Monte Carlo Tree Search for the Game of Hex

Xiyu Xie 580 Report

Content

Motivation

The game of Go has been a notorious AI challenge for it huge search space in tree-based planning method, comparing with other two player games, say chess, . The new algorithms, Monte-Carlo tree search (MCTS), has revolutionised the performance of computer Go programs. Hex is the modern variation of Go, invented by mathematician Piet Hein and John Nash in 1940th. It has some nice math properties and simple rules. In this report, I will use Hex as case study to illustrate and implement the main idea of MCTS algorithms, (pattern learning in GO game is ignored here). .

History (1)

Game Tree Search

Game tree search is the general AI platform to solve two players game, (only deterministic policy and perfect information are considered in this report). A game tree organized the all the possible future action sequences. The root node is the initial state, the leaf is the end state of the game with a final reward. The goal is find a top-down path from root to the leaf with the optimal rewards for a particular player. Reward for each node is backtrack from its children. It is a complete search algorithm actually. Minmax search is used for two playes game. Some pruning methods are used to reduce search space, with heuristic evaluation function instead of real reward from leaf node in iterative deepening.

Monte-Carlo Simulation (MC)

One way to estimate a value or policy is through simulation. Self-play, randomly choose actions until the end of game, and estimate value of the sate by taking average/max these results. Monte-Carlo methods is only used to evaluate actions, but not to improve the simulation policy.

Monte-Carlo Tree Search (MCTS)

Monte-Carlo Tree search is the combination of the these two, it use Monte-Carlo simulation to evaluate the nodes of a search tree. It have four phases:

Descent phase, tree policy, iteratively choose the child with the highest action value;

Roll-out phase, simulation policy, with a fixed stochastic policy, simulation action for both players until a terminal state. This is the default policy;

Update phase, action value for each node visited during descent phase are updated according to results of simulation, the number of wins over the number of visited;

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i, \tag{MC}$$

It is really useful to use the incremental equation to each simulation

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1, \\ N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1, \\ Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{N(s_t, a_t)}. \end{aligned} \tag{Update}$$

Growth phase, the first state visited in the roll-out phase is added to the tree.

Figure. 1 is selected from ref 3, illustrate how this steps work.

