

Abstract After last check point, we have already implemented Alpha-Beta Pruning algorithm on Gomoku AI. This time we construct a brand-new AI with the aid of MCTS, and inspired by the online learning Multi-Armed Bandits (MAB) scenario[2][3], we implement the UCT version at last, which is an improvement of HMCTS algorithm.

1 Introduction

MCTS is a well-known framework for board game AI. The well-known Go AI AlphaGo takes this framework and have acknowledged achievement, beating top Go player in the world. This time we apply this method on Gomoku AI. The basic MCTS framework has four stages[2], Selection, Expansion, Simulation, and Backpropagation. Unlike Alpha-Beta Pruning algorithm, MCTS involves less operations in one simulation, i.e. using simpler policy to simulate the game. As central limit theorem, with enough simulation the approximate of the winning rate of each move will be good enough. So base on these facts, the HMCTS algorithm[1] has the theoretical foundation. But as HMCTS base on the so-called "point estimation", it needs sufficient simulations to provide good enough approximation of the true expect winning rate of each step, where "sufficient simulation" might be a prodigious work for personal computer. So as mentioned in [1], we should apply UCT algorithm to give a confidence interval of each approximation.

In fact this idea comes from the Multi-Armed Bandit scenario[3][4], and it could simply be seen as the so-called UCB algorithm, and the HMCTS could be transfered into the previous edition, ETC algorithm, so the advantage of UCT (UCB) is that it doesn't needs to tuning the rounds of simulations (and it needs the knowledge of gaps[3][4]), and still provides the same asymptotic bounds as HMCTS (ETC) does.

The organization of the report is as follows. In the next section we give a precise description of our algorithms, and follows by the description, we will give a concise explanation of the difference between HMCTS and UCT, together with some simple analysis learned from [3][4]. In the fourth section we will show the optimization we take to boosting the performance of our AI. We will illustrate our experiment result in the fifth section and in the last one, we will conclude several works could be done in the future.

2 HMCTS and UCT algorithm

2.1 General Framework

The general Frame work of Monte Carlo Tree Search (MCTS) involves four stages, Selection, Expansion, Simulation and Backpropagation. The followings are the concise explanation of each steps[2].

- *Selection* Starting from root node, selection means a policy of each node to select its child until we reach a terminal stage.
- *Expansion* Adding one or more child nodes to expand the tree, according to the available actions.
- *Simulation* It means run from the new nodes

according to certain policy.

- *Backpropagation* Returns the simulation all along the way back to the selected nodes to update their statistics.

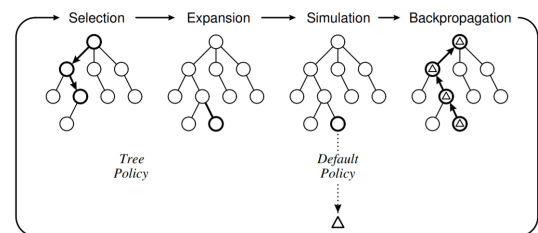


Figure 1: One iteration of the general MCTS approach[2]

To make the above framework works, the policy of each step is crucial. The first algorithm Heuristic Monte Carlo Tree Search, a.k.a HMCTS, is a nice try based on it.

2.2 HMCTS algorithm

The general algorithm pseudocode is as follows

Algorithm 1 HMCTS algorithm

Input: original state s_0 , Heuristic Knowledge h

Output: action a with the highest value of MCTS

```

1: Get possible moves set  $M$  of state  $s_0$ 
2: for move  $m$  in  $M$  do
3:    $R_m \leftarrow 0$ 
4:   while simulation times  $t < T$  do
5:      $R_m \leftarrow R_m + \text{Simulation}(f(s_0, m), h)$ 
6:      $t \leftarrow t + 1$ 
7:   end while
8:   add  $(m, R_m)$  into data
9: end for
10: return action  $m^* = \arg \max_{m \in M} R_m$ 

```

Algorithm 2 Simulation

Input: Original state s_0 , Heuristic Knowledge h

Output: Reward of this state R

```

1: if  $s_0$  is terminal then
2:   if  $s_0$  is win then
3:     return 1.0
4:   else if  $s_0$  is lose then
5:     return 0.0
6:   else
7:     return 0.5
8:   end if
9: end if
10: if  $s_0$  satisfies  $h$  then
11:   obtain forced move  $a$ 
12:    $s_{t+1} \leftarrow f(s_0, a)$ 
13: else
14:   get possible moves set  $M$ 
15:   randomly choose one action  $a$  from  $M$ 
16:    $s_{t+1} \leftarrow f(s_0, a)$ 
17: end if
18: return  $\text{Simulation}(s_{t+1}, h)$ 

```

Where f is a function of the board, given state s and action a to attain the new state s_{new} .

