CS584 FINAL PROJECT REPORT

IMPLEMENTATION OF MONTE CARLO SEARCH TREE USING TIC TAC TOE GAME

Liu, Cong (A20343692)

Liu, Man (A20355651)

DEPARTMENT OF COMPUTER SCIENCE

ILLINOIS INSTITUTE OF TECNOLOGY

Content

1. Abstract

Monte Carlo algorithm is a randomized algorithm [1]. Monte Carlo Search Tree (MCST) is a method using Monte Carlo algorithm for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games like Go games [2]. MCTS combines the generality of random simulation with the precision of tree search. Researches interest in MCTS has risen sharply due to its spectacular success with computer Go and potential application to a number of other difficult problems. MCTS can theoretically be applied to any domain that can be described in terms of {state, action} pairs and simulation used to forecast outcomes. We implement MCST algorithm using Upper Confidence Bounds (UCB) and train this tree in a Tic Tac Toe game.

2. Problem statement

Monte Carlo Search Tree is typically a game tree. The structure is shown as Figure 1.

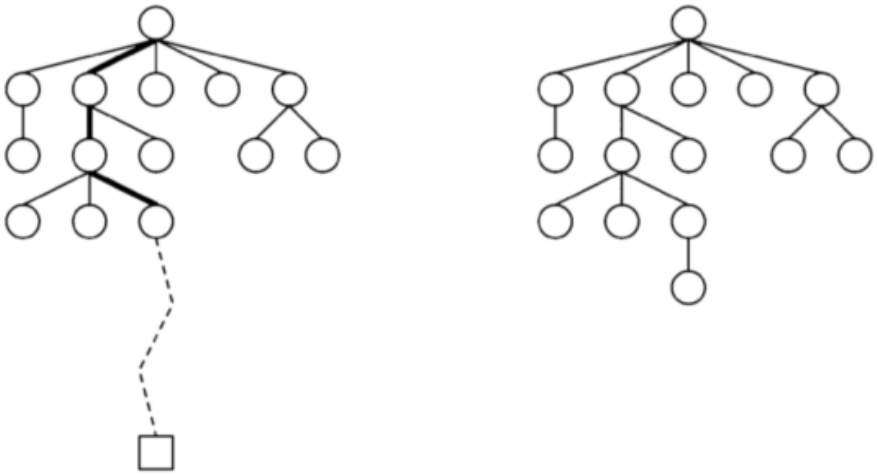

Fig 1. The basic MCTS process [3]

MCST rests on two fundamental concepts: first, the true value of an action may be approximated using random simulation (Monte Carlo algorithm); second, the values can be used to justify the next action using best-first strategy. This algorithm will build a partial game tree. After each game finishing, the values of the tree will update by the result of the game. That means the more games it played, the better

action or efficiency it could make.

2.1 MCST Algorithm

In this algorithm, it will iteratively to build a search tree until some computational budget is reached. And then return the best performing root action.

For each iteration, with in the computational budget the algorithm searches the children of each node forms a tree policy or a default policy. These two policies attempts to balance exploration and exploitation of MCST. Exploration is to look in areas that have not been visited before, exploitation looks in areas with appeared before. Each node has two basic properties.

