Automatic Feature Construction and Optimization for General Game Player

Tomoyuki KANEKO Kazunori YAMAGUCHI
Satoru KAWAI
Graduate School of Arts and Sciences
The University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo, 153-8902, JAPAN
{**kaneko, yamaguch, kawai**}**@graco.c.u-tokyo.ac.jp**

Abstract

In this paper, we describe our method that automatically constructs evaluation functions without any human analysis of a target game. Such automated constriction of evaluation functions is crucial to develop a general game player that can learn and play an arbitrary instance of a certain class of games. Our approach is to construct features written in logic programs from the game definition and translate them into specialized evaluator in order to get such efficiency that learning methods can try and test so many features that it produces accurate evaluation functions. We also introduced the decomposition of logical features called *thin features* in order to improve both accuracy and efficiency. Experiments on Othello endgames show that the accuracy and efficiency of evaluation functions generated by our method are approaching to those of the patten based evaluation function which is the state-of-the-art technique.
keywords: automatic feature construction, logical feature

1 Introduction

1.1 Game playing programs and evaluation functions

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 games. Game playing programs use a min-max search method combined with evaluation functions that estimates the probability to win (or preference to the player) of a *position*. Here, a position is an intermediate status of a match. In order to develop strong players, both the accuracy and efficiency of the evaluation function are important. Since an evaluation function is specific to a target game, the main issue of developing general game players is how to construct automatically evaluation functions without knowledge of human experts.

1.2 Learning evaluation functions

The popular way to automatically construct an evaluation function is to make it some combination (such as a linear combination) of evaluation primitives called *features*, and adjust the parameters of the combination [2]. In most of researches, the features have been provided by human experts of the game. The fully automated generation of appropriate features is known to be a difficult task.

Among few works on the automatic generation of features, we found Fawcett's work [3, 4] most promising. In the work, feature is represented by Horn Clause in the first-order logic. We call the clause in such use *logical feature*. His system can generate features by syntactic translation of logic programs using just only the definition of the game. However, logical features prohibitively cost on position evaluation, and thus the method was not practical until now.

Recently, Buro developed a pattern based method [2] and generated good evaluation function used in the strongest programs in Othello. In the method, the feature is Boolean conjunction of *atomic features* that are a state of a square when it is applied to Othello. The method is very practical because the use of exclusive set of configurations called *pattern* as well as the representation itself make position evaluation and learning very efficient. However, choosing appropriate patterns requires the knowledge of the important shapes in Othello. Such knowledge is not available to general game players.

1.3 Our approach

We adopted the approach of Fawcett because a logical feature allows a uniform description of rules, a goal, and position of a game, and is suitable for automatic construction. However, the problem is the cost of position evaluation. We solved this problem by the combination of techniques: partial evaluation, Boolean network with counters, and incremental propagation [8, 6]. The effectiveness of the solution is demonstrated by experiments. We developed a technique of *decomposition* of logical features into *thin features* which improves efficiency further. These speedups enabled the learning method to use and test much more features to produce more accurate evaluation functions.

This paper is organized as follows. The next section briefly reviews the definitions of logical features and Sect. 3 describes the technique for efficient position evaluation with them. Sect. 4 describes a basic idea of decomposition. Sect. 5 shows the experimental results in Othello and Sect. 6 concludes this paper.

2 Logical Features

We uniformly describe positions, features and the rules of the game in the first-order logic. Because the first-order logic is a logically well-founded language, the adoption is quite natural. This representation is general and can be applied to most games; The original work by Fawcett [3] is on Othello and a single-agent search problem, Pell used it in symmetric chess like games [10] and Kaneko applied it to Tsume-Shogi [7]. This section focuses on how to compute the values of features defined in the first-order logic. See Fawcett [3] for more details and for the way of automated construction of features.

2.1 Definition of positions

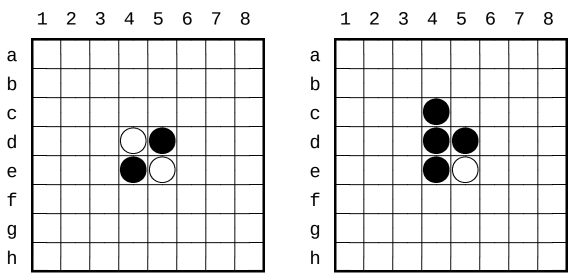
A *position*, which is an intermediate status of a match, is described by a set of facts. A fact is a clause without body. Such facts are redefined when a position changes according to the progress of a match.