2.3 Heuristic Knowledge

Inherent from check mate function (or seek must) function in our last check point, the Heuristic Knowledge is a simple policy for AI to check whether

there are some steps it must take. During the whole *Simulation* routine, if we choose every step randomly, we will easily run into some cases that is unreasonable, such as letting the opponent win from a single block four threat. To simply prune these unreasonable occasions, we have the following rules:

- If we have a chance to win, we must take it
- If opponent have a chance to win, we must try to block it
- If we have a chance to make four-in-a-row, we will make it
- If the opponent have a chance to make four-in-a-row, we simply block it

Moreover, in the following part, we will take another kind of heuristic knowledge, which is the scoring technics in our Alpha-Beta Pruning, to heuristically shrink the state space of each round of simulation.

2.4 Scoring the board

The score table for each pattern just goes like below










Pattern Type	Definition	Value
FIVE		10000000
FOUR		100000
BLOCKED_FOUR		10000
THREE		1000
BLOCKED_THREE		100
TWO		100
BLOCKED_TWO		10
ONE		10
BLOCKED_ONE		1

Table 1: Score For Each Pattern

The score of a move is evaluated based on how well it attack and how much it weaken the opponent. Here We divide dots into two categories, empty points and colored points.

For assigning a value to a colored point, we use the radius of 6 grid, scan its horizontal(h), vertical(v), and diagonal directions(d1/d2). Calculate

patterns with the same color formed in each direction respectively, and take the largest in that direction. Finally, sum up the scores in the four directions to get a colored point score.

For assigning a value to an empty point, we assume it's black and white and calculate the score respectively. Thus it has a positive white score for the enemy and black score for AI player.

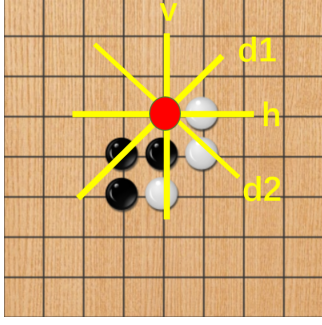


Figure 2: An example of evaluating an empty point

In this example the white score of red point is:

$$\begin{aligned} S_{white} &= S_{TWO} \times 2 + S_{BLOCK_ONE} \times 2 \\ &= 100 \times 2 + 1 \times 2 \\ &= 202 \end{aligned}$$

Besides, the black score of red point is:

$$\begin{aligned} S_{black} &= S_{BLOCK_TWO} + S_{TWO} + S_{BLOCK_ONE} \times 2 \\ &= 10 + 100 + 1 \times 2 \\ &= 112 \end{aligned}$$

We also inherit all the quick update strategy in Alpha-Beta Pruning to avoid too much cost on the simulation step.

2.5 UCT algorithm

As we mentioned in introduction, HMCTS algorithm has a large drawback that there is a huge work load for personal computer to provide point estimator of winning rate of each move. So here in UCT algorithm, we use a confidence interval to upper bound the estimator, which could have a better trade-off between "exploration and exploitation". The routine will be like that

3 Multi-Armed Bandit

Multi-Armed Bandit (MAB) is a famous scenario in online learning, it's basic setting is in each round, we could check one arm of the bandit, and receive corresponding reward due to an unknown distribution (*stochastic bandit* setting). Our goal is to find the best arm, or in online learning, minimize our expect regret. We will show two algorithms of this problem, ETC and UCB algorithm, which could be translate into HMCTS and UCT algorithm. Moreover, we will show some analysis of them (due to the page limit, we will not show the proof of it, you could find them in our reference).

Algorithm 3 UCT algorithm

Input: original state s_0 , Heuristic Knowledge h
Output: action a with the highest value of UCT

- 1: **while** within computational budget **do**
- 2: $m \leftarrow \text{Treepolicy}(s_0)$
- 3: $r \leftarrow \text{Simulation}(f(s_0, m), h)$
- 4: Update(m, r)
- 5: **end while**
- 6: **return** BestMove(s_0)

Algorithm 4 Treepolicy

Input: original state s_0
Output: Act m

- 1: Get possible moves set M
- 2: **if** M hasn't been fully expanded **then**
- 3: **return** any unexpanded move m
- 4: **else**
- 5: **return** BestMove(s_0)
- 6: **end if**

Algorithm 5 BestChild

Input: original state s_0
Output: best move m^* of state s_0

- 1: Get possible moves set M
- 2: **return** $\arg \max_{m \in M} \frac{Q(m)}{N(m)} + c\sqrt{2\frac{\ln N(m)}{N}}$

Algorithm 6 Update

Input: move m and its reward in one round of simulation r

- 1: $Q(m) \leftarrow Q(m) + r$
- 2: $N(m) \leftarrow N(m) + 1$
- 3: $N \leftarrow N + 1$

In the simulation of UCT, we also expand and choose the child node with maximum UCB value. To show the advantage of UCT, we will take a glimpse at the origin of them, the Multi-Armed Bandit scenario in the next section.

