Monte Carlo Tree Search

2/13/17

General Game Playing

Suppose you wanted to write an AI that could play any game. How could you do that?

First idea: backward induction

- Plays optimally once the full tree is explored.
- Exploring the whole tree may not be feasible.

Second idea: generalized heuristics

- When the tree is too big, we use depth-bounded search.
- `basicEval` and `betterEval` are connect-four specific.
- How can we make a heuristic that works for any game?

Monte Carlo Simulation Heuristic

Key idea: play out a bunch of random games.

To evaluate a state:

- Play random moves until the end of the game.
- Record the result.
- Repeat a bunch of times.
- Return the average result over the random games.

Monte Carlo Heuristic Pseudocode

```
function MC_BoardEval(state):
    wins = losses = draws = 0
    for i=1:NUM_SAMPLES
        next_state = state
        while non_terminal(next_state):
            next_state = random_move(next_state)
        increment wins/losses/draws
  return (2*wins + draws) / NUM_SAMPLES
```

Here, I'm using the following utility function:

- win: 2pts, draw: 1pt, loss: 0pts
- Many other other utility functions are possible.

Monte Carlo Board Evaluation

Advantages

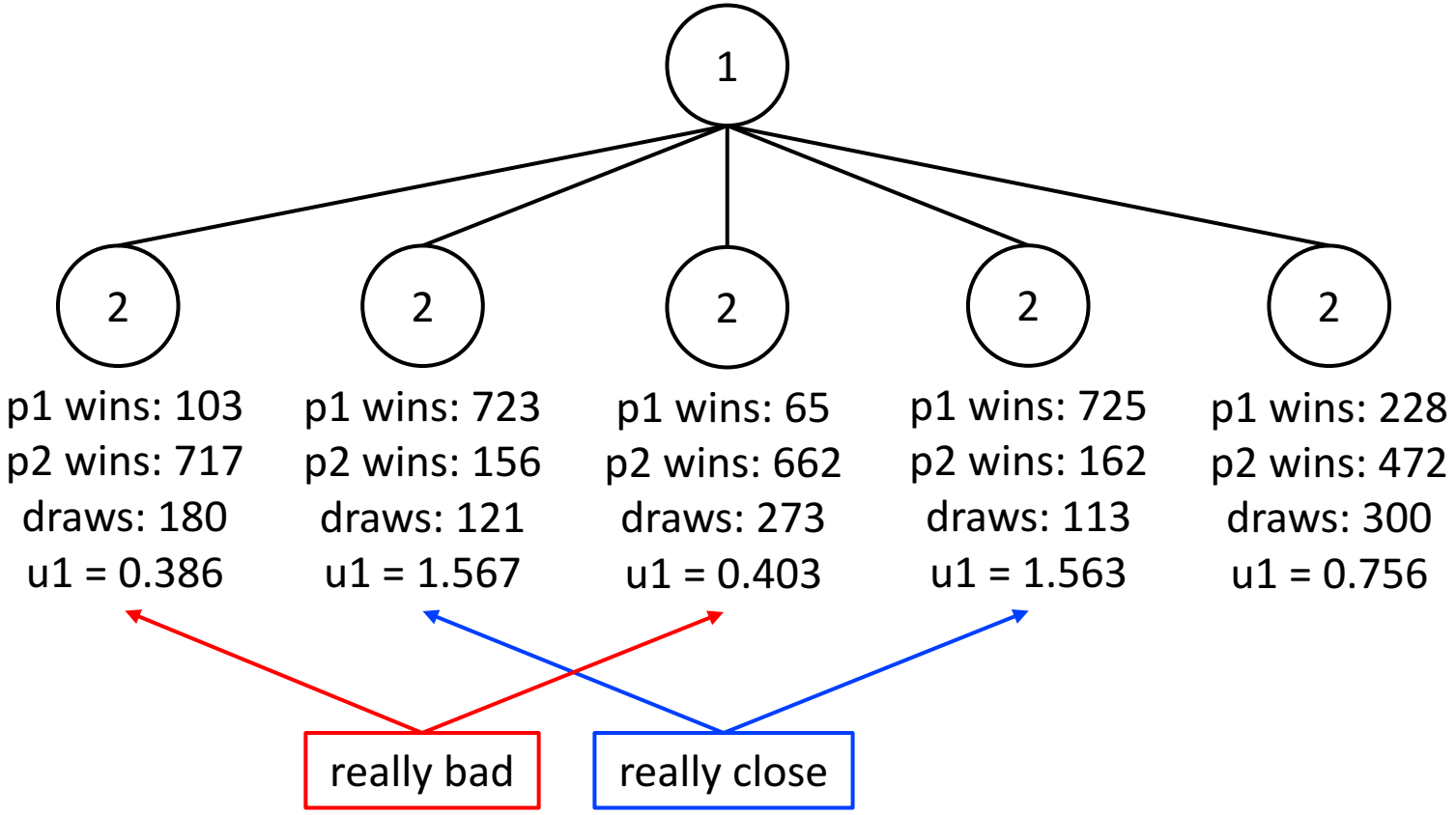
- simple
- domain independent
- anytime algorithm

