Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2015, Article ID 576201, 20 pages
http://dx.doi.org/10.1155/2015/576201

Research Article

Mining Experiential Patterns from Game-Logs of Board Game

Liang Wang,[1] Yu Wang,[1] and Yan Li[2]

[1]*School of Computer Science and Technology, Hebei University, Baoding 071000, China*
[2]*School of Mathematics and Information Science, Hebei University, Baoding 071002, China*

Correspondence should be addressed to Liang Wang; wangl@hbu.cn

Received 21 September 2014; Revised 20 December 2014; Accepted 22 December 2014

Academic Editor: Hanqiu Sun

In board games, game-logs record past game processes, which can be regarded as an accumulation of experience. Similar to a real person, a computer player can gradually increase its skill by learning from game-logs. Therefore, the game becomes more interesting. This paper proposes an extensible approach to mine experiential patterns from increasing game-logs. The computer player improves its strategies by utilizing these growing patterns, just as it acquires experience. To evaluate the effect and performance of the approach, we designed a sample board game as a test platform and elaborated an experiment consisting of a series of tests. Experimental results show that our approach is effective and efficient.

1. Introduction

Gaming is one of the main intellections in our daily life. To some extent, gaming shows the wisdom of humankind more directly. Artificial intelligence for games (Game AI) is an important subfield of AI research. The current generations of computer games offer an interesting testbed for Game AI [1]. Meanwhile, applying AI techniques to game design will make the games more interesting. The research of Game AI plays an important role in promoting the development of the computer game industry.

Board game is one of the earliest objects that AI focused on [2, 3]. By contrast with other types of games, board games reflect human's intelligence more purely and thus attract the interest of AI researchers [4, 5]. At first, researchers concentrated on the algorithm design about the chess road searching and the chess manual matching and made some important progresses. John von Neumann presented the Minimax algorithm in 1920s [6]. Shannon [2] proposed the game tree searching algorithm with his chess program in 1950. Knuth and Moore deeply analyzed the alpha-beta pruning algorithm in 1958 [7]. Then, studies on AI in board games were widely launched [8–13]. Many contests of board games between machines and real persons showed the advances in AI research [14–16]. Recently, researchers find that game-logs (game records) cover a lot of important and interesting knowledge. Chen et al. [17] proposed an approach to abstract expert knowledge from annotated Chinese chess game records. GO game records are used for the life and death predicting, the pattern acquisition, and the pattern matching [18–20]. Esaki and Hashiyama [21] carried out some experiments to extract human players' strategies from the game records of Shogi. Weber and Mateas [22] used a data mining approach to process game-logs for the strategy prediction. Takeuchi et al. [23] presented a way of evaluating game tree search methods by game records. Moreover, Wender [24] expounded how to use data mining and machine learning techniques to analyze the game-logs in his project.

The common design of board games is to factitiously set the difficulty level on an invincible computer player, but this way is just a simulation. However, human players often expect a computer player to act and think as a real person rather than a superman. Game-logs in a board game are considered as cumulative experience, which will become richer and more effective with the increase of game times. With the accumulation of the experience, computer players can gradually enhance their intelligence level, just like human beings. Data mining techniques can be used to extract experience from game-logs. Nevertheless, there are few works on this topic.

In this paper, an approach is presented to mine experiential patterns from the growing game-logs of a sample board

game. The computer player can learn from these patterns to improve its intelligence naturally. Firstly, a Chinese checkers game was designed as a test platform for collecting game-logs and launching empirical studies. Then, series of algorithms based on a sequence-tree were proposed to mine experiential patterns from game-logs. These patterns included Experience Rules, Key States, and Checker Usage. Finally, the mined patterns were applied back to the test platform, and then a series of experimental tests was elaborated to verify the effectiveness and performance of our approach.

Experimental results demonstrate that our approach is effective and efficient and easy to be transplanted to fit for other types of games.

2. Methods

The application of the experiential pattern mining requires two basic conditions listed below.

(i) The computer players of a game have the minimum ability to play the game correctly without any experiential pattern.

Doing so ensures that the game can run properly in the initial stage.

(ii) There are sufficient and valid game-logs to make the pattern mining task runnable.

Only when the number of game-logs reaches a certain size can the regularity hidden in them appear. In addition, our purpose is to make computer players learn from human players' strategies. So, these game-logs should be from human players' games.

To satisfy the above conditions, we designed a simplified Chinese checkers game as a test platform to provide the basic move algorithms and collect the game-logs. Subsequent sections cover the following topics in detail:

(a) the concepts concerned with the game and the mining algorithms;

(b) the algorithms for mining experiential patterns;

(c) the methods for applying experiential patterns to the game.

2.1. Key Concepts. The traditional Chinese checkers game has flexible gameplay (see [25] for more details). In this work, the game was simplified to be more convenient for experimenting (the design details are provided in Appendix). The key concepts used in the following sections are briefly explained below.

(i) Checker. The checker means the position used to put pieces on the chessboard. Each checker has its own coordinate.

(ii) Checkers-State. The checkers-state describes all the checkers on the chessboard and can be traversed to get or update the information of every checker.

(iii) Piece. Each player holds ten pieces. The coordinate of a piece is dependent on the current position of the piece.

(iv) Pieces-State. Pieces-states are used to store the coordinates of pieces. A pieces-state is a set that consists of several subsets. The number of the subsets depends on how many players are there in the game. Each subset contains ten elements, which correspond to the ten pieces of a player. For 2-player mode, there are two subsets, called the active side (SAS) and the passive side (SPS), respectively.

(v) Game-Log. During gaming, once a piece moves a step, the test platform will save the corresponding pieces-state to a game-log. The pieces-states in a game-log are sorted by their creation time. Hence, a game-log is a sequence of pieces-states. Separate XML files are used to store the game-logs.

(vi) Experiential Pattern. Experiential patterns are mined from game-logs and easily used by computer players. In this work, we considered three kinds of experiential patterns, including Experience Rule, Key State, and Checker Usage.

An Experience Rule is similar to an if-then rule. If pieces-state PS_a appears, then the player should achieve the pieces-state PS_b by moving a piece. The rule can be represented as $PS_a \rightarrow PS_b$, where PS_a is called last state and PS_b next state.

Key States are a group of particular pieces-states. They appear in victor games frequently and thus indicate some good game situation. By algorithm designing, one can help the computer player in approaching these Key States. Furthermore, approaching the Key States may also promote the applying of some Experience Rules, because some Key States may just be the last states or the next states.

Checker Usage describes the usage of a checker and is denoted by CU. The CU of a checker can be calculated by

$$CU_C = \begin{cases} \dfrac{FC\_Count_C}{N}, & N > 0 \\ 1, & N = 0, \end{cases} \tag{1}$$

where CU_C represents the CU of checker C, FC_Count_C holds the times that C has been used as the falling checker, and N stores the number of game-logs in database.

2.2. Mining Experiential Patterns. By using some traditional move algorithms, the computer player may figure out the best move within several rounds. However, being subject to the computing capability of the computer and the requirement of game runtime, the computer player is hard to make more skillful moves or longer layouts. Humans improve their chess skill by accumulating the playing experience. A computer player can simulate this process to study experience from humans' excellent games and thus make the moves more strategic. The three experiential patterns introduced in Section 2.1 are mined from the game-logs generated in the games of human-human mode. The data mining process is run in server-side, and its workflow is shown in Figure 1.

Table 1: Attributes of the *PiecesState*.

Attribute name	Description	Data type	Restrictions
ID	The ID of pieces-state.	Big integer	Unique, nonnull
SAS_0	The coordinate of the first piece in active side.	Coordinate	Nonnull
SAS_1	The coordinate of the second piece in active side.	Coordinate	Nonnull
⋮	⋮	⋮	⋮
SAS_9	The coordinate of the tenth piece in active side.	Coordinate	Nonnull
SPS_0	The coordinate of the first piece in passive side.	Coordinate	Nonnull
SPS_1	The coordinate of the second piece in passive side.	Coordinate	Nonnull
⋮	⋮	⋮	⋮
SPS_9	The coordinate of the tenth piece in passive side.	Coordinate	Nonnull
Count	The times this pieces-state appeared in victor games.	Big integer	Nonnull

Table 2: Attributes of the *PiecesStateSequence*.

Attribute name	Description	Data type	Restrictions
ID	The ID of the sequence.	Big integer	Unique, nonnull
SequenceStr	Sequential string.	Long text	Nonnull

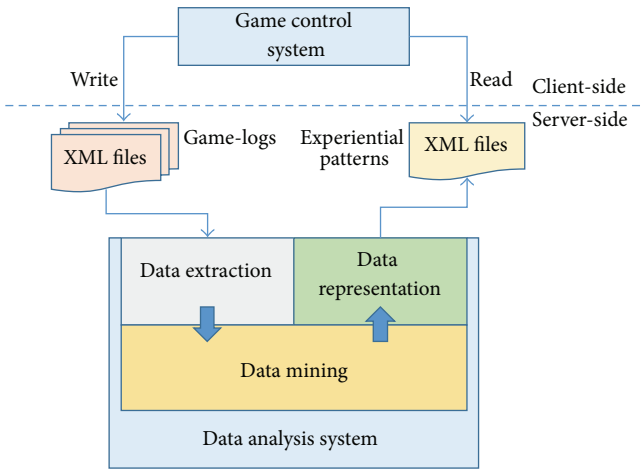

Figure 1: Workflow of the data mining process.

