Monte Carlo Tree Search (MCTS)

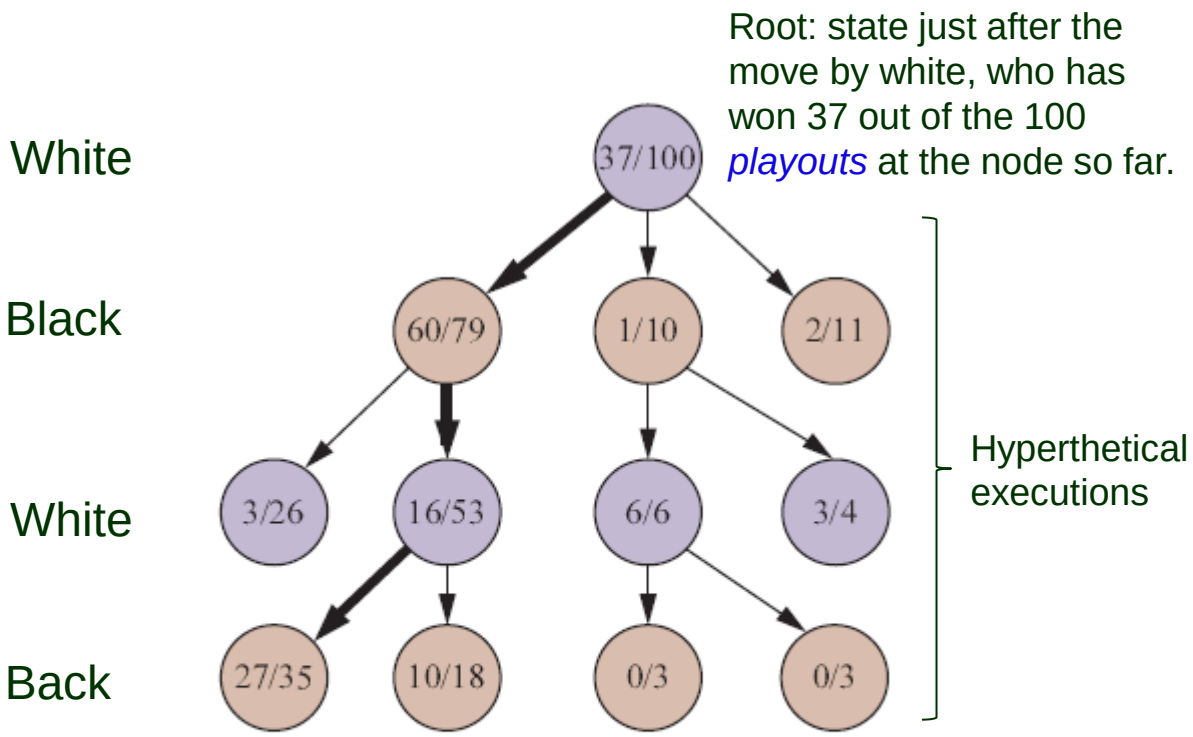
Casino de Monte-Carlo, Monaco

- Online planning algorithm.
- Intelligent tree search that balances exploration and exploitation.
- Random sampling in the form of simulations (*playouts*).
- Storage of the statistics of actions to improve choices in the future.
- Combined with neural nets, it is behind many major successes of AI applications such as games (e.g., AlphaGo), scheduling, planning, and optimization.

* Figures/images are from the textbook site (or by the instructor) unless the sources are given otherwise.

Step 1: Selection

We need to decide a move for black (at the root) to make.



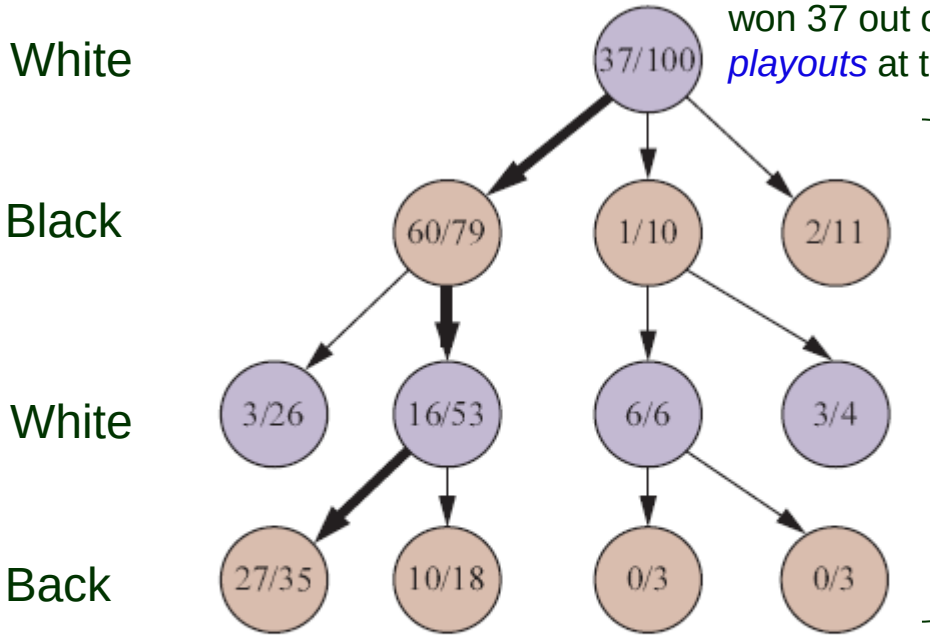
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

Step 1: Selection

We need to decide a move for black (at the root) to make.

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



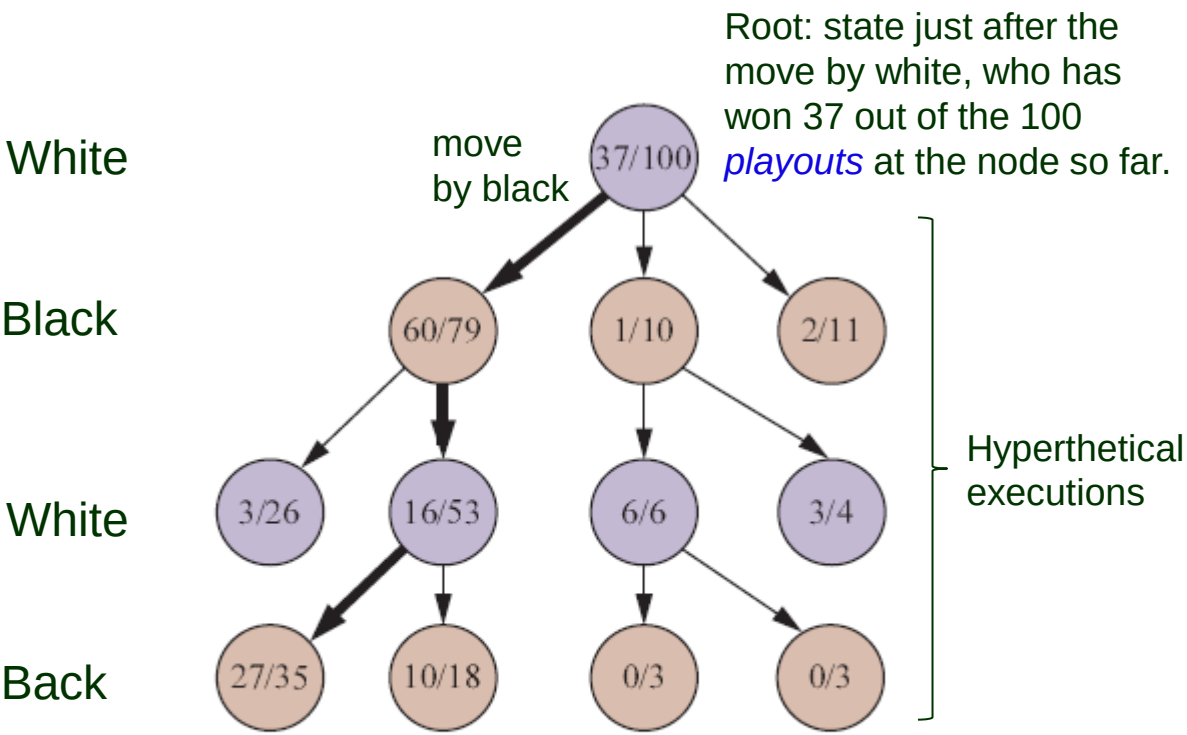
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

Step 1: Selection

We need to decide a move for black (at the root) to make.



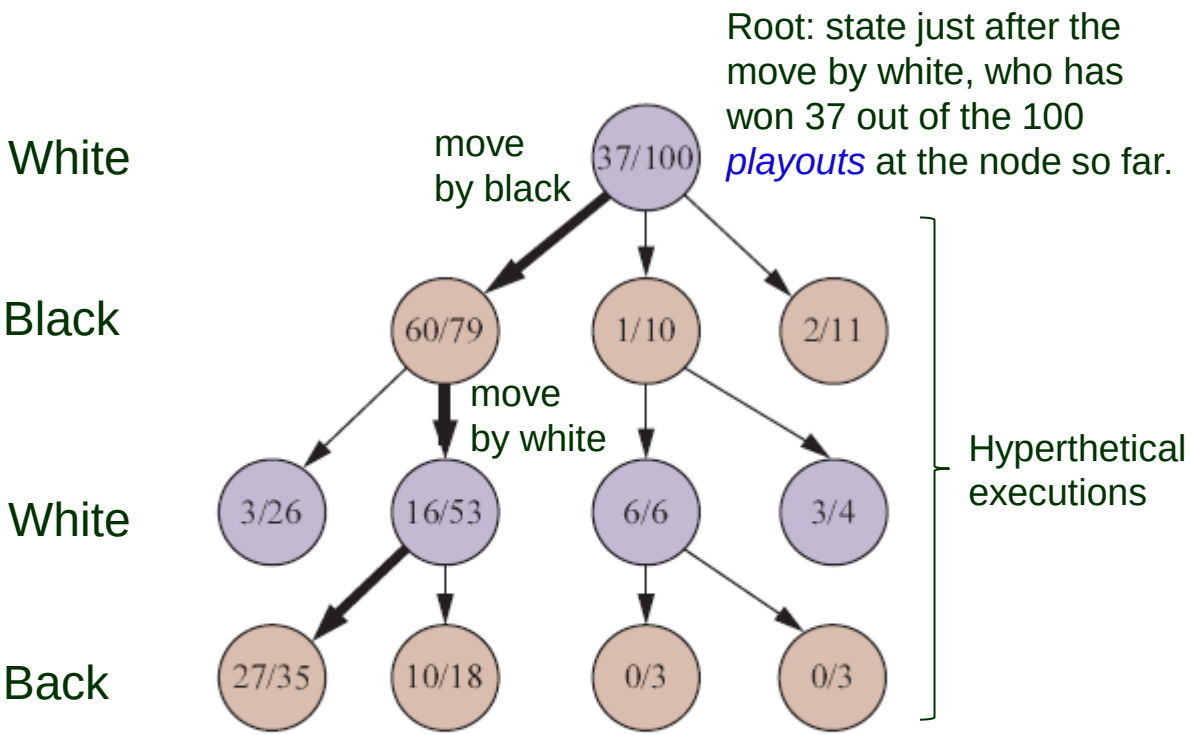
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

Step 1: Selection

We need to decide a move for black (at the root) to make.



