A Pattern Based Incremental Model in K-in-a-row Games

XU Chang-ming[1], Z.M. MA[2], Yu Chang-yong[1], XU Xin-he[2]

1. Institute of Computer and Communication Engineering, Northeastern University at Qinhuangdao, Qinhuangdao 066004
E-mail: changmingxu@gmail.com

2. Information Science and Commutation Engineer, Northeastern University, China, 110819
E-mail: zmm@mail.neu.edu.cn

Abstract: This paper designs an elegant model in *k*-in-a-row games, *Connect6* as an example. In this model, domain knowledge is involved in the special pattern called *Connection*. A Connection covers several consecutive intersections of a line on the board, and is also the basic element of the data structure sketching the blueprint of the model. Based on Connection, we can show how to construct a complete database containing the knowledge of each possible Connection, and then explain how to design the data structure of such a model. As a result, the speed of searching the same tree in the model is as about 3 times as that in the regular method. Finally, because of the inherent character of the modal, the incremental method is easy to taken to enhance it further, which is used to reduce the updating cost of each tree node. The experiments show about more than 1.9 times improvements are achieved if incremental applies.

Key Words: incremental model, pattern, Connection.

1 INTRODUCTION

Connect(*m*, *n*, *k*, *p*, *q*) denotes a family games of *k*-in-a-row. There are two players, the black and the white. The first player, always the black side, places *q* stones for the first move. Then, two players alternately place *p* stones on *m*×*n* board in each turn. The player who gets *k* consecutive stones first win. *Connect*(*m*, *n*, 6, 2, 1), also called Connect6, is first introduced in [1], which is more complex than any solved game. The most popular board size of Connect6 is 19×19. Connect6 is taken as the example to introduce our mode.

In general, the board is always considered as a reticular formation of intersections. In some games, e.g. *k*-in-a-row, some adjacent intersections can be considered as one group, called *pattern*s. In the perspective of the intersection, the state of a board can be represented uniquely, but it seems difficult to reach it in the perspective of patterns. The main reason is that the sphere of a pattern is often unclear in many games. In this paper, we proposed a sphere well-defined pattern, called *Connection*, which helps us to split a line of any *k*-in-a-row board into several meaningful parts, and then a board is divided as a set of Connections with such a division being unique. Such Connection based division of the board is good for performance.

Pattern database contains the knowledge of each possible pattern. While the knowledge stored in it is quite different. Some pattern databases may have the perfect knowledge, such as the endgame database in chess [2] and in checkers [3]. Some pattern database may store the dynamic heuristic knowledge, such as that in *n*-puzzle [4]. In addition, some pattern database may have a learning function [5]. In this paper, we construct a complete pattern database contained the précised and constant type for each pattern. Thus, all the possible paths of a normal pattern promotion to one of winning patterns it can achieve can be driven by the simple operation that adds a stone at an empty intersection each time.

Because of the high complexity of the game, an absolute updating may consume too much time on boards. In the field of Computer Go, an implementation has used sets of related intersections to classify the intersections in order to update selectively [6]. Considering the particularity of *k*-in-row game, in this paper, we put forward a new incremental updating approach, we maintenance a series of board changes, and then we reduce the expense with a special replacement method instead of creating new duplicated states at all.

The main contributions of this paper are as follow: first, a model, called PBIM is proposed to divide a board into a set of patterns without any ambiguous. Second, we developed a CBKDB to save the knowledge of all the Connections; third, an incremental method is given to enhance the model further.

2 PATTERN BASED KNOWLEDGE DATABASE

2.1 Decomposing a Line into Connections

First, we give the definition of *Connection*, which is denoted as *c* in *Connect*(*n*, *n*, *k*, *p*, *q*). It is defined as a maximal consecutive sequence at least covering *k* intersections in a straight line, which consists of either stones in the same color or empty intersections. Each Connection *c* has its color, denoted as *color*(*c*). In addition, Given a Connection *c* and a stone *s*, and the color of s as *scolor*(*s*), we always say that, *s* is its own one if *color*(*c*) = *scolor*(*s*); *s* is an enemy one if *color*(*c*) ≠ *scolor*(*s*), here, *color*(*c*) = *not*(*scolor*(*s*)), and *not* is an operation to inverse color. Then, *color*(*c*) is determined by the following rules: 1) if *c* has any stone *s* (see (1) and (2) in Figure 1), *color*(*c*) ← *scolor*(*s*); 2) if *c* has no stone and both boundaries of it are also the boundaries of the game board (see (4) and (5) in Figure 1), we should emphasized that there are two empty

This work is supported by National Nature Science Foundation (Grant No. 61100021 and 201102057) and the Fundamental Research Funds for the Central Universities of China (Grant No. N100323002).

