Other Search Strategies for Games

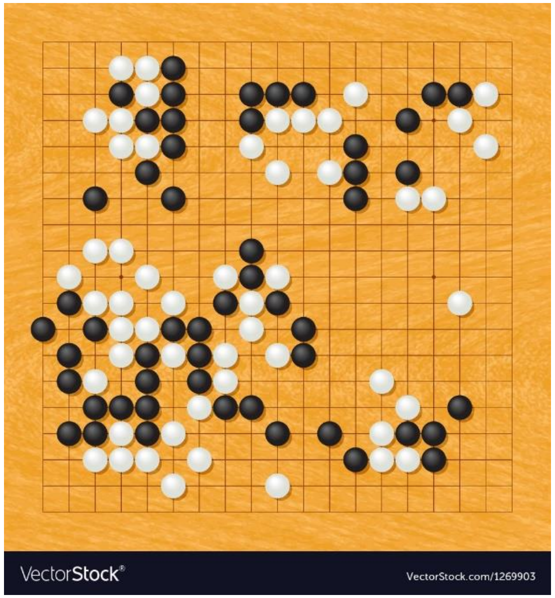
Outline

I. Monte Carlo tree search

II. Stochastic games

* 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.

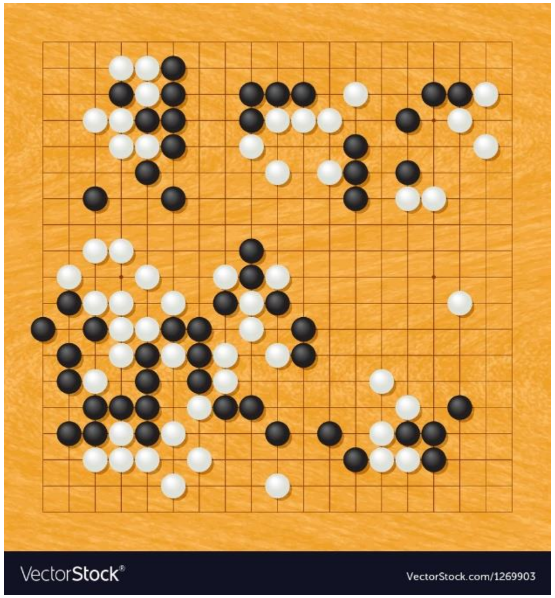
I. Monte Carlo Tree Search



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 ($361^5 \approx 6.13 \times 10^{12}$).

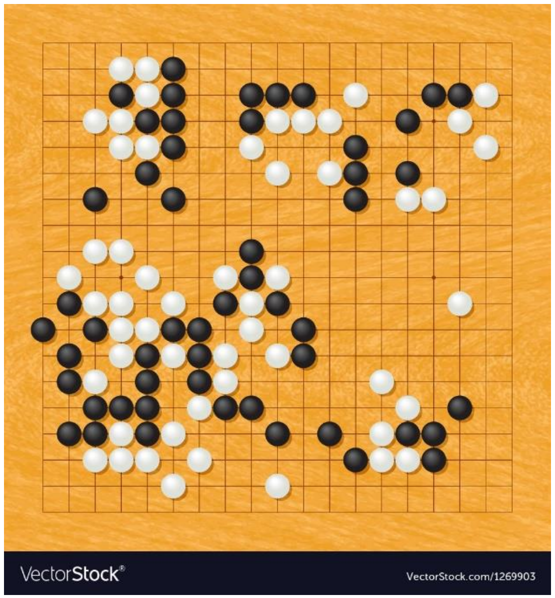
I. Monte Carlo Tree Search



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.

I. Monte Carlo Tree Search



Two weaknesses of heuristic alpha-beta search on Go:

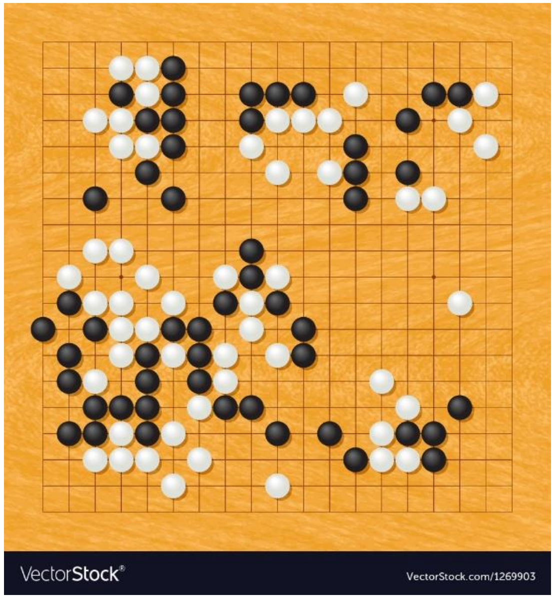
- ♠ Its high branching factor ($b = 361 = 19^2$) limits the search to only 4 or 5 ply ($361^5 \approx 6.13 \times 10^{12}$).

- ♠ 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

I. 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 ($361^5 \approx 6.13 \times 10^{12}$).

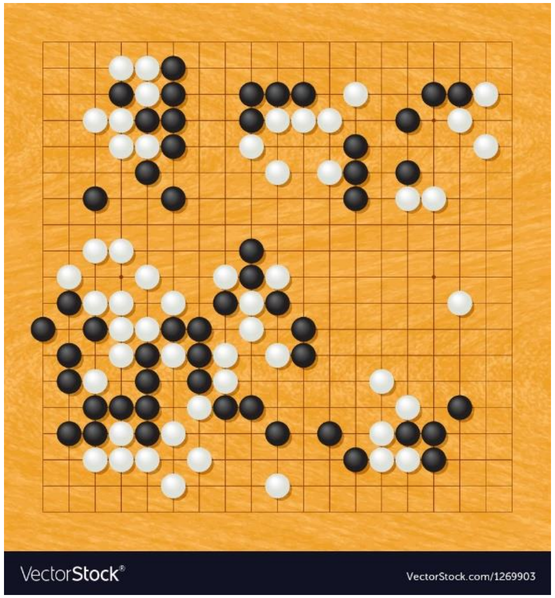
♠ It is difficult to define a good evaluation function.

- #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.

I. 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.

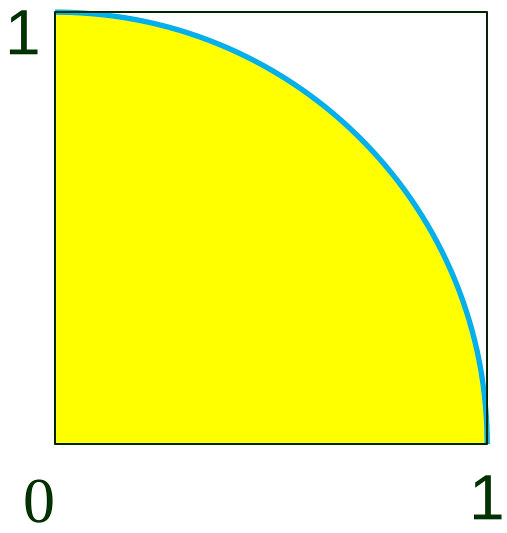
Monte Carlo Method

Idea: Use random sampling to evaluate a function.

How to estimate π?

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

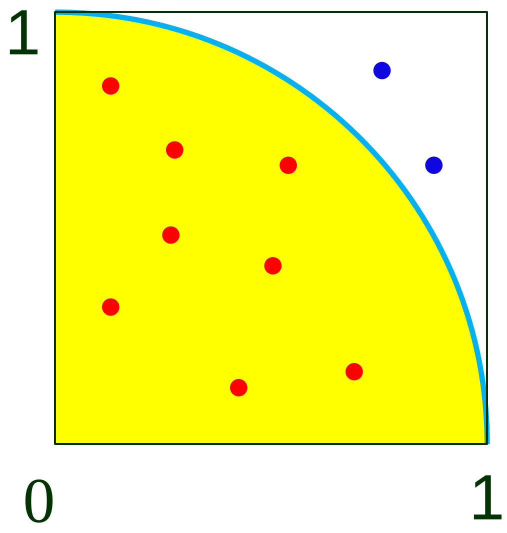


How to estimate π?

- Inscribe a quadrant within a unit square.

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

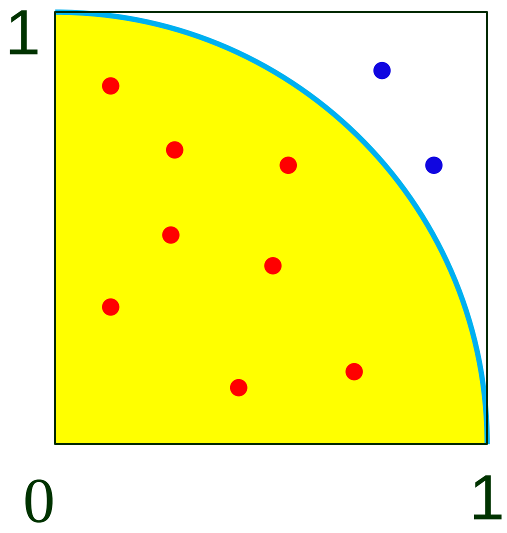


How to estimate π?

- 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.

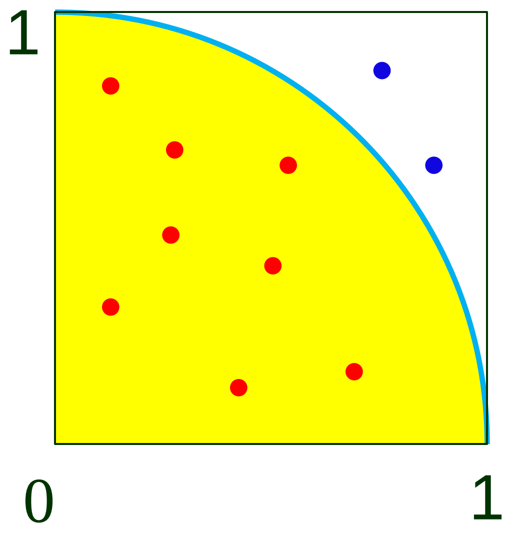


How to estimate π?

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

Monte Carlo Method

Idea: Use random sampling to evaluate a function.

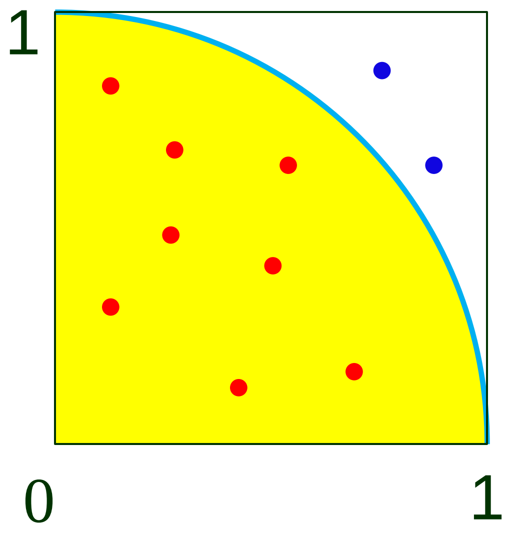


How to estimate π?

