# Technique Analysis and Designing of Program with UCT Algorithm for NoGo

Rui Li, Yueqiu Wu, Andi Zhang, Chen Ma, Bo Chen, Shuliang Wang

School of Software, Beijing Institute of Technology, Beijing 100081
E-mail: 1120112131@bit.edu.cn

**Abstract:** As a typical example of dynamic search algorithm, the UCT algorithm was initially used on the computerized game of GO. This paper briefly introduces the Markov Decision process, the Multi-armed Bandit model, and the Upper-Confidence Bandit formula. It analyzes the source and structure of the UCT algorithm in theory, and proves that the UCT algorithm is suitable for the design of the program of NoGo. According to the characteristics of NoGo, in the paper we improved the algorithm in terms of move generation and data reuse. We also tried to establish an off-line knowledge database for research. With experimental data we have tested and evaluated the above methods. The above algorithm and technology have been successfully used in WTShadows – the NoGo game program, which enabled us to have won the champion in national competition.

**Key Words:** NoGo, UCT Algorithm, Knowledge Base, Dynamic Move Queue, Markov Decision Process, MAB Model

## 1 INTRODUCTION

The game NoGo was invented by members of the Banff International Research Station in 2011 [1]. Rules of the game established in Chinese Computer Game Competition [2] are defined as follow. Two players alternatively place a stone on the board designed for the Go game without move them. The player who either kills opponent's stones or suicides loses the game. It has been testified that NoGo is solvable in polynomial time by an alternating Turing machine [1]. It is a PSPACE problem, which means that the game is relatively highly sophisticated. Therefore, it is essential to apply modern algorithms besides traditional congeners to solve the game.

The Alpha-Beta algorithm is one of the effective methods for compressing the searching space [3]. The algorithm is able to cut out the game tree in order to make the exploration more sufficient. However, as a member of static search algorithms, the evaluation module and the searching module of it are independent. The accuracy of the evaluation module is totally determined by the valuation function based on the expert knowledge and human thinking, which defined the results of the algorithm. But the confidence of the solution cannot be guaranteed when the algorithm is applied to elaborate game problems without adequate expert knowledge. Hence, in this paper, traditional static search and evaluation algorithms are abandoned. To resolve the NoGo game problem, dynamic search algorithms combining search and evaluation are used in this paper, drawing support from random process theories, including the Markov Decision Process, the Monte-Carlo Method [4] and the Multi-Bandit Model.

## 2 DYNIMIC SEARCH ALGORITHM

### 2.1 Markov Decision Process

The Markov Chain was defined as follow [5]. For a random process $\{X_n, n=0,1,2,...\}$ who has countable or finite possible values, if to any state $i_0, i_1, ..., i_{n-1}, i, j$ and any number $n \geq 0$, there exists

$$P\{X_{n+1} = j \mid X_n = i, X_{n-1} = i_{n-1}, ..., X_1 = i_1, X_0 = i_0\} = P_{ij}.$$

Then the process $\{X_n, n=0,1,2,...\}$ is a Markov Chain.

Suppose transition probability $P_{ij}^n$ is the probability when a process moves from *i* to *j*, then

$$P_{ij}^n = P\{X_{n+k} = j \mid X_k = i\},\ n \geq 0, i, j \geq 0.$$

Therefore, the Markov Decision Process Model of NoGo can be established according to theories of the Markov Chain and the Markov Decision Process Model [6].

The process of NoGo game is represented by a quintuple

$$\{X, U, \{U(x) \mid x; x \in X\}, \{p(i,j,u) \mid i, j \in X, u \in U\},$$
$$\{c(x,u) \mid x \in X, u \in U\}\}.$$

And each tuple is defined as follows.

The State Space *X*: Here the state space of the chessboard is used as the state space in the quintuple. A bijective relationship exists between the group of the process states and the group of the chessboard states.

The Control Space *U*: The control space in the quintuple is equivalent to the group of different position on the chessboard.

The Feasible Control Set *U(x)*: Players change the situation of the chessboard by placing stones. This paper regard a move that the player selected when facing the situation *x* as a control. Thus all the legal moves under the situation *x* comprise the Feasible Control Set *U(x)*. If there are no legal moves then the winner has been determined and the game ends.