As shown in Figure 1, the data mining process consists of three phases, including data extraction, data mining, and data representation.

2.2.1. Data Extraction. Each game makes a single game-log file, and this will generate lots of XML files stored in server-side. These XML files are not suitable to be the direct data source of data mining. Therefore, the sequential data with consistent format needs to be extracted from game-logs and stored in database first.

Three data entities, *PiecesState*, *PiecesStateSequence*, and *CheckerUsage*, are used to format the game-logs. Their attributes are, respectively, shown in Tables 1, 2, and 3. The pieces-states in *PiecesState* are distinct, and each of them has a unique ID. The sequences in *PiecesStateSequence* are repeatable, and each of them corresponds to a game-log. The tuples in *CheckerUsage* have the same number as the checkers on the chessboard.

Table 3: Attributes of the *CheckerUsage*.

Attribute name	Description	Data type	Restrictions
ID	Serialized ID	Big integer	Unique, nonnull
c	The coordinate of the falling checker	Coordinate	Nonnull
Times	The usage of the falling checker	Big integer	Nonnull

Algorithm 1 describes the process of data extraction.

In Algorithm 1, lines (29)–(37) add a win/loss mark to the nondraw game and use minus to mark the pieces-states created by the loss player. Lines (38)–(41) combine all the PiecesState IDs from the same game into a single sequential string.

2.2.2. Data Mining. Experiential pattern mining means finding useful patterns from a large number of sequences composed of pieces-states. The types of these patterns include but are not limited to those mentioned in Section 2.1. Mining the Experience Rules, which are the most important experiential patterns in this work, is an objective of sequential pattern mining. Sequential pattern mining is a topic of data mining concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence [26]. In this field, many efforts of mining sequential patterns have been devoted to developing efficient algorithms, such as *GSP* [27], *SPADE* [28], *CloSpan* [29], *PrefixSpan* [30], and *MEMISP* [31]. In recent years, researchers have paid more attention to the applied research of sequential pattern mining as discussed elsewhere [32–36].

In this section, we describe the methods for mining the three kinds of experiential patterns explained in Section 2.1.

Experience Rule. Experience Rules describe the experience of the human players most directly. The problem of mining Experience Rules can be described as follows. Let S be

```
Procedure DataExtraction
Input:
  statesD, the data table of PiecesState in database.
  files: the XML files of game-logs.
Output: Non.
Method:
(01) let stateIds be the set of PiecesState ids;
(02) let file be a file of game-log;
(03) let statesF be the set of the pieces-states extracted from game-logs;
(04) let newStateId be the new PiecesState id;
(05) while files.empty == false do
(06)   file = files.getOne(); //Read a game-log.
(07)   Extract the pieces-states from file, and save them to statesF;
(08)   Extract the game result from file, and save it to result;
(09)   for i = 0 to statesF.length − 1 do
(10)     //Get the sequence of ids of PiecesState.
(11)     ifFound = false;
(12)     Update the times field of CheckerUsage with statesF[i].
(13)     for j = 0 to statesD.length − 1 do
(14)       if equal(statesF[i], statesD[j].state) then
(15)         stateIds.add(statesD[j].id);
(16)         isFound = true;
(17)         count += result;
(18)         break;
(19)       endif
(20)     endfor
(21)     if isFound == false then
(22)       Insert statesF[i] into PiecesState;
(23)       count = result;
(24)       Save the new id to newStateId.
(25)       stateIds.add(newStateId);
(26)       statesD.add(newStateId, statesF[i]);
(27)     endif
(28)   endfor
(29)   if result != 0 then
(30)     k = 1;
(31)     if result == −1 then
(32)       k = 0;
(33)     endif
(34)     for n = 0 to stateIds.length − 1 do
(35)       stateIds[n] *= (−1) ^ ((i − k) % 2);
(36)     endfor
(37)   endif
(38)   sequenceString = stateIds[0];
(39)   for p = 0 to stateIds.length − 1 do
(40)     sequenceString += (":" + stateIds[p]);
(41)   endfor
(42)   Insert sequenceString into PiecesStateSequence;
(43)   delete(file); //Delete the current game-log.
(44) endwhile
```

Algorithm 1: *DataExtraction*.

a set of items, where each item is a sequence of pieces-states. Given a minimum support threshold *minSupport*, the aim is to find out all the consecutive binary subsequences (CBSSs) whose frequency is not less than *minSupport*. Each CBSS consists of two pieces-states that are consecutive in the original sequence, and this can be regarded as a constraint in data mining. In this paper, an algorithm based on pattern-growth is proposed for mining CBSSs. This algorithm uses a sequence-tree (see Figure 2) as the data structure to load all the sequences in database. The sequence-tree is different from the FP-tree [37] of frequent pattern mining, and we do not have to scan database previously to obtain the supports

```
Procedure SequenceTreeCreation
Input: sequencesD, the data table of PiecesStateSequence in database.
Output: root, the root node of the sequence-tree.
Method:
(01) let stateIds save the ids of PiecesState;
(02) root = new Node(value = 0, branches = null);
(03) for i = 0 to sequencesD.length − 1 do
(04)    stateIds = sequencesD[i].sequenceStr.split(":");
(05)    //Eliminate redundant pieces-state of stateIds.
(06)    for  j = stateIds.length − 1 to 2 do
(07)       for  k = j − 2 to 0 by 2 do
(08)          if stateIds[j] ==stateIds[k] then
(09)             stateIds.delete(k + 1, j);
(10)             j = k;
(11)          endif
(12)       endfor
(13)    endfor
(14)    node = root;
(15)    for m = 1 to stateIds.length − 1 do
(16)       if node.value == 0 then
(17)          node.value = |stateIds[0]|;
(18)       endif
(19)       isFound = false;
(20)       if node.branches != null then
(21)          for  n = 0 to node.branches.length − 1 do
(22)             if node.branches[n].next.value == |stateIds[m]| then
(23)                node.branches[n].weight += (stateIds[m]/|stateIds[m]|);
(24)                node = node.branches[n].next;
(25)                isFound = true;
(26)                break;
(27)             endif
(28)          endfor
(29)       endif
(30)       if isFound == false then
(31)          branch = new Branch(
(32)             weight = stateIds[m]/|stateIds[m]|,
(33)             next = new Node(value = |stateIds[m]|, branches = null)
(34)          );
(35)          node.branches.add(branch);
(36)          node = node.branches[node.branches.length − 1].next;
(37)       endif
(38)       //Merge the nodes of same values.
(39)       BranchShifting(root, node);
(40)    endfor
(41) endfor
(42) return root;
```

Algorithm 2: *SequenceTreeCreation*.

of every pieces-state. Algorithm 2 describes the procedure of creating a sequence-tree.

In Figure 2, *value* stores the ID of the pieces-state, *weight* represents the support of the related CBSS, and *next* stores the linkage pointing to the next node.

On a certain player's perspective, if two same pieces-states appear, the moves between the two pieces-states will become noneffective interference data. Eliminating these data will reduce the unnecessary data analyses and improve the reliability of mining results (see lines (05)–(13) in Algorithm 2).

Not all the frequent CBSSs are good, since some of them may be from losing games. The support count of a CBSS denoted by *suppCount* can be calculated by

$$suppCount = posSuppCount - negSuppCount, \quad (2)$$

where *posSuppCount* is the support count of the CBSS generated in winning games and *negSuppCount* in losing games.

By means of the win/loss mark, the *suppCount* of each CBSS has been calculated during the process of creating the sequence-tree. Hence, the weight of each branch is

```
Procedure BranchShifting
Input:
  root, the root node of the sequence-tree.
  node, the pointer that points to the current node.
Output:
  Updated node.
  isShifted, the result of shifting.
Method:
(01) if node.value == root.value then
(02)   node = root;
(03)   return true;
(04) else
(05)   if root.branches == null then
(06)     return false;
(07)   else
(08)     for i = 0 to root.branches.length − 1 do
(09)       if BranchShifting(root.branches[i].next, node) == true then
(10)         return true;
(11)       endif
(12)     endfor
(13)     return false;
(14)   endif
(15) endif
```

Algorithm 3: *BranchShifting*.