- 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.



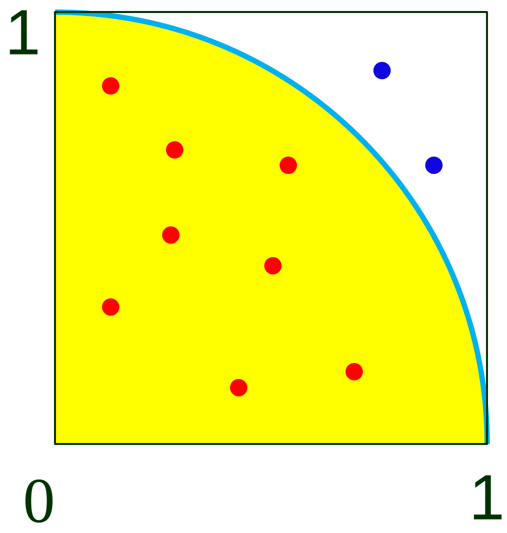
How to estimate π?

- 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

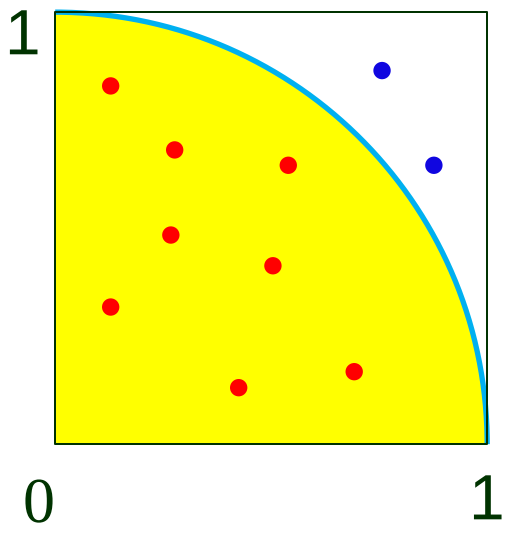
$$\frac{m}{n} \approx \frac{quadrant\ area}{square\ area} = \frac{\pi/4}{1}$$

$$\Downarrow$$

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

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?

$\Updownarrow$?

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.

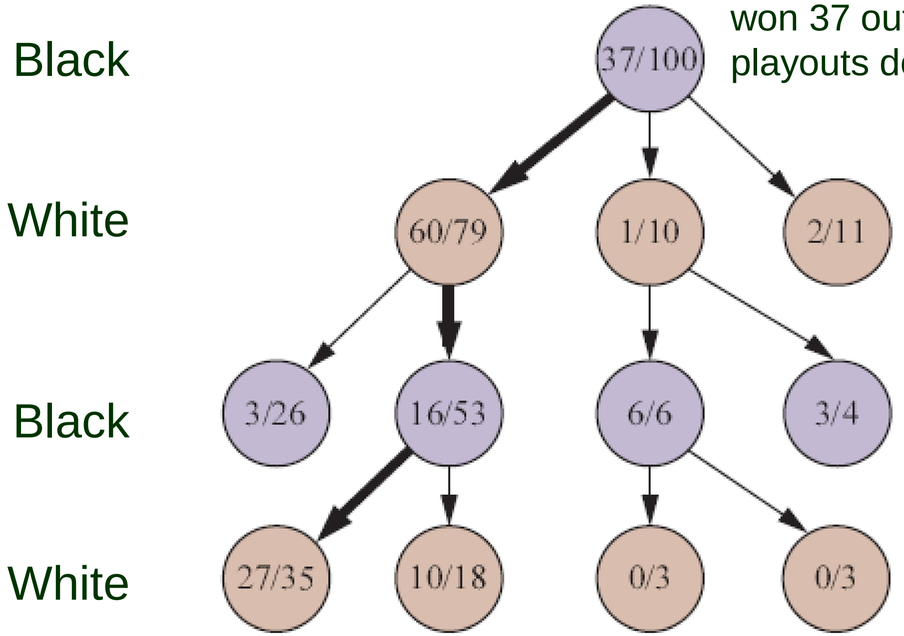
How to improve? Need a selection policy that balances

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

Step 1 of MCTS (One Iteration): Selection

Which move should Black make (at the root)?

Root: state just after the move by white, who has won 37 out of the 100 playouts done so far.



(a) Selection

Step 1 of MCTS (One Iteration): Selection

Which move should Black make (at the root)?

Root: state just after the move by white, who has won 37 out of the 100 playouts done so far.

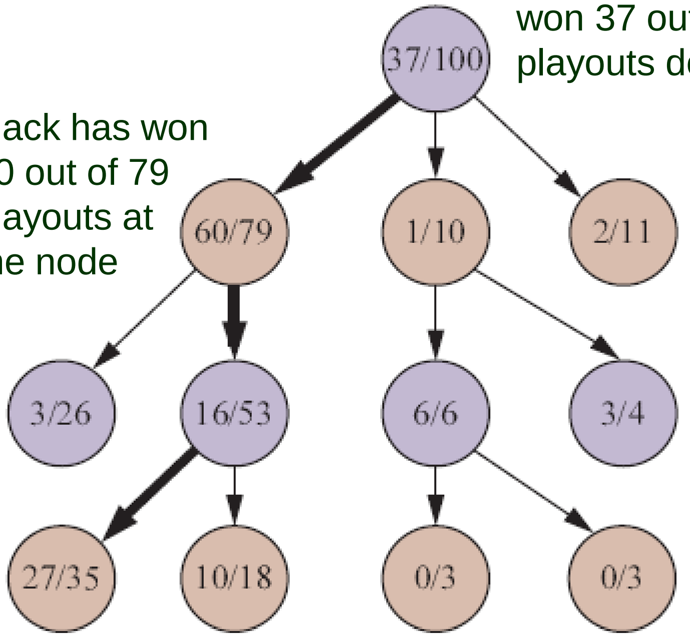
black has won 60 out of 79 playouts at the node

Black

White

Black

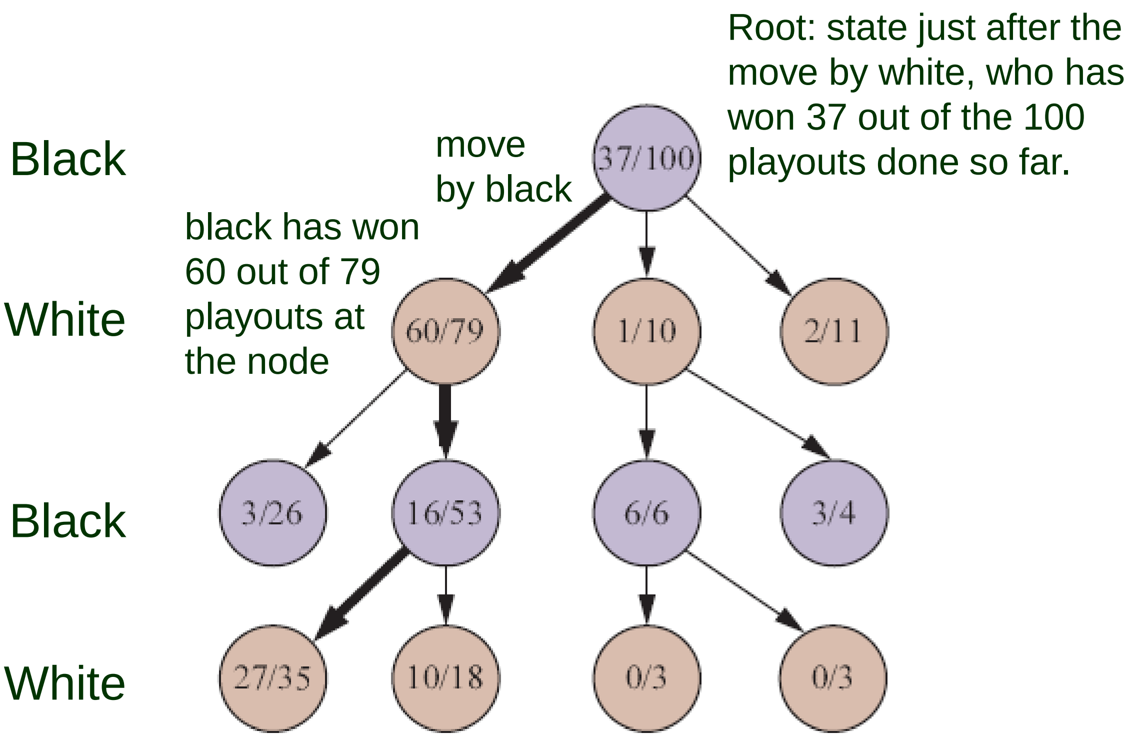
White



(a) Selection

Step 1 of MCTS (One Iteration): Selection

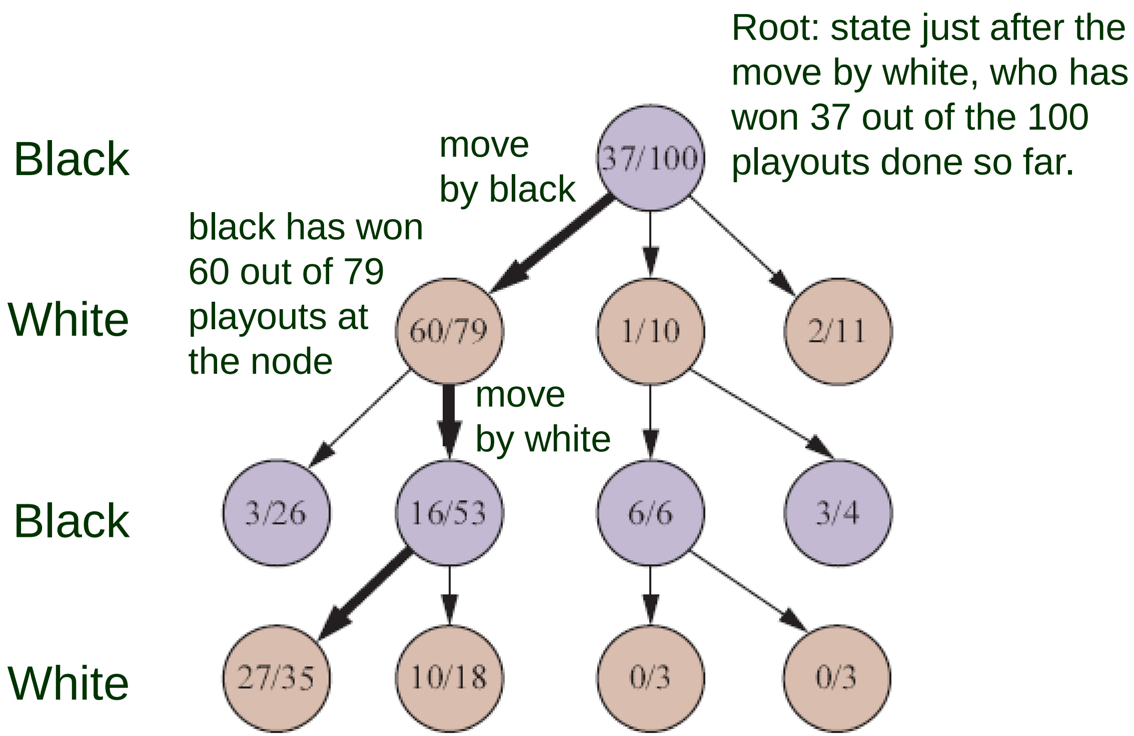
Which move should Black make (at the root)?



