# COMP90054 — AI Planning for Autonomy

## Week 8a. Monte Carlo Tree Search

*Balancing Exploitation and Exploration*

Adrian Pearce

THE UNIVERSITY OF MELBOURNE

Semester 2, 2020

# Agenda

# Learning Outcomes

1. Explaining the difference between offline and online planning for MDPs.
2. Apply MCTS solve small-scale MDP problems manually and program MCTS algorithms to solve medium-scale MDP problems automatically
3. Construct a policy from Q-functions resulting from MCTS algorithms
4. Select and apply multi-armed bandit algorithms
5. Integrate multi-armed bandit algorithms (including UCB) to MCTS algorithms
6. Compare and contrast MCTS to value/policy iteration
7. Discuss the strengths and weaknesses of the MCTS family of algorithms.

# Relevant Reading

- Chapter 2 Multi-armed Bandits; Chapter 5 Monte Carlo Methods; and Section 8.11 Monte Carlo Tree Search of *Reinforcement Learning: An Introduction, second edition*, Sutton and Barto 2020. Freely downloadable at `http://www.incompleteideas.net/book/RLbook2020.pdf`

  → Text book covering key concepts

- *A Survey of Monte Carlo Tree Search Methods* by Browne et al. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012 Available at `https://ieeexplore.ieee.org/document/6145622` (Unimelb library access required—you can use the *Library Access* plugin in Chrome to automatically gain access)

  → Good "entry level" resource, with lots of pointers to seminal papers

- Regret Analysis of Stochastic and Non-stochastic Multi-armed Bandit Problems by S. Bubeck and N. Cesa–Bianchi, 2012 `http://arxiv.org/pdf/1204.5721v2.pdf`

  → All you want to know about regret analysis and multi-armed bandits

# Offline Planning & Online Planning over MDPs

We saw value iteration and policy iteration in the previous lecture. These are **offline planning** methods, in that we solve the problem offline for all possible states, and then use the solution (a policy) online. In the offline planning approach:

- We can define policies, $\pi$, that work from any state in a convenient manner,
- Yet the state space $S$ is usually *far* too big to determine $V(s)$ or $\pi$ exactly.
- There *are* methods to approximate the MDP by reducing the dimensionality of $S$, but we will not discuss these until later.

In **online planning**, planning is undertaken immediately before executing an action. Once an action (or perhaps a sequence of actions) is executed, we start planning from the current state. As such, planning and execution are interleaved.

- For each state $s$ visited, many policies $\pi$ are *evaluated* (partially)
- The quality of each $\pi$ is approximated by averaging the expected reward of trajectories over $S$ obtained by repeated *simulations* of $r(s, a, s')$.
- The chosen policy $\hat{\pi}$ is then selected and the action $\hat{\pi}(s)$ executed.

The question is: how to we do the repeated simulations? *Monte Carlo methods are by far the most widely-used approach.*

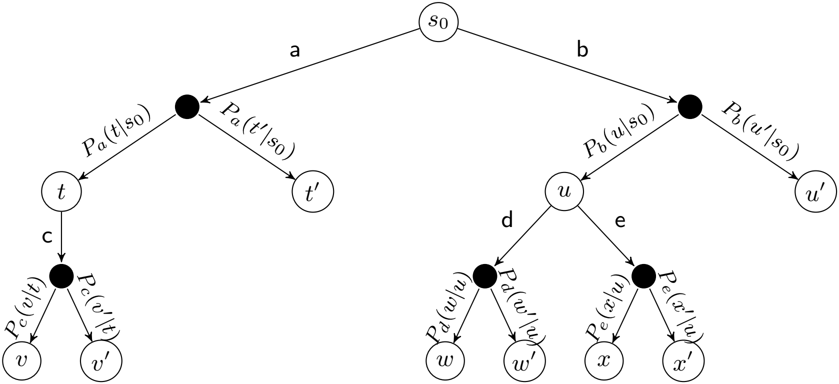
# Monte Carlo

Monte Carlo Tree Search (MTCS) is a name for a *set* of algorithms all based around the same idea. Here, we will focus on using an algorithm for solving single-agent MDPs online.

*Monte Carlo* is an area within Monaco (small principality on the French riviera), which is best known for its extravagant casinos. As gambling and casinos are largely associated with chance, methods for solving MDPs online are often called *Monte Carlo* methods, because they use *randomness* to search the action space.