Connection (*i.e.* Connection without any stones) overlapped each other, and the color of one Connection is black, while that of the other is white; 3) if c has no stone and at least one of its boundaries is not the board's boundary, that is, there is an enemy stone s close to c (see (3) in Figure 1), then $color(c) \leftarrow not(scolor(s))$. As is shown in Figure 3, there are five Connections exactly for Connect6 enclosed by the ellipses and marked with a number respectively. According to the rules mentioned above, we can easily draw that: (1) is a black Connection; (2) is a white Connection; (3) is a black Connection; (4), (5) are overlapped each other nicely, and one of them should be consider as black, while the other should be white.

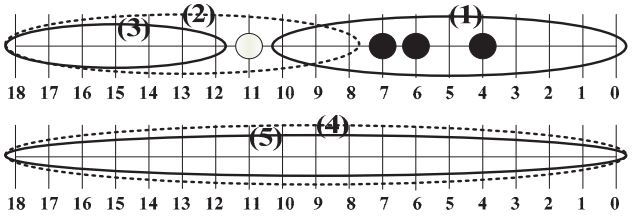


Figure 1. 3 Connections of one line in *Connect*(19, 19, 6, 2, 1)

By using the concept of Connection, any *line* (horizontal, vertical, and diagonal) can be separated into several determinate Connections without any ambiguity as seen in Figure 3. Moreover, the meanings of Connection in this paper are plentiful. Firstly, a Connection implies that it is such a place on the board where a k-in-a-row may occur. Secondly, it clearly outlines all the spheres of influence belonging to the black or the white within a line. Although the spheres of influence may overlap, they keep independent and integrated to each other as whole units. The other attributes of Connection is defined as follows:

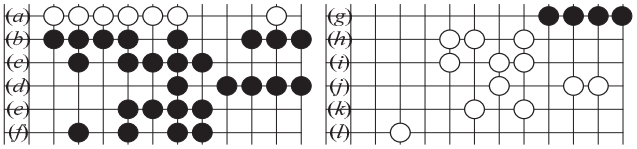
- The *length* of c, denoted as $|c|$, is defined as the number of intersections possessed by c.
- The shape of c, denoted as $||c||$. To describe the states of an intersection, we define: $s(c, i) = 1$, if i^{th} intersection of c is occupied by a stone; and $s(c, i) = 0$, if not. Given Connection c_1 and c_2, $f(c) = \sum_{0 \le i < |c|} \cdot s(c, i) \cdot 2^i$, c_1 and c_2 have the same shape, written as $||c_1||=||c_2||$, iff: 1) $|c_1|=|c_2|$; 2) $f(c_1)=f(c_2)$.
- The type of c, as an evaluation of c, will be introduced in Section 2.2 in detail.

We should remark that the definition of Connection is applicable for all the k-in-a-row games. While Connection Type in section 2.2 and Intersection Type in section 2.4, as the domain knowledge of game may vary with the rules of concrete k-in-a-row game somewhat, we take Connect6 as the example in this paper.

2.2 Connection Type

In most games, the evaluation of a game position is a challenge. Similarly, it is also a difficult problem to evaluate a Pattern. Fortunately, [1] has made a good groundwork on this issue. *Wu* has introduced a sli*ding window* method to figure out *threats* (definition should be given) and *the number of threats in a line*, furthermore, the *threat patterns*, similar to Connection in this paper, is given for k-in-a-row games. However, because the definitions of threat patterns aren't precise like Connection at all, it hardly covers all possible patterns. In this paper, we adopt wu's *sliding window* method, and then apply it to classify all Connection into 12 distinguished types precisely. S_{CT} = {*'WIN'*, *'DW'*, *'L5'*, *'D5'*, *'L4'*, *'S4'*, *'D4'*, *'L3'*, *'S3'*, *'D3'*, *'L2'*, *'O'*} is the universal set of all possible Connection types. It can be seen as an evaluation schema, because of the fact that Connections with the same type have the identical impact to the game, no matter what the shapes they have based on the following description.