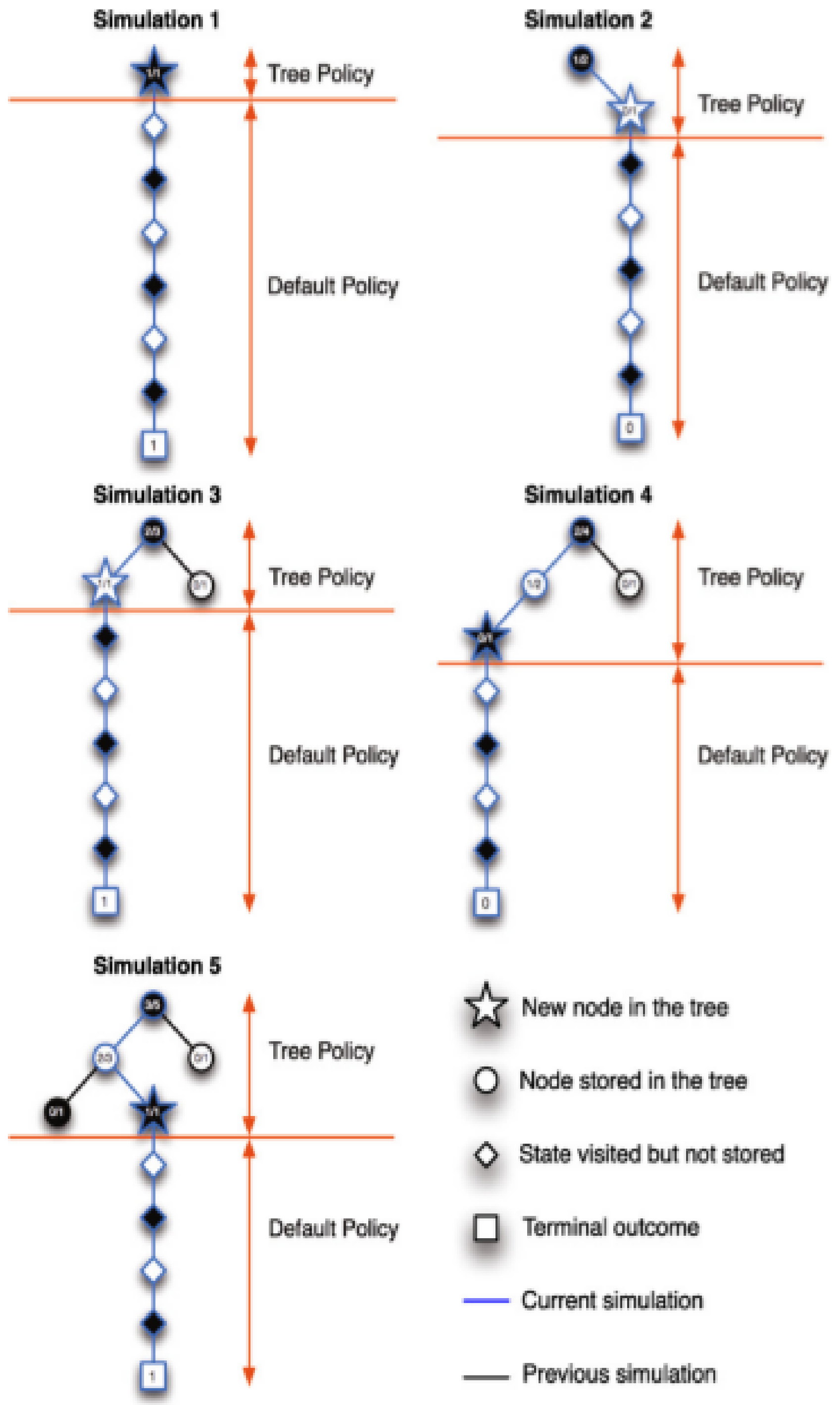

Fig. 1. Five simulations of a simple Monte-Carlo tree search. Each simulation has an outcome of 1 for a black win or 0 for a white win (square). At each simulation a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) is then updated to count the number of black wins, and the total number of visits (wins/visits).

Upper Confidence Bounds on Trees (UCT) algorithm

Idea here is to balance the greedy selection and random exploration in tree selection stage, the same as bandit problems, treating each child as an arm. It is called UCT algorithm here. Action value, Z(s,a) for a pair of state s, and action a is obtained by

$$Z(s,a) = \frac{W(s,a)}{N(s,a)} + B(s,a), \qquad B(s,a) = C\sqrt{\frac{\log N(s)}{N(s,a)}}, \tag{UCT}$$

Where N(s, a) is number of simulations selected from state s with action a,
W(s, a) is total reward collected at for these round of simulation
C is a exploration constant
$N(s) = \Sigma_a N(s, a)$ is the number of simulation from state s
B(s, a) is exploration bonus, it encourages exploration to less visited states

UTC algorithm is consistent, converge to minmax action value function, and empirically proven to be successful.

Rapid action-value estimation (RAVE)

In UCT algorithm, individual simulations for actions given the same state are required. One key observation in incremental games, such as Go and Hex, is the value of an action may not be critically depend on the state in which it was played. Rapid action-value estimation (RAVE) used this idea, which is called all-moves-as-first (AMAF) heuristic. That is, the value of action a is estimated from all simulations involving it at any time after s is encountered. The modified action value, called AMAF value is:

$$\tilde{Q}(s,a) = \frac{1}{\tilde{N}(s,a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a) z_i, \tag{AMAF}$$

Where $\tilde{N}(s, a)$ is the AMAF visit count, the number of simulations where a happened after s. AMAF value is used in tree policy to selection action which gives the largest AMAF value.

