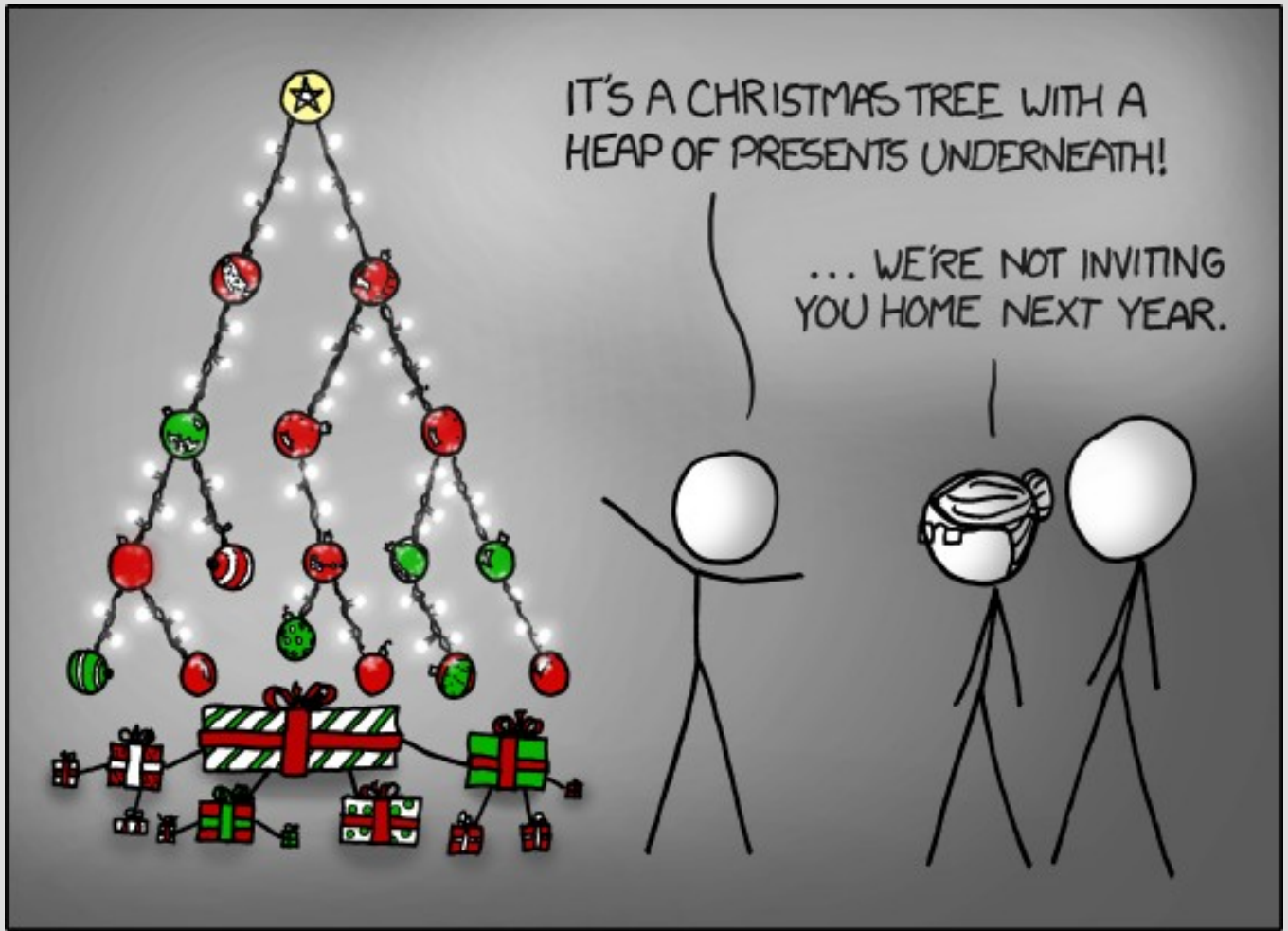
More on games (Ch. 5.4-5.7)



Mid-state evaluation

So far we assumed that you have to reach a terminal state then propagate backwards (with possibly pruning)

More complex games (Go or Chess) it is hard to reach the terminal states as they are so far down the tree (and large branching factor)

Instead, we will estimate the value minimax would give without going all the way down

Mid-state evaluation

By using mid-state evaluations (not terminal) the “best” action can be found quickly

These mid-state evaluations need to be:

1. Based on current state only
2. Fast (and not just a recursive search)
3. Accurate (represents correct win/loss rate)

The quality of your final solution is highly correlated to the quality of your evaluation

Mid-state evaluation

For searches, the heuristic only helps you find the goal faster (but A* will find the best solution as long as the heuristic is admissible)

There is no concept of “admissible” mid-state evaluations... and there is almost no guarantee that you will find the best/optimal solution

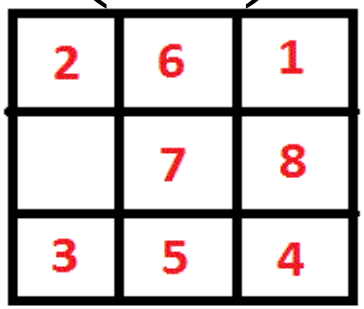
For this reason we only apply mid-state evals to problems that we cannot solve optimally

Mid-state evaluation

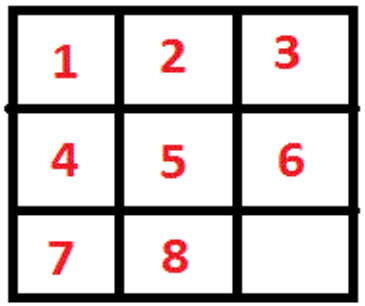
A common mid-state evaluation adds features of the state together

(we did this already for a heuristic...)

eval(START)=17



GOAL



We summed the distances to the correct spots for all numbers

Mid-state evaluation

We then minimax (and prune) these mid-state evaluations as if they were the correct values

You can also weight features (i.e. getting the top row is more important in 8-puzzle)

A simple method in chess is to assign points for each piece: pawn=1, knight=4, queen=9... then sum over all pieces you have in play

Mid-state evaluation

What assumptions do you make if you use a weighted sum?

Mid-state evaluation

What assumptions do you make if you use a weighted sum?

A: The factors are independent (non-linear accumulation is common if the relationships have a large effect)

For example, a rook & queen have a synergy bonus for being together is non-linear, so queen=9, rook=5... but queen&rook = 16

Mid-state evaluation

There is also an issue with how deep should we look before making an evaluation?

Mid-state evaluation

There is also an issue with how deep should we look before making an evaluation?

A fixed depth? Problems if child's evaluation is overestimate and parent underestimate (or visa versa)

Ideally you would want to stop on states where the mid-state evaluation is most accurate

Mid-state evaluation

Mid-state evaluations also favor actions that "put off" bad results (i.e. they like stalling)

In go this would make the computer use up ko threats rather than give up a dead group

By evaluating only at a limited depth, you reward the computer for pushing bad news beyond the depth (but does not stop the bad news from eventually happening)

Mid-state evaluation

It is not easy to get around these limitations:

1. Push off bad news
2. How deep to evaluate?

A better mid-state evaluation can help compensate, but they are hard to find

They are normally found by mimicking what expert human players do, and there is no systematic good way to find one

Forward pruning

You can also use mid-state evaluations for alpha-beta type pruning

However as these evaluations are estimates, you might prune the optimal answer if the heuristic is not perfect (which it won't be)

In practice, this prospective pruning is useful as it allows you to prioritize spending more time exploring hopeful parts of the search tree

Forward pruning

You can also save time searching by using “expert knowledge” about the problem

For example, in both Go and Chess the start of the game has been very heavily analyzed over the years

There is no reason to redo this search every time at the start of the game, instead we can just look up the “best” response