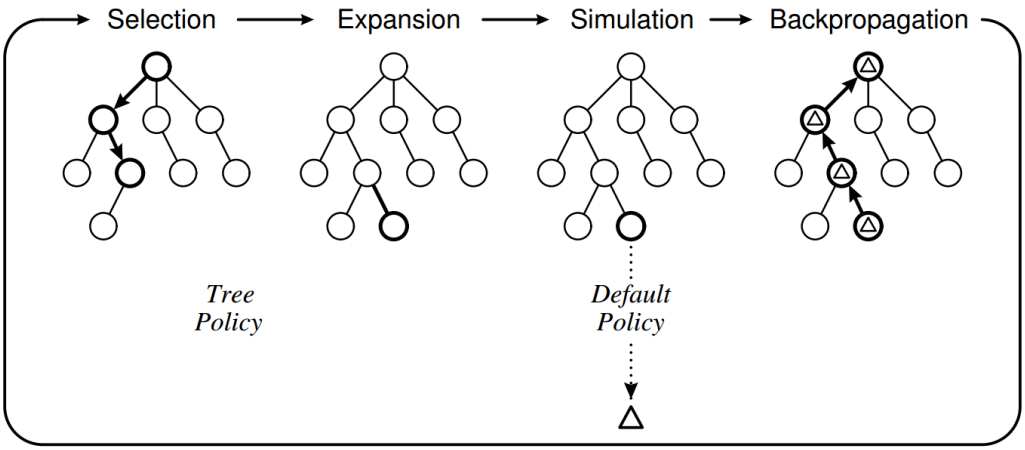

Fig 2. One iteration of the general MCTS approach [4]

Each iteration needs four steps to apply: [4]

1) Selection

Starting from the root node, when within computational budget, select a child node depends on a tree policy until reach an unexpanded child.

2) Expansion

One (or more) child nodes are added to the expanded tree.

3) Simulation

A simulation rule runs on the expanded node, then outcome a reward value according to default policy.

4) Backpropagation

The reward of the nodes along the chosen path return back to root node to update its statistics.

The two policies mentioned above are:

1) Tree policy: Select or create a leaf node in the search tree (selection and expansion)

2) Default policy: Applied on a node (or nodes) that isn't existed in this search tree yet to give an estimate value.

The backpropagation steps will update the node statistics for future tree policy decisions.

After the computation budget is reached, the search terminates and an action a of the root t_0 is selected based on the reward of its children.

2.2 Upper Confidence Bounds for Trees (UCT)

Multi-armed bandit problems (K-armed bandit problems) are known as a class of sequential decision problems [5,6]. It is a kind of problems in which a gambler at a row of slot machines decides which machine to play, how many times to play each machine and in which order to play them. The aim of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls [6]. The object of Upper Confidence Bounds(UCB) is to find the optimal arm. The simplest policy is proposed by Auer et al. [6] called UCB1. The function is to find the arm j that maximize:

$$\mathrm{UCB1} = \overline{X}_j + \sqrt{\frac{2\ln n}{n_j}}$$

where $\overline{X}_j$ is the average reward from arm j, and n_j is the number of times arm j was played and n is the overall number of plays so far[6].

The Upper Confidence Bounds for Trees (UCT) is the most popular algorithm is MCST algorithms. Since the goal of UCT is to choose the action of the maximize probability of current statement, a partial search tree is built iteratively using this UCB1 as tree policy [7, 8]. The success of MCST, especially in Go game, is primarily due to this tree policy. A child node j is selected to maximize:

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

where n is the number of visit time of the node's parent, n_j is the number of visit time of this node. Cp is a positive constant chosen by user. If more than one child has the same maximum value, a child node will be chosen randomly among them. The value of $X_{i,t}$ and $\overline{X}_j$ are in range of [0,1]. If the node hasn't been visited before, $n_j = 0$ that would lead UCT to infinity. So an iterated local search will be generated that to assign unvisited children with a largest possible value so that all children of a node are considered before any child is expanded further. [4]

2.3 Details of implementation

In each node v, contains four properties associated with it. Its state s(v), the incoming action a(v), the total simulation reward Q(v) and its visit count N(v). The term $\Delta(v,p)$ represents the reward of player p at node v. So that Q(v)/N(v) is an approximation of the node's game-theoretic value (e.g. typically to assign non-terminal states a reward of 0 and terminal states of values of 0, -1, 1 to tie, loss and win).

Every time a node is part of the play out from the root, its values updated. Once some computational budget runs out, the algorithm terminates and returns the best child reward found corresponding with the highest visit count.

To calculate the best child, there are fore criteria for selecting the wining action, based on the work of Chaslot et al [9]. And in our case, we choose the "Max Child" as our policy, which is to select the most visited root child. The three other policies are: robust child (select the most visited root child); max-robust child (select the root child with both highest visit count and the highest reward) [10]; secure child (select the child that maximizes a lower confidence bound).

Pseudocode of the UCT algorithm:

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Fig 3, The UTC algorithm in pseudocode [4]

The return of the total search in this case is a(BSTCHILD(V_0,0)) which gives child node with the highest reward of all children. The reason that set C_p to 0 in this final call is that we could return the child with the largest visit count directly. Because in the final return, these two options usually describe the same thing.

