AUTOMATED IDENTIFICATION OF PATTERNS IN EVALUATION FUNCTIONS

T. Kaneko, K. Yamaguchi, S. Kawai
Graduate School of Arts and Sciences (Kaneko, Kawai) and Information Technology Center (Yamaguchi), The University of Tokyo, Tokyo, Japan
{kaneko,yamaguch,kawai}@graco.c.u-tokyo.ac.jp, http://www.c.u-tokyo.ac.jp/~kaneko/

Abstract This paper proposes a general and automated method that generates accurate evaluation functions, without expert players' knowledge of a target game. Patterns (which are partial descriptions of a game state) are widely used as primitives of evaluation functions in game programming. They have to be carefully selected in order to generate accurate evaluation functions. Our approach consists of three steps: (1) generation of logic formulae by using the specifications of a target game, (2) translation of the formulae into patterns, and (3) selection of a set of suitable patterns from those generated. The problem, in the automated identification of suitable patterns, is that it is difficult either to generate only useful patterns or to examine all possible patterns. The latter obstacle is due to the prohibitive numbers involved. We solved this dilemma by a combination of two methods, where one method generates patterns of good quality, and the other method entails a lightweight selection based on statistics that could handle a large number of candidates. Experiments in Othello revealed that about 100,000 patterns from more than eight million automatically generated patterns could be successfully selected with our method, and that accurate evaluation functions were constructed. This accuracy is comparable to that of specialized Othello programs and is much better than that of the evaluation functions generated by existing general methods.

Keywords: feature generation, feature selection, evaluation function, Othello

1. General Game Players

One of the most ambitious goals of artificial-intelligence research is the development of a general game player that can learn and play an arbitrary instance of a certain class of game. Strong game programs must have an accurate and efficient evaluation function that can estimate the results of a game based on the notion *position*. Since an evaluation function is specific to a target game, the development of general game players requires evaluation functions to be automatically constructed without assistance of human experts.

1.1 Learning of Evaluation Functions

A popular way of constructing an evaluation function is to make it a (linear) combination of evaluation primitives called *features*, and adjust the parameters of the combination (Samuel, 1967; Tesauro, 1992; Buro, 2002). Generally, the construction of evaluation functions requires the acquisition of features, and the training of a prediction model (e.g., linear combination).

1.2 Learning of Features

The main difficulty in constructing evaluation functions is identifying appropriate features. In most preceding investigations, these features have been provided by human experts for the game involved.

Our first goal is to identify appropriate features mechanically. To achieve this we employed a method of constructing features written in logic programs (we called them *logical features*). However, logical features are not practical because they are too slow in evaluating logic programs. Yet, the advantage is that practical evaluation functions were constructed with a large number of patterns as features (Buro, 1998; Buro, 2002). A pattern is a logical formula in a specific form. We introduce a rigorous definition for this in Subsection 3.3. Even though a pattern is just a logical formula in a specific form, the mechanical identification of suitable pattern sets to derive a good evaluation function is a difficult task.

1.3 The Approach

Our second goal is to construct efficient and accurate evaluation functions through game-independent methods. Here we propose a combination of methods that yields patterns similar to Buro's (1998) methods by translation from logical features. These methods are:

1 generation of logical features,
2 extraction of patterns from logical features, and
3 selection of suitable patterns.

A large number of patterns are produced in steps 1 and 2, and useful patterns are selected in step 3. The claim of the paper is that this selection is indispensable for generating useful evaluation functions. The reason why we have to generate such a large number of patterns in steps 1 and 2 is that they are required to achieve accuracy in the evaluation functions constructed. There is no known method of generating only useful patterns.

The method of selection must be so lightweight that a machine can evaluate numerous pattern candidates within practical time limitations. We demonstrate the effectiveness of our solution through experiments.

The paper is organized as follows. Section 2 reviews related work and other issues that need to be resolved to construct general game players. Section 3 introduces the basic terminology. Methods to generate logical features and evaluate positions are briefly explained in Sections 4 and 5. In Section 6 a method of selection is proposed. Section 7 shows the experimental results in Othello. Section 8 concludes the paper.

2. Related Work

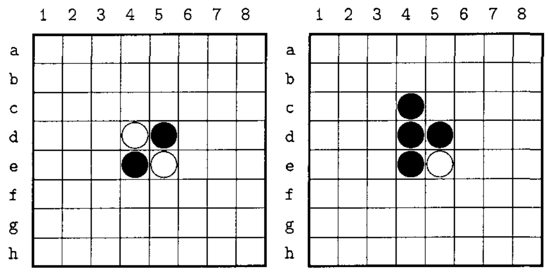
The construction of general game players requires the acquisition of game-specific search enhancements as well as evaluation functions, such as realization probabilities (Tsuruoka, Yokoyama, and Chikayama, 2002), opening books (Lincke, 2001), and endgame books. This paper only addresses evaluation functions, even though we are aware that our method can be applied to the acquisition of other knowledge.

In constructing evaluation functions, the training of prediction models requires unbiased training positions and an appropriate labeling (Buro, 1998). It is well known that the usefulness of learned evaluation functions depends on the training positions used. Thus, unbiased positions are needed to develop strong programs. Because this paper primarily focuses on the acquisition of features, the experiments were conducted on a game where both the training positions and the labeling were available (near endgame in Othello). In games where these are not available, we can apply methods of gathering positions via self-play and temporal-difference learning (Tesauro, 1992, 2002).

We simply use linear regression for prediction because we could use a method that iteratively adjusts the weights in a linear model, even when a very large number of features are used (Barrett et al., 1994). Other prediction models, such as neural networks, could be used with our method, too.

