Compiling Logical Features into Specialized State-Evaluators by Partial Evaluation, Boolean Tables and Incremental Calculation

KANEKO Tomoyuki, YAMAGUCHI Kazunori, and KAWAI Satoru

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. A good evaluation function is needed for a good game program, and good features, which are primitive metrics of a state, are needed for a good evaluation function. In order to obtain good features, automatic generation of features by machine learning is promising. However, the generated features are usually written in logic programs, whose evaluation is much slower than that of other native expressions due to the interpretive evaluation of the logic programs. In order to solve this problem, we propose a method which constructs a specialized evaluator using a combination of techniques: partial evaluation, Boolean tables, and incremental calculation. It exhaustively unfolds logical programs until they can be represented as simple Boolean tables. The constructed specialized evaluator is efficient since it consults only these compiled tables. Experiments with Othello showed that speed can be increased approximately 2,000 times.

1 Introduction

1.1 Evaluation Function and Features

In order to make computer players of games strong, an *evaluation function* of possible states in a match plays a crucial role, and the automatic construction of a good evaluation function is a challenging research goal. The popular way to construct an evaluation function automatically is to make it a linear combination[1] of evaluation primitives called features, and to adjust the parameters of the combination[10] [3]. In most of this research, the features have been provided by human experts of the game, and the automatic generation of features remains an ambitious research goal.[2]

R. Mizoguchi and J. Slaney (Eds.): PRICAI 2000, LNAI 1886, pp. 72-82, 2000.
©Springer-Verlag Berlin Heidelberg 2000

[1] More complex mechanisms such as neural networks are often used as well.

[2] Although we use two-player games as an example of search problems in this paper, the proposed method can be directly applied to single- (and multi-) agent search problems as well.

1.2 Logical Features

Among few works on the automatic generation of features, we found Fawcett[4][5] most promising. In the work, a feature is represented by Horn Clause in the first-order logic. We call the clause in such use *logical feature*.[3]

Because the first-order logic is a logically well-founded language and rules of a game can be described in it, the adoption of the first-order logic as the description of features is quite natural. This is an example of a logical feature.[4]

```
f(A):-owns(black,A).   % PIECES FOR BLACK
```

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)` emits the number of squares currently owned by black.

A logical feature allows a uniform description of rules, a goal, and states of a game, and is suitable for automatic construction. However, its universality makes the evaluation quite costly. We solved this problem by a new combination of techniques: partial evaluation, Boolean tables with index, and incremental calculation with counters. The effectiveness of the solution is demonstrated by experiments.

This paper is organized as follows. The next section briefly reviews the problem and previous works and presents the main idea of our approach. Sect. 3 details our method. Sect. 4 shows the experimental results. Sect. 5 concludes this paper.

2 Logical Feature Evaluation Problem

2.1 A Domain Theory and Dynamic Facts

Features consist of two types of predicates: those of a domain theory and those defined for a state.

A *domain theory* is the specification of a game, which is described by a set of Horn Clauses that specify the rules of the game and the goal conditions. The domain theory is independent of matches.

A *state* is an intermediate status of a game, which is described by a set of facts. A fact is a clause without body. A state changes according to the progress of a match. Such facts are called *dynamic facts*.

---

[3] It is also used in Metagame[8].

[4] It is written as `f2(Num):-count([A],(owns(black,A)),Num)` in the work by [4]. In this paper, we assume counting as the default semantics of logical features and omit the predicate "count".

2.2 State Change and Logical Feature Evaluation

As a state changes according to the progress of a match, solutions of predicates which depend on dynamic facts change. We call such predicates *dynamic rules.*

An example of domain theory for Othello-4x4 is shown in Appendix A. In the example, `neighbor/3` and `square/1` represent the board topology and never change throughout matches. So, they are non-dynamic facts. `Owns/2` and `blank/1` represent the stones in squares in a state. Since they change according to the progress of matches, they are dynamic facts. `Legal_move/2` is a predicate which depends on a state. So, it is a dynamic rule.

Because a logical feature often includes a component of dynamic rules, it is required to efficiently calculate their solutions in order to evaluate the logical feature.