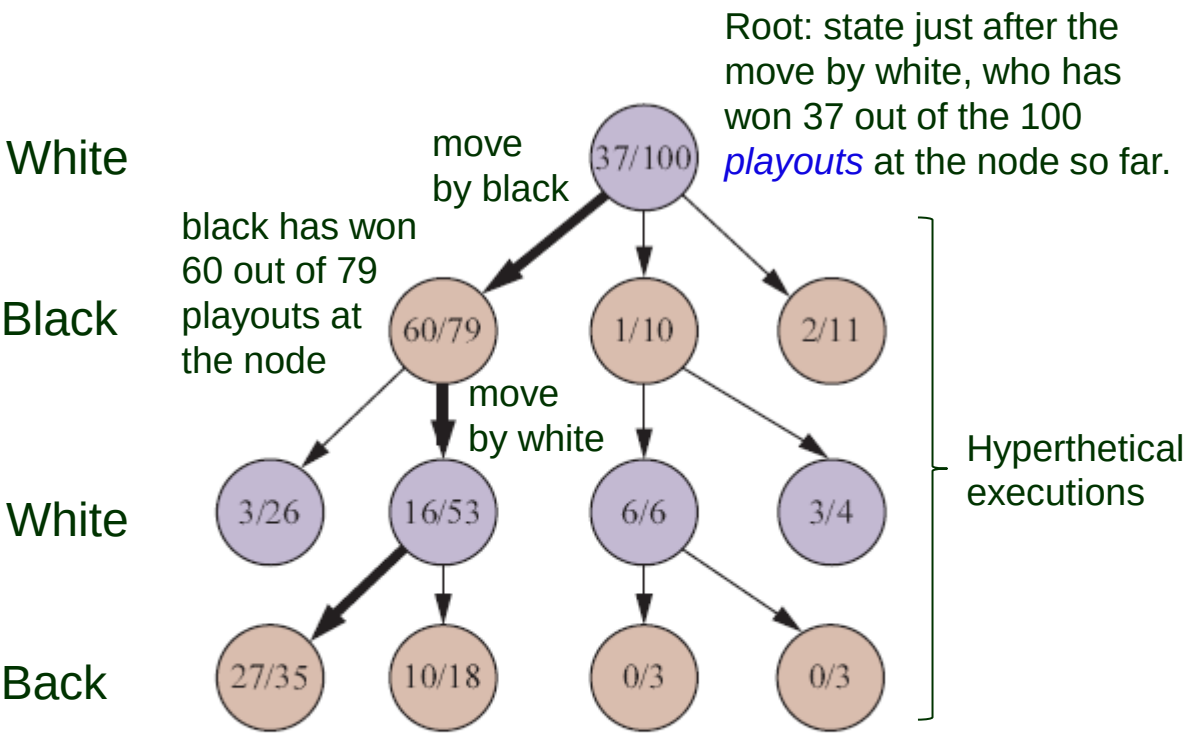
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

Step 1: Selection

We need to decide a move for black (at the root) to make.



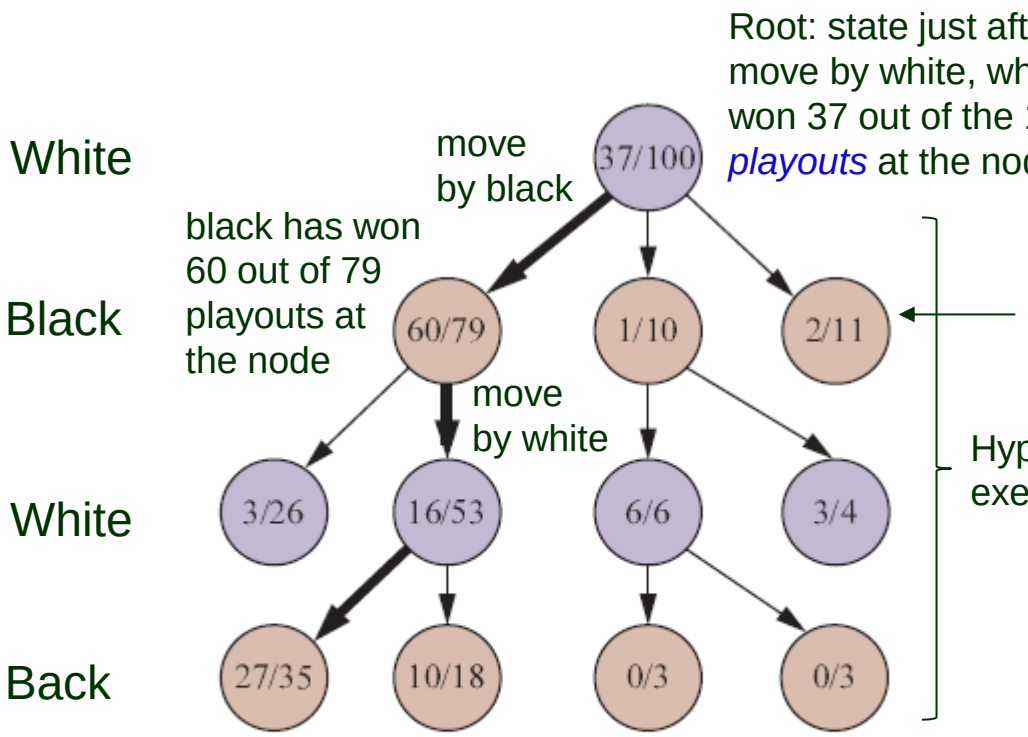
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

Step 1: Selection

We need to decide a move for black (at the root) to make.



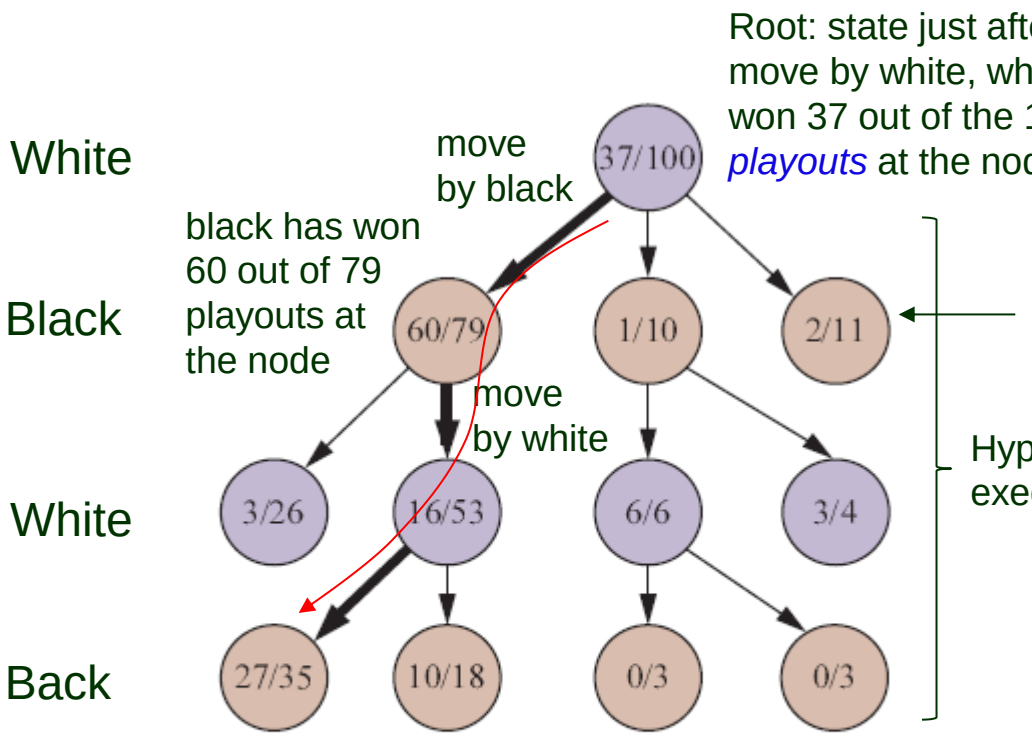
(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

Step 1: Selection

We need to decide a move for black (at the root) to make.

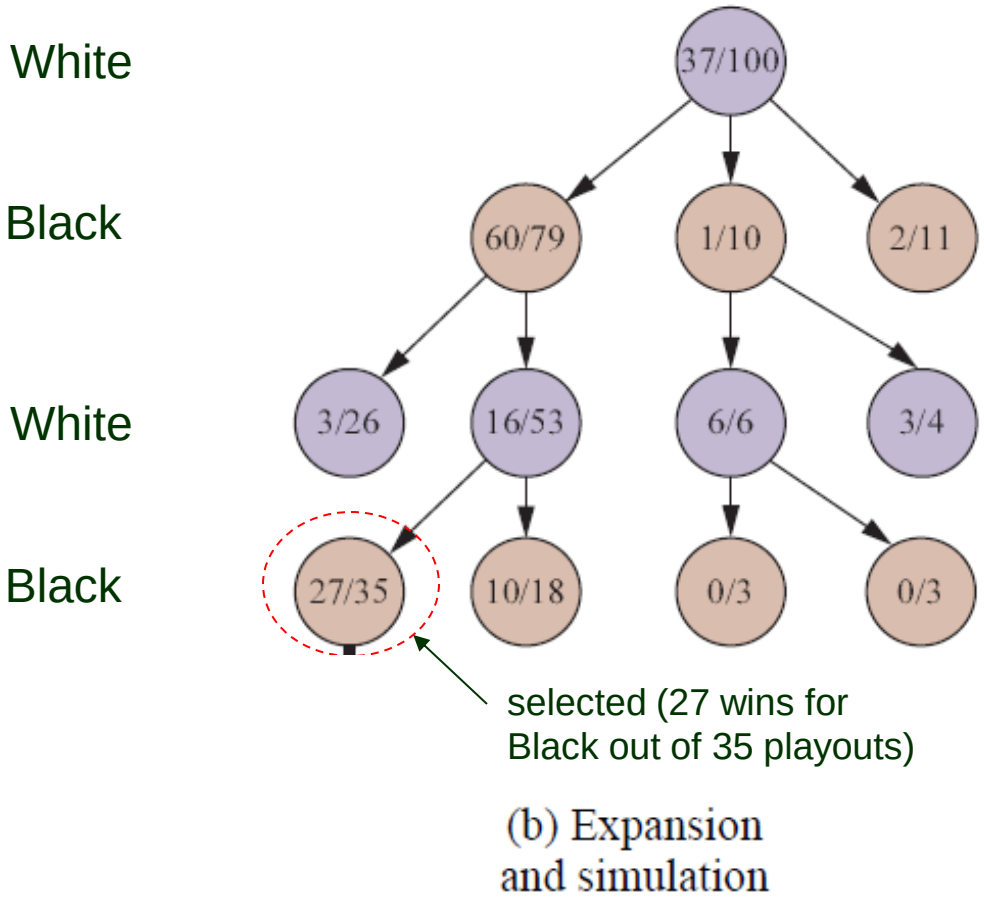


(a) Selection

- Unlike in the minimax game tree, a directed edge represents a move by the player at the edge's destination.

At each node a move is selected according to some policy (e.g., the upper confidence bound to be introduced).