In Figure2, A *'WIN'* Connection is a Connection contained 6-in-a-row, also the goal of each player to achieve. A DTC(*Direct Threat Connection*) is such a Connection c that can be formed into a *'WIN'* Connection by adding one/two additional own stone, e.g., (b)~(g) in Figure2. An ITC(*Indirect Threat Connection*) is a Connection c which can be formed into a *DTC* by adding one/two own stone into c, e.g., (h)~(k) in Figure2. *The number of threat for a* DTC c, written as $thn(c)$, is a very important measure that tells us how many stones are needed for enemy to resolve the threats and block us to win. By the way, a *'DW'* (*Definitively Win*) Connection is a Connection with more than 2 threats. The method to work out $thn(c)$ we used is the sliding window method, see the pseudo of line 1, 8, and 9 in Algorithm 1 for detail and [Wu and Huang 2005] for more. A Connection c is an *Live-l Connection*, where $2 \le l \le 5$ and l meaning that $6-l$ additional stones is needed to form a *'WIN'* Connection, if any of the following three cases can be satisfied: 1) c is a DTC with $thn(c)=2$; 2) a *live*-($l+1$) DTC can be formed by adding a own stone into c; 3) a *live*-($l+2$) DTC can be formed by adding two own stones into $\boldsymbol{c}$. A Connection $\boldsymbol{c}$ is a *Sleep-l Connection* if it satisfies any of the following cases: 1) c is a DTC with $thn(c)=1$ and at the same time $\boldsymbol{c}$ can be formed into *live*-($l+1$) DTC by adding a own stone; 2) c is a DTC with one threat can be formed by adding one stone into $\boldsymbol{c}$, and at the same time a *live*-(l+2) DTC by adding two additional stones of $color(c)$ into c. $\boldsymbol{c}$ is a *Dead Connection* if it is a non-sleep DTC with one threat, or a non-sleep ITC which can be formed into a DTC by adding one/two own stones into c.



(*a*) *'WIN'*; (*b*) *'DW'*; (*c*) *'L5'*; (*d*) *'D5'*; (*e*) *'L4'*; (*f*) *'S4'*; (*g*) *'D4'*; (*h*) *'L3'*; (*i*) *'S3'*; (*j*) *'D3'*; (*k*) *'L2'*; (*l*) *'O'*;

Figure 2. Examples for each type of Connection

Two classifiers, DTCTC(*Direct Threat Connection Type Classifier*) and ITCTC(*Indirect Threat Connection Type Classifier*) are given. DTCTC makes the first pass through the set of all connection to identify each DTC with the exact type, as is shown in Algorithm1. After that, to assign each ITC with the corresponding type, ITCTC is executed as the second pass scan as is shown in Algorithm2.

Algorithm 1. Implementation of DTCTC

/*Note: In the process of sliding a 6-length window ***from left to right*** in Connection c, w is used to number the current window; $nStonesIn(w)$ is the number of stones in w; $isMarked(w)$ indicates whether a marked intersection locates in w. */

```
01.  for a Connection c:
02.      nThreatWindows ← 0;
03.      isFive ← false;
04.      isFour ← false;
05.      for w in range(0,|c|-6) :
06.          if 6 = nStonesIn(w):
07.              tc(c) ← 'WIN';
08.              break;
09.          elif 4 ≤ nStonesIn(w) and false = isMarked(w):
10.              nThreatWindows ← nThreatWindows+1;
11.              #Marking the rightmost empty intersection …
12.              if 4 = nStonesIn(w):   isFour ← true;
13.              if 5 = nStonesIn(w):   isFive ← true;
14.      if (3≤nThreatsWindows): tc(c) ← 'DW';
15.      elif 2 = nThreatsWindows and true = isFive:   tc(c) ← 'L5';
16.      elif 2 = nThreatsWindows and true = isFour:   tc(c) ← 'L4';
17.      elif 1 = nThreatsWindows and true = isFive:   tc(c) ← 'D5';
18.      elif 1 = nThreatsWindows and true = isFour:
19.          for i in range(0, |c|):
20.              if 1 = s(c, i):   continue;
21.              if ('L4' = tc(putStone(c, i) or 'L5' = tc(putStone(c, i)) and tc(c) < 'S4':
22.                  tc(c) ← 'S4';
23.                  break;
24.      tc(c) ← 'UNKNOWN';
```

