

CMPUT 396: Monte Carlo Tree Search

AlphaGo was an amazing accomplishment, but it benefitted from two ideas that came about at around the same time:

- 1) From image classification domain: “is this a picture of a snowball or a kitten?”. Humans were better at this task until about 2011. Geoffrey Hinton at the U of T was behind an idea with convolutional neural nets, “if you make them deeper, they will be stronger”.
- 2) MCTS: the first Go program that used this idea (Crazystone by Rémi Coulom) had a huge jump in performance

The Basics

There are *many* different versions of MCTS, but we will look at a simple one in this class.

Also note that the field of research around MCTS is very active so there is lots of new research all the time.

Monte Carlo Tree Search (MCTS):

*It is an **anytime** algorithm.*

This means that it returns a meaningful result after any amount of time. Basically, you can let it run any amount of time, say stop, and ask for the *recommended* move so far. We could let it run a few seconds or a few hours. Probably if you let it run for longer you will get a “better” answer, but you can get an answer whenever you want it which is nice.

This is in contrast to something like depth-d minimax, which *cannot* return a result until the search to depth-d is complete. You could have a truncated version of minimax, say you only go to depth 4 and then use a heuristic to estimate lower depths. But still, this is *not* an anytime algorithm because you still need to get to depth 4 and solve the leaves before continuing.

*It is a (mostly) **best first** algorithm.*

When expanding the search tree, it expands the most promising lines first. Basically, when I am at a certain node and I have several children, I go to the child that looks the best.

MCTS search is highly non-uniform: at any fixed level, some subtrees will be much larger than others (unlike say depth-d minimax, where for each node at each level, each subtree is the same size).

With MCTS, we grow the search tree one iteration at a time.

While time remains:

- a) Select a leaf
- b) Expand the leaf: find some children of the current node, at least one
- c) Simulate the result: evaluate the child with a simulation or rollout, the most traditional/simple approach is to play randomly and record the winner.
- d) Backpropagate the result

Return the best move (e.g. by most visits or best winrate).

A **visit** is an experiment to see who wins at that node.

When the while loop finished, we can see that the tree that was one node has grown. How do we pick our moves now? If simulations were accurate, we should do minimax to figure out which move is best, **but** we're playing stochastically so there will be some errors. What works better is to take the average winrates (rather than minimax).

Pseudocode

```
class Node:
    def __init__(self, m, p): # move is from parent to node
        self.move, self.parent, self.children = m, p, []
        self.wins, self.visits = 0, 0

    def expand_node(self, state):
        if not terminal(state):
            for each non-isomorphic legal move m of state:
                nc = Node(m, self) # new child node
                self.children.append(nc)

    def update(self, r):
        self.visits += 1
        if r==win:
            self.wins += 1

    def is_leaf(self):
        return len(self.children)==0

    def has_parent(self):
        return self.parent is not None

def mcts(state):
    root_node = Node(None, None)
    while time remains:
```

```

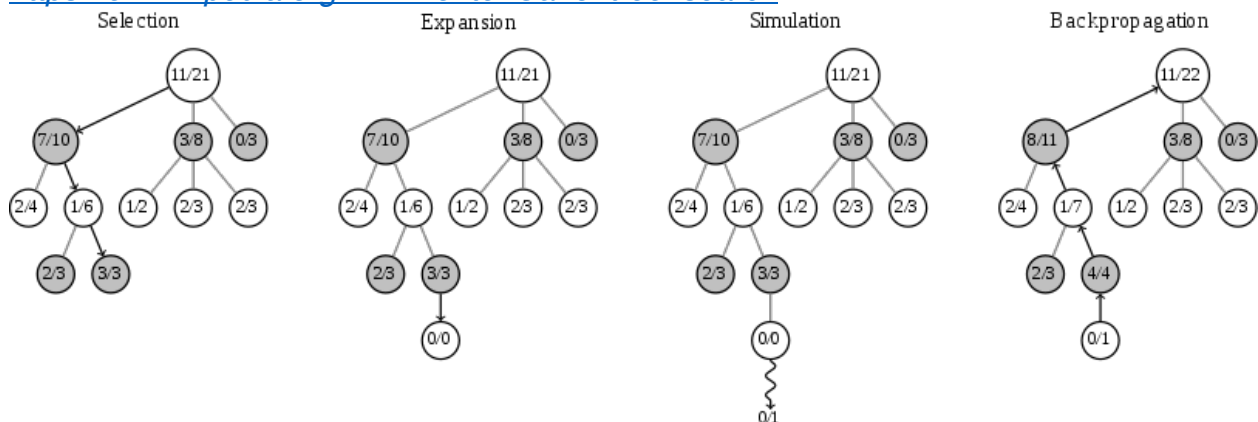
n, s = root_node, copy.deepcopy(state)
while not n.is_leaf(): # select leaf
    n = tree_policy_child(n)
    s.addmove(n.move)
n.expand_node(s) # expand
n = tree_policy_child(n)
while not terminal(s): # simulate
    s = simulation_policy_child(s)
result = evaluate(s)
while n.has_parent(): # propagate
    n.update(result)
    n = n.parent