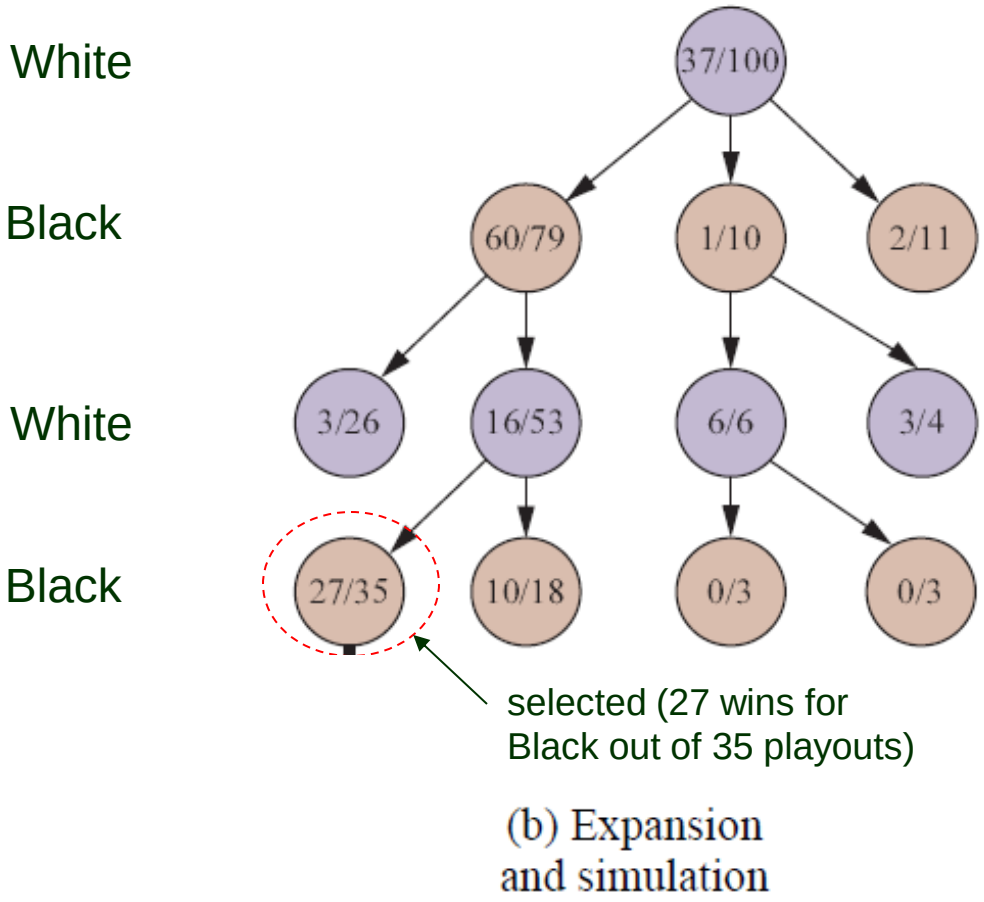
Steps 2 & 3: Expansion & Simulation



(b) Expansion and simulation

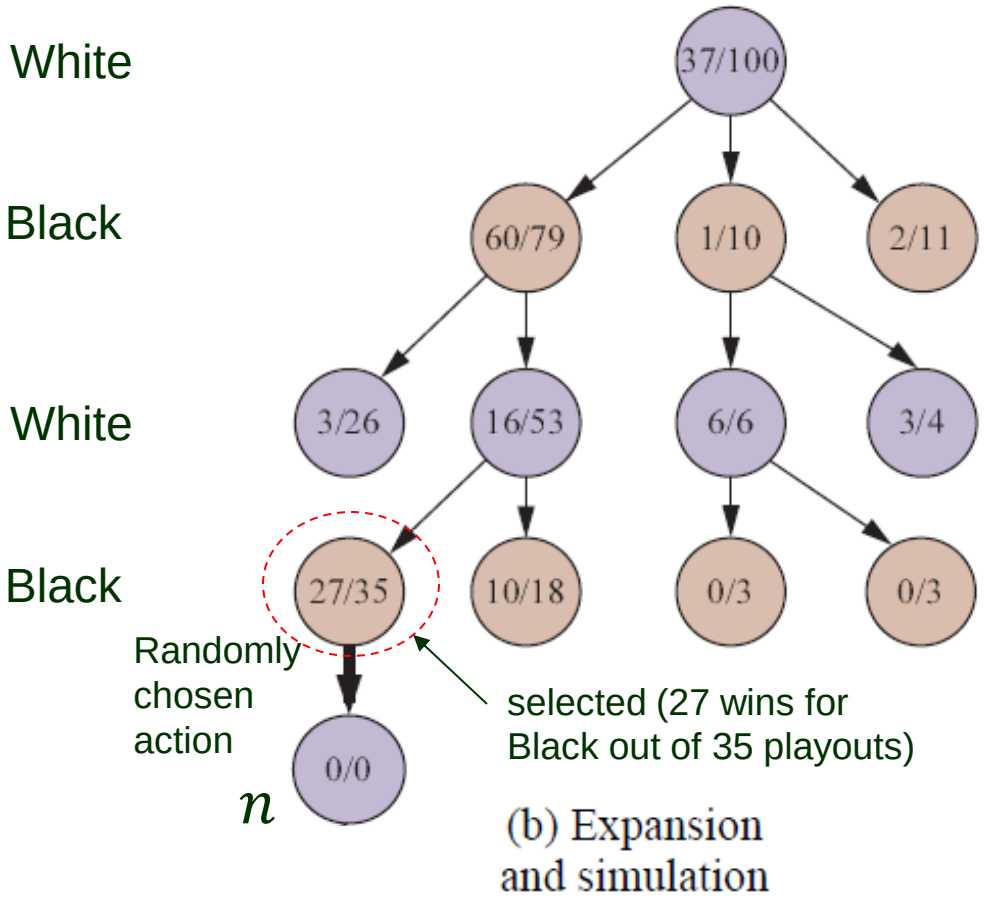
- Create *one or more* child nodes of the selected node.
  - ♦ You may generate a child node by executing every feasible action.
  - ♦ Or you may generate child nodes by executing a subset of feasible actions.
  - ♦ But you shouldn't *always* generate just one child node when there is more than one feasible action. Otherwise, the search tree within each iteration will have just one leaf – which will then be selected – and the tree will degenerate into a chain.

Steps 2 & 3: Expansion & Simulation



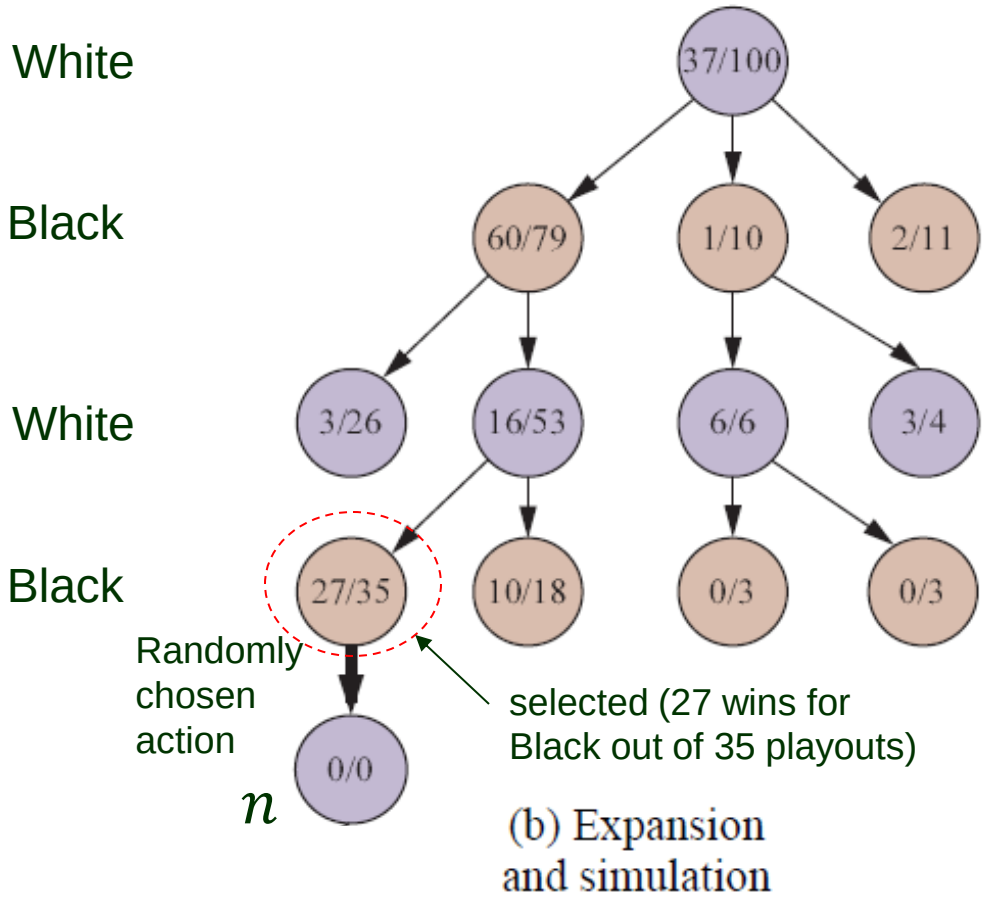
- Create *one or more* child nodes of the selected node.
  - ♦ You may generate a child node by executing every feasible action.
  - ♦ Or you may generate child nodes by executing a subset of feasible actions.
  - ♦ But you shouldn't *always* generate just one child node when there is more than one feasible action. Otherwise, the search tree within each iteration will have just one leaf – which will then be selected – and the tree will degenerate into a chain.
- One of the child nodes, say, n, is (randomly) chosen.

Steps 2 & 3: Expansion & Simulation



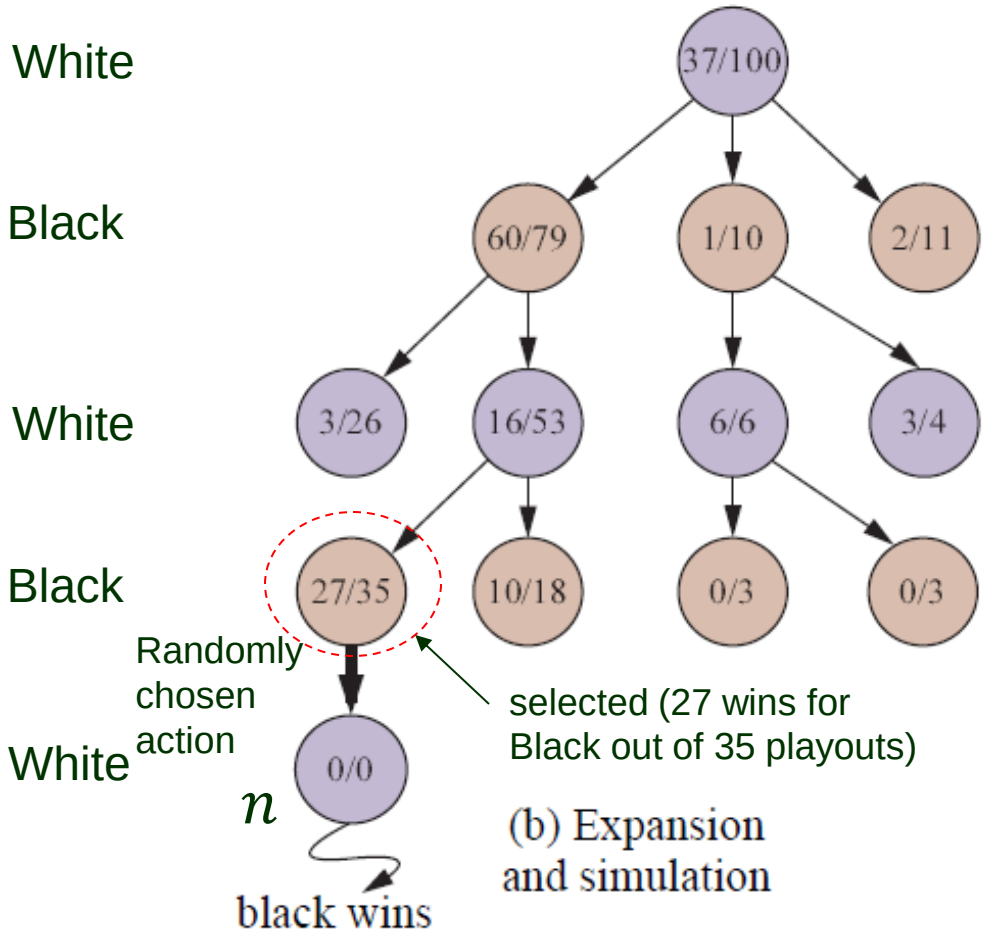
- Create *one or more* child nodes of the selected node.
  - ♦ You may generate a child node by executing every feasible action.
  - ♦ Or you may generate child nodes by executing a subset of feasible actions.
  - ♦ But you shouldn't *always* generate just one child node when there is more than one feasible action. Otherwise, the search tree within each iteration will have just one leaf – which will then be selected – and the tree will degenerate into a chain.
- One of the child nodes, say, n, is (randomly) chosen.

