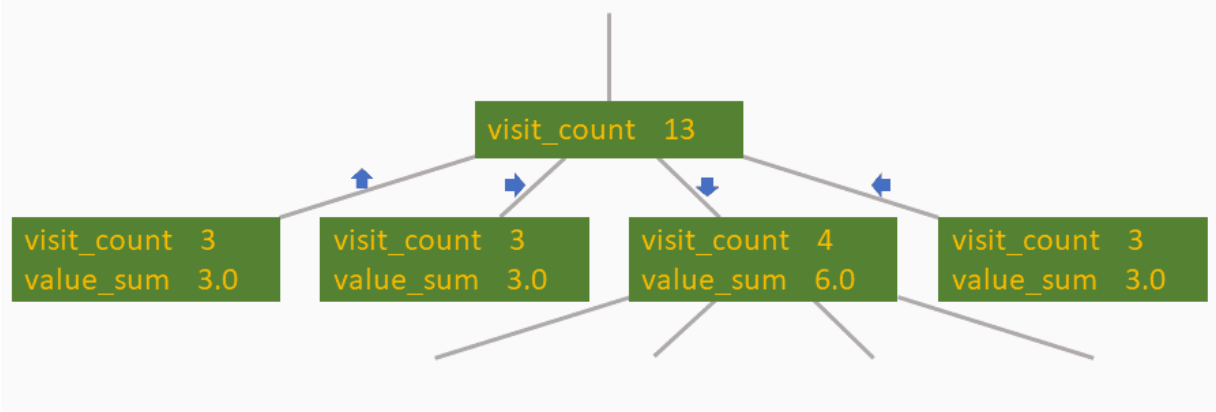



# The Animated Monte-Carlo Tree Search (MCTS)

The algorithm at the heart of AlphaGo, AlphaGo Zero, AlphaZero and MuZero

**Thomas Kurbiel**

Follow

TDS Archive a11y-light ~18 min read ·
January 31, 2025 (Updated: January 31, 2025) · Free: Yes

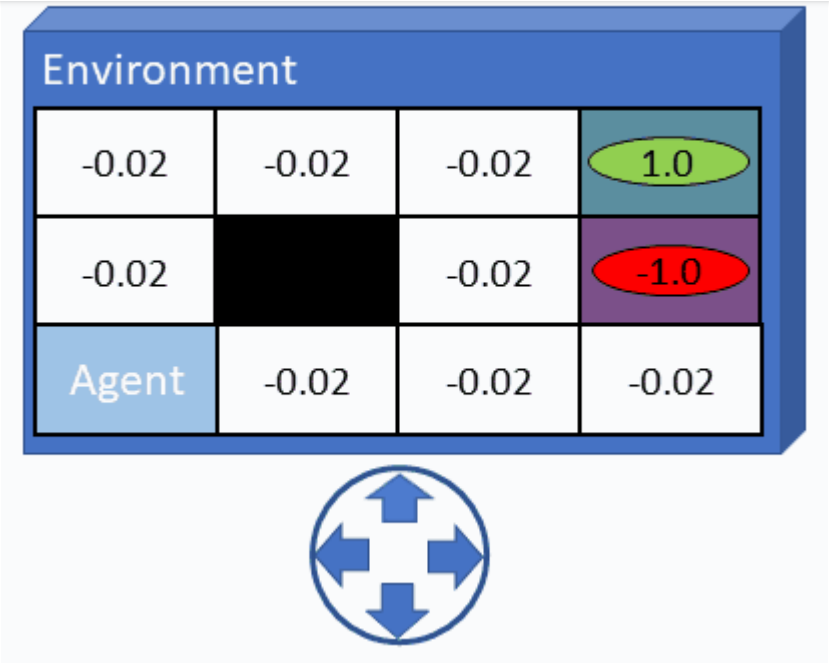
Monte-Carlo Tree Search (MCTS) is a heuristic search method, whose main purpose is to choose the most promising move, given the current state of a game. It does so through repeatedly playing the game starting from the current state, thus identifying promising moves and quickly abandoning the unpromising ones. As we will shortly see, the challenge lies in performing as little play-outs as possible, while looking ahead into the future as far as possible. MCTS is the key algorithm behind many major achievements in recent AI applications such as the impressive AlphaGo showdown in 2016, where AlphaGo outplayed the second-highest ranked Go player.

"exploitation-exploration trade-off". Due to the highly dynamic nature of the algorithm — the **search tree** grows and is updated on each iteration — it is best explained using animations, a lot of animations. This will help you to quickly develop a solid mental concept of MCTS, making it much easier to understand groundbreaking advances like the famous AlphaGo, where MCTS is combined with neural networks.

In this article, we will start with the motivation behind MCTS. Next, we will introduce all the single steps the algorithm is made of: **selection**, **expansion**, **simulation** and **backpropagation**. We take a particularly close look at the **backpropagation**, as it is neglected in many tutorials. Finally, we will take a very close look at the mechanism behind the "exploitation-exploration trade-off".

## Environment

We are going to use a deterministic version of the famous environment **grid world** to get familiar with MCTS. In this version the agent accurately obeys our commands without slipping to the sides randomly:



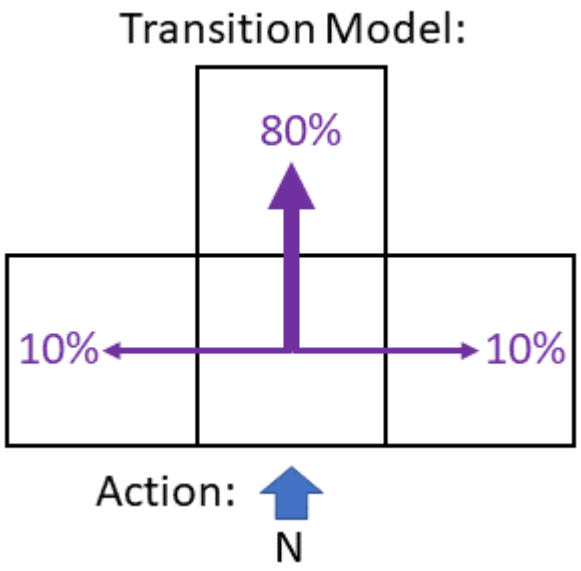

Deterministic grid world (Image by author)

As you can see from the animation, we can choose from the four different compass directions (North, West, South, East). There are two final states: a winner state with a reward of +1 and a loser state with a reward of -1. All other states return a small negative reward of -0.02, which discourages wandering around aimlessly and that encourages taking the shortest route. The black cell is an obstacle and the agent cannot go through this cell.

If the agent bounces off the wall, it just stays where it is, however collecting a negative reward of -0.02.

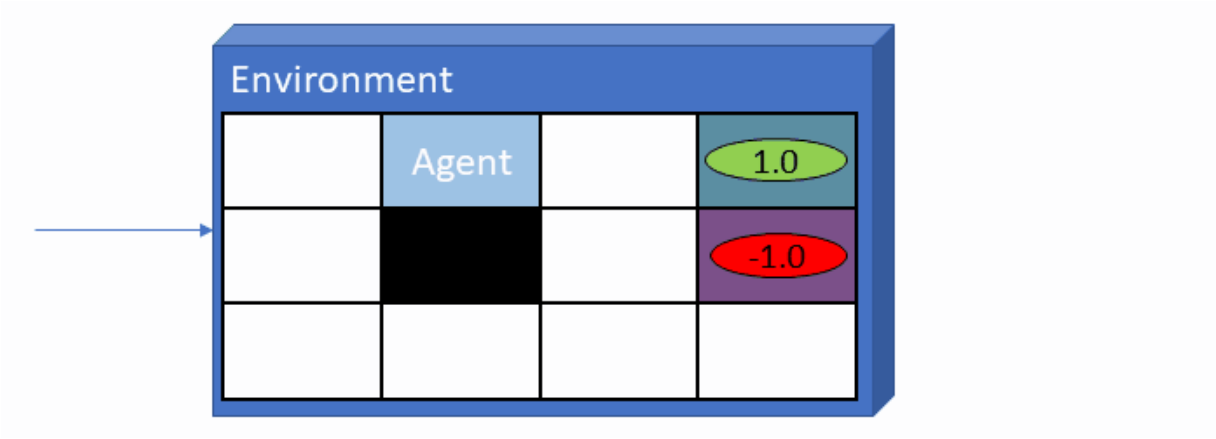

Agent bounces off the wall (Image by author)

At the end of this post we will extend the **grid world** to have stochastic dynamics: under these conditions, if we command the agent to go north, there is, for example, a 10% chance that it will accidentally gear off to the left and a 10% chance that it will gear off to the right, and only a 80% chance that it will manage to go the direction we commanded.




