Artificial Intelligence: Simulation-Based Search

Stephan Schiffel

stephans@ru.is
Reykjavík University, Iceland

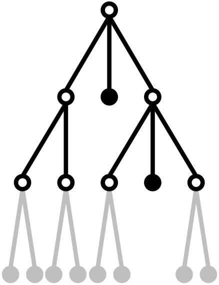
Outline

So far ...

- Complete search for single player games:
  BFS, DFS, ...
- Complete search for multi-player games:
  Minimax, $\alpha - \beta$-Pruning
- **Problem:**
  What if the game is too large to search completely?

Game-Tree Search

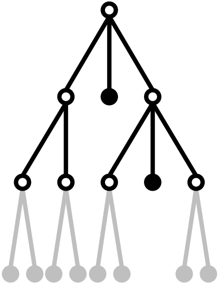
heuristic search



We need:

Game-Tree Search

heuristic search



We need:

state evaluation function / knowledge

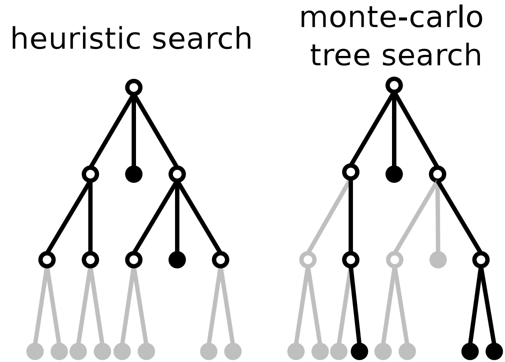
So far ...

- Complete search for single player games:
  BFS, DFS, ...
- Complete search for multi-player games:
  Minimax, $\alpha - \beta$-Pruning
- **Problem:**
  What if the game is too large to search completely?

So far ...

- Complete search for single player games:
  BFS, DFS, ...
- Complete search for multi-player games:
  Minimax, $\alpha - \beta$-Pruning
- **Problem:**
  What if the game is too large to search completely?
- **Solution:**
  Heuristic $=$ evaluation function for non-terminal states
- **New problems:**
  How to come up with a good heuristic?

Game-Tree vs. Simulation Search



We need:

state evaluation function / knowledge

only the game rules

Monte-Carlo Search

- Simple Heuristics:
  Evaluation of a node is the average reward of random play starting in this node.
- Prerequisite:
  Being able to simulate the game.
- No game specific knowledge needed!

Monte-Carlo Search - Algorithm

mc_search(role r, state s)
(returns the "best" move for role r in state s)

- $Q(a) := 0$ for all a
- $N(a) := 0$ for all a
- while there is time left
  - randomly select a move a from the legal moves of r in s
  - $s' := update(a, s)$
  - $score := run\_simulation(r, s')$
  - $N(a) := N(a) + 1$
  - $Q(a) := Q(a) + \frac{score - Q(a)}{N(a)}$
- return $argmax_a Q(a)$

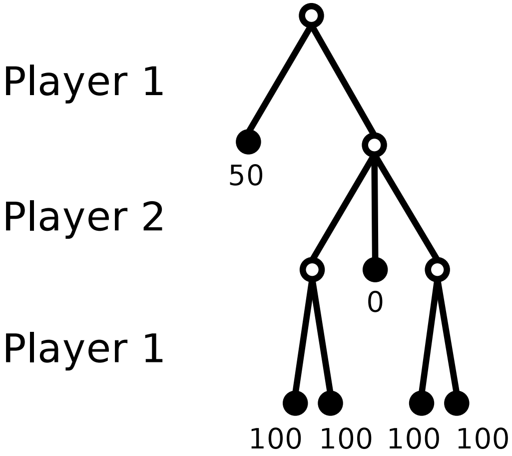
Monte-Carlo Search - Algorithm

run_simulation(role r, state s)
(returns the score for role r if the game is in state s and randomly played to the end)

- if $terminal(s)$ then
  - return $reward(r, s)$
- else
  - randomly select a move a from the legal moves in s
  - $s' := update(a, s)$
  - return $run\_simulation(r, s')$