Logical features are general and were actually applied to many games. We mention Othello and a single-agent search problem by Fawcett (1993), symmetric chess-like games (Pell, 1993) and a variant of Shogi (Kaneko, Yamaguchi, and Kawai, 2002). However, the cost of evaluating positions is prohibitive when there are logical features due to the slow evaluation of logic programs, despite the recent efforts that have increased speeds more than 4,000 times (Kaneko, Yamaguchi, and Kawai, 2000, 2001).

Buro (2002) used patterns in fixed shapes. This is effective in achieving highly efficient pattern matching, even when a large number of patterns is involved. However, there is no established method of identifying effective shapes mechanically, and we do not know whether patterns in such fixed shapes are useful in other games. Kojima, Ueda, and Nagano's (1997) method acquired patterns from game records in Go through genetic programming. This requires



owns(d5, x). owns(e4, x).
owns(d4, o). owns(e5, o).

blank(a1). blank(a2). ...

owns(c4, x), owns(d4, x).
owns(d5, x). owns(e4, x).
owns(e5, o).
blank(a1). blank(a2). ...

Figure 1. Othello initial position (left) and position after Black has played c4 (right). Facts below each board define the position shown above.

game-specific adaptation to apply it to other games because it depends on the importance of adjacent stones.

We recently developed a method of generating patterns from logical features (Kaneko et al., 2001). However, the accuracies of the generated evaluation functions did not reach those that Buro obtained. This is because we only used about 4,000 patterns, while Buro used about 200,000. For our method it was impossible to provide a sufficient number of useful patterns because effective methods of selection were up to then unknown.

The selection of features is a central research topic in artificial intelligence, and many methods have been developed (Guyon and Elisseeff, 2003; Jain, Duin, and Mao, 2000). It is a combinatorial optimization problem. Heuristics are essential because the computational costs identifying an optimal pattern subset are known to be exponential in terms of the number of candidates (Jain, Duin, and Mao, 2000). Such costs are not acceptable. Moreover, popular selection methods such as the F-test in statistics cannot be used here. To illustrate this difficulty, we used about eight million candidates in the experiments that will be described later. Obviously, their covariances cannot be stored on normal computers.

3. Basic Terminology

This section introduces the basic terminology, including the specifications of a game written in logic (Subsection 3.1), the logic features (Subsection 3.2) and the definition of patterns (Subsection 3.3).

3.1 Positions and Domain Theory

A *position* is an intermediate status of a game. It is described by a set of special facts. A fact is a clause without a body. In Othello, `owns` and `blank`

```
legal_move(S, Player):-square(S), bs(S,_End,Player).
bs(S1,S3,P):-blank(S1), opponent(P,Opp),
  neighbour(S1,D,S2), span(S2,S3,D,Opp),
  neighbour(S3,D,S4), owns(P,S4).
span(S1,S2,D,Owner):-
  square(S1),square(S2), player(Owner), owns(Owner, S1),
  neighbour(S1,D,S3),span(S3,S2,D,Owner).
span(S,S,D,Owner):-
  square(S),player(Owner),owns(Owner,S), direction(D).
line(S,S,D):-square(S),direction(D).
line(From,To,D):-neighbour(From,D,Next), line(Next,To,D).
opponent(x, o). opponent(o, x).
direction(n).direction(ne).direction(e).direction(se).
direction(s).direction(sw).direction(w).direction(nw).
square(a1). square(a2). square(a3). (···)
square(d2). square(d3). square(d4). (···)
neighbour(a1, s, a2). neighbour(a2, n, a1).
neighbour(a2, s, a3). neighbour(a3, n, a2). (···)
neighbour(c4,ne,d3). neighbour(d3,sw,c4). (···)
```

Figure 2. Sample domain theory for Othello.

are used to represent a position. To demonstrate this, we have shown the facts defined in the initial position in Othello and the position after Black has played c4 in Figure 1. Here, Black is denoted by x, and White is denoted by o. In the initial position, owns(d5,x), owns(e4,x), owns(d4,o), and owns(e5,o) are defined for squares with a disc, and blank is defined for each empty square.

The main part of the specifications of a game consists of the rules of the game and the goal conditions. This is called *domain theory* and described by a set of Horn Clauses. The example Othello domain theory in Figure 2 is used throughout this paper.

3.2 Logical Features

Logical features are defined as Horn Clauses of the predicate logic where predicates in their body are defined by domain theory or position. The following clause is an example of a logical feature.[1]

f(A):-owns(x,A). % pieces for Black

[1]This is written as "f(N) :- count([A], (owns(x,A)), N)" in Fawcett (1993). In this paper, "count" has been assumed to be the default semantics of logical features and has therefore been omitted.

The *value* of a logical feature for a state is defined as the number of solutions, where solutions are the bindings of such constants to variables that make the clause true. In the above feature, `A` is a variable, and the solutions in the initial position in Figure 1 (left) are `d5` and `e4` (two solutions), which is the number of squares currently owned by Black.

3.3 Patterns

A pattern is defined as a conjunction of facts describing a part of a position. The value of a pattern is 0 or 1 according to its Boolean value; in a given position this value of a fact is 1 if it is defined (or 0 if undefined). For example, the following is a pattern.

$$\texttt{blank(a1)} \wedge \texttt{owns(x,a2)} \wedge \texttt{owns(o,a3)}$$

This pattern is a logical formula for "White can play on square a1."

4. Pattern Generation

Patterns are generated through the following steps:

1 generation of logical features with Fawcett's (1993) method,
2 translation of logical features into propositional logic by unfolding, and
3 extraction of patterns from propositional logic.