In Othello, `owns` and `blank` represent a position. For example, the facts defined in the initial position in Othello and the position after black played c4 are shown in Figure 1. Here, we use `x` for black, and use `o` for white. In the initial position, `owns(d5,x)`, `owns(e4,x)`, `owns(d4,o)`, `owns(e5,o)` are defined for squares with a disc. Also `blank` is defined for each empty squares.

2.2 Definition of features

A feature is represented by Horn Clause in the first-order logic. The following is an example of a logical feature written in Prolog notation.[1]

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



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

blank(a1). blank(a2). blank(a3)...

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

blank(a1). blank(a2). blank(a3)...

Figure 1: Othello initial position (left) and a position after black played c4 (right). Facts below each board define a position.

We call the bindings of constants to variables which make the clause true *solutions* of the logical feature and the number of the bindings *value* of the logical feature. In the above example, `A` is a variable, `owns` is a predicate which means that the black player owns square `A`. So, the value of this feature `f(A)` means the number of squares currently owned by black.

In the initial position shown in Figure 1 (left), the solutions of `f(A)` are {`d5,e4`} and the value is two. In a position after black played c4 shown in Figure 1 (right), the solutions of `f(A)` are {`c4,d4,d5,e4`} and the value is four.

2.3 A domain theory

A *domain theory* is the specification of the game, which is described by a set of Horn Clauses that specify the rules of the game and the goal conditions. While facts representing a position would change according to position change, the definition of the domain theory is invariant through matches. As an example in Othello, we use a domain theory shown in Appendix A in this paper.

2.4 Position evaluation

As a position changes according to the progress of a match, solutions of predicates which depend on a position will change. In the example shown in Appendix A, `neighbor` and `square` represent the board topology and never change throughout matches. `Owns/2` and `blank/1` represent the discs in the squares in a position. `Legal_move/2` is a predicate that (indirectly) depends on a position.

Because a logical feature includes predicates which depend on a position, it is required to efficiently calculate their

[1]It is written as `f2(Num):-count([A],(owns(x,A)),Num)` in the work by [3]. In this paper, we assume counting as the default semantics of logical features and omit the predicate "count".

solutions in order to evaluate a position by logical features.

3 Efficient Position Evaluation with Logical Features

This section describes our method that efficiently evaluate positions with logical features. The method improved efficiency more than 4,000 times compared to naive interpretation by deductive databases [13] or by Prolog.

The outline of the method is as follows. First, given features are translated into the equivalent set of ground clauses (i.e., clauses without variables) by partial evaluation. Then, they are folded into a Boolean network where incremental calculation on the network can efficiently compute the solutions of the features.

3.1 Partial evaluation

First, in order to transform given features into the equivalent set of ground clauses, two operations, *unfolding* and *pruning* in partial evaluation of logic programming [1], are used.

3.1.1 Unfolding

Unfolding is an operation to replace a clause A :- A_1, ...,A_i,..,A_n with clauses $(A$:- $A_1,\ldots, A_{i-1}, B_1,\ldots, B_h, A_{i+1},\ldots,A_n)\theta_j$ for B :- $B_1,\ldots,B_h$ such that $B\theta_j = A_i\theta_j$ for some substitution θ_j. In this paper, we apply the unfolding from the left term to the right term in the depth first order. For example, unfolding a clause

```
legal_move(S,P):-square(S),bs(S,F,P).
```

with a fact `square(a1)` will produce

```
legal_move(a1,P) :- bs(a1,F,P).
```

for substitution `[S/a1]`.

3.1.2 Pruning

Pruning eliminates a clause whose body has no chance to be true. Such type of clauses can be detected by the fact that

1. its body has an unsatisfiable term, or
2. its body has a term not unifiable to any head of clauses, or
3. its body has terms unifiable to the body of some integrity constraint.

```
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).
```

Figure 2: Partial result of unfolding `legal_move`

legal_move(a1, o) = (blank(a1) $\wedge$ owns(x, a2) $\wedge$ owns(o, a3)) $\vee$ (blank(a1) $\wedge$ owns(x, b1) $\wedge$ owns(o, c1)) $\vee$ (blank(a1) $\wedge$ owns(x, b2) $\wedge$ owns(o, c3)))

Figure 3: A part of propositional definition of `legal_move(a1,o)`

In general, it is difficult to know that a given clause is unsatisfiable. In order to simplify the task to prove that a clause is unsatisfiable, we introduce integrity constraints [12] so that we can say explicitly that some combination of terms is unsatisfiable.