2.3 Related Works

The evaluation of predicates is studied in a few fields under slightly different contexts:

Logic Programming In the field of logic programming, the emphasis is on the flexibility and the SLD-resolution is still the most popular way to find a solution. For the complete enumeration of solutions, a technique called tabling (or often called memorization) [9] is used.

Deductive Databases In the field of deductive databases[11], the emphasis is on the complete enumeration of the solutions. Also, an incremental update has been studied and is called materialized view maintenance[6].

Production Systems In the field of production systems[1], the emphasis is on detecting a change in the truth values of rules in order to trigger events. For such change propagation, the discrimination network has been studied (RETE[7].)

2.4 Our Approach

For logical feature evaluation, we have to find out the number of solutions of rules, and a complete enumeration is required. Therefore, we use the techniques of the materialized view maintenance as a reference. The rules of a game often depend on a state. For example, in the game of Othello, the rule `span` depends on the `owns` where truth values depend on a state. The evaluation of such rules cannot be accelerated by just materializing intentional databases. Since state changes are usually not drastic for popular games, the true dynamic facts in two different states tend to only slightly vary. In such cases, the incremental maintenance technique of such materialized views can be useful.

In our approach, we use the partial evaluation technique[2] for speeding up the evaluation of such rules. By repeatedly applying unfolding and pruning, we can make each rule fully expanded until the body of the rule consists of dynamic facts only. Theoretically, the unfolding and pruning process may continue

infinitely, but, for the rules of popular games which have finite boards, it terminates and produces fully expanded rules.

The RETE network[7] represents the dependency of rules on dynamic facts, and it is often used for propagating a change in the truth values of dynamic facts to changes in those of rules. The unfolding and pruning process is a symbolic way to calculate this propagation process in advance in order to speed up the evaluation.

Once we have a direct relation between rules and dynamic facts, we can encode the relation in Boolean tables. Then, the incremental evaluation on the tables can be sped up further by associating the tables with counters in order to detect the crossing of marginal numbers of trues and falses and proper indexing.

In summary, our approach is a combination of techniques: partial evaluation, Boolean tables with indices, and incremental calculation with counters. We applied this approach to the game of Othello, and see that this combination generates a specialized evaluator which is approximately 2,000-times faster than the reference. So, we believe that this is the right combination of techniques and is therefore worth further study.

3 Generation of Specialized Evaluator

In this section, we explain how we generate a specialized evaluator by the combination of the techniques of the partial evaluation, Boolean tables, and incremental calculation.

3.1 Partial Evaluation

First, given features are transformed into the equivalent set of ground clauses (i.e., clauses without variables). Two operations, *unfolding* and *pruning* in partial evaluation of logic programming[2] are used.

Unfolding Unfolding is an operation to replace a clause A :- $A_1, ..., A_i, .., A_n$ with clauses $(A$:- $A_1, ..., A_{i-1}, B_1, ..., B_h, A_{i+1}, ..., A_n)\theta_j$ for B :- $B_1, ..., B_h$ 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.

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.

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 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 in previous works.

Exhaustive Partial Evaluation Algorithm Our method performs unfolding and pruning repeatedly until all the remaining clauses are grounded so that they have no variables in their head or body.

Appendix A shows a sample predicate `span` which represents a consecutive line of stones in Othello. In our strategy, terms are unfolded from left to right: i.e., terms `square(S1)` and `neighbor(S1,Dir,S3)` are unfolded before `span(S3,S2,Dir,Owner)` when the first definition of `span` is unfolded. Since all variables in the term `span(S3,S2,Dir,Owner)` are bound in the prior process, the partial evaluation of the term will stop after the unfolding operation is applied four times in this case, which is the size of the board. Finally, the exhaustive partial evaluation on `span` produces the following clauses:

- span(a1,a2,s,black) :- owns(black,a1),owns(black,a2).
- span(a1,a3,s,black) :- owns(black,a1),owns(black,a2),owns(black,a3).

  ...
- span(a1,a1,s,black) :- owns(black,a1).
- span(a2,a2,s,black) :- owns(black,a2).

  ...