First, logical features are generated by means of syntactic translation of Horn Clauses, which are extracted from the domain theory of a target game. For example, the following feature (called a mobility feature) can be generated.

```
f(A):-legal_move(A,o).% mobility for White
```

Complex features can be generated by taking the preconditions of existing features. Fawcett (1993) has more details on automated construction.

In the next step, generated features are translated into propositional logic by *unfolding*. This is a technique in partial evaluation of logic programming (Bossi, Cocco, and Dullie, 1990), and is repeatedly applied until features only consist of ground facts. In conventional games with reasonable rules, it is easy to write a domain theory so that the unfolding of generated features stops even if they contain recursively defined clauses, due to the finiteness of the number of squares and satisfiable terms. Detailed translation methods have been described by Kaneko et al. (2001). The following clauses are part of the results we obtained for the unfolding of the feature in the above example.

```
legal_move(a1,o) :- blank(a1), owns(x,a2), owns(o,a3).
legal_move(a1,o) :- blank(a1), owns(x,b1), owns(o,c1).
legal_move(a1,o) :- blank(a1), owns(x,b2), owns(o,c3).
```

Finally, we extracted patterns from the unfolded features simply by taking the conjunctive part of their propositional formulae. The following formulae are patterns extracted from the unfolded features listed above.

- `blank(a1)` $\wedge$ `owns(x,a2)` $\wedge$ `owns(o,a3)`
- `blank(a1)` $\wedge$ `owns(x,b1)` $\wedge$ `owns(o,c1)`
- `blank(a1)` $\wedge$ `owns(x,b2)` $\wedge$ `owns(o,c3)`

Each pattern has a corresponding clause whose body (right hand of clause) is equivalent to the pattern.

5. Pattern Matching

Below, we briefly discuss a pattern matching method to justify the selection method of the next section. The purpose of the selection is to identify sets of patterns that produce efficient and accurate evaluation functions, where their efficiency depends on how the patterns are evaluated. Basic ideas in efficient matching are (1) performing incremental calculations and (2) utilizing a partial order on patterns.

5.1 Incremental Matching with a Diagram

Incremental matching was efficiently implemented with a Hasse diagram (Gries and Schneider, 1993) on the partial order of patterns, as outlined in Figure 3. Let each a, b, and c be a fact describing a position (such as `blank(a1)`), and consider that there are six patterns $\{abc$, ab, bc, a, b, and $c\}$. Here, ab means the conjunction of a and b. In the figure, a pattern is denoted by a square, and a fact is denoted by a circle. For each pattern, the question whether matching is required can quickly be determined by using the diagram. For example, matching of pattern 'abc' is only required when the value of pattern 'ab' or 'bc' changes.

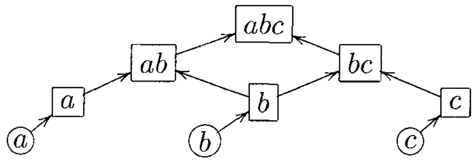


Figure 3. Hasse diagram of sample patterns.

The computational costs of incremental matching can be estimated by the number of nodes visited. Because each edge will be visited once at most, the cost for the worst case is proportional to the number of edges. Cube extraction (Rudell, 1996) was applied to a diagram here to reduce edges, as well as other optimizations. Details are discussed in Kaneko et al. (2001).

5.2 Counters for Matching

To speed up matching of individual patterns, an integer counter $cur(p)$ was associated with each pattern p such that the matching was determined by integer comparison instead of naively computing the logical conjunction of each fact in the pattern.

Let $dep(p)$ ($upd(p)$) be children (parents) of pattern p in a diagram. Counter $cur(p)$ is defined as the number of children of p whose current value is true. Then, as long as $cur(p)$ is properly maintained, the Boolean value of $cur(p) = |dep(p)|$ coincides with the value of pattern p.

6. Pattern Selection

This section introduces a lightweight selection method, which consists of two methods that are computationally inexpensive. These are:

- *preliminary filtering* by using the frequency of patterns, and
- *approximated forward selection* by assessing the contribution of patterns to the accuracy of a prediction model.

The latter method takes into account the accuracy of a linear model that uses selected patterns. Consequently, it requires that the model is trained (by weight fitting) for each subset of patterns; thus the method is relatively expensive. The former method is more efficient because it only uses the frequency of each pattern. However, it cannot be used to select useful patterns by itself. Hence, we first need to filter the candidates with the former method, and then select useful patterns with the latter method, to reduce its weight-fitting time.

6.1 Preliminary Filtering by Frequency

First, useless patterns are heuristically determined and filtered out by analysing their frequency, before approximated forward selection is done in the next step. There are two background considerations: (1) if low-frequency patterns are used, the efficiency of evaluating positions by using the method detailed in Section 5 will improve,[2] and (2) the use of extremely low-frequency patterns tends to cause over-fitting. We claim that high- or extremely low-frequency patterns can safely be rejected without a loss of quality in the generated evaluation functions. Although this may seem similar to the filtering in existing work (e.g.,

[2]More precisely, it is better to measure the frequency at which the patterns change from one position to another to improve search efficiency. We used the frequency of the patterns themselves, because this could be measured more easily. Moreover, to reduce the computational costs of weight fitting, reducing the frequency of the patterns themselves is also essential, as discussed in Subsection 7.2.2.