Algorithm 2. Implementation of ITCTC

/* Note: *nMax* traces the maximum number of stones in a window; *putStone*(*c*, *i*) represents to put an own stone at i^{th} intersection; *putStone*(*c*, *i*, *j*) represents to put two own stones at i^{th} and j^{th} intersections, respectively. */

```
01.  for a Connection c :
02.      if 'UNKNOWN' ≠ tc(c):   continue;
03.      nMax ← 0;
04.      for w in range(0,|c|-6):
05.          if nMax < nStonesIn(w):   nMax ← nStonesIn(w);
06.      if 3 = nMax:
07.          for i in range(0, |c|):
08.              if 1 = s(c, i):   continue;
09.              elif 'L4' = tc(putStone(c, i) and tc(c) < 'L3':
10.                  tc(c) ← 'L3';
11.                  break;
12.              else:
13.                  for j in range(0, |c|):
14.                      if 1 = s(c, j):   continue;
15.                      if 'L5 =tc(putStone(c, i, j) and tc(c)<'S3':
16.                          tc(c) ← 'S3';   BREAK;
17.          tc(c) ← 'D3';
18.      elif 2 = nMax:
19.          for i in range(0, |c|-1):
20.              for j in range(0, |c|):
21.                  if 1 = s(c, i) or 1 = s(c, j):   continue;
22.                  elif 'L4' = tc(putStone(c, i, j):   tc(c) ← 'L2';   BREAK;
23.                  else:                               tc(c) ← 'O';
```

2.3 Promotions

Let CS be the set of all the Connection. Given $0 \le i < |c|$, $s(c, i)=1$ and $c' = putStone(c, i)$, then $c' \in$ CS. If $tc(c)=tc(c')$, c and c' are equally important; if $tc(c) \ne tc(c')$, then c' will be better than c because the added own stone strengthens c. In the case of the latter, we say that a direct promotion from c to c' occurs, moreover, always let $tc(c)<tc(c')$, which indicates that c' is better than c. By programming, we find out all the direct promotions, as is shown in Figure 3. In Figure 3, each arrow represents a possible direct promotion by adding a own stone at an empty intersection; the node at the beginning of such an arrow represents Connection type to be promoted; the node pointed by the arrow represents Connection type promoted to.

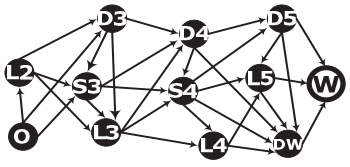


Figure 3. Possible promotions for each Connection type

There exist more indirect promotions. In Figure 3, if two nodes are direct connected by an arrow, it is a direct promotion; if one of two nodes can reach to the other via some other nodes, it is an indirect promotion. Direct promotion can be achieved by add one additional own stone, and indirect promotion needs two more stones. In fact, the relationships between Connection types shown in Figure 3 are a partial relationship on S_{CT}. Moreover, Figure 3 manifests the following two general rules to sort Connection types roughly: first, *Dead-l* < *Sleep-l* < *Live-l*; second, $T\text{-}l_1 < T\text{-}l_2$, where $2 \le l_1 < l_2 < 5$ and T can be *Dead*, *Sleep*, or *Live*. Here, '<' should be read as *strictly worse than*.

2.4 Type of Empty Intersection

There are two types of empty intersections. One type intersection belongs to certain Connection, while the other type intersection doesn't belong to any Connection. The first type intersections deserve to be considered further, while the second intersections are no use.

An empty intersection *i* of *c* being occupied by a new stone may cause a change of *c*. Of course, the results are quite different if the stone putted into *i* belong to own, or not. But, there is a Uniform perspective to understand it. To Connection *c*, that an empty intersection *i* to be occupied by its own stone will convert *c* into a new one *c*′, here, *c*′ will be no worse than *c*, can be interpreted as that the player hope this can strengthen *c* to be better; while that the same intersection *i* to be occupied by an enemy stone will slice *c* into two separate pieces, can be interpreted as the enemy want to prevent *c* to become better. That is to say, an intersection type will not alter with the stone to be put at. See the detail in Algorithm 3 and the time complexity of it is O(n) under the assumption that the type of all the Connection is known.