Stochastic Environments (Image by author)

we can perform multiple play-outs, starting from the same state.

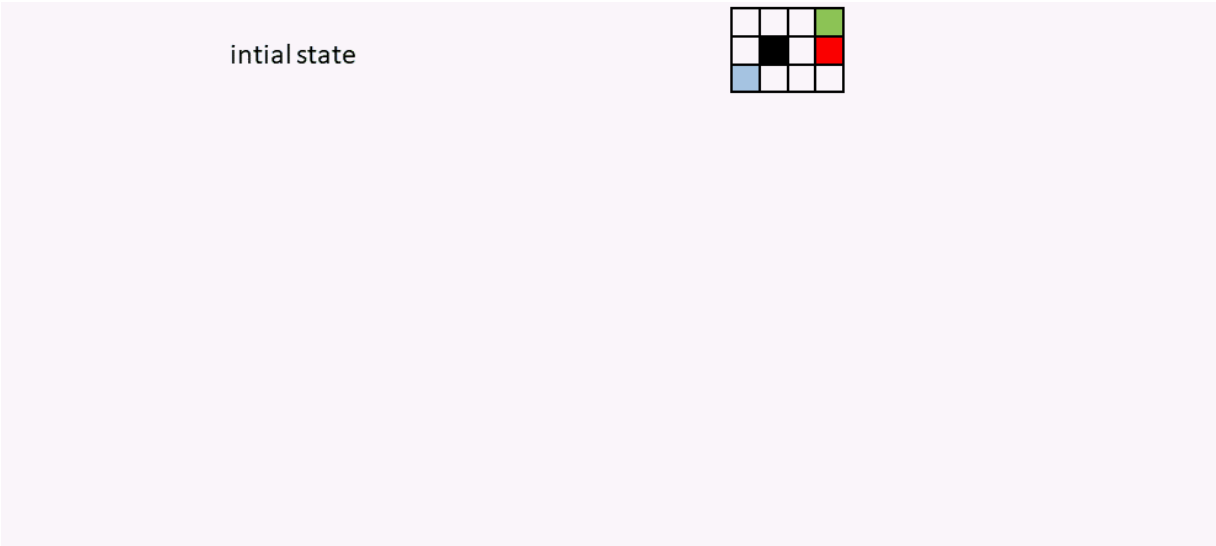



Storing and reloading of a state (Image by author)

As a side note: the **grid world** is a small example of what is called a Markov decision process (MDP).

## Motivation

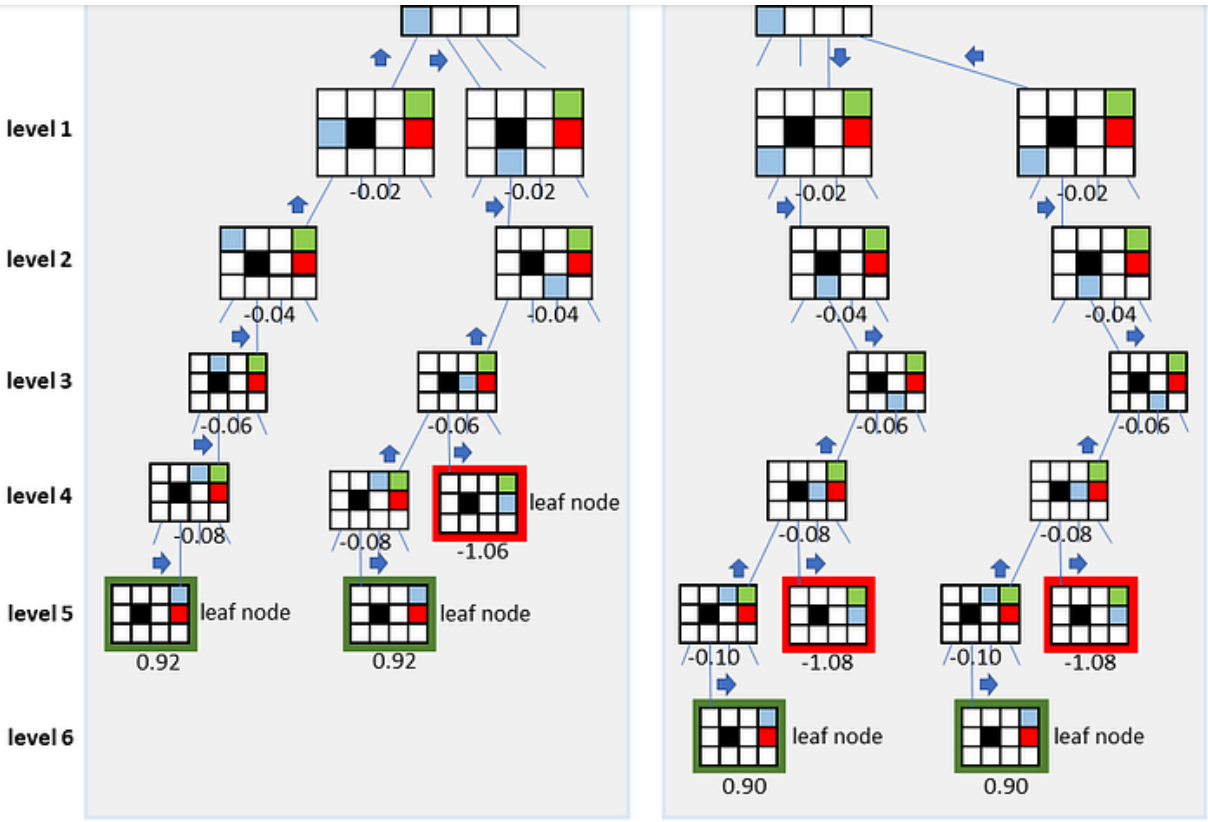
Our goal is to find the quickest path to the positive reward in the upper right corner of the **grid world.** We will first devise a simple brute-force algorithm, which systematically tries out all possible combinations of actions.



First notice, how the representation of all the possible future states naturally leads to a tree-like structure. Each node of the tree represents a particular state in the game, whereas edges represent different actions. The child nodes of each node hence represent all possible next states a player can reach from there. The terminal states of the game are represented by leaf nodes and do not possess any children. The node at the top is called root node.

Please notice the hierarchical organization of the tree: the root node belongs to **level 0**, its child nodes belong to **level 1** and their children in turn belong to **level 2** and so on. Moreover, each level represents a different future point in time: **level 0** represents the current step, **level 1** represents all possible future states after taking a single action, **level 2** represent all possible future states after taking two actions and so on.

Now, we notice an interesting thing: the **total rewards** (the numbers underneath each node) in all levels explored so far are identical. In particular, this means, that even after reaching **level 2**, we still cannot decide which action to take next. Thus, we need to expand the tree much deeper:




Shortest paths to the winner state after choosing an initial action (Image by author)

We see in the left part of the figure above, that by either taking the "up" or "right" action in the root state, we can reach the winner state in 5 steps. The maximal possible reward amounts to 0.92. Taking the "down" or "left" action, depicted on the right-hand side of the figure, involves at least one bump against a wall, only allowing the agent to reach the winner state at level 6 with the final reward reduced to 0.9.

We can conclude that expanding the tree in depth is crucial for proper decision-making. The flip side of the coin however is, that the amount of states we need to visit rises exponentially with each additional level:

- At level 1: 4
- At level 2: $4^2 = 16$
- At level 3: $4^3 = 64$
- At level 4: $4^4 = 256$

- ...
- At level 10: $4^{10} = 1.048.576$

The 4 used above is the so called **branching factor** and corresponds to the number of actions we can take in each state. Go has a branching factor of approximately 250 (!!!), whereas chess typically has around 30 actions to choose from. Expanding just 5 steps ahead in Go, leads us to an astronomical number of possible states: $250^5 = 976.562.500.000$.

