**Artificial Intelligence in Games**

**Session 2**

1. **A Survey of Monte Carlo Tree Search Methods**(Essential Reading notes):
   1. *Overview*:
      1. A tree is built in an incremental and asymmetric manner (the tree grows asymmetrically, exploring the most promising parts of the search space).
      2. For each iteration of the algorithm, a tree policy is used to find the most urgent node of the current tree. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas which appear to be promising).
      3. A simulation (a random or statistically biased sequence of actions applied to the given state until a terminal condition is reached) is then run from the selected node and the search tree updated according to the result. This involves the addition of a child node corresponding to the action taken from the selected node, and an update of the statistics of its ancestors. Moves are made during this simulation according to some default policy, which in the simplest case is to make uniform random moves.
   2. A great benefit of MCTS is that the values of intermediate states do not have to be evaluated, as for depth-limited minimax search, which greatly reduces the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is required.
   3. *Markov Decision Process (MDP)*: Models sequential decision problems in fully observable environments using four components:
      1. : A set of states
      2. : A set of actions
      3. : Probability of reaching the initial state if action is applied to state
      4. : A reward function
   4. *Q*-value: Expected reward of an action.
   5. *Bandit problems*: Bandit problems are a well-known class of sequential decision problems, in which one needs to choose amongst actions (e.g., the arms of a multi-armed bandit slot machine) to maximise the cumulative reward by consistently taking the optimal action.
2. **MCTS Algorithm**: involves iteratively building a search tree until some predefined computational budget – typically a time, memory, or iteration constraint – is reached, at which point the search is halted and the best performing root action returned. Each node in the search tree represents a state of the domain and directed links to child nodes represent actions leading to subsequent states.  
     
   Four steps are applied:
   1. *Selection*: Starting at the root node, a child selection policy is recursively applied to descend through the tree until the most urgent expandable node is reached. A node is expandable if it represents a nonterminal state and has unvisited (i.e., unexpanded) children.
   2. *Expansion*: One (or more) child nodes are added to expand the tree, according to the available actions.
   3. *Simulation*: A simulation is run from the new node(s) according to the default policy to produce an outcome.
   4. *Backpropagation*: the simulation result is “backed up” (i.e., backpropagated) through the selected nodes to update their statistics.

Each node of the tree represents a game state. Each node holds the following statistics:   
  
- N(*s*): Number of times the state *s* has been visited by the algorithm.   
- N(*s*, *a*): Number of times the action *a* has been played from state *s*.   
- Q(*s*, *a*): An estimation of how good it is to play action a from state *s*. These statistics are updated, in certain nodes, at every iteration. Q(*s*, *a*) is the average (i.e. an estimation) of rewards seen when playing action *a* from state *s*.  
  
Rewards (R) indicate how good a state is:   
  
- If the state is terminal (the game is over), this is simple: did I win? R = 1. Did I lose? R = 0.   
- If the state is not terminal, we need a function that tells us how good this is. This is typically a heuristic, designed with knowledge of the goal state in mind, R = f(*s*).  
  
  
  
  
  
  
  
  
  
  
  
  
**Step-by-step**:  
  
1. The algorithm starts with the current game state. Only one node in the tree. In this example, we assume there are 3 possible actions from each state.  
  
2. In the original algorithm, the selection step can only be applied when all children nodes from the current state have been visited. This is not the case at the start, so we move to the next step: expansion.  
  
 We choose one action (at random, or in a predefined order), use the Forward Model to roll the state forward to the next state, and add it as a new node to the tree.  
  
3. The playout (also called simulation or rollout) step plays actions uniformly at random from the state added at the previous step. This is known as the Default Policy.  
  
4. We capture the reward of the state at the end of the rollout. Either a victory condition or a heuristic on the state. The backpropagation phase takes this reward to the nodes visited in this iteration.  
  
N(*s*) = N(*s*) + 1

N(*s*, *a*) = N(*s*, *a*) + 1

Q(*s*, *a*) = Q(*s*, *a*) + (R - Q(*s*, *a*)) / N(*s*, *a*)  
  
5. The first iteration is now over. We still can’t do selection from the root node (only one action has been explored from this state). We expand, simulate and backpropagate in this second iteration.  
  