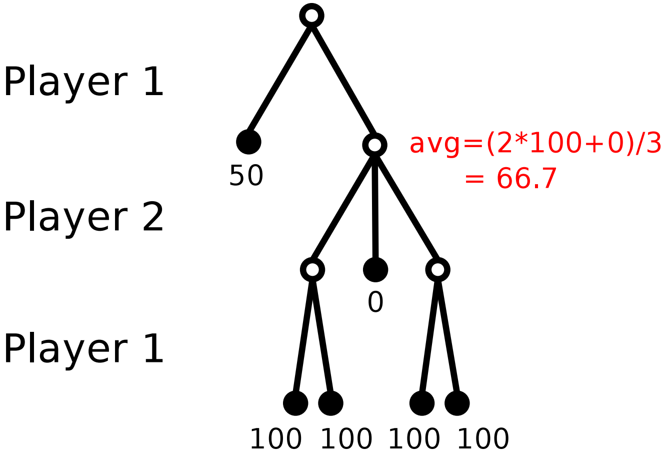
Monte-Carlo Search - Problems (1)

Too optimistic:



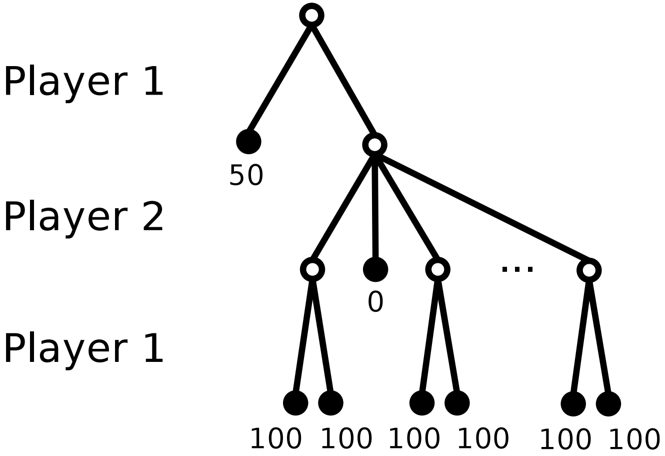
Monte-Carlo Search - Problems (1)

Too optimistic:



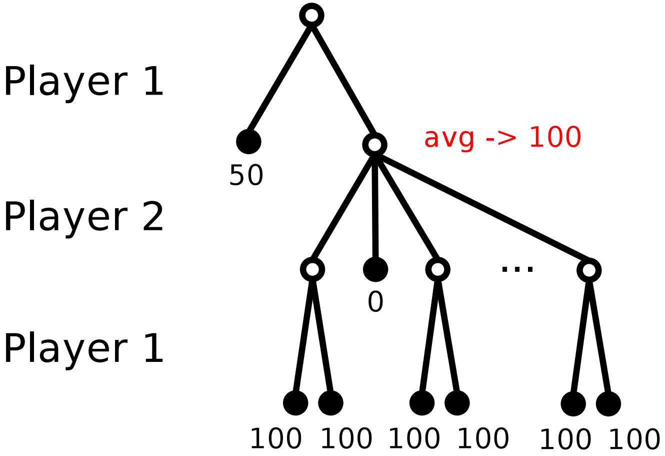
Monte-Carlo Search - Problems (1)

Too optimistic:



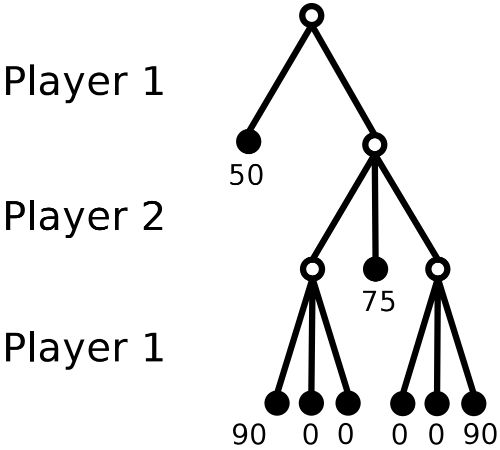
Monte-Carlo Search - Problems (1)