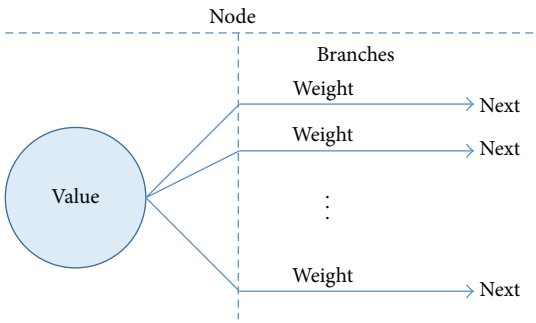

Figure 2: Data structure of a node on the sequence-tree.

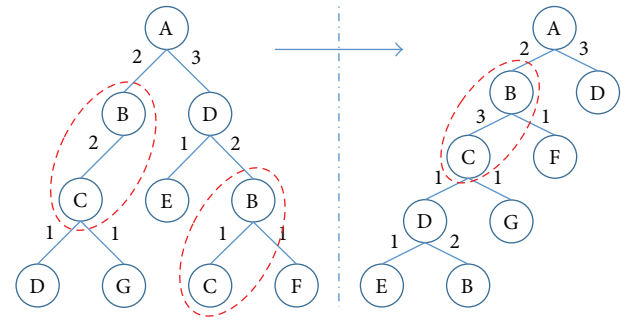

Figure 3: Branch shifting. The two pairs of nodes B-C marked with dotted circles on the left tree should be merged together, and their branch weights should be added up too. The right tree is the result of this branch shifting.

equal to the *suppCount* of the relevant CBSS. Moreover, the same pairs of nodes are not allowed to appear on the same sequence-tree; otherwise the *suppCount* values will be incomputable (as illustrated in Figure 3). To solve this problem, the algorithm for shifting branches is designed (see Algorithm 3).

To obtain the frequent CBSSs, the following formula is given:

$$supp = \frac{suppCount}{totalSeqs} \times 100\%, \tag{3}$$

where *supp* denotes the support of a CBSS and *totalSeqs* denotes the total number of the sequences in database. Given a *minSupport*, the frequent CBSSs are those whose supports are not less than *minSupport*. All the frequent CBSSs can be found out by traversing the sequence-tree only once, and the script to do so is shown in Algorithm 4.

In Algorithm 4, the structure of the object SequentialPattern is shown in Figure 4. The found CBSSs will serve as the data source for selecting Experience Rules.

Key State. Given a threshold percentage *minFreq*, Key States are those pieces-states of the frequencies not less than *minFreq*. The frequency (denoted by *freq*) of a pieces-state can be calculated by

$$freq = \frac{count}{N} \times 100\%, \tag{4}$$

where N stores the total number of sequences in database and *count* denotes the occurrence number of the pieces-state in the net winning games. The value of *count* is figured out

```
Procedure CBSSFinding
Input:
   minSupport, the minimum support threshold.
   root, the root node of the sequence-tree.
   total: the total number of sequences.
   sps, the set of the found CBSSs.
Output:
   Updated sps.
Method:
(01) let supp be the support of a CBSS;
(02) let sp be the instance of SequentialPattern;
(03) if root.branches == null then
(04)    return;
(05) endif
(06) for i = 0 to root.branches.length − 1 do
(07)    supp = root.branches[i].weight/total;
(08)    if supp ≥ minSupport then
(09)       sp = new SequentialPattern();
(10)       sp.premise = root.value;
(11)       sp.result = root.branches[i].next.value;
(12)       sp.support = supp;
(13)       sps.add(sp);
(14)    endif
(15)    CBSSFinding(minSupport, root.branches[i].next, total, sps);
(16) endfor
```

Algorithm 4: *CBSSFinding*.

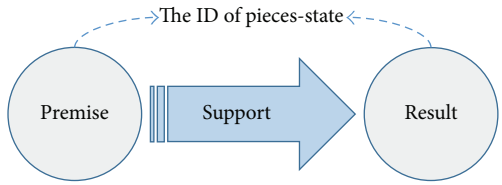

Figure 4: The structure of SequentialPattern.

during the process of data extraction (see lines (17) and (23) in Algorithm 1).

In respect of threshold selection, adjusting *minFreq* dynamically in accordance with the number of Experience Rules can make the Key States more efficient.

Checker Usage. This pattern can be calculated by formula (1), in which the FC_Count_C is figured out during the process of data extraction (see line (12) in Algorithm 1).

2.2.3. Data Representation. In this phase, the main task is to make the mined experiential patterns recognizable to the computer player. The experiential pattern here mainly refers to Experience Rule. The following format is used to represent an Experience Rule:

$$premise \longrightarrow result : support. \tag{5}$$

According to the data structure of the Experience Rule, the process of data representation can be described as follows. Replace the IDs, which denote the last states or next states of Experience Rules, with the corresponding pieces-states, and then write these pieces-states to an XML file in server-side.

However, the mined patterns may not be all reliable. For example, given a *minSupport*, the obtained Experience Rules are as follows:

$$PS_a \longrightarrow PS_b : 15\% \qquad PS_a \longrightarrow PS_c : 13\% \qquad PS_a \longrightarrow PS_d : 17\%. \tag{6}$$

That is, one last state has multiple next states. When pieces-state PS_a appears, it is clearly reliable to select the pieces-state PS_d, which is of the highest *support*. Hence, the reliability selection should be done before data presentation. Algorithm 5 shows this process.

2.3. Applying Experiential Patterns. The way of applying experiential patterns depends on the specific gameplay. In this section, we give a computer player's strategy as shown in Figure 5 and then explain how to apply the mined experiential patterns to the actual game.

In the course of the game, the computer player uses Experience Rules first. Algorithm 6 demonstrates the procedure of matching Experience Rules.

When there are no available Experience Rules, the computer player will automatically approximate a certain Key State. Doing so can promote the formation of some good game situation and may activate some Experience Rules. Obviously, the approached Key State is the nearest pieces-state after the current one. So, all of the Key States are sorted

```
Procedure ReliabilitySelection
Input:
   minSupport, the minimum support threshold.
   total: the total number of sequences.
   sequencesD, the data table of PiecesStateSequence in database.
Output:
   sps, the set of reliable Experience Rules.
Method:
(01) root = CreatSequenceTree(sequencesD);
(02)    CBSSFinding(minSupport, root, total, sps);
(03) for i = 0 to sps.length − 2 do
(04)   for j = i + 1 to sps.length − 1 do
(05)     if sps[i].premise == sps[j].premise then
(06)       if sps[i].support < sps[j].support then
(07)         sps.delete(i);
(08)         i − −;
(09)         break;
(10)       else
(11)         sps.delete(j);
(12)         j − −;
(13)       endif
(14)     endif
(15)   endfor
(16) endfor
```

Algorithm 5: *ReliabilitySelection*.

by their occurrence time. Let PS_a and PS_b be two different pieces-states. Their order is defined below.

Definition 1. Assume that $PS_a \neq PS_b$. If $PS_a \cdot \mathrm{SAS} \geq PS_b \cdot \mathrm{SAS}$ and $PS_a \cdot \mathrm{SPS} \geq PS_b \cdot \mathrm{SPS}$, then PS_a is behind PS_b, which can be expressed as $PS_a \geq PS_b$.

The concept of dissimilarity (denoted by *Diss*) is used to represent the closeness between two pieces-states. The *Diss* between PS_a and PS_b is equal to the least moves from PS_a to PS_b. The less the value of *Diss*, the closer the two pieces-states. For calculating *Diss*, the following definitions are given.

Definition 2. Let P be a piece and let PS be a set of pieces-states. If the coordinate of P is included in PS, then P belongs to PS, which can be expressed as $P \in PS$.

Definition 3. Let C_a and C_b be two checkers, (x_a, y_a) and (x_b, y_b) their coordinates, respectively, and $x_a + y_a \geq x_b + y_b$. Then, the minimal number of steps for a move from C_a to C_b is called move distance from C_a to C_b and denoted by MD_{ab}, which can be calculated by

$$MD_{ab} = \varphi(x_a - x_b) + \varphi(y_a - y_b), \tag{7}$$

where the function φ is defined as

$$\varphi(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0. \end{cases} \tag{8}$$

Definition 4. Let P_a and P_b be two pieces, C_a and C_b their checkers, respectively, PS_a and PS_b two pieces-states, $P_a \in PS_a \cdot \mathrm{SIS}$, and $P_b \in PS_b \cdot \mathrm{SIS}$. Then, the minimal number of steps for a move from C_a to C_b is called the Distance in Active Side between P_a and P_b, which is denoted by DAS_{ab}. Similarly, one can define the Distance in Passive Side between P_a and P_b, which is denoted by DPS_{ab}.

Algorithm 7 demonstrates the calculation of the distance between two pieces.