# Foundation: MDPs as ExpectiMax Trees

To get the idea of MCTS, we note that MDPs can be represented as trees (or graphs), called *ExpectiMax* trees:




The letters a-e represent actions, and letters $s$-$x$ represent states. White nodes are state nodes, and the small black nodes represent the probabilistic uncertainty: the 'environment' choosing which outcome from an action happens, based on the transition function.

# Monte Carlo Tree Search – Overview

The algorithm is online, which means the action selection is interleaved with action execution. Thus, MCTS is invoked every time an agent visits a new state.

**MCTS** has the following fundamental features:

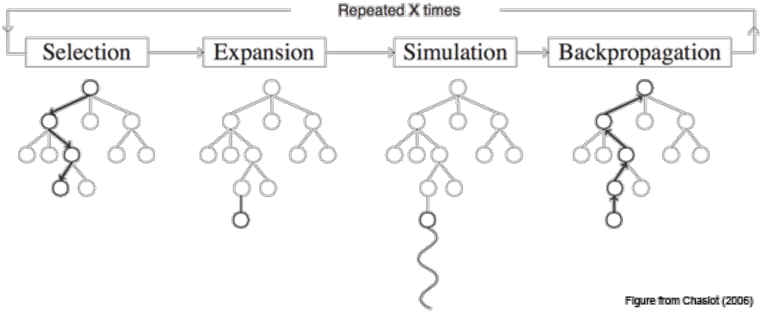
1. The value $V(s)$ for each is approximated using *random simulation*.
2. An ExpectiMax *search tree* is built incrementally
3. The search terminates when some pre-defined computational budget is used up, such as a time limit or the number of expanded nodes;—therefore, it is an *anytime algorithm*, as it can be terminated at any time and it still provides an answer.
4. The best performing action is returned.
   - → This is complete if there are *no* dead–ends.
   - → This is optimal if an entire search can be performed (although this is unusual—if the problem is that small we should just use value/policy iteration).

# The Framework: Monte Carlo Tree Search (MCTS)

MCTS builds up an MDP tree using simulation. The evaluated states are stored in a search tree. The set of evaluated states is *incrementally* built be iterating over the following four steps:

- *Select*: Select a single node in the tree that is *not fully expanded*. By this, we mean at least one of its children is not yet explored.
- *Expand*: Expand this node by applying one available action (as defined by the MDP) from the node.
- *Simulation*: From one of the new nodes, perform a complete random simulation of the MDP to a terminating state. This might typically assume that the search tree is finite, however versions for infinitely large trees exist in which we just execute for some time and then estimate the outcome.
- *Backpropagate*: Finally, the value of the node is *backpropagated* to the root node, updating the value of each ancestor node on the way using expected value.

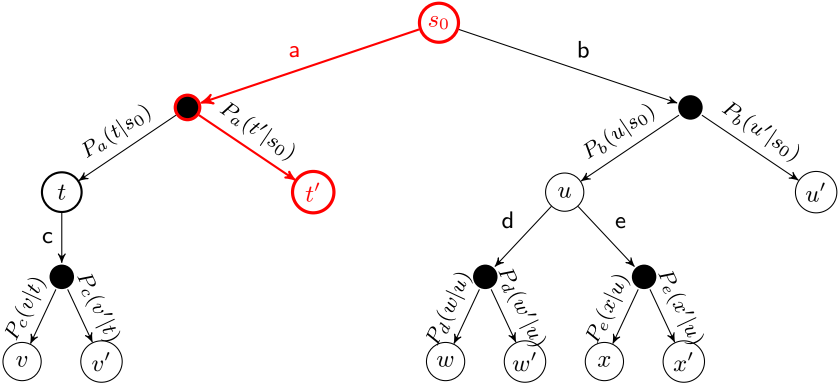
# The Framework: Monte Carlo Tree Search (MCTS)




From: Chaslot, Guillaume, Sander Bakkes, Istvan Szita, and Pieter Spronck. *Monte-Carlo Tree Search: A New Framework for Game AI.* in *AIIDE*. 2008.
`https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf`

