Bandits and Monte-Carlo Tree Search

Tor Lattimore

DeepMind, London (Sheffield)

Attendance code

Slides:

http://tor-lattimore.com/downloads/talks/2023/Sheffield.pdf

Please ask questions anytime

Program

- Two player full information games
- Monte-Carlo Tree Search (MCTS)
- Bandit problems
- Learning via self-play
- **Lab tomorrow** MCTS search implementation on Connect4

Huge thanks to Finnian Lattimore for the code

A mini history of computer game-playing

1950's Turing, Shannon and others develop first chess programs

1997 DeepBlue defeats Gary Kasparov using search and enormous (by the standards of the day) computation power

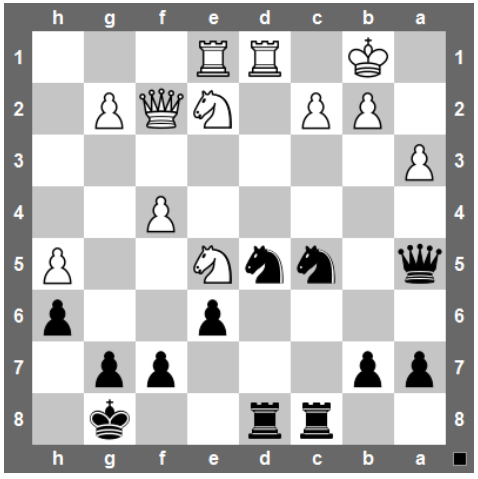
2003 I authored a chess engine that came 9th (of 12) in the Australian Computer Chess Championships

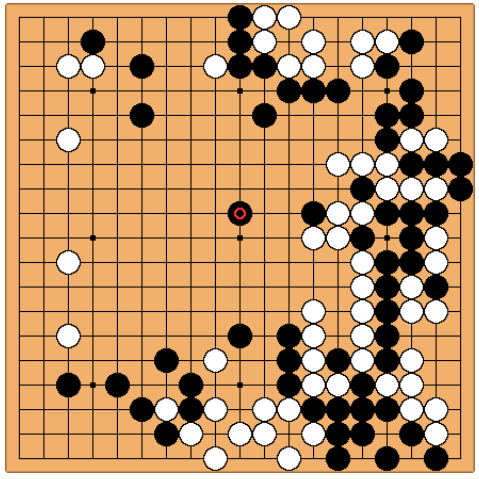
2016 AlphaGo defeats to Lee Sedol in Go using a highly selective search powered by machine learning (I joined DM in 2017)

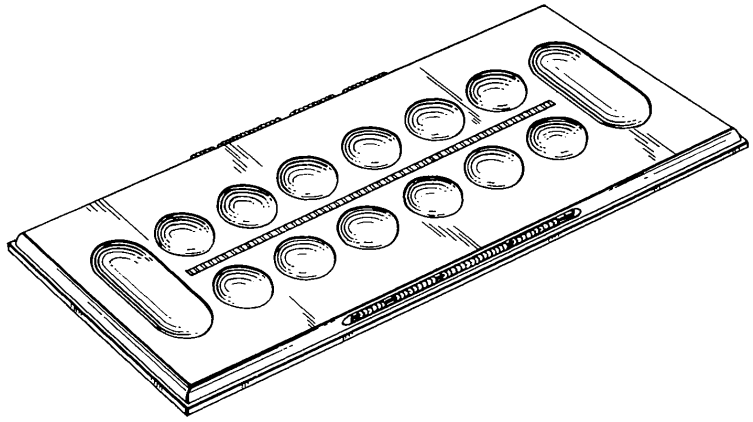
Two player full information deterministic zero-sum games

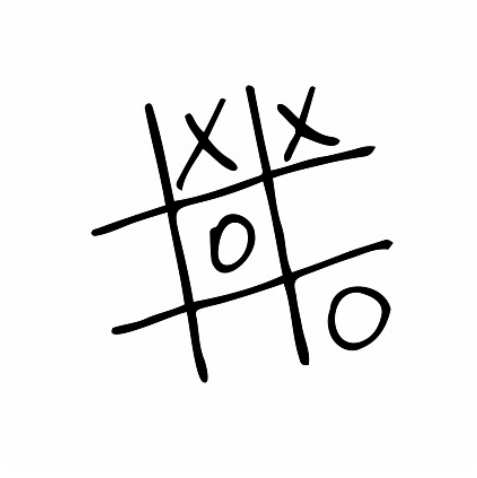
- Two players take turns taking actions
- At some point one of the players wins, the other loses (or they draw)
- The game is deterministic (no dice)
- No hidden knowledge