From ref 3

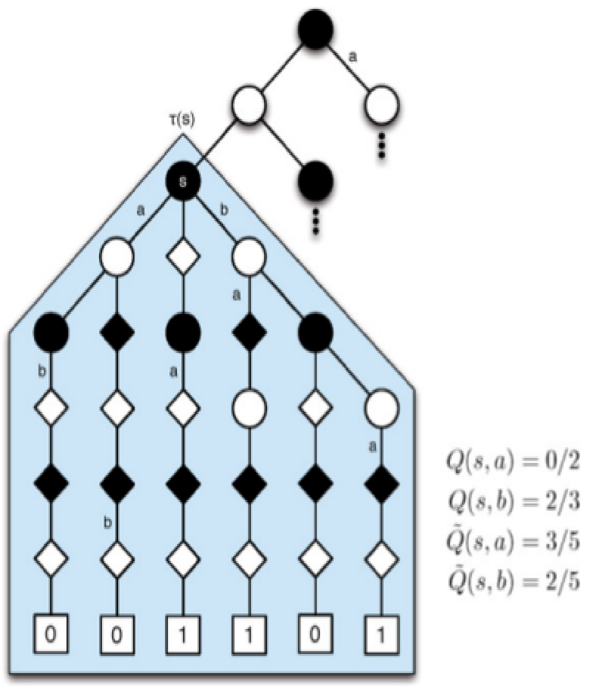

Fig. 4. An example of using the RAVE algorithm to estimate the value of Black moves a and b from state s. Six simulations have been executed from state s, with outcomes shown in the bottom squares. Playing move a immediately led to two losses, and so Monte-Carlo estimation favours move b. However, playing move a at *any* subsequent time led to three wins out of five, and so the RAVE algorithm favours move a. Note that the simulation starting with move a from the root node does not belong to the subtree $\tau(s)$ and does not contribute to the AMAF estimate $\tilde{Q}(s,a)$.

MC-RAVE

RAVE is a fast algorithms giving a rough estimate, based on the assumption, action value remain unchanged in subtree. That is open not true, as nearby or remote changes will affect the value of a action.

MC-RAVE is the weighted sum of MC action value and AMAF action value

$$Q_\star(s,a) = \left(1 - \beta(s,a)\right) Q(s,a) + \beta(s,a)\, \tilde{Q}(s,a) \qquad \text{(MC-RAVE)}$$

Q(s,a) is MC action value given in MCTS section
Q~(s, a) is AMAF action value given in RAVE section
β(*s, a*) is a weighting parameter that decays from 1 to 0, as simulation goes. So it will asymptotically converge to MCTS estimate.
In way to choose *β*(*s, a*) given in ref 3 Algorithm 2 is :

$$b = pretuned\ constant\ bias\ value$$
$$\beta = \frac{\tilde{N}(s,a)}{N(s,a) + \tilde{N}(s,a) + 4N(s,a)\tilde{N}(s,a)b^2}$$

Incorporate the UCT algorithm with an exploration bonus, as mentioned above

$$Q_\star^\oplus(s,a) = Q_\star(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}. \qquad \text{(UCT-MC-RAVE)}$$

The game of Hex (2)

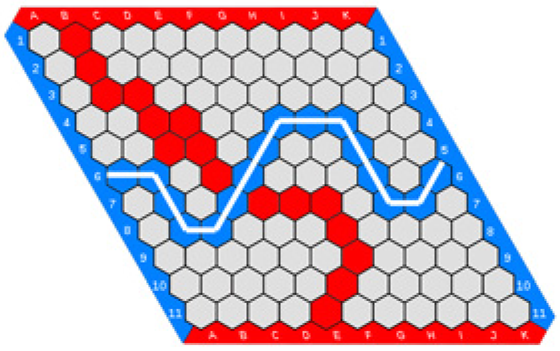

Figure 1. 11×11 Hex gameboard showing a winning configuration for Blue

Each player has an allocated color, Red and Blue as in Figure 1. Players take turns placing a stone of their color on a single cell within the overall playing board. The goal for each player is to form a connected path of their own stones linking the opposing sides of the board marked by their colors, before their opponent connects his or her sides in a similar fashion. The first player to complete his or her connection wins the game. The four corner hexagons each belong to both adjacent sides.