Too optimistic:



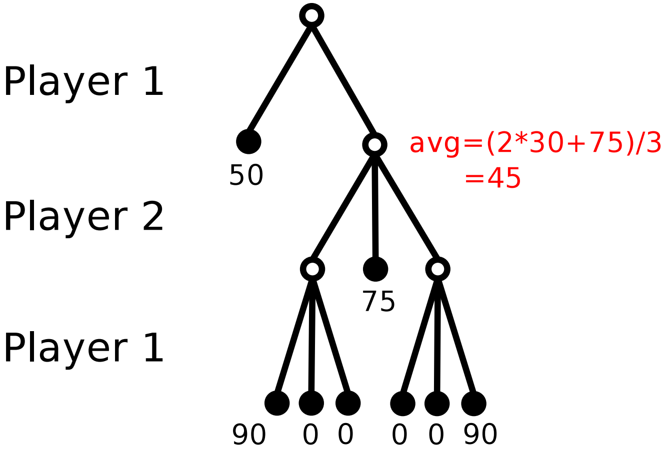
Monte-Carlo Search - Problems (2)

Too pessimistic:



Monte-Carlo Search - Problems (2)

Too pessimistic:



Monte-Carlo Search - Pros and Cons

Advantages:

- Easy to implement
- Low memory requirements
- No game specific knowledge needed

Disadvantages:

- Does not terminate
- Wrong assumption: Everyone (including opponents) plays random moves.
- Does not produce correct results (Minimax always computes the game theoretic best move!)
- All information is lost in the next step

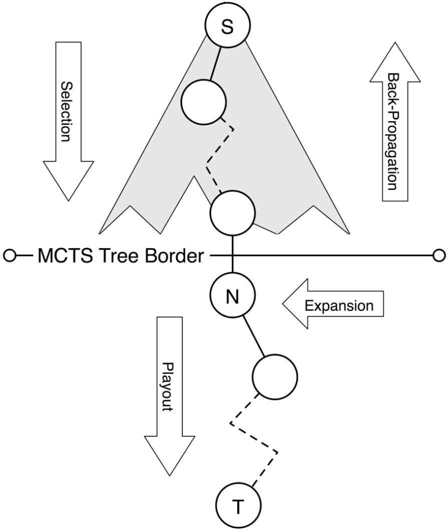
Monte-Carlo Tree Search

- Expand more than one level of the tree
- Keep track of average score in every node of the tree
- Advantages over pure Monte-Carlo Search:
    - Subtree can be used for next step
    - Node expansion in tree is faster (no need to compute legal moves and state update)
- How much to expand?
  In practice often: Expand one node per simulation!

Monte-Carlo Tree Search with UCT

- UCT="Upper Confidence Bounds applied to Trees"
- Idea: Use values in the tree to guide exploration
- For each state s in the tree keep:
  - $Q(s, a)$ .. the average score of action a for the current player in s
  - $N(s, a)$ .. the number of simulations run with action a
  - $N(s)$ .. the number of simulations run from state s
- Phases:
  1. Selection: Select a leaf node of the tree
  2. Expansion: Expand the node
  3. Playout: Run a random simulation of the game
  4. Back-Propagation: Update the values of the nodes in the tree

A Single Simulation in MCTS/UCT



UCT - Selection

- Start with the root of the tree (s =current state)
- While s is in the tree:
  - Select the action a with the highest UCT value:

$$a = argmax_{a \in legals(s)} \left\{ Q(s,a) + C * \sqrt{\frac{ln(N(s))}{N(s,a)}} \right\}$$

    C used to control exploration vs. exploitation
  - $s := update(a, s)$
- $expand(s)$ .. add all direct successors of s to the tree
- $playout(s)$ .. run a random simulation starting in s