(a) Selection

Step 1 of MCTS (One Iteration): Selection

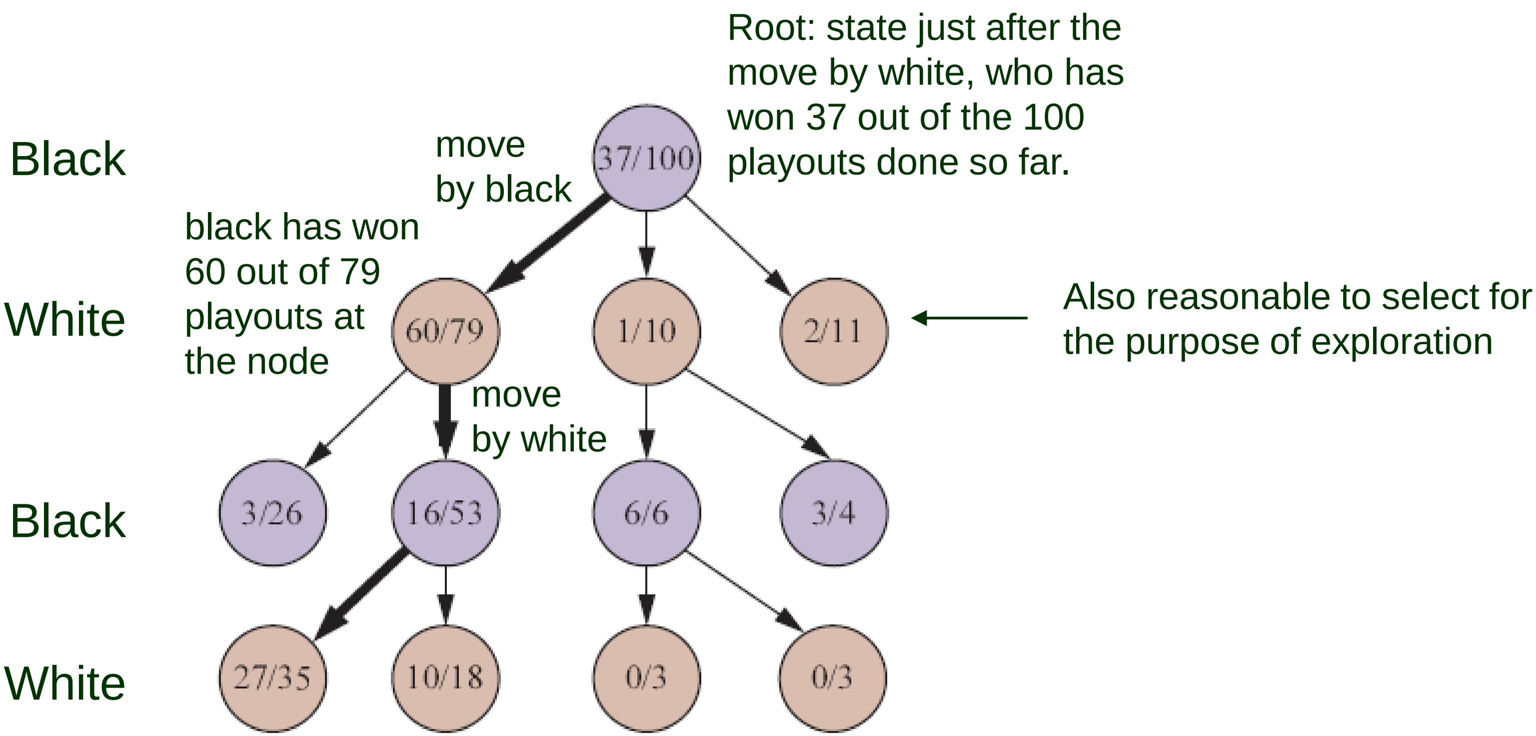
Which move should Black make (at the root)?



(a) Selection

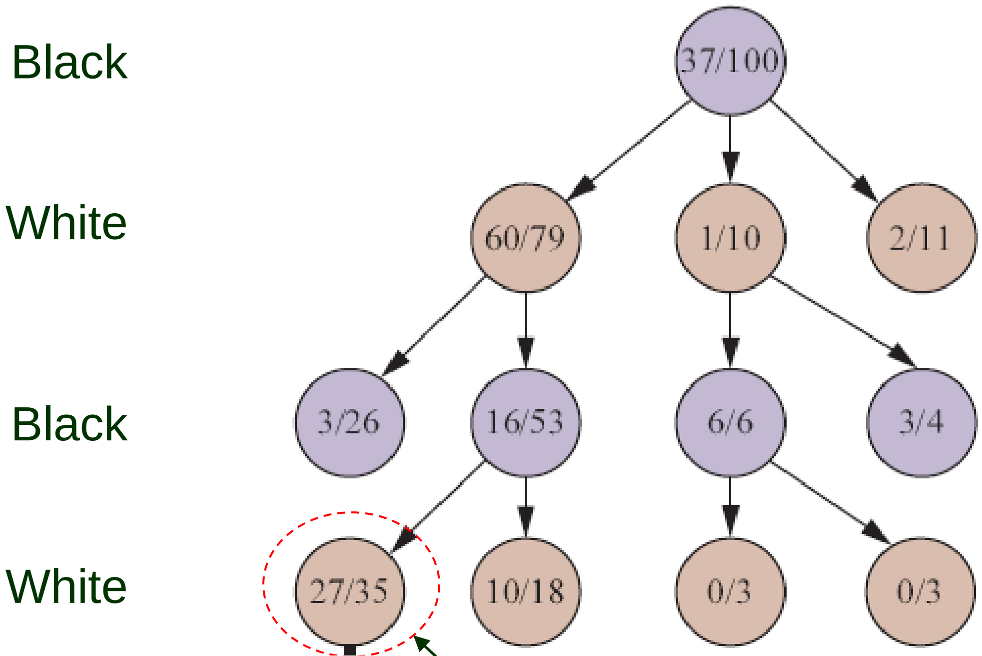
Step 1 of MCTS (One Iteration): Selection

Which move should Black make (at the root)?



(a) Selection

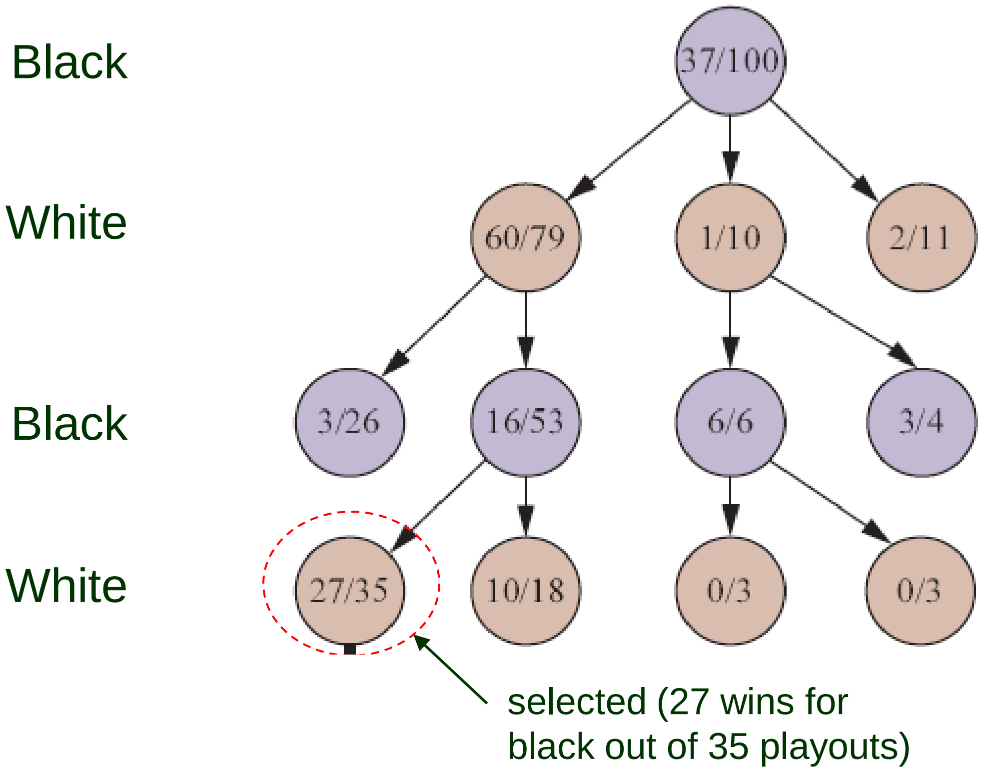
Steps 2 & 3: Expansion & Simulation



selected (27 wins for black out of 35 playouts)

(b) Expansion and simulation

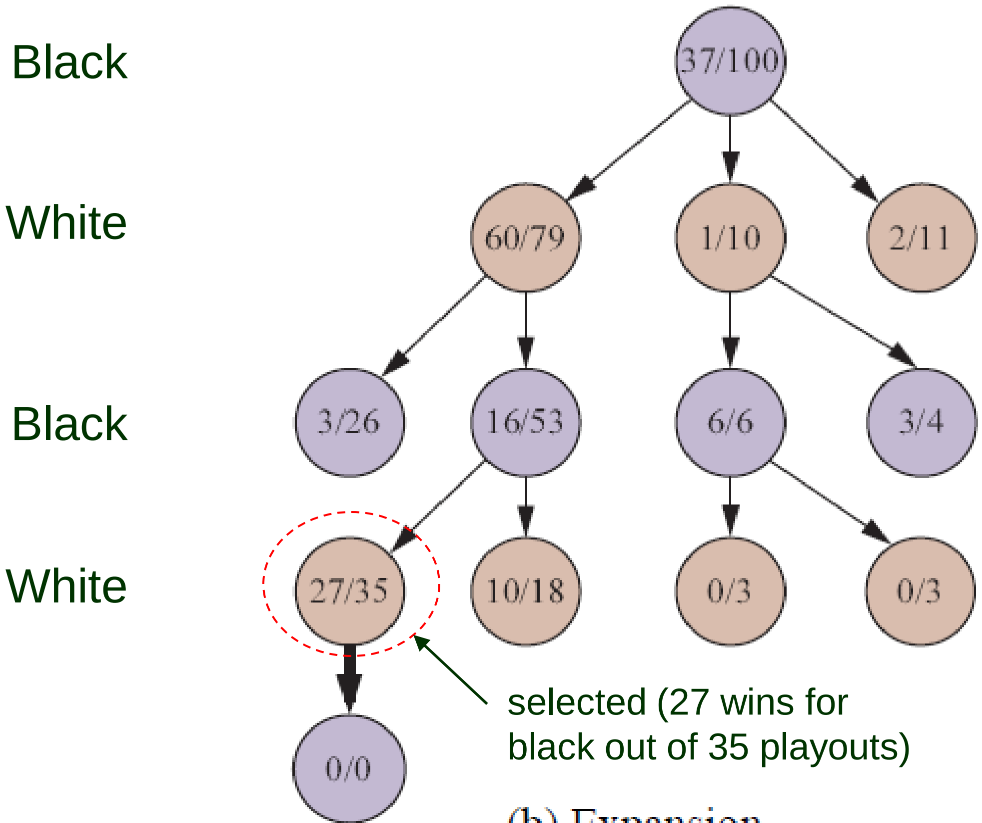
Steps 2 & 3: Expansion & Simulation



(b) Expansion and simulation

- Generate a new child of the selected node.