Appendix B shows an example of integrity constraints of the game of Othello. `Ic1` means that a square (`Square`) cannot be blank and owned by some player at the same time. `Ic2` means that a square (`Square`) cannot be owned by both black and white players. These are some of the specifications of Othello, although they have not been utilized so far.

3.2 Translation into propositional logic

Our method performs unfolding and pruning repeatedly until all the remaining clauses become ground so that they have no variables in their head or body. A partial result of unfolding `legal_move` is shown in Figure 2. Since the truth value of ground terms other than position definition can be statically computed, the unfolded clauses have only position definitions in their body.

A ground term can be treated as a Boolean variable. We call a fact of position definition (such as `owns(x,a1)`) *input variable* and call a head of unfolded clauses (such as `legal_move(a1)`) *output variable*. A true output variable corresponds to a solution of a feature. Each output variable is a disjunction of conjunctions of input variables. Figure 3 shows a propositional definition of `legal_move(a1)`.

3.3 Boolean networks

Propositional definitions of the features are folded into a Boolean network.[2] Each node in a network has its proposi-

[2]While we first proposed incremental calculation on Boolean tables in [8], incremental propagation on a multi-layer Boolean network improved efficiency further [6].

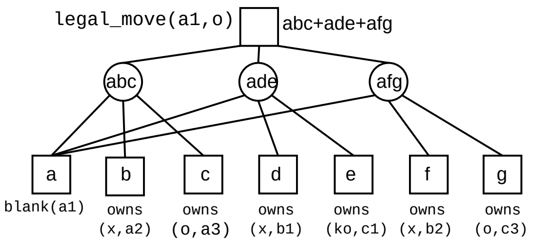


Figure 4: Network of `legal_move(a1,o).` We give a nickname a to g for each leaf, and write ab for $a \wedge b$ and $a+b$ for $a \vee b$ for brevity.

tion; a leaf has an input variable, and a non-leaf node has a conjunction (we call such a node *and-node*) or disjunction (we call such a node *or-node*) of the propositions of their children. Figure 4 is a network of the proposition shown in Figure 3.

3.3.1 Counters as Boolean values

Each non-leaf node has a counter in order to efficiently compute the truth value of its proposition. Let $dep(x)$ be the children of a node x, and $cur(x)$ be a counter that shows the number of the true children of x. [3]

$$cur(x) = |\{c \in dep(x) | val'(c) = T\}| \tag{1}$$

Here, $val'(x)$ represent a truth value defined upon the counter of x.

$$val'(x) = \begin{cases} val(x) & x \text{ is a leaf} \\ cur(x) = |dep(x)| & x \text{ is an and-node} \\ cur(x) > 0 & x \text{ is an or-node} \end{cases} \tag{2}$$

where $val(x)$ is the truth value of the proposition of the node x.

Property 1 *For each node x, the following equation holds.*

$$val(x) = val'(x) \tag{3}$$

Thanks to this property, we can compute the truth values of all the output variables by adjusting counters of nodes.[4]

3.3.2 Incremental calculation

Figure 5 shows the breadth first algorithm that visit nodes and adjust counters from lower to upper using a priority queue. Breadth first processing is better than depth first one because changes on lower variables may cancel out the

```
 1 for each n in modified leaf node
 2   push n into the priority
     queue with height 0

 3 while the queue is not empty
 4   pop the lowest node n from the queue
 5   if n is a leaf node, or
     val'(n) != previous-val'(n)
 6     for each p ∈ n's parents
 7       if n becomes true
 8         increment p's counter;
 9       else // n becomes false
10         decrement p's counter;
11       if p is not in the queue
12         push p into the queue
           with height h(p);
```

Figure 5: Incremental propagation of Boolean values

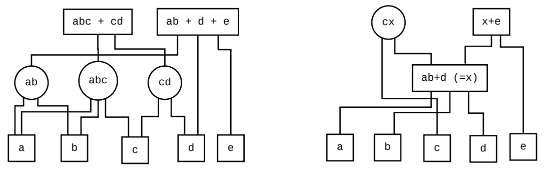


Figure 6: Edge reduction by kernel extraction

change on upper variables. Each node remembers its previous value *previous-valp* in order to detect such cancellation. For example, suppose that a becomes T where all input variables a to g were F in the network shown in Figure 4. First, a is put into the queue at line 1 and popped at the line 4. Each parent of a, i.e., abc, ade and afg, is put into the queue after its counter is incremented. In the next loop, abc is popped. The value of $cur(abc)$ $(= 1)$ is not equal to $dep(abc)$ $(= 3)$, so the propagation stops here.