3. How to use Tic Tac Toe GUI

Tic Tac Toe is a very simple combinatorial game. The game position of Tic Tac Toe is 10^4 and it is proved that a computer intelligent can work perfectly on this game. The board size of the game is 3*3 (our game implementation also has the peritoneal of playing on a n*n board where n is any integer). The rule of this game is also very simple, the first player who get his or her 3 chess in the same column or row or diagonal wins (the first game board on the second row below).

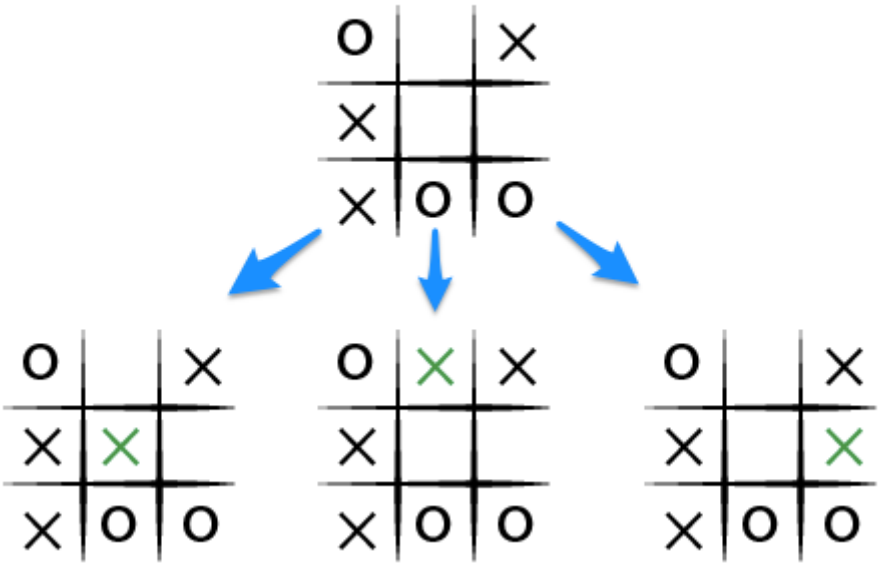

Fig 5. An example of tic tac toe game.

4. Project Implementation

Our MCST implementation is in mcts.py using Python 2.7. It performs the tree search, and it has a tree policy, a default policy and a backup strategy according to "A survey on Monte Carlo tree search", written by Browne et al. (2012) [4].

In the MCST implementation, the properties of every node in the tree are: board states, the parent (None if it is the root), a list of child nodes, its visit count, its simulation reward, its UCT value, its incoming move and finally a boolean value to show this move is for itself or opponent. Class node provides method 'get child' to return the list of its child nodes, method 'update number' for back propagation process to update properties, method ‘update value’ to calculate UCT value.

There is an independent method ‘random one set’ that for one simulation process.

The class SearchTree is to implement pseudo code UCT algorithm. The function search corresponds to UCTSEARCH. The function selection corresponds to BESTCHILD. The function expansion corresponds to EXPAND.

The Tic Tac Toe game body is also written using Python 2.7 using a Python GUI package Tkinter.

We also implement two other AIs to train MCST automatically and also show multiple difficulty levels. One AI is a blind AI, this AI understand nothing about Tic Tac Toe game and will play randomly. Another AI is a bit “cleverer”, which always tries to make a win move first, then check win move of the other player and stop it, and then make move according to some strategy [11]. The Strategy and “way of thinking” of this AI shown below:

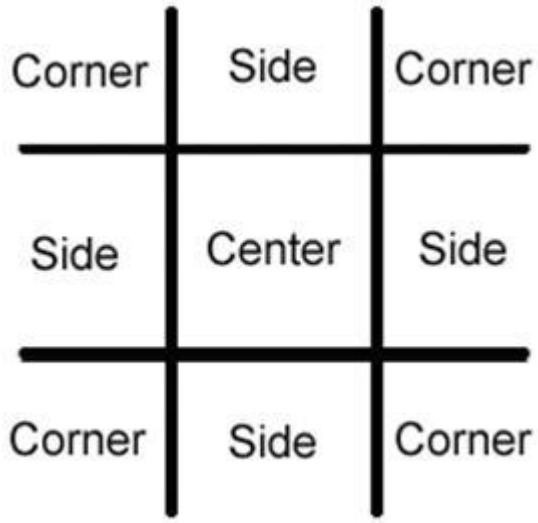

Fig 6. Locations of the side, corner, and center places.