978-1-4673-5534-6/13/$31.00 ©2013 IEEE

The Transition Probability $p(i,j,u)$ : The Transition Probability $p(i,j,u)$ indicates the probability when the state *i* transform to state *j* under the effect of the control *u*.

The Immediate Loss *c*(*x*,*u*): The NoGo game terminates if and only if one of the players does not have any rational move to select. Define *c*(*x*,*u*) as the decrement of the amount of rational moves after applying control *u* under situation *x*. Then *c*(*x*,*u*) is equal to the increment of the sum of loss moves and illegal moves.

### 2.2 Multi-Armed Bandit Problem

The Multi-Armed Bandit Problem is a type of resources continuous allocation problems [7]. These problems maximize profits of decisions by choosing a policy between using existing strategies and exploring new strategies.

Suppose the slot machine has *n* arms and all of them are independent to each other. If *t* is the total control number and $T_i(t)$ is the control number of the arm *i,* then the equation $\sum_{i=1}^{n} T_i(t) = t$ exists. If control methods of the slot machine at the time of *t* are expressed as a function $U(t) = (U_1(t), U_2(t), ..., U_n(t))$ , then we have

$$U_i(t) = \begin{cases} 0, & \text{the arm i does not be operated at time t.} \\ 1, & \text{the arm i is operated at time t.} \end{cases}$$

$$\text{And } T_i(t+1) = \begin{cases} T_i(t)+1, & U_i(t) > 0 \\ T_i(t), & U_i(t) = 0 \end{cases}.$$

Define function $X_i(T(t))$ as the situation of arm *i* at time *t* and $R_i(t)$ as profits, then

$$X_i(T(t)) = \begin{cases} X_i(T_i(t)+1), & U_i(t) = 1 \\ X_i(T_i(t)), & U_i(t) = 0 \end{cases};$$

$$R_i(t) = \begin{cases} R(X_i(t)), & U_i(t) = 1 \\ 0, & U_i(t) = 0 \end{cases}.$$

The aim of the multi-armed bandit problem is to maximize the expectation $E[\sum_{t=0}^{t\max} \beta^t \sum_{i=1}^{n} R_i(X_i(T_i(t)))]$ by finding a proper function *U* (*t*) [7]. Variable *tmax* is the time of the last operation while $\beta$ is the discount factor.

The multi-armed bandit model is a Markov Decision Model while the immediate loss can be as follow [8]. Suppose C(t) is the total loss after *t* operations,

$$C(t) = (\max_{1 \le i \le k} \mu_i)t - \mu_j \sum_{j=1}^{k} E[T_j(t)] .$$

In the formula, $\mu_i$ is the profit under control *i*. Therefore the immediate loss after each control is C (t)-C (t-1).

### 2.3 Upper Confidence Bandit Function

The UCB function is one of the functions that can solve the multi-armed bandit problem [9], basic roles of which are to maximize total profits by making balance between achieving profits and exploring information.

For any state *x*, if only consider impacts proceeded from profit factors, the average profit Pf(x) can be obtained by calculating according to previous operations. If $\mathrm{Pf}_M$ , selected from Pf(x), satisfy the condition

$$(\forall \mathrm{Pf}_i \in \mathrm{Pf(x)}) \rightarrow (\mathrm{Pf}_M \ge \mathrm{Pf}_i) .$$

Then the control *u* corresponding to $\mathrm{Pf}_M$ is the control maximizing profits basing on known information. Yet, considering the lack of information, it is difficult to assert that whether the control v exists which lack relevant information and satisfy Pf (v)>Pf (u). Hence, precise selection cannot be ensured if only depends on profits factor.

Define the upper confidence function Ep(v(u)) to utilize controls that hardly being explored and to expand their data libraries, while v(u) represents the operating frequency of control *u*. Pf(u) converges to the profit expectation of *u* when v(u) converges to positive infinity. Meanwhile, with the increment of v(u), profit information of *u* tends to be more accurate. Therefore, Ep(v(u)) monotonically decreases when *u* increases and the control maximizing Ep(v(u)) is chosen by the UCB function.