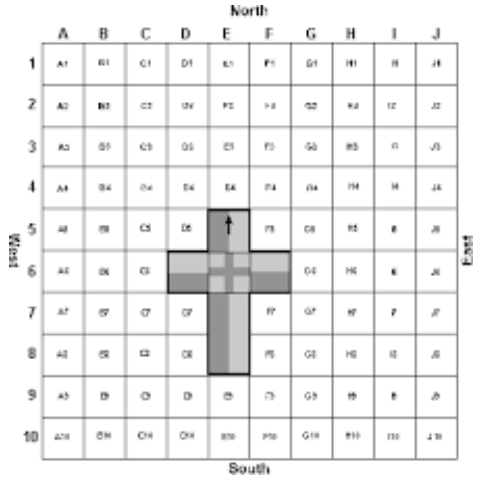
Examples



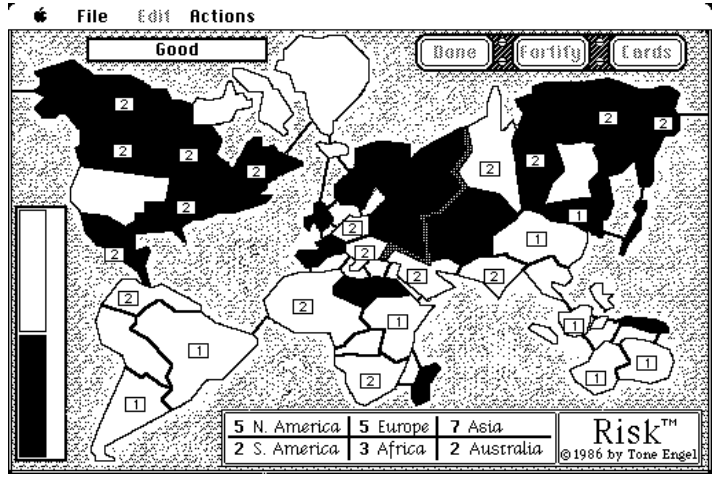








Non-examples



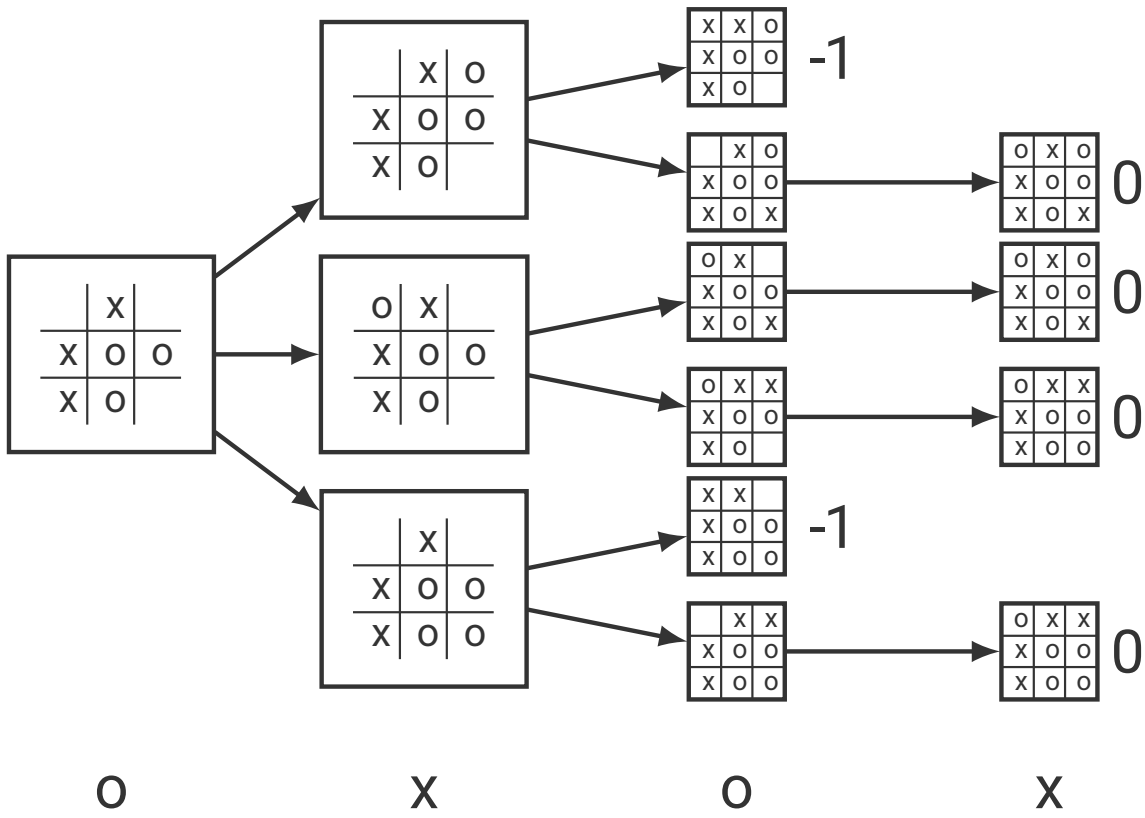
Formalising games

A two-player deterministic zero-sum full information game is defined by a tuple $(S, A, \mathrm{T}, Terminal, V, s_{init})$

- A set of states S
  - all possible board positions
- A collection of action sets: $\{A_s : s \in S\}$
  - A_s is the set of actions available in state s
- A transition function $\mathrm{T} : S \times A \to S$
- A set of terminal states $Terminal \subset S$
  - The game ends at terminal states
- An initial state $s_{init} \in S$
- An evaluation function $V : Terminal \to [-1, 1]$

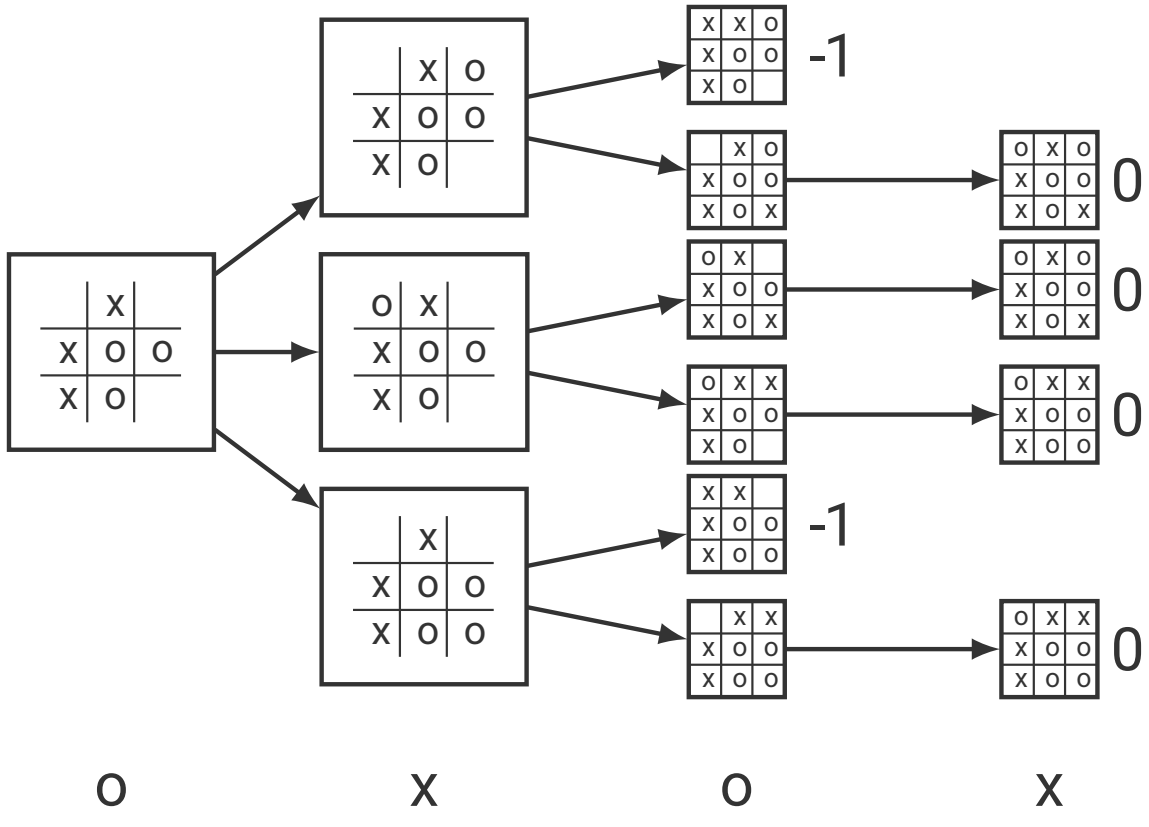
Game trees

- An alternative way of representing a game
- Nodes correspond to states
- Root node is the initial state
- Children of a given node/state correspond to states reachable from that node



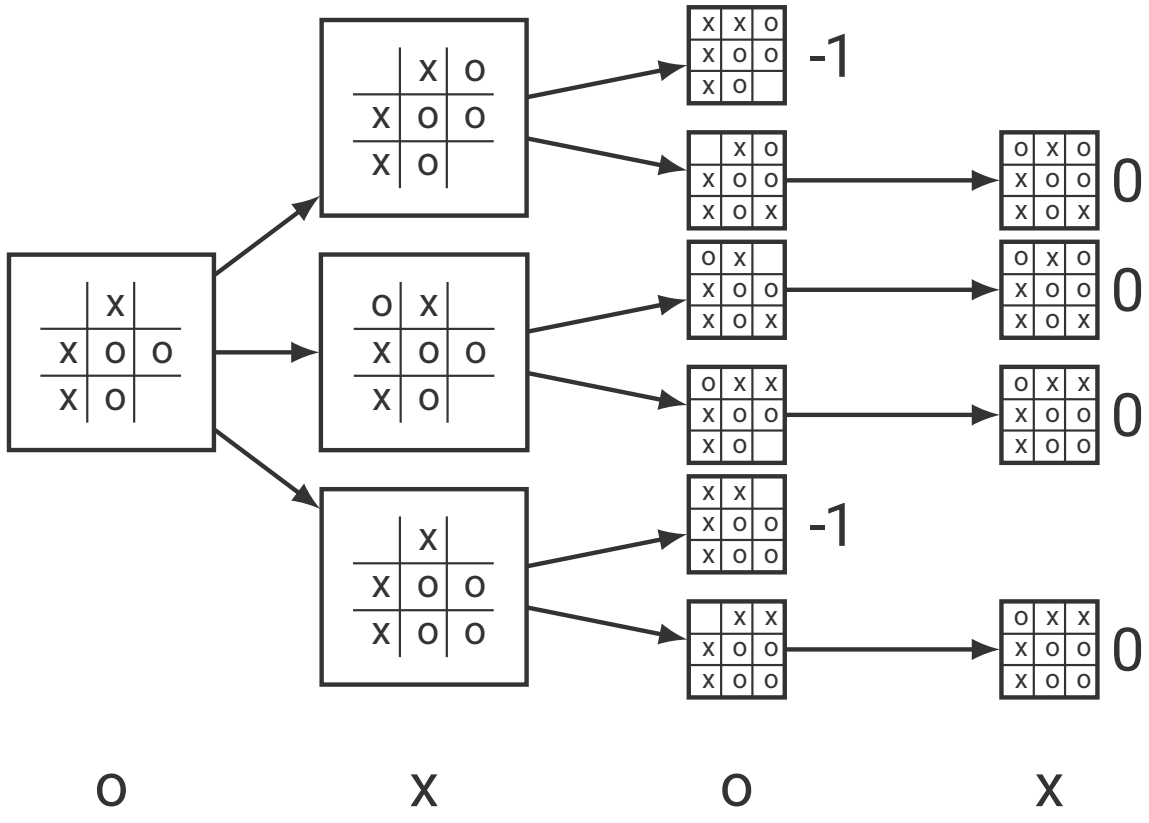
What is the best move?