Property 2 *The algorithm eventually terminates and $val'(p) = val(p)$ after the termination.*

3.4 Efficiency and optimization

The computational cost of the algorithm can be estimated by the number of adjustments of the counters. Because the algorithm will go along at most once for each edge, the worst cost is propositional to the number of edges. Kernel extraction in logic optimization [11] can reduce edges by network transformation. Figure 6 shows an example of such transformation. While both networks represent two propositions $abc + cd$ and $ab + d + e$, the right one has smaller number of edges.

[3] $|A|$ is the cardinality of a set A. T and F are the true and false respectively.

[4] The proof of Property 1 is trivial [6].

f(a1)	:-	b1,	b2.	$\Longrightarrow$	b1∧b2
f(a1)	:-	b2,	b3.		b2∧b3
f(a2)	:-	b3,	b4.		b3∧b4
f(a3)	:-	b1,	b4.		b1∧b4

Figure 7: Decomposition of a logical feature `f` (left) into four thin features (right)

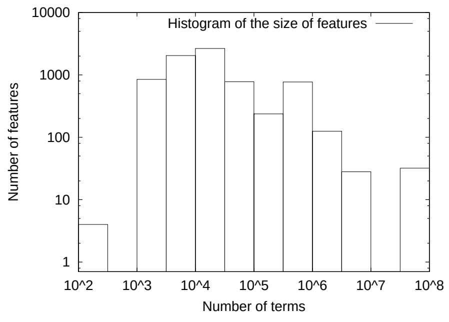


Figure 8: Histogram of the number of terms of the generated features after unfolding

4 Further Enhancements: Thin Features

Buro showed in [2] that the use of the large number of simple features produces better evaluation function than that of the small number of complicated features. As a simple application of the policy into our framework, we introduce *thin feature* that is a conjunction of input variables. The value of a thin feature is 0 or 1 according to its Boolean value. *Decomposition* is a syntactic operation that translates logical features into thin features by extracting bodies of unfolded clauses. An example of decomposition is shown in Figure 7.

Position evaluation can be efficiently performed by composing Hasse diagram [5] on the partial order of thin features and by using a slightly modified propagation method described in Sect. 3. In this propagation, we can use depth first algorithm because thin features do not contain disjunction.

It is said that using features that rarely match positions tends to cause over-fitting and makes evaluation functions unstable [2]. So, we select thin features that matches a sufficient number of training positions. This selection improves efficiency of position evaluation.

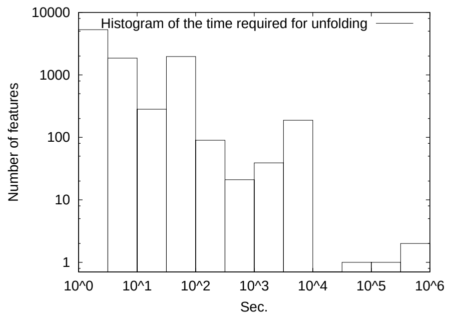


Figure 9: Histogram of the time required for unfolding

5 Experimental Results

5.1 Evaluation Functions

We trained four evaluation functions with varying features and compared the accuracy and efficiency of them. The first group consists of ones using logical features described in Sect. 2 and be evaluated with the method described in Sect. 3.

A. The first one uses logical features shown in Appendix C.1. They are slightly simplified version of the ones shown in the work by Fawcett [3]. We take this as a baseline of the evaluation .