Visiting each and every node of the search tree is impractical, especially since for proper decision making we must expand the tree in depth to look further ahead into the future. But how can we make the brute-force tree-search algorithm above feasible for more complex tasks?

The solution that MCTS pursues, is to run repeated simulations called **roll-outs** or **play-outs**. In each play-out, the game is played out to the very end by selecting moves at random. The final game result is used to quickly identify and abandon unpromising paths and only concentrate on the more promising ones.

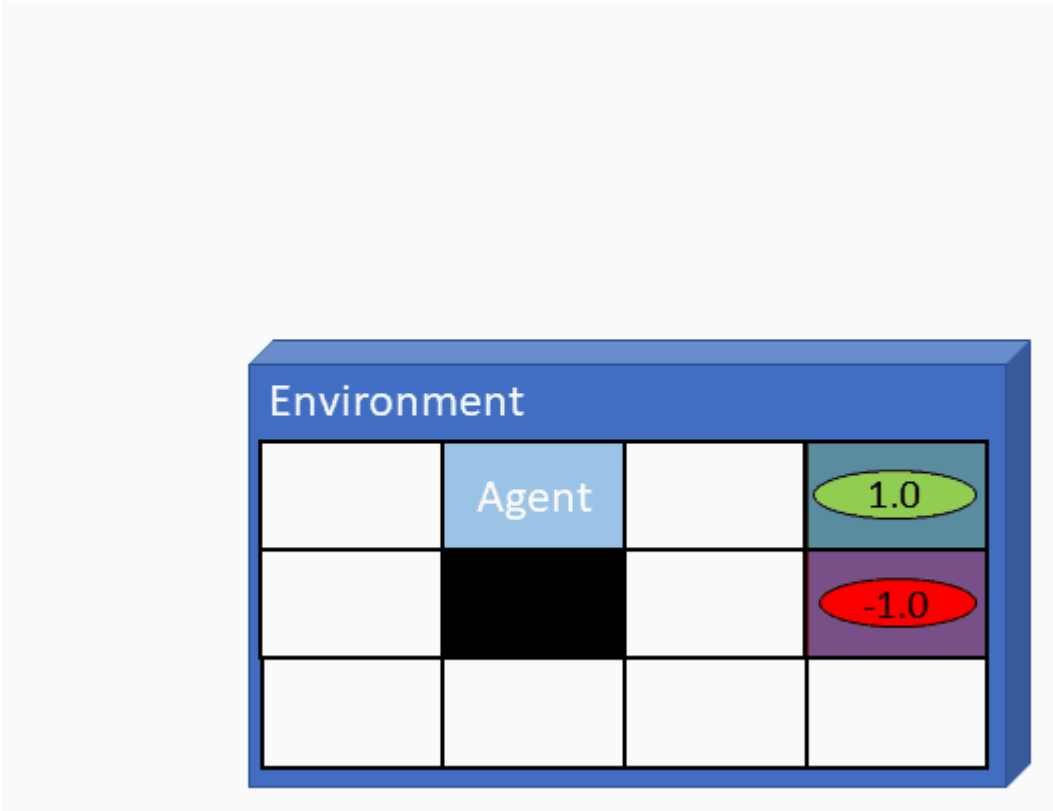
Random sampling of the search space (Image by author)

The total reward we obtain in the roll-out depicted above amounts to 0.62. Remember, that the shortest paths we explored before yielded a total reward of 0.92. We hence see a high variance in the outcomes of different simulations. In order to gain confidence about the statistics, we need to repeatedly runs simulations in the environment. This repeated random sampling of the search tree is what is implied by the name Monte-Carlo.

One last remark regarding the second phrase "tree-search" in MCTS: remember that the systematic traversal of a tree in order to find an optimal path is called: tree-search.

### Initialize the root-node

The first thing we do is to reset the environment in order to start from a controlled state. We than save the state (it is actually a checkpoint which fully describes the initial state of the environment). Subsequently, we create a new search tree node and append the just obtained state to it:




Initializing the root (Image by author)

As we have seen in the introduction, a search tree is a hierarchical structure, where each node can be connected to many children, but must be connected to exactly one parent, except for the root node, which has no parent. An abstract data type representing a node must hence keep a list with pointers to the child nodes and a pointer to the parent node. Other important member variables of the node structure are:

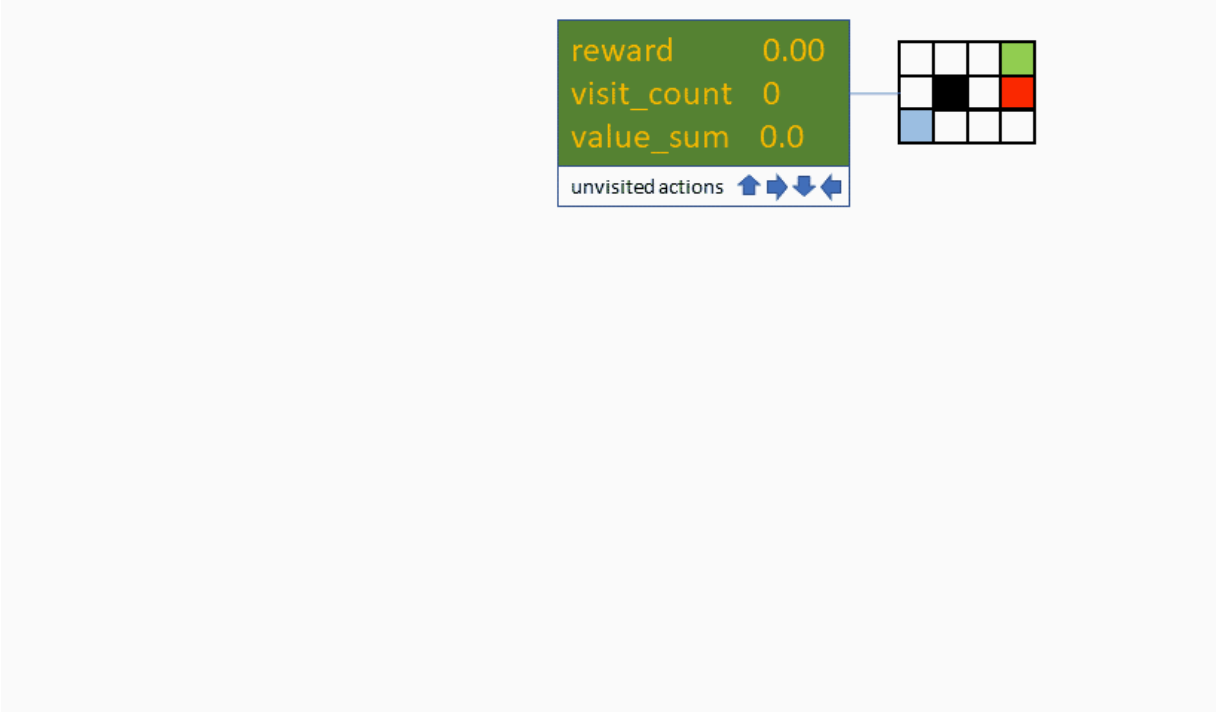
- **reward**: reward for taking action *a* in state *s*

- **visit_count:** how many play-outs contribute to value_sum
- **state:** checkpoint describing the full state of the environment

In addition we will keep a list of all yet unvisited actions and a boolean variable indicating whether the state is terminal or not.

### Expansion Step

We start by loading the state connected to the root node into the environment. Next, we pick an unexplored action from the set of **unvisited actions**. We step the environment using the just picked action and note down returned values such as **reward**, **done**, etc.




Expansion (Image by author)

Similar to the initialization step, we then load the new state from the environment, create a new search tree node and connect it to the state. Finally, we populate the reward member variable (and maybe other variables) with the just obtained values. Last but not least, we

Please note, that in the MCTS version presented in this tutorial, we expand only a single node before continuing with the simulation step. Other flavors of MCTS might expand all child nodes first, before continuing with the simulation steps.