UCT - Back-Propagation

- Update values as before, but now for every state s on the path in the tree
- Number of simulation with action:

$$N(s, a) := N(s, a) + 1$$

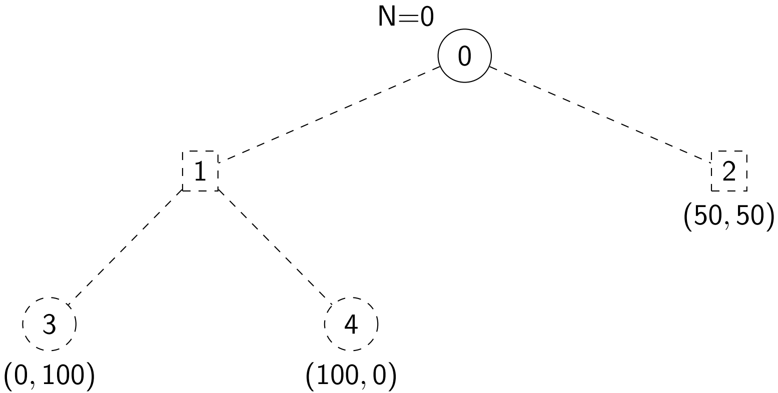
- Average score of action (if it is r's turn in s):

$$Q(s, a) := Q(s, a) + \frac{score[r] - Q(s, a)}{N(s, a)}$$

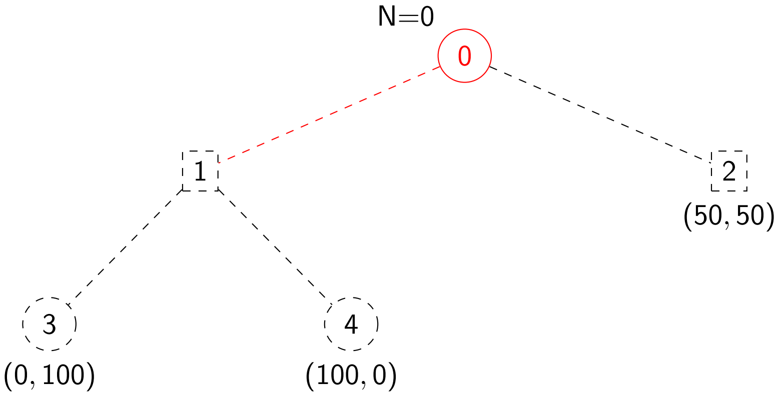
- Number of simulation with state:

$$N(s) := N(s) + 1$$

UCT - Example

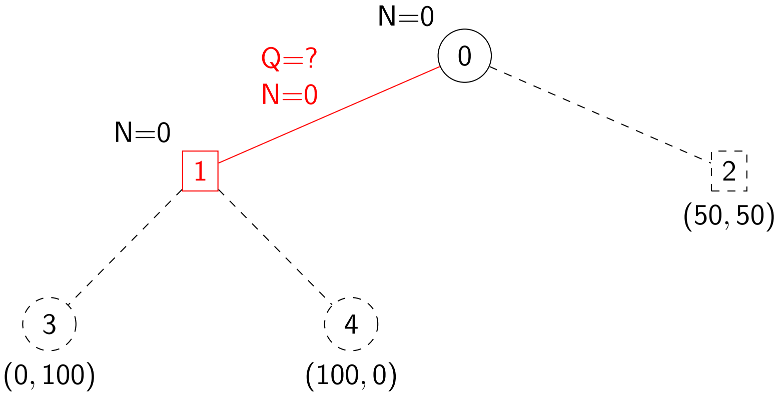


UCT - Example



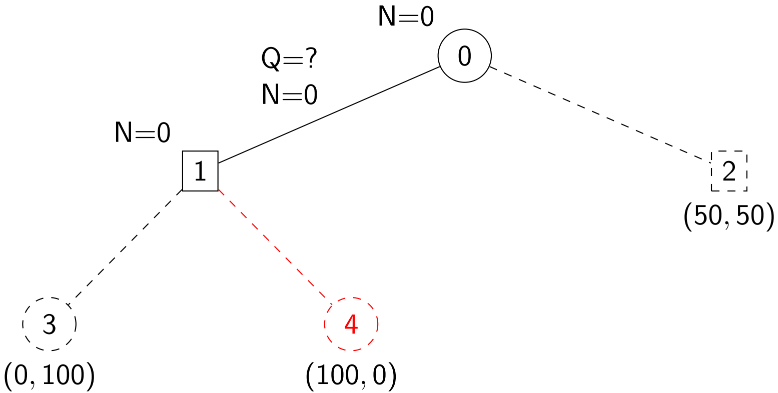
1. Iteration - Selection: select the first unexplored child of node 0