A value of -1 is a win for ‘X’ and a 0 is a draw



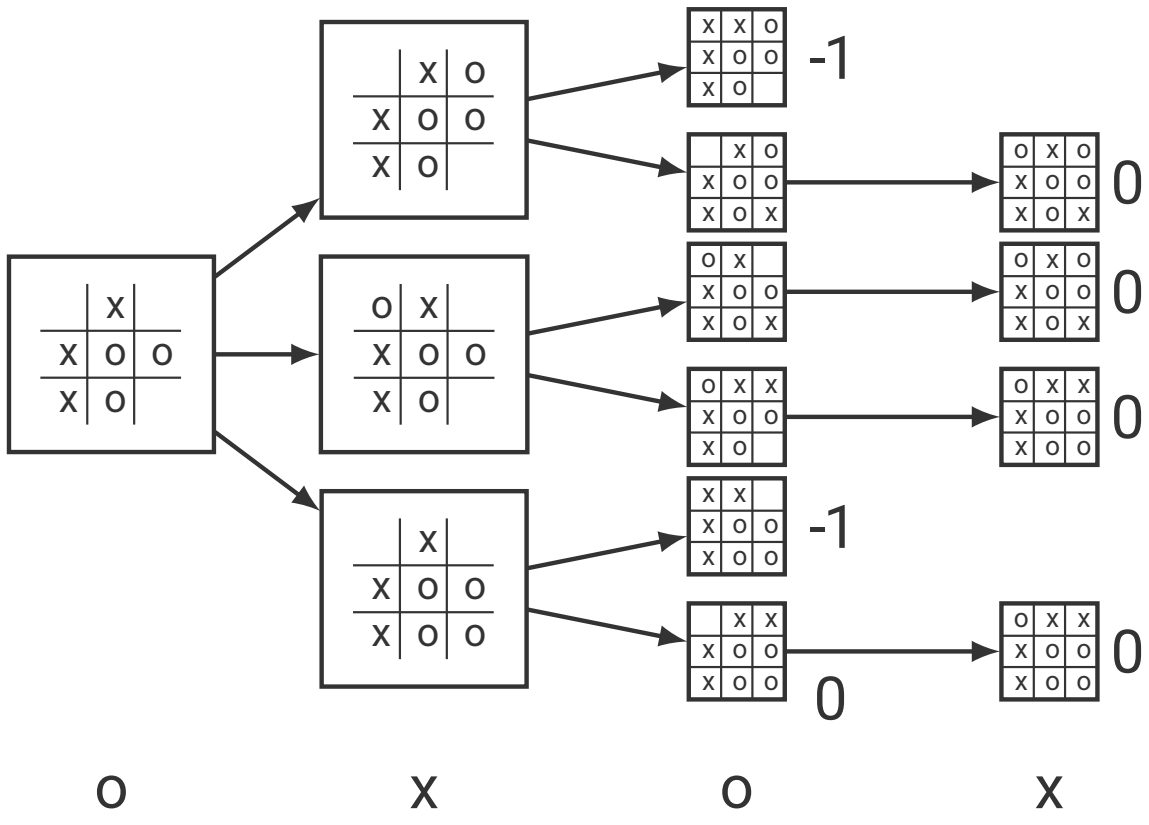
Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



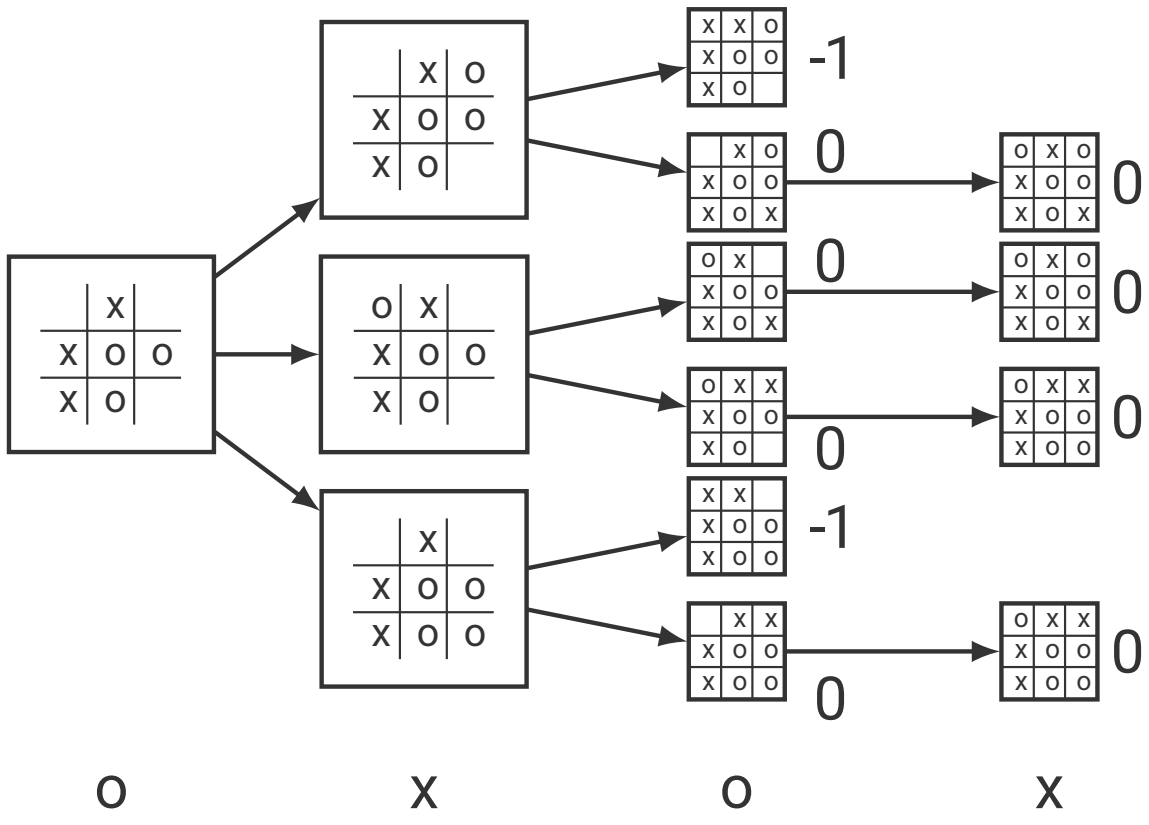
Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