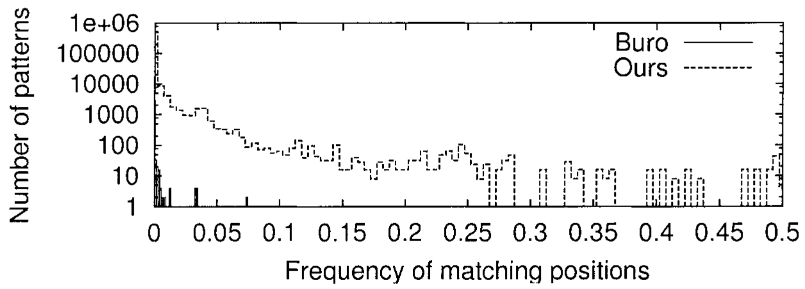


Figure 4. Histogram for frequency of matching positions.

Kojima et al., 1997; Buro, 1998), our approach is different in the sense that our colleagues do not explicitly reject high-frequency patterns as we do.[3]

We measured the frequency of part of Buro's (2002) horizontal patterns to estimate an appropriate frequency range (Figure 4). Because the highest frequency in Buro's patterns was 0.075, we expected that good evaluation functions could be generated with only patterns with a frequency below this value. Figure 4 also has the results of measurement for our patterns. It can be seen that many patterns can be filtered out by frequency. The preferable frequency ranges were determined by the experiments, which are discussed in Section 7.

6.2 Approximated Forward Selection

After filtering, we applied a method of statistically selecting explanatory variables, by treating a pattern as a binary variable. Approximated sequential forward selection was adopted from many existing methods (Guyon and Elisseeff, 2003). It is so efficient that it was used already for manual computation before computers became widely available (Okuno et al., 1981).

The algorithm is listed in Figure 5. It is used to select a subset of variables (S) that are effective in predicting a target variable (y_0), from a set of candidates (i.e., patterns, X). A target variable is the difference between the number of black and white discs (explained below in the experiments).

One pattern (x_{α_i}) is added to the selected set (S) at the seventh line for each loop, as in sequential forward selection. A priority function, also explained later, is used to select a pattern. Let n be the number of candidates and m be the number of patterns finally selected. Because variables in S are never removed, the method tries m subsets of candidates, which is far less than the possible number of subsets, 2^n.

[3]Kojima et al.'s (1997) method and the inductive algorithm proposed by Buro (1998), which was not used in preparing the evaluation functions for LOGISTELLO, tend to discard high-frequency patterns because they prefer specific patterns in matching. As patterns for specific given shapes contained at least eight squares, high-frequency patterns were not used in constructing evaluation functions for LOGISTELLO.

```
// (input)   $y_0$: a target variable,
//           $X = \{x_0, x_1, \ldots, x_p\}$: a set of explanatory variables
// (output)  $S$: a set of selected variables
//           $y_{i+1}$: residuals after selection
$i \leftarrow 0, S \leftarrow \emptyset$
while (termination criterion is not satisfied)
    pick $x_{\alpha_i}$ of the highest priority
    compute $a_{\alpha_i}, b_{\alpha_i}$ by univariate regression s.t. $a_{\alpha_i} x_{\alpha_i} + b_{\alpha_i}$ predicts $y_i$
    $y_{i+1} \leftarrow y_i - (a_{\alpha_i} x_{\alpha_i} + b_{\alpha_i})$ // residuals
    $S \leftarrow S \cup \{x_{\alpha_i}\}$
    $i \leftarrow i + 1$
```

Figure 5. Approximated forward selection algorithm.

A priority function is used to estimate the usefulness of the pattern for selection at the seventh line, and this should be carefully adopted taking the purpose of selection into consideration. For practical game programming, efficiency and accuracy should be taken into account to estimate the usefulness of a pattern in terms of priority. In this paper, we used the correlation with residuals after the i-th regression y_i as a priority function to achieve accuracy.

Here, if explanatory variables have no correlation with one another, variables selected with this method are equivalent to the ones selected by normal sequential forward selection, where the multiple regression coefficient in predicting y_0 using all variables in S is used as the priority function.[4] The order of candidates affects the results in other cases (Okuno et al., 1981). However, this method is more efficient than sequential forward selection because it uses univariate regression instead of multivariate regression.

```
// (input)   $y'_0$: be a target variable
//           $X_0, X_1, \ldots X_n$: sets of variables
// (output)  $R$: selected variables
$R \leftarrow \emptyset$
for each $i$ in $0, \ldots, n$
    $(S, y'_{i+1}) \leftarrow$ approximated forward selection$(y'_i, X_i)$
    $R \leftarrow R \cup S$
```

Figure 6. Iterative selection algorithm.

The improved computation applied so far leads to appropriate results. Yet, the most expensive computation is to determine the priority (i.e., correlation) of each pattern in each loop. Naively, it requires pattern matching over all training positions for every loop, but then the computational costs are unacceptable. The

[4]The method approximates sequential forward selection by using the accumulation of univariate regressions instead of multivariate regression.

priority of each pattern can be incrementally updated by means of a table holding the number of pattern co-occurrences if there are not too many candidates.

Thus, to avoid frequent pattern matching, patterns were split into sets of a moderate number of patterns $\{X_0, X_1, \ldots X_n\}$ in advance, and approximated forward selection was iteratively applied to each X_i in turn, as shown in Figure 6. A test to determine whether the priority of a selected pattern went beyond a given threshold worked well as a termination criterion in each approximated forward selection. We selected variables from X_0 up to a given threshold, and then selected variables from X_1 up to the given threshold. This step was repeated to X_n. Preferable priority thresholds were estimated in the experiments and are discussed in Section 7. There were 1,000 candidates (X_i) in each approximated forward selection in our experiments. Although accuracy improves with greater numbers, only slight improvements could be observed for 4,000 candidates in our experiments.

