Monte Carlo Tree Search

In this exercise we will implement a simple version of Monte Carlo Tree Search for deterministic gym environments. We use a simple gym environment implemented in `tictactoe.py` which conforms to `ResettableEnv`. This means that we can get and set the environment's state, and thus use this environment to start a simulation from any state that we desire. Note that the environment is *deterministic*, and the adversary actions do not depend on the agent actions. The environment provides the following methods to help you:

1. `env.get_legal_actions(state) -> [action]` Returns all legal actions in a given state.
2. `env.step_state(state, action) -> new_state` Performs a single round of Tic-Tac-Toe in the environment, starting from `state`, and returns the successor state.

Programming Tasks:

1. **Preliminaries** First, we need to implement some methods that help to implement MCTS. Implement the following methods in `env_node.py`:
   (a) `expand()` In MCTS, we expand the search tree by expanding nodes. We expand a node by choosing an action that has not been performed in any other child node, simulating a state transition (with `env.step_state()`, and creating a new child node. Implement this behavior in `env_node.py:expand()`
   (b) `best_child()` When a node is fully expanded, MCTS uses a heuristic based on the previously gathered statistics to choose a child to explore from. This should be implemented in `env_node.py:best_child()`. The parameter `c_param` defines how to weigh between exploration and exploitation: if `c_param == 0`, MCTS will exploit and favor child nodes with a high average return. If `c_param` is large, MCTS will favor nodes that have not been visited often. Think about the UCB definition, i.e. $a \sim V_i + C\frac{ln(N)}{n_i}$.
2. **Tree Search** MCTS works by iteratively creating a search tree defined by the possible actions. We define the basic algorithm for MCTS in `search.py:best_action()`. Your task is to write code that selects the next node that MCTS should do a rollout from, starting from `self.root`. Tip: Use the node methods `is_terminal_node()`, `is_fully_expanded()`, `expand()`, and `best_child()`.
3. **Rollout** After the node is selected, MCTS performs a rollout from this node, and backpropagates the result from the node up the search tree. Implement this rollout in `env_node.py:rollout()`. Accumulate and return the total reward gained in the environment for the backpropagation.
4. **Backpropagation** Next, MCTS backpropagates the result of the rollout up the search tree. Implement this behavior in `env_node.py:backpropagate()`. Update `self._total_children_reward` and `self._number_of_visits`.
5. **Experiments** Run MCTS by running `search.py`. Try changing `c_param`, and observe the results.
6. **Non-deterministic environments** Right now, our implementation is only compatible with single-player deterministic environments. Why is this the case? What would we need to change for
   (a) two-player
   (b) non-deterministic

   environments? *You do not need to implement this.*