UCT - Example



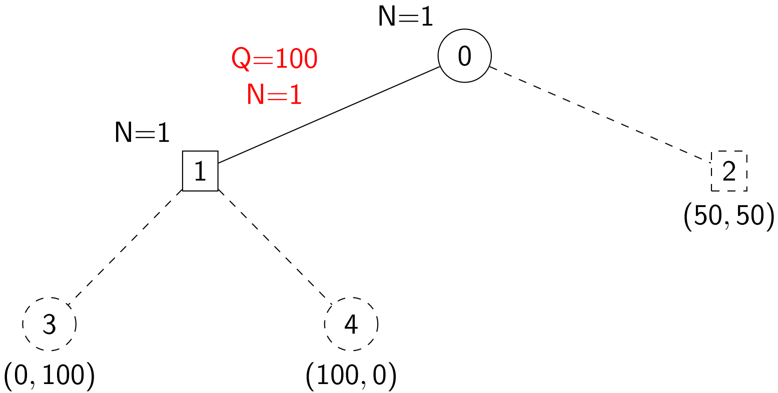
1. **Iteration - Expansion:** add node 1 to the tree

UCT - Example



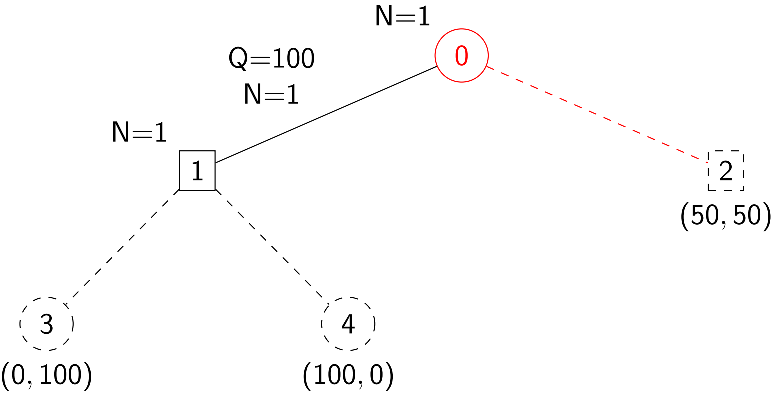
1. Iteration - Playout: play randomly to a terminal state, for example, node 4

UCT - Example



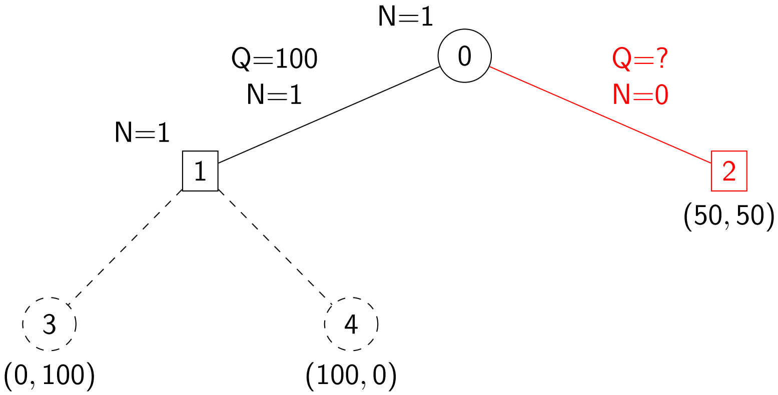
1. Iteration - Backpropagation: score[player1]=100, gets applied to Q(0,left)