```

```
return best_move(tree)
```

Example from Wikipedia:

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search



Assume all win rates are from the perspective of the player whose turn it is at that node (so the white nodes are white's win rate at that node and the black nodes are black's win rate at that node).

Selection:

- Say we're black. We start by choosing a leaf from the root (11/21). But which one?
- Using a best-first approach, we should choose (7/10) because that means that I, the black player, have won 7 out of 10 times we visited that node.
- From this node, we'll select the white (1/6) node (this is better for me because I'm black, the node where white has lost the most is the best for black).
- From this node, I go to the one where black has the highest win rate (3/3), again, because I'm black so a higher win rate is better for me on a black node.
- *In some ways this feels like minimax because I'm maximizing the win rate on my black nodes and minimizing the win rate on the opponent's white nodes.*

Expansion:

- Add a new node to the tree under this leaf node that we've selected (it's a white node since it will be white's turn to play at that level).

Simulation:

- Random/pseudo-random playout of the game, we see that white loses.
- In this simple version of MCTS, each move in the playout is uniform random over all legal moves.

Back-propagation:

- Update white's win rate to be (0/1) on our new leaf node and back propagate the results (remember to keep track of whose turn it is, so the 3/3 win rate on the black node becomes 4/4 because white lost at the last step, which means that for the parent (black) it was a win).
- Update the rest of the win rates for all the parent nodes all the way up the tree to the root node.

We can loop as many times as we want and stop whenever we need to. When we stop, we know that we have the best move *so far*.

What happens at the **selection** step if there are many best children that are equal (for example, we have multiple nodes that have the best win rate)? We can simply do a random pick of any best child.

In the last example, we used the win rate as our criteria for the **selection** step to choose the best child, but there are actually two criteria we could consider:

- 1) Ratio of wins to visits (win rate)
- 2) Number of visits

Why would you not pick the move with the best win rate?

Let's say that in our previous example (after the back propagation stage), we had another node that had 2000 visits but the same ratio of wins to visits (win rate) of 8/11 (the current best black node)? Also, let's consider earlier on in the process when we had 0 visits for one node. How do we know what the win rate of a node with 0 wins and 0 visits if we can't divide by zero (the number of visits)?

There's a lot of noise in this evaluation process so we need to consider the error probability (the probability of making a wrong decision about what the best child node is). The error probability drops with more visits (so we can be more sure that the best child node is really the best child node the more we visit it).

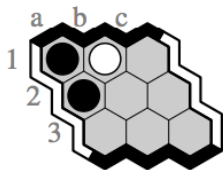
Solution: use a simple priority formula

- Replace the ratio of **wins/visits** with some function that gives unvisited nodes a score of 0.5.
- For example, pick $t = 1$ and set $f(w, v) = \frac{w+t}{v+2t}$, where **w** is the number of wins and **v** is the number of visits.
 - o Now when $w = 0, v = 0$, we have $f(0,0) = \frac{0+1}{0+2(1)} = \frac{1}{2} = 0.5$ (our “unvisited node score”)
- At even depths (my turn), pick random node in $\{ \operatorname{argmax}(f(w, v)) \}$ (to maximize the win rate at the nodes where it’s my turn)
- At odd depths (my opponent’s turn), pick random node in $\{ \operatorname{argmin}(f(w, v,)) \}$ (to minimize the win rate at the nodes where it’s my opponent’s turn)
 - o Wow doesn’t this look at lot like minimax?