7. Experimental Results

We did experiments on Othello to prove the effectiveness of the generation and selection methods proposed. We compared evaluation functions generated by our methods with those generated by other general methods, and with the evaluation functions used in specialized Othello programs. We used a computer with an Athlon MP 2100+ CPU (1.7 GHz) for these experiments. The program was implemented in GNU C++.

7.1 Pattern Generation and Selection

First, 11,079 logical features were generated by Fawcett's (1993) method. Subsequently, 8,502,664 unique patterns were extracted from the logical features with the method proposed in Section 4. We then did selection by frequency as described in Subsection 6.1. Several sets of patterns were selected with various frequency ranges. Finally, we applied the iterative selection described in Subsection 6.2 to the resulting sets with various priority thresholds. The priority function used here was correlation, and candidates were sorted by frequency.

7.2 Accuracy of Evaluation Functions

This subsection contains the heart of our experimental research. It is subdivided into six sub-subsections, each of them dealing with a relevant item.

7.2.1 Training Positions and Labeling. Evaluation functions made up of selected patterns were constructed to enable the usefulness of patterns to be estimated. Each of the functions was a linear model of patterns. The weights were adjusted by means of least mean squares to predict the final score (difference between number of black and white discs at the end of the game after both players had played the best moves). We separately constructed the evaluation functions for the positions of 60 discs and those for 55 discs. We only used the positions of 60 and 55 discs because positions of the near the endgame can be immediately labeled with the results of a complete search.

The positions we used in selection and training were extracted from games played between LOGISTELLO and KITTY.[5] It should be noted that our proposed method works without the game records of strong game programs. The purpose of using positions taken directly from games is to gather unbiased positions and to demonstrate the method's learning ability in positions that strong programs face. About 50,000 positions were selected by eliminating duplicate positions considering the symmetry of the geometry and players. We then generated two disjoint sets of positions expanding the symmetric ones.[6] One set contained about 800,000 positions for training and the other had about 6,000 positions for testing.

7.2.2 Adjustment of Weights. Weights in evaluation functions with fewer than 10,000 patterns were adjusted with LAPACK[7], and an iterative method (BiCGSTAB[8])(Barrett et al., 1994)) was used instead, due to memory limitations, in other cases. The time for weight fitting depends on the efficiency of an evaluation function and on the number of matching patterns for a position on average. This efficiency was primarily important because the iterative method requires pattern matching over all training positions for many repetitions. The number of multiplications required for each position is about the number of matching patterns squared. Thus, it was not feasible to use all patterns generated without selection. The time for weight fitting tended to be more than a week if there were more than 100,000 patterns. Buro could use more patterns because his efficiency is much better, as will be described below, and because only 50 patterns at most should match each position due to the carefully crafted shapes.

7.2.3 Accuracy of Proposed Evaluation Functions. The graph in Figure 7 illustrates the accuracy of our evaluation functions and the others. Here, "error" in the vertical axis is the square root of mean square errors. The horizontal axis plots the number of patterns on a logarithmic scale. Our evaluation functions ("with selection") for positions with 60 discs are denoted by the '+', and those for positions with 55 discs are denoted by the '■'. The errors for 55

[5]Both are available at `ftp://external.nj.nec.com/pub/igord/IOS/misc/`.
[6]To generate evaluation functions that yield the same value at symmetric positions, symmetric patterns should have the same weight in the evaluation functions. We achieved this by simply instantiating all symmetric positions when adjusting weights.
[7]`http://www.netlib.org/lapack/`
[8]`http://netlib2.cs.utk.edu/linalg/html_templates/Templates.html`

discs are larger than those for 60 discs, because it is more difficult to predict scores for the positions of earlier game stages. The frequency ranges used in filtering were $[1.25 \cdot 10^{-5}, 0.075]$; 540,724 patterns were selected. The priority thresholds used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01; we selected approximately 3,000 to 140,000 patterns. The accuracy of our evaluation functions improved as the number of patterns increased.

7.2.4 Comparison with Logical Features or All Patterns. We compared our evaluation functions with those using logical features and those using all patterns without selection to demonstrate improvements over existing general methods.

We have already reported on a comparison of all patterns and logical features (Kaneko et al., 2001). The accuracy of evaluation functions using 18 logical features that Fawcett had selected was 12.9 and 12.5, and the accuracy of evaluation functions with 42 logical features that were statistically significant and selected with an F-test from 10,000 features was 8.90 and 12.4 for positions with 60 discs and 55 discs, respectively. Evaluation functions with logical features were more than 20 times slower than those with patterns extracted from the same logical features. The results indicate that extracted patterns are much more effective than the logical features themselves.

In Figure 7, "without selection" means the accuracy of evaluation functions that use automatically generated patterns without selection. The accuracy of our evaluation functions was far better than that of patterns without selection (plotted with '∗' and '●'). The accuracy of the latter functions were established in and taken from the authors' previous work (Kaneko et al., 2001). The results indicate that the proposed methods are more effective than existing general methods.

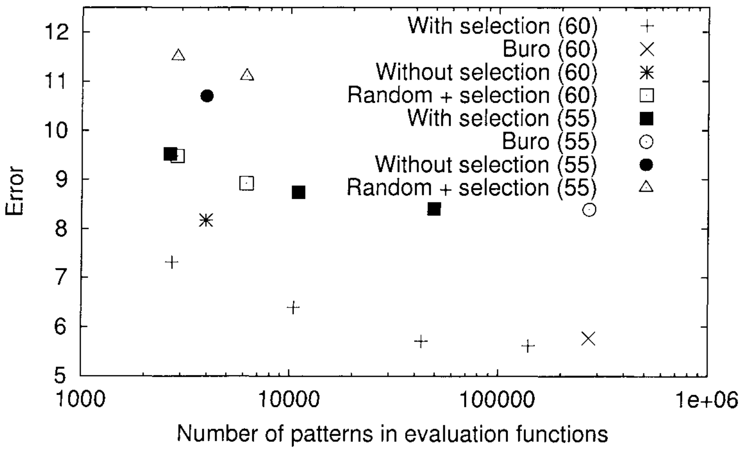


Figure 7. Accuracy of evaluation functions.