### Simulation Step

The main purpose of the simulation step is to obtain an estimate of how promising the currently visited state is. Remember, that we define the notion of "how good" a state is, in terms of the cumulative reward that is expected to be collected from that state on, going into the future:

$$R_t = \sum_{k=0}^{T} r_{t+k}$$

The cumulative reward, also called **return**, is calculated for the trajectory obtained during a random roll-out:

$$\mathbf{s}_t,\ \mathbf{a}_t,\ \mathbf{s}_{t+1},\ \mathbf{a}_{t+1},\ \mathbf{s}_{t+2},\ \mathbf{a}_{t+2},\ \dots,\ \mathbf{s}_{t+T},\ \mathbf{a}_{t+T}$$

which starts at current state at timestep *t*, is played to the very end of the game, i.e. until a terminal state is reached, and whose actions are taken according to a random roll-out policy.

future:

$$R_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k}$$

Furthermore, using a discount factor prevents the total reward from going to infinity for longer roll-outs (would you rather receive $1000 today or $1000 ten years from now?).

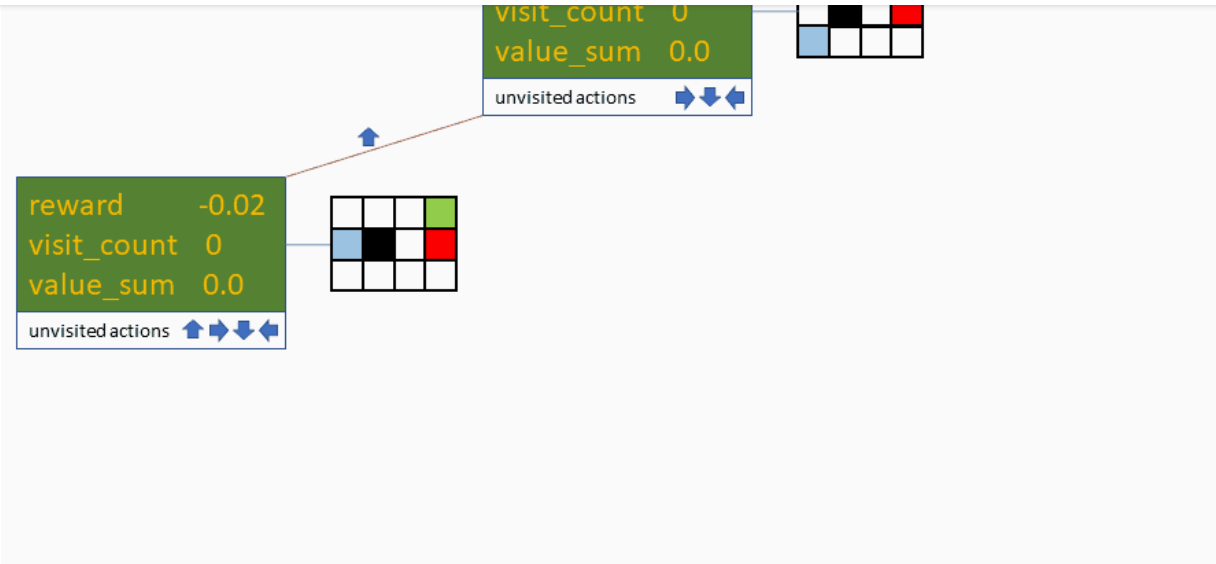
The above value is calculated on a single roll-out and may differ a lot depending on the randomly selected actions (we say: it has high-variance). To gain confidence about the statistics, we must therefore average over multiple roll-outs:

$$V^{\pi}(\mathbf{s}) = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k} \;\middle|\; \mathbf{s}_t = \mathbf{s} \right\},$$

The expression above is called **value-function** in the literature: it is simply the expected sum of discounted future rewards, when starting in state *t* and following policy π thereafter.

Please note, that the expected return, of course depends on the specific policy the agent is following, which is indicated by π in the subscript of the expectation operator. In general **value-functions** are not only calculated for random policies, but for arbitrary policies.

Now, let us come back to the running example:



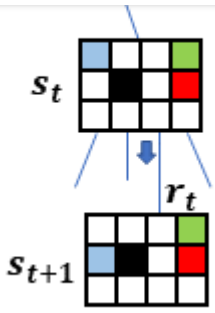

Play-out using a discount of $\gamma = 1.0$ (Image by author)

For convenience we use a gamma factor $\gamma = 1.0$. The first thing we do is load the state connected to the "up" child node into the environment. Then we play the game to the very end using the random roll-out policy. Finally, we note down the obtained return.

## Backpropagation / Backup

First notice that the term backpropagate is a little misleading, since it has nothing to do with the backpropagation used in neural networks to compute the gradients.

The main purpose of the backpropagation step is to recursively update the **value estimates** of all nodes lying along the path from root to the just visited leaf node. The update is performed starting from the bottom of the search tree all the way up to the root. The entity being backpropagated is the **value estimate** obtained in the last simulation.




Backpropagation from timestep $t + 1$ to $t$ (Image by author)

Please notice a little peculiarity: the reward with index $t$ is actually granted for taking the "down" action in state $t$:

$$r_t = r(\mathbf{s}_t, \mathbf{a}_t)$$

In our implementation it is however connected to the state at time $t$+1. This is a deliberate choice because this way we only have to keep a single reward per node. Alternatively, we would have to keep a list of four rewards (belonging to each action) in the state at time $t$.

Let us next derive the recursive formula: assume we have obtained an estimate of the value for the state at timestep $t$+1 by rolling out a random policy:

$$\begin{aligned} R_{t+1} &= \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+1+k} \\ &= r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \cdots \end{aligned}$$

To get the estimate for the state at timestep $t$, we just need to add the previous reward to the above expression:

$$r_t \quad r_t + \gamma \quad r_{t+1}$$

Of course, we have to consider the discount factor. The proof of the above expression is very simple: just plug in the first equation above into the second:

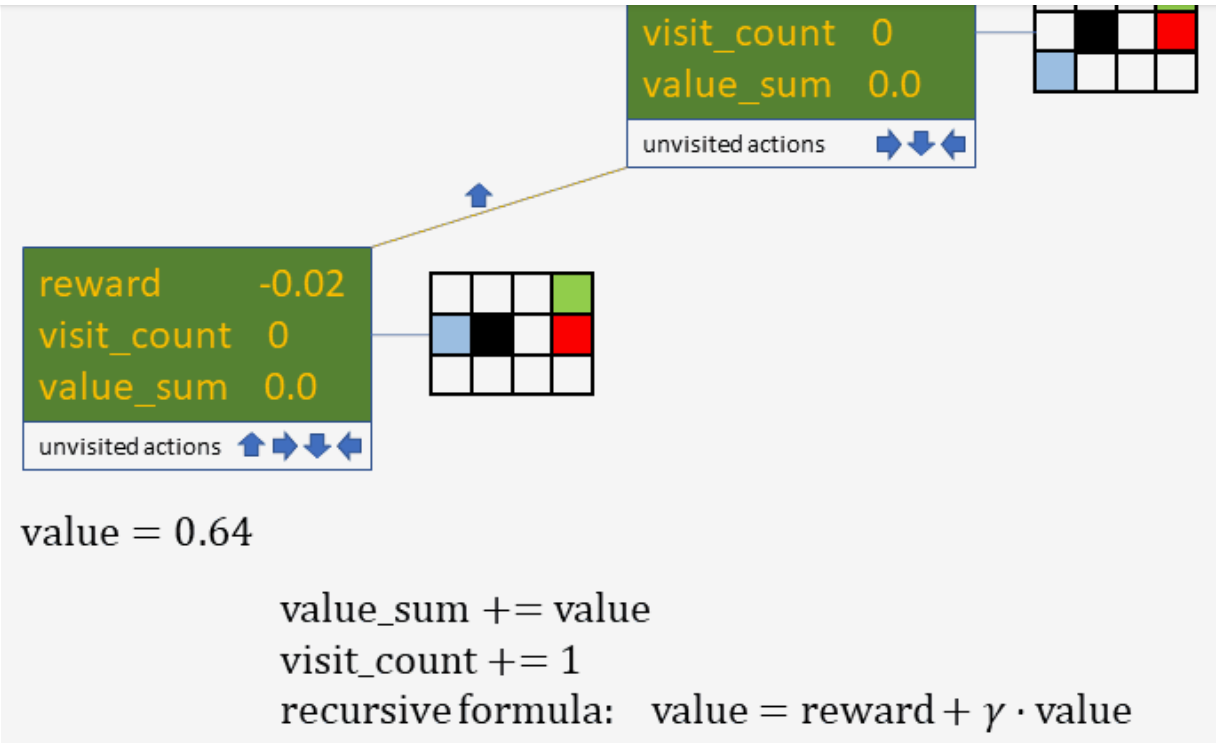
$$\begin{aligned} R_t &= r_t + \gamma \cdot (r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \cdots) \\ &= r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} + \gamma^4 \cdot r_{t+4} + \cdots \end{aligned}$$