B. The second one uses logical features selected among more than ten thousands of automatically generated ones. We applied feature generation rules [3] in breadth first order to depth five, and generated about 53*k* features. Our program unfolded about 10*k* of them with about a month of computation. The histogram of the size of generated features are shown in Figure 8. The horizontal axis shows the number of terms in unfolded clauses in log-scale. Figure 9 shows the histogram of the time required for unfolding. While most features are small and quickly unfolded, there are some prohibitively huge features that take much time to unfold. By eliminating such features. the whole computation can be finished in practical time. Then, we chose about 8*k* features that have relatively small size. Finally statistically significant 42 features are selected by F-test. They are shown in Appendix C.2

The next evaluation function uses thin features described in Sect. 4.

C. The third one uses thin features translated from logical features used in A.

The last one is a pattern based evaluation function.

Table 1: Accuracy of evaluation functions

discs		A	B	C	D
60	r	0.67	0.85	0.88	0.94
	$\sqrt{V_e}$	12.9	8.90	8.17	5.77
	$\sqrt{V_e^*}$	14.9	9.64	8.90	6.02
55	r	0.74	0.74	0.81	0.89
	$\sqrt{V_e}$	12.5	12.4	10.7	8.39
	$\sqrt{V_e^*}$	14.6	14.0	12.5	9.25

Table 2: The number of features

A	B	C	D
18	42	3952	272032

Table 3: Efficiency of evaluation (k positions/sec.)

A	B	C	D
3.18	0.872	86.6	104

D. The fourth one uses configurations in the standard eleven patterns in [2]. They contain four to ten atomic features. We take this evaluation in order to measure the difference between our evaluation functions and the one produced by the state-of-the-art techniques.

5.2 Training

All evaluation functions use linear combination and their weights are adjusted by means of least mean squares so that they predict the final score (the difference between the number of black discs and that of white ones at the end of the match after both players did the best). The weights of evaluation functions A and B were determined directly by solving covariance matrices and the weights of others were iteratively adjusted by using a conjugate gradient method. Training positions are extracted from IOS records.[5] We selected about 306k positions after removing duplicate positions considering symmetry of geometry and players. Then we used for training about 4.8M positions expanding symmetric positions.

5.3 Accuracy of Evaluation Function

We tested the accuracy of evaluation functions by using about 50k positions extracted from matches played between LOGISTELLO and KITTY. [6] We removed about fifty positions in them that are also in training positions.

Table 1 shows the result where r is the correlation coefficient, and V_e is the variance of errors, and V_e^* is the variance of errors measured for training instances. Figure 10 shows the scatter plots between prediction (horizontal axis) and real score (vertical axis). The real scores are determined by using the full width search.

Comparing evaluation function A and B, we can see that using features selected among large number of features would produce more accurate evaluation function. Comparing evaluation function A and C, we can see that using thin features instead of logical features would produce more accurate evaluation function.

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

[6]They are available at `ftp://external.nj.nec.com/pub/igord/IOS/misc/`.

5.4 Efficiency of Evaluation Function

We gathered about 3M positions by df-pn$^+$ search[9]. The search started at positions with 49 discs which are extracted from 23 matches in IOS records. Table 3 shows the average speed (kilo positions/sec.) of each evaluation function. The number of features used in each evaluation function is shown in Table 2. For the experiment, a computer with 933-MHz CPU Pentium III running FreeBSD is used and the program is implemented in GNU C++.

Since evaluation function B uses much more complex features than those of A as well as uses about twice number of features, the speed is worse than A. Comparing A and C, the use of thin features will improve the efficiency further, though the speed still did not reach to that of D.

Buro reported in [2] that his program searches about 270k nodes in a second on 333-MHz CPU Pentium II PC while our implementation of his algorithm only evaluate about 104k in a second on 933-MHz PC. This can be explained that in our implementation we avoid rigorous optimization. The weights in evaluation functions are represented as not integers but floating points. Also we do not share weights among symmetrical features.

6 Concluding Remarks

In this paper, a method to construct practical evaluation functions without human analysis of the game are described, which is crucial to construct a general game player. We showed that the accuracy and efficiency of generated evaluation functions in Othello are approaching to those of the pattern based evaluation function. Though still there is a room for improvements, the difference is diminishing.

Moreover, preliminary experiments of decomposing logical features into thin features showed improvement on both accuracy and efficiency. We are now developing methods utilizing thin features.