Disadvantages

- slow
- non-static
- nondeterministic

Improving MC_BoardEval

Consider one level up. Suppose we're doing min/max search with a depth limit of 4 and using `MC_BoardEval` as our heuristic. What's happening at depth 3?



Improving MC_BoardEval

- We'd like to do more simulations from the more-important states (those more likely to be picked).
- We're willing to do fewer simulations from the less important (low value) states.

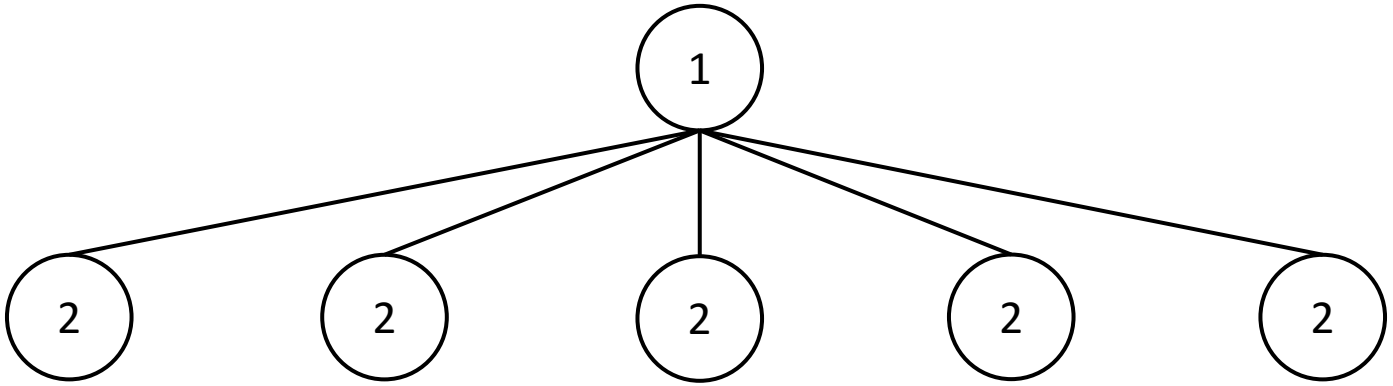
Key idea: explore/exploit tradeoff

- Explore: try something new
- Exploit: try something you know is good

Multi-armed bandit problem

It turns out we've stumbled upon a widely-studied problem.

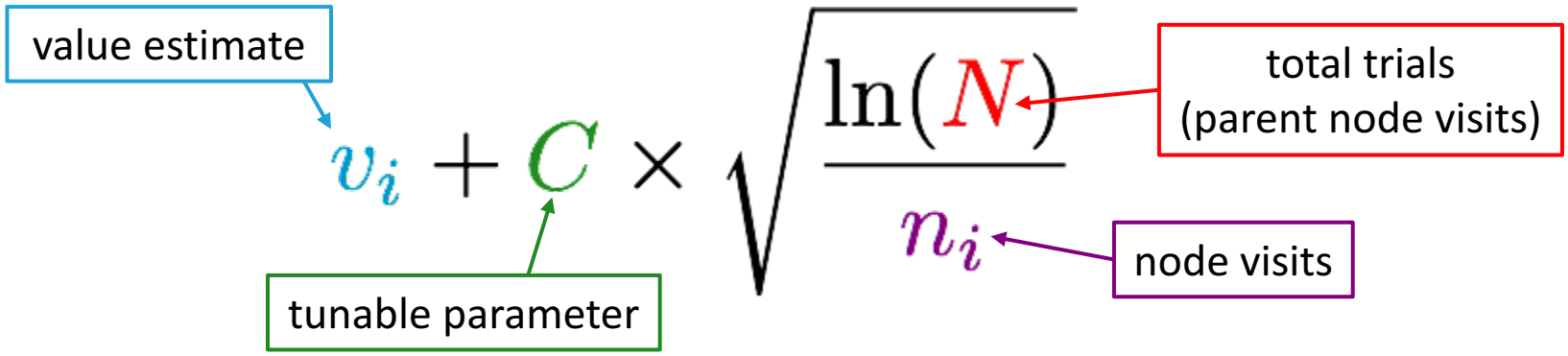
Given a row of slot machines (bandits), with different, unknown, probabilities of winning a jackpot, use a fixed number of quarters to win as many jackpots as possible.



