of points inside the quadrant, $x^2 + y^2 \leq 1$.
- We have

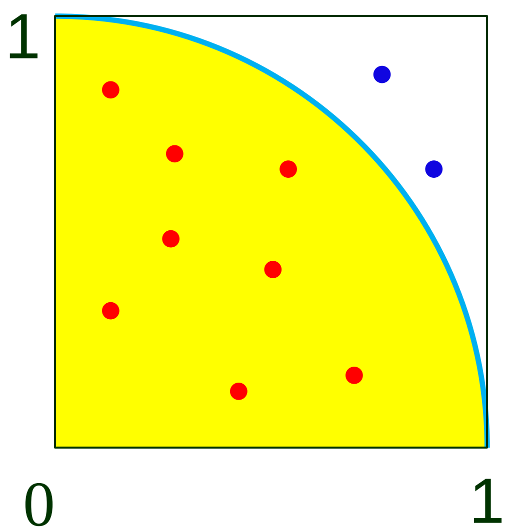
$$\frac{m}{n} \approx \frac{quadrant\ area}{square\ area} = \frac{\pi/4}{1}$$

$$\Downarrow$$

$$\pi \approx \frac{4m}{n}$$

Monte Carlo Method

Idea: Use random sampling to evaluate a function.



How to estimate π?

- Inscribe a quadrant within a unit square.
- Generate n random points (x, y) inside the square, with x, y uniformly distributed in $[0, 1]$.
- Let m be the number of points inside the quadrant, $x^2 + y^2 \leq 1$.
- We have

$$\frac{m}{n} \approx \frac{quadrant\ area}{square\ area} = \frac{\pi/4}{1}$$

$$\Downarrow$$

$$\pi \approx \frac{4m}{n}$$

For accuracy, many points should be generated.

Borrowing the Idea

- No use of a heuristic evaluation function.
- The value of a state estimated as the average utility over a number of simulations of complete games.

Borrowing the Idea

- No use of a heuristic evaluation function.
- The value of a state estimated as the average utility over a number of simulations of complete games.

winning percentage

Borrowing the Idea

- No use of a heuristic evaluation function.
- The value of a state estimated as the average utility over a number of simulations of complete games.

winning percentage

A simulation (a playout or rollout) proceeds as below:

- Choose moves alternatively for the two players.
- Determine the outcome when a terminal position is reached.

Choice of a Move

What is the best move if both players play randomly?

What is the best move if both players play well?

Choice of a Move

What is the best move if both players play randomly?

?

What is the best move if both players play well?

Choice of a Move

What is the best move if both players play randomly?

⇕ ?

True for simple games
but false for most games

What is the best move if both players play well?

Choice of a Move

What is the best move if both players play randomly?

⇕ ?

True for simple games
but false for most games

What is the best move if both players play well?

- Need a *playout policy* biased toward good moves.

Choice of a Move

What is the best move if both players play randomly?

⇕ ?

True for simple games
but false for most games

What is the best move if both players play well?

- Need a *playout policy* biased toward good moves.
- These policies are often *learned from self-play* using neural networks.

Pure Monte Carlo Search

- From what positions do we start the playout?
- How many playouts are allocated to each position?

Pure Monte Carlo Search

- ♣ From what positions do we start the playout?
- ♣ How many playouts are allocated to each position?

Algorithm:

- Conduct N simulations starting from the current state s.
- Track which move from s has the highest win percentage.

Pure Monte Carlo Search

- From what positions do we start the playout?
- How many playouts are allocated to each position?

Algorithm:

- Conduct N simulations starting from the current state s.
- Track which move from s has the highest win percentage.

How to improve? Need a selection policy that balances

- exploration of states that have few playouts, and
- exploitation of states that have done well in the past.