7.2.5 Comparison with Buro's Patterns. We compared our evaluation functions with those of a specialized Othello program to evaluate our accuracy. In Figure 7, "Buro" means our previous reproduction of Buro's method (Kaneko et al., 2001). The accuracy of our evaluation functions improves as the number of patterns increases, going beyond that of Buro's (plotted with the '$\odot$' and 'X'). The results indicate that accurate evaluation functions are mechanically generated, without having to incorporate manually important shapes in Othello.

7.2.6 Comparison with Randomly Generated or Selected Patterns. To demonstrate the importance of both pattern generation and selection, we constructed evaluation functions with random generation/selection instead of the proposed generation/selection, and compared their accuracies.

Random Generation + Proposed Selection. In Figure 7, "random + selection" means evaluation functions that use patterns selected with our method, from randomly generated patterns instead of the ones generated by this method. First, 8,502,664 patterns were generated, each of which was a conjunction of the randomly selected status of squares. Then, about 3,000 and 6,000 patterns were selected with the selection we propose. The difference between the accuracy of randomly generated patterns and that of ours means that our method of generating patterns is indispensable in producing useful patterns.

Proposed Generation + Random selection. We measured the accuracy of evaluation functions with 4,147 patterns that were randomly selected instead of with the selection we propose. The error was more than 14.5 and is not plotted in the graph. The difference between the accuracy of randomly selected patterns and that of our method means that our pattern selection method is indispensable in producing useful patterns.

7.3 Efficiency of Evaluation Functions

Figure 8 illustrates the efficiency and accuracy of our evaluation functions selected for various frequency ranges. The horizontal axis plots the number of patterns used in the evaluation functions and the vertical axis plots efficiency by the number of positions evaluated in one second. The priority thresholds we used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01. As the number of patterns increased, the efficiency of evaluation functions deteriorated while the accuracy improved, almost regardless of frequency ranges. For this experiment, we collected a sequence of about $3,000,000$ positions. Then the df-pn^{+}

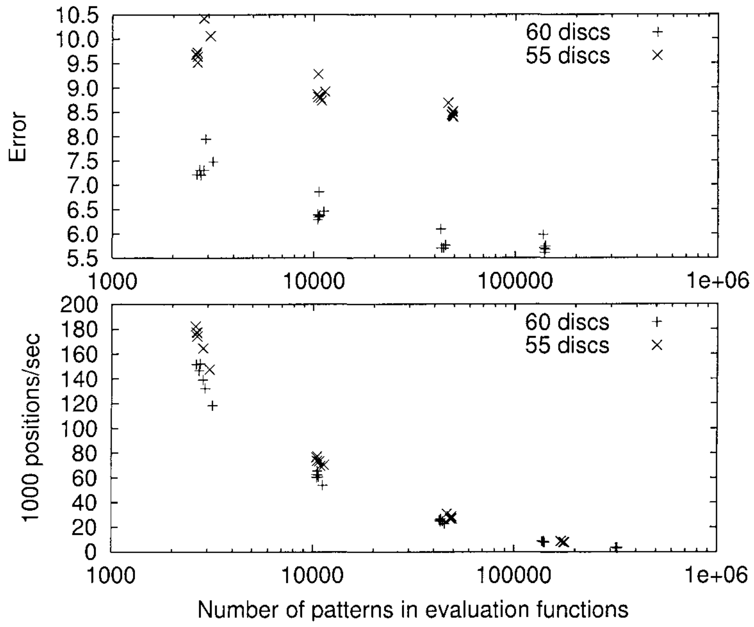


Figure 8. Efficiency of evaluation functions for various numbers of patterns (55 and 60 discs).

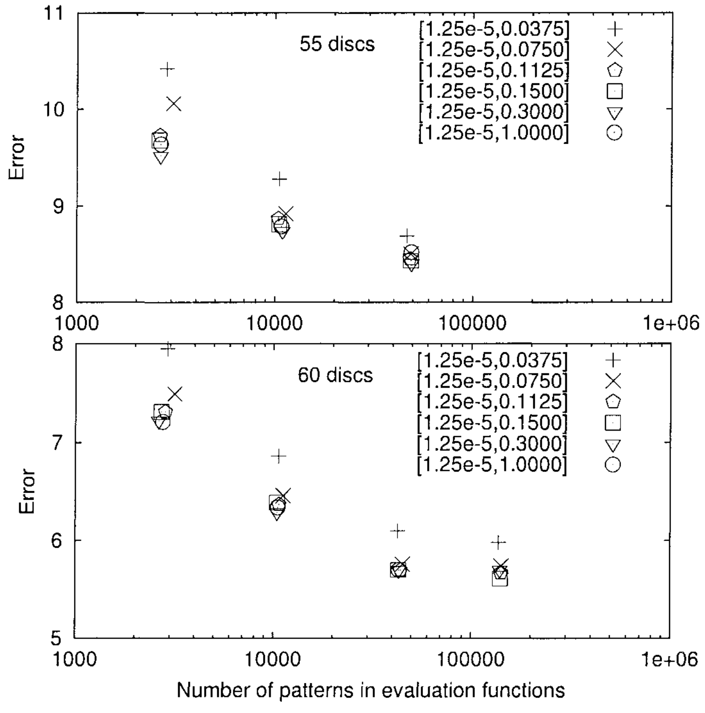


Figure 9. Accuracy of evaluation functions with various priority thresholds (55 and 60 discs).