A similar recursive formula is also available for the **value-function**:

$$V^{\pi}(\mathbf{s}_t) = r_t + \gamma \cdot E\{V^{\pi}(\mathbf{s}_{t+1})\}$$

which we however won't derive in this post. By the way, the above formula is called **Bellman equation** and is the foundation of many RL algorithms.

Now, let us come back to the running example:




Backpropagating from child node to the root (Image by author)

The first thing we do, is to add the estimate (value = 0.64) obtained in the simulation step to value_sum:

$$value\_sum = value\_sum + \gamma \cdot value$$

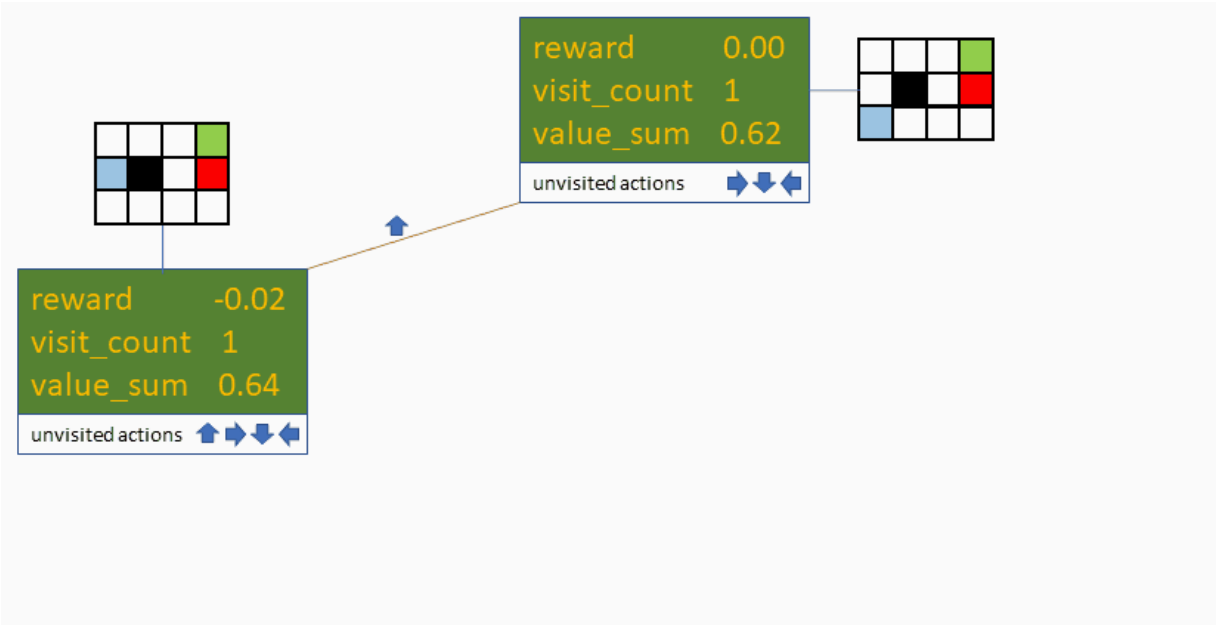
Next, we increase the visit_count:

$$visit\_count = visit\_count + 1$$

This way, we keep track on how many play-outs our **value estimate** is based. Last but not least, we apply the above recursive formula to obtain a **value estimate** for the root node.

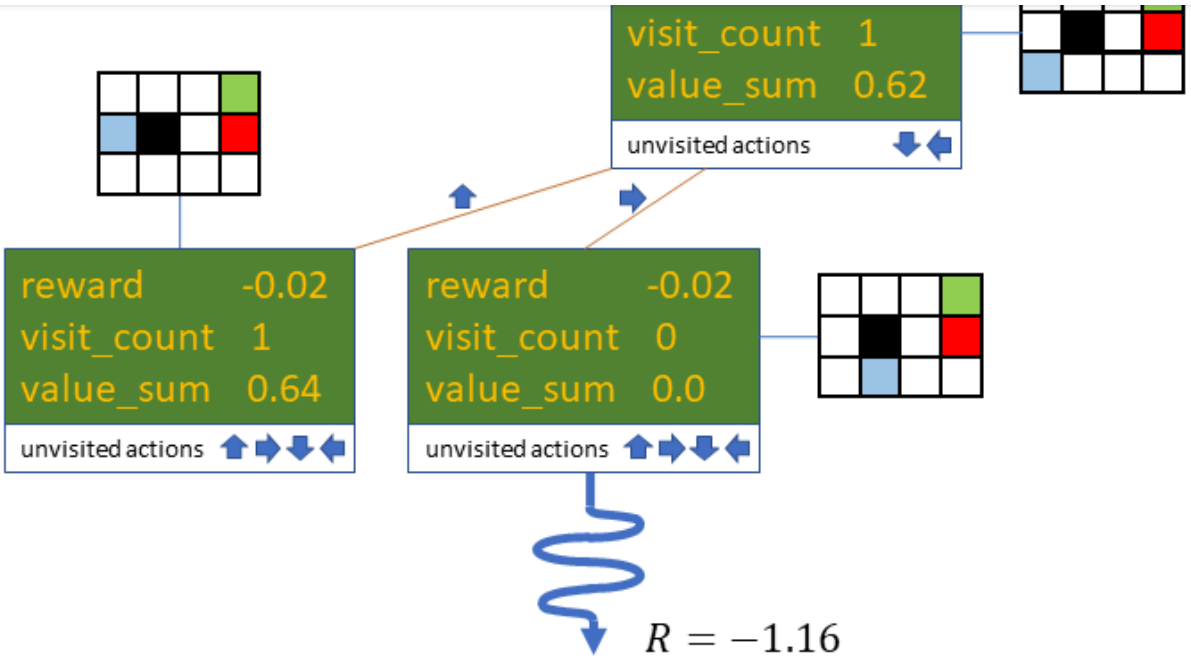
## Fully expanding the root node

knowledge, let us repeat all the steps for the child node belonging to the "right" action:



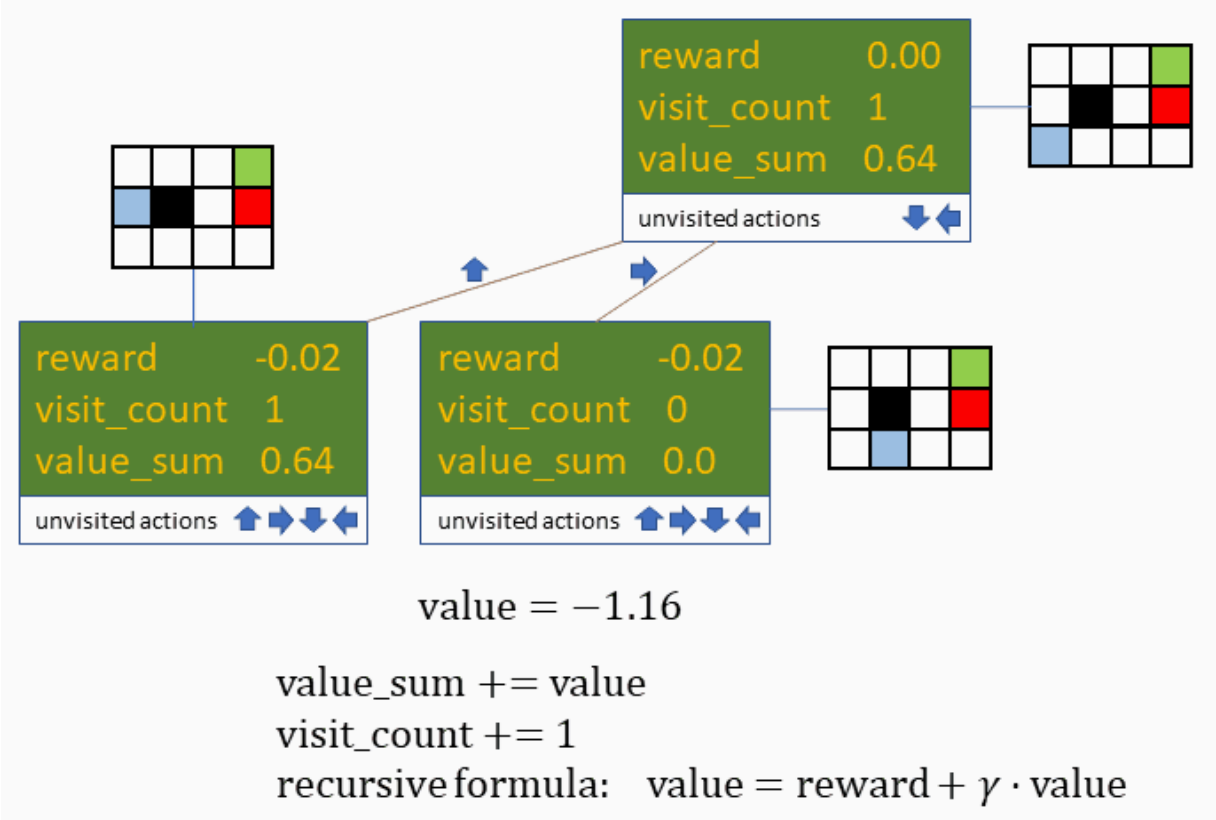

Expansion (Image by author)

The play-out performed in the simulation step happens to end up in the red loser state after taking 8 actions, yielding a negative **value estimate** of -1.16.




Play-out using a discount of $\gamma = 1.0$ (Image by author)

The last step we need to perform is to backpropagate. Please note, how the **visit_count** of the root node is always identical to the sum of the **visit_counts** of its child nodes.

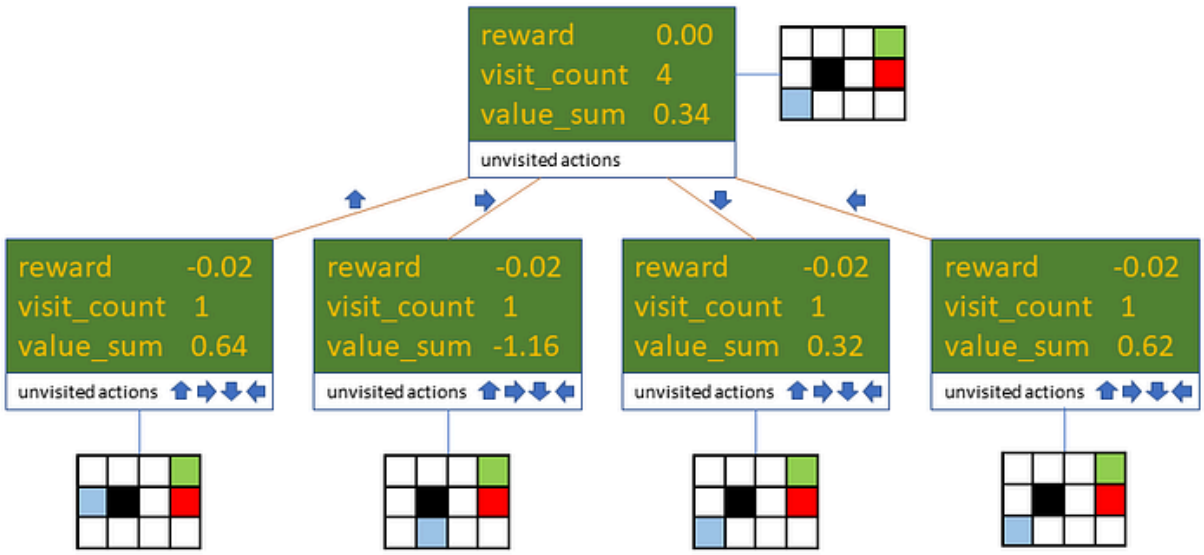



Backpropagation (Image by author)

the root node: "down" and "left". Since the procedure should be clear by now, we skip the corresponding animations.

### Selection Step

We arrive at the point where all child nodes of the root have been expanded and a single simulation has been run on each one of them. The root is **fully expanded** now:




Fully expanded root node (Image by author)

The question arises as to which of the child nodes we should select for further exploration. Remember, the goal of MCTS is to focus on promising paths, while quickly abandoning unpromising ones. The formula that satisfies all these conditions is called *UCT* (Upper Confidence Bound 1 applied to trees):

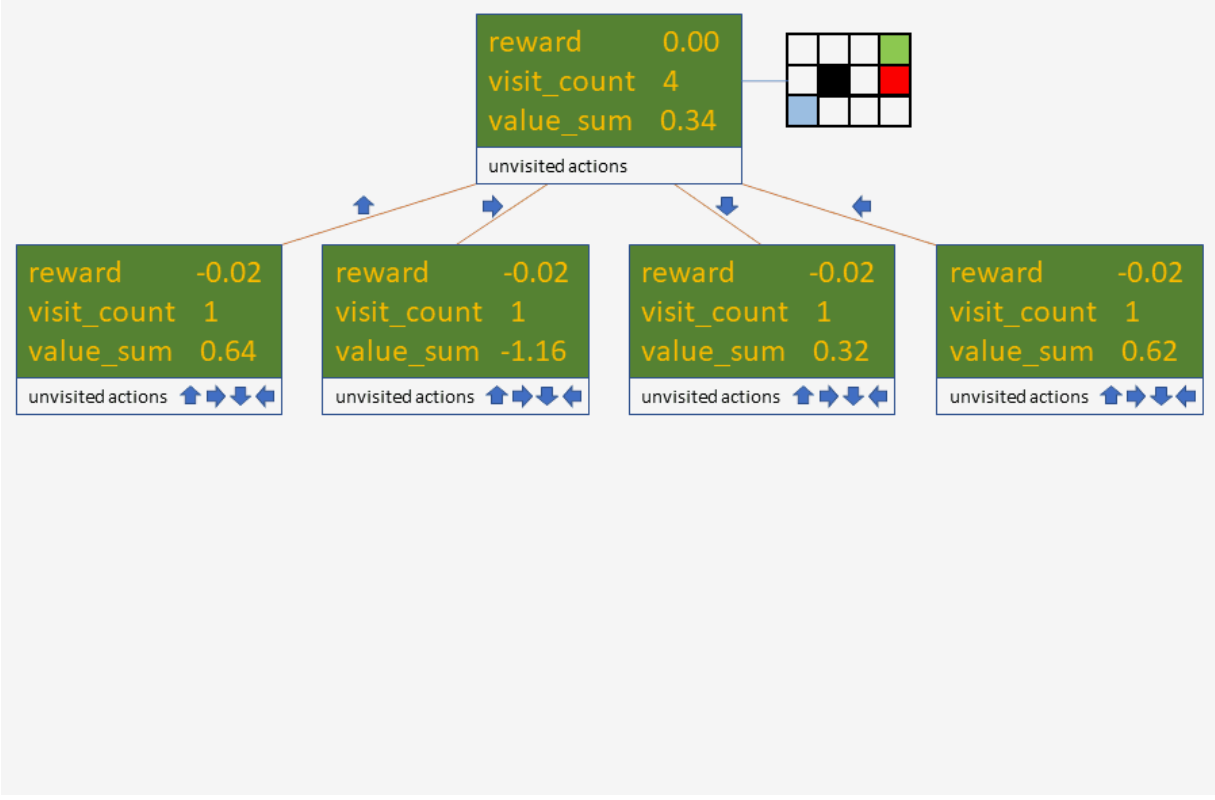
$$UCT(\text{child}) = \frac{\text{child.value\_sum}}{\text{child.visit\_count}} + c\sqrt{\frac{\log(\text{parent.visit\_count})}{\text{child.visit\_count}}}$$

is simply the estimate of the **value-function**, we've introduced in a previous section, and reflects "how promising" the associated state is. The second part of the formula represents a bonus for rarely visited child nodes. We will see in a later section, that the first part corresponds to what is called **exploitation**, and the second part to what is called **exploration**.