UCT - Example



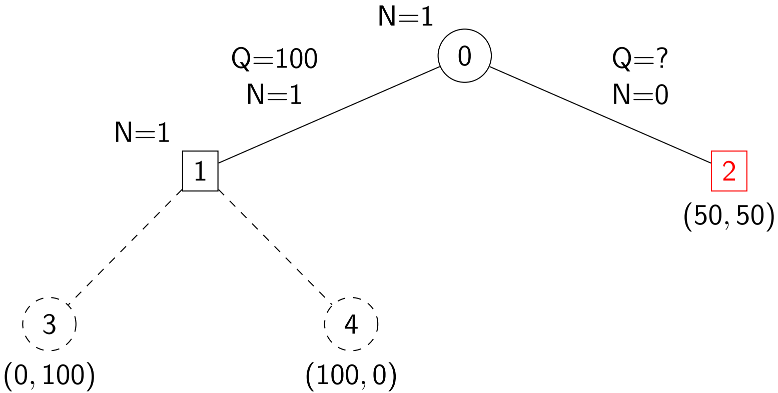
2. Iteration - Selection: select the first unexplored child of node 0 (node 2)

UCT - Example



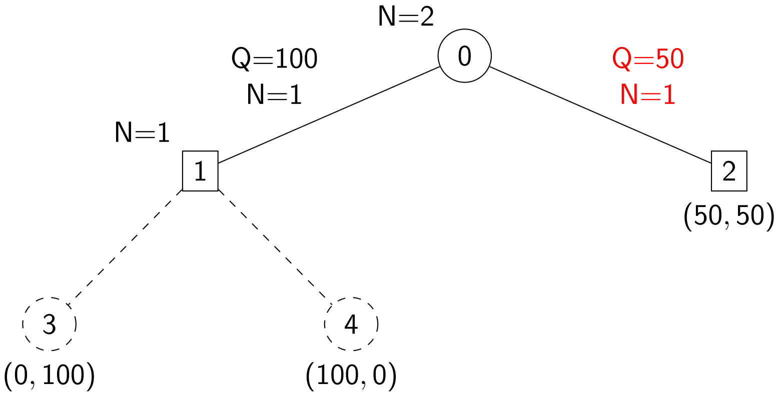
2. Iteration - Expansion: add node 2 to the tree

UCT - Example



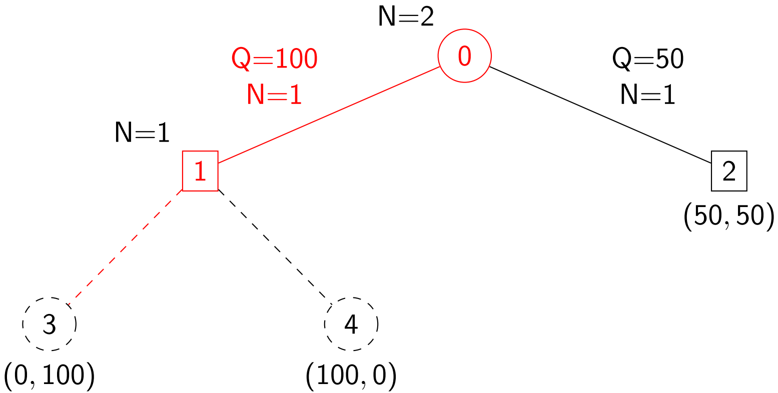
2. Iteration - Playout: node 2 is terminal, no more moves to play

UCT - Example



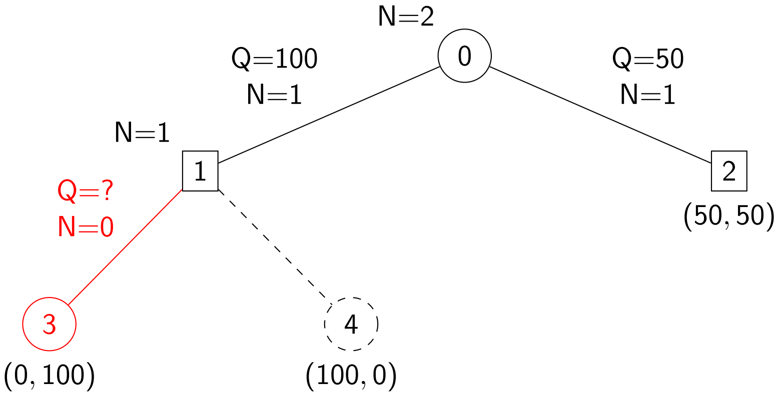
2. Iteration - Backpropagation: score[player1]=50, gets applied to Q(0,right)

