AI Programming (IT-3105)
Project Module #2:
Implementing Monte Carlo Tree Search (MCTS)

Purpose: Gain a deep understanding of MCTS by implementing a generic version, easily extendable to a wide variety of search problems.

1 Introduction

For many years, the standard AI algorithm for two-player games was Minimax with Alpha-Beta Pruning. IBM's Deep Blue used it to defeat Gary Kasparov in 1997, and up until very recently, it has been the core of AI chess players, the best of which now routinely beat top human players.

In addition to Minimax, these algorithms include a host of expert knowledge embodied in a) huge databases of book-move opening and end-game strategies, and b) complex heuristic functions, which often include hundreds of weighted factors, such as center control, piece advantage, pawn advancement, etc. All of this expertise seemed mandatory for an AI chess bot until December of 2017, when Google DeepMind's *Alpha Zero* system soundly defeated *Stockfish*, the reigning world-champion computer chess player. Alpha Zero employs no expert knowledge whatsoever. It acquired chess expertise by playing millions of games against itself, improving via a combination of Reinforcement Learning (RL) and Deep Learning (DL) (a.k.a. Deep Reinforcement Learning - DRL).

Alpha Zero, like its predecessors: AlphaGo (2016) and AlphaGo Zero (2017), uses Monte Carlo Tree Search (MCTS) as its core RL algorithm. Designers of AI Go-playing systems have long recognized MCTS as superior to Minimax for the game of Go for the simple reason that Go heuristics, on which Minimax crucially depends, have been extremely evasive: even Go experts cannot produce effective heuristics. In MCTS, instead of evaluating leaf nodes of the search tree with heuristics, a rollout simulation (using a default action-choice scheme, a.k.a. *policy*) produces a chain of states from the leaf to a final state, which is trivial to evaluate since it represents a win, loss or tie for the current player. MCTS performs hundreds or thousands of such rollouts and averages the results across the search tree as the basis for each move in the actual game. Over time, the evaluations of nodes and edges in the tree approach those achieved by Minimax, but without the need for a heuristic. MCTS only needs to know the rules of the game and how to evaluate final states.

In this project module, you will build a general-purpose MCTS system that supports game play in many domains. However, you will only need to demonstrate it on a single, very simple game (and one that is easily solved with Minimax as well). In a future module, you will apply your system to a second, more complicated game.

2 Your Kernel MCTS System

Details of the MCTS algorithm can be found in lecture notes for this class, in the article "Monte-Carlo tree search and rapid action value estimation in computer Go" (Gelly and Silver, 2011) – provided on the "Materials" section of the course web page – and in a host of online sources. The exact details of the MCTS algorithm that you decide to implement may vary slightly from these sources, but your code must perform these four basic processes:

1. Tree Search - Traversing the tree from the root to a leaf node by using the *tree policy*.
2. Node Expansion - Generating some or all child states of a parent state, and then connecting the tree node housing the parent state (a.k.a. *parent node*) to the nodes housing the child states (a.k.a. *child nodes*).
3. Leaf Evaluation - Estimating the value of a leaf node in the tree by doing a rollout simulation using the *default policy* from the leaf node's state to a final state.
4. Backpropagation - Passing the evaluation of a final state back up the tree, updating relevant data (see course lecture notes) at all nodes and edges on the path from the final state to the tree root.

Each process must exist as separate, modular unit in your code: a method, function or subroutine. These units should be easy to isolate and explain during a demonstration of your working system. In a future project module, you will have the option to include a critic for leaf evaluation, but in this assignment, a rollout is required.

2.1 Actual Games versus Rollout Games

Taken together, the four processes listed above constitute one *simulation* in MCTS. Whenever leaf evaluation involves a rollout, an entire *rollout game* is played to assess the value of that leaf node. The rollout game should not be confused with the *actual game* that the system is playing. Each actual game constitutes an RL *episode*, and each move in the actual game stems from the results of M simulations in MCTS, i.e. M pairs of tree search (to a leaf node) followed by a rollout.

Your system must include M as a user-specifiable parameter. Your laptop's computing power will determine the range of M's that your MCTS can handle, but for games involving 10 pieces (N=10), an M value of several hundred should be feasible.

2.2 The State Manager

One key to producing a general-purpose MCTS is the separation of game logic from search code. Thus, your code must contain a separate unit (preferably a class) specialized for a given domain (i.e. game). It should contain code that *understands* game states and a) produces initial game states, b) generates child states from a parent state, c) recognizes winning states, and many other functions associated with the domain. Your MCTS code will not make any references to a specific game but rather have generic calls to a state-manager object requesting start states, child states, confirmation of a final state, etc. Failure to follow this simple design principle will incur a (potentially serious) point loss during the demonstration.