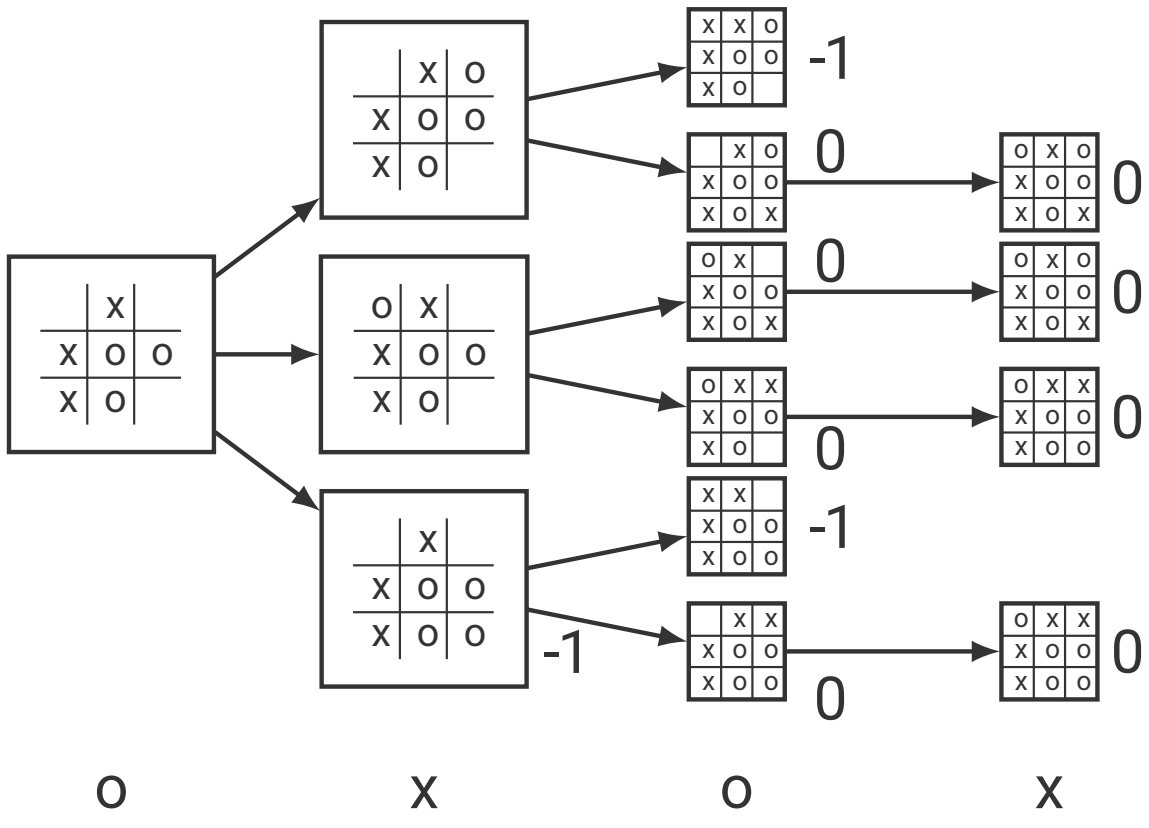
Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



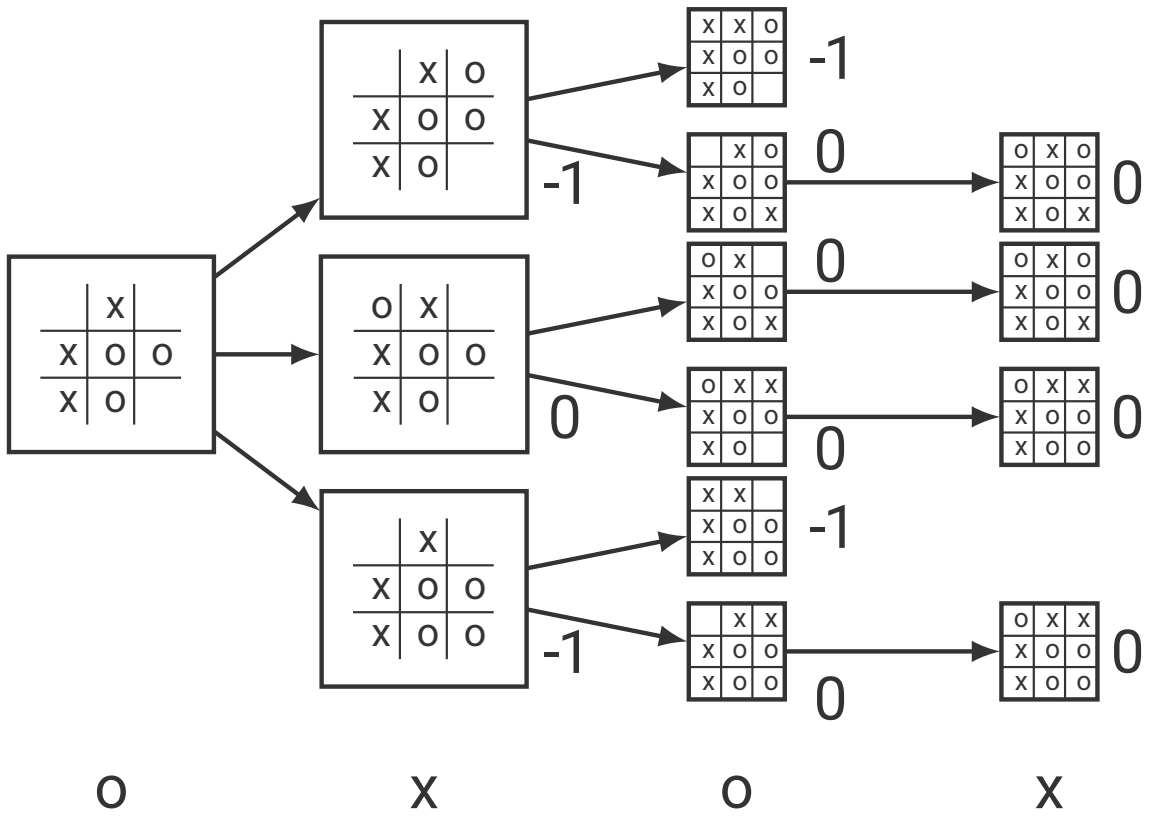
Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



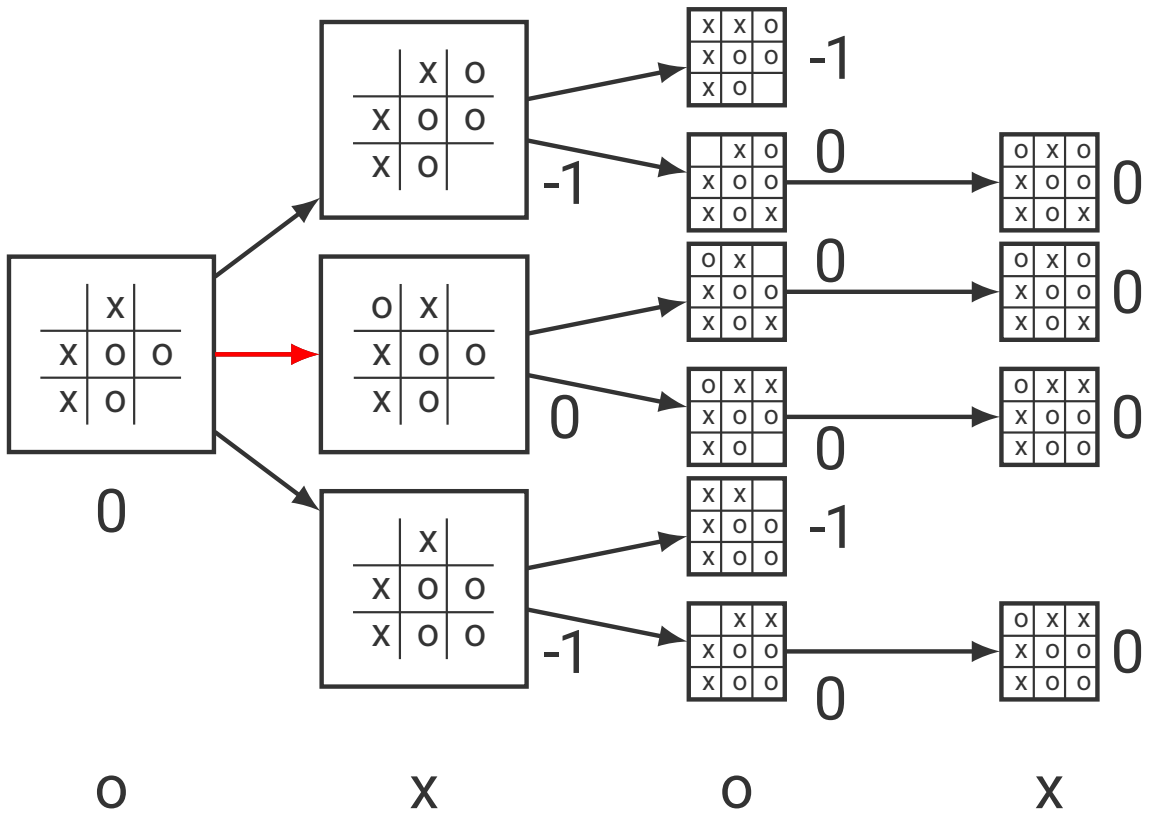
Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