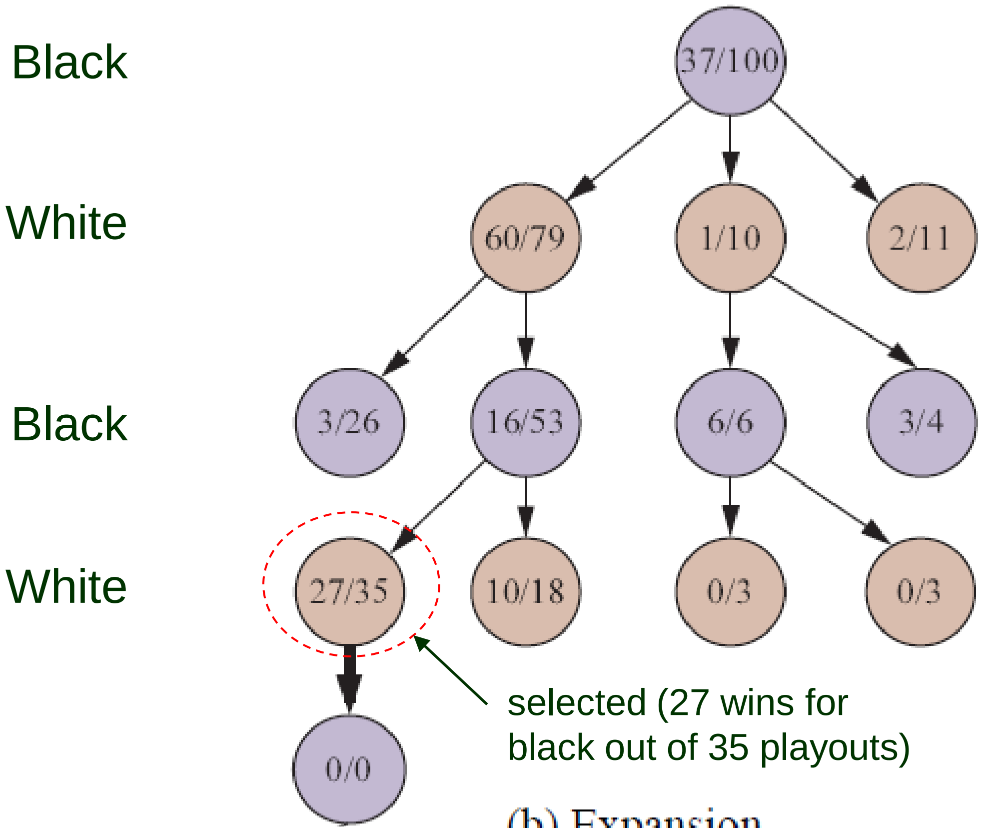
Steps 2 & 3: Expansion & Simulation



(b) Expansion and simulation

- Generate a new child of the selected node.

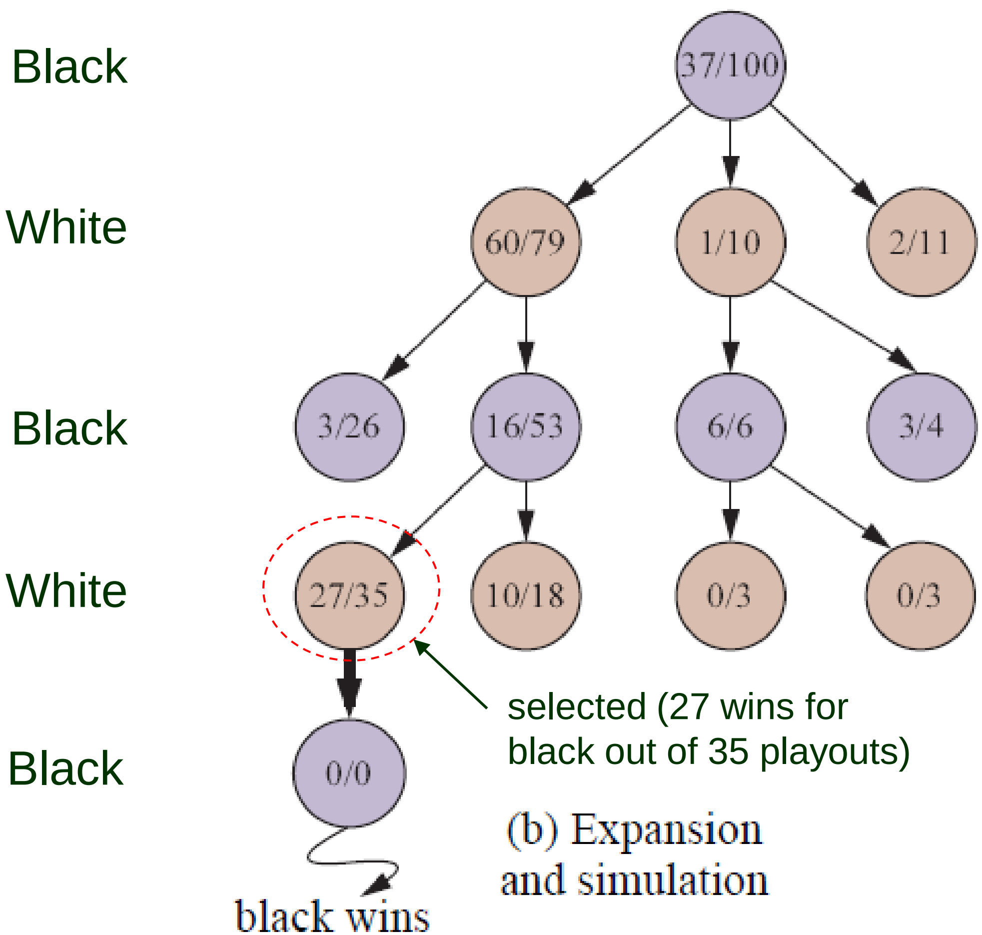
Steps 2 & 3: Expansion & Simulation



(b) Expansion and simulation

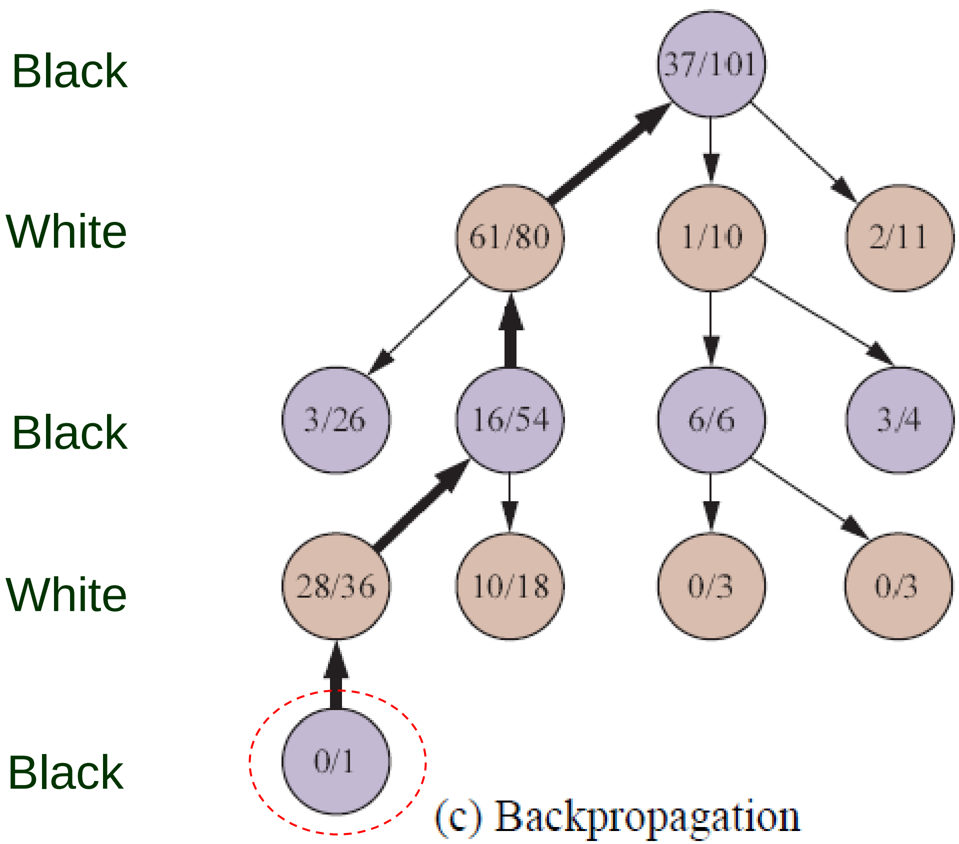
- Generate a new child of the selected node.
- Perform a playout from the newly generated child node.

Steps 2 & 3: Expansion & Simulation



- Generate a new child of the selected node.
- Perform a playout from the newly generated child node.

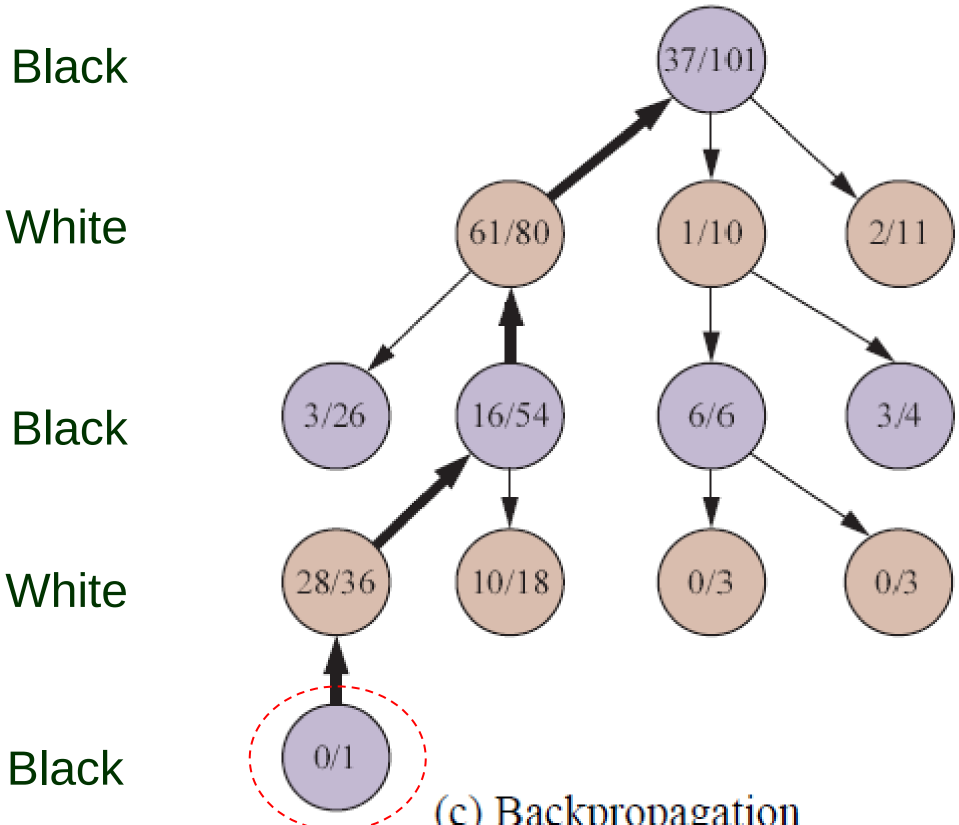
Step 4: Back Propagation



(c) Backpropagation

- Update all the nodes upward along the path until the root.