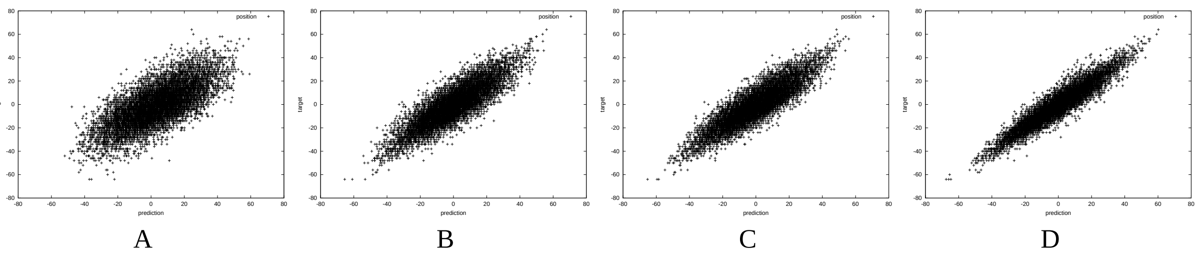


Figure 10: Scatter-plots of prediction by evaluation functions (about 10k positions with 60 discs)

References

[1] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Trans. Prog. Lang. Syst.*, 12(2):253–302, Apr. 1990.

[2] M. Buro. From simple features to sophisticated evaluation functions. In *Proceedings of the First International Conference on Computers and Games*, pages 126–145, Tsukuba, Japan, Nov. 1998. Springer-Verlag.

[3] T. E. Fawcett. *Feature Discovery for Problem Solving Systems*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, 1993.

[4] T. E. Fawcett and P. E. Utgoff. Automatic feature generation for problem solving systems. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Conference on Machine Learning*, pages 144–153. Morgan Kaufmann, July 1992.

[5] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.

[6] T. Kaneko, K. Yamaguchi, and S. Kawai. Compiling logical features into specialized boolean networks with incremental propagation. (In Japanese, sumbitted).

[7] T. Kaneko, K. Yamaguchi, and S. Kawai. Toward automatic construction of evaluation function in tsume-shogi: An efficient evaluation method for logical features. In *Game Programming Workshop in Japan '95*, pages 137–144, Oct. 1999. (In Japanese).

[8] T. Kaneko, K. Yamaguchi, and S. Kawai. Compiling logical features into specialized state-evaluators by partial evaluation, boolean tables and incremental calculation. In *PRICAI 2000 Topics in Artificial Intelligence*, pages 72–82, Melbourne, Australia, Aug./Sept. 2000.

[9] A. Nagai and H. Imai. Application of df-pn+ to othello endgames. In *Game Programming Workshop in Japan '99*, pages 16–23, Kanagawa, Japan, Oct. 1999.

[10] B. D. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, University of Cambridge, 1993.

[11] R. L. Rudell. Tutorial: Design of a logic synthesis system. In *Design Automation Conference*, pages 191–196, Las Vegas, NV USA, June 1996.

[12] J. D. Ullman. *Prinsiples of Database and Knowledge-Base Systems, Volume I: Classical Database Systems*. Computer Science Press, Maryland, 1988.

[13] J. D. Ullman. *Prinsiples of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, Maryland, 1989.

A A Simple Domain Theory of 4x4 Othello

```
%%% Rules
legal_move(Square, Player) :-
    square(Square), bs(Square, _FlipEnd, Player).
bs(S1,S3,P) :- blank(S1), opponent(P,Opp),
    neighbor(S1,Dir,S2), span(S2,S3,Dir,Opp),
    neighbor(S3,Dir,S4), owns(P,S4).
span(S1, S2, Dir, Owner) :- square(S1), square(S2),
    player(Owner), owns(Owner, S1),
    neighbor(S1, Dir, S3), span(S3, S2, Dir, Owner).
span(S, S, Dir, Owner) :-
    square(S), player(Owner), owns(Owner,S),
    direction(Dir).
%%% Dynamic Facts defined upon a position.
% owns(Player, Square).
% blank(Square).
%%% Static Rules
line(From,From,Dir) :- square(From), direction(Dir).
line(From,To,Dir) :-
    neighbor(From, Dir, Next), line(Next, To, Dir).
%%% Static Facts
opponent(black, white). opponent(white, black).
direction(n).direction(ne).direction(e).direction(se).
direction(s).direction(sw).direction(w).direction(nw).

square(a1). square(a2). square(a3). square(a4).
...
square(d1). square(d2). square(d3). square(d4).

neighbor(a1, s, a2). neighbor(a2, n, a1).
neighbor(a2, s, a3). neighbor(a3, n, a2).
...
neighbor(d4, nw, c3).
neighbor(c4, ne, d3). neighbor(d3, sw, c4).
```

```
%%% ‘‘goal-regression’’ feature generation rule
%%% required some predicates.
span_star(Begin,End,Dir,Owner) :-
  span(Begin,End,Dir,Owner).
span_star(Begin,Begin,Dir,Owner) :-
  square(Begin),direction(Dir),player(Owner).
span_with_subspan(Begin,End,Dir,Owner) :-
   span_star(Begin,MS,Dir,Owner), blank(MS),
   opponent(Owner,Opp), neighbor(MS,Dir,Next),
   span(Next,Next2,Dir,Opp),
   neighbor(Next2,Dir,BSend), owns(Owner,BSend),
   span_star(BSend,End,Dir,Owner).
```