Minimax search

- A method for finding the best move
- Naive implementation works backwards from the leaves of the game tree



Limitations of minimax search

- Minimax search is only practical for miniscule games
- Theoretically needs to search *all* states
  - **TicTacToe** 765
  - **Connect4** $4,531,985,219,092$
  - **Chess** 10^{123}
  - **Go** 10^{170}

Alpha-beta search

- A huge improvement on minimax
- Used by most chess engines until quite recently (e.g., DeepBlue or Stockfish)
- Not as flexible as MCTS
- **Basic idea** Prune nodes early that you know are suboptimal

MCTS search

MCTS takes as input a state and returns a move

- Deep searches, selective coverage
- No need for evaluation function
- Stop anytime

MCTS initialisation

MCTS runs for as many iterations as possible before a time limit

Builds a tree that grows by a single node in each iteration

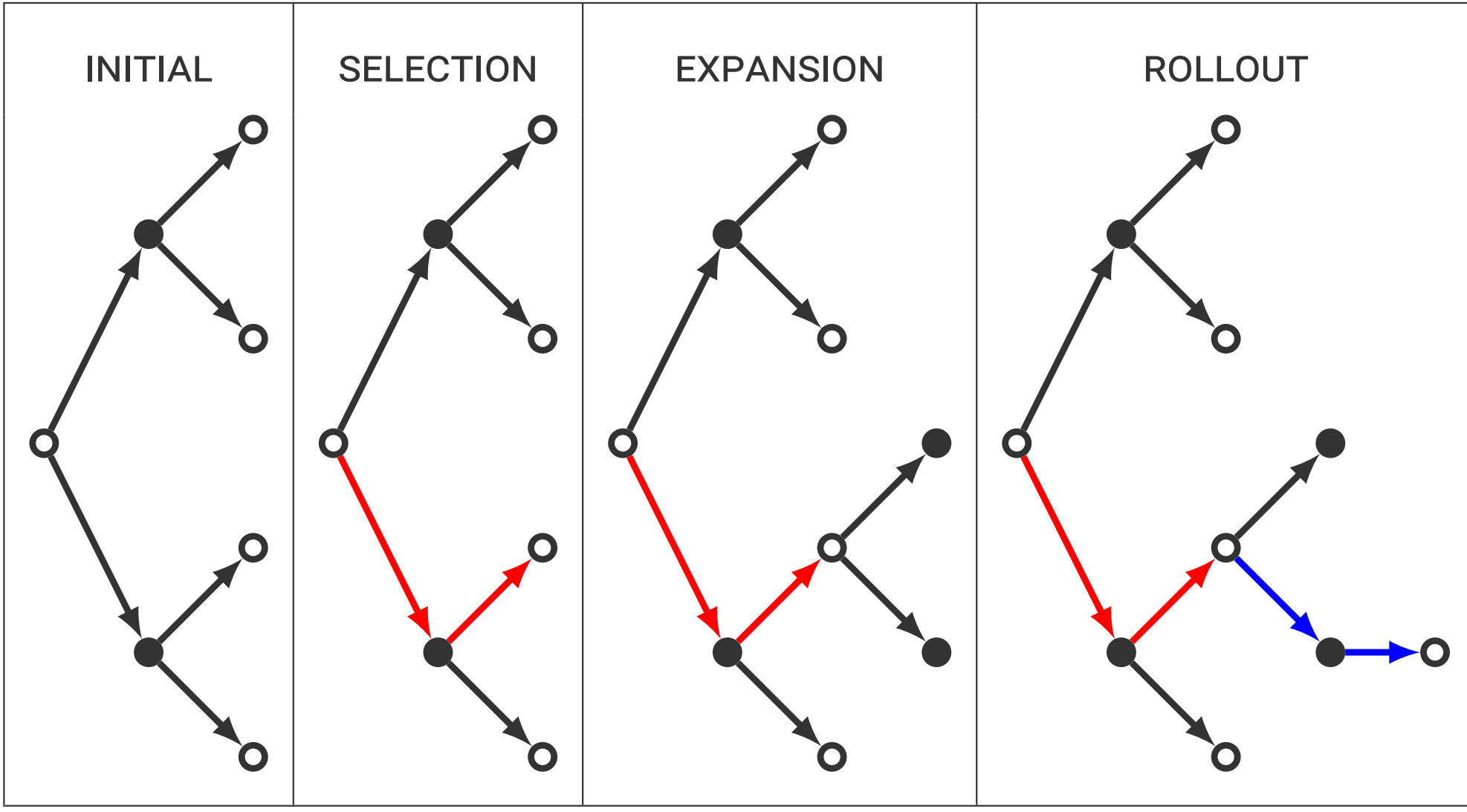
Tree is initialised with a single node

```
function search(board):
  while time_left():
    do_iteration()
  return select_move()
```

MCTS iteration

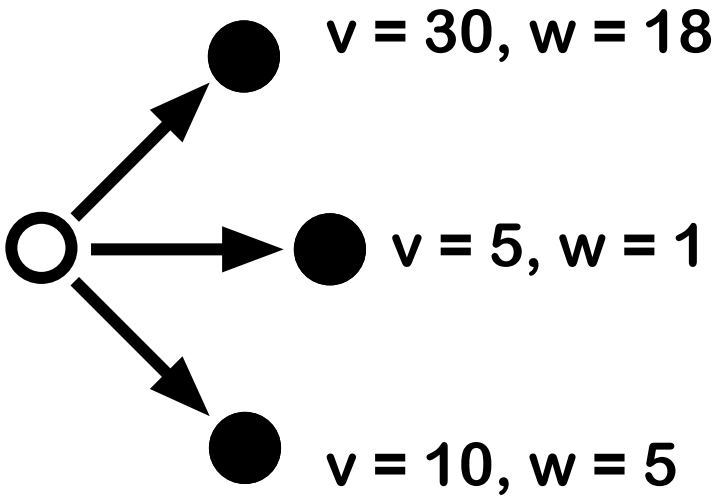
- Traverse the tree to a leaf using a **selection algorithm**
- **Expand** all possible moves at the leaf (tree gets bigger)
- Estimate the value of the leaf using a **rollout**
- **Propagate** the outcome back through the tree

MCTS iteration



Selection

- Selection algorithm operates at the node level
- Needs to choose which child to explore
- Should explore bad children just enough to know they are bad




```
class Node
  parent        # parent of this node
  children      # children of this node
  board         # board state

  visits        # number of visits
  wins          # number of wins
```

Selection

OPTION 1 Select $\arg\max_k \frac{w_k}{v_k} + c\sqrt{\frac{\log \|v\|_1}{v_k}}$

OPTION 2 Select $\arg\max_k \frac{w_k}{v_k} + \frac{c\sqrt{\|v\|_1}}{v_k}$

Same principle for both

select arms that either led to good outcomes in the past or have not been selected much

The constant c is tuned empirically. What do smaller/larger values mean?