3.1 ETC algorithm

The easiest idea of this problem is, we simply try each arm several times to give an estimator of it's expect.

To analyze this algorithm, we should take a look at the word "regret". In stochastic bandit setting, regret means

$$\begin{aligned} \text{Regret}_T &:= \mathbb{E} \left[\sum_{t=1}^T g_{t,A_t} \right] - \min_{i=1,\dots,d} \mathbb{E} \left[\sum_{t=1}^T g_{t,i} \right] \\ &= \mathbb{E} \left[\sum_{t=1}^T g_{t,A_t} \right] - \min_{i=1,\dots,d} d\mu_i \end{aligned}$$

Then in [3][4] we know the analysis for it's regret is (under some mild assumption)

$$\text{Regret}_T \leq m \sum_{i=1}^d \Delta_i + (T - md) \sum_{i=1}^d \Delta_i \exp \left(-\frac{m\Delta_i^2}{4} \right)$$

We could see that inside this upper bound, we need the actual knowledge of gaps Δ_i to tuning parameter m for optimal bound, which is unreasonable in this setting because it makes this problem completely trivial.

This algorithm is not completely the same as HMCTS, but we could see the left term in the upper bound as the cost for exploration of new moves. This idea is almost the same in HMCTS, we need to pay something to search for the best arm, which leads to more regret in exploration term.

Algorithm 7 ETC algorithm

Input: $T, m \in \mathbb{N}, 1 \leq m \leq \frac{T}{d}$

- 1: $S_{0,i} = 0, \mu_{0,i} = 0, i = 1, 2, \dots, d$
 - 2: $t = 1$
 - 3: **for** $t \leq T$ **do**
 - 4: Choose $A_t = \begin{cases} (t \bmod d) + 1 & t \leq dm \\ \arg \max \mu_{dm,i} & t > dm \end{cases}$
 - 5: Observe g_{t,A_t}
 - 6: $S_{t,i} = S_{t-1,i} + \mathbf{1}[A_t = i]$
 - 7: $\hat{\mu}_{t,i} = \frac{1}{S_{t,i}} \sum_{j=1}^t g_{j,A_j} \mathbf{1}[A_j = i], i = 1, \dots, d$
 - 8: $t = t + 1$
 - 9: **end for**
-

3.2 UCB algorithm

The drawback of the ETC algorithm is that it can not has a good balance between exploration and exploitation without knowledge of gaps Δ_i . To modify this disadvantage, we could change our estimation of one arm from point estimation to confidence interval. That is to say, we balance exploration, which we try to figure out which arm is good, and exploitation, that means we use the selected arm to gain reward, in a dynamic way. Because confidence interval means with enough rounds of exploration, we could bound the actual expect reward of one arm with a probability. By comparing the upper confidence bound, we could have a smooth transition between exploration and exploitation (without tuning the exploration phase like ETC does). Here is the pseudocode of it

Algorithm 8 UCB algorithm

Input: $\alpha > 2, T \in \mathbb{N}$

- 1: $S_{0,i} = 0, \hat{\mu}_{0,i} = 0, i = 1, \dots, d$
 - 2: **for** $t < T$ **do**
 - 3: $A_t = \arg \max_{i=1,\dots,d} \begin{cases} \mu_{t-1,i} + \sqrt{\frac{2\alpha \ln t}{S_{t-1,i}}} & S_{t-1,i} \neq 0 \\ -\infty & \text{otherwise} \end{cases}$
 - 4: Observe g_{t,A_t}
 - 5: $S_{t,i} = S_{t-1,i} + \mathbf{1}[A_t = i]$
 - 6: $\hat{\mu}_{t,i} = \frac{1}{S_{t,i}} \sum_{j=1}^t g_{j,A_j} \mathbf{1}[A_j = i], i = 1, \dots, d$
 - 7: **end for**
-

Under some mild assumption, we have the following two bounds for UCB

$$\text{Regret}_T \leq \frac{\alpha}{\alpha - 2} \sum_{i=1}^d \Delta_i + \sum_{i:\Delta_i > 0} \frac{8\alpha \ln T}{\Delta_i}$$

$$\text{Regret}_T \leq 4\sqrt{2\alpha d T \ln T} + \frac{\alpha}{\alpha - 2} \sum_{i=1}^d \Delta_i$$

So compare bounds in ETC and UCB we could see that, without any knowledge of gaps Δ_i , the UCB could achieve the same asymptotic expect regret bound as UCT.