Next, let's plug all the values from the above figure into the *UCT* formula using $c = 1.2$:

| Up | Right | Down | Left |
|---|---|---|---|
| 2.05 | 0.25 | 1.73 | 2.03 |

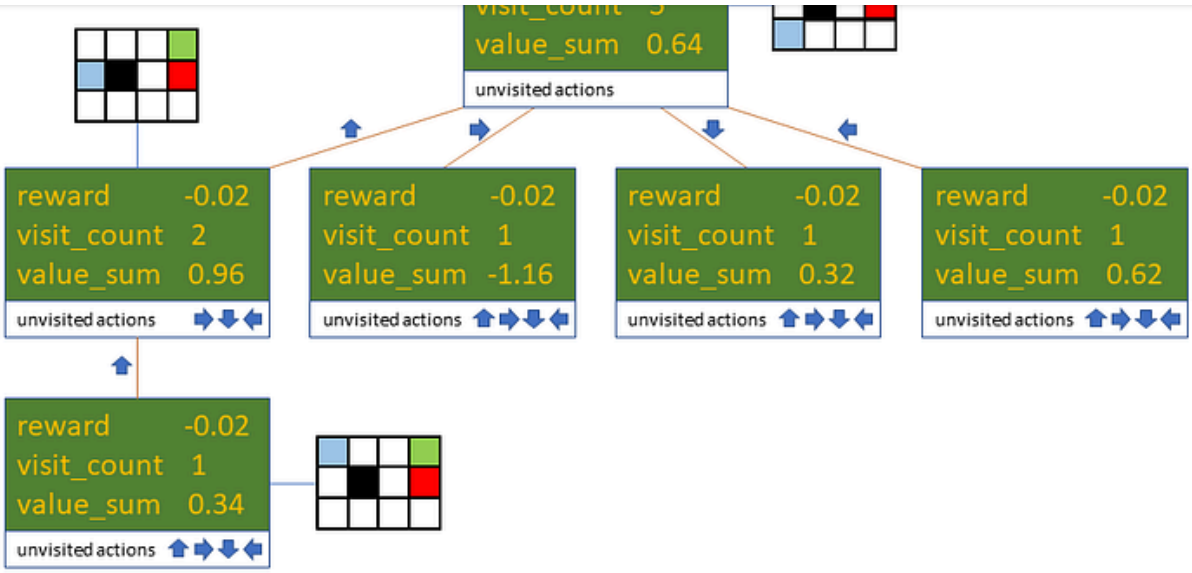
The child node belonging to the "up" action is the best child and is selected. Let's continue with the animation:



Let's quickly recap what we know about the selection step: its main purpose is to find the most promising path from the root node to a leaf node using the *UCT* formula. To this end, we start at the top of the search tree and using *UCT* select the best child in **level 1**. Depending on the depth of the tree, we then continue with **level 2** performing again *UCT* and selecting its best child. We repeat this procedure, until we reach a not (!) **fully expanded** node (aka leaf node) or a terminal node.

Once we reach a not fully expanded node (a leaf node that still has unvisited actions), we then proceed with the Simulation and Backpropagation steps. Remember that during Backpropagation all the value_sums and visit_counts along the path to the leaf node are updated. In the next iteration, we start again from the root node and use *UCT* to identify the most promising path.

Keep in mind that this path may differ significantly from the previous iteration due to the updates made during the last Backpropagation step: plugging the numbers from the last frame of the above animation




Selection, Expansion, Simulation and Backpropagation (Image by author)

gives us the following *UCT* values for the root node:

| Up | Right | Down | Left |
|---|---|---|---|
| 1.56 | 0.36 | 1.84 | 2.14 |

This time the child node belonging to the "left" action is selected.

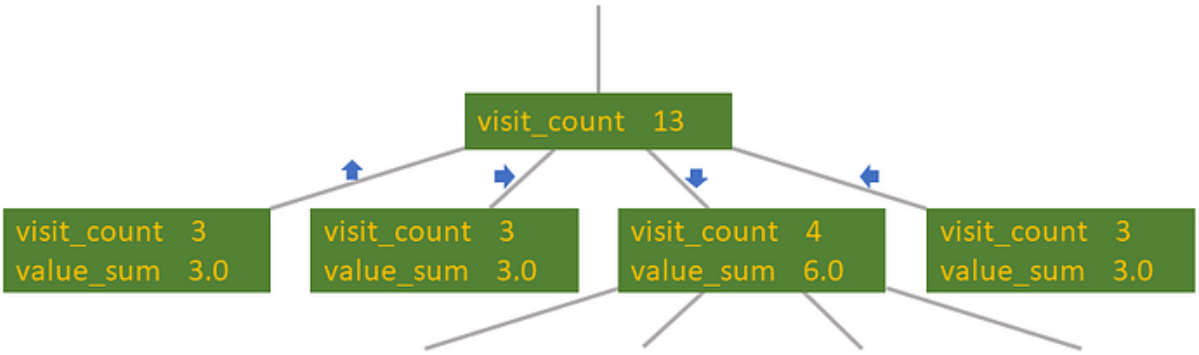
The Expansion step, as introduced above, refers only to selecting one of the unvisited actions and adding the corresponding child node to the tree. However, in the following, we will use the term **expand a node** to collectively refer to the three steps: Expansion, Simulation, and Backpropagation.

The two distinct steps of selection and expansion are oftentimes summarized under the term **tree-policy**:

```
    current_node = root

    # selection step
    while current_node is fully expanded:
        current_node = current_node.best_child()

    if current_node is terminal:
        return current_node

    # expansion step
    return current_node.expand()
```

## Exploitation-Exploration Trade-Off

In this section we will explain the mechanism behind the *UCT* formula in detail. For illustrative purposes we start with an already built up search tree. The node depicted below is fully expanded (has four child nodes) and is located somewhere in, say, **level 4.**




Search tree (Image by author)

First, let us have a look at the **visit_count** of its child nodes. We see, that the "down" action was visited 4 times, while the others were visited only 3 times. If we compare the **value_sum** of each node, we find that the value associated with the "down" action is twice as large.

| Up | Right | Down | Left |
|---|---|---|---|
| 1.0 | 1.0 | 1.5 | 1.0 |

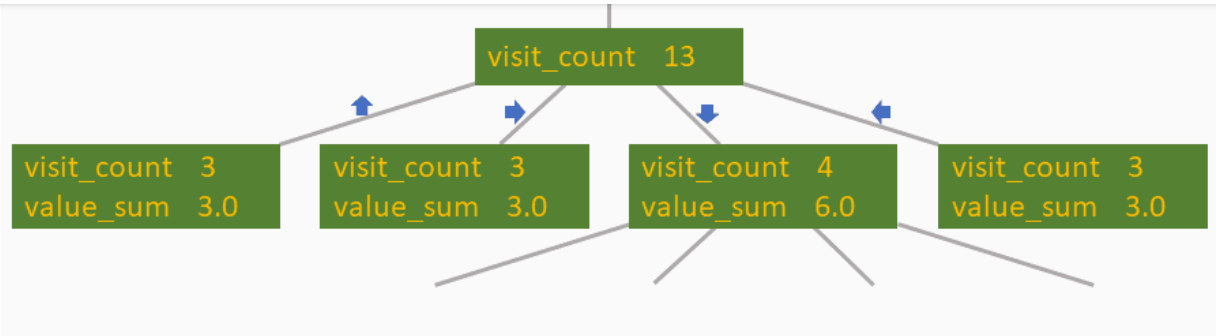
Now, in order to select the best child, we could just go for the highest value above. In other words, we already have acquired some knowledge about how promising each state is, and now we just want to "exploit" that knowledge.