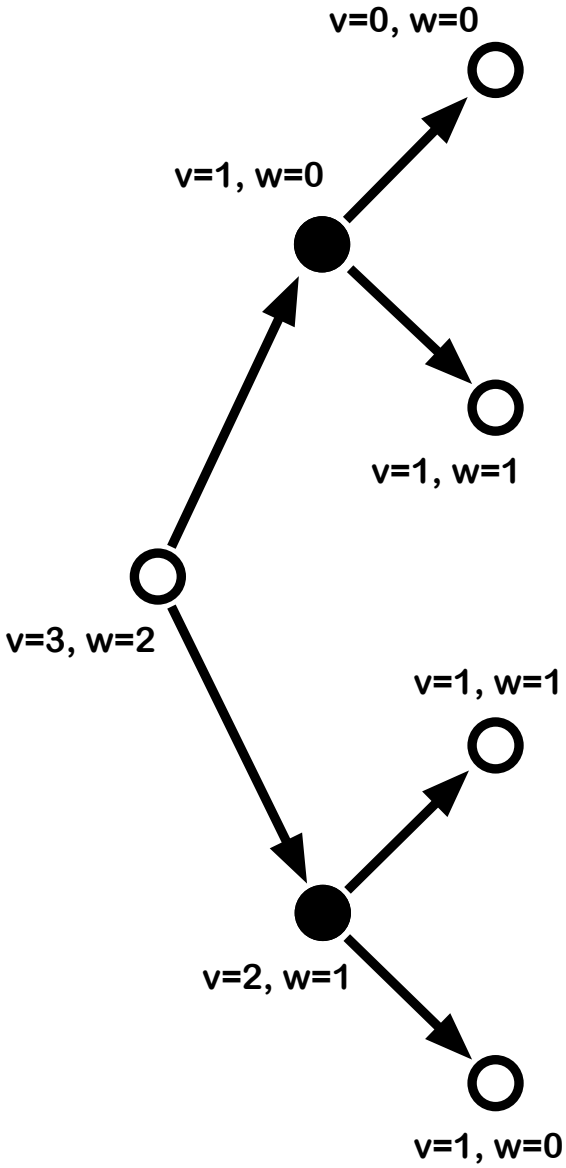
Rollout

- After expanding a new node, MCTS performs a **rollout** until the end of the game
- Actions during the rollout are chosen using a computationally cheap algorithm
- **Option 1** Random actions! Naive, but often effective
- **Option 2** Train a neural network and use it to propose actions

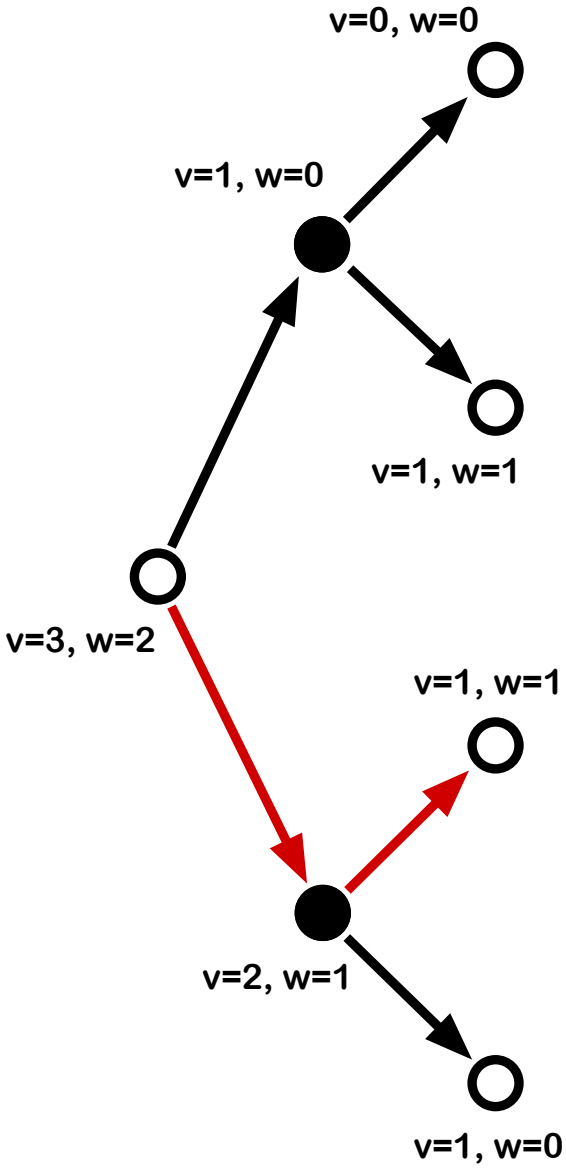
Propagation

- The rollout gives a result
- Backpropagate the result to the root of the tree and all the nodes in-between