Generally speaking, this process of unfolding and pruning may continue forever due to some recursively defined clauses. In the case of conventional games with reasonable rules, however, it is easy to write features and a domain theory so that this process stops, due to the finiteness of the number of squares and satisfiable terms.

Property 1. The exhaustive partial evaluation terminates under the following assumption.

1. The number of solutions of each dynamic fact is finite. Also, it is assumed that they are bound by other terms which are placed to the left of dynamic facts in each clause. For example, `square(S1), player(Owner)` bind the solutions of `owns(Owner, S1)` in the first definition of span.
2. For each predicate, the solutions are finite and can be enumerated by the SLD-resolution at any state. The leftmost term is selected in the SLD-resolution.

Proof. First, it should be noted that the unfolding operation preserves solutions. When one unfolding operation replaces a clause c in a program with a set of clauses $c_1, ...c_n$, the original program and the new program with replaced clauses

have the same solutions for all goals. The pruning operation preserves solutions as well.

Under the assumption (1), exhaustive unfolding of a clause c totally produces clauses equivalent to nodes in the SLD-tree whose goal is the body of c.

Because of the assumption (2), such nodes are finite, the exhaustive partial evaluation of a clause terminates after a finite number of clauses are produced. □

Therefore, this method cannot be applied to such games in which the number of the solutions of some dynamic fact is infinite. However, the board and pieces are finite and the solutions of dynamic facts are also likely to be finite in conventional games.

3.2 Boolean Tables

Each clause in the final set of clauses after successful exhaustive partial evaluation has a ground head (`span(a1,a2,s,black)`,) and a body which is a conjunction of dynamic facts (`owns(black,a1)-owns(black,a2)`.) We decompose this relationship into two tables: an and-table and or-table. The and-table represents the relationship between the conjunction of dynamic facts and a dynamic fact, and the or-table represents the relationship between a set of the conjunctions of dynamic facts and a set of ground heads.

Table 1 shows a part of the and-table for exhaustively partially evaluated clauses in Sect. 3.1. The and-table shows that a conjunction in the left column becomes true if and only if all the dynamic facts with ◯ in the same row are true.

Table 1. A part of an and-table

conjunction of dynamic facts	dynamic facts owns(b,a1)	 owns(b,a2)	 owns(b,a3)
owns(b,a1)-owns(b,a2)	◯	◯	
owns(b,a2)-owns(b,a3)		◯	◯
owns(b,a1)-owns(b,a2)-owns(b,a3)	◯	◯	◯

*'b' stands for black.

Table 2 shows a part of the or-table for exhaustively partially evaluated clauses in Sect. 3.1. The or-table shows that all the ground heads in the right column are true if and only if any one of the conjunctions of dynamic facts in the left column is true. Ground heads with the same set of conjunctions are gathered and placed in a single row of the or-table.

3.3 Incremental Calculation

Counting In order to speed up the detection of conjunctions which change their truth values, we associate each row in an and-table with a counter. The counter

Table 2. A part of an or-table

the set of conjunction	the set of ground heads
{owns(b,a1)-owns(b,a2)}	{span(a1,a2,s,b),span(a2,a1,n,b)}
{owns(b,a2)-owns(b,a3)}	{span(a2,a3,s,b),span(a3,a2,n,b)}
{owns(b,a1)-owns(b,a2)-owns(b,a3)}	{span(a1,a3,s,b),span(a3,a1,n,b)}

*'b' stands for black.

represents a number of true dynamic facts in the row, and it is incremented or decremented when the related dynamic facts change their truth values. A conjunction is true if and only if its associated counter has the same value as the number of related dynamic facts. So, the change in the truth values of conjunctions is promptly detected by adjusting their counters.

Similarly, a counter is associated with the set of ground heads in each row. The counter represents a number of true conjunctions in the left column of the row. Since a ground head is true if and only if any one of the related conjunctions is true, the truth value of the head is promptly determined by checking whether its associated counter is zero or not.

The standard indexing technique is employed to find out the related rows in the and-table and or-table.