Let PS_a and PS_b be two pieces-states. According to Definitions 2–4, the dissimilarity matrices between PS_a and PS_b are defined as follows:

$$DissMatrixAS = \begin{bmatrix} \mathrm{DAS}_{00} & \cdots & \mathrm{DAS}_{09} \\ \vdots & \ddots & \vdots \\ \mathrm{DAS}_{90} & \cdots & \mathrm{DAS}_{99} \end{bmatrix}$$
$$DissMatrixPS = \begin{bmatrix} \mathrm{DPS}_{00} & \cdots & \mathrm{DPS}_{09} \\ \vdots & \ddots & \vdots \\ \mathrm{DPS}_{90} & \cdots & \mathrm{DPS}_{99} \end{bmatrix}, \tag{9}$$

where *DissMatrixAS* is defined on the active side and *DissMatrixPS* on the passive side. Then, the *Diss* between two pieces can be calculated by

$$Diss = \min\left(\sum_{a=0}^{9}\sum_{b=0}^{9}\mathrm{DAS}_{ab}X_{ab} + \sum_{a=0}^{9}\sum_{b=0}^{9}\mathrm{DPS}_{ab}Y_{ab}\right), \tag{10}$$

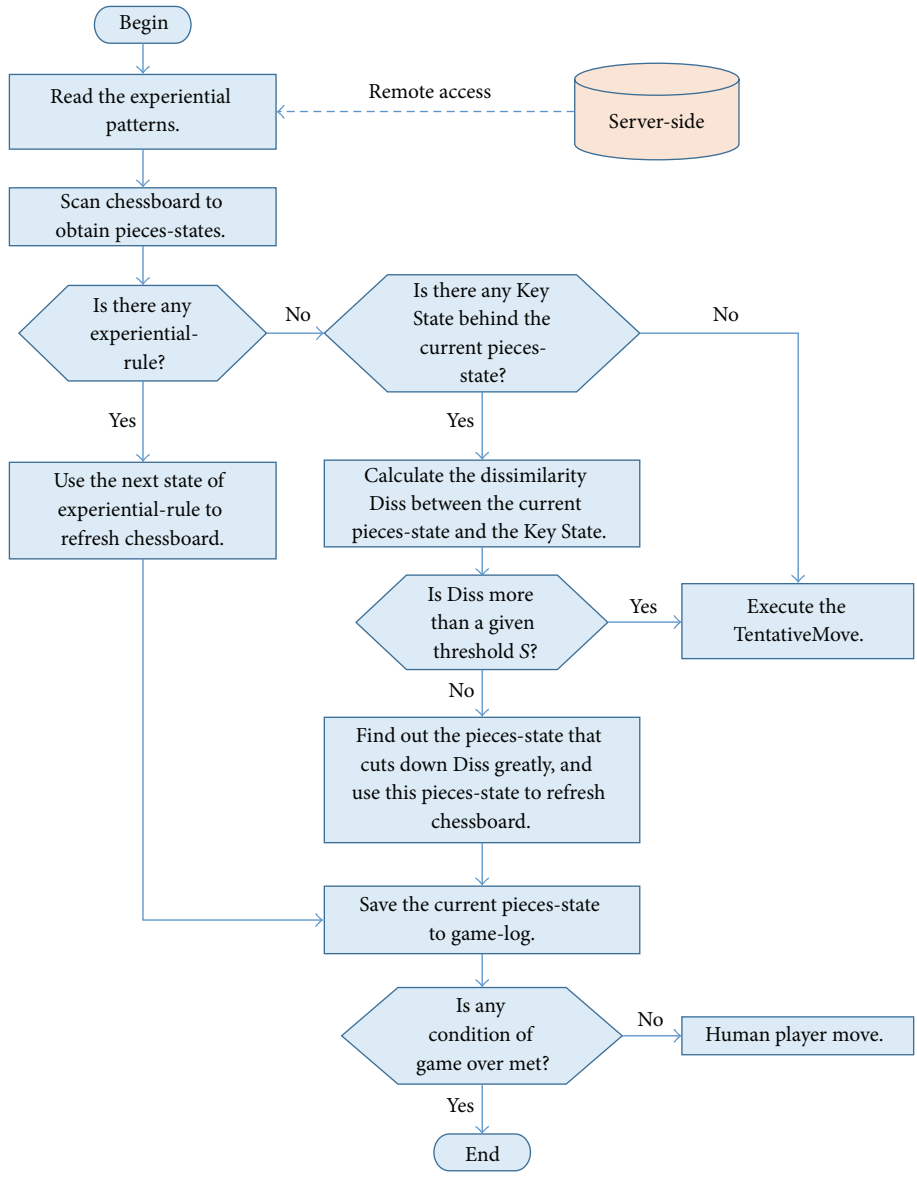

Figure 5: Strategy of the computer player.

where X_{ab} and Y_{ab} satisfy the following conditions, respectively:

$$\begin{aligned}
&\sum_{a=0}^{9} X_{ab} = 1 \\
&\sum_{b=0}^{9} X_{ab} = 1 \\
&a, b = 0, 1, 2, \ldots, 9 \\
&X_{ab} = 0, 1, \\
&\sum_{a=0}^{9} Y_{ab} = 1 \\
&\sum_{b=0}^{9} Y_{ab} = 1 \\
&a, b = 0, 1, 2, \ldots, 9 \\
&Y_{ab} = 0, 1.
\end{aligned} \tag{11}$$

It follows that the calculation of the *Diss* between pieces is a typical assignment problem in linear programming. This problem can be solved by *Hungarian* algorithm as discussed by Edmonds [38].

When there are no Key States behind the current pieces-state or the *Diss* between the nearest Key State and the current pieces-state is more than a given threshold, the computer

```
Procedure ExperienceRuleMatching
Input:
   PS, the current pieces-state.
   ERs: the set of Experience Rules.
Output:
   nextPS, the next pieces-state.
Method:
(01) for each er in ERs do
(02)    if PS == er.last then
(03)       nextPS = er.next;
(04)       break;
(05)    else
(06)       nextPS = null;
(07)    endif
(08) endfor
```

Algorithm 6: *ExperienceRuleMatching*.

player will execute a tentative move algorithm named *TentativeMove*, which was designed based on the traditional alpha-beta pruning algorithm [7] (see Appendix for more details). Moreover, the pattern Checker Usage is one of the parameters of *TentativeMove*.

3. Results and Discussions

In order to evaluate the effect and performance of our proposed approach, we conducted an experiment on our designed test platform mentioned in Section 2. The experiment contains a series of tests. To maintain the objectivity of the experiment, we chose the same group of testers, who are ten experienced human players.

3.1. Test in Human-Computer Mode without any Experiential Patterns. Each tester played against the computer player for 30 games, serving as offensive player (OffP) and defensive player (DefP) alternately. The computer player was only allowed to use *TentativeMove* using a default value of Checker Usage as the parameter. A total of 300 game-logs in this test were recorded. The test results are shown in Table 4.

Furthermore, we observed the performance of our algorithms by using the parameter ATMC (the average time for each move of the computer player). A script embedded into the program was used to calculate the response time of the computer player during testing and give the value of ATMC in the end. In this test, the value of ATMC was 5.85 s (s = seconds).

Descriptions of Effects. (a) The computer player has the ability to play correctly; (b) the winning rate of the computer player in competition with every tester is not more than a half; (c) the computer player spends a little long time to move a piece.

Discussions. (a) Condition (i) mentioned at the beginning of Section 2 is satisfied by the TentativeMove algorithm; (b) the moves driven by TentativeMove may not be optimal due to the error of its evaluation function and the restraint of searching depth, which affects the winning rate of the computer player; (c) TentativeMove has a deeper recursion depth. As the game goes on, the checkers to be searched in each recursive layer will increase rapidly, which degrades the performance of TentativeMove.

Table 4: Data records of the test in human-computer mode without any experiential pattern.

Tester	Gts	Role	Wts	Anrs	Wr (%)
00	15	OffP	5	45	33.33
00	15	DefP	4	42	26.67
01	15	OffP	5	47	33.33
01	15	DefP	3	45	20.00
02	15	OffP	4	44	26.67
02	15	DefP	1	40	6.67
03	15	OffP	3	45	20.00
03	15	DefP	2	41	13.33
04	15	OffP	4	42	26.67
04	15	DefP	4	47	26.67
05	15	OffP	4	44	26.67
05	15	DefP	1	40	6.67
06	15	OffP	6	49	40.00
06	15	DefP	4	42	26.67
07	15	OffP	2	37	13.33
07	15	DefP	2	36	13.33
08	15	OffP	6	45	40.00
08	15	DefP	3	45	20.00
09	15	OffP	5	45	33.33
09	15	DefP	2	40	13.33

Note: Gts: game times; Role: role of the computer player; Wts: winning times of the computer player; Anrs: average number of the rounds per game; and Wr: winning rate of the computer player.

3.2. Test in Human-Human Mode for Collecting Game-Logs. Each tester played with all the others for 30 games, serving as OffP and DefP alternately. A total of 1350 game-logs in this test were recorded. This satisfied condition (ii) mentioned at the beginning of Section 2. Subsequently, additional tests in human-computer mode were performed to select the appropriate *minSupport* and *minFreq*. These tests were similar to those in Section 3.1, and the results are shown in Table 5. Finally, the experiential pattern mining program was executed with the selected *minSupport* and *minFreq*. As a result, we gained 528 Experience Rules and 575 Key States, and the Checker Usage of every checker was updated in real time.