However, there is a danger with this strategy: please remember, that we use random simulations to obtain the estimates and those can be very noisy sometimes. A promising state could hence easily end up getting a low **value estimate** and vice versa. Just selecting a state, which only by chance got a high value, would completely stop further exploration for more promising paths.

For exactly this reason the *UCT* formula has a second term, which we will analyze next:

$$UCT(\text{child}) = \frac{\text{child.value\_sum}}{\text{child.visit\_count}} + c\sqrt{\frac{\log(\text{parent.visit\_count})}{\text{child.visit\_count}}}$$

Let us first take a quick look, at same traversals going through our parent node:

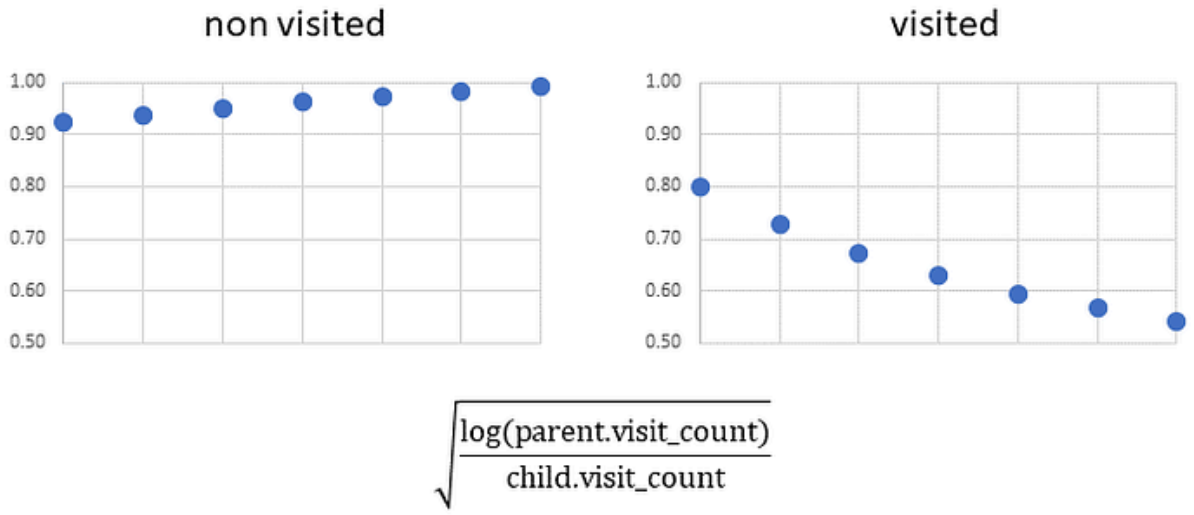



Selection and Backpropagation through a node (Image by author)

For illustrative purposes, all the numbers in the example above have been chosen such that when using the *UCT* formula with $c = 1.2$, the "down" action is selected 5 times in a row.

Now, when you look carefully, you'll notice, that during each backpropagation step the **visit_count** of the selected child node is incremented by one, whereas the **visit_count** of the remaining child nodes remains unchanged. Furthermore, the **visit_count** of the parent node is also incremented by one, since the backpropagations path always leads through it.

Next, let us plot the "exploration" part of the *UCB* formula for all four child nodes:




Exploration terms with increasing iterations (Image by author)

remains constant, while the parent's **visit_count** in the nominator is incremented. In contrast, for the selected "down" node, the value is decreasing. Here the log of the **visit_count** of the parent node is increasing slower that the "unlogged" **visit_count** of the child nodes. Now, we see, that there must exist a future point in time, where the "exploration" part of the unvisited nodes has grown so large, such that they will eventually be revisited again.

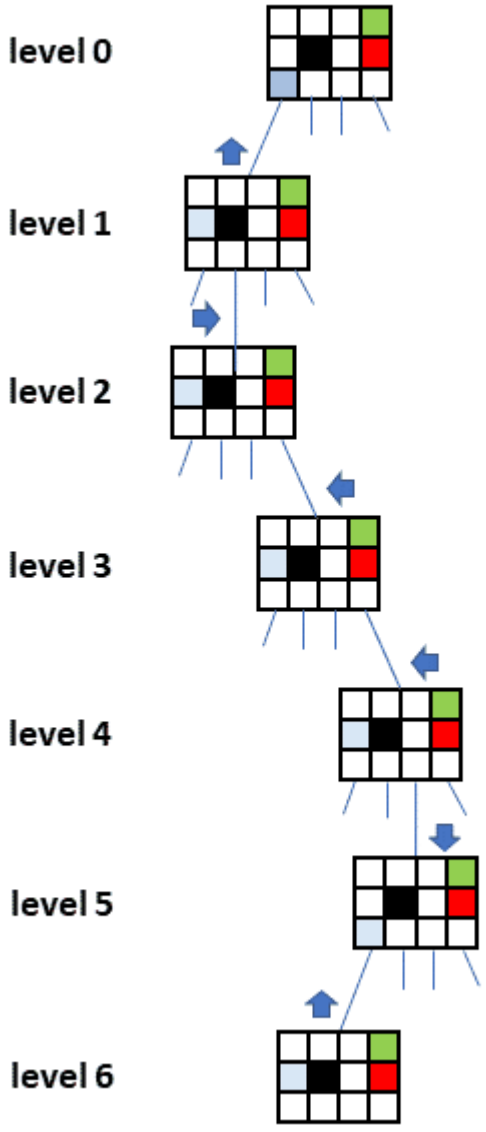
This mechanism ensures "exploration" so that every once in a while paths that don't look very promising at first are nevertheless explored in the future and could turn out to be very promising. Without this mechanism MCTS would completely disregarding paths that have not yet been explored. Exploration by *UCB* takes into account the number of times each action was chosen, and gives additional weight to the less-explored. The parameter *c* allows us to trade-off between "exploration" and "exploitation".

### Stochastic Environments

Let us quickly recap what defines a deterministic environment: we start from a particular state of our game and take an action, that teleports us to the next state. Now, we reset the environment to the same original state and repeat the same action. In a deterministic environment we always end up in the exactly same next state. However, in stochastic environments, the next state we end up in is uncertain.

The biggest adaptation we have to apply to MCTS, when dealing with stochastic environments, is the **selection step**. Remember, in the selection step we use the **tree policy** to construct a path from the

In the deterministic case, starting in the root state and applying this sequence of actions would always lead us to exactly the same states along the path. In the stochastic case, the visited states could however be totally different, as the animation below shows:




Reapplying of each action along the best path (Image by author)

The pale blue states show the states which were visited during the last iteration. The dark blue states are the states we reach in the current iteration when reapplying the same actions.

these rewards will later be used in the **Backpropagation step** to update the **value estimates**. Hence, we have to update the **reward** member variables of each node along the path.

Let us quickly summarize the stochastic version the **selection step**: we start at the root node and choose the best child according to *UCT* formula. However, unlike in the deterministic case, we don't just continue with the best child node, but reapply the corresponding action to root's state (in the environment). We transit to the next state and obtain a reward. Due to the stochastic nature of the environment both may differ from iteration to iteration. Hence, we need to update the **reward** member variable of the best child and store the latest state. We repeat this process until we reach a leaf-node.

The repeated updates along the best path in each iteration of course require more computational power. Furthermore, depending on the degree of stochasticity, more overall iterations may be needed to get a sufficient confidence into the statistics.

## References

Monte Carlo Tree Search — beginners guide State Values and Policy Evaluation Relationship between state and action value function in Reinforcement Learning. Value Functions Monte Carlo Tree Search MCTS

#artificial-intelligence #machine-learning #reinforcement-learning #game-theory #editors-pick