A Planner with Quality Goal and Its Speed-up Learning for Optimization Problem

Masahiko Iwamoto
NEC Corporation
1–1 Miyazaki 4-chome
Miyamae-ku, Kawasaki 216 Japan
+81-44-856-2315
iwamoto@swl.cl.nec.co.jp

Abstract

Aims of traditional planners had been limited to finding a sequence of operators rather than finding an optimal or near-optimal final state. Consequently, the performance improvement systems combined with the planners had only aimed at efficiently finding an arbitrary solution, but not necessarily the optimal solution. However, there are many domains where we call for quality of the final state for each problem. In this paper, we propose an extension of a planner for optimization problems, and another application of EBL to problem solving: learning control knowledge to improve searching performance for an *optimal* or a *near-optimal* solution by analyzing reasons a solution is better than another. The proposed method was applied to technology mapping in LSI design, a typical optimization problem. The system with the learned control knowledge synthesizes optimal circuits four times faster than that without the control knowledge.

Introduction

Aims of traditional planners had been limited to finding a sequence of operators, i.e. plan, which transforms an initial state into a state satisfying a given goal, rather than finding an optimal or near-optimal final state. In other words, a solution in those planners means a plan, but not a final state. However, in some domains, the final state needs to be considered as a part of solution. There are many domains where we call for quality of the final state for each problem.

On the other hand, analytical learning methods such as EBL(Mitchell, Keller, & Kedar-Cabelli 1986)(DeJong & Mooney 1986)have been approved to be effective learning methods to improve problem solving performance(Minton *et al.* 1989)(Bhatnagar 1992)(Gratch & DeJong 1992). All of the learning methods had aimed to efficiently find an arbitrary solution, but not necessarily the optimal solution. For example, PRODIGY(Minton *et al.* 1989), which is a planner, learns control knowledge to improve problem solving performance by analyzing reasons for success, failure, and goal interference.

1. Input is represented logically in terms of "unmapped-and", "unmapped-or", and "unmapped-not".
2. There is a particular library where the number of cells to implement each component is specified;

g-nand2	two-input-nand-gate	2 cells
g-nor2	two-input-nor-gate	2 cells
g-inv1	inverter	1 cell.

3. The goal is to map input equations on to the library (in other words, to build a circuit using the library), and minimize the total number of cells.

Figure 1: Technology mapping domain.

In this paper, we present one of the domains where we call for quality of the final state, LSI design, and an extension of the PRODIGY planner(Blythe *et al.* 1992) to solve optimization problems such as LSI design. Furthermore, a learning method to learn control rules to efficiently find an optimal or near-optimal solution follows. Finally, experimental results and comparison with other learning methods are presented.

Planner for Optimization Problem

This section gives an example of optimization problems, LSI design problem, and problem solving method for optimization problems based on planner architecture.

LSI Design Problem

The process of LSI design is mainly composed of architecture design, function design, logic design, layout and fabrication. Here, we will pick up logic design, especially technology mapping, which is a process to map a technology-independent description of a circuit into a particular technology (e.g. CMOS). Figure 1 shows a simplified form of technology mapping domain.

Quality Goal

A part of the goal in the technology mapping domain, "to minimize the total cell number", cannot be repre-

```
quality goal ::= (MINIMIZE (FUNCTION e-function)
                           (MIN min)
                           (PREDICATES
                            quality-predicates))
min ::= number
e-function ::= function of variables in
                        quality-predicates
quality-predicates ::= predicate+
```

(a) Syntax

```
(NGOAL (mapped))                    ; necessity-goal
(QGOAL (MINIMIZE (FUNCTION <x>) ; quality-goal
                 (MIN 4)
                 (PREDICATES
                  (total-cell-num <x>))))
```

(b) Example

Figure 2: Quality goal.

sented in PRODIGY's language, as well as other planner languages based on first order logic. We introduce a new concept, "quality goal". In order to distinct the new concept from a traditional goal, let us call the traditional goal a "necessity goal".

Necessity goal: a necessary condition required for the final state.

Quality goal: certain quality of solution to be optimized in the final state.

Figure 2(a) shows the syntax of the quality goal. The quality goal requires to find a solution whose e-function of variables in quality-predicates returns a value equal to or less than MIN. In other words, if a solution whose quality value (e-function's return value) is equal to or less than MIN is found, it is assumed to be an optimal solution. The default value of MIN is 0. In such cases, where it is impossible to find an optimal solution as resources are limited, a solution with the least quality value is said to be near-optimal.

For example, in case of the LSI design domain, quality goal is represented as Figure 2(b). The necessity goal directs to find a state where a circuit is mapped. A circuit is said to be "mapped" if all logical expressions ("unmapped-and2", "unmapped-or2" , and "unmapped-not") are mapped to gate-level expressions ("g-nand2", "g-nor-2", and "inverter") in the library. On the other hand, the quality goal directs to find a solution where the total-cell-num is equal to or less than 4.