Steps 2 & 3: Expansion & Simulation



- Create *one or more* child nodes of the selected node.
  - ♦ You may generate a child node by executing every feasible action.
  - ♦ Or you may generate child nodes by executing a subset of feasible actions.
  - ♦ But you shouldn't *always* generate just one child node when there is more than one feasible action. Otherwise, the search tree within each iteration will have just one leaf – which will then be selected – and the tree will degenerate into a chain.
- One of the child nodes, say, n, is (randomly) chosen.
- Perform a playout from the node n.

Steps 2 & 3: Expansion & Simulation



- Create *one or more* child nodes of the selected node.
  - ♦ You may generate a child node by executing every feasible action.
  - ♦ Or you may generate child nodes by executing a subset of feasible actions.
  - ♦ But you shouldn't *always* generate just one child node when there is more than one feasible action. Otherwise, the search tree within each iteration will have just one leaf – which will then be selected – and the tree will degenerate into a chain.
- One of the child nodes, say, n, is (randomly) chosen.
- Perform a playout from the node n.

Step 4: Back Propagation

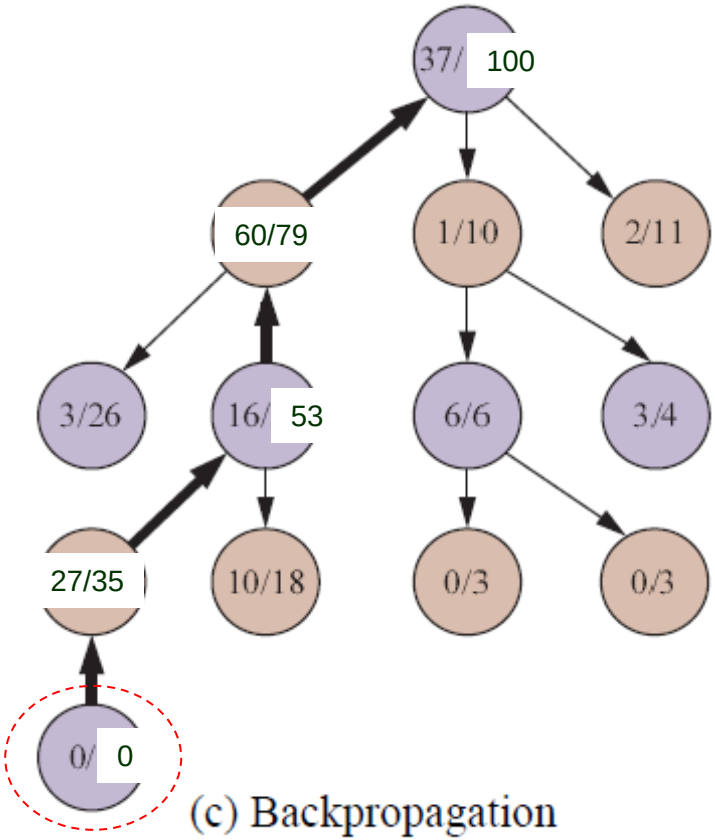
White

Black

White

Black

White



(c) Backpropagation

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

Step 4: Back Propagation

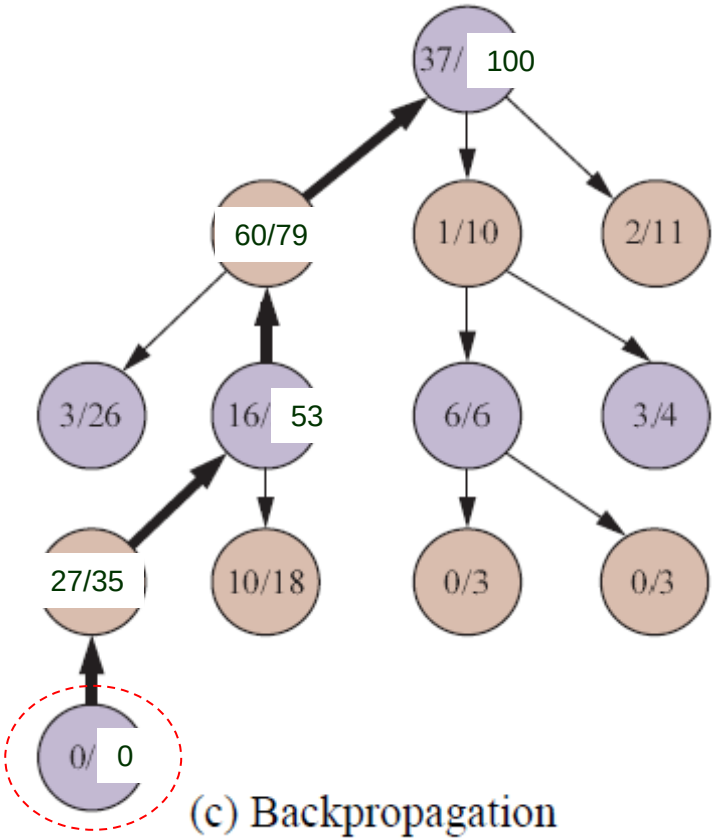
White

Black

White

Black

White



(c) Backpropagation

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

Black wins this playout:

Step 4: Back Propagation

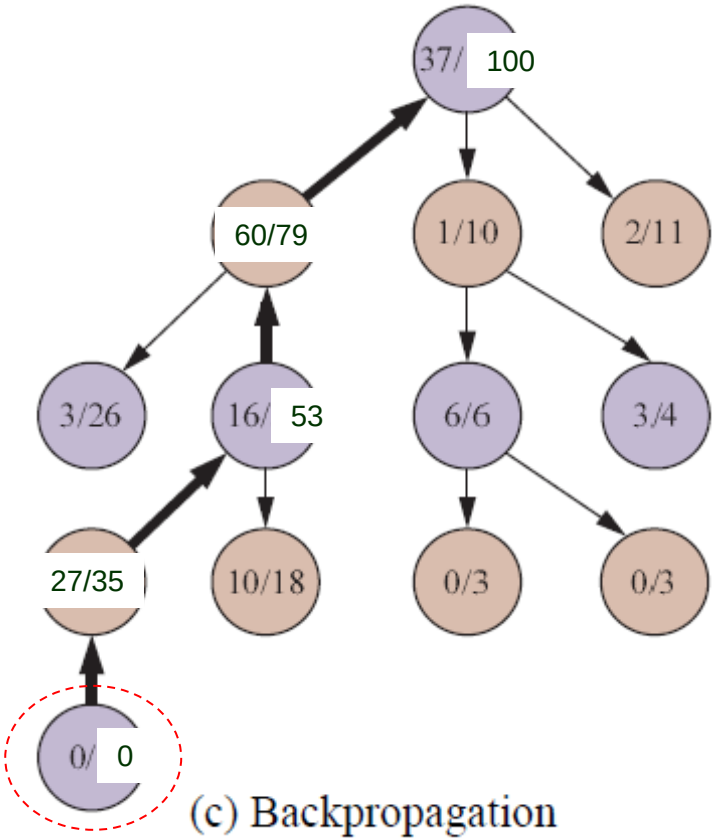
White

Black

White

Black

White



(c) Backpropagation

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

Black wins this playout:

- At a white node, increment #playouts only.

Step 4: Back Propagation

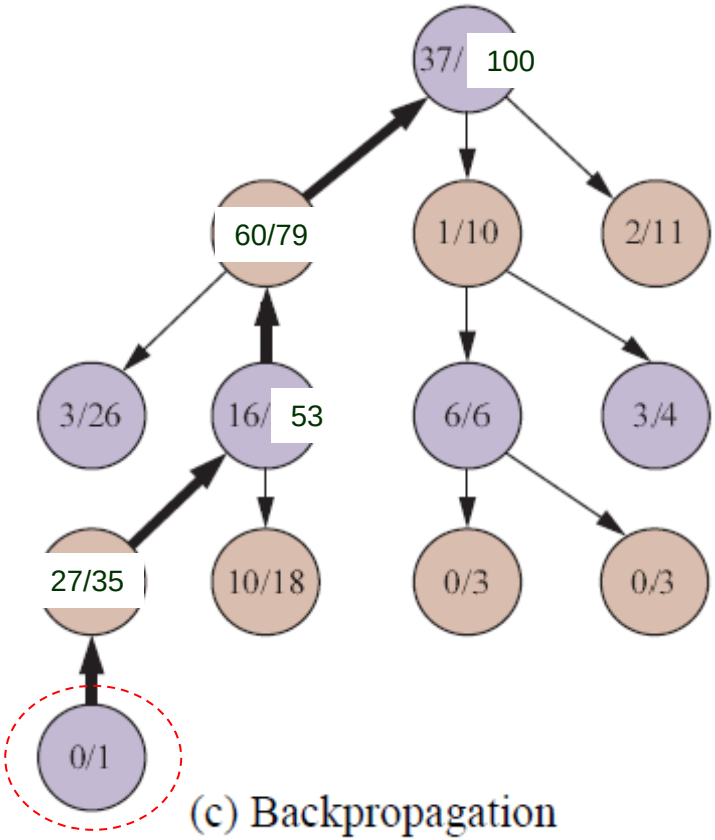
White

Black

White

Black

White



(c) Backpropagation

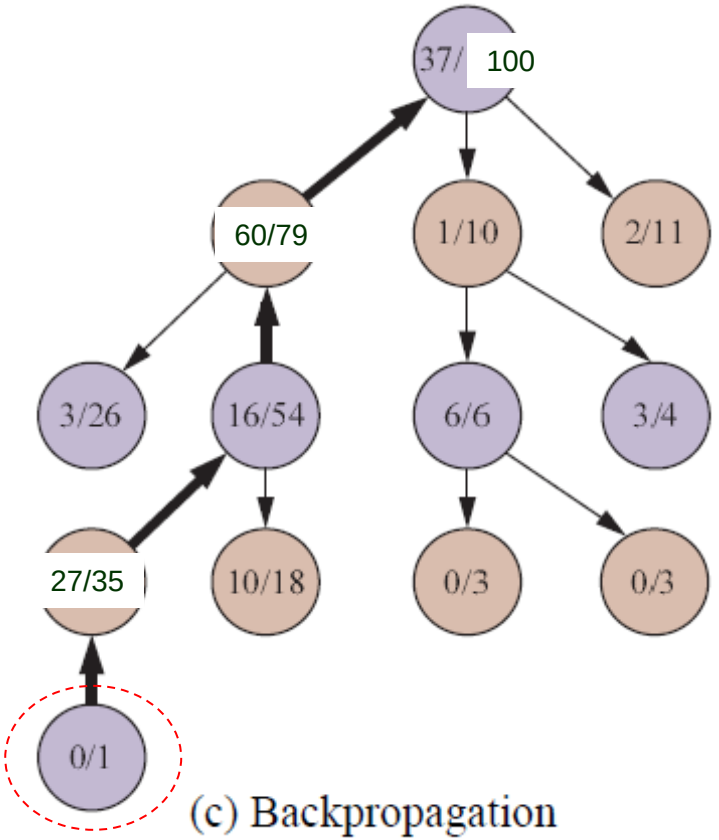
- Update all the nodes upward along the path until the root.

Black wins this playout:

- At a white node, increment #playouts only.

Step 4: Back Propagation

White
Black
White
Black
White



(c) Backpropagation

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

Black wins this playout:

- At a white node, increment #playouts only.