4 Optimization

4.1 Node structure

We construct the *Node* data structure to store UCB value of each stage, number of visit, parent, child and player information. With the aid of this data structure, we could construct our MCT tree, and lay a firm foundation for optimization following.

4.2 ϵ -Greedy

In our simulation, it is easy to see that we greedily choose the child node with biggest UCB value, this policy make sense in the first glance, however, in fact, the simulation process is just an approximation to the real world, which means both of the players takes the same policy we imagine, and this policy even not that make sense as we did in Alpha-Beta pruning (we sacrifice some of them for quicker simulation). Therefore, continuing select the son with biggest UCB value might cause so-called "Over-fitting", the win rate approximation might perform too good for the given policy, but might perform way worse than the real world. So we import ϵ -greedy method, by selecting some other moves randomly with a probability. Which could effectively avoid the case the AI being trapped inside the given simple policy. In our implementation we take the following type of move

selection policy

$$\pi^\epsilon(s) = \begin{cases} a, & P = \frac{\epsilon}{|A|} \\ \operatorname{argmax}_a UCB(a), & P = 1 - \epsilon \end{cases}$$

In our program $|A|$ means the number of moves in structure *activemove*. After we import this selection policy, we see a prominent improve in performance.

4.3 Quick activemove update & withdraw

In our last report, we conclude one future work, which is quickly update the active move in each alter on board. This time we implement the *add_move* module, inside the module, we update the active move for just searching the area near our new move to check if there are something new could be added inside, which avoid scanning the whole board everytime we change the board. This optimization highly accelarating our algorithm. We also import some optimization in the withdraw module, which is a more tricky optimization. As all the active moves in the parent node will be expanded, so we could simply recover the active move by taking all the brother nodes and this node inside the activemove structure to recover it to the previous stage.

5 Experiments

—	mix1		mix1
pisq7	0 : 12	Yixin2018	0 : 12
Noesis	0 : 12	Eulring	1 : 11
Wine18	0 : 12	valkyrie	6 : 6
Sparkle	0 : 12	fiverow	6 : 6
zetur2017	0 : 12	PureRocky	6 : 6
pela	0 : 12	mushroom	11 : 1

6 Future work

6.1 Natural Selection

Notice that we have plenty of parameters inside our program, the job of tuning is laborious. But we could use a natrual selection way, to autonomous tuning the parameters. We simply set up some battle between current AIs, kill the AI that lose the game and generate new AIs from the winner. In the process we could involve some survival pres-

We could see that our AI could easily defeat MUSHROOM, and could match PUREROCKY, FIVEROW, VALKYRIE. Obviously it works not as good as Alpha-Beta Pruning because to make the whole frame work work better, it needs some help from neural network for better policy (which is what AlphaGO did). But actually in future work section, we still got some more jobs to do.

sure and variation for better performance.

6.2 Policy network & Value network

In the work of AlphaGo we could easily see that they take advantage of neural network to support their MCTS, and using reinforcement learning to adjust all the parameters inside networks[5]. At this time we are not equipped with knowledge of

neural network, but with some knowledge of reinforcement learning, we are looking forward to apply some of the strategy from it to GOMOKU in next check point, and maybe obtain some idea from the neural network background.

References

- [1] Zhentao Tang, Zhao, D., Kun Shao, Le Lv. (2016). ADP with MCTS algorithm for Gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI).
- [2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, pp. 1-43, 2012.
- [3] Francesco Orabona, "A Modern Introduction to Online Learning"
- [4] Tor Lattimore and Csaba Szepesvári, "Bandit Algorithms", Cambridge University Press
- [5] Chang, H.S., M.C. Fu, J. Hu, and S.I. Marcus. 2016. "Google Deep Mind's AlphaGo". OR/MS Today.
- [6] L. Allis, H. Herik, and M. Huntjens. Go-moku and threat-space search. Computational Intelligence, 12, 10 1994.
- [7] Pearl, Judea. "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality." Communications of the ACM 25.8 (1982): 559-564.
- [8] Albert Zobrist (1970). A New Hashing Method with Application for Game Playing. Technical Report, Computer Science Department, The University of Wisconsin, Madison, WI, USA.
- [9] J. Lawrence Carter, Mark N. Wegman (1977). Universal classes of hash functions. STOC'77