B Integrity Constraints of Othello

```
ic1(Square) :- blank(Square), owns(_Player,Square).
ic2(Square) :- owns(black,Square), owns(white,Square).
```

C Features Used in Experiments

C.1 Features generated by the Zenith system

```
discs_x(S) :- square(S), owns(x,S).
discs_o(S) :- square(S), owns(o,S).
moves_x(S) :- legal_move(S,x).
moves_o(S) :- legal_move(S,o).
axes_x(A,B,C) :- owns(x,A), bs(B,C,o), in_line(A,B,C).
axes_o(A,B,C) :- owns(o,A), bs(B,C,x), in_line(A,B,C).
aaas_x(S) :- owns(x,S), corner(S).
aaas_o(S) :- owns(o,S), corner(S).
pre_aaas_x(S) :- corner(S), legal_move(S,x).
pre_aaas_o(S) :- corner(S), legal_move(S,o).
% predecessor of frontier directions
pfd_x(C) :- blank(A), neighbor(A,C,D), owns(o,D),
  neighbor(D,C,E), owns(x,E).
pfd_o(C) :- blank(A), neighbor(A,C,D), owns(x,D),
  neighbor(D,C,E), owns(o,E).
% similar to Rosenbloom frontier
srf_x(A) :- blank(A), neighbor(A,C,D), owns(o,D),
  neighbor(D,C,E).
srf_o(A) :- blank(A), neighbor(A,C,D), owns(x,D),
  neighbor(D,C,E).
% similar to Rosenbloom empty
sre_x(E) :- blank(A), neighbor(A,C,D), span(D,E,C,o),
  neighbor(E,C,F).
sre_o(E) :- blank(A), neighbor(A,C,D), span(D,E,C,x),
  neighbor(E,C,F).
% similar to Rosenbloom empty
srse_x(A,C) :- blank(A), neighbor(A,C,D), owns(o,D),
  neighbor(D,C,E).
srse_o(A,C) :- blank(A), neighbor(A,C,D), owns(x,D),
  neighbor(D,C,E).
```