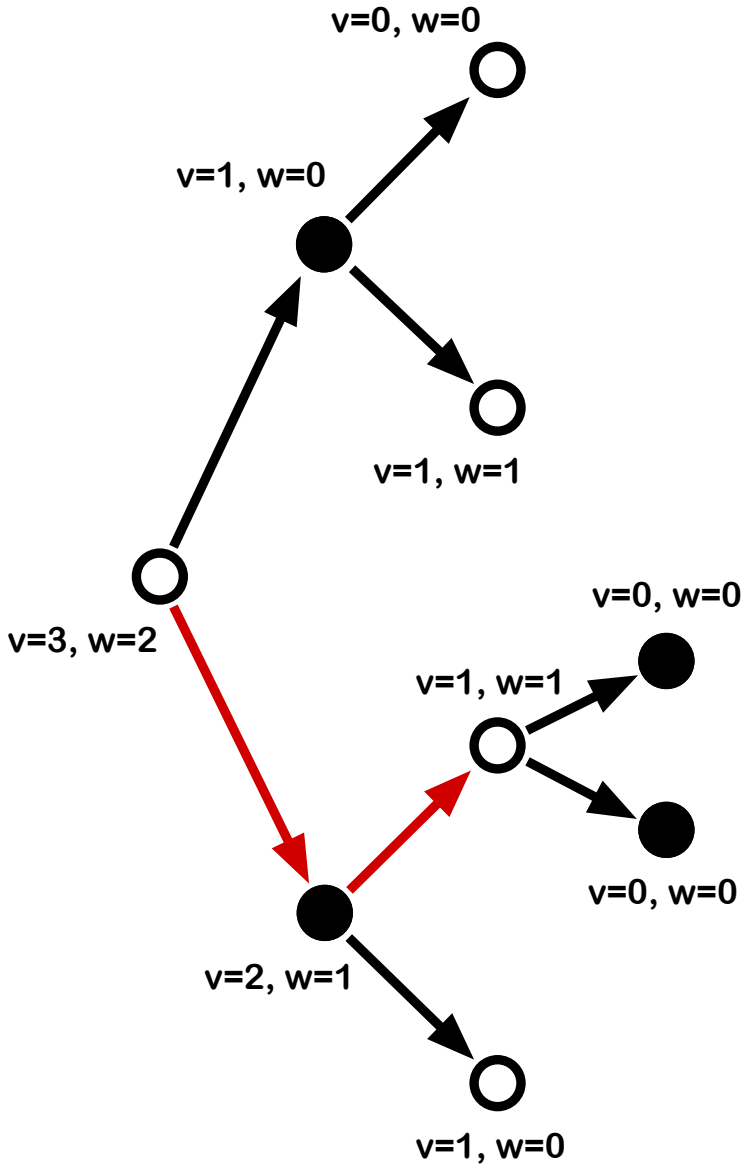
Example



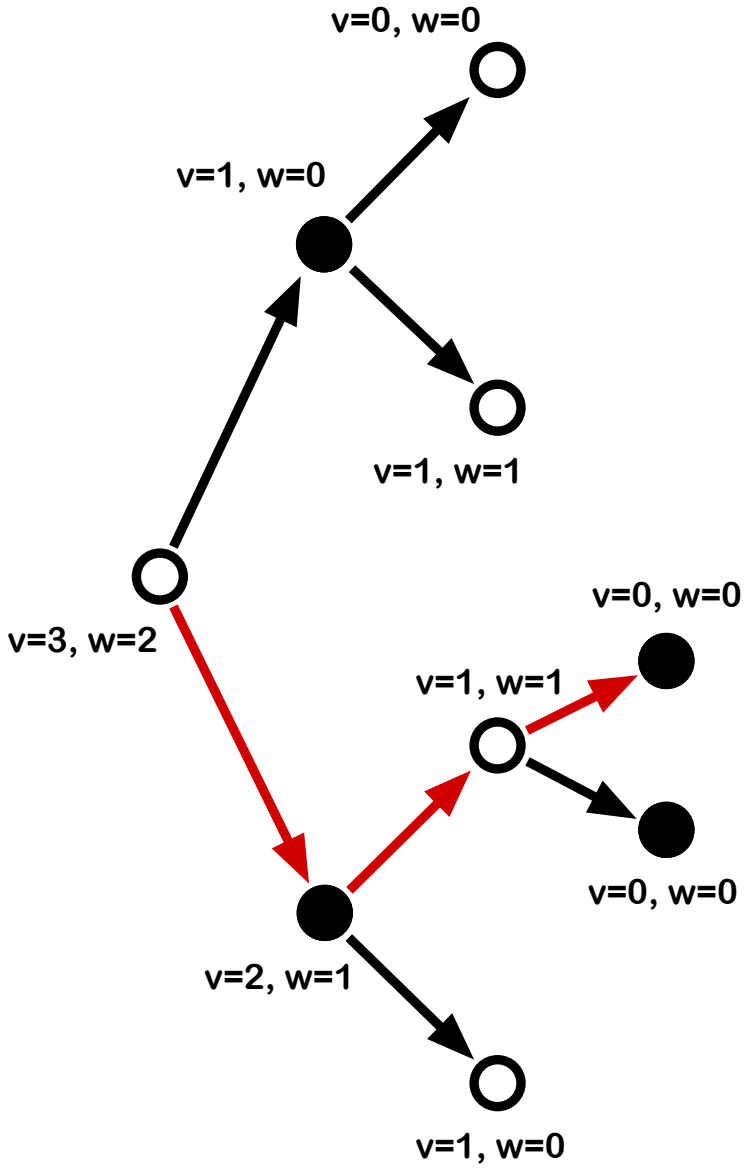
Example



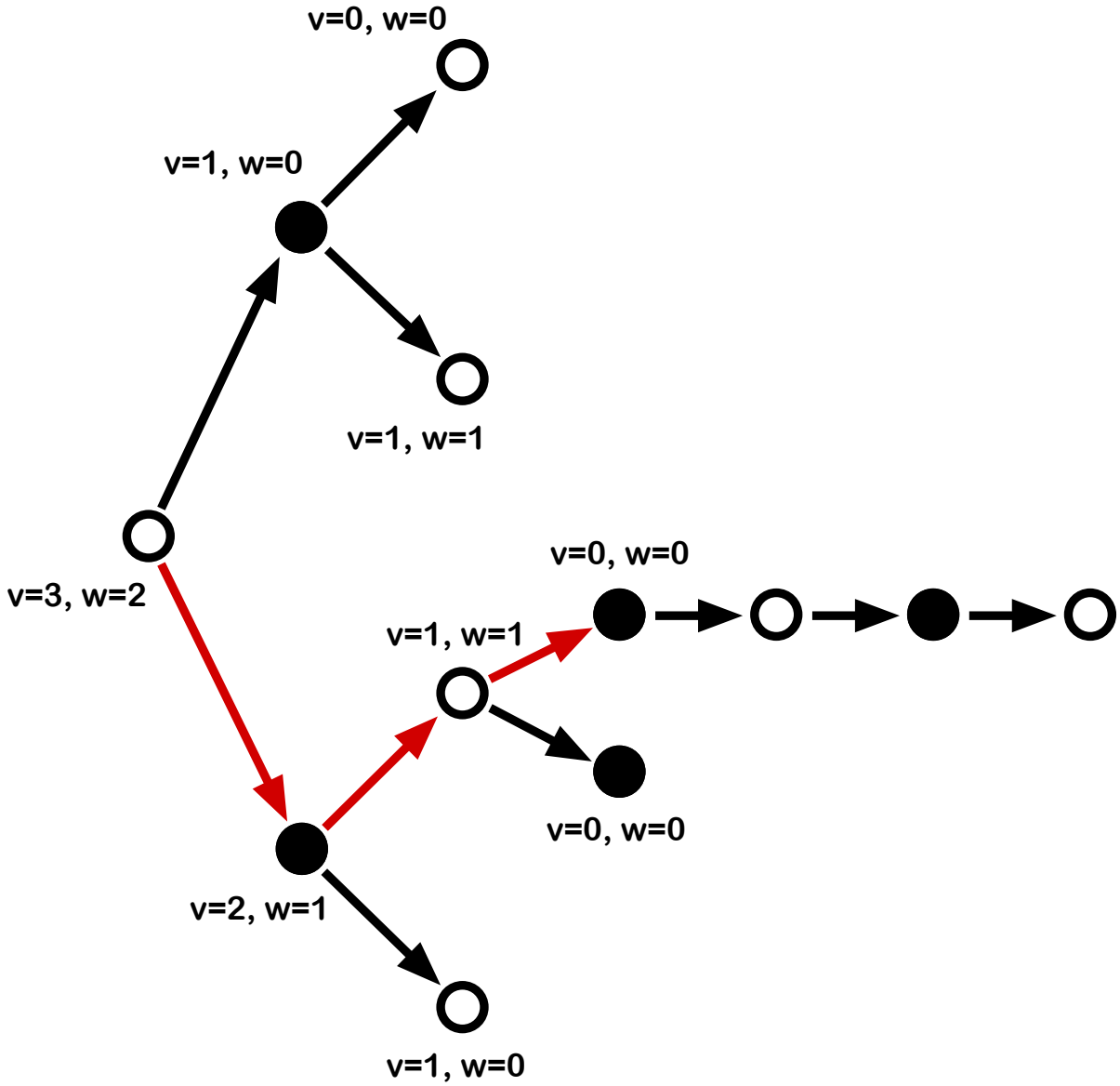
Example



Example

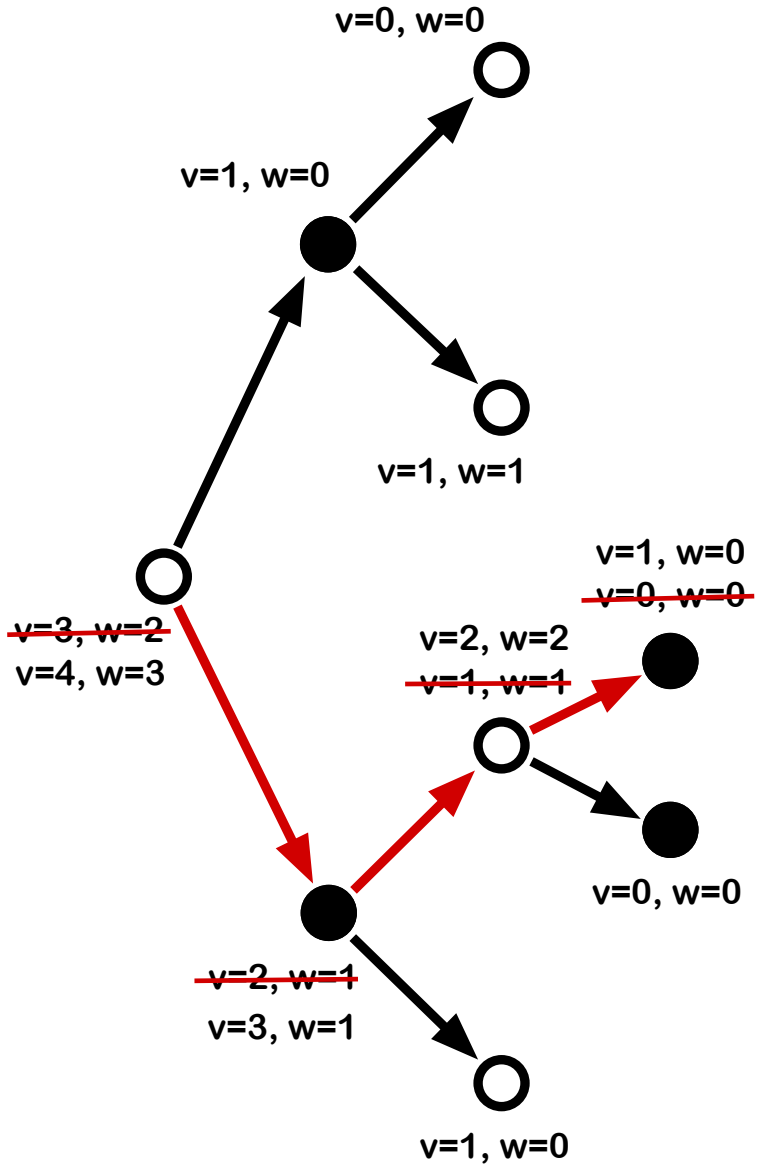


Example



black wins

Example



black wins

Using MCTS

- Now we have all the ingredients
- Each iteration the algorithm expands a node, performs a rollout and backpropegates the result
- Run as many iterations as you have time
- Play the action at the root given by