As shown in Table 5, when *minSupport* was 2% and *minFreq* was 1%, the crest value of *Auep* was reached, which meant the mined experiential patterns were better utilized.

3.3. Test in Human-Computer Mode with Experiential Patterns. The test plan here was the same as that in Section 3.1, except for the applying of the experiential patterns obtained in the test mentioned in Section 3.2. The results are shown in Table 6, and the comparison with the test results in Section 3.1

```
Procedure PiecesDistanceCalculation
Input:
  PS, the current pieces-state.
  P_a, piece P_a.
  P_b, piece P_b.
Output:
  Dab, the distance between P_a and P_b.
Method:
(01) let CS be the checkers-state;
(02) let vAllFC be a vector to store the falling checkers of every layer.
(03) //The maximum distance between P_a and P_b.
(04) Calculate the move distance MDab;
(05) if MDab == 0 or MDab == 1 or MDab == 2 then
(06)   //If MDab is less than 3, there is no less distance.
(07)   return MDab;
(08) endif
(09) for i = 0 to 9 do
(10)   CS[PS·SAS[i]·x][PS·SAS[i]·y] = 1;
(11)   CS[PS·SPS[i]·x][PS·SPS[i]·y] = −1;
(12) endfor
(13) FindVirtualFallingCheckers(0, CS, P_a · x, P_a · y, true, vAllFC, 0);
(14) isBreak = false; //Denotes the end of searching.
(15) Dab = MDab; //The distance in the worst case.
(16) for each allFC in vAllFC do
(17)   for each c in allFC do
(18)     if allFC.index == (MDab − 1) then
(19)       isBreak = true; break;
(20)     endif
(21)     if c · x == P_b · x and c · y == P_b · y then
(22)       Dab = allFC.index + 1; isBreak = true;
(23)       break;
(24)     endif
(25)     if allFC.index == (MDab − 2) then break; endif
(26)       FindVirtualFallingCheckers(0, CS, c · x, c · y, true, vAllFC, allFC.index + 1);
(27)   endfor
(28)   if isBreak == true then break; endif
(29) endfor
(30) return Dab;
(31) //Search the virtual falling checkers of specified checkers.
(32) Function FindVirtualFallingCheckers(d, CS, x, y, isFirstStep, vAllFC, index)
(33) begin
(34)   for i = 3 to −3 do
(35)     if i == −d then continue; endif
(36)     if i == 0 then continue; endif
(37)     Search the next jumping falling checker (x′, y′) in direction i of checker (x, y).
(38)     if (x′, y′) exists then
(39)       vAllFC[index].add(new Coordinate(x′, y′));
(40)       FindVirtualFallingCheckers(3, CS, x′, y′, false, vAllFC, index);
(41)     else if the piece on (x′, y′) is of own side then
(42)       FindVirtualFallingCheckers(3, CS, x′, y′, true, vAllFC, index + 1);
(43)     else
(44)       if isFirstStep == true then
(45)         vAllFC[index].add(new Coordinate(x′, y′));
(46)       endif
(47)     endif
(48)   endfor
(49) end
```

Algorithm 7: *PiecesDistanceCalculation*.

Table 5: Data records of the additional tests for selecting thresholds. A total of 1650 game-logs were used as the data source.

Gts	Ms (%)	Mf (%)	Ner	Nks	Anep	Auep (%)
15	10	5	113	160	4	1.47
15	8	4	275	323	11	1.84
15	6	3	386	421	15	1.86
15	4	2	460	512	19	1.95
15	**2**	**1**	**528**	**575**	**26**	**2.36**
15	1	0.5	582	649	26	2.11
15	0.5	0.25	638	691	27	2.03
15	0.25	0.125	690	743	28	1.95

Note: Gts: game times; Ms: minSupport; Mf: minFreq; Ner: number of Experience Rules; Nks: number of Key States; Anep: average number of experiential patterns used in each game; and Auep: average usage of experiential patterns per game.

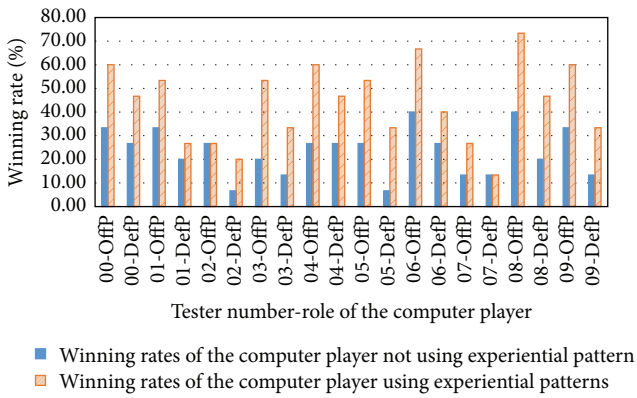

Figure 6: Comparison of the winning rates of the computer player.

is illustrated in Figure 6. In this test, the value of ATMC was 0.93 s.

Descriptions of Effects. (a) Figure 6 shows that, by using experiential patterns, the computer player has increased its winning rates in most of the games at different degrees; (b) the average number of the rounds per game has increased; (c) the value of ATMC has declined dramatically.

Discussions. (a) The effectiveness of experiential patterns will become more obvious with the increase of game-logs, and this will give the human players an impression that the computer player is learning from the experience; (b) in a few games such as 02-OffP and 07-DefP shown in Figure 6, although the computer player's winning rates are not increased, the average number of the rounds per game is greatly raised (see the lines with bold font shown in Table 6). The testers obviously feel the improvement of the computer player's chess skill; (c) if the Experience Rules exist, the computer player will match and use them first. Compared with TentativeMove, the pattern matching algorithm is of linear complexity and consumes less time (see Algorithm 6).

3.4. Test in Computer-Computer Mode. In this test, we set two computer players. One could use the experiential patterns and the other could not. They took turns to serve as OffP and played with each other for 30 games. The results are shown in Table 7.

Table 6: Data records of the test in human-computer mode with experiential patterns.

Tester	Gts	Role	Wts	Anrs	Wr (%)
00	15	OffP	9	51	60.00
00	15	DefP	7	51	46.67
01	15	OffP	8	52	53.33
01	15	DefP	4	54	26.67
02	**15**	**OffP**	**4**	**56**	**26.67**
02	15	DefP	3	50	20.00
03	15	OffP	8	51	53.33
03	15	DefP	5	49	33.33
04	15	OffP	9	54	60.00
04	15	DefP	7	52	46.67
05	15	OffP	8	52	53.33
05	15	DefP	5	48	33.33
06	15	OffP	10	56	66.67
06	15	DefP	6	50	40.00
07	15	OffP	4	48	26.67
07	**15**	**DefP**	**2**	**51**	**13.33**
08	15	OffP	11	52	73.33
08	15	DefP	7	52	46.67
09	15	OffP	9	52	60.00
09	15	DefP	5	50	33.33

Note: Gts: game times; Role: role of the computer player; Wts: winning times of the computer player; Anrs: average number of the rounds per game; and Wr: winning rate of the computer player.

Table 7: Data records of the test in computer-computer mode. The computer player using experience patterns is denoted by CP1 and the other by CP2.

Player	Wts	Anep	ATMC (s)	Wr (%)
CP1	56	26	0.91	93.33
CP2	4	0	5.77	6.67

Note: Wts: winning times; Anep: average number of experiential patterns used in each game; ATMC: value of ATMC; and Wr: winning rate.

Description of the Effect. The computer player using experience patterns has a great advantage in winning rate and ATMC.

Discussion. Experiential patterns can help the computer player in making more strategic moves; still, we are aware of four losing games of CP1 (as shown in Table 7). This indicates that some of the experiential patterns may not be really good. However, the invalid patterns will die out with the growth of game-logs.

3.5. Comparative Analysis on the Algorithms of Pattern Mining. A sequence database consists of sequences of ordered elements, and these elements contain some unordered items. The common objectives of the previous works [27–31] are

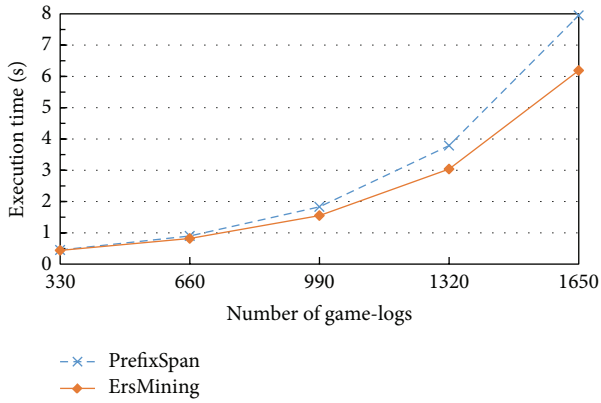

Figure 7: Performance of the two algorithms on a given data set. *ErsMining* denotes the set of algorithms, including *SequenceTreeCreation*, *BranchShifting*, *CBSSFinding*, and *ReliabilitySelection*, for mining Experience Rules.

to find the interesting relations between these items. In this work, however, the pattern mining algorithms aim at finding only the frequent CBSSs in the game-logs. This can be considered as a special case of the sequential pattern mining on a sequence database. For ease of comparison, we implemented the traditional algorithm *PrefixSpan* and added some restraints to make it fit for mining in our game-logs. These restraints were (a) setting length of each subsequence to 2 and (b) setting size of each element to 1. The comparative results are shown in Figure 7.