An effective equation for exploration is displayed as follow considering offsets possibly existed [9].

$$Ep(u_i) = 2\mathrm{C}_p \sqrt{\frac{\ln(\sum_{i=1}^{k} v(\mathrm{u}_i))}{v(u_i)}}$$

In the equation, constant $\mathrm{C}_p$ varies according to specific forms of problems.

### 2.4 Upper Confidence Tree Algorithm

The UCT algorithm is one of the applications of the UCB function to tree structures. This paper analyzes the structure and functions of the UCT algorithm with the help of the game tree model.

The UCB function and the MAB model are appropriate for solving the NoGo game because it has simple rules, complex variability and inadequate expert knowledge. In this solution, situations of the chessboard, move selections and profits produced by moves are regarded as situations of the slot bandit, controls upon different arms and profits produced by controls. The decrement of special conditions, as a result of the lack of expert knowledge, expands the extent of applications of the UCB function.

Traditional static search algorithms traverse most of nodes in the same level of the tree, which guarantees the accuracy of the search but reduce the depth of it. The UCT algorithm executes the evaluation model and the search model synchronously by exploring the best subtree instead of the best path. The algorithm reduces the number of accesses to subtrees whose average profits maintains at a low level, but develops subtrees with significant average profits. Because confidences of average profits are positive correlated with scales of subtrees, the UCT algorithm selects the subtree to be expanded by employing the UCB function, Thus keep the balance between average profits and scales.

For a tree that has complex structure, the profit of the root node and profits of its leaf nodes are strongly coupled. The

UCT algorithm first seeks leaf nodes by repeatedly applying the UCB function, then expands the node and executes the simulation module. Results of simulation module are returned to the root finally. This process not only reduces the length of the random simulation sequence, but also enhances the convergence of results of the Monte-Carlo simulation. Therefore, with the frequency of random simulations increasing, searching out the most satisfactory subtree becomes easier. The basic structure of the UCT algorithm is exhibited in Figure 1.

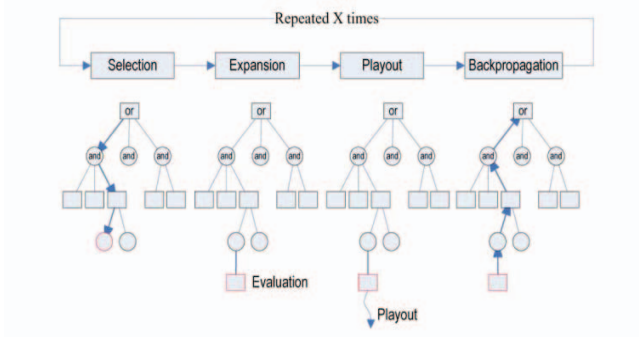


Fig 1. The structure of the UCT algorithm [10]

The UCT algorithm contains several cyclic UCB-falling processes which consist of four steps as follows.

(1) Selection. According to upper confidence values of nodes, the UCT algorithm continuously selects the node with maximal upper confidence in the existed tree nodes until the chosen node is a leaf node.

(2) Expansion. The UCT algorithm produces all child nodes of the chosen node and appends them to the tree.

(3) Simulation. The Simulation of the UCT algorithm is a Monte-Carlo Simulation starts from the chosen leaf node. After the simulation terminates, results are returned to the chosen node.

(4) Feedback. Nodes obtained the return value yield a feedback to their parent node. A UCB-falling process terminates until the root obtains the feedback.

The UCT algorithm chooses the child of the root, who has the highest winning percentage, as the best move and returns it. The pseudocode of the UCT algorithm is displayed in Table 1.

```
Function Search_A_Good_Move()
{
  Get the System Time;
  Update the Chessboard;
  if(GameTree is empty)
    Create the GameTree;
  if(The game is finished)
    return;
  while(LocalTime < TimeLimit)
  {
    do
    {
      TargetNode = UCB_descend();
      Update the UCT-Chessboard;
    }
    while(TargetNode is not a leaf);
    Expand(TargetNode);
    TargetNode -> win = Simulate(TargetNode);
    UpdateNode(TargetNode);
  }
}

Function UCB_descend()
{
  Calculate the total visit-value of all nodes;
  while(PresentNode is not a leaf)
  {
    Calculate the UCB-value by applying the
      UCB-formula as mentioned in section 1.3;
    if(PresentNOde -> UCB-value > MaxValue)
      Update the MaxValue and the ResultNode;
  }
  return ResultNode;
}

Function Expand(TargetNode)
{
  create all the childnodes of the TargetNode;
}

Function Simulate(TargetNode)
{
  Update the chessboard;
  while(limit of the simulation does not reached)
    result += DQ_Simulation(TargetNode -> color);
//Function DQ_Simulation()
// will be explained carefully in Section 2.2;
  return result;
}
```