Random games

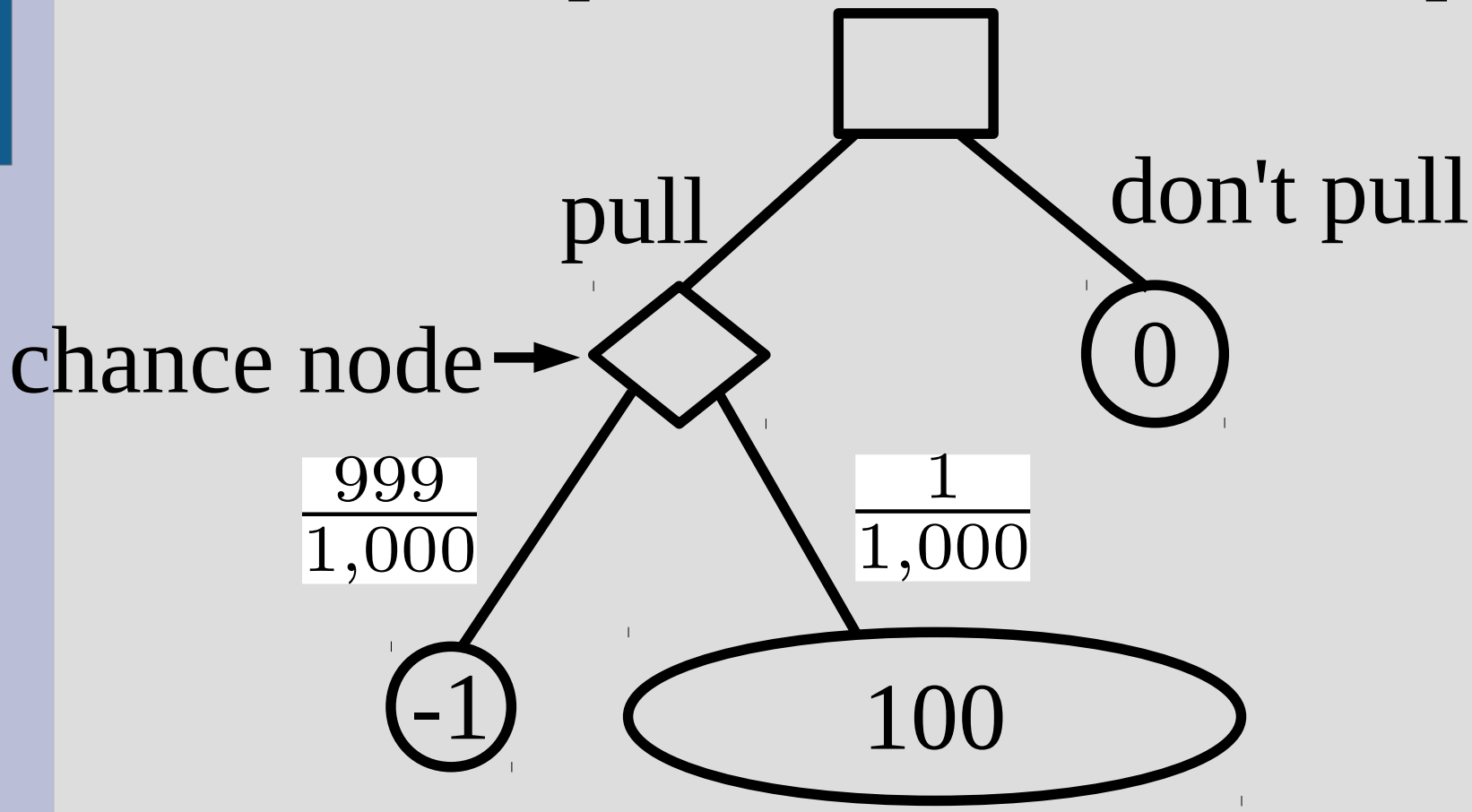
If we are playing a “game of chance”, we can add chance nodes to the search tree

Instead of either player picking max/min, it takes the expected value of its children

This expected value is then passed up to the parent node which can choose to min/max this chance (or not)

Random games

Here is a simple slot machine example:

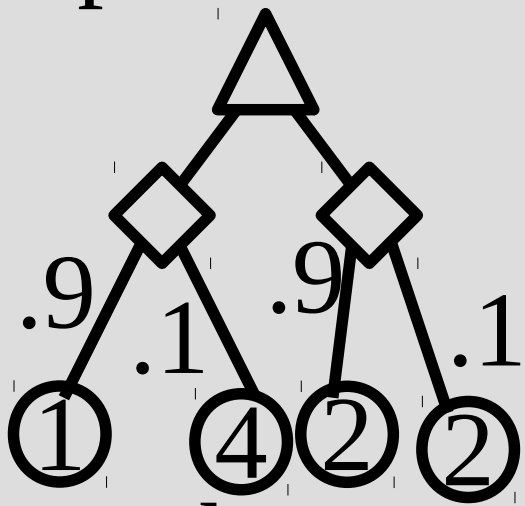


$$V(\text{chance}) = -1 \cdot \frac{999}{1,000} + 100 \cdot \frac{1}{1,000} = -0.899$$

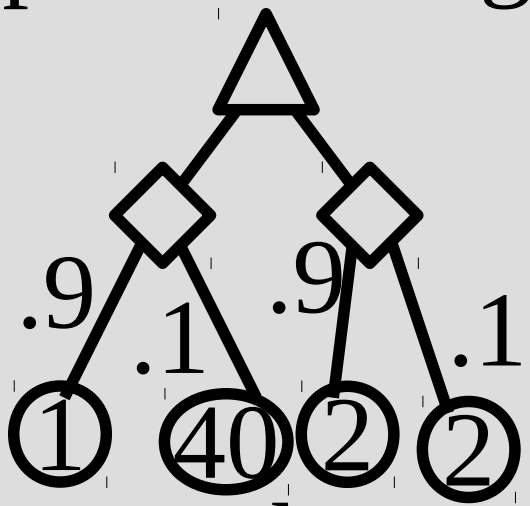
Random games

You might need to modify your mid-state evaluation if you add chance nodes

Minimax just cares about the largest/smallest, but expected value is an implicit average:



R is better



L is better

Random games

Some partially observable games (i.e. card games) can be searched with chance nodes

As there is a high degree of chance, often it is better to just assume full observability (i.e. you know the order of cards in the deck)

Then find which actions perform best over all possible chance outcomes (i.e. all possible deck orderings)

Random games

For example in blackjack, you can see what cards have been played and a few of the current cards in play

You then compute all possible decks that could lead to the cards in play (and used cards)

Then find the value of all actions (hit or stand) averaged over all decks (assumed equal chance of possible decks happening)

Random games

If there are too many possibilities for all the chance outcomes to “average them all”, you can sample

This means you can search the chance-tree and just randomly select outcomes (based on probabilities) for each chance node

If you have a large number of samples, this should converge to the average

MCTS

Suppose there are three slot machines and you know they have different winning percentages

How should you determine which machine to play?

What is a “good” action at any given point in time?

MCTS

This has been well studied and is called a multi-armed bandit problem

The key idea is that there is a balance between "exploring" options and playing "good" ones

-If you try each one of the slot machines 1/3 of the time, you'll probably lose money
-If you play each one once, then play the rest the highest machine, might lose out on best

MCTS

How to find “good” actions for minimax trees?

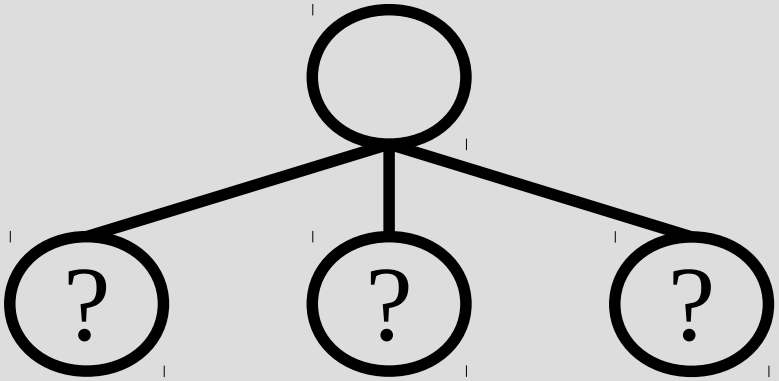
The “Upper Confidence Bound applied to Trees” UCT is commonly used:

$$\max_{n \in children} \left(\frac{win(n)}{times(n)} + \sqrt{\frac{2 \ln times(parent(n))}{times(n)}} \right)$$

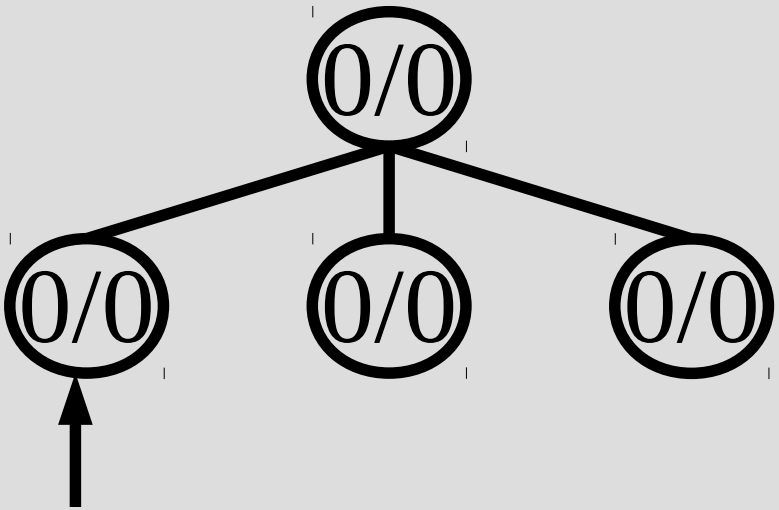
This ensures a trade off between checking branches you haven't explored much and exploring hopeful branches

(https://www.youtube.com/watch?v=Fbs4lnGLS8M)