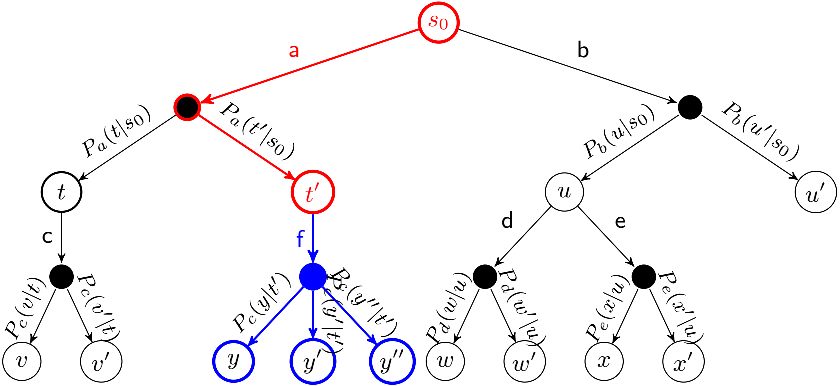
# Monte Carlo Tree Search: Selection

Start at the root node, and successively select a child until we reach a node that is not fully expanded.



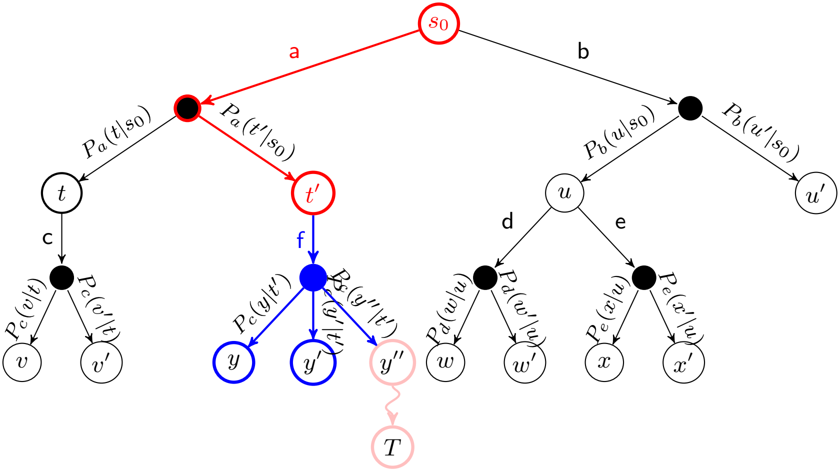
# Monte Carlo Tree Search: Expansion

Unless the node we end up at is a terminating state, expand the children of the selected node by choosing an action and creating new nodes using the action outcomes.



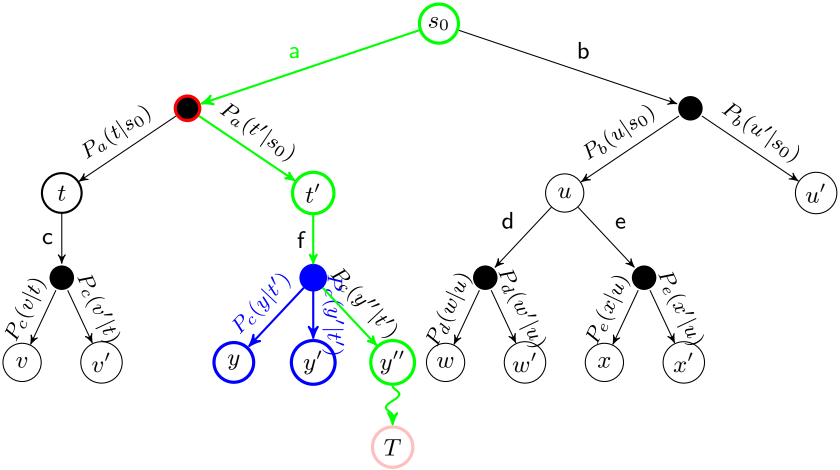
# Monte Carlo Tree Search: Simulation

Choose one of the new nodes and *perform a random simulation of the MDP to the terminating state*:



# Monte Carlo Tree Search: Backpropagation

Given the reward $r$ at the terminating state, *backpropagate* the reward to calculate the value $V(s)$ at each state along the path.