Step 4: Back Propagation

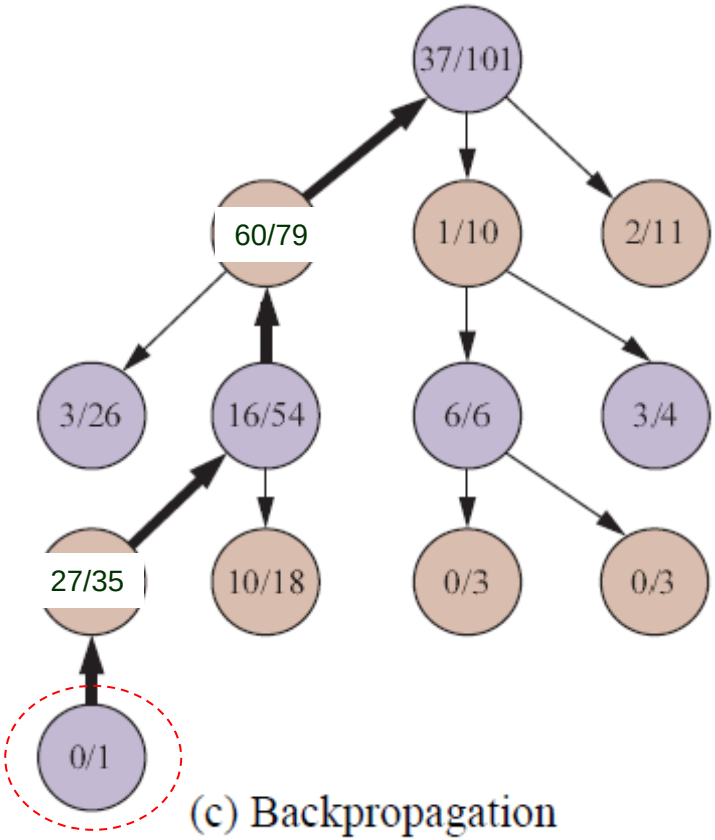
White

Black

White

Black

White



(c) Backpropagation

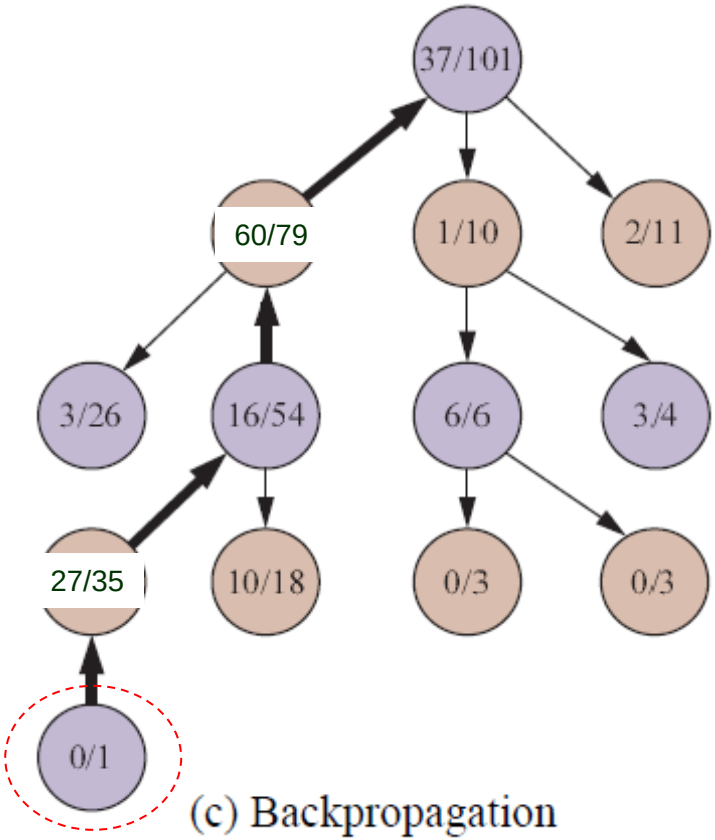
- Update all the nodes upward along the path until the root.

Black wins this playout:

- At a white node, increment #playouts only.

Step 4: Back Propagation

White
Black
White
Black
White



(c) Backpropagation

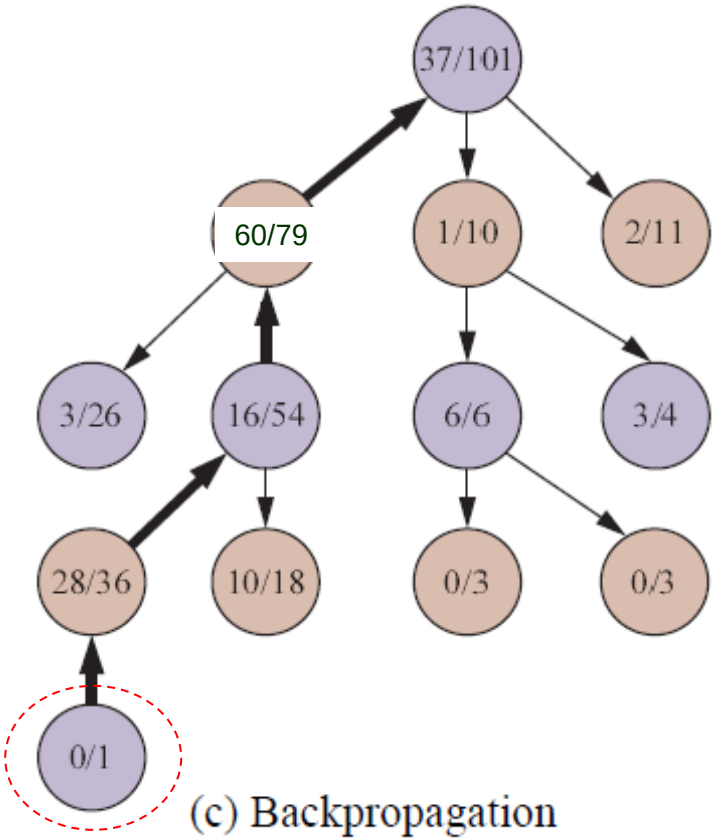
- Update all the nodes upward along the path until the root.

Black wins this playout:

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

Step 4: Back Propagation

White
Black
White
Black
White



(c) Backpropagation

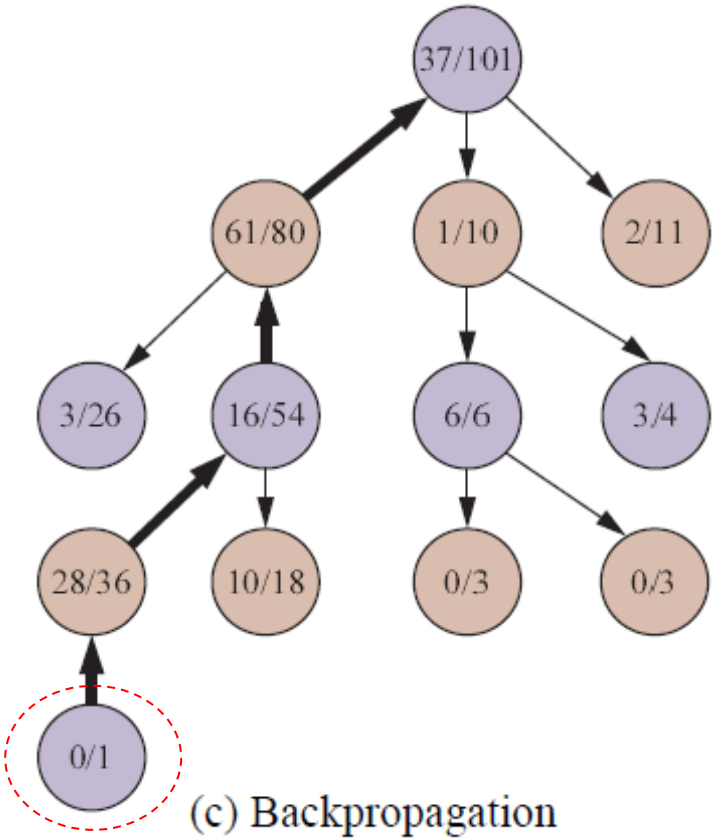
- Update all the nodes upward along the path until the root.

Black wins this playout:

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

Step 4: Back Propagation

White
Black
White
Black
White



(c) Backpropagation

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

Black wins this playout:

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

Termination

- MCTS repeats the four steps (selection, expansion, simulation, back-propagation) in order until
  - a set number N of iterations have been performed, or
  - the allotted time has expired.
- It returns the move with the highest number of playouts.

Termination

- MCTS repeats the four steps (selection, expansion, simulation, back-propagation) in order until
  - a set number N of iterations have been performed, or
  - the allotted time has expired.
- It returns the move with the highest number of playouts.

Why not the highest ratio?

Termination

- MCTS repeats the four steps (selection, expansion, simulation, back-propagation) in order until
  - a set number N of iterations have been performed, or
  - the allotted time has expired.
- It returns the move with the highest number of playouts.

Why not the highest ratio?

  - Since better moves are more likely to be chosen, the most promising move is expected to have the highest number of playouts.

Termination

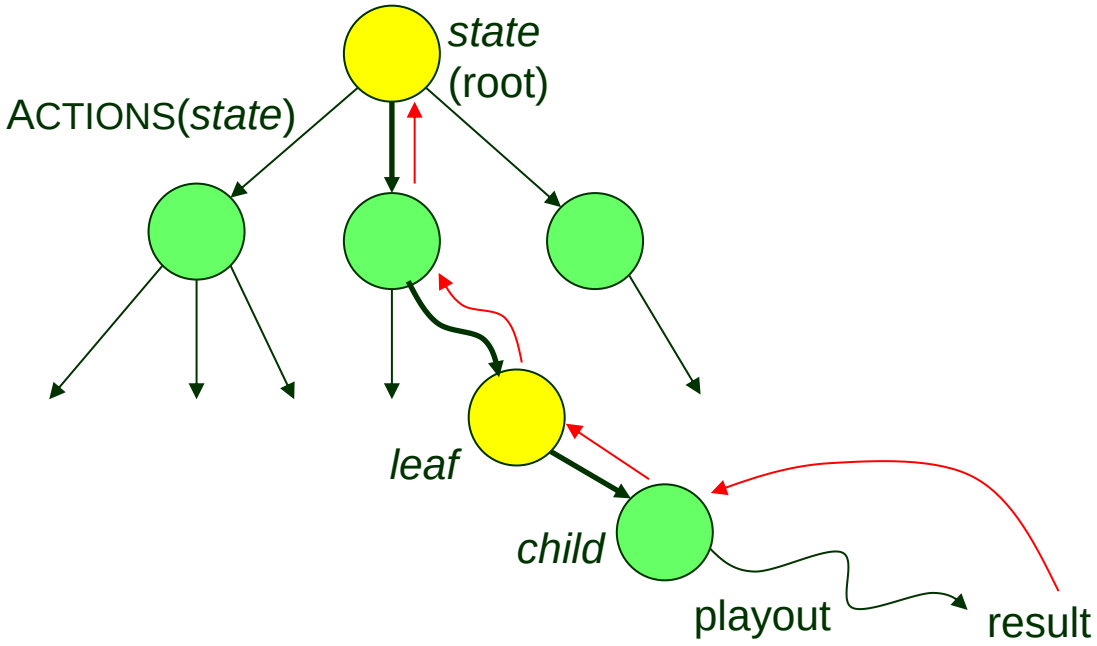
- MCTS repeats the four steps (selection, expansion, simulation, back-propagation) in order until
  - a set number N of iterations have been performed, or
  - the allotted time has expired.
- It returns the move with the highest number of playouts.

Why not the highest ratio?