Remember this problem and this ratio selection function because we’re going to come back to some of the same ideas later.

Example with Hex:

<http://webdocs.cs.ualberta.ca/~hayward/396/mcts.pdf>



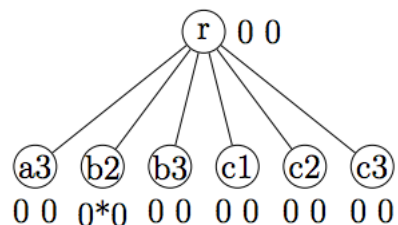
We start from this Hex position with white to play. From the root, there are 6 possible positions we could play at (a3, b2, b3, c1, c2, c3).

These 6 positions are our child nodes from the root.

Iteration 1:

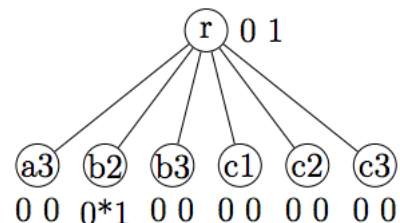
Select: select the root because it’s our only leaf (r) 0 0

Expand: expand from the root with the six possible moves we have and pick the best child (at this point they’re all equal with 0 wins and 0 visits each so it doesn’t matter which one we pick). Let’s choose **b2**.



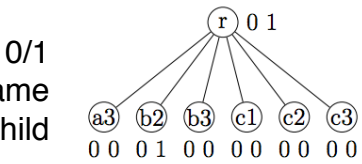
Simulate: from **b2**, play randomly (uniform random over all the possible legal moves). Say the play continues like this: **black at c1, white at c3, black at a3**, black has a path through **a1, a2, a3**, so **black wins**.

Back-propagate: the six leaf nodes represent possible white moves from this position so this becomes **0 wins** and **1 visit** because with the random playout we pursued white lost, so playing at this node was not a great move for white, the root player.

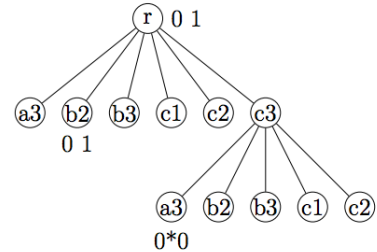


Iteration 2: (from now on, unlabelled nodes are 0 0, or 0 wins and 0 visits)

Select: which is the best child now? We have a win rate of 0/1 for **b2**, which is not great, but the rest of the leaves are the same with a win rate of 0/0. We'll pick randomly from these equal child nodes, say **c3**.

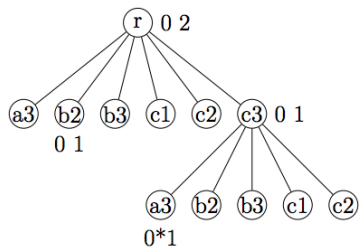


Expand: get all children from **c3** (all the legal moves from this board position) and pick one randomly (because they all have the same win rate), say **a3**.



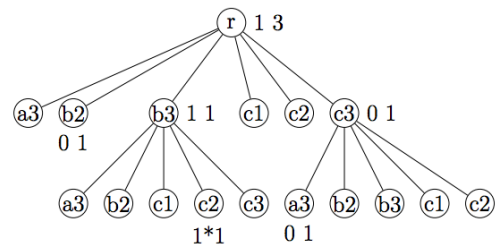
Simulate: we simulate from the board position **root – c3 – a3** with a random play out. Say black wins.

Back propagate: we're thinking of all of these nodes from the player-to-move's perspective (white's perspective), so we'll update **a3** with 0 1, because white lost in this play out, and update the parent with the same.