Tab 1. Pseudocode of the UCT algorithm

## 3 MOVE GENERATION

### 3.1 Legal Move and Rational Move

The UCT algorithm chooses suicide moves only if there are no rational moves to select or there are few explorations to suicide moves. Because the UCT algorithm keeps balance between profits and explorations, several simulations have to be retained by suicide moves. This process brings about the waste of simulations for known results of suicide moves. Therefore, suicide moves can be rejected before being expanded to make the algorithm more efficient.

Thus, the algorithm has to differentiate between rational moves and suicide moves before expanding nodes. According to rules of the NoGo, simple search methods such as Depth-First-Search and Breadth-first-search can complete the task. Experiments indicate that the BFS algorithm is more sufficient than the DFS algorithm so this paper adopts the BFS algorithm to distinguish moves.

In experiments, the algorithm appending discrimination module obtained total victory. Hence, the discrimination module is necessary for the algorithm.

### 3.2 Dynamic Move Queue

The program with advanced UCT algorithm is analyzed by the performance tester built in the Visual Studio 2012. Results are displayed in Figure 2 and Figure 3.

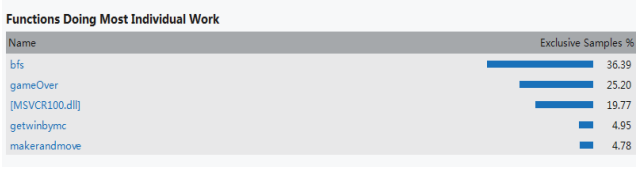


Fig 2. Table of functions doing most individual work

Figure 2 indicates that the BFS function consumes 36.39% of the total runtime. In this experiment the program takes 10s as a cycle and executes the UCB-falling process for fifty thousand times every cycle. Figure 3 collects working times of all functions and sorting them with descending order.

| Function Name | Inclusive Samples | Exclusive Samples | Inclusive Samples % | Exclusive Samples % |
|---|---|---|---|---|
| bfs | 2,244 | 2,244 | 36.39 | 36.39 |
| gameOver | 4,858 | 1,554 | 78.79 | 25.20 |
| [MSVCR100.dll] | 1,356 | 1,219 | 21.99 | 19.77 |
| getwinbymc | 5,494 | 305 | 89.10 | 4.95 |
| makerandmove | 2,715 | 295 | 44.03 | 4.78 |
| createmovetree | 399 | 185 | 6.47 | 3.00 |
| [ntdll.dll] | 124 | 124 | 2.01 | 2.01 |
| playonesequence | 6,160 | 118 | 99.90 | 1.91 |
| _memset | 40 | 40 | 0.65 | 0.65 |
| gameover | 88 | 32 | 1.43 | 0.52 |
| Bfs | 28 | 28 | 0.45 | 0.45 |
| [kernel32.dll] | 10 | 10 | 0.16 | 0.16 |
| [KERNELBASE.dll] | 7 | 7 | 0.11 | 0.11 |
| __security_check_cookie | 2 | 2 | 0.03 | 0.03 |
| SearchAGoodMove | 6,162 | 2 | 99.94 | 0.03 |
| __CIlog | 1 | 1 | 0.02 | 0.02 |
| __tmainCRTStartup | 6,162 | 0 | 99.94 | 0.00 |
| main | 6,162 | 0 | 99.94 | 0.00 |
| mainCRTStartup | 4 | 0 | 0.06 | 0.00 |
| Unknown | 4 | 0 | 0.06 | 0.00 |

Fig 3. Working time table of functions produced by Visual Studio

According to results of experiments, function *creatmovetree* and its subfunction for expanding the game tree consume 6.47% of the total runtime, while function *getwinbymc* for simulations consumes 89.19% of the total runtime. The proportion of runtime between those two functions is 1:13. This fact indicates that the runtime of expanding trees is occupied by simulation, which decreases the depth of the game tree; therefore constrict the intelligence of the program.

