CS 2

Introduction to Programming Methods

Monte-Carlo Tree Search

A **branching factor** is how many times a node splits at each level. In Tic-Tac-Toe, for a random position, the average branching factor is:

4

The average Tic-Tac-Toe game lasts about

9 Moves

A **branching factor** is how many times a node splits at each level. In Othello, for a random position, the average branching factor is:

10

The average Othello game lasts about

58 Moves

A **branching factor** is how many times a node splits at each level. In Chess, for a random position, the average branching factor is:

35

The average Chess game lasts about

70 Moves

A **branching factor** is how many times a node splits at each level. In Go, for a random position, the average branching factor is:

250

The average Go game lasts about

150 Moves

A **branching factor** is how many times a node splits at each level. In Go, for a random position, the average branching factor is:

250

The average Go game lasts about

150 Moves

Somewhere between Chess and Go, Alphabeta becomes completely useless...

Say Goodbye to Alphabeta

- Alphabeta can't deal with large branching factors.

Say Goodbye to Alphabeta

- Alphabeta can't deal with large branching factors.

- Alphabeta requires a ton of domain knowledge to write the evaluation function.

Say Goodbye to Alphabeta

- Alphabeta can't deal with large branching factors.
- Alphabeta requires a ton of domain knowledge to write the evaluation function.
- Alphabeta must reach the top of the tree to get **any** useful answer.

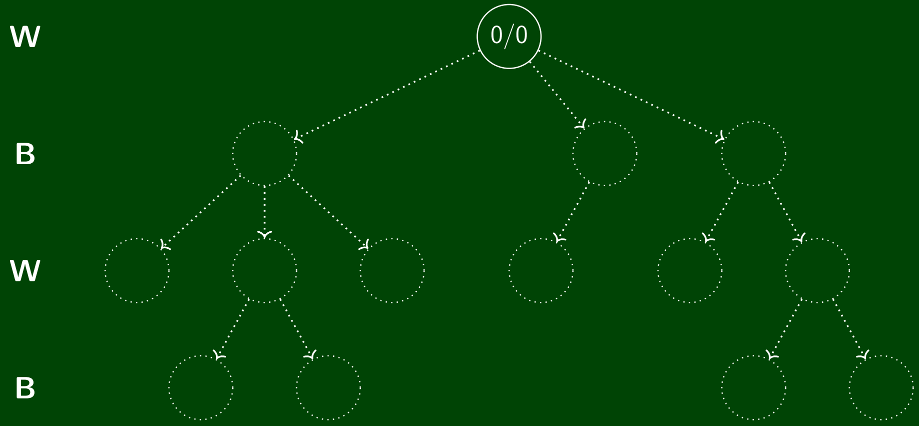
The fundamental problem with Alphabeta is that even pruning where we can, the state space is still too large. So, **we can't explore all of it**. Then, how do we choose which part to explore?

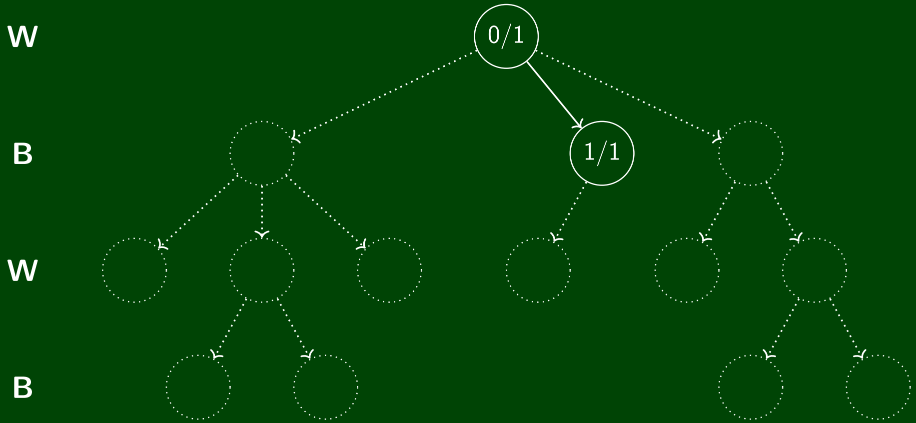
The fundamental problem with Alphabeta is that even pruning where we can, the state space is still too large. So, **we can't explore all of it**. Then, how do we choose which part to explore?

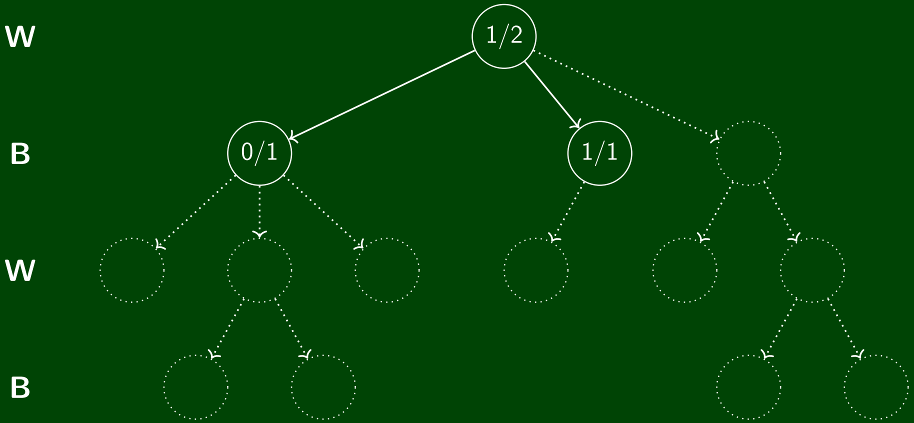
Like many problems in CS, an answer is "throw randomness or ML at the problem". In this case, it's both.

Apply **randomness** to solve a deterministic problem.

In the case of games, randomly play a bunch of games and keep track of who wins via each move.


W
B
W
B
0/0


W
B
W
B
0/1
1/1


W
B
W
B
1/2
0/1
1/1

Don't choose randomly!
Instead, compute an upper confidence bound on each estimation and choose the max node!

$$\mathrm{UCB}(i) = \frac{w_i}{n_i} + C\sqrt{\frac{\log(n_{\mathsf{parent}(i)})}{n_i}}$$

The idea is to balance exploring new nodes with using known good nodes.

The Algorithm

1. Selection: Select nodes recursively using the UCB formula until we hit a node without data for all of its children.
2. Expansion: If the selected node doesn't end the game, create a new node by choosing a move randomly.
3. Simulation: Run a simulated playout until the game is over.
4. Backpropagation: Update all the nodes we explored with the simulation result.

How do we run the playouts?

Playouts

How do we run the playouts?

- Randomly

Playouts

How do we run the playouts?

- Randomly
- Deep Learning

How do we run the playouts?

- Randomly
- Deep Learning
- ...

More Details?

- https://www.cs.swarthmore.edu/~mitchell/classes/cs63/f20/reading/mcts.html
- http://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/