Step 4: Back Propagation

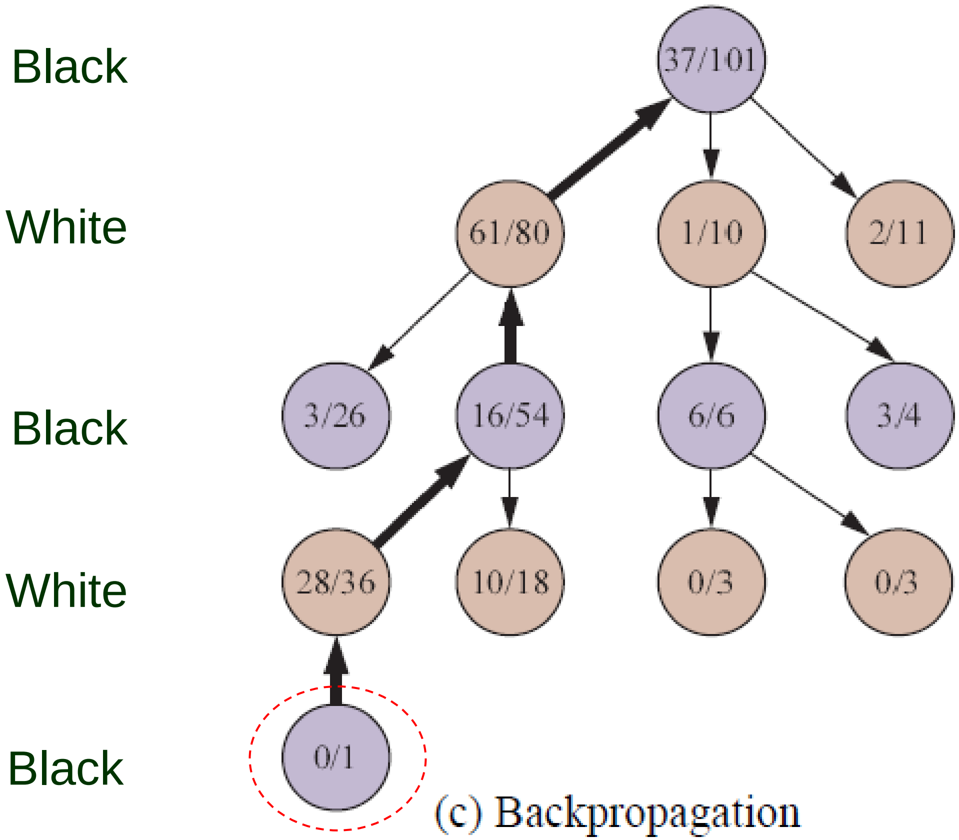


(c) Backpropagation

- Update all the nodes upward along the path until the root.

Black wins this playout:

Step 4: Back Propagation



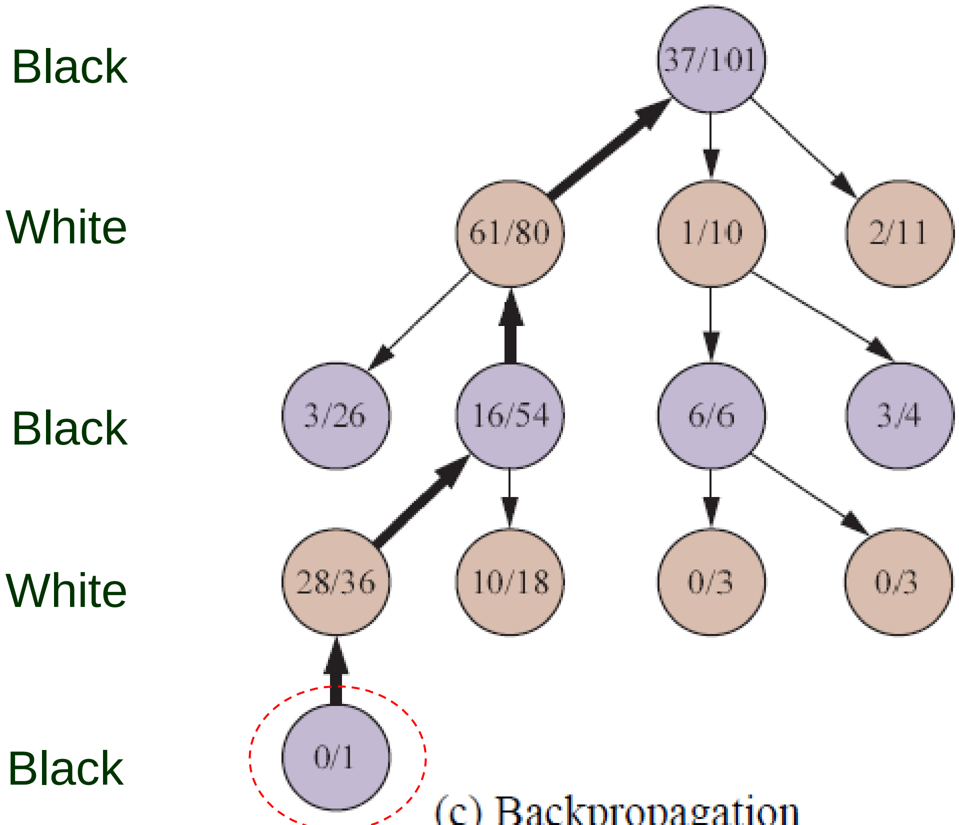
(c) Backpropagation

- Update all the nodes upward along the path until the root.

Black wins this playout:

- ♦ At a white node, increment #wins and #playouts.

Step 4: Back Propagation



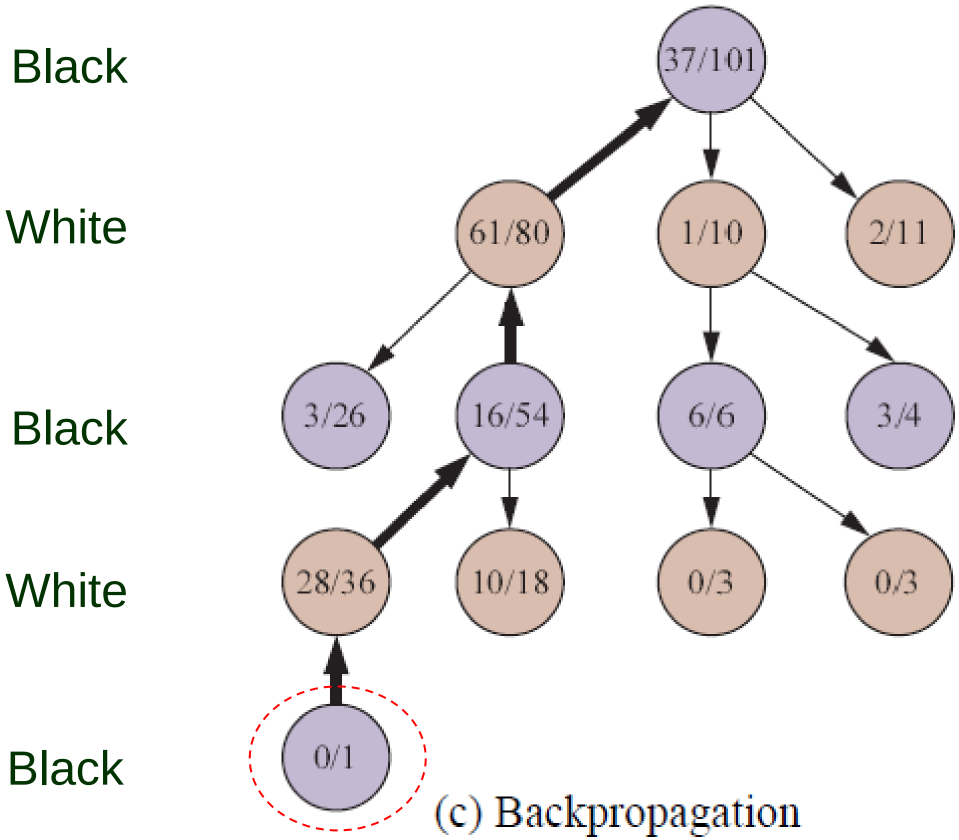
(c) Backpropagation

- Update all the nodes upward along the path until the root.

Black wins this playout:

- ♦ At a white node, increment #wins and #playouts.
- ♦ At a black node, increment #playouts only.

Step 4: Back Propagation



(c) Backpropagation

- Update all the nodes upward along the path until the root.

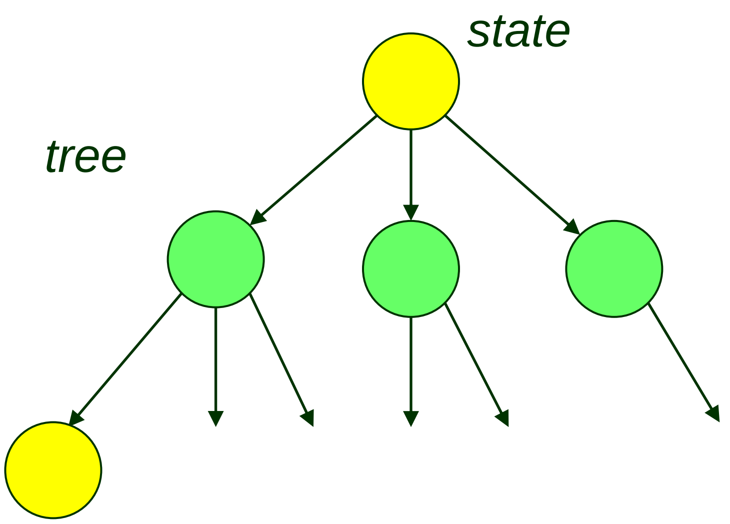
Black wins this playout:

- ♦ At a white node, increment #wins and #playouts.
- ♦ At a black node, increment #playouts only.