# Monte Carlo Tree Search: Algorithm

**Input**: MDP $M$, with initial state $s_0$ and time limit $T$.

**function** $\textsc{MctsSearch}\ M = \langle S, s_0, A, P_a(s'|s), R(s)\rangle$ **returns** action $a$
  **while** $current\_time < T$ **do**
    $expand\_node := \textsc{Select}(root)$;
    $children := \textsc{Expand}(expand\_node)$;
    $child := \textsc{Choose}(children)$; – choose a child to simulate
    $reward := \textsc{Simulate}(child)$; – simulate from $child$
    $\textsc{Backup}(expand\_node, reward)$;
  **return** $\mathrm{argmax}_a\ Q(s_0, a)$;

$\rightarrow$ Each node stores an estimate of the value of the state, $V(s)$, for its state, the number of times the state has been visited, and a pointer to their parent node.

# Monte Carlo Tree Search: Algorithm

**Input**: MDP $M$, with initial state $s_0$ and time limit $T$.

**function** $\textsc{MctsSearch}\ M = \langle S, s_0, A, P_a(s'|s), R(s)\rangle$ **returns** action $a$
  **while** $current\_time < T$ **do**
    $expand\_node := \textsc{Select}(root)$;
    $children := \textsc{Expand}(expand\_node)$;
    $child := \textsc{Choose}(children)$; – choose a child to simulate
    $reward := \textsc{Simulate}(child)$; – simulate from $child$
    $\textsc{Backup}(expand\_node, reward)$;
  **return** $\text{argmax}_a\ Q(s_0, a)$;

$\textsc{Select}(root)$

→ Recursively selects the next node using some probabilistic policy (we'll see more of this in 'Multi-Armed Bandits' section later), until we reach a node that is not fully expanded

→ At each choice point, selects one of the edges in the tree.

→ Then, using $P_a(s|s')$, select the outcome for that action. Thus, our *simulation* follows the probability transitions of the underlying model.

# Monte Carlo Tree Search: Algorithm

**Input**: MDP $M$, with initial state $s_0$ and time limit $T$.

> **function** $\textsc{MctsSearch}\ M = \langle S, s_0, A, P_a(s'|s), R(s)\rangle$ **returns** action $a$
>   **while** $current\_time < T$ **do**
>     $expand\_node := \textsc{Select}(root)$;
>     $children := \textsc{Expand}(expand\_node)$;
>     $child := \textsc{Choose}(children)$; – choose a child to simulate
>     $reward := \textsc{Simulate}(child)$; – simulate from $child$
>     $\textsc{Backup}(expand\_node, reward)$;
>   **return** $\mathrm{argmax}_a\ Q(s_0, a)$;

$\textsc{Expand}(expand\_node)$

- → Take the selected node and randomly select an action that can be applied in that state *and has not been selected previously in that state*.
- → Expand all possible outcomes nodes for that action.
- → Check if the generated nodes are already in tree. If not in the tree, *add* these nodes to the tree.

**Note**: $P_a(s|s')$ is *stochastic*, so several visits (in theory an infinite number) may be necessary to generate all successors.

# Monte Carlo Tree Search: Algorithm

**Input**: MDP $M$, with initial state $s_0$ and time limit $T$.

```
function MctsSearch M = ⟨S, s0, A, Pa(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s0, a);
```

Simulate($child$)

- → Perform a random simulation of the MDP until we reach a terminating state. That is, at each choice point, randomly select an enable action from the MDP, and use transition probabilities $P_a(s'|s)$ to choose an outcome for each action.
- → We can use non-random simulation as well by following some heuristic, but we will not look at this in these notes.
- → $reward$ is the reward obtained over the entire simulation.
- → To avoid memory explosion, we *discard* all nodes generated from the simulation. In any non-trivial search, we are unlikely to ever need them again.

# Monte Carlo Tree Search: Algorithm

**Input**: MDP $M$, with initial state $s_0$ and time limit $T$.

```
function MctsSearch M = ⟨S, s0, A, Pa(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s0, a);
```

$\textsc{Backup}(\mathit{expand\_node}, \mathit{reward})$

→ The reward from the simulation is backpropagated from the expanded node to its ancestors recursively.

→ We must not forget the *discount factor*!

→ For each state $s$, get the expected value of all actions from that node:

$$V(s) := \max_{a \in A(s)} \sum_{s' \in children} P_a(s'|s)\ [r(s, a, s') + \gamma\ V(s')]$$

Look familiar?!

This is why the tree is called an ExpectiMax tree: we maximise the expected return, and this calculation is done over two layers. The summation $(\sum_{s' \in S} \ldots)$ is calculating the value of the small black nodes in the tree, while the maximisation $(\max_{a \in A(s)})$ calculates the value of the large white nodes (the state nodes).