UCT - Example



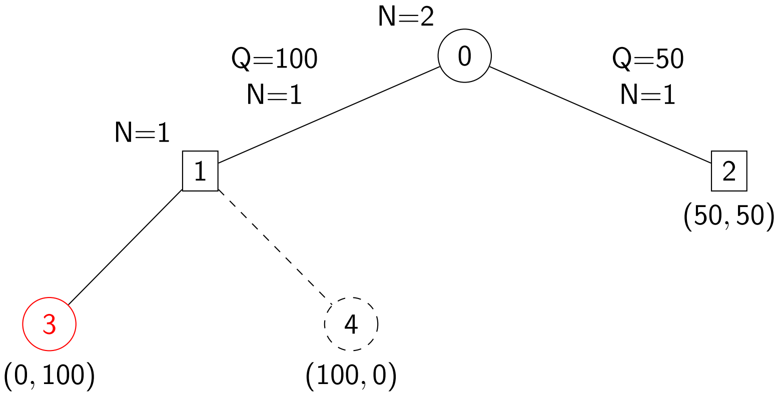
3. Iteration - Selection: There are no explored children of node 0, thus select child of node 0 with highest UCT value (node 1). Then select first unexplored child of node 1 (node 3).

UCT - Example



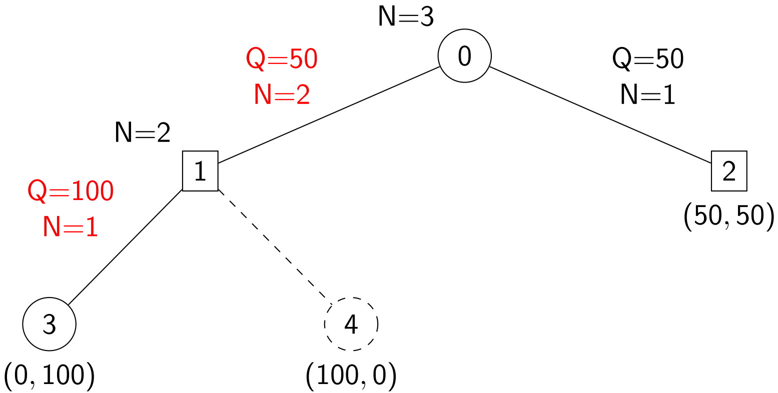
3. Iteration - Expansion: add node 3 to the tree

UCT - Example



3. Iteration - Playout: node 3 is terminal, no more moves to play

UCT - Example



3. Iteration - Backpropagation:
score[player1]=0 gets applied to Q(0,left)
score[player2]=100 gets applied to Q(1,left)

MCTS/UCT - Pros and Cons

Advantages:

- Converges to game-theoretic value (in turn-taking games, if the whole tree gets expanded)
- Not too optimistic/pessimistic about moves in the tree
- Still relatively easy to implement
- Still no game specific knowledge needed
- Successful in practice (General Game Playing, Go, ...)

Disadvantages:

- May need long to converge even if tree is fully expanded
- Unusable for single-player games, unless they have gradual goal values (not just win or loss)
- Still random (=unrealistic) simulations

Heuristics Again

Problem:

- Random simulations are unrealistic
  $\Rightarrow$ slow convergence to good values
  $\Rightarrow$ too optimistic/pessimistic in some situations

Solutions:

- Use heuristics to guide the selection if only few simulations have been run
- Use heuristics to guide the playouts (select good moves with higher probability)

Summary

- MCTS/UCT is an alternative to Minimax
- works without heuristics, but can use heuristics to improve performance