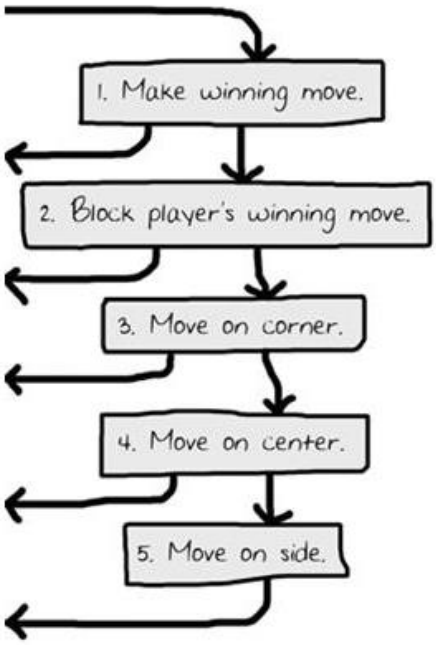

Fig 7. The five steps of the “Get computer's move” algorithm. The arrows leaving go to the “Check if computer won” box.

4.1 How to use our project

According to our settings, MCST AI will run first, its action shown in “O”, and all the children possibilities(rewards) of the previous action node is shown directly on the game board. Of cause the highest possibility is already chosen by the AI.

A message shown on the top of this window tells the action of next move in this game. Player 1 is the player who plays first at the beginning of this game (in our test case, it always be the AI, but this can be easily set in the code) and Player 2 is the player who plays after (which is the human in our case).

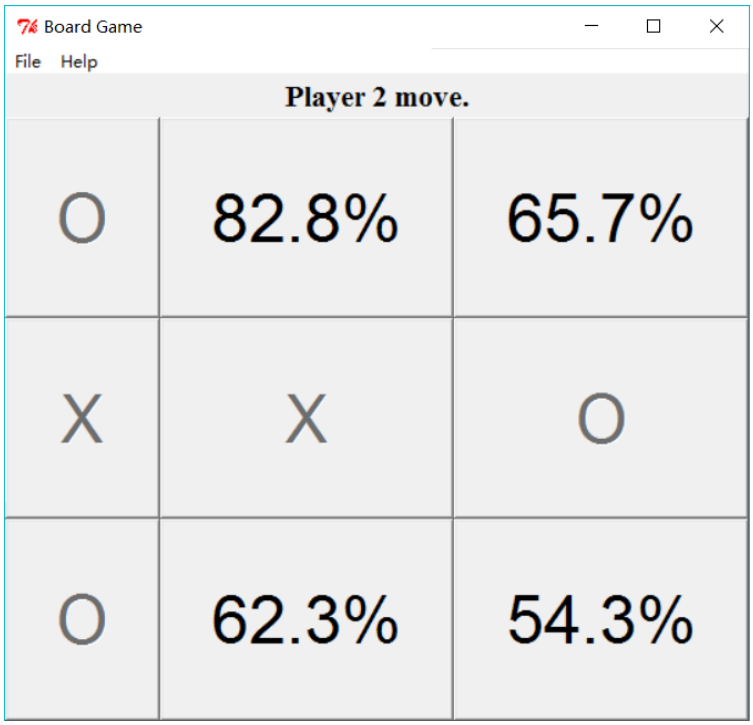

Fig 8. The game board version that waiting for player 2 to move

After a game is over, the result will be shown on the top of the window.