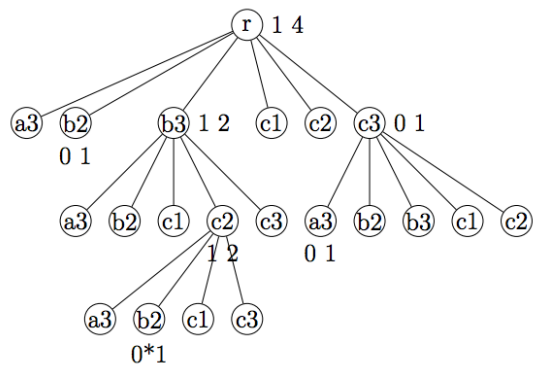
Iteration 3: (we're going to simplify a bit from this point, check the handout for a more complete traversal)

We ended up with two nodes with 0 wins for 1 visit (not a great win rate), so we will choose from the other four nodes in our selection (a3, b3, c1, c2). Say we choose b3, expand it and choose a random child, say c2. After our random play out say that white wins. We back-propagate our results so c2 at the lowest level and the parent, b3 both have a win rate of 1/1.



Iteration 4:

We have two nodes with a win rate of 0/1 (b2 and c3), which is not so great, and one node with a win rate of 1/1 (b3), which is really great (100%!). Since b3 is the best looking child we'll select it, then choose c2 (again, a win rate of 1/1), and expand from this point, choosing b2 as our node to simulate from. Let's say black wins and we back propagate all the results.



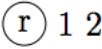
We're starting to see some of the issues we saw in the Wikipedia example. Namely, how do we compute the win rate of a node with no visits (we can't divide by zero to get the ratio in this case)?

We will prefer a win rate of 0/0 over 0/1 because nothing could be worse than losing all simulations. Similarly, we prefer 1/1 over 0/0 because nothing could be better than winning all simulations. So how do we implement this?

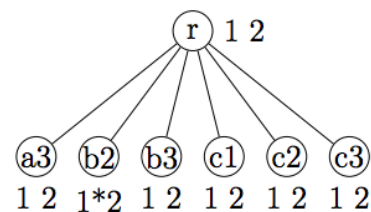
One way is similar to the simple priority formula as we used earlier. Rather than initializing all new nodes with 0 wins and 0 visits (which is true), we'll give each node a fake seed where the win rate is 50%, so the win rate is the ratio of T wins to $2T$ visits for some integer T . For example, we'll give each new node a seed value of 2 visits and 1 win, for a win rate of $\frac{1}{2}$ or 50%.

How would this look if we re-do MCTS on the same starting point as before?

Iteration 1:

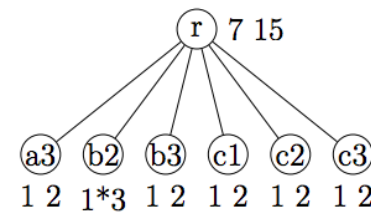
Select: again, we select the root because that's all we have, but this time we start with 2 visits and 1 win. 

Expand: we expand on the root with the six children (the six possible legal moves) and initialize each with the seed value of 2 visits and 1 win.



Simulate: from b2, our selected node, simulate a random playout, say black wins.

Back propagate: when we back propagate we need to back propagate *all* the real and fake visits to the parent. The child visits are now $2 + 3 + 2 + 2 + 2 + 2 = 13$ plus the 2 from the visits on the root, the total visits for the root is 15. Likewise, for the wins we can add the child wins, $1 + 1 + 1 + 1 + 1 + 1 = 6$ plus the 1 win from the root, which is 7 (this is how we get the 7/15)



Note that we have lost the property where the number of visits (or wins) of the parent value is equal to the sum of the number of visits (or wins) to the children (we're saying the parent has 15 visits but the children have a sum of 13 visits because we have the initial fake 2 visits). It doesn't really matter, we can forget about it, but if you notice this difference between the two different versions, this is why.

The handout goes through another 3 iterations but it's basically the same as the above.

So far, what we've done is a purely best-first approach (aside from the random selection of the leaves). However, because the leaves are evaluated stochastically, there is a problem with this approach.

However, I might be unlucky. If my first visit to a node A is a loss and my first visit to a node B is a win, I will never go back to node A. But how can I actually trust these win rates if I only visited node A once?

I want a stochastically significant number of visits to each node so that I can actually trust my win rates. This is what we will look at in our MCTS improvements (specifically in the exploration section to come).