C.2 Selected features

```
score_o(S):-owns(o,S).      f10(S):-legal_move(S,x).
f13(S):-legal_move(S,o).    f26(S,_T):-bs(S,_T,x).
f35(S,T,M):-legal_move(S,x), bs(M,T,o), in_line(S,M,T).
f39(S,_T):-bs(S,_T,o).
f140(S,D,T,U,M,Q):-owns(o,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,o),
  neighbor(T,D,U), owns(x,U), in_line(S,M,T).
f143(S,D,T,U,M):-owns(o,S), blank(M),
  neighbor(M,D,T), owns(o,T), neighbor(T,D,U),
  owns(x,U), in_line(S,M,T).
f162(_T):-bs(S,_T,x).
f265(S,D,T,U,M,Q):-owns(x,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,x),
  neighbor(T,D,U), owns(o,U), in_line(S,M,T).
f268(S,D,T,U,M):-owns(x,S), blank(M),
  neighbor(M,D,T), owns(x,T), neighbor(T,D,U),
  owns(o,U), in_line(S,M,T).
f287(_T):-bs(S,_T,o).
f421(S,D,T):-owns(o,S), line(S,T,D).
f1278(S,D,T,U,M):-owns(o,S), neighbor(M,D,T),
  owns(o,T),neighbor(T,D,U),owns(x,U),in_line(S,M,T).
f1316(S,D,T,U,M):-owns(o,S),blank(M),neighbor(M,D,T),
  owns(o,T), neighbor(T,D,U), in_line(S,M,T).
f1571(S,D,T,U,M,Q):-owns(o,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,o),
  neighbor(T,D,U), legal_move(U,x),in_line(S,M,T).
f1619(S,D,T,U,M):- owns(o,S), blank(M),
  neighbor(M,D,T), owns(o,T), neighbor(T,D,U),
  legal_move(U,x), in_line(S,M,T).
f1665(S,D,T,U,M,Q):-owns(o,S),blank(M),neighbor(M,D,Q),
  owns(o,Q), neighbor(Q,D,T), owns(o,T),
  neighbor(T,D,U), owns(x,U), in_line(S,M,T).
f1884(S,D,Next,M):-owns(o,S), bs(M,T,x), line(M,S,D),
  neighbor(S,D,Next), line(Next,T,D).
f2124(S,M):-legal_move(S,x), bs(M,T,o), in_line(S,M,T).
f2798(x,T):-legal_move(S,x), bs(M,T,o), in_line(S,M,T).
f2951(S,D,T,U,M,Q):-owns(x,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,x),
  neighbor(T,D,U), in_line(S,M,T).
f2952(S,D,T,U,M):-owns(x,S),blank(M),neighbor(M,D,T),
  owns(x,T), neighbor(T,D,U), in_line(S,M,T).
f3196(x,D,T,U,M,Q):-owns(x,S), blank(M),
  neighbor(M,D,Q),span_with_subspan(Q,T,D,x),
  neighbor(T,D,U), owns(o,U), in_line(S,M,T).
f3207(S,D,T,U,M,Q):-owns(x,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,x),
  neighbor(T,D,U), legal_move(U,o), in_line(S,M,T).
f3255(S,D,T,U,M):-owns(x,S), blank(M),
  neighbor(M,D,T), owns(x,T), neighbor(T,D,U),
  legal_move(U,o), in_line(S,M,T).
f3301(S,D,T,U,M,Q):-owns(x,S),blank(M),neighbor(M,D,Q),
  owns(x,Q), neighbor(Q,D,T), owns(x,T),
  neighbor(T,D,U), owns(o,U), in_line(S,M,T).
f3504(S,D,T):-owns(x,S), line(S,T,D).
f3522(S,D,T,Next):-owns(x,S), bs(M,T,o), line(M,S,D),
  neighbor(S,D,Next), line(Next,T,D).
f3881(S,D,T,U,M):-neighbor(M,D,T), owns(o,T),
  neighbor(T,D,U), owns(x,U), in_line(S,M,T).
f4166(S,D,T,U,M):-blank(M), neighbor(M,D,T),
  neighbor(T,D,U), owns(x,U), in_line(S,M,T).
f4176(S,D,T,U,M):-blank(M), neighbor(M,D,T),
  legal_move(T,o), neighbor(T,D,U), owns(x,U),
  in_line(S,M,T).
f4841(S,D,T):-legal_move(S,o), line(S,T,D).
f4939(S,D,T,M):-owns(o,S),neighbor(M,D,S),line(S,T,D).
f4943(o,D,T):-owns(o,S), line(S,T,D).
f5041(o):-owns(o,S), in_line(S,M,T).
f5672(o,T):-legal_move(S,o), bs(M,T,x), in_line(S,M,T).
f5702(S,D,T,M):-legal_move(S,o), bs(M,T,x),
  line(M,S,D), line(S,T,D).
f5732(S,D,T,U,M):-legal_move(S,o), blank(M),
  neighbor(M,D,T), owns(o,T),
  neighbor(T,D,U), owns(x,U), in_line(S,M,T).
f5825(S,D,T,U,M,Q):-owns(o,S), blank(M),
  neighbor(M,D,Q), span_with_subspan(Q,T,D,o),
  neighbor(T,D,U), in_line(S,M,T).
f6070(D,T,U,M,Q):-owns(o,S),blank(M),neighbor(M,D,Q),
  span_with_subspan(Q,T,D,o), neighbor(T,D,U),
  owns(x,U), in_line(S,M,T).
f6172(S,V,D,T,U,M,Q):-owns(o,S), blank(M),
  neighbor(M,D,Q), owns(o,Q), neighbor(Q,D,V),
  span_with_subspan(V,T,D,o), neighbor(T,D,U),
  owns(x,U), in_line(S,M,T).
```