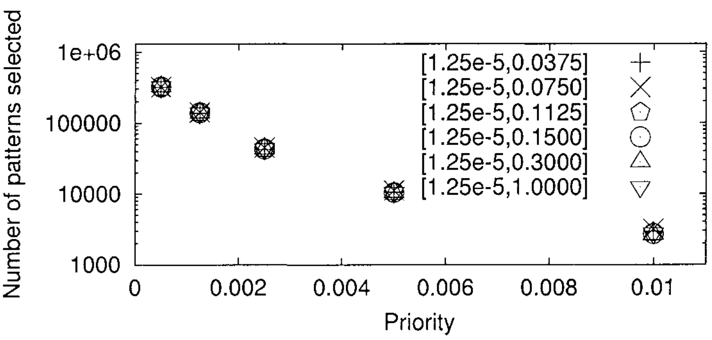


Figure 10. Number of patterns selected with various priority thresholds.

search (Nagai and Imai, 1999) visited the root positions of 49 discs, which were extracted from 23 matches in IOS records.[9]

Although the efficiency of our evaluation functions was much better than the efficiency of evaluation by logical features (Kaneko et al., 2001), it was worse than that of a specialized Othello program. LOGISTELLO's speed was about 270,000 nodes/sec when running on a Pentium-II 333 MHz (Buro, 1998). This speed would have been about 1.4 million nodes/sec (by extrapolation) if it had been run on a 1.7-GHz CPU. Further research is required to make practical evaluation functions because efficiency is usually more important than accuracy.[10] These differences were partly because we did not take efficiency into account in the selection of patterns and partly because we could have used a much more efficient pattern matching algorithm than the one we proposed if we had restricted our patterns to Buro's (1998) shapes.

7.4 Parameters for Selection

To determine appropriate values for frequency ranges and priority thresholds so that the proposed selection would work well, we investigated their influence on the efficiency and accuracy of the generated evaluation functions and on the time required for selection.

The graphs in Figure 9 plot the accuracy of our evaluation functions for positions with 60 and 55 discs, consisting of patterns selected with various frequency ranges and various priority thresholds. We can see that the frequency ranges do not distinctly affect the quality of selected patterns, if its upper boundary is greater than 0.15. Thus, we concluded that the accuracy of evaluation functions is mainly determined by the number of patterns used in them.

The priority thresholds used in selection determine the number of patterns that are finally selected. Figure 10 plots the relation between the number of

[9]These are available at `ftp://external.nj.nec.com/pub/igord/othello/ios/`.

[10]Future advances in hardware will favour the accuracy because these will eventually compensate for serious delays when in-depth searches reach a saturation point (Heinz, 2001).

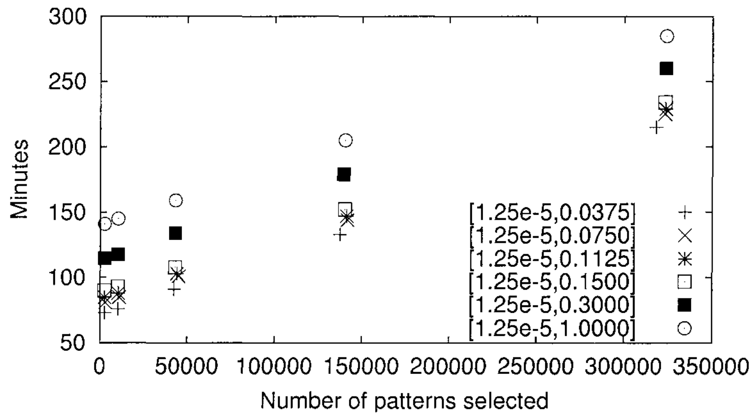


Figure 11. Time for iterative selection according to frequency ranges.

selected patterns and priority thresholds. The vertical axis plots the number of patterns on a logarithmic scale, and the horizontal axis plots the priority thresholds. Here, we used correlation for priority. We can see that the number of patterns selected is mainly determined by priority thresholds regardless of frequency ranges (denoted by symbols), and that the symbols in the graph are plotted at almost the same location if the same priority thresholds are used. Also, larger numbers of patterns are selected as lower thresholds are used. Thus, one can control the trade-off between the accuracy and efficiency of evaluation functions by adjusting the priority thresholds, because these are mainly determined by the number of patterns in them as previously discussed.

The time for iterative selection depends on frequency ranges as well as the number of selected patterns. Figure 11 plots the relation between time and the number of selected patterns with various priority thresholds and frequency ranges. The priority thresholds we used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01. The horizontal axis plots the number of patterns finally selected by iterative selection, and the vertical axis plots the time for selection in minutes. These results are acceptable because we have to inspect a larger number of candidates during iterative selection for frequency ranges with larger upper bounds.

Figure 12 plots the relation between the efficiency and accuracy of evaluation functions. The vertical axis plots accuracy by the square root of mean square errors, and the horizontal axis plots efficiency by the number of positions evaluated in one second. The one right below is to be preferred.

Considering the time for selection, accuracy, and efficiency of evaluation functions, the recommendable upper boundary for the frequency range is between 0.15 and 0.3. This value is obviously larger than the expected value 0.075 in Figure 4. It is partly because most of our patterns had fewer squares than Buro's (1998).