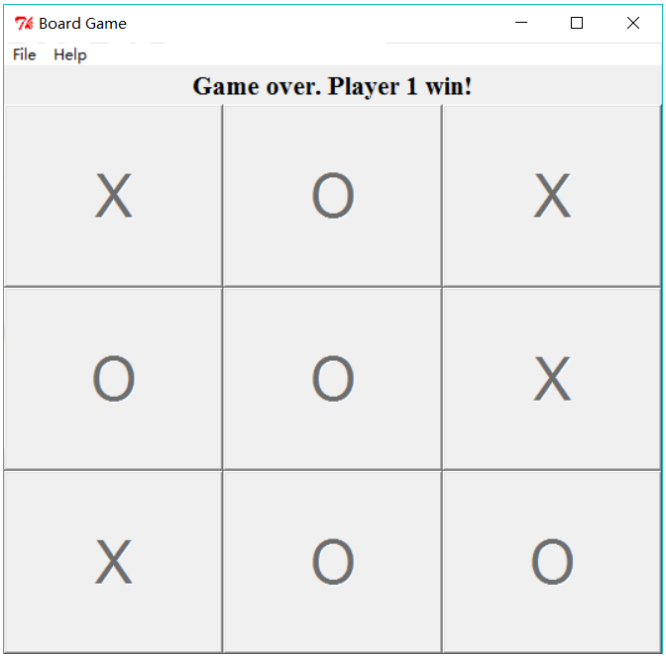

Fig 9. The result of one game (One player wins or tie)

When starting a new game, just click on “File” – “New”, showing as below:

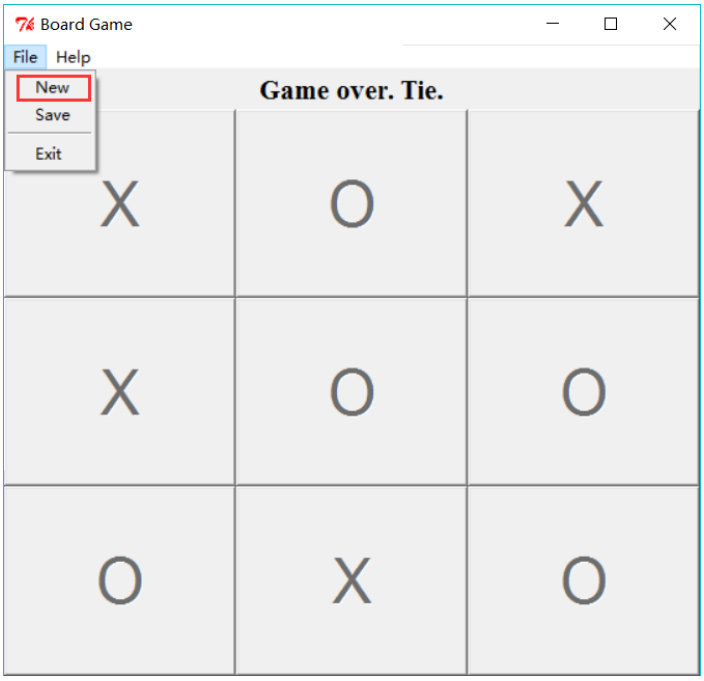

Fig 10. How to start a new game

5. Proposed solution

5.1 Solutions during doing this project

During working on this project, we meet lots of difficulties, but finally we try our best and walk through them.

The first problem we met is that how to test our Monte Carlo Search Tree. I(Cong) had written a Tic Tac Toe game in Java before, so we decide to use this it. However, after researches on the internet, we cannot find an efficient way to share data in memory between Java and Python. At last, we decide to implement a Python Tic Tac Toe game by ourselves.

Another biggest problem we met during the project is to understood the Monte Carlo Search Tree algorithm correctly and built the data structure of it.

During the implementation, there are several issues we met:

1.) Copy issue.

Both of us are familiar with Java, but none of us are familiar with Python. And we were facing some weird problems when copying objects in Python. Finally, we have figured out the right way to deep copy a numpy array: B[:,:] = A, and nothing will change in array A while modifying elements in B.

2.) Using iterative instead of recursive causing infinite loops.

The solution is adding a counter for the iterative loop and breaking the loop if the counter went bigger than expected, and then find the incorrect status and fix it.

3.) Interface problem.