Here the minSupport was set to 2%, and the given data set contained 1,650 sequences (i.e., game-logs), 155,100 elements (i.e., pieces-states). Figure 7 indicates that our algorithms compared with *PrefixSpan* are more efficient.

3.6. Discussion on the Using of Game-Logs. Previous works [17–24] mainly discuss the methods of extracting expert knowledge from game records. The extracted knowledge is commonly used to construct a game agent of high intelligence. Differently, our work aims at helping a computer player in learning from growing game-logs, acquiring human's experience, and gradually becoming more skillful. It is a practical approach to make the game more fascinating. Furthermore, by satisfying the two conditions mentioned at the beginning of Section 2, any board game can utilize this approach after a minor adjustment. And this shows the extensibility of our approach.

4. Conclusions

In this paper, a novel approach is proposed to mine experiential patterns from the game-logs of board game. Those experiential patterns can be utilized to improve the intelligence of a computer player. This approach makes computer players learn from human's experience progressively during gaming and become more experienced with the increase of game-logs. This will make the game more interesting. We conducted an experiment on our designed test platform of

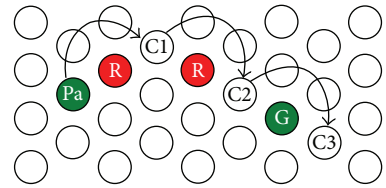

Figure 8: Single-piece-jump. Pa can jump to C1, C2, and C3.

Chinese checkers game, and the results demonstrate that our approach is effective, efficient, and extensible.

Nevertheless, our approach still needs improvements from several aspects, such as the following:

(i) designing algorithms to automatically adjust the parameters *minSupport* and *minFreq*;

(ii) optimizing the related algorithms to improve their access performance for a huge sequence-tree;

(iii) researching the fast matching problem in massive experiential patterns;

(iv) developing the approach for online analysis of game-logs;

(v) looking for more experiential patterns making games interesting.

Appendix

(1) Rules and Data Structures. The rules and data structures used for programming are described as follows.

(i) Only use the 2-player mode.

(ii) Move rules: there are two move rules, shift and jump. The shift means moving a single piece one step in any direction to an adjacent empty checker. The jump here means single-piece-jump, which is the simplest and most usual rule. An example of single-piece-jump is shown in Figure 8.

(iii) Checker: a computer can get the complete appearance of a game by scanning all the checkers on the chessboard. A checker has three properties including *color*, *position*, and *status*. The *color* stores a hexadecimal number to represent the color of the piece on the checker. The *position* stores the coordinate of the checker. The *status* uses the values of 1, −1, and 0 to describe who is on the checker.

(iv) Chessboard: for the convenience of programming, the chessboard is designed to fit for 2-player mode. That means the other four useless star corners on traditional chessboard are cut out (as illustrated in Figure 9).

We used a coordinate system to locate the positions on the chessboard. This coordinate system is unfixed and has an included angle of 60°. Players have their own frames of axes with the same form (as shown in Figure 10). This design has two advantages. First,

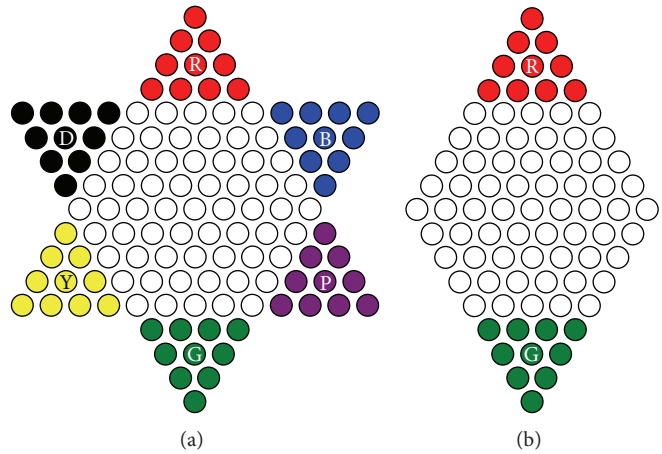

Figure 9: Chessboard. (a) The traditional chessboard. (b) The simplified chessboard without the useless star corners.

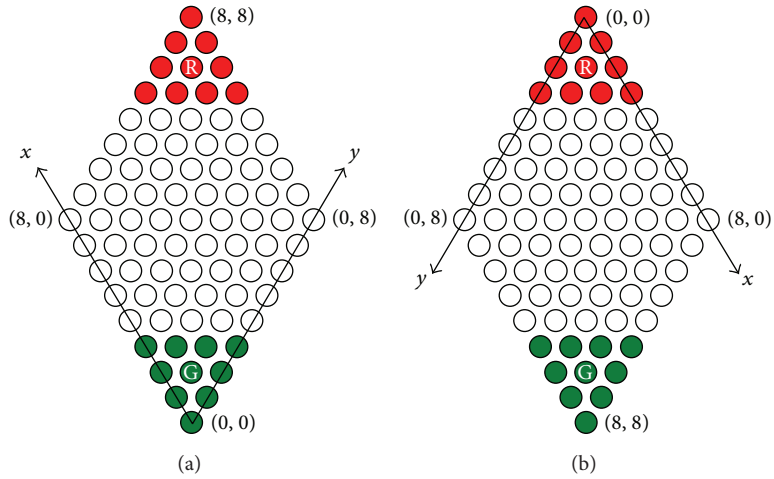

Figure 10: Unfixed coordinate system. The frames of axes in (a) and (b) are used by green and red player, respectively.

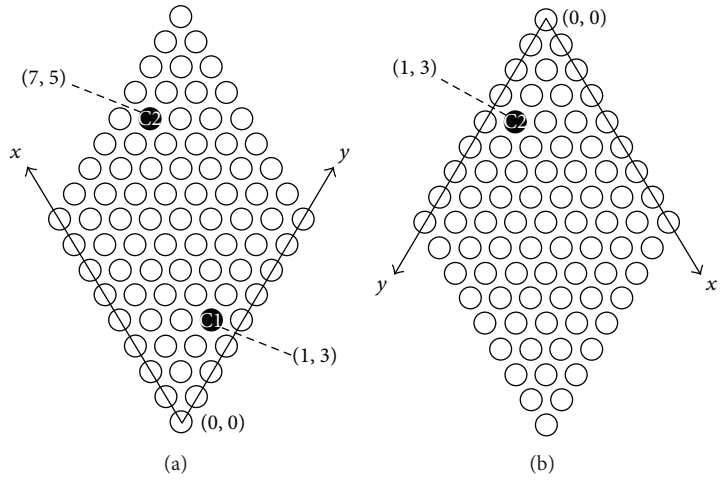

Figure 11: Every checker has two coordinates due to different frames of axes. For example, checkers C1 in (a) and C2 in (b) have the same coordinate. The two coordinates of checker C2 are (7, 5) and (1, 3), respectively.

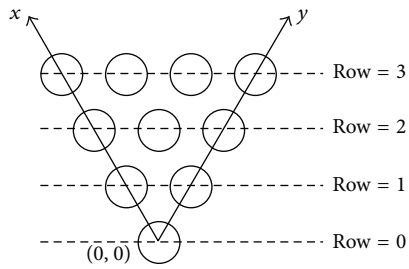

Figure 12: Parameter row.

the space of the coordinate system is utilized sufficiently. Secondly, the coordinate discrepancy between checkers is eliminated naturally.

Unavoidably, the unfixed coordinate system causes a checker to have dual coordinates (as shown in Figure 11).

To solve this problem, it is necessary to consider one of the two frames of axes as the reference, which is called normal frame of axes (NFoA). The other one called reverse frame of axes (RFoA) can be converted to the NFoA with

$$(x, y)\begin{cases} x = 8 - x' \\ y = 8 - y'. \end{cases} \tag{A.1}$$

In formula (A.1), (x, y) and (x', y') are the coordinates in NFoA and RFoA, respectively.

(v) Row: the *row* means the vertical distance from current checker to the origin of the coordinate system (as illustrated in Figure 12). For checker $C(x, y)$, the following equation holds:

$$row = C \cdot x + C \cdot y. \tag{A.2}$$

(vi) FD: the *FD* represents the distance moved forward by a piece. It can be calculated by

$$FD = rowFC - rowSC, \tag{A.3}$$

where *rowFC* is the row of the falling checker and *rowSC* is the row of the starting checker. For example, if a player moves a piece from (1, 3) to (5, 6) in one step, then the *FD* of this move is 7 (as illustrated in Figure 13). Note that the *FD* is negative when moving a piece backward.