MCTS



MCTS

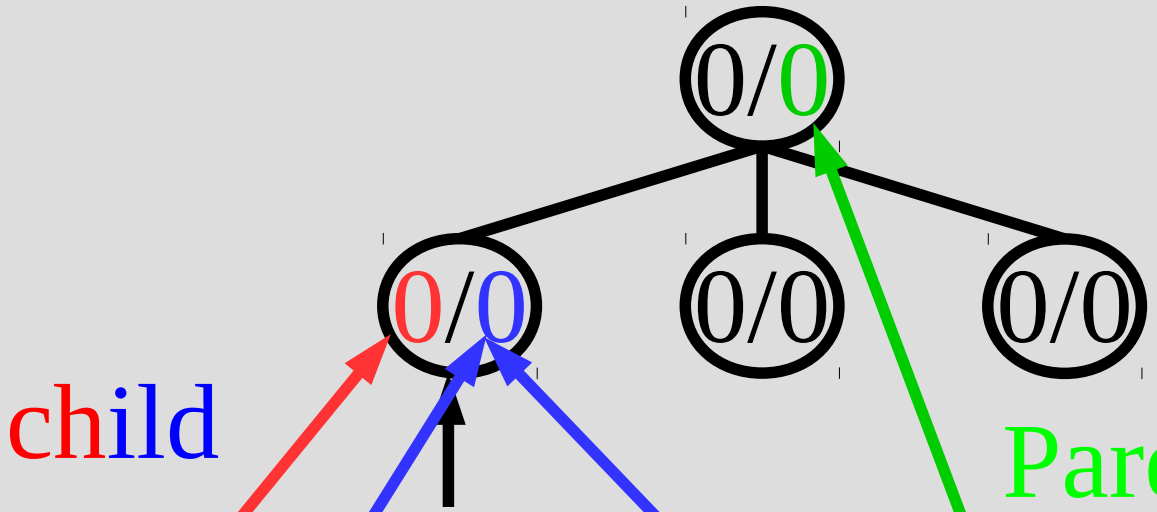


$$\frac{win(n)}{times(n)} + \sqrt{\frac{2\ln times(parent(n))}{times(n)}}$$

$$= \frac{0}{0} + \sqrt{\frac{2ln0}{0}}$$

$$= \infty$$

MCTS

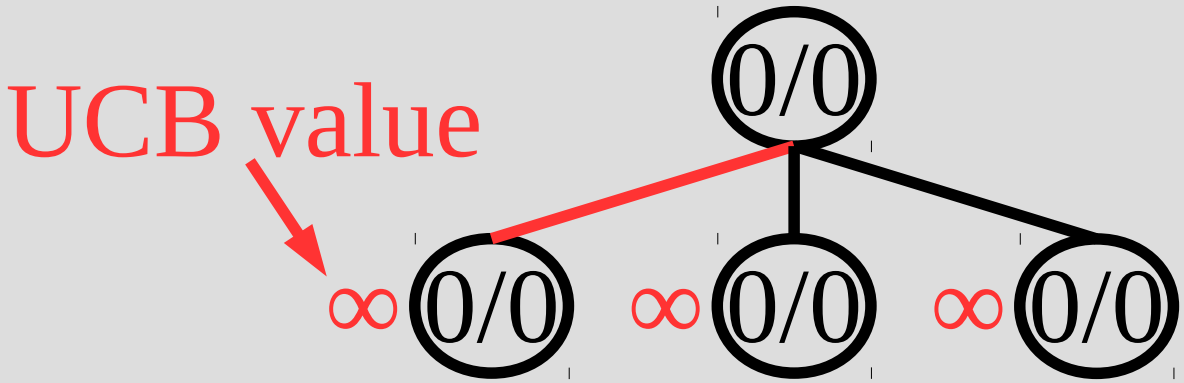


$$\frac{win(n)}{times(n)} + \sqrt{\frac{2\ln times(parent(n))}{times(n)}}$$

$$= \frac{0}{0} + \sqrt{\frac{2ln0}{0}}$$

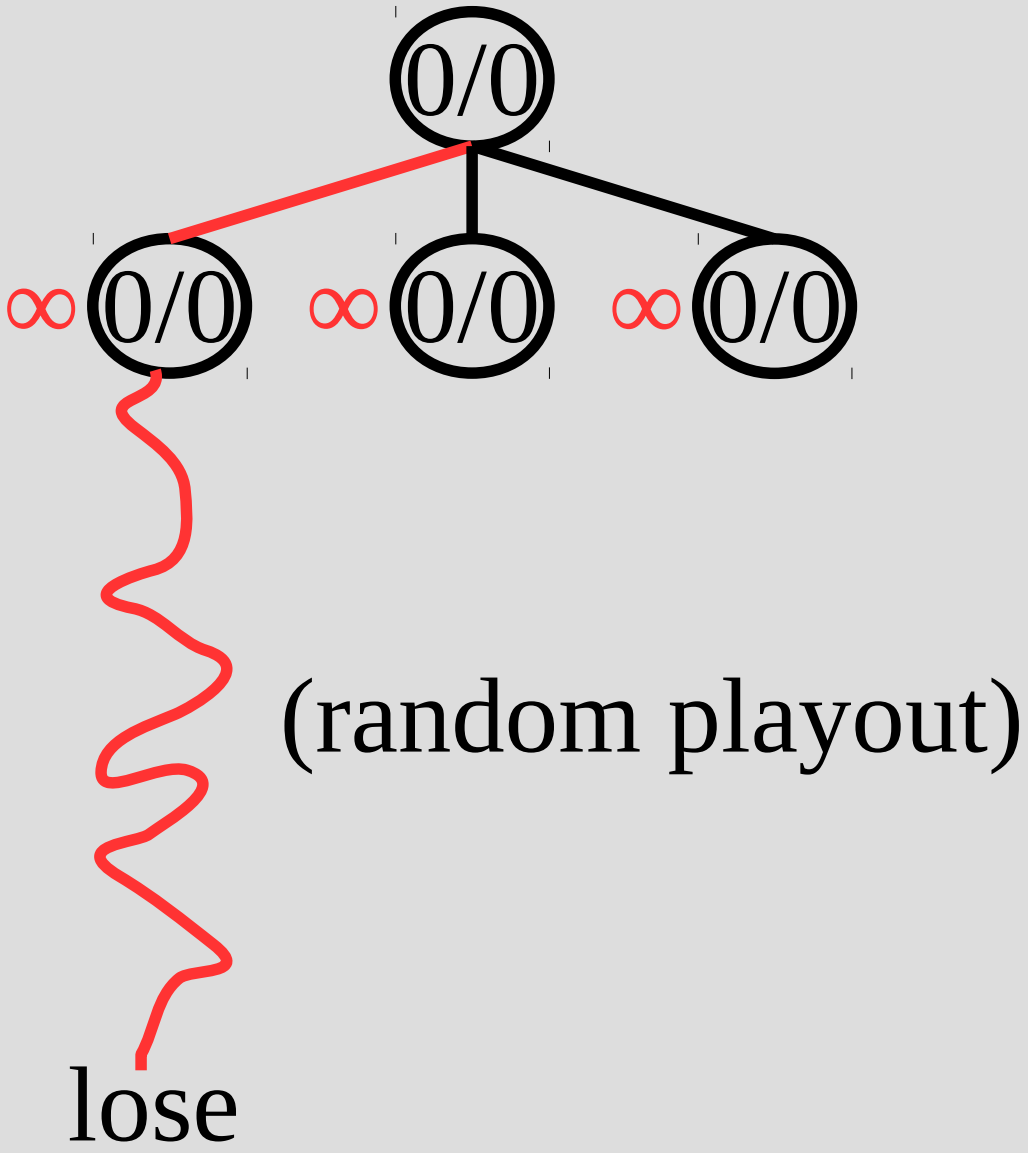
$$= \infty$$

MCTS

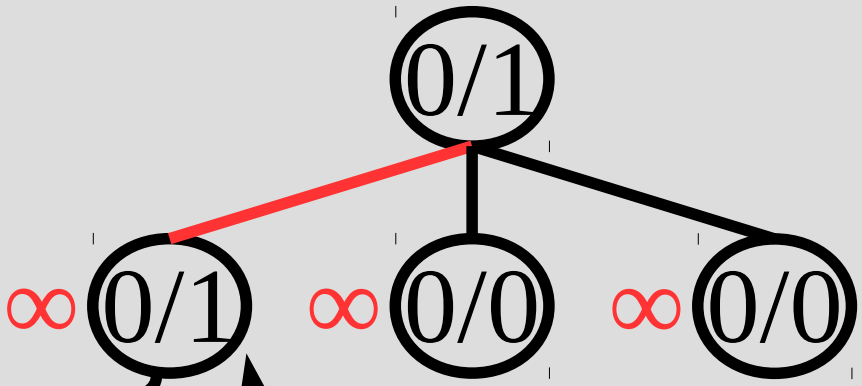


Pick max on depth 1 (I'll pick left-most)

MCTS



MCTS

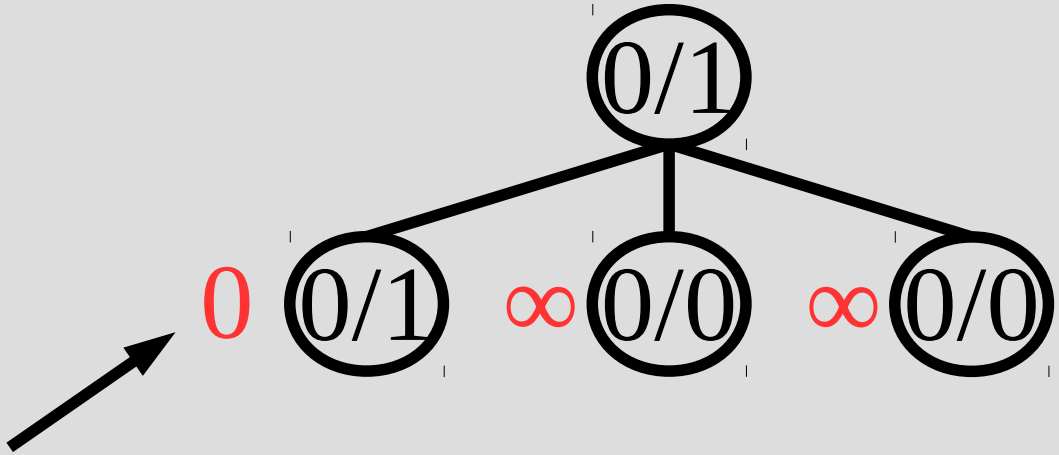


update (all the way to root)