Monte Carlo Tree Search Algorithm

For deciding a move at the current state during the game:

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts



Ranking of Possible Moves

Upper confidence bound formula:

$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player
making a move at n
out of all playouts
through the node

$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player
making a move at n
out of all playouts
through the node

$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

#playouts through
the node

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player
making a move at n
out of all playouts
through the node

$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}$$

#playouts through
the node

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player
making a move at n
out of all playouts
through the node

#playouts through
the parent of n

$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}$$

#playouts through
the node

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player making a move at n out of all playouts through the node

#playouts through the parent of n

$$\mathrm{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}$$

#playouts through the node

Exploitation term: average utility of n

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player making a move at n out of all playouts through the node

#playouts through the parent of n

$$\text{UCB}(n) = \boxed{\frac{U(n)}{N(n)}} + C \times \boxed{\sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}}$$

#playouts through the node

Exploitation term: average utility of n

Exploration term: high value if n has been explored a few times.

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player making a move at n out of all playouts through the node

#playouts through the parent of n

$$\mathrm{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}$$

#playouts through the node

balance between exploitation and exploration

Exploitation term:
average utility of n

Exploration term:
high value if n has been explored a few times.

Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player making a move at n out of all playouts through the node

#playouts through the parent of n

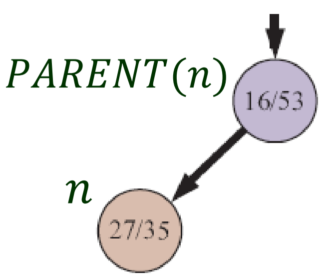
$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}$$

#playouts through the node

balance between exploitation and exploration

Exploitation term: average utility of n

Exploration term: high value if n has been explored a few times.



Ranking of Possible Moves

Upper confidence bound formula:

#wins for the player making a move at n out of all playouts through the node

#playouts through the parent of n

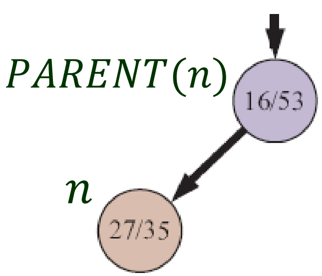
$$\mathrm{UCB}(n) = \boxed{\frac{U(n)}{N(n)}} + C \times \boxed{\sqrt{\frac{\log \boxed{N(PARENT(n))}}{N(n)}}}$$

#playouts through the node

balance between exploitation and exploration

Exploitation term: average utility of n

Exploration term: high value if n has been explored a few times.



$$U(n) = 27$$
$$N(n) = 35$$
$$N\big(PARENT(n)\big) = 53$$
$$C = \sqrt{2}$$ (choice by a theoretical argument)

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

E.g., branching factor b = 32,
100 ply on average for a game.
enough computing power to consider 10^9 states

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

E.g., branching factor b = 32,
100 ply on average for a game.
enough computing power to consider 10^9 states

Minimax can search 6 ply deep: $32^6 \approx 10^9$.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

E.g., branching factor b = 32,
100 ply on average for a game.
enough computing power to consider 10^9 states

Minimax can search 6 ply deep: $32^6 \approx 10^9$.

Alpha-beta can search up to 12 ply.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

  E.g., branching factor b = 32,
  100 ply on average for a game.
  enough computing power to consider 10^9 states

  Minimax can search 6 ply deep: $32^6 \approx 10^9$.

  Alpha-beta can search up to 12 ply.

  Monte Carlo can do 10^7 playouts.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

  E.g., branching factor b = 32,
  100 ply on average for a game.
  enough computing power to consider 10^9 states

  Minimax can search 6 ply deep: $32^6 \approx 10^9$.

  Alpha-beta can search up to 12 ply.

  Monte Carlo can do 10^7 playouts.

- MCTS has advantage over alpha-beta when b is high.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

  E.g., branching factor b = 32,
  100 ply on average for a game.
  enough computing power to consider 10^9 states

  Minimax can search 6 ply deep: $32^6 \approx 10^9$.

  Alpha-beta can search up to 12 ply.

  Monte Carlo can do 10^7 playouts.

- MCTS has advantage over alpha-beta when b is high.

- MCTS is less vulnerable to a single error.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

E.g., branching factor b = 32,
100 ply on average for a game.
enough computing power to consider 10^9 states

Minimax can search 6 ply deep: $32^6 \approx 10^9$.
Alpha-beta can search up to 12 ply.
Monte Carlo can do 10^7 playouts.

- MCTS has advantage over alpha-beta when b is high.
- MCTS is less vulnerable to a single error.
- MCTS can be applied to brand-new games via training by self-play.

More on MCTS

- Time to compute a playout is linear in the depth of the game tree. Compute many playouts before taking one move.

E.g., branching factor b = 32,
100 ply on average for a game.
enough computing power to consider 10^9 states

Minimax can search 6 ply deep: $32^6 \approx 10^9$.

Alpha-beta can search up to 12 ply.

Monte Carlo can do 10^7 playouts.

- MCTS has advantage over alpha-beta when b is high.

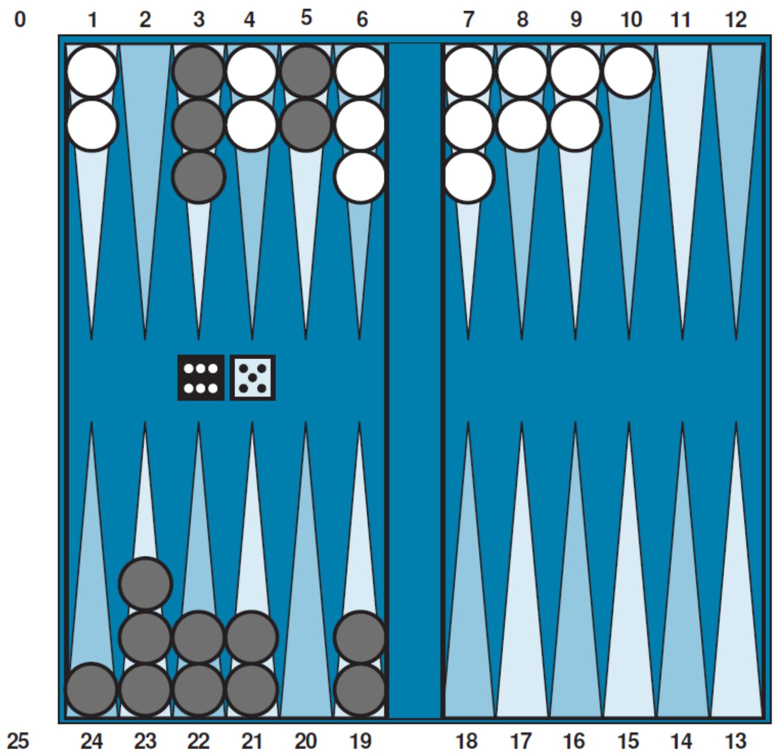
- MCTS is less vulnerable to a single error.

- MCTS can be applied to brand-new games via training by self-play.

- MCTS is less desired than alpha-beta on a game like chess with low b and good evaluation function.

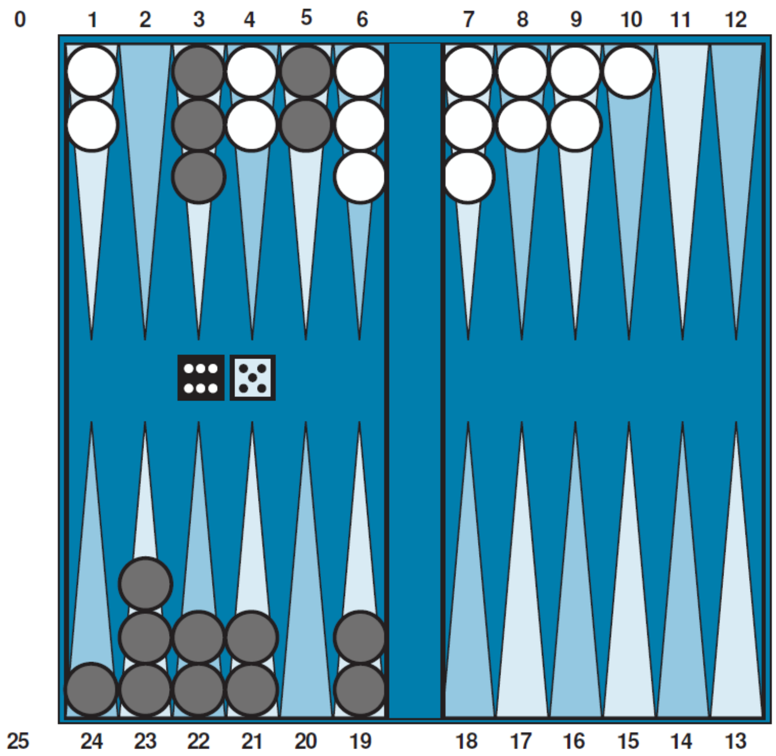
Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



Stochastic Games

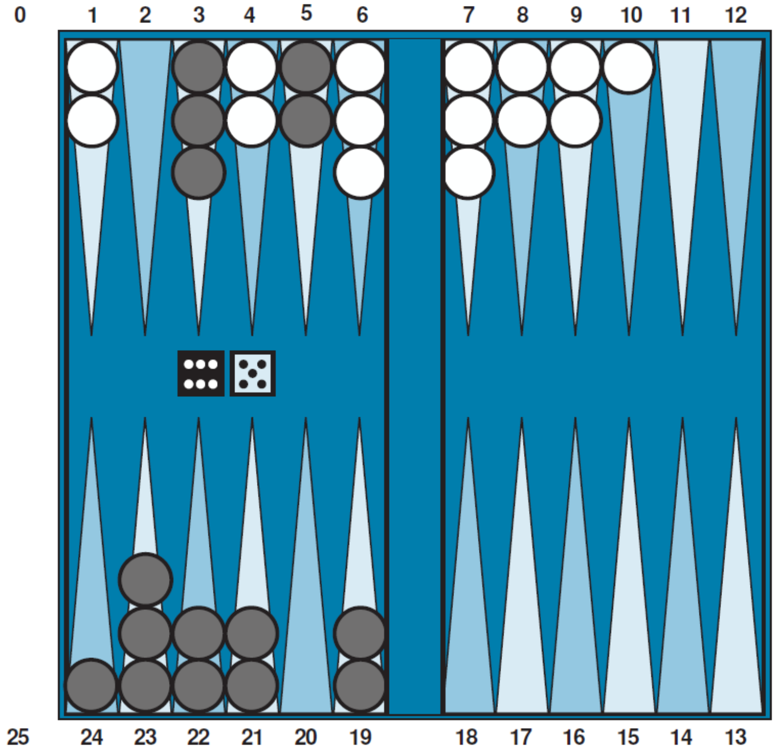
Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



- Include *chance nodes* in addition to MAX and MIN nodes.

Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.