Algorithm 3. Implementation of ITCC

```
01.  for a Connection c :
02.      if tc(c) = 'WIN':
03.          continue;
04.      else:
05.          for each intersection i:
06.              if 1 = s(c, i):
07.                  if tc(c)<tc(putStone(c, i)):   ti(i) ← tc(putStone(c, i)));
08.                  else:                          ti(i) ← 'O';
```

3 INCREMENTAL MODEL

3.1 Connection-Based Knowledge Database

So far, we have got the types of all the Connections, as well as those of all the empty intersections. After all, Algorithm 1, 2 are all the on-line computing functions, considered the frequency of invoking those functions will be very high in the search, so we'd better save them in a table if the memory requirements can be satisfied. Such a table is called CBKDB (*Connection Based Knowledge Database*).

The simplest CBKDB is only to save the Connection type, such that $2^k+2^7+\ldots+2^n = 2^{n+1}-2^k$ items is to be saved in any *Connect*(*n*, *n*, *k*, *p*, *q*). For *Connect*(19, 19, 6, 2, 1), 1,048,512 items need to be saved, each of which only takes up $\lfloor \log^{12} \rfloor = 4$ bits, thus, about 512K bytes memories in total is required. If save all the intersection type of *connect*(19,

19, 6, 2, 1), there are no more than 20M bytes memory is paid for the storing.

The other problem of constructing CBKDB is how to allocate a unique entry to each Connection. By the propositions below, it can be solved easily.

Proposition 2. *In Connect(n, n, k, p, q),* $h(c)= 2^{|c|+1}-2^k$, $f(c)= \sum_{0\le i<|c|} s(c,i)\cdot 2^i$, $g(c)=h(c)+f(c)$. $||c_1|=|c_2||$, *iff,* $g(c_1)=g(c_2)$. □

With CBKDB, O(n) time complexity of carrying out the Connection type is decreased into O(1).

3.2 Data Structure Design

As fundamental elements, Intersections are always used to structure the state of a game position. Well, in our model, because Connection is also a basic element, the design of the data structure should support it well certainly. As is shown in Figure 4, each board can be represented by the intersections, as well as lines. Next, we will introduce how many Connections exit in a line.

Proposition 3. *Given an x-length line,* $\gamma(x, k)=\lfloor (x+1)/(k+1) \rfloor$ *there are no more than* $\gamma(x, k)$ *black Connections and no more than* $\gamma(x, k)$ *white Connections simultaneously. Where,* $\lfloor d \rfloor$ *is an operation to take the integer part of the decimal d.*

Proof. For the sake of the color's symmetry, we only need to prove that there are no more than $\gamma(x, k)$ black Connection in an x-length line**.** Let t be *the maximum number* of black Connection in such a line. Consider such an extreme case, using $t-1$ white stones to separate the whole x-length line just into $t-1$ Connection with k-length and in addition one Connection, named $\boldsymbol{c_0}$, with its length satisfied $k\le|c_0|\le 2k$, which can ensure t is the maximum number of black Connection indeed. It is easy to see that $x = (t-1)+(t-1)k+|c_0|$. Noticed the value of t should never be changed when $|c_0|$ varies from k to $2k$, such that $|c_0|\leftarrow k$ is sound. Thus, $t=\lfloor (x+1)/(k+1) \rfloor=\gamma(x, k)$.□

Considered black Connection in an x-length line, we number all the possible Connection of it as 0, 1, …, $\gamma(x, k)$, and using a array[$\gamma(x, k)$] to store them. An Connection with number i, is written as array[i] = ϕ. Suppose in a real x-length line of board, $y\le x$ Connections real exist, $0\le i<\gamma(x, k)$, each real Connection can be save into array[$\gamma(x, k)$] and we can make it true all the time that if array[i] ≠ ϕ, then for any $j < i$, array[j] ≠ ϕ.