(random playout)

lose

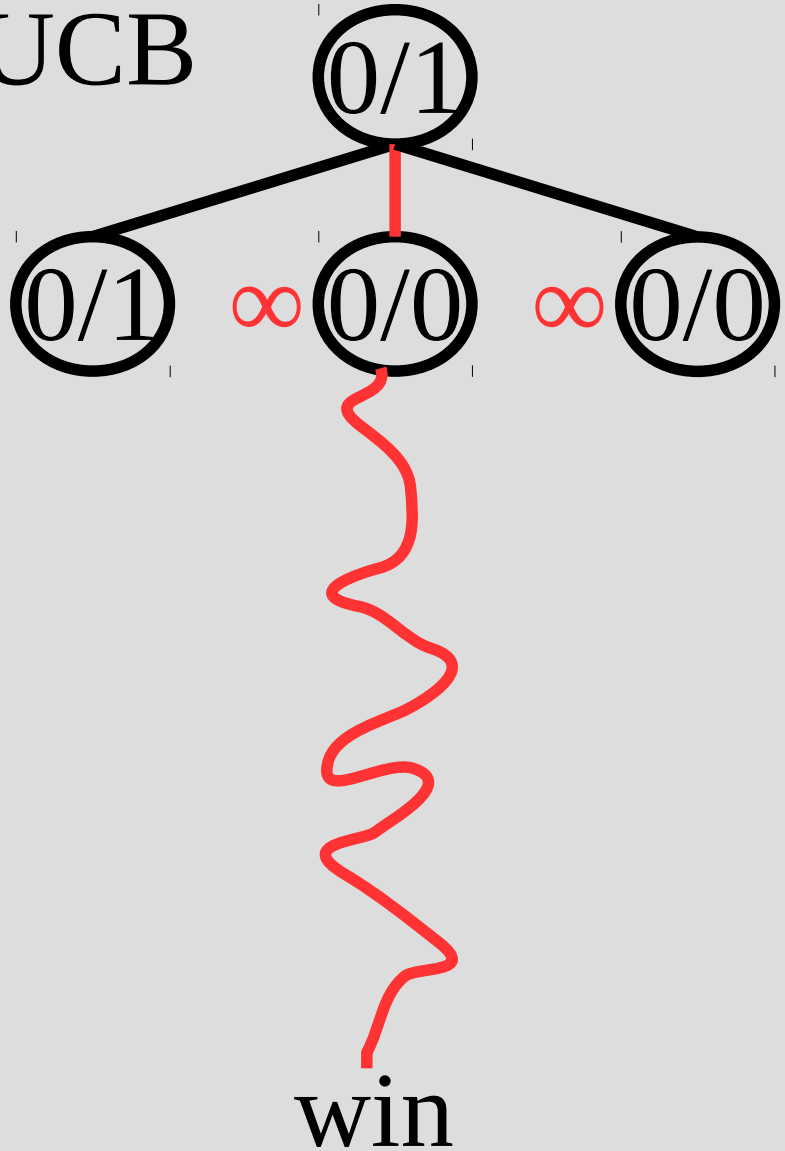
MCTS



update UCB values (all nodes)