The efficiency of the BFS algorithm cannot be optimized because the size of the chessboard is limited. But the number of calls of the BFS function is enormous because the function must be called to differentiate rational moves. So the calls should be reduced to decrease the individual runtime of the BFS function.

In the UCT algorithm, the expanding module just expands one node in the game tree in a single execution. That is to say, new nodes produced in the same cycle can be seen as nodes transformed once from their parents. Because the stones cannot move once placed, suicide moves of the parents node are also suicide moves to child nodes. Similarly, rational move set to child nodes is a proper subset of rational move set to parents. Therefore, number of rational moves decreases as the node depth increase. Thus we can construct an appropriate data structure to store rational moves on the blank chessboard. With the game progressing, moves that transform to suicidal from rational are rejected from the data structure so that the calls of the BFS function are reduced. To accomplish this task, the dynamic move queue is devised and its pseudocode is exhibited in Table 2.

```
1. Function DQ_Simulation(color)
2. for each blank on chessboard do
3. makemove(black)
4. if (!gameover)
5.    ourstack[] <-- blackmove
6. makemove(white)
7. if (!gameover)
8.    enemystack[] <-- whitemove
9. unmakemove
10. end for
11. make_random_sequ(ourstack)
12. make_random_sequ(enemystack)
13. //if enemy move first
14. while !gameover
15.   make_enemy_move(enemystack)
16.   if (gameover)
17.     return win
18.   make_our_move(ourstack)
19.   if (gameover)
20.     return loss
21. end while
```

Tab 2. Pseudocode of the dynamic move queue

Here we also use the Visual Studio performance tester to analyze the improvement. Figure 4 gives functions doing most individual work after improving while Figure 5 provides working time of all functions.

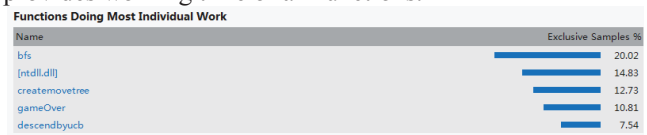


Fig 4. Functions doing most individual work after improving

| Function Name | Inclusive Samples | Exclusive Samples | Inclusive Samples % | Exclusive Sampl... |
|---|---|---|---|---|
| bfs | 563 | 563 | 20.02 | 20.02 |
| [ntdll.dll] | 417 | 417 | 14.83 | 14.83 |
| createmovetree | 838 | 358 | 29.80 | 12.73 |
| gameOver | 1,024 | 304 | 36.42 | 10.81 |
| descendbyucb | 426 | 212 | 15.15 | 7.54 |
| getwinbymc | 1,293 | 147 | 45.98 | 5.23 |
| _VEC_memset | 132 | 132 | 4.69 | 4.69 |
| _libm_sse2_sqrt_precise | 115 | 115 | 4.09 | 4.09 |
| memset | 101 | 101 | 3.59 | 3.59 |
| _libm_sse2_log_precise | 91 | 91 | 3.24 | 3.24 |
| gameover | 215 | 82 | 7.65 | 2.92 |
| freemem | 243 | 72 | 8.64 | 2.56 |
| Bfs | 47 | 47 | 1.67 | 1.67 |
| [kernel32.dll] | 188 | 40 | 6.69 | 1.42 |
| __crtFlsGetValue | 54 | 25 | 1.92 | 0.89 |
| [KERNELBASE.dll] | 19 | 19 | 0.68 | 0.68 |
| _getptd_noexit | 98 | 17 | 3.49 | 0.60 |
| _getptd | 112 | 14 | 3.98 | 0.50 |
| _memset | 11 | 11 | 0.39 | 0.39 |
| malloc | 265 | 11 | 9.42 | 0.39 |
| rand | 122 | 10 | 4.34 | 0.36 |
| free | 171 | 9 | 6.08 | 0.32 |

Fig 5. Working time table of functions after improving

According to Figure 4, the consuming time of the BFS function decreases to 20.03% from 36.39%, meanwhile the individual runtime of expanding increases to 12.73% from 3.00%. In addition, according to Figure 5, the runtime of simulations and that of expanding respectively consume 45.98% and 29.80% of the total time. In that case, the proportion between those two basic modules is 2:1 instead of 13:1 revealed before. Besides, after applying the dynamic move queue, the program executes nearly 200 thousand times UCB-fallings and is quadruple than before. Thus, the dynamic move queue evidently strengthens the intelligence of the program.