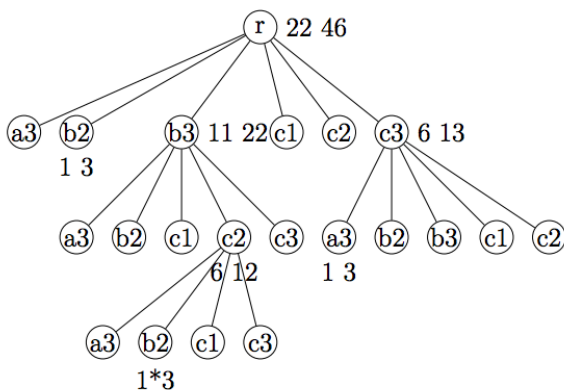
MCTS Improvements

How can we improve the version of MCTS that we've talked about so far? One way could be to not use random rollouts and use a heuristic instead (do the right thing when you know/think you can).

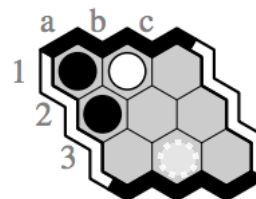
Let's go back to the MCTS Hex example. One of the "Questions to think about" at the end of the handout asks:

*How close is the current tree to finding the best move?
Or to finding the correct win rate?*

At the end of Iteration 4,
We end up with this for the tree:



And on the board it looks like this
(with the suggestion to play at **b3**):



If we stopped here and asked what move we should take, what should we say? Remember we can choose from two move selection criteria: win rate or number of visits.

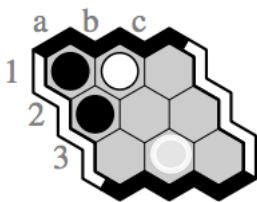
Win rate: **a3**, **b3**, **c1**, and **c2** are tied for the highest win rate of 50%
Visits: **b3** is the highest with 22 visits

If we're looking at the intersection of both groups (if there is overlap between both criteria it's probably a good bet), we can see that **b3** has both the highest win rate *and* the highest number of visits. Great! So we'll play at **b3**.

When Aja Huang was working on Hayward's Hex program, he added an "expand on unstable search" feature which worked basically the same way. Previously in their Hex program, they only expanded on visits. In this new feature, they checked which nodes had the highest win rate *and* what nodes had the most visits. If they agree, great! you have your move. If they don't agree, keep searching. This made their Hex program a lot stronger.

Going back to the example, based on our search, our suggestion is to play at **b3**. Let's look at the Hex board as a human for a second. Do we like this move?

The answer is no.



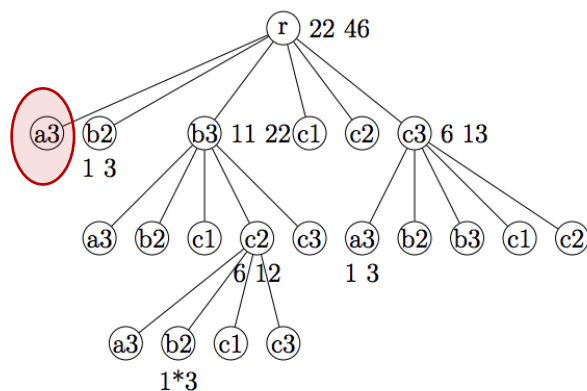
Say we do make this move as white, so we get a board like this.

What is black's next move? Black can play at **a3** and win the game.

This means that for white there is a must-play region of size 1 at **a3**. If we don't play here on this move, black will play here on their next move and win the game.

What happened in our search? How did we miss this? Why didn't we realize that we should play at **a3**?

Let's look back at our tree:



If we take a look at move **a3** on our tree, we can see that we never did any simulations on this move, it's a really small tree. It still has the win rate and visits from our initialization.

This shows us that we need a lot more nodes in our tree. Clearly this didn't cut it. Ideally, you want a few thousand nodes in your tree. If we had this number of nodes in our tree, we would have seen that we would lose anywhere other than at **a3**.