6. On the next iteration we can apply the selection step. This selection is made according to a Tree Policy. This tree policy is key in the MCTS algorithm. The objective is to balance between exploiting the best action seen so far and exploring actions that haven’t been seen often enough (thus our estimate could be inaccurate).  
  
The action must be selected maximizing the Upper Confidence Bound (UCB) of the form:  
  
a = Q(*s*, *a*) + U(*s*, *a*)

where, Q(*s*, *a*) is the value of the action ‘a’ in state ‘s’ (how good it seems to be), and U(*s*, *a*) indicates the uncertainty we have on the value of an action ‘a’ in state ‘s’ (U(s,a) is inversely proportional to the number of times action ‘a’ has been selected in state ‘s’ (N(s,a))).

The most popular, default in MCTS, is Upper Confidence Bounds (UCB1). The action *a* selected is the one that maximizes the following expression:

a = Q(*s*, *a*) + C times sqrt(natural logrithm N(*s*) / N(*s*, *a*))

where, Q(*s*, *a*) is the exploitation term and C is the exploration term (C = sqrt(2) if R is contained between the interval [0, 1]).  
  
7. After selecting a node, we continue with the next steps: expand a new node from there, rollout and backpropagation.  
  
8. We repeat this procedure iteration after iteration. Because of the tree policy, the tree grows asymmetrically, exploring the most promising parts of the search space.  
  
When to stop? When the budget is over. This can be time, or iterations, or usages of the forward model.  
  
9. When time or a number of iterations is up, MCTS stops and recommends one action a to execute in the real game.  
  
This can be the action visited more often, or the action with the highest Q(*s*, *a*) or UCB1

1. **Bandit-based Tree Selection**:  
     
   UCB1-Tuned:  
     
   The action *a* selected is the one that maximizes the following expression:  
     
   a = Q(*s*, *a*) + C times sqrt(natural logarithm N(*s*) / N(*s*, *a*)) \* min(1/4, V(*s*, *a*))  
     
   where,   
     
   V(*s*, *a*) = variance of the rewards + exploration of first *t* rewards.
2. **Bayesian UCT (B-UCT)**:  
     
   B-UCT1:  
   a = ua + sqrt(2 \* natural logrithm N(*s*) / N(*s*, *a*))  
   B-UCT2:  
   a = ua + sqrt(2 \* natural logrithm N(*s*) \* N(*s*, *a*) \* sigmaa)  
     
   where, each node in the tree holds a probability distribution ua over the true expected reward value.
3. **MCTS Enhancements**:
   1. *Transpositions*: Use a hash table, where the key is the identifier of the state, and the value is a pointer to the object that holds the state.   
        
      Every time a node needs to be updated or traversed; the object referenced in the transposition table is used instead of the copy in the tree. That way, statistics can be aggregated and centralized in a common place.
   2. *Progressive bias*: Adds domain specific knowledge to MCTS. This is especially useful when a node hasn’t been visited often enough and a game-dependant heuristic can help the selection step.
   3. *All Moves as First* (AMAF): Treats all moves played during selection and simulation steps as if they were played on a previous selection step.

The rationale of this is to reward those actions that are good to be taken independently of when are they taken. If the impact of these actions is not time-dependent and state dependent, the statistics gathered for each move can be based on more samples.  
  
alpha-AMAF is an AMAF variant that keeps the AMAF score (Aa) separate from the traditional UCT score (Q(*s*, *a*)). By defining a new parameter alpha, we can weight the strength of each value to define a new exploitation term:  
  
Qalpha-AMAF(*s*, *a*) = (alpha times Aa) + (1 – alpha) \* Q(*s*, *a*)   
  
Rapid Action Value Estimate (RAVE) is a particular variant of alpha-AMAF, where the value of *a* decays with each visit. Given N(*s*) visits to a certain node:  
  
alpha = the maximum value between 0 and (V – N(*s*) / V)