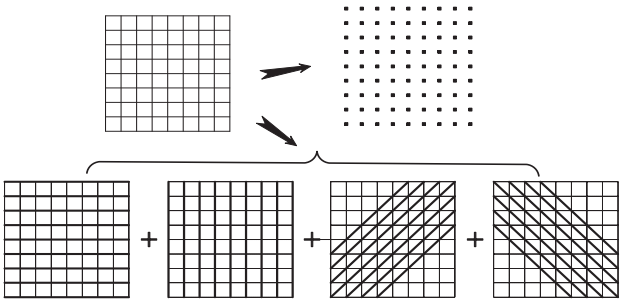


Figure 4. Board representation in *Connect*(9,9,6,2,1)

Proposition 4. *Given n≤m, on the board of Connect(m, n, k, p, q), there are: 1) exact* $\chi(m, n, k)=3(m+n)-4k+2$ *lines; 2) at most* $\lambda(m, n, k) = ((2m+n+2)\cdot\gamma(n, k)+n\cdot\gamma(m, k)+\sum_{k\le i<n}\gamma(i, k))$ *possible Connections.*

Proof. 1) It is clear that there are m horizontal n-length lines, n vertical m-length lines, $2(m-n+1)$ diagonal n-length lines, and 4 diagonal i-length lines for each i ($k\le i<n$), thus, we get $m+n+2(m-n+1)+4\sum_{k\le i<n}1 = 3(m+n)-4k+2 = \chi(m, n, k)$ lines. 2) In view of Proposition3 and the proof procedure of 1), there are no more than $m\cdot\gamma(n, k)+n\cdot\gamma(m, k)+(m+n+2)\cdot\gamma(n, k)+\sum_{k\le i<n}\gamma(i, k)$ Connections on the board simultaneously, i.e., $\lambda(m, n, k)$ Connections at most. □

The Proposition 3 and 4 given above manifest that a board can be divided into at most $\lambda(m, n, k)$ Connections. Thus, a board can be represented by an array with $\lambda(m, n, k)$ Connections.

Next, we need to number various elements of the board, such as the intersection, the line, the Connection, and the direction of line, etc., such that the same type of elements can be distinct from each other. Then, the natural mapping relations among them are stored into several constant tables in form of array. Therefore, different types of elements can be convenient to transform into each other. For example, given an intersection i and a direction d, there will be a line l, that is, (i, d) decides l, therefore a table recording such a mapping should be created. By this way, we can generate all the necessary tables.

When a stone is put at i^{th} intersection on the board, meanwhile, it may be also at k^{th} intersection of one of its own connections, c, in a certain direction. Of course, the state of i^{th} intersection on board should be updated from empty to *color*(s). However, if we quit from the updating process here, it may lead to an inconsistent situation. Because we neglect the fact that such a stone has changed the shape of c as well, which has been stored as a part of the state of game position, we need to maintain them into consistency. In this case, $f(c)\leftarrow f(c)+2^k$.

3.3 Incremental Method

The incremental method will reduce the searching cost at least in the following ways: reducing the number of units to be scanned; and saving the replaced part, called *deta* by us, to avoid the high cost inverse computation; avoiding recalculation, avoiding over-calculation; we will explain in detail how to save the cost at several critical components of a tree search.

3.3.1 Putting a Stone

When putting a stone s in a position of *Connect*(n, n, k, p, q) in 4 directions, it will strengthen no more than 4 Connections of his own, and weakens no more than 4 Connections to the opposite. Note that stone s may slice 4 opposite Connections into 8 Connections, so it will cause no more than 12 Connections, taking up $12/(6n-4k+2)$ = 13% changes at most. Also, no more than $(4\times n-3)$ intersections may change their type, taking up about $(4\times n-3)/n^2$ = 21% need to be updated on the whole board.

3.3.2 Removing a Stone

Obviously, removing a stone is the inverse operation of putting a stone. On implementation of the removing operation, what comes to mind most easily is that takes on a regular routine to execute a set of inverse instructions counter to each of putting operation.

Noticed the fact that only a small alteration has taken place on the whole board, we give the following idea: on the search goes down along the tree at a node, before altering the old state into a new one, the replaced parts are saved into a specific buffer; and then, when the search goes back, the replaced parts will be taken out from the buffer and be restore them back to where they located before, thus the state of this node will be recovered at all. Such a method is an incremental version and is better than the non-incremental version. The reason is that, in Connect6, the time of expenses in saving and restoring the changed parts of the former is less than that of the latter and the space costs of the former is not necessarily more than that of the latter.

We should remark that such an incremental method prefers DFS (*Depth First Search*) than BFS (*Breath First Search*). In the former method the buffer of *d*-ply belongs to only one *d*-ply node at any time in DFS.