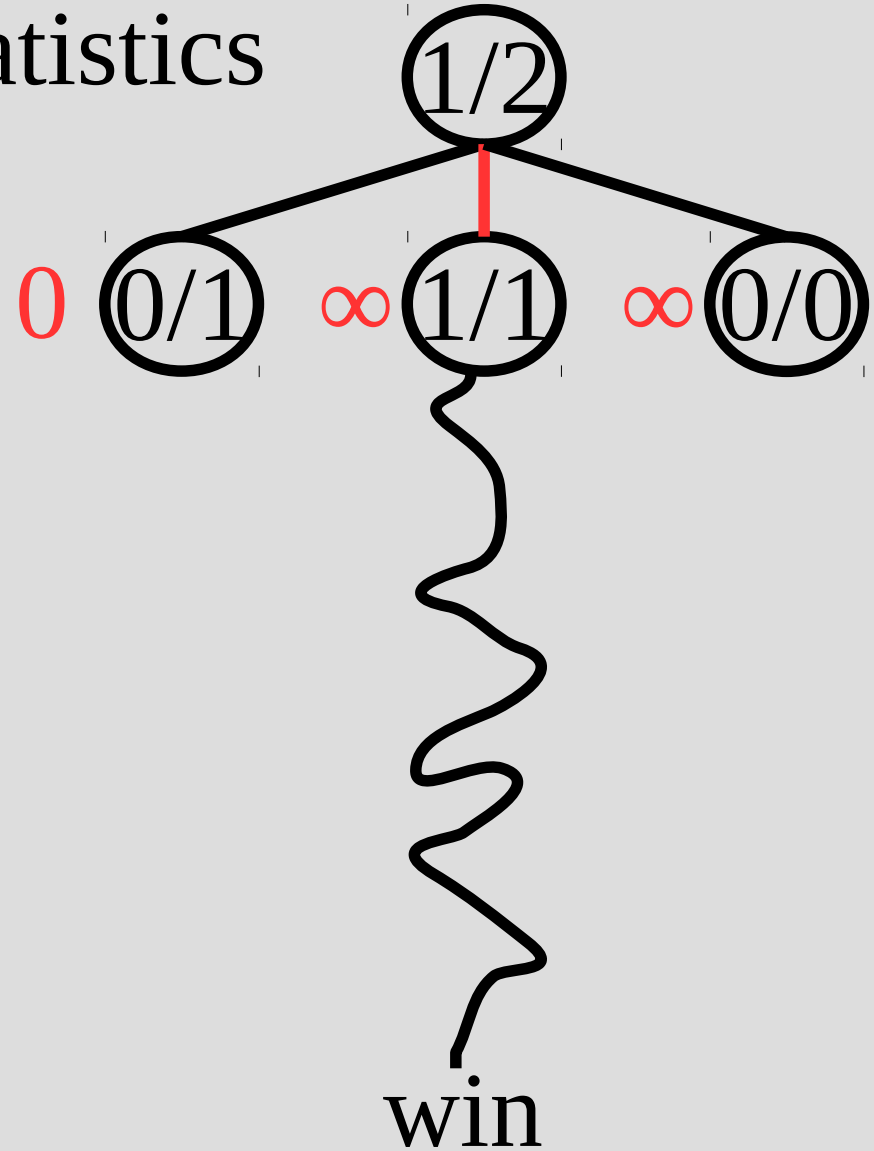
MCTS

select max UCB
on depth 1
& rollout



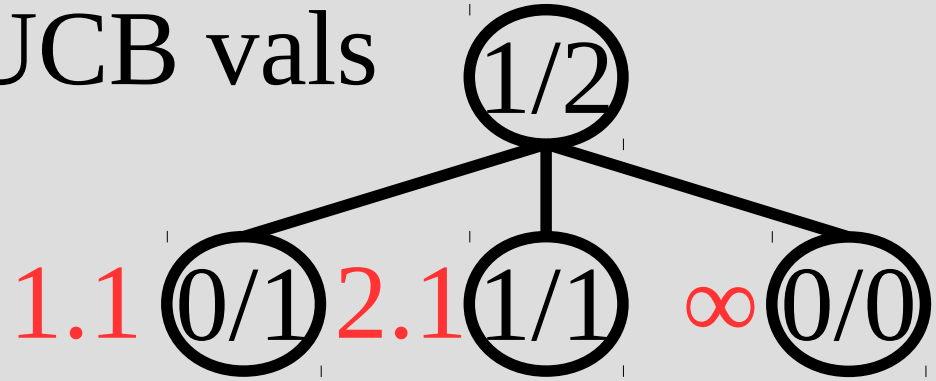
MCTS

update statistics



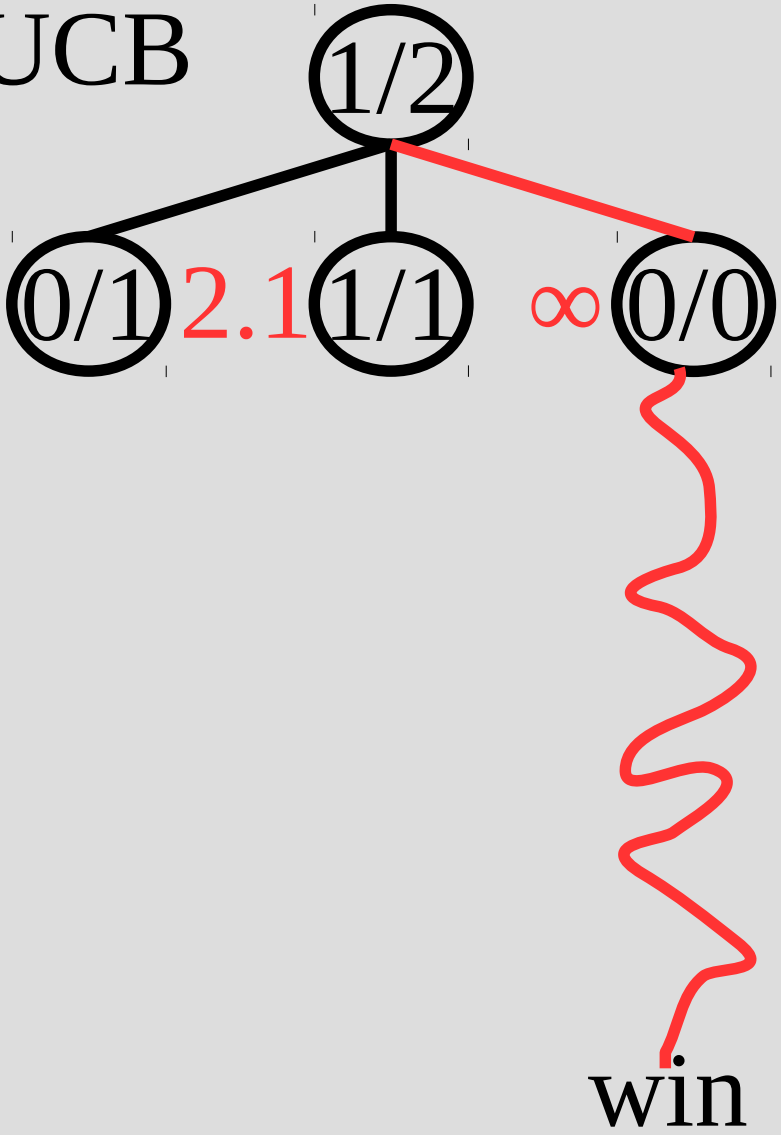
MCTS

update UCB vals



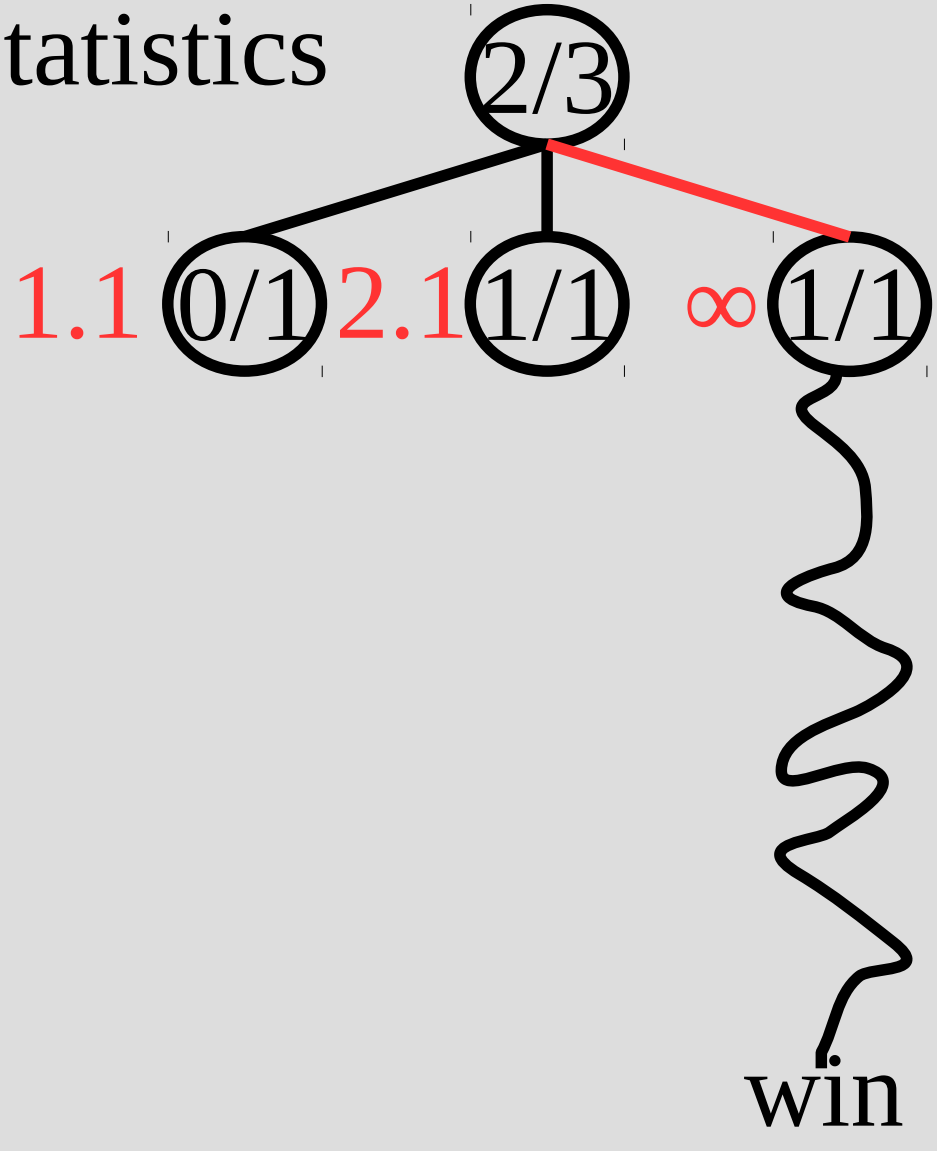
MCTS

select max UCB
on depth 1
&rollout