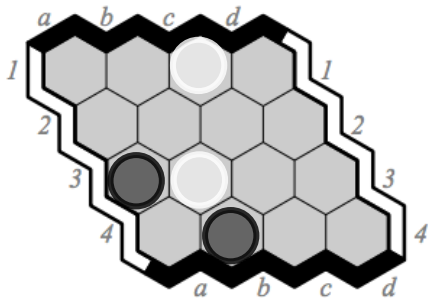
So to answer the first question: No, our tree is not close to finding the best move.

What about the second question? How close are we to the correct win rate? For **b3**, for example, based on our win rate of 50%, it looks like we have pretty good odds. However, the actual win rate is zero. So again, we're not close.

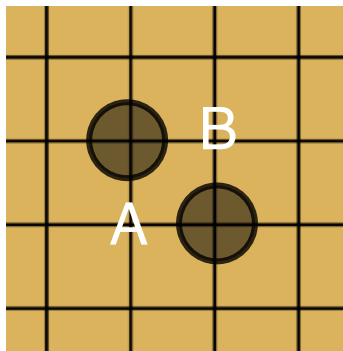
Going back to our original topic of discussion, improvements to MCTS, we have found our first method of improvement, improve simulations through doing the right thing in obvious situations, like we did here by realizing that there was a must-play region at **a3** that we need to play in.

Another way to improve random simulations is through looking for *patterns* to help us choose which moves to take in our simulations.

Based on this position with black to play, what should black do?



Suppose we're in a Go simulation and we have some stones in this position on the board:



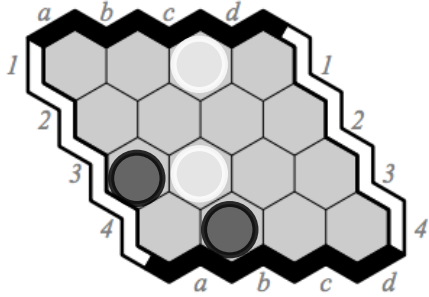
This is a well-known pattern in Go. These stones are said to be *loosely connected*. If black ever see this pattern in a simulation, we should make a note of it. If white ever plays in one of the two corners (A or B), threatening the loose connection, black should play in the other corner to connect the stones. Otherwise, black should make a random move.

This is actually how many Go solvers work: play randomly until a loose connection is threatened and then protect it.

This is called the **save-miai** simulation pattern. *Miai* is a Japanese term that in Go refers to a situation where there are two different options such that, if one player takes one (e.g. A), the other player can take the other (e.g. B). Also, it typically does not matter which player gets which option, but sometimes each player has only one of the options.

The same idea applies in Hex. If a virtual connection is ever threatened, we should save it. If not, we'll play randomly.

Going back to our earlier Hex example:



In this situation black should play at **a4** because white threatened black's virtual connection with their move at **b3**.

This analog to the save-miai pattern in Go, is called the **save-bridge** simulation pattern in Hex.

Improving simulations by taking the save-bridge approach before making random moves makes a Hex program significantly stronger.

MCTS Improvement: Exploration

MCTS simulations will have some randomness, so when we're selecting a leaf based on the results of our simulations, how can we account for variance in win rates? For example, if we have a *move j* with 52 wins and 100 visits (a win rate of 52%) and a *move k* with 529 wins and 1000 visits (a slightly higher win rate of 52.9%), are we sure that *move k* is better than *move j*?

One way we approached this earlier was with the ratio selection function: $f(w, v) = \frac{w+t}{v+2t}$

Another approach is to balance **exploitation**, where we pick the child with the best win rate, with **exploration**, pick the child that we have visited less. Exploitation is basically a greedy, best-first search (which is what the vanilla version of MCTS is), but exploration allows us to add some breadth to the search (remember the example earlier where we didn't pick the right move because we hadn't explored its subtree at all yet). We need to have a balance of both exploitation and exploration.

Imagine that you are faced with a row of slot machines with different payout probabilities and amounts. How can you play with a strategy to maximize your net gain? A good strategy would be to balance playing all of the machines with concentrating your plays on the best observed machine.