3.3.3 Moves Generation

In PBIM model, we can generate moves in stages more easily, because we can generate moves of one candidate line each time. Before generating the moves of a line, we can judge that if there are possible moves we need of it. Only possible, the generation operation can be executed, such that, we avoid to scan many unused intersections of lines. Thus, Generation some moves of all possible move at each time can save the expenses of calculation, because if the moves has been developed produce a cut-off, the generation of the remaining moves is over-calculation.

3.3.4 Trans-Line Table

The CBKDB has supported the reusing of knowledge among the same shape Connections. But, there are no any measure has been taken to reuse the line now. Although the putting stone operation introduced by section 3.2 gets a great improvement by reduced the zone to scan, it is also the most expensive operation of the program still, which takes up about 50% time expenses.

It is so frequent to reach the same position via more than one path that most game playing programs resort to a so-called transposition table. Transposition table is a table to reuse the results only of the same positions, unfortunately not of the resemble positions; it still enhances the performance remarkably. As a relative small zone, the line or the Connection of a position can be more reusable than the whole position, because among the resemble positions, most of the line are same. Thus, we develop a new table to store the same lines, called *trans-line table*, which is implemented as a hash table just like the regular transposition table in chess.

To construct an identifier of a line, we use a ternary number to represent it, where 0 represents an empty intersection, 1 represents a black stone, 2 represents a white stone. We save such an identifier together with all the results of putting operation into the corresponding entry, such that when a line on board matches with an item of trans-line table, we just read out the results from trans-line table and doesn't need any recalculation.

The role of trans-line table is something like that of the operation of removing a stone, which is introduced in section 3.3. One of the differences between in them is that the former is only save the result in one line, while the other will save the results of all lines the intersection located. Well, the most importance difference between them should be that the trans-line table can reduced a number of putting stone operations, while the removing operation cannot.

4 EXPERIMENTS AND DISCUSSIONS

4.1 Experiments Setting Up

Firstly, Threat Based Search algorithm [7] runs through our experiments, because: 1) It can help to focus on the essence of the model at a large degree, and omit the other problem not closely, such as the complex evaluation function, etc. 2) Threat based search is a very useful method to *k*-in-a-row games, and has solved the well known Go-Moku [8]. 3) Considering the characters of connect6 and also to simplify the search tree, we use double-thread based search algorithm to solve the game tree. Secondly, the so-called double-threats based search algorithm is a method described as follows: one side, called attacker, must make two threats by putting two stones, while the other side, called defender, should always destroy all the two threats to keep alive. When the attacker cannot make two threats or the defender cannot destroy the threats, we do not search this position any more, and we call it known. The attacker position has one winning child is called known either; the defender position has one non-winning child-position as well. After all the child-positions are known the position is known. The program ends just when the beginning position is known. Finally, the following techniques are also used in the programs: 1) Transposition Table. Transposition table is used to diminish the transposition to be researched. The size of transposition table of our program has 2M entries, and holds 256M memories. 2) Iterative deepening. It is used to find the shortest solution path and it can achieve a many benefits to work with transposition table together. 3) Depth first search. Because incremental updating is favor of depth first search, to focus on the performance of our method, we only consider the depth first search.

To compare the performance, we write four programs of different editions On the basis of the threat-based search algorithm. They are: 1) Raw: without any improvements; 2) PBIM: use PBIM without incremental; 3) IU: use incremental updating without PBIM; 4) PBIM & IU: use all the improvements. Then we select 80 positions randomly in each depth from 3 to 7, each depth represents four stones of the two sides. Now the performance is displayed in the figure 5 and 6.

Figure 5 shows the results in the non-winning position set,

while figure 6 shows the results in winning position set. The x axis in the figures means the depth of the programs searched while solving the game tree. For example 4 in figure 5 means at most 4 more stones were putted on the board when to prove the position is non-winning from the beginning position when defender is smart. The y axis means time used in each depth, the more time the program used the slower the program is. But we should say the result is not accurate when in 4-depth and 8-depth because of the inaccuracy in computer. At the same time, it rarely wins in short depth so the depths under 16 had not counted. As a result the program IU&PBIM is the fastest program while the program Raw is the slowest, same according to the two figures.