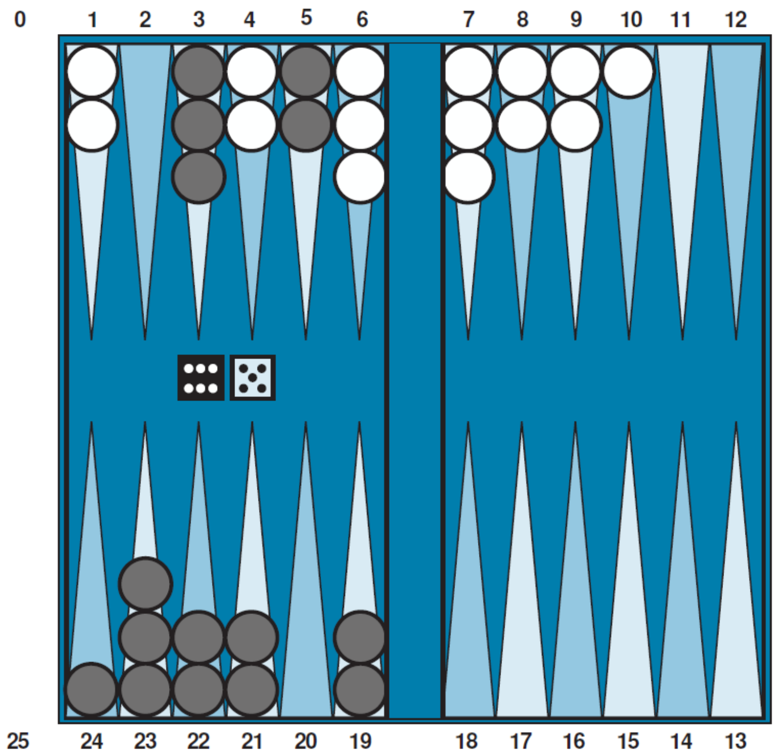


- Include *chance nodes* in addition to MAX and MIN nodes.

Throwing two dice (unordered):

Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



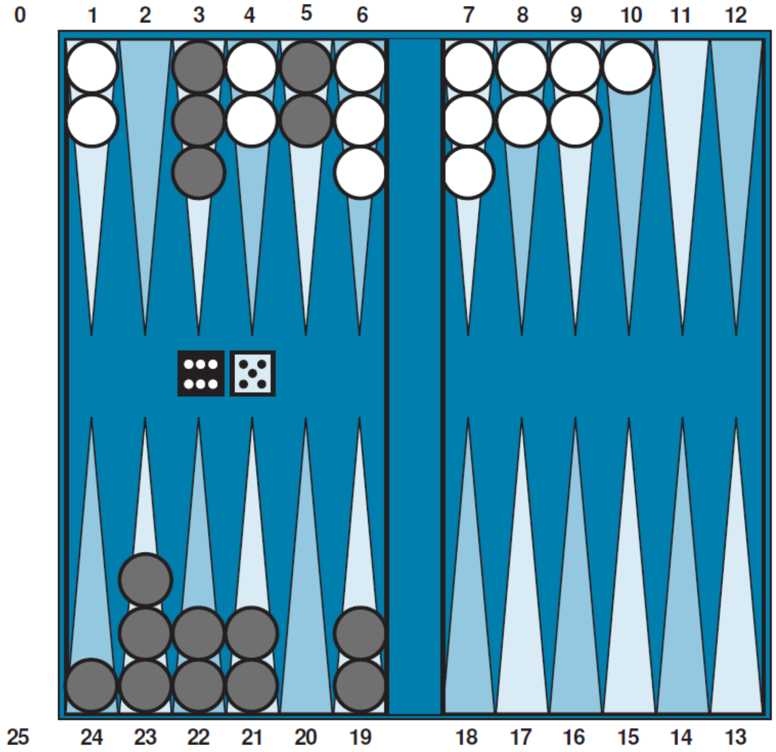
- Include *chance nodes* in addition to MAX and MIN nodes.

Throwing two dice (unordered):

1-1, …, 6-6: probability $1/36$ each

Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



- Include *chance nodes* in addition to MAX and MIN nodes.

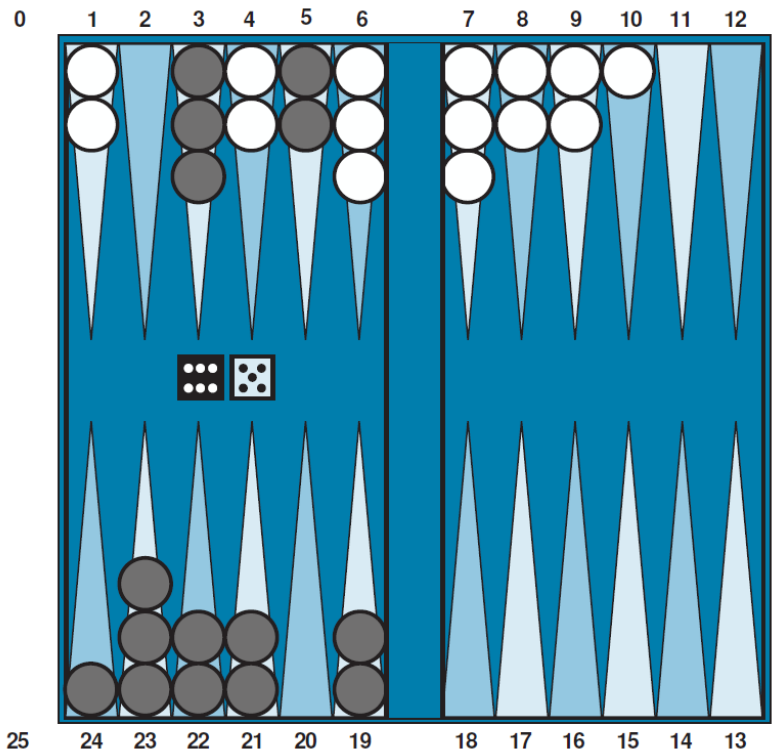
Throwing two dice (unordered):

1-1, …, 6-6: probability $1/36$ each

1-2,…,1-6,2-3,…, 6-6: probability $1/18$ each

Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



- Include *chance nodes* in addition to MAX and MIN nodes.

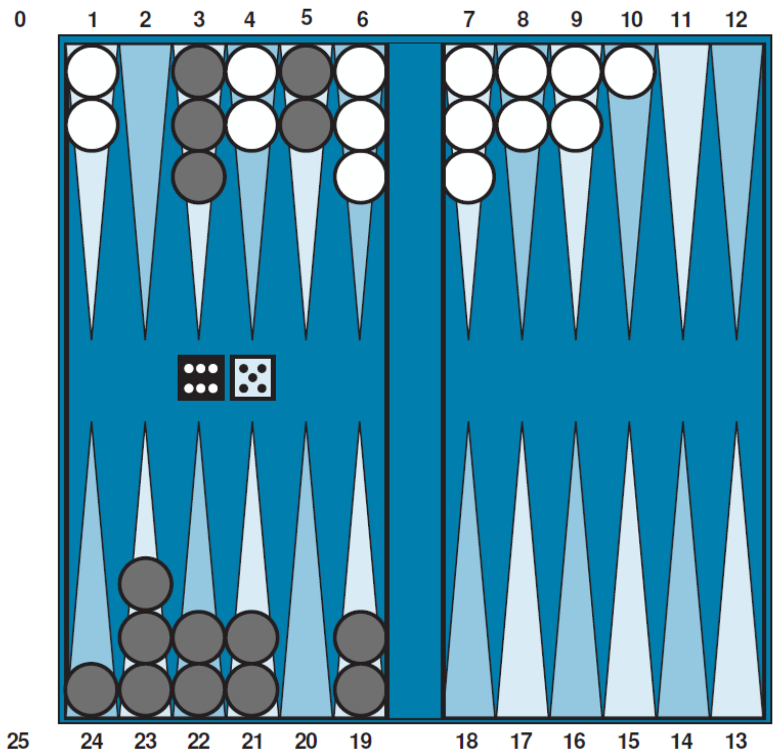
Throwing two dice (unordered):

1-1, …, 6-6: probability $1/36$ each

$\underbrace{\text{1-2,…,1-6,2-3,…, 6-6}}_{15}$: probability $1/18$ each

Stochastic Games

Some games (e.g., backgammon) have randomness due to throwing of dice. They combine both luck and skill.



- Include *chance nodes* in addition to MAX and MIN nodes.

Throwing two dice (unordered):

1-1, …, 6-6: probability $1/36$ each

$\underbrace{\text{1-2,…,1-6,2-3,…, 6-6}}_{15}$: probability $1/18$ each

- Calculate expected value (called *expectiminimax* value) of a position.

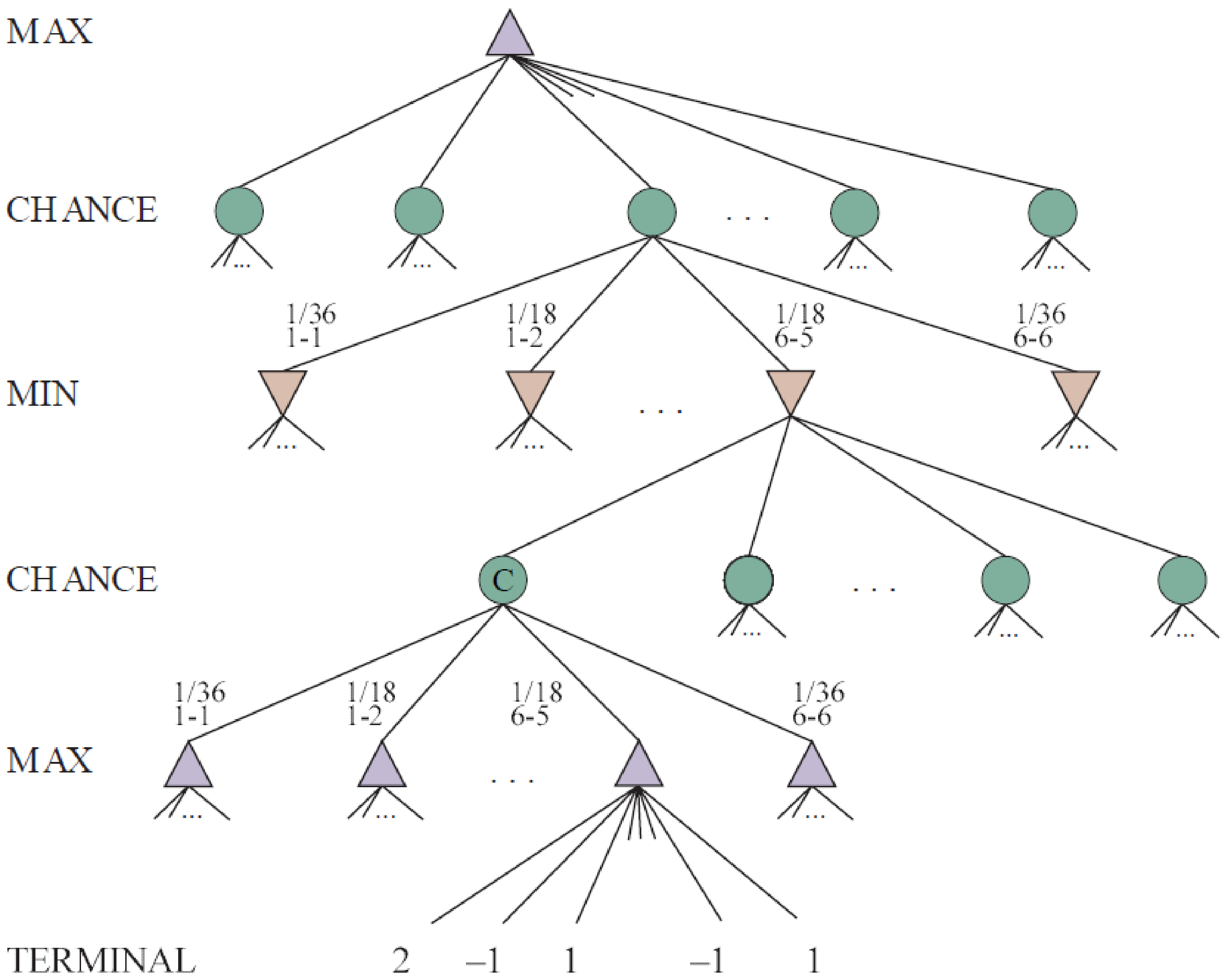
Expectiminimax Value

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{Max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{Max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{Min} \\ \boxed{\sum_r P(r)\, \textsc{Expectiminimax}(\textsc{Result}(s,r))} & \text{if } \textsc{To-Move}(s) = \textsc{Chance} \end{cases}$$