We only made an oral agreement on interfaces when we were splitting coding work, which turns into a huge problem when we tried to merge code. For example, we agreed on 9 elements in an array to show the status of the board. And one chose a 1*9 array and the other chose a 3*3 array. It took us almost one day to work on inconsistent interfaces. What did we learn from that? Write it down! Write it down! Write it down.

4.) We made a mistake of not checking parameters at the beginning of functions.

It occurred several times that error message 'NoneType' object has no attribute xxx. Even though as Python programmers we waste no time declaring the types of arguments or variables, we wasted huge time on fixing 'NoneType' object bugs.

5.) Simulation process.

The research paper said that the simplest case of simulation default policy is to uniformly random. But we think we can do better than that. After discussion, because this is a simple game which can reach the result in at most 5 steps, we decided to set several random moves as the simulation process. And it worked acceptable so far.

6) update UCT value:

At first, we want to update UCT values during processing back propagation according to the research paper [6]. That means to update the UCT value of the current node and recursively update UCT value of its parent node until reaching the root. In the meanwhile, updating one node's UCT value means to change all of its child nodes' UCT value. This whole process created a lot of chaos in the code. After debugging several hours, we decided to update UCT values when choosing best child, which may cause duplicated calculation but makes much easier to implement.

7) Expand process.

On pseudo code, it says expanding one untried child node one time. And we made a comprise on a perfect solution with easy coding. Expand all of child nodes at a time. It costs time and memory, for sure, but it worked given the condition that we are so lack of time.

5.2 Improvements in the future.

Because of lacking time and limits of our skills, we left some features to be improved in the future.

Now, we cannot store our MSCT structure after the program shutting down. That means each time we run the whole program, the MSCT is a new one. Also in our implementation, if several child nodes have the equal UCT value, it should randomly choose one from them. But now the implementation only chooses the first one. So this two parts are just technical issues and can be improved in later version.

The second improvement is that we think our UCT functions can be improved

to make it more efficiency. The value of Cp is a constant and set default value to ($1/\sqrt{2}$) according to Kocsis and Szepesvari[8]which can be improved later.

Third, because the game is so simple, it is hard to evaluate the performance of this method. We can implement our algorithm on 4*4 or higher to evaluate and improve the time performance of our project.

Last but not least, in the future, we can also do some optimization to let human go first. Or let AI join the game during the play.

6. References

[1]. Motwani R, Raghavan P. Randomized algorithms[M]. Chapman & Hall/CRC, 2010.

[2]. Madani K, Lund J R. A Monte-Carlo game theoretic approach for multi-criteria decision making under uncertainty[J]. Advances in water resources, 2011, 34(5): 607-616.

[3]. H. Baier and P. D. Drake, “The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go,” IEEE Trans. Comp. Intell. AI Games, vol. 2, no. 4, pp. 303–309, 2010.

[4]. Browne C B, Powley E, Whitehouse D, et al. A survey of monte carlo tree search methods[J]. Computational Intelligence and AI in Games, IEEE Transactions on, 2012, 4(1): 1-43.

[5]. Kocsis L, Szepesvári C. Bandit based monte-carlo planning[M]//Machine Learning: ECML 2006. Springer Berlin Heidelberg, 2006: 282-293.

[6]. Auer, P.; Cesa-Bianchi, N.; Fischer, P. (2002). "Finite-time Analysis of the Multiarmed Bandit Problem".Machine Learning 47 (2/3): 235–256.doi:10.1023/A:1013689704352.

[7].L. Kocsis and C. Szepesvari, “Bandit based Monte-Carlo Planning,” in Euro. Conf. Mach. Learn. Berlin, Germany: Springer, 2006, pp. 282–293.

[8]. L. Kocsis, C. Szepesvari, and J. Willemson, “Improved Monte-Carlo Search,” Univ. Tartu, Estonia, Tech. Rep. 1, 2006

[9].G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, “Progressive Strategies for Monte-Carlo Tree Search,” New Math. Nat. Comput., vol. 4, no. 3, pp. 343–357, 2008.

[10]. Coulom R. Efficient selectivity and backup operators in Monte-Carlo tree search[M] Computers and games. Springer Berlin Heidelberg, 2006: 72-83.

[11].Sweigart A. Invent your own computer games with python[M]. CreateSpace, 2009. 10