$$a = \arg\max_k \nu_k$$

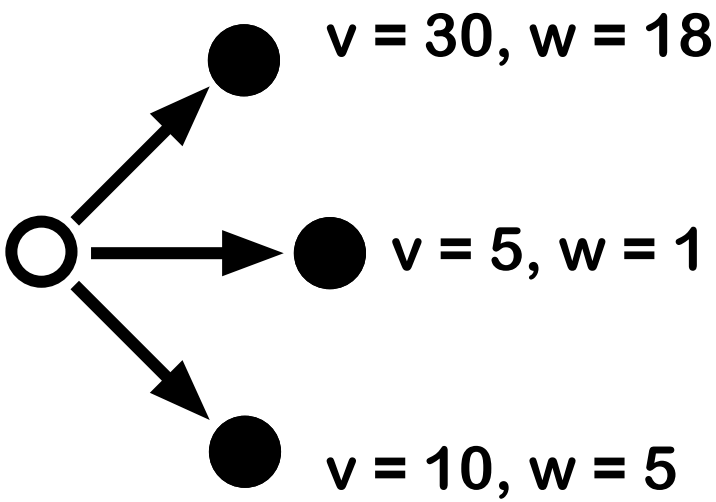
- **Alternative** sample your action randomly in proportion to ν or play the action with the largest average return

```
function search(board):
  root = Node(board)
  while time_left():
    do_iteration(root)
  return select_move(root)

function do_iteration(node):
  while not node.is_leaf():
    node = node.select_best_child()
  node.expand()
  node = node.select_best_child()
  value = rollout(board)
  while not node.is_root():
    node.visits += 1
    node.wins += value
```

Gotchas and tips

Store/use the win counts with the correct sign!




```
class Node
  parent        # parent of this node
  children      # children of this node
  board         # board state

  visits        # number of visits
  wins          # number of wins
```

Select child node that maximises

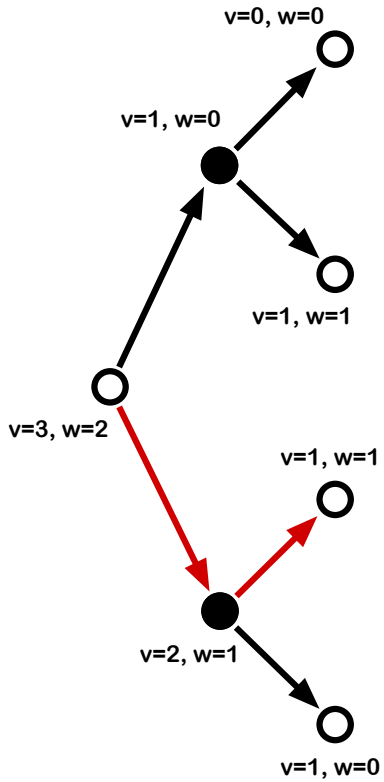
$$\arg\max_k \frac{v_k}{w_k} + c\sqrt{\frac{\log(\sum_j v_j)}{v_k}}$$

Win counts are with respect to the white player, even though we are storing the counts in nodes associated with the black player!

Gotchas and tips

You can’t expand the tree beyond the end of the game!

If you get to a node that is a terminal node, you can just propagate the result back directly. No rollout required.



Flexiblity

Core algorithm can be configured by changing the components

- Replace the selection function
- Modify the way rollouts are performed
- Mix rollouts with evaluation
- Change how the move at the root is selected

All components can make use of modern ML techniques

First, any questions about the algorithm

Learning a rollout policy

- A rollout policy is a function f_θ : states $\rightarrow$ distributions over actions
- The parameter θ determines the policy
- Usually f_θ is a neural network
- Need to choose input representation (how to represent the state?)
- Need to choose a neural network architecture

Learning a rollout policy

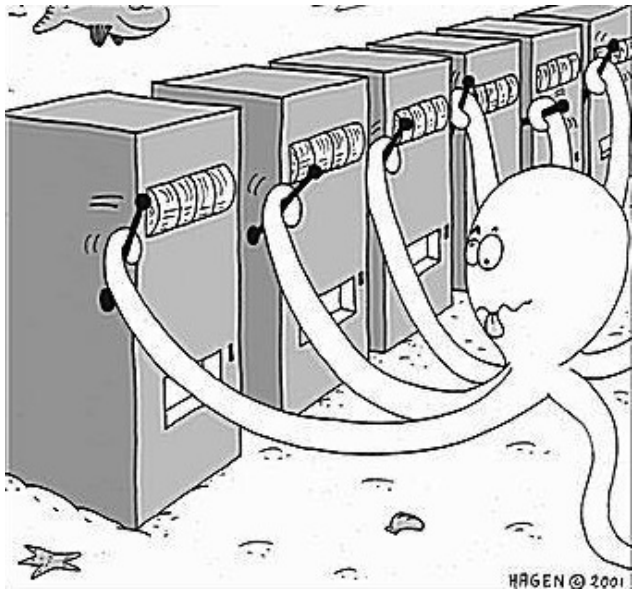
- What data do we use to learn the weights?
- **Option 1** Use human expert data, $\mathcal{D} = \{\text{state}_k, \text{action}_k\}_{k=1}^{n}$
- What if there is no human data? What if no humans are experts?
- **Option 2** Generate your own data with **self play**

Self play

- Start with an untrained network
- Use MCTS playing against itself to generate n games
- Use the moves played by MCTS to train the network
- Repeat for many many iterations

Bandit problems

- The algorithm used to select nodes is called a **bandit algorithm**
- A framework for modelling decision-making in the face of uncertainty



Applications

A/B testing A company wants to test whether version A or B of a website generates more sales. Users arrive sequentially and are allocated a website using a bandit algorithm

Drug evaluation A pharmecutical company wants to evaluate whether or not a new drug is effective relative to a placebo. Patients arrive sequentially and are allocated either the placebo or the treatment

Traffic routing A driver wants to get from point A to B, but which route should they take?

Advertising placement Company X wants to decide which add to show user Y

Simplified reinforcement learning Bandits have been useful to study RL in a simplified setting

Formal setup and optimistic algorithm

- A k-armed bandit is defined by **unknown** probability distributions $P_1, \ldots, P_k$
- The learner interacts with the bandit for n rounds
- In round t the learner chooses an action in $\{1, 2, \ldots, k\}$
- Observe a reward X_t sampled from distribution P_{A_t}

Upper Confidence Bound Algorithm

1. Play each action once
2. Subsequently in round t play action

$$A_t = \underset{a \in \{1,\ldots,k\}}{\arg\max}\ \hat{\mu}_k(t-1) + \sqrt{\frac{2\log(t)}{T_k(t-1)}}$$

$\hat{\mu}_k(t-1)$ is the average reward received playing action k

$T_k(t-1)$ is the number of times you played action k

Pre-work for lab

- Make sure you have Python3 and numpy
- Familiarise yourself with coordinate-wise operations on numpy arrays. E.g., square roots, argmax
- Look up the rules of Connect4

Questions to think about

- The minimax algorithm needs to visit all states of the game tree, but do you need to store the whole game tree? Try to come up with a depth-first implementation that uses memory that is linear in the depth
- We used a bandit algorithm when deciding which actions to expand towards the leaves. What would happened if you chose leaves at random. Would the algorithm work? Total failure? Or just learn more slowly?
- Bandit algorithms mostly assume that the reward of an action today is the same tomorrow (iid rewards for a fixed action). Does the application in MCTS satisfy this assumption?
- Why is it important that the game is zero sum? What about more than two players? Think about the richer effects that appear in these more general games. Can you think of real life (beyond games) examples?

Further resources

- Original MCTS paper: Kocsis & Szepesvári, “Bandit based Monte-Carlo Planning”
- AlphaZero paper: Silver et al.. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”
- Bandit algorithms: http://banditalgs.com