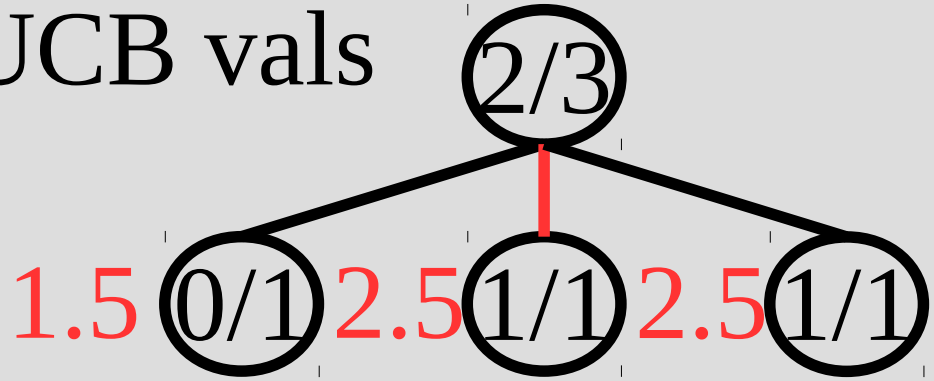
MCTS

update statistics



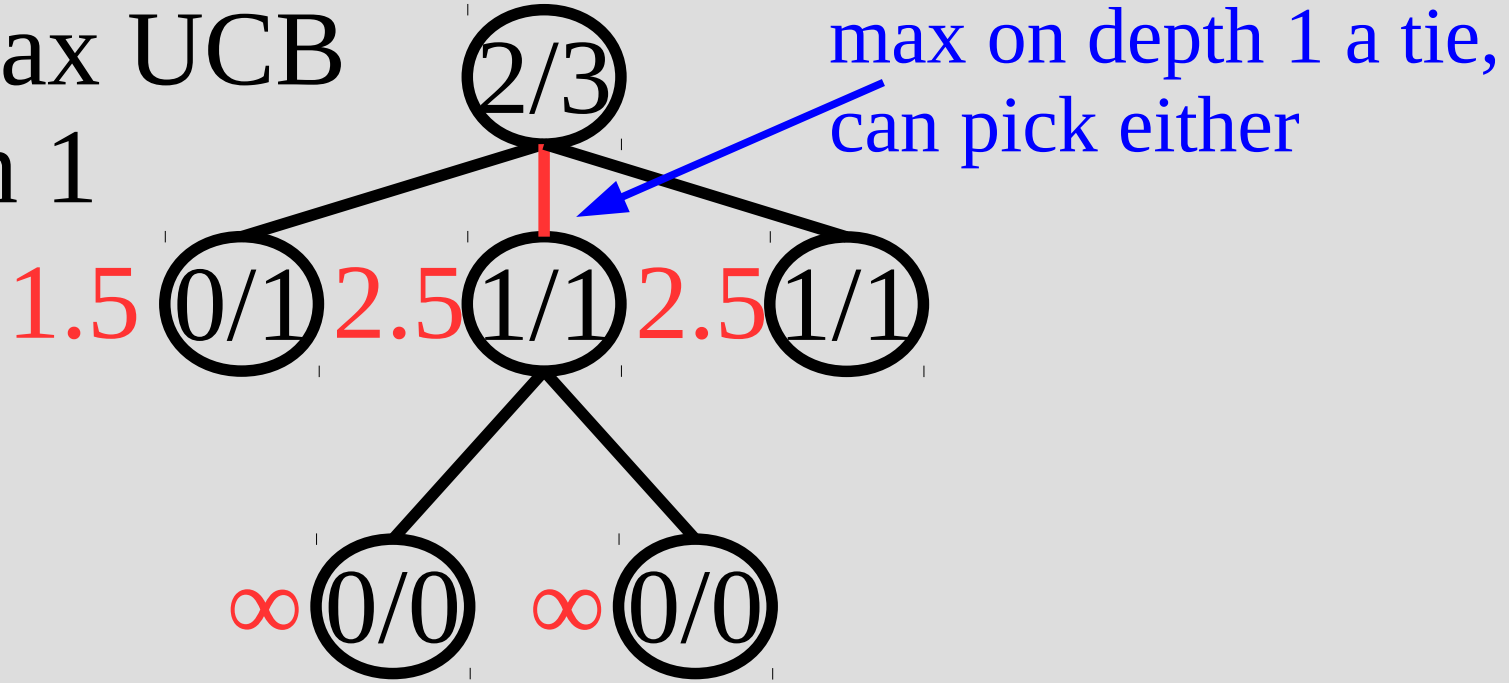
MCTS

update UCB vals



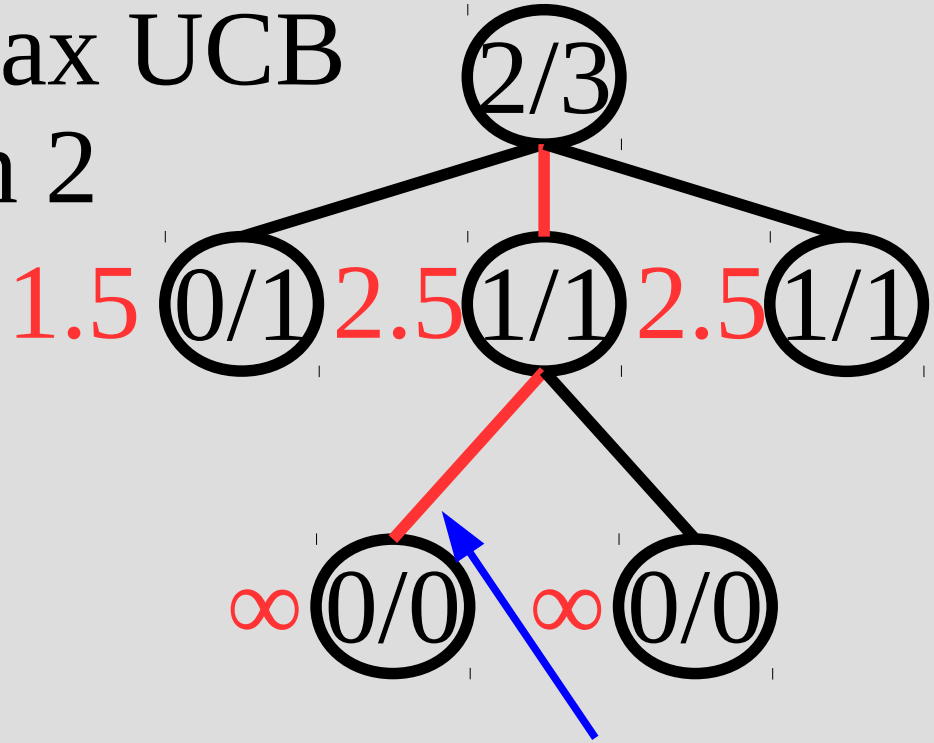
MCTS

select max UCB
on depth 1



MCTS

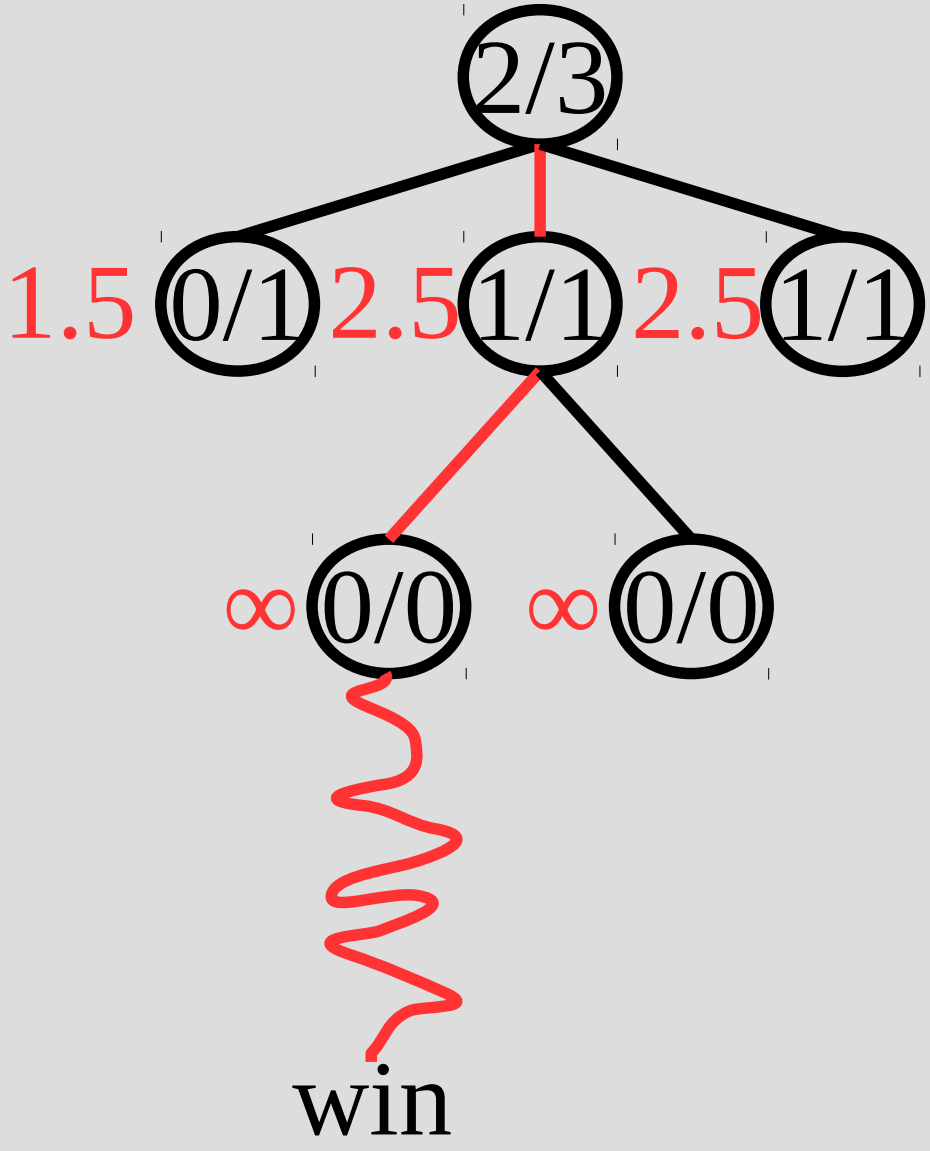
select max UCB
on depth 2



also a tie on depth 2,
can pick either (I go left)