Upper confidence bound (UCB)

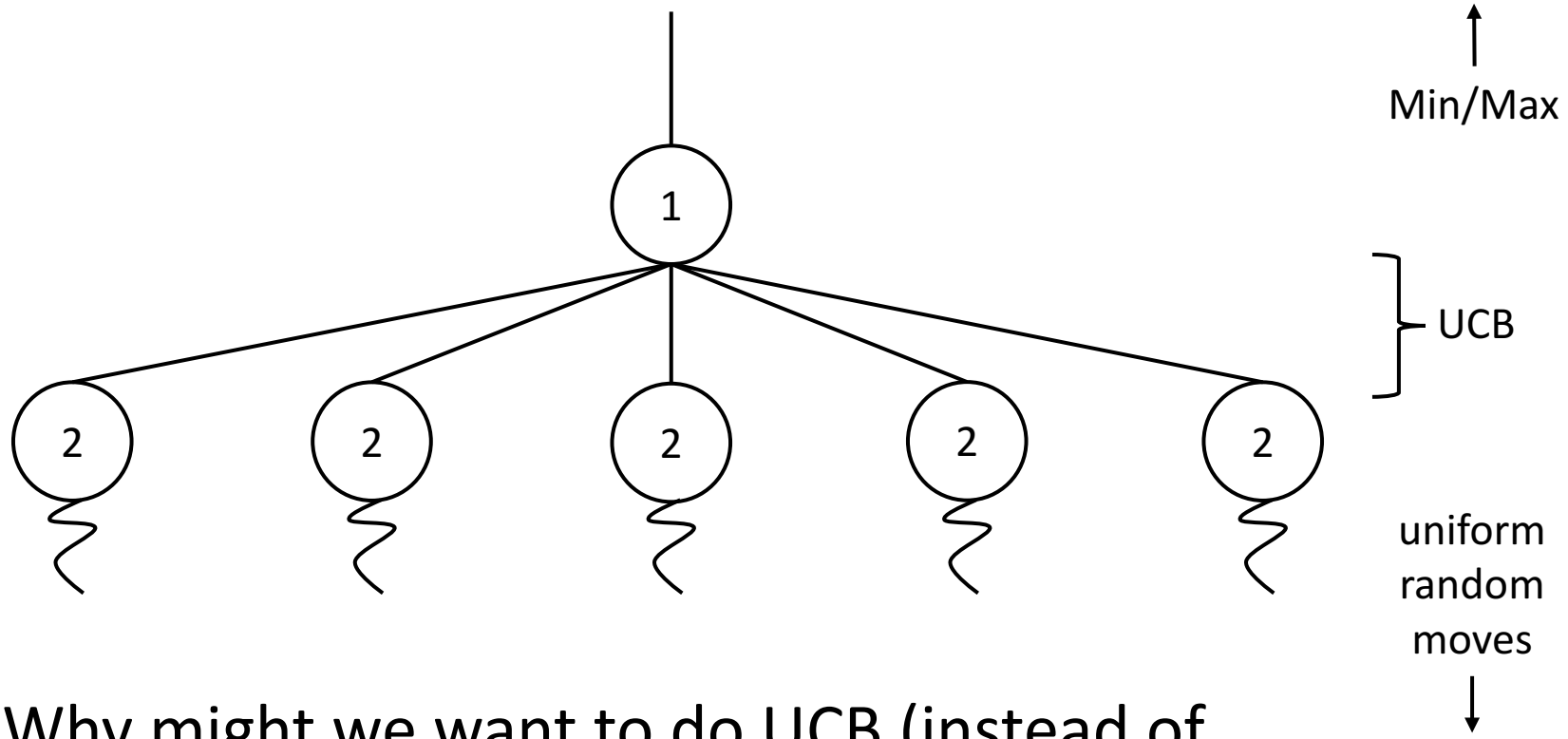
Provably optimal solution to multi-armed bandit.

Pick nodes with probability proportional to:



- probability decreases in number of visits (explore)
- probability increases in a node's value (exploit)
- always tries every option once

Why do UCB at only one level?



1. Why might we want to do UCB (instead of Min/Max) at shallower levels?
2. Why might we want to do UCB (instead of uniform random moves) at deeper levels?