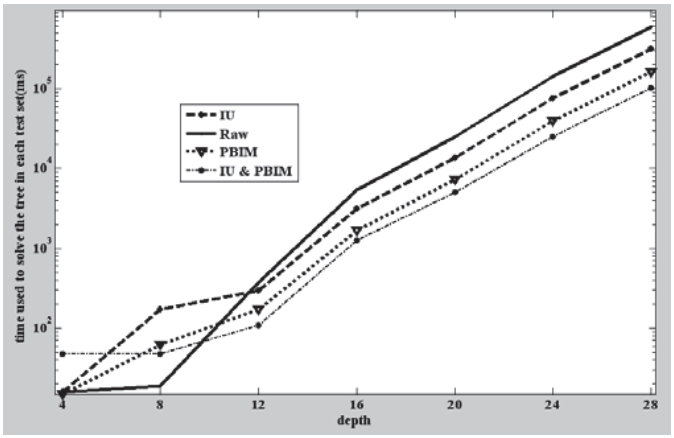


Figure 5. Time cost in winning positions

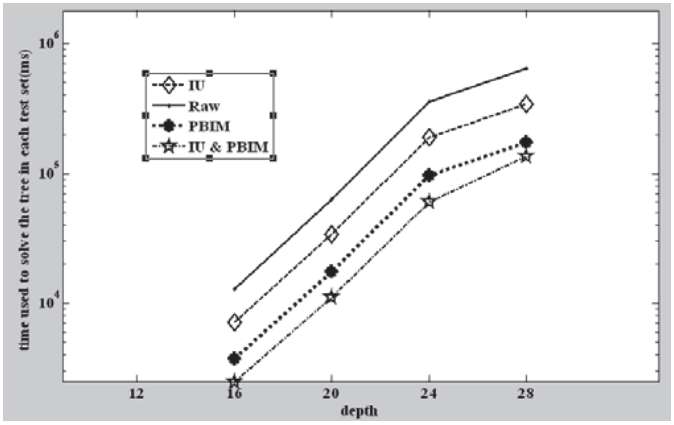


Figure 6.time cost in non-winning positions

We test the programs in the same computers, an Intel core pc clocked at 3.00GHz and1.98GHz, at the same time with the memory of 2.00GB. The data in table 1 is nodes each program searched in one millisecond approximately represent the speed of each program; the speed is higher when the number is bigger.

Table 1. The number nodes per milli-seconds

	Raw	IU	PBIM	IU&PBIM
nodes/ms	40	71	117	201

4.2 Discussions

In this paper, the model, named PBIM, which enhanced the program about 3 times in Connect6, and then the measure of incremental updating is taken, which also enhances about 1.9 times again. In our experiments, we just take the threat based search as test bed. Although we believe that this model in *k*-in-a-row games and its relevant methods will work well in other search algorithms, such as alpha-beta search, proof number search, etc., it needs further researches yet. In addition, we also think it is feasible that the model also can be partly adopted by such a game, which has a large board and each time only updates a small part at each node.

REFERENCES

[1] I-Chen Wu and Dei-Yen Huang. A New Family of k-in-a-row Games. In *Proceedings of The 11th Advances in Computer Games Conference*, 88-100, 2005.

[2] L. Stiller, Multilinear algebra and chess endgames. *Games of No Chance*, 29, 1996.

[3] R. Lake, J. Schaeffer, Lu P. (1994). Solving Large Retrograde-Analysis Problems Using a Network of Workstations. *Advances in Computer Chess 7*, pages: 135-162. 1994.

[4] A. Felner, U. Zahavi, R. Holte, j. Schaeffer. Dual lookups in pattern databases. In *International Joint Conference on Artificial Intelligence*. 2005

[5] A. Junghanns, J. Schaeffer. Single-Agent Search in the Presence of Deadlocks. In *Association for the Advancement of Artificial Intelligence*, 98.

[6] Bruno Bouzy, Incremental updating of objects in Indigo. in: *Proceedings of the fourth Game Programming Workshop*, pages: 179-88, 1997.

[7] LV Allis, *Searching for solutions in games and artificial intelligence*. Ph.D. thesis, Maastricht, University of Limburg, 1994.

[8] L.V. Allis, H.J. van den Herik, M.P.H. Huntjens, Go-Moku solved by new search techniques, *Jorurnal of Computational Intelligence*. 12 (1): 7–24, 1995.