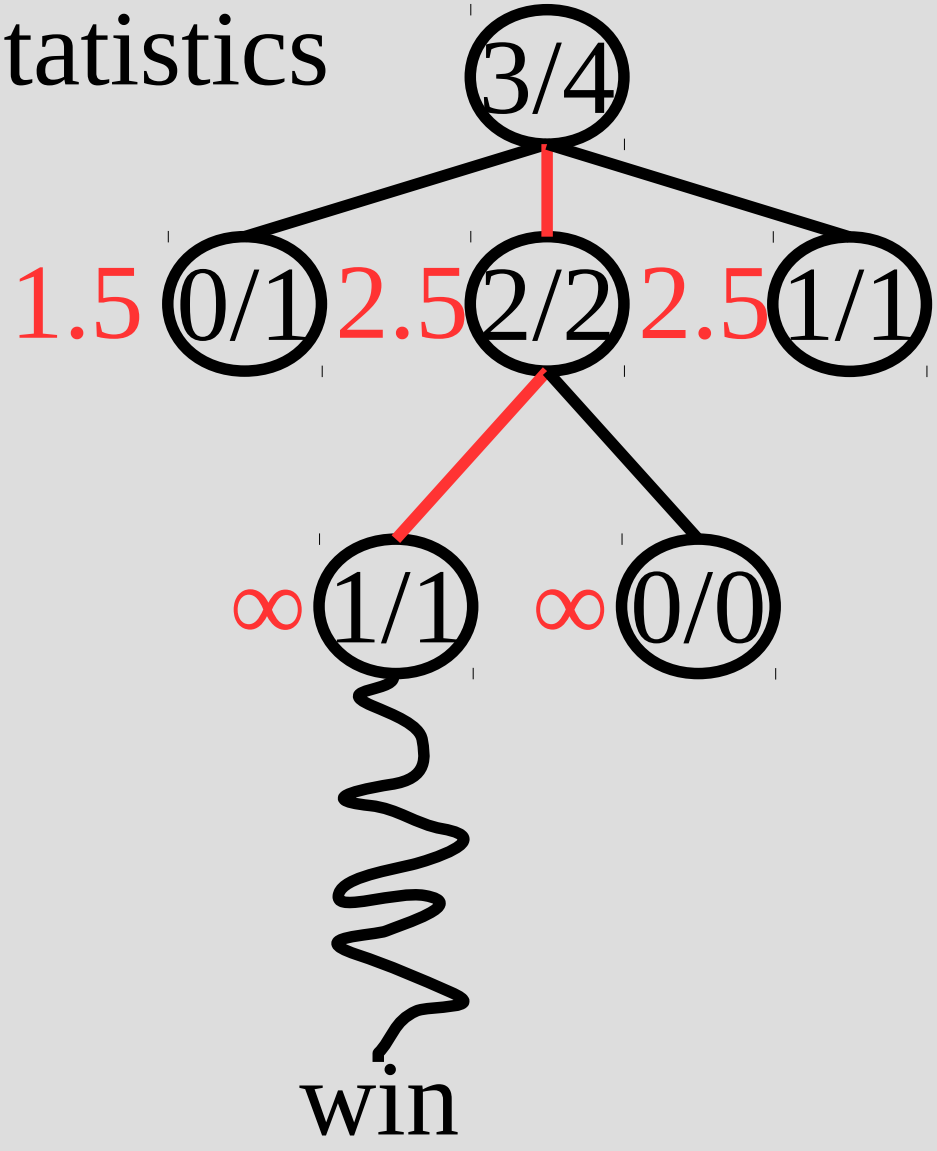
MCTS

rollout



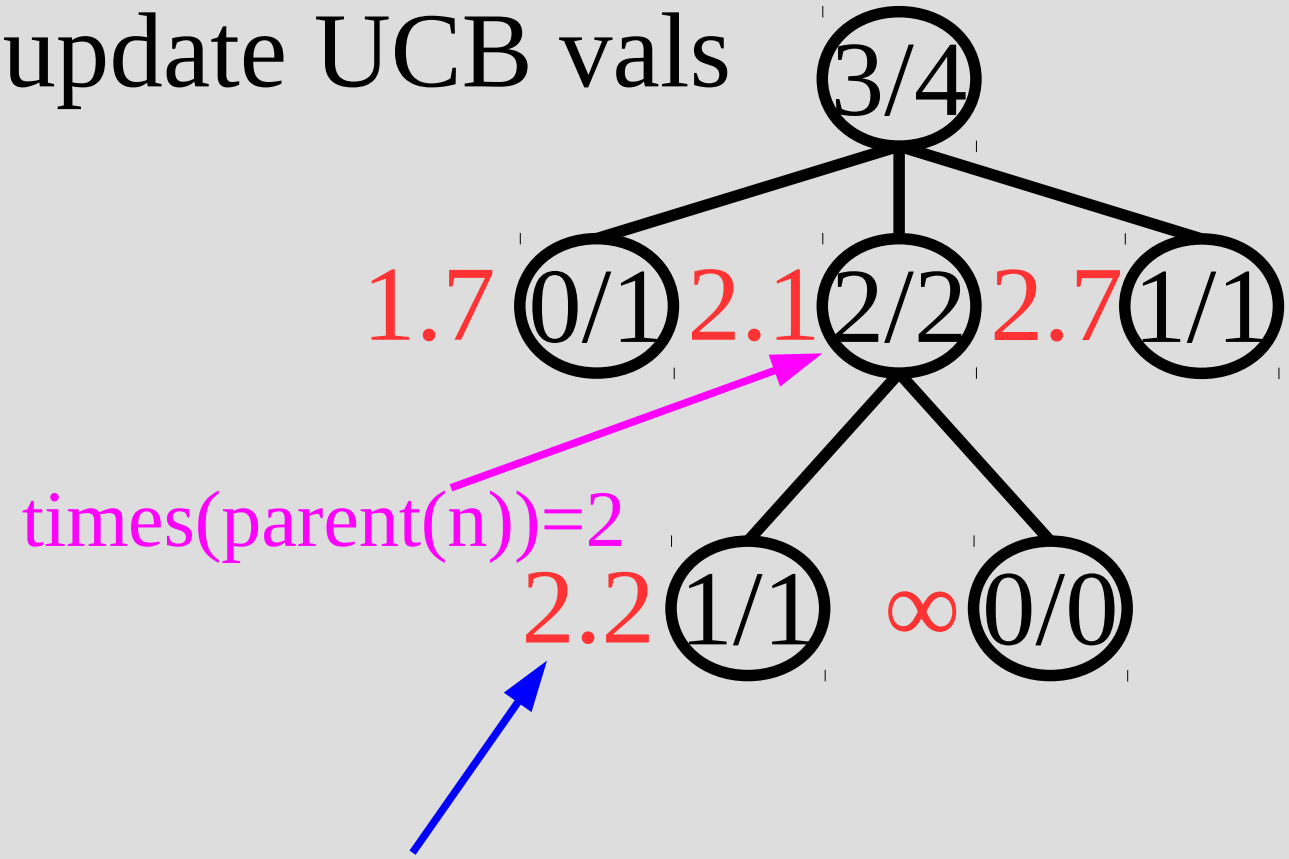
MCTS

update statistics



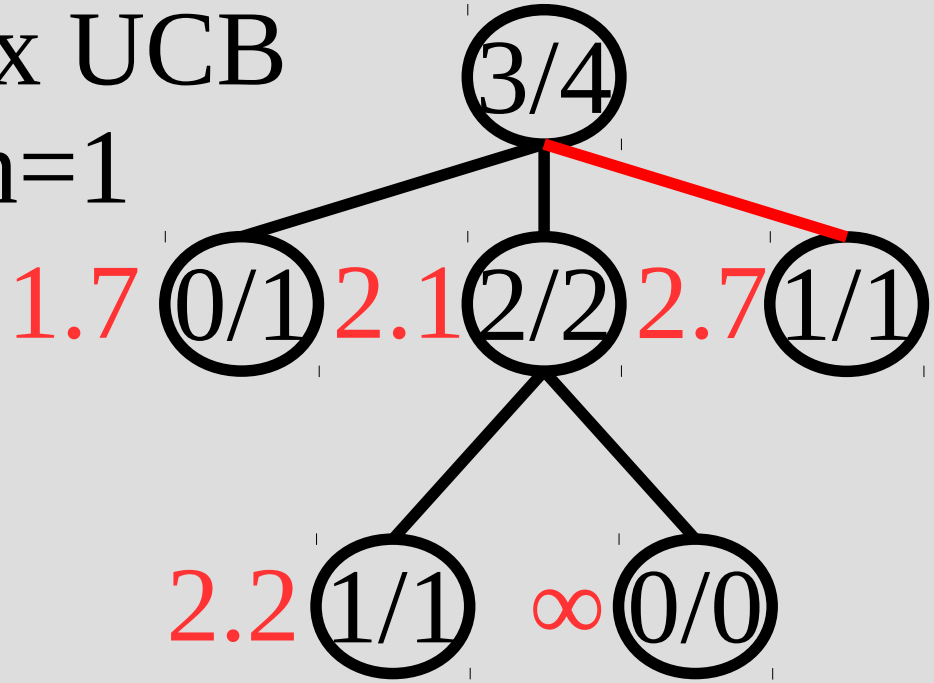
MCTS

update UCB vals



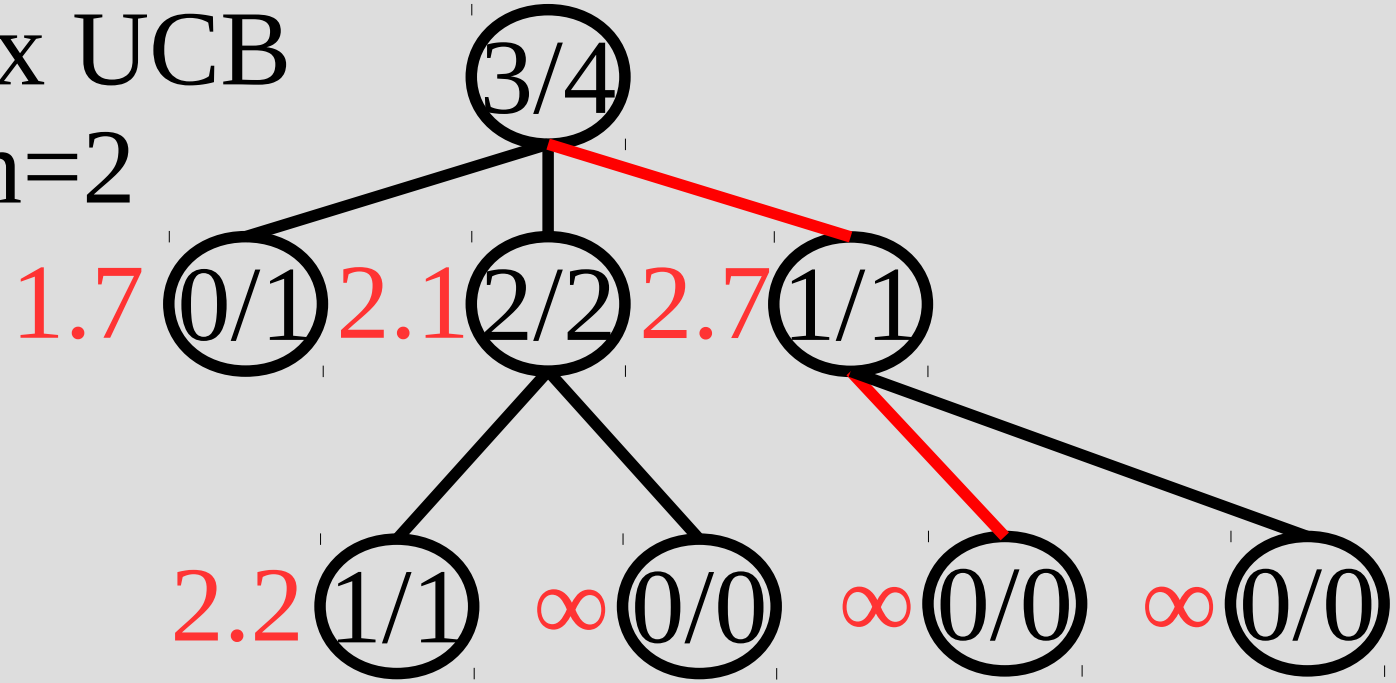
MCTS

pick max UCB
on depth=1



MCTS

pick max UCB
on depth=2



MCTS

So the algorithm's pseudo-code is:

Loop:

(1) Start at root

(2) Pick child with best UCB value

(3) If current node visited before,
goto step (2)

(4) Do a random "rollout" and record
result up tree until root

MCTS

Pros:

(1) The “random playouts” are essentially generating a mid-state evaluation for you

(2) Has shown to work well on wide & deep trees, can also combine distributed comp.