UCB1 is does this by constructing statistical *confidence intervals* for each machine:

$$\bar{x}_i \pm \sqrt{\frac{2 \ln n}{n_i}}$$

where:

- \bar{x}_i is the mean payout for machine i
- n_i is the number of plays of machine i
- n is the total number of plays (number of times slot pulled)
- i is the number of machines

(above is from <https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/>)

Say that you have five slot machines and you pulled a total of 1000 times ($n = 1000$, n_1 is how many times you pulled the lever for machine 1 for example). x_i is the win rate on machine i . Say my favourite machine is machine 5 so I pulled it 800 times and I won 200 times, so $x_5 = \frac{200}{800} = \frac{1}{4}$.

Note that this method assumes that slot machines behave differently and have different intrinsic win rates.

Say we only pulled the lever 5 times on machine 2 out of the total 1000 pulls. When comparing this to the 800 for machine 5, with this formula, machine 2 will have a much higher exploration value.

$$\text{Machine 2: } \sqrt{\frac{2 \ln 1000}{5}} \approx 1.66 \quad \text{vs.} \quad \text{Machine 5 } \sqrt{\frac{2 \ln 1000}{800}} \approx 0.22$$

Kocsis and Szepesvari came up with a formula to balance exploration and exploitation based on the UCB1 formula called UCT (Upper Confidence Bound 1 applied to trees) to address this. Basically it asks the question: "I like this move, but am I sure that it's better than the moves I haven't explored much yet?"

Pick move j that maximizes $f(w_j, v_j) + c \sqrt{\frac{\ln V}{v_j}}$

where:

- $f(w_j, v_j)$ is the win rate of the node
- c is some experimental constant to adjust the amount of exploration (incorporates the $\sqrt{2}$ from UCB1, usually for Hex and Go, you get good performance with a small c , like 0.001)
- $\sqrt{\frac{\ln V}{v_j}}$ is the exploration boost (from UCB1)
 - o v_j is the number of visits to the node
 - o V is the total number of visits to the parent node

In the class repo there is a small program **simple/mcts/ucb1.py** that you can use to calculate these values.

Why would I prefer to expand a node with a lousy win rate compared to one with a really high win rate? Exploration! If the node that has a lousy win rate has a low number of visits we probably prefer to explore there rather than exploit the node with the highest win rate (maybe we've just had bad luck in our simulations so far in the few visits we've had).

MCTS Improvement: RAVE

MCTS RAVE (Rapid Action Value Estimate) is a statistical technique created by Sylvain Gelly (MoGo team) and David Silver that can be used to bias selection. In uniform random playouts, we can think of a playout as a sequence of uniform random moves *or* as uniform random subsets of black/white cells. If we think about them as groups of cells instead of specific moves, we can use the information from previous simulations when exploring similar positions.

Say we split the game space randomly so I get some and you get some, who wins? When we look at the board like this, there is no real notion of which move was the first move. Any move could be the first move, but it doesn't really matter. What is important is what the board looks like now. RAVE is like this, it uses an "all moves as first" heuristic.

We gather statistics to see which board positions correlate the most strongly with winning. The playouts, rather than being a sequence of moves, are a subset of moves. Through this we can see which cells positively correlate to wins and use this as a heuristic in future selection processes,

Basically we want to re-use data from other parts of the tree in this part of the tree (like a transposition table).

Last time this course was taught they talked about this idea a lot since it was before the neural net movement, but it's since been replaced by better ideas (like neural nets) so we don't need to be super familiar with all the details.

RAVE is effective if simulations have some randomness to them. However, if you have a good idea which children are good, you don't need RAVE. But, if you have no idea which children are better than others, RAVE is useful.

In the case of AlphaGo, for example, it has a really good child selection process using neural nets so it doesn't need RAVE.

MCTS Improvement: Prior Knowledge

After the MCTS expansion when we need to choose a child from the new child nodes, instead of picking a random child, use a **prior knowledge** heuristic to select the most promising node.

MCTS Improvement: Neural Nets

Use a neural net (policy net) and win rate for selection.

Use a neural net (policy net) for prior knowledge.

Use a neural net (value net) and MCTS for move selection.

What Improvements Popular Go/Hex Programs Used

- AlphaGo: Exploration, prior knowledge, **no RAVE**
- Patchi: Exploration, prior knowledge, RAVE
- Fuego: **No exploration**, prior knowledge, RAVE
- MoHex: Exploration, prior knowledge, RAVE