2.3 Policy Learning

As described in the lecture notes, MCTS may involve three different policies: the tree policy guides behavior from tree root to a leaf node, the default policy (a.k.a. behavior policy) chooses actions during the rollout phase, and a target policy can be learned via repeated episodes (e.g., actual games) whose actions are governed by the MC tree.

In this project module, the only learning will involve the tree policy, whose behavior changes as Q(s,a) and u(s,a) values change. This entails that the intra-episode actions may appear more intelligent as further simulations (rollouts) are performed. However, since all of the tree information, including Q(s,a) and u(s,a) values, are discarded at the end of each episode, no inter-episode learning should occur.

In the next project module, you will add a target policy (and also use it as a behavior policy). Then, you should witness inter-episode learning as well. But in this module, the main route to improved performance is increasing the number of simulations (rollouts) per episode (parameter M, described above), thus producing a larger MC tree and enhancing tree-policy learning.

3 A Basic NIM Game

Believed to have originated in ancient China, NIM is actually a variety of games involving pieces (or *stones*) on a non-descript board that players alternatively remove, with the player removing the last piece being the winner, or loser, depending on the game variant). Typical constraints of the game include a minimum and maximum number of pieces that can be removed at any one time, a partitioning of the pieces into heaps such that players must extract from a single heap on a given turn, and boards with distinct geometry affecting where pieces reside, how they can be grouped during removal, etc.

For this project, a very simple version of NIM will consttitute the test domain for your MCTS. The 2-player game is defined by two key parameters: N and K. N is the number of pieces on the board, and K is the maximum number that a player can take off the board on their turn; the minimum pieces to remove is ONE.

Given N and K, the remaining rules of play are extremely simple: Players take turns removing pieces, and the player who removes the **last** piece is the **winner**.

3.1 Your NIM Simulator

By changing the values of N and K, we can produce many variations of the game, some of which are harder than others. **Your system must be able to play NIM with any values of N and K, where** $100 > N > K > 1$**.**

In addition, your system must include a toggle for *verbose* mode. When on, this insures that details of a game are printed to the command line or a graphics window. These play-by-play details should look something like this for a game where N=10, K=3:

- Player 1 selects 2 stones: Remaining stones = 8
- Player 2 selects 1 stone: Remaining stones = 7

- Player 1 selects 3 stones: Remaining stones = 4
- Player 2 selects 2 stones: Remaining stones = 2
- Player 1 selects 2 stones: Remaining stones = 0
- Player 1 wins

You are welcome to use a fancier format, but the critical information (stones taken, stones remaining) must be displayed for each turn, as must the winner of the game.

In addition, your NIM simulator must support *batch* runs of G consecutive games (a.k.a. episodes) – using fixed values for N and K – and then summarizing the win-loss statistics. A typical summary (for G = 50) would be a simple statement such as: *Player 1 wins 40 of 50 games (80%).* Since there are no ties in this version of NIM, all other useful statistics can be infered from such a statement. Your system should handle G as large as 100.

It must be possible to use *verbose* mode during a batch run. In this context, *verbose* indicates whether or not the details of individual moves should be displayed during all G games. The final statistics should be shown regardless of the value of *verbose*.

The first player to select a stone in each of the G games must be controlled by a parameter (P), with options:

- Player 1 - will start EVERY game in the batch
- Player 2 - will start EVERY game in the batch
- Mix - the starting player will be randomly chosen at the start of each game in the batch.

3.2 NIM Simulator Summary

The user-specified parameters that your NIM simulator must provide:

- G - number of games in a batch
- P - starting-player option
- M - number of simulations (and hence rollouts) per actual game move.
- N - starting number of pieces/stones in each game
- K - maximum number of pieces that either player can take on their turn.

The results that it must provide:

- Play-by-Play game action when in *verbose* mode.
- Essential win statistics (for at least one of the two players) for a batch of games.

3.3 Important Fact

Given the values of N and K, one of the two players can guarantee a win. Runs in batch mode with a fixed starting player will often have very skewed final results, e.g. Player 1 wins 90%. A properly-working MCTS should eventually discover these guaranteed-win strategies, though it may take several actual games to do so (and thus, several hundred or thousand rollout games). For larger values of N (e.g. $N > 15$), your MCTS may struggle to find optimal strategies due to the enormous search space. But for moderate values of N, the results of batch runs with a fixed starting player give a good indication of the proper functioning of your MCTS. **This indicator will come into play during our evaluation of your project during the demonstration.**

4 Deliverables

1. Well-structured code that you can discuss and explain (in general and in detail) with a student assistant. (**10 points**).
2. The ability of your system to learn successful strategies for the game of NIM during the demonstration, with the parameter settings for NIM specified by the student assistant. (**10 points**)

A zip file containing your commented code must be uploaded to BLACKBOARD directly following your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 20 total points for this module are 20 of the 100 points that are available for the entire semester.