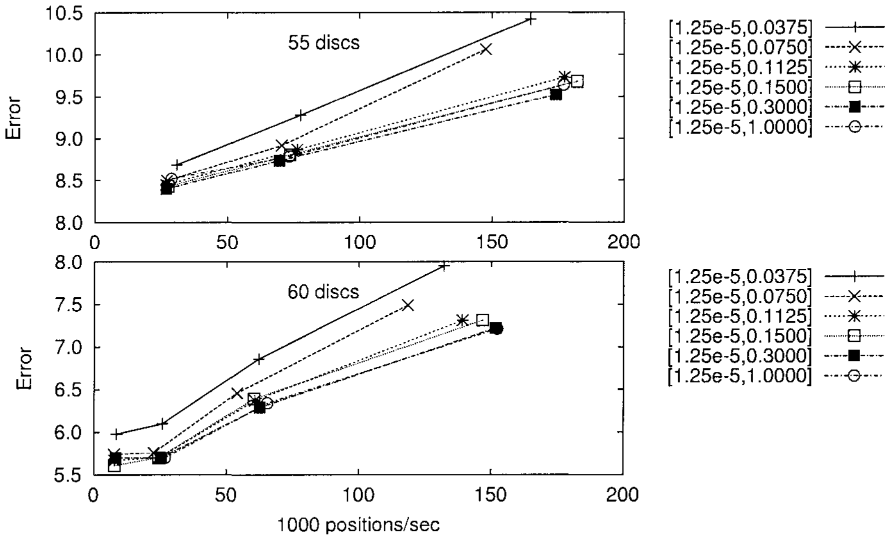


Figure 12. Efficiency and accuracy of evaluation functions (55 and 60 discs).

8. Concluding Remarks

In this paper, we described a method of constructing accurate evaluation functions by using only the specifications of a target game and a set of training positions, which is crucial in constructing a general game player. Experiments on Othello revealed that a combination of pattern generation using logic and a lightweight pattern selection could efficiently search for and identify useful patterns. The method actually constructed accurate evaluation functions. The accuracy was by far superior to the evaluation functions generated by existing general methods, and was comparable (although slightly worse) to that of Buro's (2002) which is part of a specialized Othello program.

Our intended future work aims at demonstrating the generality of the approach proposed here on other games, such as Shogi, where patterns with variable shapes are needed, and also at improving the efficiency of the generated evaluation functions in order to investigate total game-playing performance. The development of selection criteria taking efficiency into account seems promising, though investigations into their impact on accuracy would be required. It would also be challenging to develop a general method that introduces game-specific optimizations, including the use of patterns in fixed shapes, through an analysis of domain theory.

Acknowledgments

The authors would like to express their appreciation to the anonymous referees, whose comments led to significant improvements in this paper.

References

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

Bossi, A., Cocco, N., and Dulli, S. (1990). A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302.

Buro, M. (1998). From simple features to sophisticated evaluation functions. In *Proceedings of the First International Conference on Computers and Games*(eds. H.J. van den Herik and H. Iida), pages 126–145. Springer-Verlag, Berlin, Germany.

Buro, M. (2002). Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1–2):85–99.

Fawcett, T. E. (1993). *Feature Discovery for Problem Solving Systems.* PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, MA.

Gries, D. and Schneider, F. B. (1993). *A Logical Approach to Discrete Math.* Springer-Verlag, New York, NY.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182. Special Issue on Variable and Feature Selection.

Heinz, E. A. (2001). New self-play results in computer chess. In Marsland, T. A. and Frank, I., editors, *Computer and Games* (eds. H.J. van den Herik and H. Iida), number 2063 in LNCS, pages 262–276. Springer-Verlag, Berlin, Germany.

Jain, A., Duin, P., and Mao, J. (2000). Statistical pattern recognition: a review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4 – 37.

Kaneko, T., Yamaguchi, K., and Kawai, S. (2000). Compiling logical features into specialized state-evaluators by partial evaluation, boolean tables and incremental calculation. In *PRICAI 2000*, pages 72–82, Melbourne, Australia.

Kaneko, T., Yamaguchi, K., and Kawai, S. (2001). Automatic feature construction and optimization for general game player. In *The Sixth Game Programming Workshop*, number 14 in IPSJ Symposium Series 2001, pages 25–32.

Kaneko, T., Yamaguchi, K., and Kawai, S. (2002). Automatic construction of pattern-based evaluation functions for game programming. *IPSJ JOURNAL*, 43(10):3040–3047. (In Japanese)

Kojima, T., Ueda, K., and Nagano, S. (1997). An evolutionary algorithm extended by ecological analogy and its application to the game of Go. In *Proceedings of the 15th IJCAI*, pages 684–689, Nagoya, Japan.

Lincke, T. R. (2001). Strategies for the automatic construction of opening books. In Marsland, T. A. and Frank, I., editors, *Computer and Games*, number 2063 in LNCS, pages 74–86. Springer-Verlag, Berlin, Germany.

Nagai, A. and Imai, H. (1999). Application of df-pn$^+$ to Othello endgames. In *Game Programming Workshop in Japan '99*, pages 16–23.

Okuno, T., Kume, H., Haga, T., and Yoshizawa, T. (1981). *Multivariate Analysis.* Nikka-giren. (in Japanese) (this citation information is translated in English by the author of the paper)

Pell, B. D. (1993). *Strategy Generation and Evaluation for Meta-Game Playing.* Ph.D. thesis, University of Cambridge.

Rudell, R. L. (1996). Tutorial: Design of a logic synthesis system. In *Design Automation Conference*, pages 191–196, Las Vegas, NV.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II - recent progress. *IBM Journal of Research and Development*, 11(6):601–617.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–278.

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1–2):181–199.

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):145–153.