- Since better moves are more likely to be chosen, the most promising move is expected to have the highest number of playouts.
- A node with 65/100 wins is better than one with 2/3 wins (which has a lot of uncertainty).

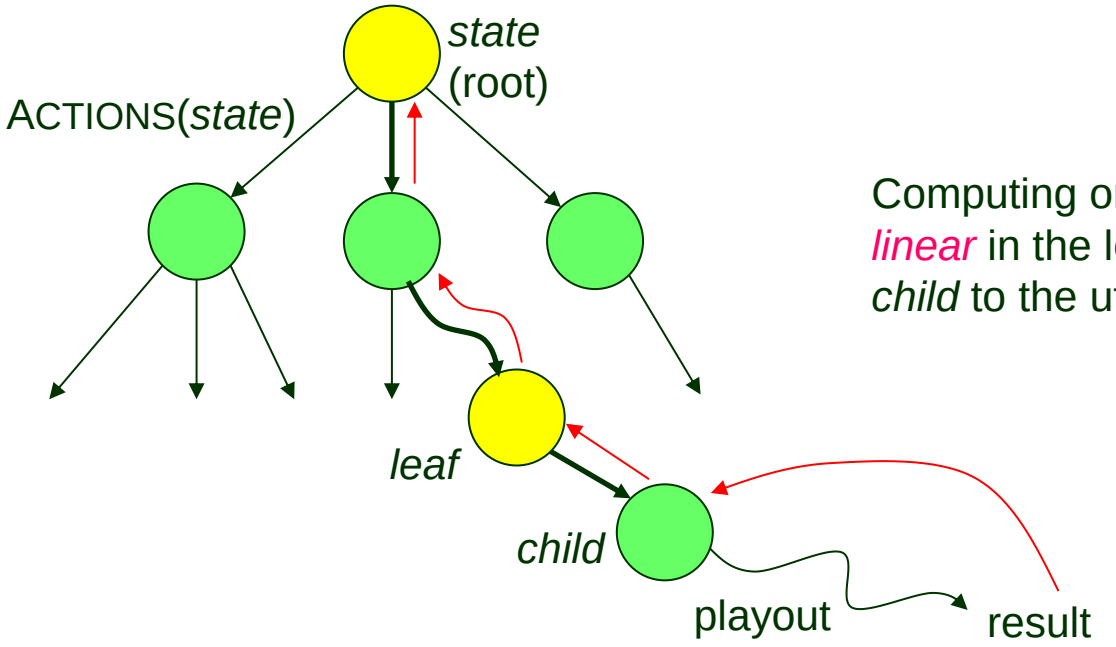
Monte Carlo Tree Search Algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action* // decide a move at *state.*
  tree ← NODE(*state*) // initialize the tree with *state* at the root
  while IS-TIME-REMAINING() **do** // N iterations, each expanding the tree by one node.
    leaf ← SELECT(*tree*) // the node to be expanded must be a leaf.
    child ← EXPAND(*leaf*) // tree is expanded to a child of *leaf.*
    result ← SIMULATE(*child*) // playout: moves are not recorded in the tree.
    BACK-PROPAGATE(*result*, *child*) // update nodes on the path upward to the root.
  return the move in ACTIONS(*state*) whose node has highest number of playouts



Monte Carlo Tree Search Algorithm

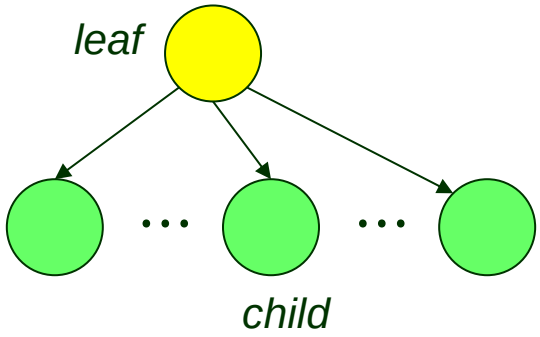
function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action* // decide a move at *state.*
  tree ← NODE(*state*) // initialize the tree with *state* at the root
  while IS-TIME-REMAINING() **do** // N iterations, each expanding the tree by one node.
    leaf ← SELECT(*tree*) // the node to be expanded must be a leaf.
    child ← EXPAND(*leaf*) // tree is expanded to a child of *leaf.*
    result ← SIMULATE(*child*) // playout: moves are not recorded in the tree.
    BACK-PROPAGATE(*result*, *child*) // update nodes on the path upward to the root.
  return the move in ACTIONS(*state*) whose node has highest number of playouts



Computing one playout takes time *linear* in the length of the path from *child* to the utility node (result).

Three Issues

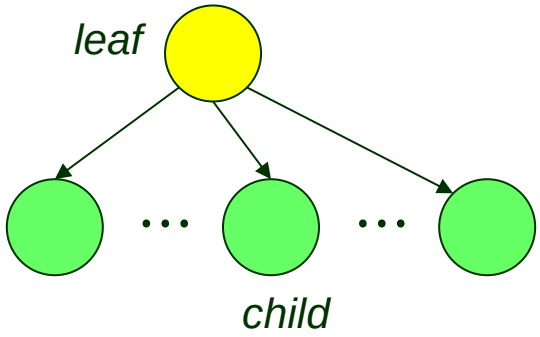
- *child* ← EXPAND(*leaf*)



Create one or more child nodes and choose node *child* from one of them.

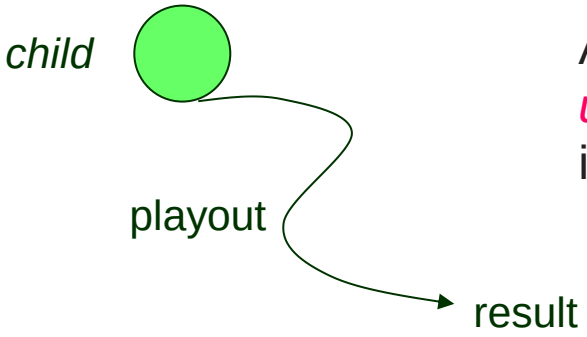
Three Issues

- *child* ← EXPAND(*leaf*)



Create one or more child nodes and choose node *child* from one of them.

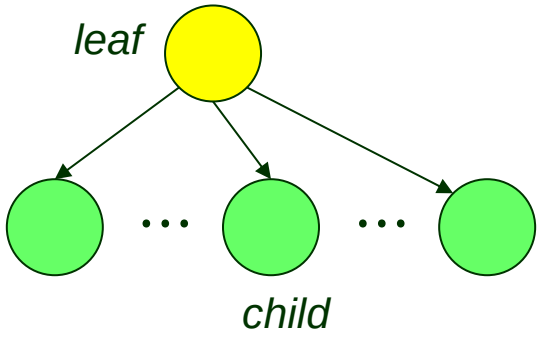
- *result* ← SIMULATE(*child*)



A playout may be as simple as choosing *uniformly random* moves until the game is decided.

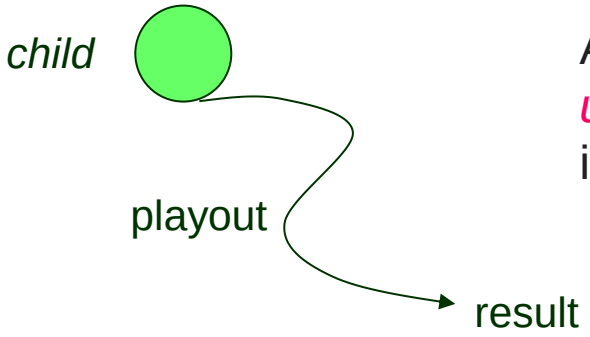
Three Issues

- *child* ← EXPAND(*leaf*)



Create one or more child nodes and choose node *child* from one of them.

- *result* ← SIMULATE(*child*)



A playout may be as simple as choosing *uniformly random* moves until the game is decided.

- IS-TIME-REMAINING()

Pure Monte Carlo search does *N* simulations instead.

Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

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

At a node n in the expanding MCTS tree rooted at the real *state*.

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

At a node n in the expanding MCTS tree rooted at the real *state*.

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

At a node n in the expanding MCTS tree rooted at the real *state*.

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

At a node n in the expanding MCTS tree rooted at the real *state*.

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

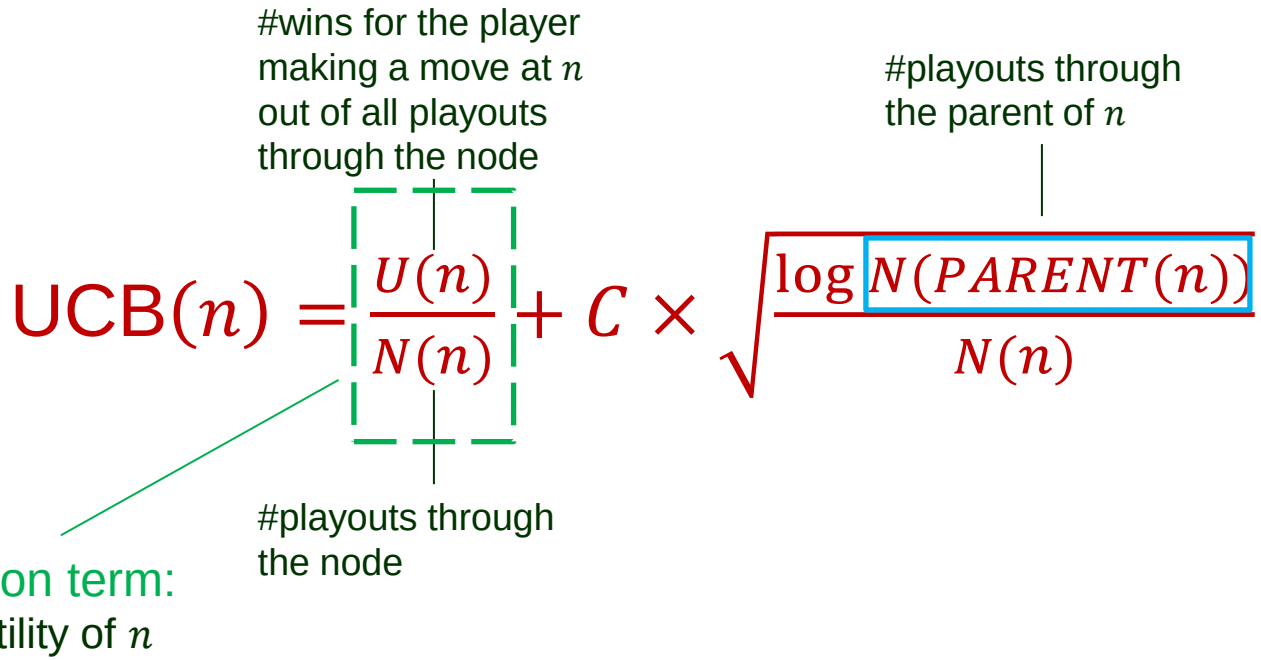
$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

#playouts through the node

Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

Upper confidence bound formula:



Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

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

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

#playouts through the node

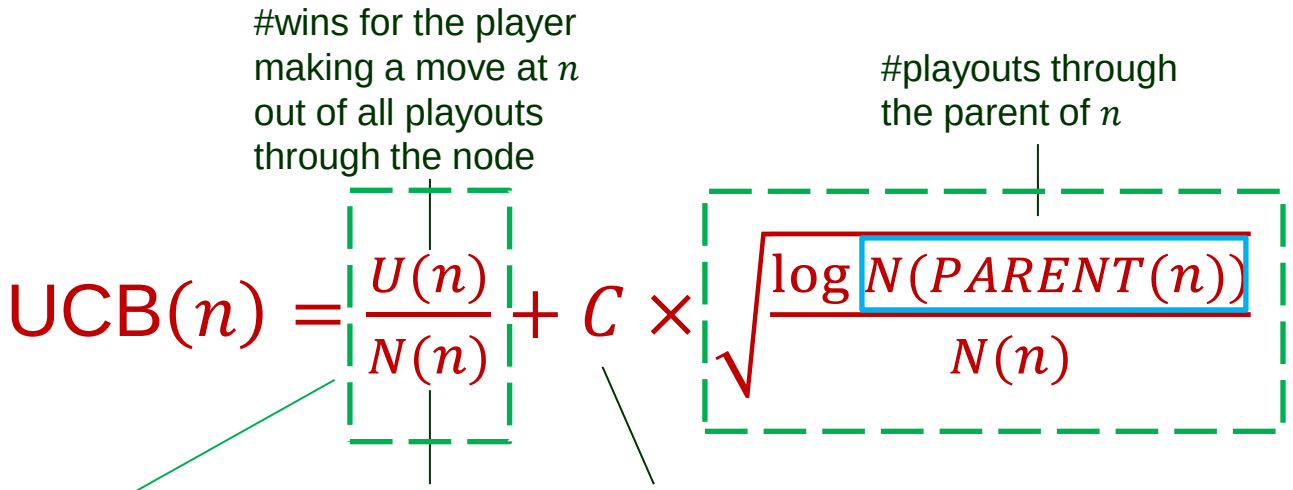
Exploitation term: average utility of n

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

Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

Upper confidence bound formula:



Exploitation term:
average utility of n

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

Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

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

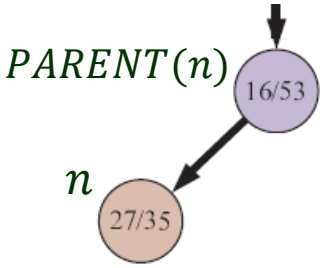
$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

#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

At a node n in the expanding MCTS tree rooted at the real *state*.

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

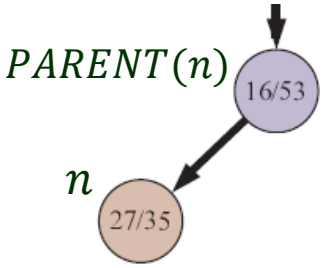
$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

#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.



$$U(n) = 27$$
$$N(n) = 35$$
$$N(PARENT(n)) = 53$$
$$C = \sqrt{2}$$

(choice by a theoretical argument)

Ranking of Possible Moves

At a node n in the expanding MCTS tree rooted at the real *state*.

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

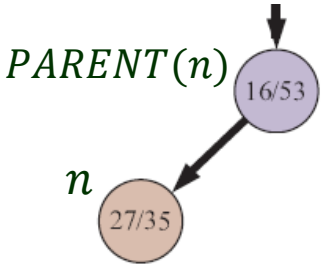
$$\text{UCB}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$$

#playouts through the node

balance between exploitation and exploration

Exploitation term: average utility of n

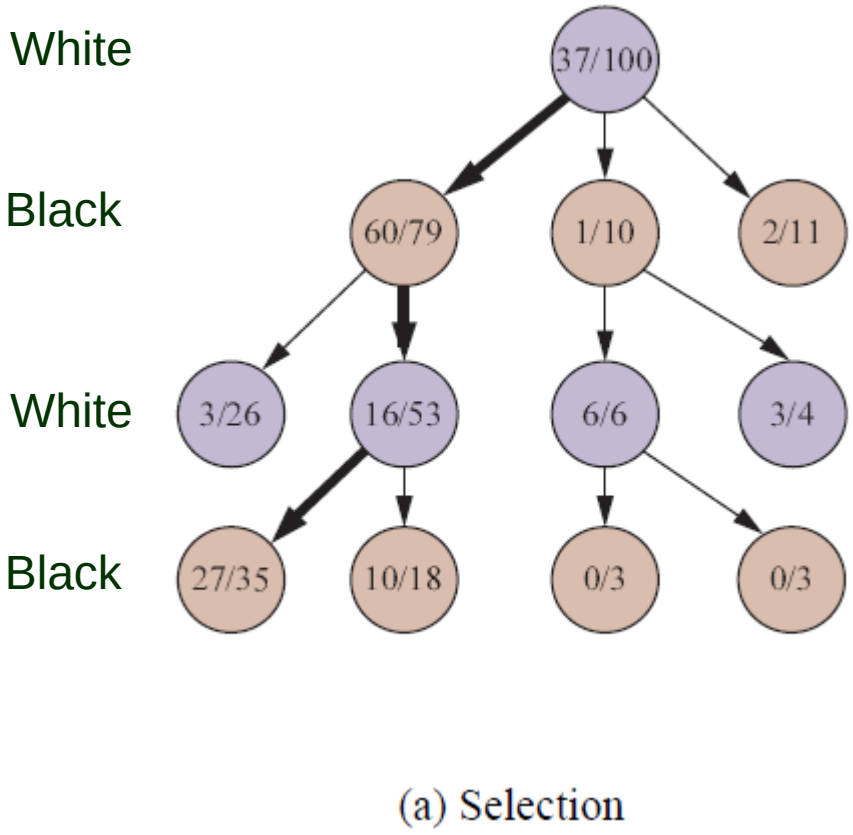
Exploration term: high value if n has been explored a few times.



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

$$\text{UCB}(n) = \frac{27}{35} + \sqrt{2} \cdot \sqrt{\frac{\log 53}{35}}$$

Constant C



(a) Selection

- Balances exploitation and exploration.
- Multiple values are tried and the one that performs the best is chosen.
  - $C = 1.4$

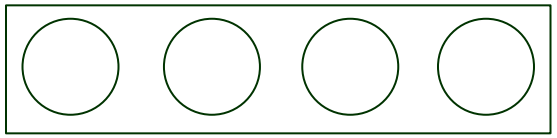
    The 60/79 node has the highest score.
  - $C = 1.5$

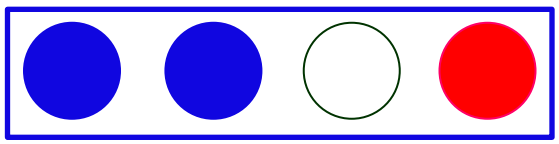
    The 2/11 node has the highest score.

Example: The Pair Game

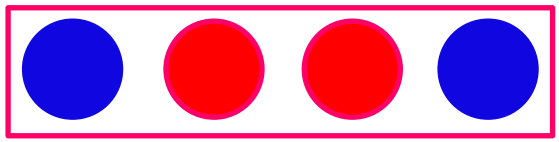
From *Monte Carlo Tree Search: A Guide*

- Four circular holes in the board.
- Two players (Blue and Red) take turns to place disks in their colors to cover the holes.
- Blue goes first.
- Whoever first covers two adjacent holes wins the game.

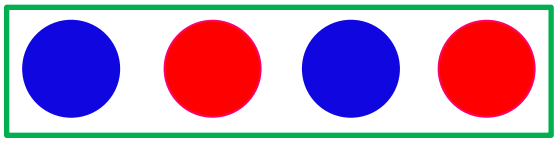




Blue wins

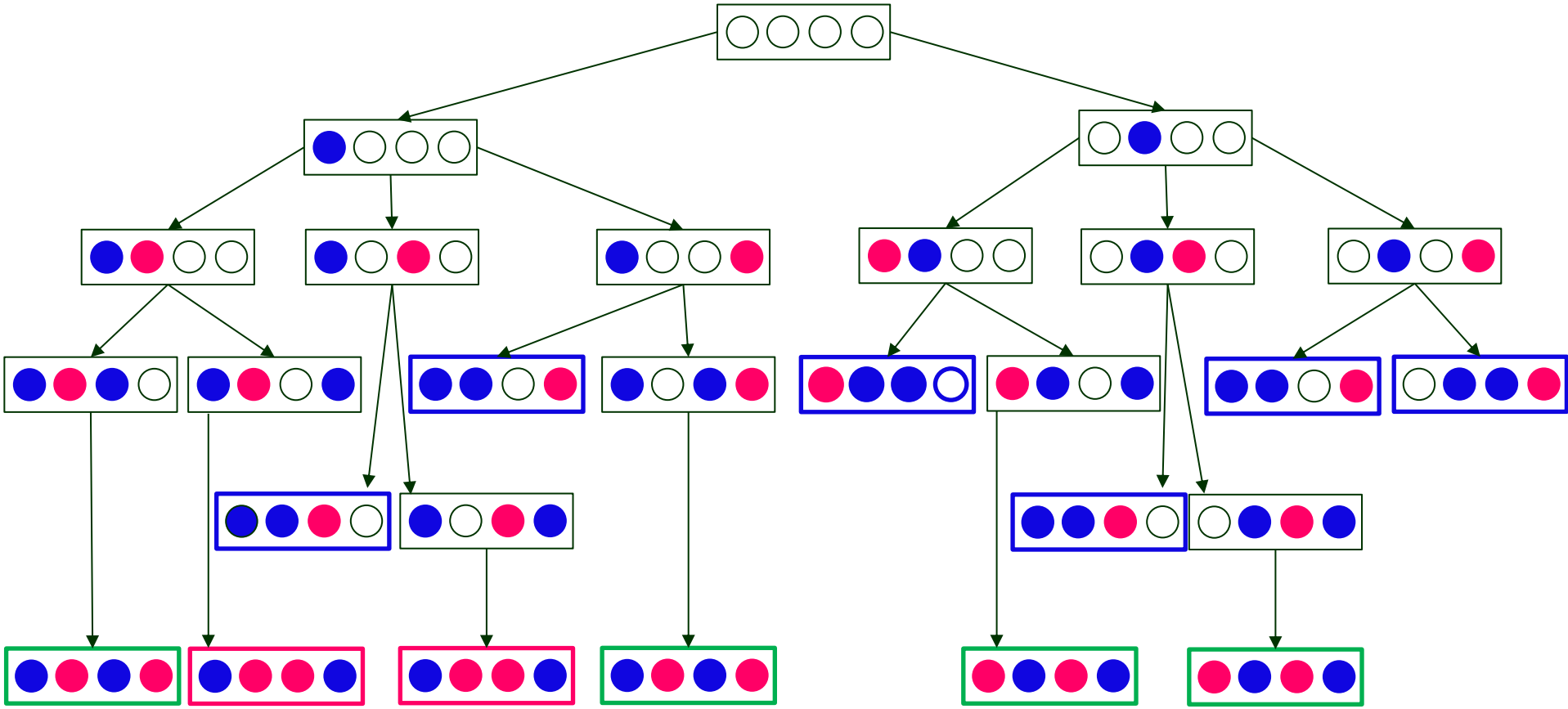


Red wins

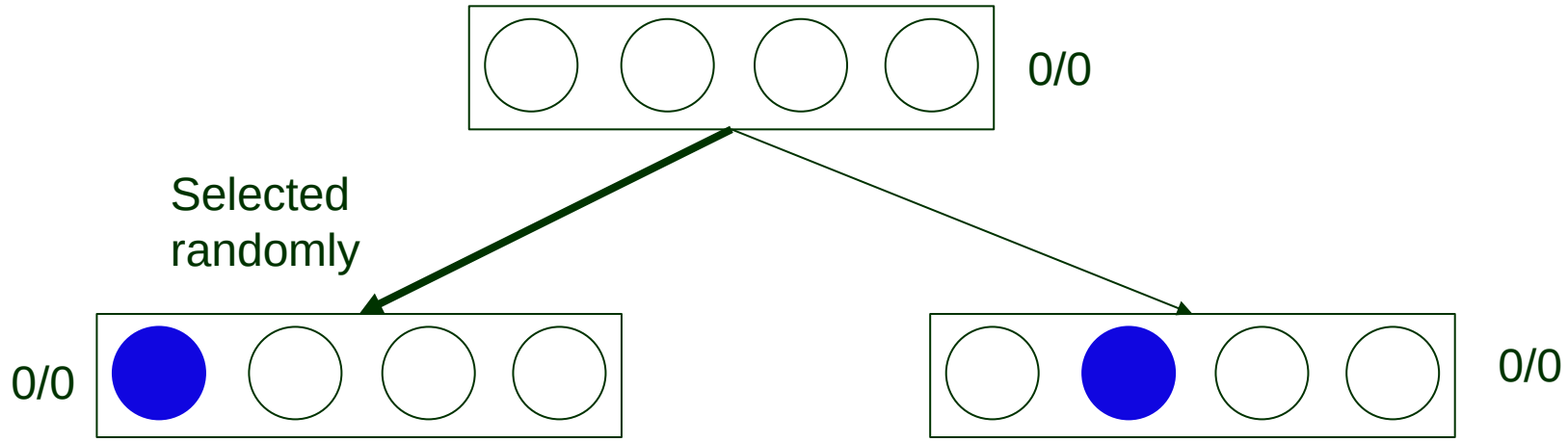


Tie

Complete Game Tree (up to Symmetry)