Cons:

(1) Does not work well if the state does not “build up” well

(2) Often does not work on 1-player games

MCTS in games

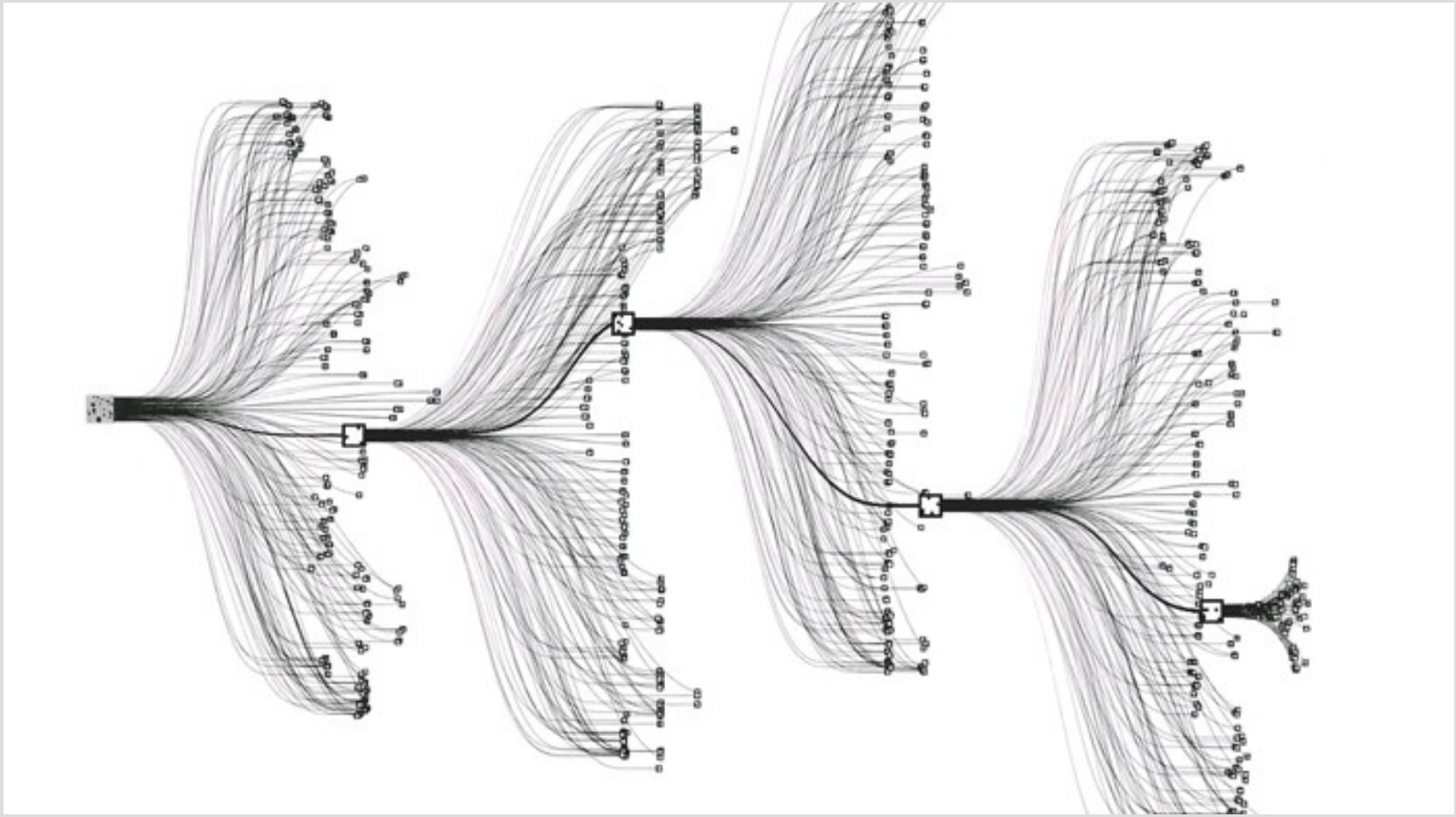
AlphaGo/Zero has been in the news recently, and is also based on neural networks

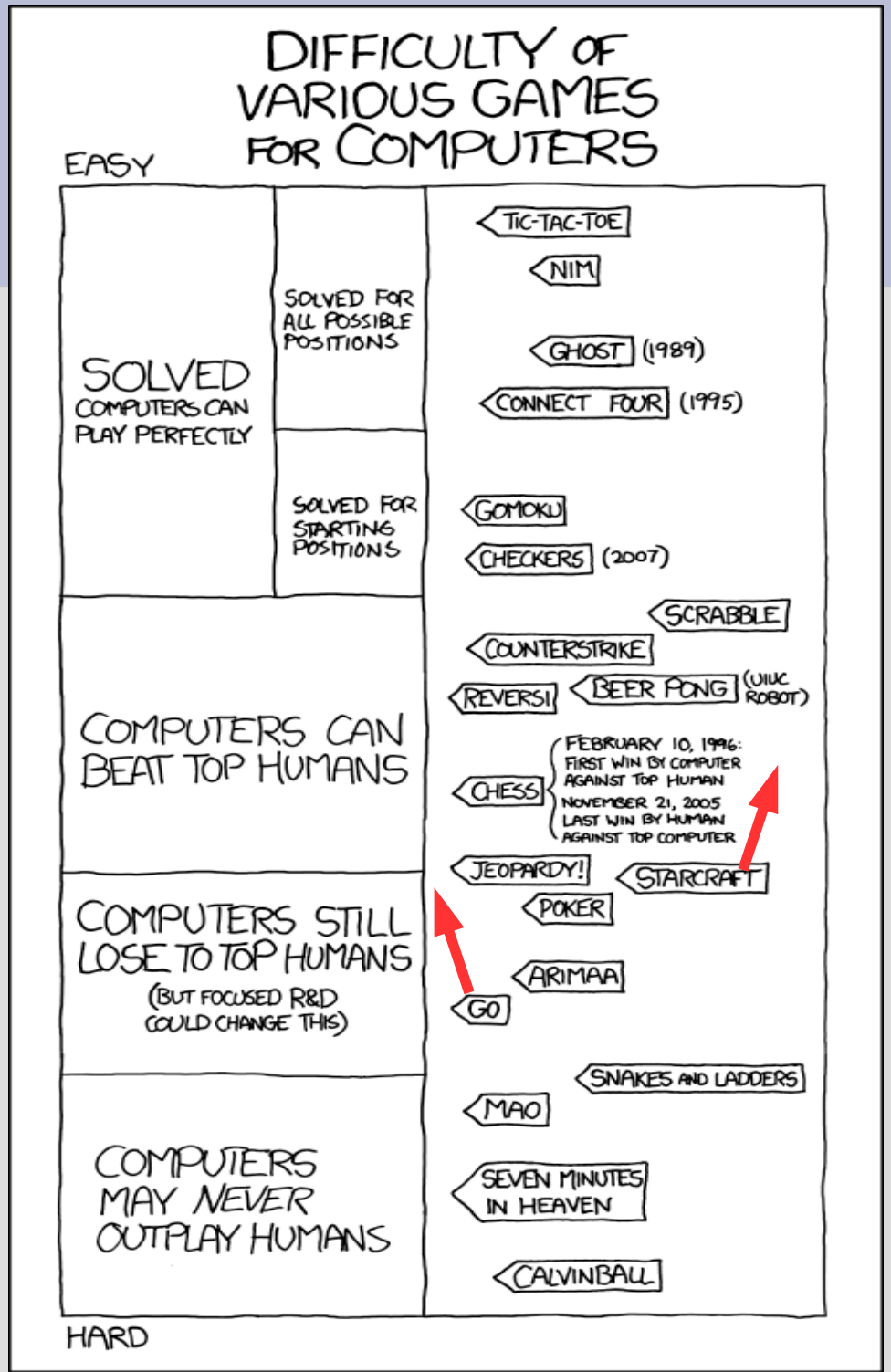
AlphaGo uses Monte-Carlo tree search guided by the neural network to prune useless parts

Often limiting Monte-Carlo in a static way reduces the effectiveness, much like mid-state evaluations can limit algorithm effectiveness

MCTS in games

Basically, AlphaGo uses a neural network to “prune” parts for a Monte-carlo search




DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS
EASY
SOLVED
COMPUTERS CAN PLAY PERFECTLY
SOLVED FOR ALL POSSIBLE POSITIONS
SOLVED FOR STARTING POSITIONS
COMPUTERS CAN BEAT TOP HUMANS
COMPUTERS STILL LOSE TO TOP HUMANS
(BUT FOCUSED R&D COULD CHANGE THIS)
COMPUTERS MAY NEVER OUTPLAY HUMANS
HARD
TIC-TAC-TOE
NIM
GHOST (1989)
CONNECT FOUR (1995)
GOMOKU
CHECKERS (2007)
SCRABBLE
COUNTERSTRIKE
REVERSI
BEER PONG (UIUC ROBOT)
CHESS
FEBRUARY 10, 1996: FIRST WIN BY COMPUTER AGAINST TOP HUMAN
NOVEMBER 21, 2005 LAST WIN BY HUMAN AGAINST TOP COMPUTER
JEOPARDY!
STARCRAFT
POKER
ARIMAA
GO
SNAKES AND LADDERS
MAO
SEVEN MINUTES IN HEAVEN
CALVINBALL