Doing UCB at more levels

Extend to deeper levels?

+ more value out of every random playout

– more information to keep track of

How can we alleviate this?

Extend to shallower levels?

+ guide the search to explore better paths first

– lose optimality of minimax

Is this a big deal?

– never completely prune branches

Is this a big deal?

The Monte Carlo Tree Search Algorithm

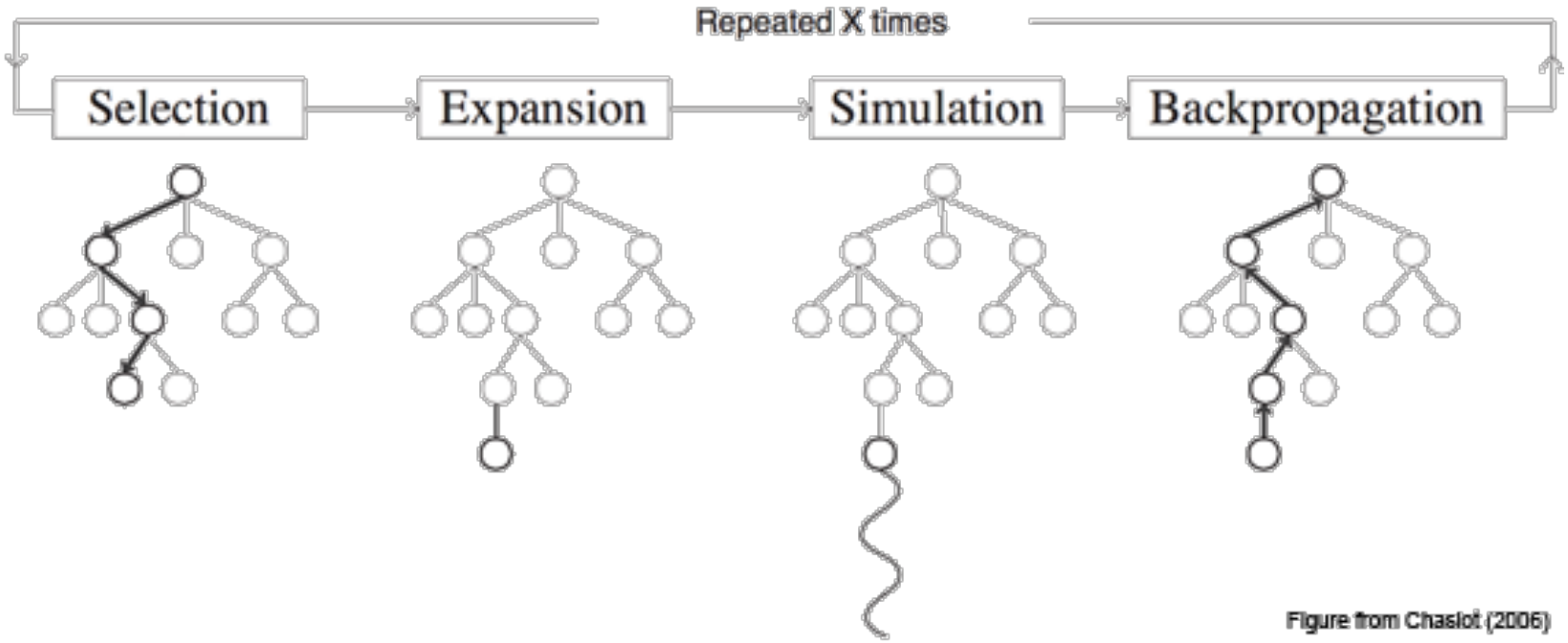


Figure from Chaslot (2006)

Selection

- Used for nodes we've seen before.
- Pick according to UCB.

Expansion

- Used when we reach the frontier.
- Add one node per playout.

Simulation

- Used beyond the search frontier.
- Don’t bother with UCB, just play randomly.

Backpropagation

- After reaching a terminal node.
- Update value and visits for all states visited in selection and expansion phases.

MCTS Pseudocode

```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

MCTS Helper Functions

```
function UCB_sample(node):
    weights = []
    for child of node:
        w = child.value
        w += C*sqrt(ln(node.visits) / child.visits)
        add w to weights
    distribution = normalize weights to sum to 1
    return child sampled according to distribution
```

MCTS Helper Functions

```
function random_playout(state):
    while state is not terminal:
        state = make a random move from state
    return outcome


function update_value(node, outcome):
    #combine the new outcome with the average value
    node.value *= node.visits
    node.visits++
    node.value += outcome
    node.value /= node.visits
```