Iteration 1: Selection & Expansion



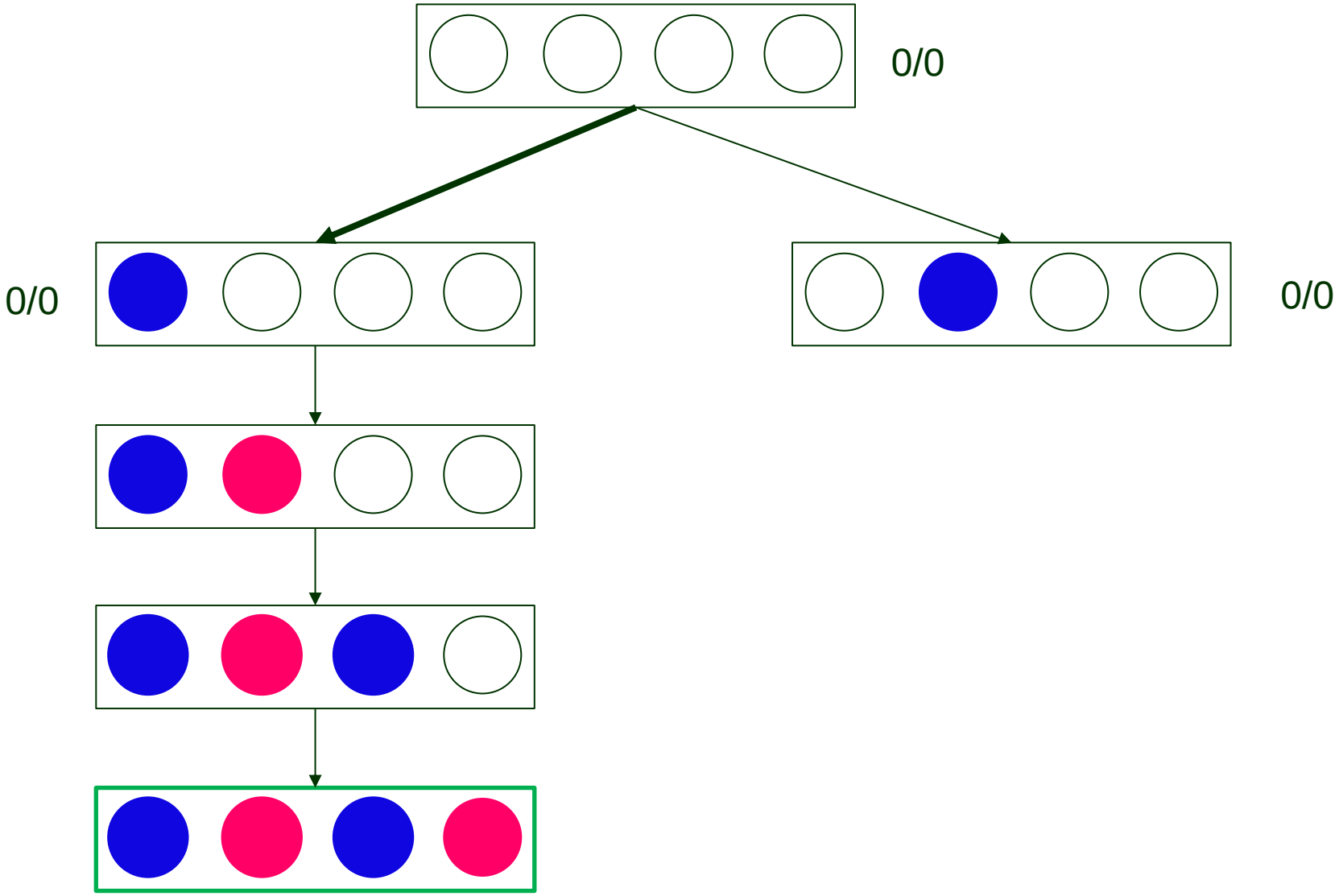
Leaf nodes

UCB ← ∞

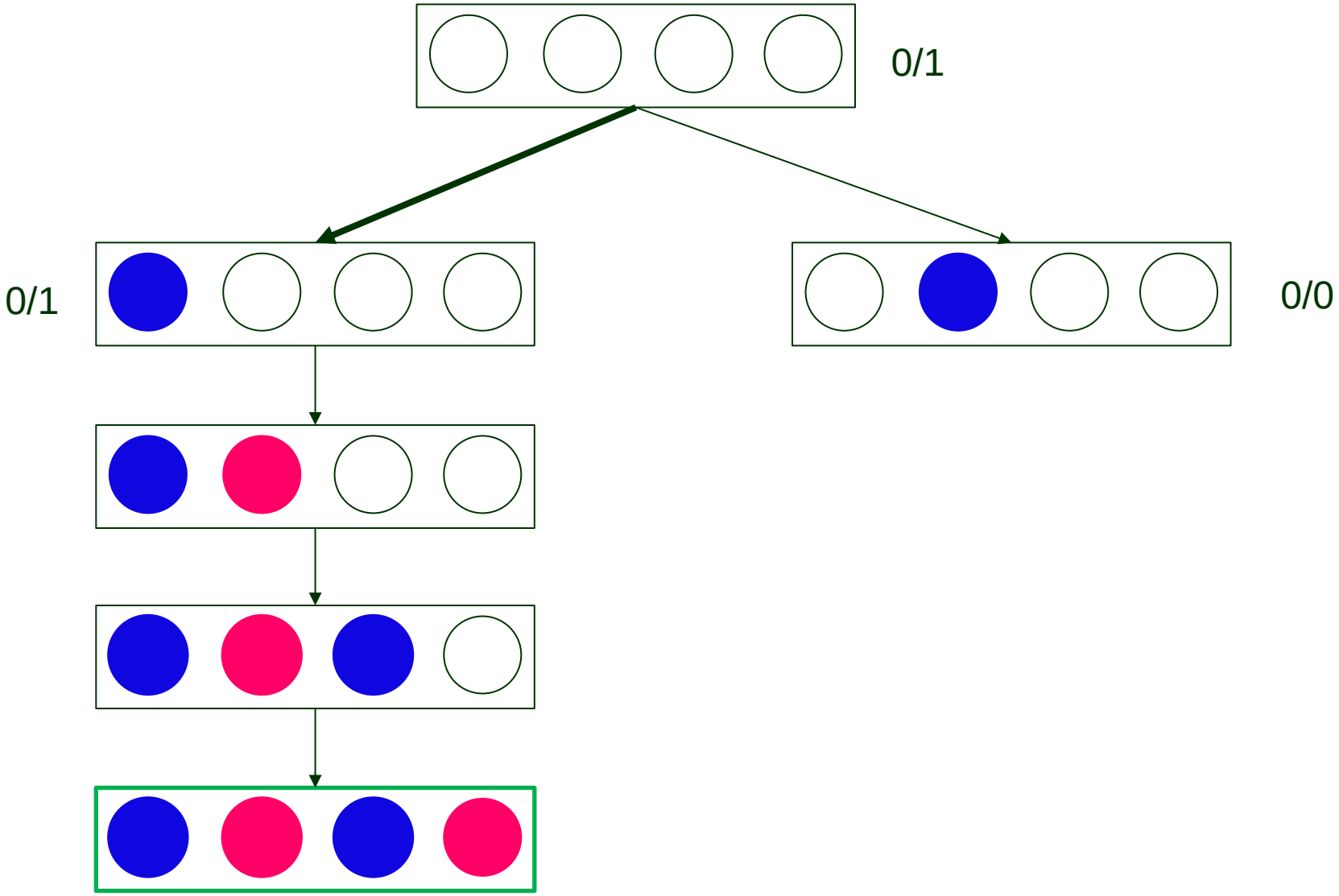
UCB ← ∞

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

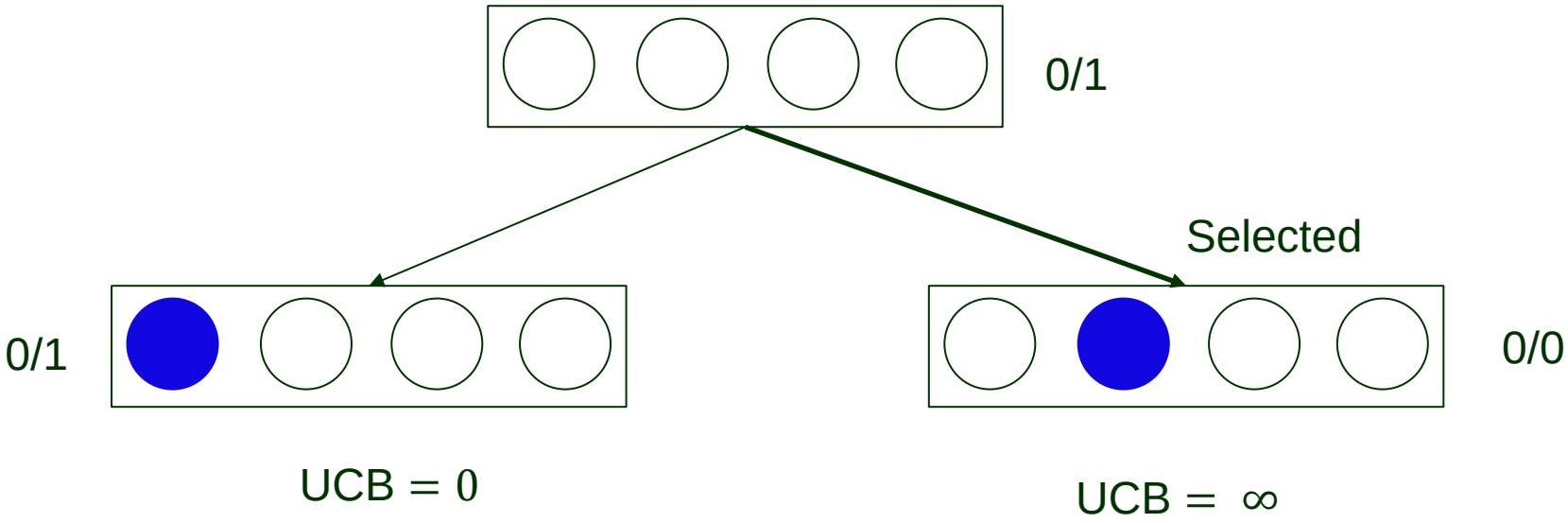
Simulation



Backpropagation



Iteration 2: Selection



$$U(n) = 0$$
$$N(n) = 1$$

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

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 plies 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 plies 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 plies 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 plies.

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 plies 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 plies.

  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 plies 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 plies.

  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 plies 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 plies.

  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 plies 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 plies.

  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 plies 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 plies.

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.