Difference Propagation Changes in the truth values of dynamic facts cause changes in those of the ground heads. We compute them by the following incremental algorithm. Suppose that dynamic fact P changes its value.

```
For each row in the and-table which is related to the dynamic fact P.
   adjust the counter associated with the row.
   if the conjunction becomes true  ... (1)
      find the rows in which the conjunction is included using the index.
      then adjust the counters associated with the rows and
      report the ground heads in the rows with counters
      changing their value from zero to nonzero. ...(2)
   if the conjunction becomes false ... (3)
      find the rows in which the conjunction is included using the index.
      then adjust the counters associated with the rows and
      report the ground heads in the rows with counters
      changing their value from nonzero to zero. ...(4)
```

We show an example. Suppose that dynamic facts {owns-a1-black, owns-a2-black} are true in Table 1 and Table 2, and we delete {owns(black,a1)} (makes owns(black,a1) false) and insert {owns(black,a3)} (makes owns(black,a3) true). First, it is reported that owns(black,a1)-owns(black,a2) becomes false at (3), and span(a1,a2,s,black) and span(a2,a1,n,black) become false at (4). Then, it is reported that owns(black,a2)-owns(black,a3) becomes true at (1), and span(a2,a3,s,black) and span(a3,a2,n,black) become true at (2).

4 Experimental Results

4.1 Time Efficiency

Experiments are performed on the game of Othello. We used a domain theory which is a slightly simpler version than that which was used in the Zenith system([4]).[5]

Table 3. Comparison of three methods in evaluation time

Method	average (sec.)	standard deviation	states/sec
Our method (incremental)	0.027	0.0048	2186.7
Our method (from scratch)	0.13	0.0072	459.7
Deductive DB	114.4	4.98	0.536

Table 3 shows the relation between the evaluation methods and the evaluation time. We applied three evaluation methods over 300 matches using 127 features. The first evaluation method is the one proposed in this paper. The second evaluation method is identical to our method except that it is without incremental calculation, i.e, for each state, all dynamic facts in the state are evaluated from scratch. The last evaluation method is a bottom-up evaluation technique used in the deductive databases[11]. 300 matches are extracted from the game records of IOS[6] consisting of 17931 states. Relatively simple 127 features are selected manually from the ones generated automatically by the Zenith's method. The purpose of the selection is to perform slow evaluation by deductive databases so as to finish in a reasonable time. For the experiment, a computer with 200-MHz CPU Pentium Pro. running FreeBSD is used and the program is implemented in GNU C++. As seen in the table, our method is about 2,000-times faster than that used in the deductive databases. The merit of the incremental calculation is in the factor of five. This efficiency is partly thanks to the efficient implementation language suitable for low-level operations such as counter adjustments.

It is said that top-level Shogi programs with relatively heavy evaluation functions can examine more than 3,000 states in a second. It seems that this method achieved almost the same efficiency. It is, however, difficult to compare them directly, because the efficiency of evaluation functions strongly depend on what they evaluate (what features they have).

Fig. 1 shows the dependency of the evaluation time on the number of features. A point on the graph shows the mean evaluation time of a match and its error-bar shows its standard deviation. The figure suggests that the evaluation time increases almost linearly as the number of features increases, although it actually depends on the methods for generating features. In this experiment, we

[5] The original version is available at ftp://ftp.ics.uci.edu/pub/machine-learning-databases/othello/.

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

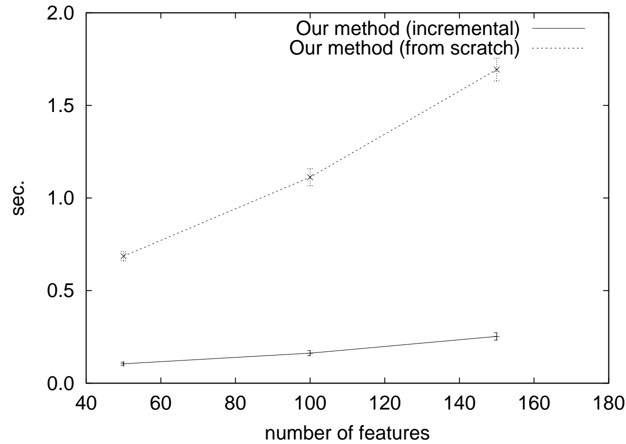

Fig. 1. Dependency of the evaluation time on the number of features

employed the features selected randomly from automatically generated ones and used a computer with 600-MHz CPU Pentium III running Linux. The matches employed are the ones we used in the previous experiment.