Experiment

Experiment is run on a Hex board with 7*7 size, with MC- RAVE algorithm described above and in ref 3 Algorithm 2.
Note, simulation action is only considered for one of the players, see code for detail.
Whether win or not is tested by connectivity algorithm. In this application, I used depth-first-search for simplicity. Shortest path algorithms would be preferable.

One important note, is AMAF value is also updated for unTree nodes, this maybe different from ref 3 Algorithm 2, based on my understanding.

Combining factor beta is dummy coded as

    beta *= 0.99999

so it will decrease with iteration

Games are played with a adversary who randomly choose a legal move.

Experiment are performed to explore the influence of

    Roll_iter: number of random self-play following a tree node in roll-out phase;
    Learn_iter: number of iterations restart from root, for continiously update of action value

Experiment are repeat 50 time, average value of success are considered.

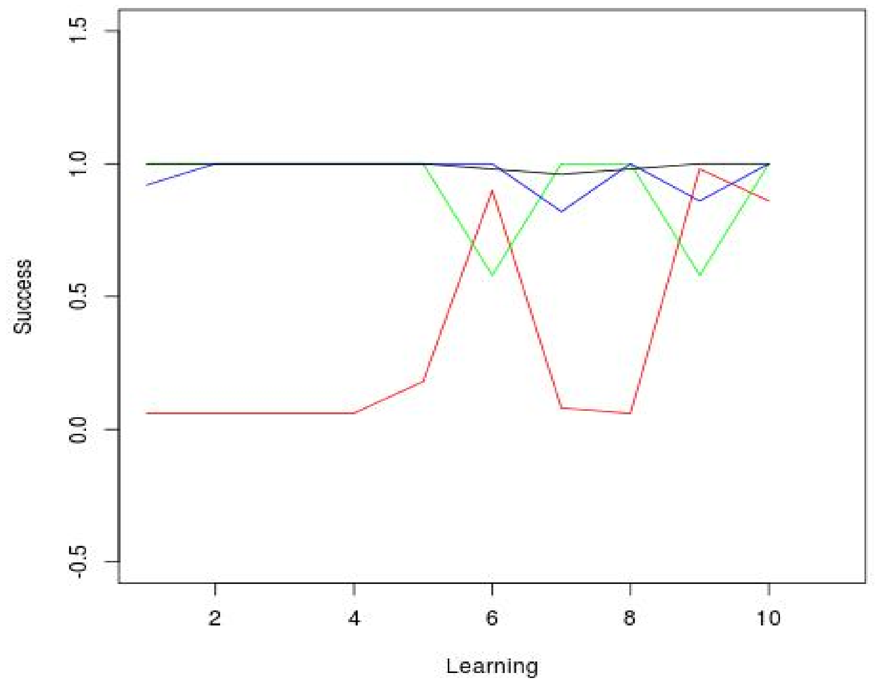

Figure 1. Plot of Success Vs Roll_iter and Learning_iter
X is Learning Iteration
Red with Roll_iter = 1; green Roll_iter = 2, blue Roll_iter=5; Black Roll_iter =10

Future Work

There are some drops of success in the middle of learning iteration, which is not expected.

I think it is because I didn't actually implement a good learning algorithms, such as UCT-MC-RAVE, or other reinforcement learning algorithm discussed in our class. So it is some thing to be completed. (it could also be coding error)

Some simple algorithm, just averaging MC action, without considering a tree structure works also quite well. (Data not show). So it could be interesting to think deep over the relations between tree based AMAF vs set configuration.

Reference

1. Gelly, Sylvain, et al. "The grand challenge of computer Go: Monte Carlo tree search and extensions." *Communications of the ACM* 55.3 (2012): 106-113.
2. http://en.wikipedia.org/wiki/Hex_%28board_game%29
3. Gelly, Sylvain, and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go." *Artificial Intelligence* 175.11 (2011): 1856-1875.