# Monte Carlo Tree Search: Execution

Once we have run out of computational time, we select the action that maximises are expected return, which is simply the one with the highest Q-value from our simulations:

$$\operatorname*{argmax}_a Q(s_0, a)$$

which is just

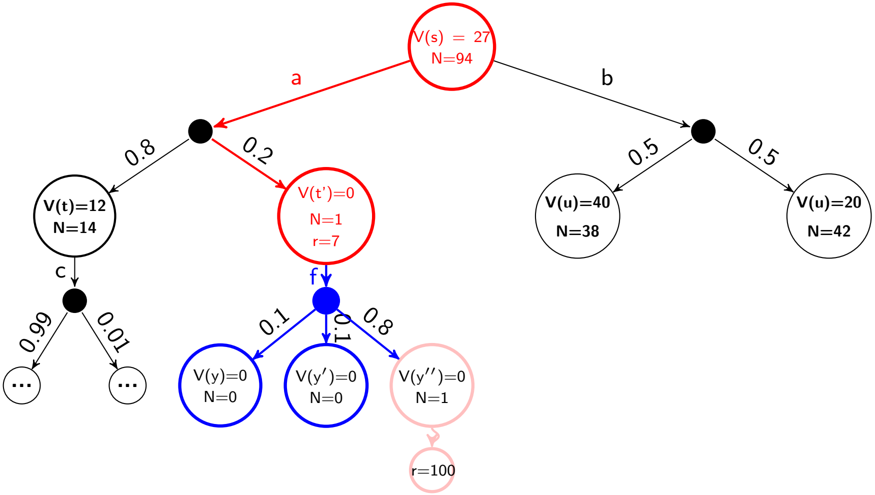
$$\operatorname*{argmax}_a \sum_{s' \in A(s_0)} P_a(s'|s_0)\ [r(s_0, a, s') + \gamma\ V(s')]$$

We execute that action and wait to see which outcome occurs for the action. Once we see the outcome, which we will call $s'$, we start the process all over again, except with $s_0 := s'$.

Importantly, we can *keep* the sub-tree from state $s'$, as we already have done simulations from that state. We discard the rest of the tree (all child of $s_0$ other than the chosen action) and incrementally build from $s'$.

# Example (after the simulation step)

Assume $\gamma = 0.9$, $r = X$ represents reward $X$ received at a state, N is the number of times the state has been visited, and the length of the simulation is 13. After the simulation step, but before backpropagation, our tree would look like this:

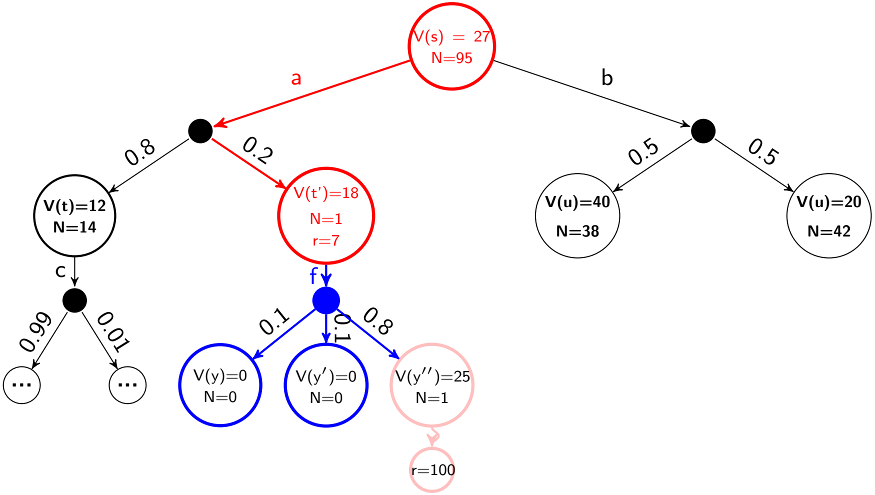


# Example (the backpropagation step)

$$
\begin{aligned}
V(y'') &= \max_{a \in A} \textstyle\sum_{s' \in children(y'')} P_a(s'|y'')\ [r(y'', a, s') + \gamma\ V(s')] \\
&= \gamma^{13} \times 100 \text{ (simulation is 13 steps long and receives reward of 100)} \\
&\approx 25 \\[1em]
V(t') &= \max_{a \in \{f\}} \textstyle\sum_{s' \in children(t')} P_a(s'|t')\ [r(t', a, s') + \gamma\ V(s')] \\
&= 0.1(0+0)\ +\ 0.1(0+0)\ +\ 0.8(0 + 0.9 \times 25) \\
&= 18 \\[1em]
V(s) &= \max_{a \in \{a,b\}} \textstyle\sum_{s' \in children(s)} P_a(s'|s)\ [r(s, a, s') + \gamma\ V(s')] \\
&= \max(0.8(0 + 0.9 \times 12) + 0.2(7 + 0.9 \times 18), \quad \text{(action a)} \\
&\qquad\quad 0.5(0 + 0.9 \times 40) + 0.5(0 + 0.9 \times 20) \quad \text{(action b)} \\
&= \max(8.64 + 4.62,\ 18 + 9) \\
&= 27
\end{aligned}
$$

# Example (after the backpropagation step)

The value of $V(s)$ does not change because action b still returns the maximum discounted future reward.



# Informed Search

There is one key question that we need to answer:

How do we select the next node to expand?

It turns out that this selection makes a big difference on the performance of MCTS.

# Multi-Armed Bandit: Informal Definition

The selection of nodes can be considered an instance of the *Multi-armed bandit* problem. This problem is defined as follows:

> *Imagine that you have $N$ number of slot machines (or poker machines in Australia), which are sometimes called one-armed bandits. Over time, each bandit pays a random reward from an unknown probability distrbution. Some bandits pay higher rewards than others. The goal is to maximize the sum of the rewards of a sequence of lever pulls of the machine.*

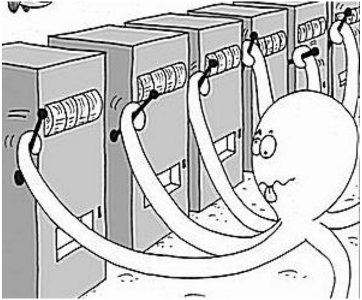


Image courtesy of Mathworks blog: `https://blogs.mathworks.com/loren/2016/10/10/multi-armed-bandit-problem-and-exploration-vs-exploitation-trade-off/`

# Multi-Armed Bandit: Formal Definition

An $N$-armed bandit is defined by a set of *random variables* $X_{i,k}$ where

- $1 \leq i \leq N$, such that $i$ is the *arm* of the bandit; and
- $k$ the index of the *play* of arm $i$.

Successive plays $X_{i,1}, X_{j,2}, X_{k,3} \ldots$ are assumed to be independently distributed according to an *unknown* law. That is, we do not know the probability distributions of the random variables.

**Intuition**: actions $a$ applicable on $s$ are the "arms of the bandit", and $Q(s, a)$ corresponds to the random variables $X_{i,n}$.

# Flat Monte Carlo (FMC) a.k.a Uniform Sampling

Given that we do not know the distributions, a simple strategy is simply to select the arm given a uniform distribution; that is, select each arm with the same probability. This is just uniform sampling.

Then, the Q-value for an action $a$ in a given state $s$ can be approximated using the following formula:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{t=1}^{N(s)} \mathbb{I}_t(s,a) r_t$$

- $N(s,a)$ is the number of times $a$ executed in $s$.
- $N(s)$ is the number of times $s$ is visited.
- $r_t$ is the *reward* obtained by the $t$-th simulation from $s$.
- $\mathbb{I}_t(s,a)$ is $1$ if $a$ was selected on the $t$-th simulation from $s$, and is $0$ otherwise

$\rightarrow$ FMC suffices to achieve *world champion level* play on Bridge (Ginsberg, 01) and Scrabble (Sheppard, 02).

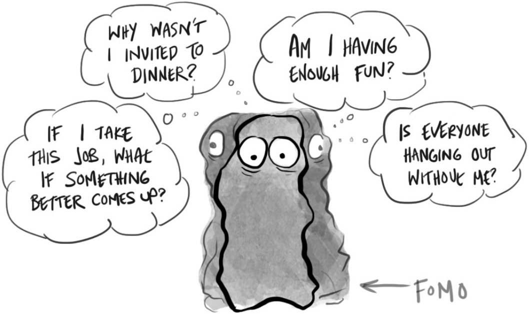
**But what is the issue?** Sampling Time is wasted equally in all actions using the uniform distribution. Why not focus also on the *most promising actions* given the rewards we have received so far.