4.2 Space Efficiency

Experiments are performed for the game of Othello with varying board sizes in order to find out the space complexity of our approach. Table 4 and Fig. 2 show the number of clauses in the original domain theory, ground clauses after exhaustive partial evaluation, and the size, that is the number of the rows of the and-table and or-table. They show that the number of the clauses and the size of the tables increase almost exponentially with respect to the number of features.

Table 4 also shows the time required for exhaustive partial evaluation.

Table 4. Comparison of unfolded domain theories for various sizes of Othello

	Othello 4x4	Othello 8x8	Othello 16x16
no. of clauses	129	513	2345
no. of unfolded clauses	6653	211101	6217082
no. of counters in and-table	2100	116940	4559737
no. of counters in or-table	773	8459	82570
time for unfolding	24.4 sec	2556.4 sec	(about a week)

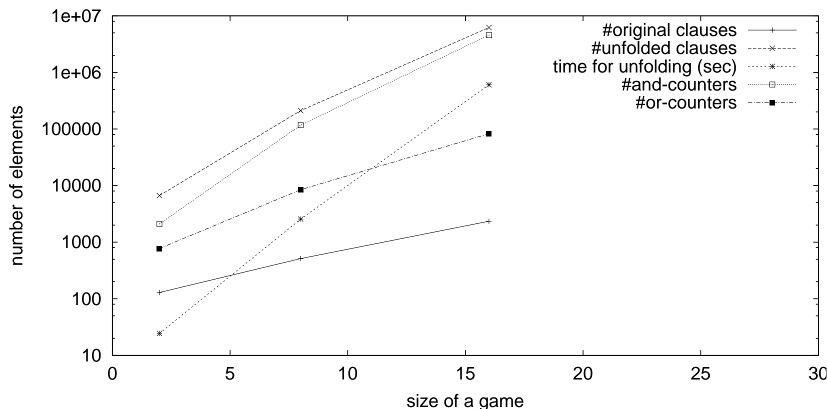

Fig. 2. Comparison of unfolded domain theories for various sizes of Othello

5 Conclusion

In this paper, we proposed an evaluation method of logical features by combining exhaustive partial evaluation, Boolean tables, and incremental calculation.

Experiments on features and a domain theory for the game of Othello showed that the proposed method was 2000-times faster than the naive bottom-up evaluation method used in the deductive databases.

There is room for improvement in our method. The efficiency of the exhaustive partial evaluation depends on the order of terms to which unfolding is applied. The automatic selection of the appropriate term to unfold is important in order to reduce users' efforts. For a more powerful description of features, we would like to incorporate aggregation predicates such as maximum or minimum in our method. These are our future works.

References

1. N. Bassiliades and I. Vlahavas. DEVICE: Compiling production rules into event-driven rules using complex events. *Information and Software Technology*, 39(5):331–342, 1997.
2. A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.
3. M. Buro. From simple features to sophisticated evaluation functions. In *Proceedings of the First International Conference on Computers and Games*, pages 126–145. Springer-Verlag, 1998.
4. T. E. Fawcett. *Feature Discovery for Problem Solving Systems.* PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, 1993.
5. 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, 1992.

6. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):157–166, June 1993.
7. H. S. Lee and M. I. Schor. Match algorithms for generalized rete networks. *Artificial Intelligence*, 54:249–274, 1992.
8. Barney Darryl Pell. *Strategy Generation and Evaluation for Meta-Game Playing.* PhD thesis, University of Cambridge, 1993.
9. Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. *ACM SIGMOD*, 5:442–453, 1994.
10. A. L. Samuel. Some studies in machine learning using the game of checkers. ii - recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.
11. Jeffrey 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 states.
% 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(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(a4).
...
square(d1). square(d2). square(d3). square(d4).

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

B Integrity Constraints of Othello

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