(vii) Checkers-state: the checkers-state is an integer two-dimensional array. The subscripts and values of the checkers-state represent the coordinates and statuses of the corresponding checkers, respectively.

(viii) Piece: it is time consuming to scan all the 81 checkers at runtime. But on the contrary, there are only 20 pieces in a game. Scanning these pieces is efficient. The piece has the same properties as the checker.

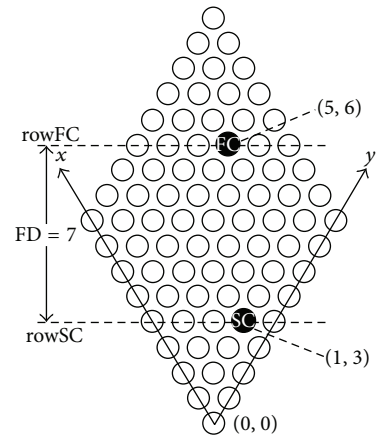

Figure 13: Parameter FD.

Active side										Passive side									
0	1	2	3	4	5	6	7	8	9	9	8	7	6	5	4	3	2	1	0
A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Figure 14: Logical expression of the pieces-state.

(ix) Pieces-state: a pieces-state is a special set consisting of two subsets; one is active side and the other is passive side. Each subset has ten elements representing the coordinates of pieces as illustrated in Figure 14. In active side, the coordinates belong to the pieces of the player whose last move has led to this pieces-state. These pieces are sorted by their row values in ascending order (sorted by y-axis when they are in the same row). The situation in passive side is just the opposite.

The pieces-state can be formally defined as follows:

$$Pieces\text{-}state = \{\text{SAS}, \text{SPS}\}$$

$$\begin{aligned} &\text{SAS} \\ &= \{A_n \mid A_n \text{ is the coordinate of piece } n \text{ in active side}, \\ &\quad n \in [0, 9]\} \\ &\text{SPS} \\ &= \{P_n \mid P_n \text{ is the coordinate of piece } n \text{ in passive side}, \\ &\quad n \in [0, 9]\}. \end{aligned} \tag{A.4}$$

By using formula (A.1), a pieces-state in RFoA can be easily transposed to that in NFoA.

(2) Gameplay. A set of computer programs is designed to control the gameplay and save the game-logs automatically. For different purposes, the game provides three play modes described as follows.

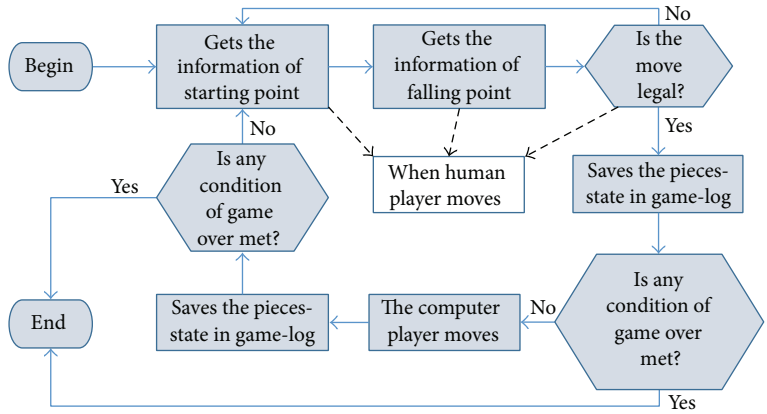

Figure 15: Gameplay of the human-human mode.

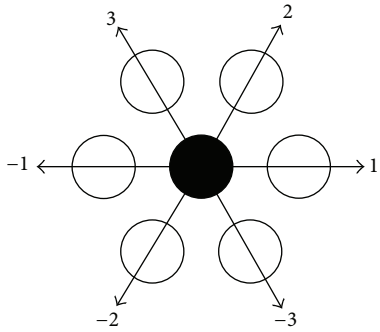

Figure 16: Searching directions.

(i) Human-human mode: this mode is mainly used to collect the experience from humans. All the players in this mode are real persons.

(ii) Computer-computer mode: this mode is only used for carrying out experiments. All the players in this mode are controlled by the computer programs.

(iii) Human-computer mode: in this mode, a human player is allowed to play against the computer player, and the game-logs are also collected. This mode is in charge of effect analysis.

Taking the human-human mode as an example, the gameplay is shown in Figure 15. And the other two play modes can be designed in a similar way.

In the gameplay, there are three problems to be solved.

Problem 1. It is to find out all the falling checkers related to the specified starting checker.

Problem 2. It is to determine the legality of a move.

Problem 3. It is to make the computer player have the ability to move spontaneously.

The solution to Problem 1 is described in Algorithm 8.

In Algorithm 8, d denotes the searching direction. According to the game rules, a piece can be moved in six surrounding directions. So, the term searching direction is defined (as illustrated in Figure 16).

Table 8: The calculations in different directions.

Direction ID	Coordinate of current position	Coordinate of the next position
3	(x, y)	$(x + 1, y)$
2	(x, y)	$(x, y + 1)$
1	(x, y)	$(x - 1, y + 1)$
−1	(x, y)	$(x + 1, y - 1)$
−2	(x, y)	$(x, y - 1)$
−3	(x, y)	$(x - 1, y)$

The features of searching direction are listed below.

(i) There is no direction 0.

(ii) The IDs of opposite directions can be changed to each other by picking minus.

(iii) The checkers in a certain direction can be found by adjusting the coordinates. Table 8 shows the calculations in every direction.

According to the three features, all checkers can be traversed recursively.

The solution to Problem 2 is described in Algorithm 9.

To solve Problem 3, we provide an algorithm based on the traditional alpha-beta pruning method (see Algorithm 10).

In Algorithm 10, the evaluation function is formally defined as follows:

$$\text{evaf}(FD, CU, FC\_Incre), \tag{A.5}$$

where CU denotes the Checker Usage of the falling checker and FC_Incre denotes the number of increased legal falling checkers arising out of a move. Negative FC_Incre means that the number of falling checkers has decreased. Then, a simple evaluation function is given below:

$$\text{eva} = (FD + FC\_Incre) * (1 + CU). \tag{A.6}$$

```
Procedure FindFallingCheckers
Input:
  d, the direction that will be searched in.
  PS, the specified pieces-state.
  x, the x-coordinate of starting checker.
  y, the y-coordinate of starting checker.
  isFS, whether it is the first step.
  allFC: the array for collecting falling checkers.
Output: Updated allFC.
Method:
(01) for i = 3 to −3 do
(02)   //Don't search the repeated falling checker.
(03)   if i == −d then continue; endif
(04)   //Don't search the direction 0.
(05)   if i == 0 then continue; endif
(06)   Search the next jumping falling checker (x′, y′) in direction i of checker (x, y).
(07)   if (x′, y′) exists then
(08)     allFC.add(new Coordinate(x′, y′));
(09)     FindFallingCheckers(3, PS, x′, y′, allFC, false);
(10)   else
(11)     if isFS == true then
(12)       allFC.add(new Coordinate(x′, y′));
(13)     endif
(14)   endif
(15) endfor
(16) return allFC;
```

Algorithm 8: *FindFallingCheckers*.

```
Procedure LegalityDetermination
Input:
  PS, the initial pieces-state.
  cSC: the coordinate of starting checker.
  cFC: the coordinate of falling checker.
Output:
  True means legal, and false not legal.
  Updated PS.
Method:
(01) let CS be the checkers-state;
(02) for i = 0 to 9 do
(03) //Initialize the checkers-state from pieces-state.
(04)   CS[PS·SAS[i]·x][PS·SAS[i]·y] = 1;
(05)   CS[PS·SPS[i]·x][PS·SPS[i]·y] = −1;
(06) endfor
(07) FindFallingCheckers(0, CS, cSC·x, cSC·y, allFC, true);
(08) for each Coordinate in allFC do
(09)   if cFC == Coordinate then
(10)     Update PS with cSC and cFC;
(11)     return true;
(12)   endif
(13) endfor
(14) return false;
```