expected value

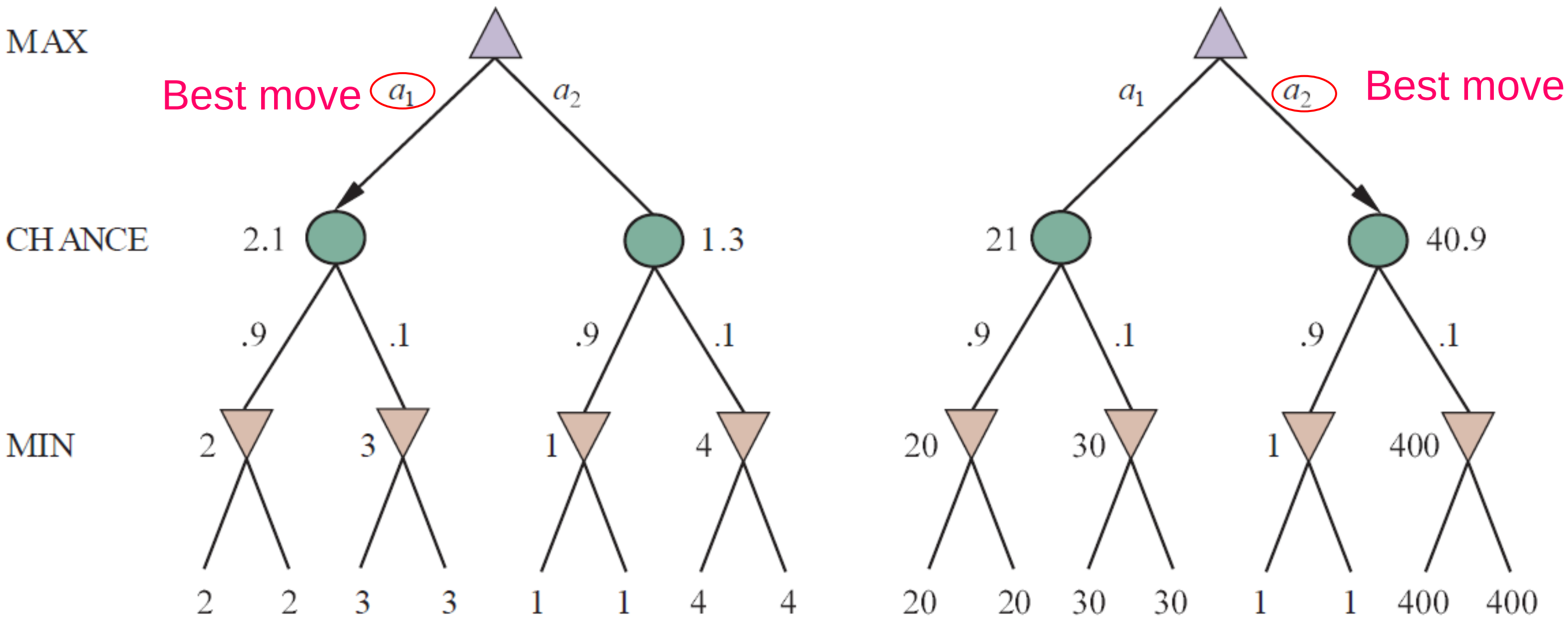
one possible dice roll

Game Tree for a Backgammon Position



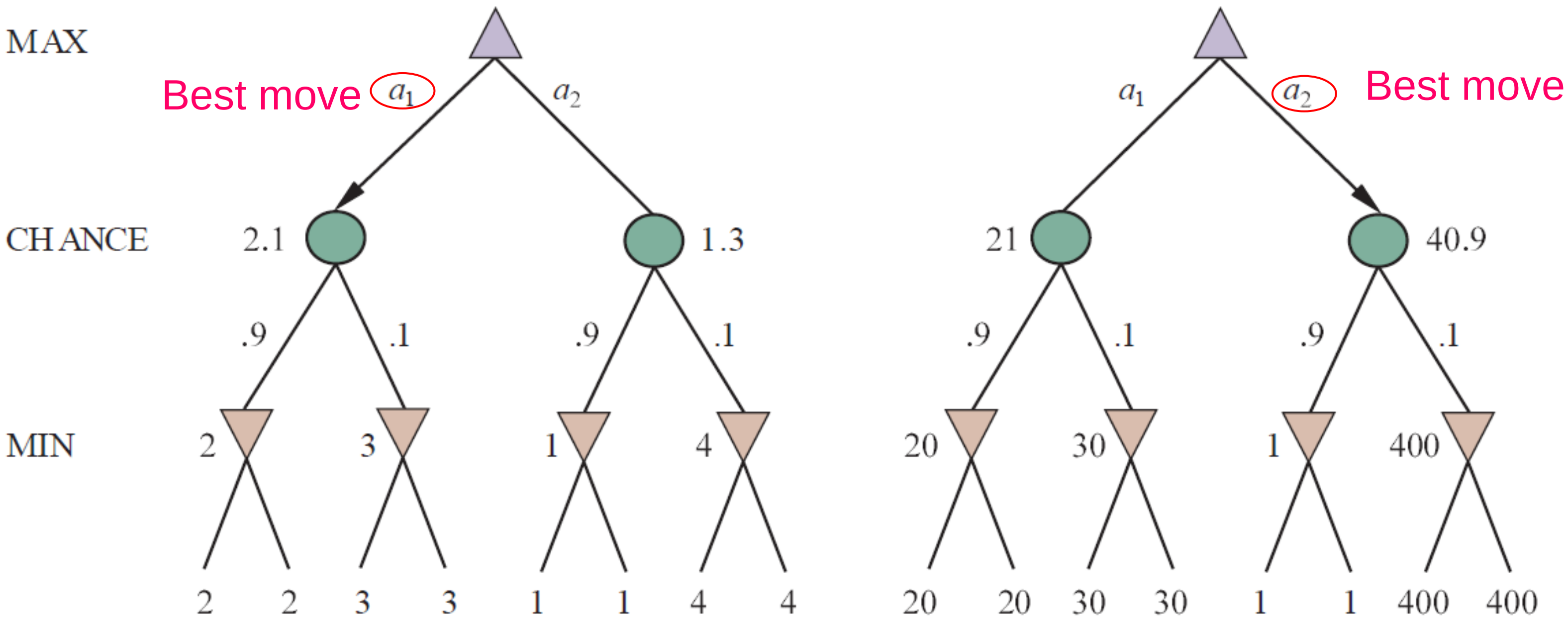
Evaluation Functions

Evaluation functions with the same order of leaf values can yield different move choices at a state.



Evaluation Functions

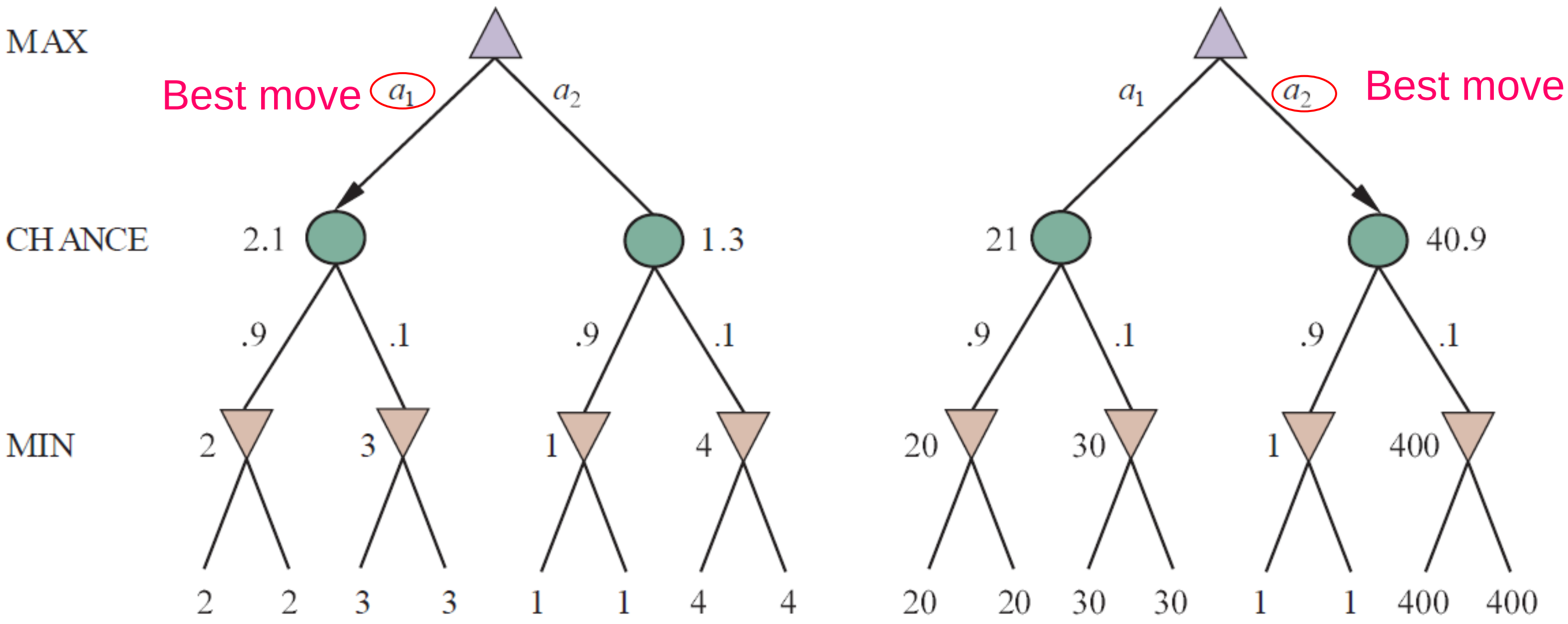
Evaluation functions with the same order of leaf values can yield different move choices at a state.



Alpha-beta pruning is still applicable if we can bound values on chance nodes.

Evaluation Functions

Evaluation functions with the same order of leaf values can yield different move choices at a state.



Alpha-beta pruning is still applicable if we can bound values on chance nodes.