# Exploration vs. Exploitation

What we want is to play only the good actions; so just keep playing the actions that have given us the best reward so far. However, our selection is randomised, so what if we just haven't sampled the best action enough times? Thus, we want strategies that *exploit* what we think are the best actions so far, but still *explore* other actions.

But how much should we exploit and how much should we explore? This is known as the *exploration vs. exploitation dilemma*.

It is driven by the *The Fear of Missing Out* (FOMO)



## The Fear Of Missing Out

We seek policies $\pi$ that *minimise regret*.

**(Pseudo)–Regret**

$$\mathcal{R}_{N(s),b} = Q(\pi^*(s), s)N(s) - \mathbb{E}[\sum_{t}^{N(s)} Q(b,s)\mathbb{I}_t(s,b)]$$

$Q(\pi^*(s), s)$ is the $Q$-value for the (unknown) optimal policy $\pi^*(s)$,
$N(s)$ is the number of visits to state $s$,
$\mathbb{I}_i(s,a)$ is $1$ if $a$ was selected on the $i$-th visit from $s$, and $0$ otherwise,
**Important**: $\mathbb{E}[\sum_t^{N(s)} Q(b,s)\mathbb{I}_t(s,b)] > 0$ for every $b$.

Informally: If I play arm $b$, my regret is the *best possible expected reward* minus the *expected reward of playing* $b$. If I play arm $a$ (the best arm), my regret is 0.

*Regret* is thus the *expected loss* due to not doing the best action.

$\rightarrow$ In multi-armed bandit algorithms, exploration is *literally* driven by FOMO.

# Solutions that aim to minimise regret

$\epsilon$-**greedy**: $\epsilon$ is a number in [0,1]. Each time we need to choose an arm, we choose a random arm with probability $\epsilon$, and choose the arm with max $Q(s, a)$ with probability $1 - \epsilon$. Typically, values of $\epsilon$ around 0.05-0.1 work well.

$\epsilon$-**decreasing**: The same as $\epsilon$-greedy, $\epsilon$ decreases over time. A parameter $\alpha$ between [0,1] specifies the *decay*, such that $\epsilon := \epsilon.\alpha$ after each action is chosen.

**Softmax**: This is *probability matching strategy*, which means that the probability of each action being chosen is dependent on its Q-value so far. Formally:

$$\frac{e^{Q(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q(s,b)/\tau}}$$

in which $\tau$ is the *temperature*, a positive number that dictates how much of an influence the past data has on the decision.

# Upper Confidence Bounds (UCB1)

A highly effective (especially in terms of MCTS) multi-armed bandit strategy is the *Upper Confidence Bounds* (UCB1) strategy.

**UCB1 policy $\pi(s)$**

$$\pi(s) := \underset{a \in A(s)}{\operatorname{argmax}}\ Q(s,a) + \sqrt{\frac{2 \ln N(s)}{N(s,a)}}$$

$Q(s,a)$ is the estimated $Q$-value.

$N(s)$ is the number of times $s$ has been visited.

$N(s,a)$ is the number of times times $a$ has been executed in $s$.

→The left–hand side encourages exploitation: the Q-value is high for actions that have had a high reward.
→The right–hand side encourages exploration: it is high for actions that have been explored less.

# Upper Confidence Trees (UCT)

$$\text{UCT} = \text{MCTS} + \text{UCB1}$$

Kocsis & Szepesvári (2006) were the first to treat the selection of nodes to expand in MCTS as a multi-armed bandit problem.

## UCT exploration policy

$$\pi(s) := \underset{a \in A(s)}{\text{argmax}}\, Q(s,a) + 2C_p \sqrt{\frac{2 \ln N(s)}{N(s,a)}}$$

$C_p > 0$ is the exploration constant, which determines can be increased to encourage more exploration, and decreased to encourage less exploration. Ties are broken randomly.

$\rightarrow$ if $Q(s,a) \in [0,1]$ **and** $C_p = \frac{1}{\sqrt{2}}$ **then** in two-player adversarial games, UCT converges to the well-known Minimax algorithm (if you don't know what Minimax is, ignore this for now and we'll mention it later in the subject).