## 4 REUSE OF THE UCT TREE

### 4.1 Immediately Reuse

The dynamic move queue successfully decreases time consumed by simulation and saves time for program to expand the game tree. The beginning of the search module decreases by 1 layer in the game tree after each move output.

Therefore, when the same player obtains the control of the game, the start point of the search has decreased by 2 layers. Saving previous results and using them in following search is an effective method to increase the depth of the game tree. However, this method, called the reuse of the UCT tree, can only utilize tree structures produced in the same game. All nodes will be cleared after the game ends.

| Best winrate | Total visits | Best visits | Reused visits |
|---|---|---|---|
| 0.50644 | 252895 | 56441 | 5036 |
| 0.510661 | 261673 | 45026 | 1212 |
| 0.525865 | 260438 | 72260 | 1851 |
| 0.526707 | 266276 | 13873 | 130 |
| 0.509076 | 265717 | 48533 | 2888 |
| 0.513263 | 280688 | 67441 | 3180 |
| 0.52206 | 286405 | 52561 | 4499 |
| 0.524112 | 296094 | 40913 | 3677 |
| 0.541881 | 304239 | 244895 | 24824 |
| 0.554047 | 338279 | 224149 | 47464 |
| 0.56465 | 372652 | 188012 | 40622 |
| 0.563274 | 376464 | 150488 | 3114 |
| 0.578079 | 337372 | 216619 | 77731 |
| 0.57934 | 430361 | 28441 | 557 |
| 0.604428 | 354372 | 319823 | 37478 |
| 0.620151 | 402300 | 361891 | 140354 |
| 0.629128 | 529515 | 420131 | 25926 |
| 0.647071 | 434343 | 370820 | 34353 |
| 0.652967 | 453306 | 90657 | 14816 |
| 0.650581 | 446961 | 94700 | 5566 |
| 0.655694 | 456988 | 161133 | 55862 |
| 0.657049 | 520955 | 251555 | 33655 |
| 0.704195 | 496846 | 477801 | 61695 |
| 0.736132 | 546411 | 534289 | 72214 |
| 0.763591 | 573501 | 554777 | 137632 |
| 0.825047 | 692081 | 648164 | 79971 |
| 0.927213 | 681441 | 673227 | 152751 |
| 0.969246 | 882792 | 880037 | 72333 |
| 0.991946 | 978377 | 976156 | 142052 |
| 0.998528 | 1185689 | 1095380 | 150999 |
| 1 | 1438320 | 226520 | 37752 |

Tab 3. Detailed information of player with black stones in one of experiments

Name those visits to nodes produced by previous search as the Reused Visits. According to Table 3, the number of reused visits shows an ascending trend in principle and reaches maximum value when the game enters mid and late period. This fact indicates that comparing to the program without node reuse; the improved program increases the depth of search by 5% to 30%. In antagonism experiments, the winning rate of improved programs is greater than 85%.

### 4.2 Knowledge Base

Although quantities of optimization methods are applied, the simulation speed of the UCT algorithm only increases linearly. Meanwhile, the expanding of the upper confidence tree occupies memory resources successively, which limits the depth of the tree. Therefore, the knowledge base of the NoGo game is designed to solve the problem.

The UCT algorithm builds an unbalanced game tree according to the upper confidence value and abandons useless branches after the move was selected. However, it is possible to meet the same chessboard situation in other matches thus those abandoned results can be used. If all the results produced from games are saved in the disk, the game tree in the disk would be enormous and distinctly strengthen the program's intelligence. In addition, this method also accumulates the expert knowledge of the NoGo game and promotes theoretical solving of the game. Storing the game tree as a whole is not practical. In experiments, 100 gigabytes disk space was occupied completely after 4 to 5 rounds. To solve the problem we devise the method as follow. Define the move confidence MCD and compute it using following equation:

$$MCD = \alpha \bullet v_i / V + winrate \cdot$$

In the equation, $\alpha$ is a constant; $v_i$ is the visit number of present node and $winrate$ is the winning rate of present node. The move is written into the knowledge only if its MCD value exceeds the given standard.