Algorithm 9: *LegalityDetermination*.

```
Procedure TentativeMove
Input:
  steps, the steps will be searched.
  PS: the initial pieces-state.
  isFL: whether it is the first level of recursion.
Output:
  maxEva: the maximum value of evaluation.
Method:
(01) let allFC be the set of falling checkers' coordinates;
(02) let cSC be the coordinate of starting checker;
(03) let cFC be the coordinate of falling checker;
(04) let CS be the checkers-state;
(05) let the elements of CS be equal to 0;
(06) for i = 0 to 9 do
(07)   //Obtain the checkers-state from pieces-state.
(08)   CS[PS·SAS[i]·x][PS·SAS[i]·y] = 1;
(09)   CS[PS·SPS[i]·x][PS·SPS[i]·y] = −1;
(10) endfor
(11) maxEva = −Infinity;
(12) steps - -;
(13) for i = 0 to 9 do
(14)   //Evaluate all moves by Depth First Search.
(15)   copyCS = clone(CS);
(16)   FindFallingCheckers(0, copyCS, PS·SAS[i]·x, PS·SAS[i]·y, allFC, true);
(17)   for each FC in allFC do
(18)     FD = (FC·x + FC·y) − (PS·SAS[i]·x + PS·SAS[i]·y);
(19)     Calculate FC_Incre;
(20)     Query CU;
(21)     thisEva = (FD + FC_Incre) ∗ (1 + CU);
(22)     if steps > 0 and isFL == true then
(23)       copyPS = clone(PS);
(24)       Update copyPS with PS·SAS[i] and FC.
(25)       PiecesStateTransposition(copyPS);
(26)       thisEva = thisEva − TentativeMove(Steps, copyPS, false);
(27)     endif
(28)     if thisEva > maxEva then
(29)       maxEva = thisEva;
(30)       cSC = PS·SAS[i];
(31)       cFC = FC;
(32)     endif
(33)   endfor
(34) endfor
(35) if isFL == true then
(36)   Update PS with cSC and cFC;
(37) else if steps > 0 then
(38)   copyPS = clone(PS);
(39)   Update copyPS with PS·SAS[i] and FC;
(40)   PiecesStateTransposition(copyPS);
(41)   maxEva = maxEva − TentativeMove (Steps, copyPS, false);
(42) endif
(43) return maxEva;
```

Algorithm 10: *TentativeMove*.

Formula (A.6) is based on the following reasons:

the longer the forward distance, the better the move;

the more the generated falling checkers, the better the move;

the higher the Checker Usage of the falling checker, the better the move.

To reduce the error of the evaluation function and ensure its execution efficiency, an appropriate value of *steps* is given. During these steps, a computer player alternately enacts

both sides' players to move, and the evaluation values of every move are recorded. Then, the algorithm calculates the difference of the sum of evaluation values between its own side and the opponent's and selects the move with the maximum difference. Through testing, we set *steps* to 11 and obtained effect preferably.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported by the Scientific Research Foundation of Hebei Education Department (Grant no. Z2012147), the Natural Science Foundation of Hebei Province (Grant no. F2014201100), and the Soft Science Research Program of Hebei Province (Grant no. 14450318D).

References

[1] A. El Rhalibi, K. W. Wong, and M. Price, "Artificial intelligence for computer games," *International Journal of Computer Games Technology*, vol. 2009, Article ID 251652, 3 pages, 2009.

[2] C. E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, no. 7, pp. 256–275, 1950.

[3] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 49, pp. 433–460, 1950.

[4] A. L. Samuel, "Some studies in machine learning using the game of checkers," *International Business Machines Corporation*, vol. 3, pp. 211–229, 1959.

[5] A. L. Samuel, "Some studies in machine learning using the game of checkers II—recent progress," *IBM Journal of Research and Development*, no. 11, pp. 601–617, 1967.

[6] T. H. Kjeldsen, "John von Neumann's conception of the minimax theorem: a journey through different mathematical contexts," *Archive for History of Exact Sciences*, vol. 56, no. 1, pp. 39–68, 2001.

[7] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," vol. 6, no. 4, pp. 293–326, 1975.

[8] J. R. Slagle and R. C. T. Lee, "Application of game tree searching techniques to sequential pattern recognition," *Communications of the ACM*, vol. 14, no. 2, pp. 103–110, 1971.

[9] J. Pearl, "Asymptotic properties of minimax trees and game-searching procedures," *Artificial Intelligence*, vol. 14, no. 2, pp. 113–138, 1980.

[10] R. L. Rivest, "Game tree searching by min/max approximation," *Artificial Intelligence*, vol. 34, no. 1, pp. 77–96, 1987.

[11] L. W. Li and T. A. Marsland, "Probability-based game tree pruning," *Journal of Algorithms*, vol. 11, no. 1, pp. 27–43, 1990.

[12] M. Levene and T. I. Fenner, "The effect of mobility on minimaxing of game trees with random leaf values," *Artificial Intelligence*, vol. 130, no. 1, pp. 1–26, 2001.

[13] Y. Björnsson and T. A. Marsland, "Learning extension parameters in game-tree search," *Information Sciences*, vol. 154, no. 3-4, pp. 95–118, 2003.

[14] Wikipedia, English Draughts, 2014, http://en.wikipedia.org/wiki/English_draughts.

[15] W. Saletan, "Chess Bump: The Triumphant Teamwork of Humans and Computers," May 2007, http://www.slate.com/articles/health_and_science/human_nature/2007/05/ches-s_bump.html.

[16] Wikipedia, Logistello, 2014, http://en.wikipedia.org/wiki/Logistello.

[17] B. N. Chen, P. Liu, S. C. Hsu, and T. S. Hsu, "Abstracting knowledge from annotated Chinese-chess game records," in *Computers and Games*, vol. 4630 of *Lecture Notes in Computer Science*, pp. 100–111, Springer, Berlin, Germany, 2007.

[18] E. C. D. van der Werf, M. H. M. Winands, H. J. van den Herik, and J. W. H. M. Uiterwijk, "Learning to predict life and death from Go game records," *Information Sciences*, vol. 175, no. 4, pp. 258–272, 2005.

[19] Z.-Q. Liu and Q. Dou, "Automatic pattern acquisition from game records in GO," *The Journal of China Universities of Posts and Telecommunications*, vol. 14, no. 1, pp. 100–105, 2007.

[20] S. J. Yen, T. N. Yang, J. C. Chen, and S. C. Hsu, "Pattern matching in GO game records," in *Proceedings of the 2nd International Conference on Innovative Computing, Information and Control (ICICIC '07)*, p. 297, Kumamoto, Japan, September 2007.

[21] T. Esaki and T. Hashiyama, "Extracting human players' shogi game strategies from game records using growing SOM," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '08)*, pp. 2176–2181, IEEE, Hong Kong, June 2008.

[22] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG '09)*, pp. 140–147, IEEE, Milano, Italy, September 2009.

[23] S. Takeuchi, T. Kaneko, and K. Yamaguchi, "Evaluation of game tree search methods by game records," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 288–302, 2010.

[24] S. Wender, *Data Mining and Machine Learning with Computer Game Logs*, Project Report, University of Auckland, Auckland, New Zealand, 2007.

[25] Wikipedia, "Chinese checkers," August 2014, http://en.wikipedia.org/wiki/Chinese_checkers.

[26] N. R. Mabroukeh and C. I. Ezeife, "A taxonomy of sequential pattern mining algorithms," *ACM Computing Surveys*, vol. 43, no. 1, article 3, 2010.

[27] R. Srikant and R. Agrawal, "Mining sequential patterns: generalizations and performance improvements," in *Advances in Database Technology—EDBT '96*, vol. 1057 of *Lecture Notes in Computer Science*, pp. 1–17, Springer, Berlin, Germany, 1996.

[28] M. J. Zaki, "SPADE: an efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, no. 1-2, pp. 31–60, 2001.

[29] X. Yan, J. Han, and R. Afshar, "CloSpan: mining closed sequential patterns in large databases," in *Proceedings of the 3rd SIAM International Conference on Data Mining*, pp. 166–173, SIAM, San Francisco, Calif, USA, 2003.

[30] J. Pei, J. Han, B. Mortazavi-Asl et al., "Mining sequential patterns by pattern-growth: the prefixspan approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[31] M. Y. Lin and S. Y. Lee, "Fast discovery of sequential patterns by memory indexing," in *Data Warehousing and Knowledge Discovery*, vol. 2454 of *Lecture Notes in Computer Science*, pp. 150–160, Springer, Berlin, Germany, 2002.

[32] R.-F. Hu, L. Wang, X.-Q. Mei, and Y. Luo, "Fault diagnosis based on sequential pattern mining," *Computer Integrated Manufacturing Systems*, vol. 16, no. 7, pp. 1412–1418, 2010.

[33] G. Yilmaz, B. Y. Badur, and S. Mardikyan, "Development of a constraint based sequential pattern mining tool," *The International Review on Computers and Software*, vol. 6, no. 2, pp. 191–198, 2011.

[34] S. Dharani, J. Rabi, N. Kumar, and Darly, "Fast algorithms for discovering sequential patterns in massive datasets," *Journal of Computer Science*, vol. 7, no. 9, pp. 1325–1329, 2011.

[35] H.-J. Shyur, C. Jou, and K. Chang, "A data mining approach to discovering reliable sequential patterns," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2196–2203, 2013.

[36] G.-C. Lan, T.-P. Hong, V. S. Tseng, and S.-L. Wang, "Applying the maximum utility measure in high utility sequential pattern mining," *Expert Systems with Applications*, vol. 41, no. 11, pp. 5071–5081, 2014.

[37] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000.

[38] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965.