# What if we do not know $P_a(s' \mid s)$?

In the coming lectures, we will look more at situations in which we do not know $P_a(s' \mid s)$, but it is important to note that we can use MCTS even if we do not know our transition probabilities or our reward function, provided that we can *simulate* them; e.g. using a code-based simulator. This leads to a new approach based on a straightforward modification:

1. *Selection* is as before.
2. In the *expansion* step, instead of expanding all child nodes of an action, we run the simulation forward one step, which will choose an outcome according to $P(s' \mid s)$ (provided, or course, that the simulator is accurate).
3. We then simulate as before, and we learn the rewards when we receive them.
4. In the *backpropagation* step, instead of using the Bellman equation to calculate the expected return, we simply use the *average* return. If we simulate each step enough times, the average will converge to the expected return.

The disadvantage of this approach is that we have to do repeated simulations for the average to converge, whereas when we know $P_a(s' \mid s)$, we need only expand an action once to know its immediate effect.
The advantage is that it is more general: as long as we have a simulator for our problem, we can apply it – we do not need an explicit model of the problem. For many problem, simulators are easier to produce than problems.

# Applications of MCTS with UCB tree policies

**Games**:

- Go: MoGo (2006), Fuego (2009), ..., Alpha Go(2010–2016)
- Board Games: Havannah, Y, Cataan, Othello, Arimaa...
- Video Games: Atari 2600

**Not Games**:

- Computer Security: Attack tree generation & Penetration testing
- Deep Learning: Automated "performance tuning" of Neural Nets and Feature Selection
- Operations Research: Optimising bus schedules, inventory stock management, logistics operations,...

# Why does it work so well (sometimes)?

MCTS addresses exploitation vs. exploration comprehensively.

UCT is *systematic*:

- Policy evaluation is *exhaustive* up to a certain depth.
- Exploration aims at *minimising regret* (or FOMO).

Watch it playing Mario Bros.
(`https://www.youtube.com/watch?v=HRiEUUC9TUA`).

Where it does not do so well...: Atari 2600 game Freeway
(`https://www.youtube.com/watch?v=YVbTbMO4rtM`)

It fails for Freeway because the character does not receive a reward until it reaches the other side of the road, so UCT has no feedback to go on.

- Compare this with Simulated Best-First Width Search (BFWS)/IW algorithm from classical planning, which does much better as it does not rely on *rewards* but instead on *novelty* (see lecture: 5. Width Based Planning)

# Value/policy iteration vs. MCTS

Often the set of states reachable from the initial state $s_0$ using an optimal policy is much small that the set of total states. In this regard, value iteration and policy iteration are exhaustive: they calculate behaviour from states that may never be encountered if we know the initial state of the problem.

MCTS (and other search methods) methods thus can be used by just taking samples starting at $s_0$. However, the result is not as general as using value/policy iteration: the resulting solution *will work only from the known initial state* $s_0$ or any state reachable from $s_0$ using actions defined in the model. Whereas value/policy iteration methods work from any state.

# Value/policy iteration vs. MCTS

| | Value/policy iteration | MCTS |
|---|---|---|
| Cost | Higher cost (exhaustive) | Lower cost (does not solve for entire state space) |
| Coverage/ Robustness | Higher (works from any state) | Low (works only from initial state or state reachable from initial state) |

This is important: value/policy iteration are thus more expensive, however, for an agent operating in its environment, we only solve exhaustively once, and we can use the resulting policy many times no matter state we are
For MCTS, we need to solve *online* each time we encounter a state we have not considered before.

## Summary

- Monte Carlo Tree Search (MCTS) is an anytime search algorithm, especially good for stochastic domains, such as MDPs.
  - Smart selection strategies are *crucial* for good performance.
- Upper Confidence Bounds (UCB1) for Multi-Armed Bandits makes a good selection policy.
  - UCB1 (with slight modifications) balances exploitation and exploration remarkable well.
  - The Fear Of Missing Out is an *excellent* motivator for exploration.
- UCT is the combination of MCTS and UCB1, and is an very successful algorithm for many problems.
  - Yet it has obvious shortcomings, e.g. relies on sufficient reward signal to work, cannot handle dead-ends (at least in standard form), etc.
  - There are alternatives to FOMO to motivate exploration, such as $\epsilon$-greedy and softmax.