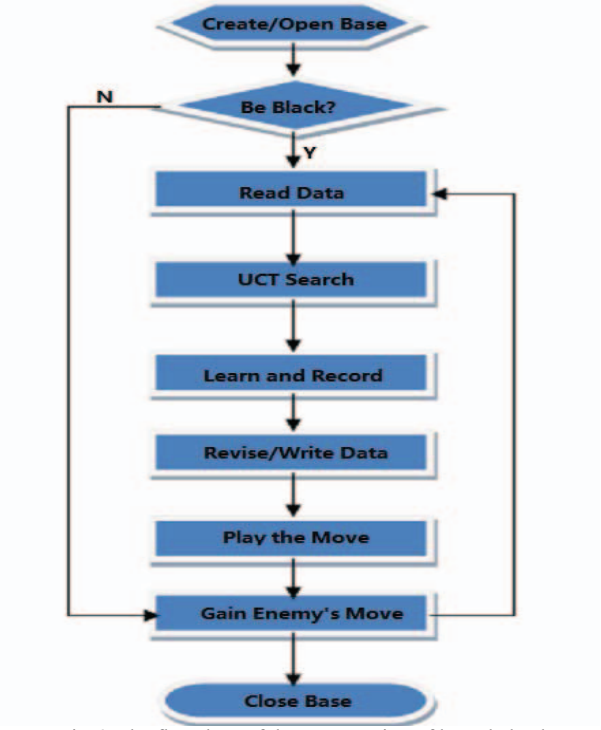


Fig 6. The flowchart of the construction of knowledge base

Finally this paper gives the flow path of building the knowledge base, as shown in Figure 6.

(1) Open the base. Memory-mapped files are used for fast access to data.

(2) Read information. Link lists are used in this part to save time. When the node satisfies $MCD \geq MCD$ the program skips to move exportation module.

(3) Learning process. In this process the program revises existed information or writes new nodes. This process is classified into guiding study and combat training. The guiding study makes programs collect special information from designed situations, while the combat training makes programs compete with each other.

(4) Generate log files. The program records all the operations applied on knowledge base to observe possible changes and to prevent program exceptions.

(5) Search and write. Memory-mapped files are used for storing results generated by search module to save

memories and guarantee speed. Nodes satisfying $MCD > MCD$ are written into the base as file link lists.

## REFERENCES


[1] Chou C, Teytaud O, Yen S. Revisiting Monte-Carlo tree search on a normal form game: NoGo[J]. Applications of Evolutionary Computation, 2011: 73-82.

[2] ICGA Tournaments. Basic Rules of the NoGo Game. From: http://www.grappa.univ-lille3.fr/icga/game.php?id=47

[3] Xiaochun Wang. PC Game Programming (Machine Adversarial)[M]. Chongqing University Press, 2002.5, 102-114.

[4] Jianzhong Zhang. Monte-Carlo Method (I)[J]. Practice and Research of Mathematics, 1974, 1: 002.

[5] S. M. Ross, Guanglu Gong. Introduction to Probability Models ($9^{th}$ Editions)[M]. The People's Posts and Telecommunications Press, 2007.12, 141-197.

[6] Jianghong Li, Zhengzhi Han. New Achievements in Adaptive Markov Decision Process[J]. Control and Decision, 2001, 16(1): 7-11.

[7] Mahajan A, Teneketzis D. Multi-armed bandit problems[J]. Foundations and Applications of Sensor Management, 2008: 121-151..

[8] Auer P, Cesa-Bianchi N, Fischer P. Finite-time analysis of the multiarmed bandit problem[J]. Machine learning, 2002, 47(2): 235-256.

[9] Kocsis L, Szepesvári C. Bandit based monte-carlo planning[J]. Machine Learning: ECML 2006, 2006: 282-293.

[10] Yen S J, Yang J K. Two-stage Monte Carlo tree search for Connect6[J]. Computational Intelligence and AI in Games, IEEE Transactions on, 2011, 3(2): 100-118.