Figure 3 shows an abstract algorithm to search for an optimal solution. The planner evaluates the status each time the necessity goal is satisfied, and continues its search until the quality goal is satisfied, i.e. a value less than min is returned, or resource bound is reached. On its search, the planner stores near-optimal solutions S_i(i=1,2,...) which satisfy the necessity-goal

1. S1 (initial solution) ← A solution which satisfies only necessity goal. Apply INTRA-SOLUTION-LEARNING to S1.
2. SNO (near-optimal solution) ← S1.
3. MAX ← e-function(SNO).
4. Until resource bound is reached or whole search-space is explored, search for Si which satisfies necessity goal and
   MAX > e-function(Si),
   4.1 if resource bound is reached or whole search-space is explored, return SNO, and FIN,
   4.2 else, apply INTER-SOLUTION-LEARNING to SNO and Si, if e-function(Si) is equal to or less than MIN (i.e. assumed optimal solution),
   yes: return Si, and FIN,
   no: SNO ← Si, and go to 3.

Figure 3: Search algorithm.

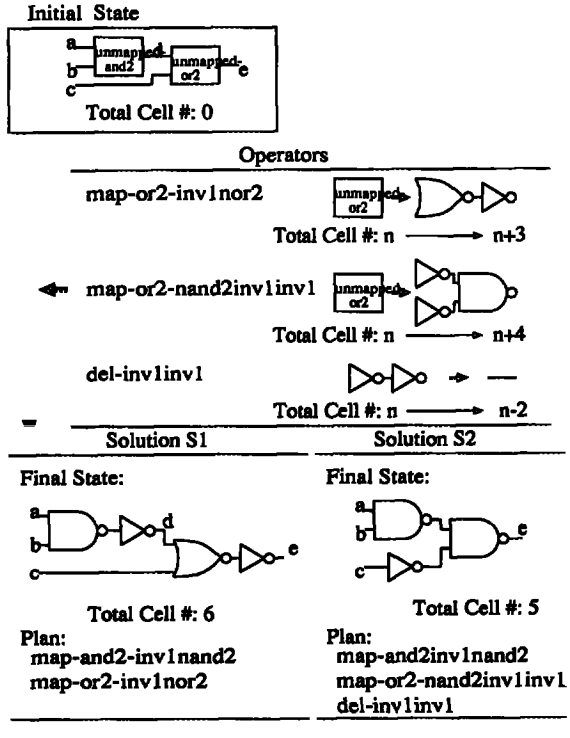

Figure 4: An example of technology mapping problem.

and e-function(S_i)>e-function(S_{i+1}). Resource bound is given as a number of node, i.e. state, the planner can visit in the search. As for the intra-solution learning and the inter-solution learning, we will explain in the next section.

When a problem illustrated in Figure 4 is given, solution S1 is found at step1. E-function(S1) is evaluated at step 3, and MAX is set at 6. Next, the system searches for a solution which satisfies e-function(Si) < 6. Then S2 is found. The difference between S1 and S2 is due to the difference of operator selection, "map-or2-inv1nor2" or "map-or2-nand2inv1inv1". In the solution S2, where the latter operator is selected, the total cell number reaches 7 after all logical expressions are mapped, but then it decreases to 5 by applying daemon[1] "del-inv1inv1", which deletes a sequence of two inverters.

Learning by Comparing Quality of Solutions

Traditional analytical learning methods such as PRODIGY/EBL(Minton *et al.* 1989) focused on one solution at a time, and then acquired search control knowledge by analyzing why goals/operators succeeded or failed using the trace result. Let us call this method an "intra-solution learning". Although it is quite effective, we propose an "inter-solution learning" method to complement the intra-solution learning. The former method learns control knowledge irrelevant to solution quality, whereas the latter learns control knowledge related to solution quality. The intra-solution learning is applied to the initial solution, and the inter-solution learning is applied to a couple of solutions whose quality values are different.

Inter-solution Learning

Control rules for optimal solution are learned by analyzing the reason that a certain case is a valid example of the following target concept with the following domain theory.

Target Concept:

```
(Operator-better op1 op2 necessity-goal
                 quality-goal node)
```

Domain Theory:

```
Domain theory in PRODIGY/EBL
  ∪
{(Operator-better op1 op2 necessity-goal
                  quality-goal node)
 if (AND (Operator-succeeds op1
                  necessity-goal node)
         (Operator-succeeds op2
                  necessity-goal node)
```

[1] Daemon is a kind of operators which fires whenever the precondition is satisfied even if it is irrelevant to the necessity goal. This function has been also added to PRODIGY.

```
(IF (and (current-goal goal)
         (candidate-operator op1)
         (candidate-operator op2)
         (operator-better op1 op2 goal
                          quality-goal node)))
(THEN prefer operator op1 op2)
```

Figure 5: Preference rule template.

```
                (> e-function(quality-goal op1)
                   e-function(quality-goal op2)))}
```

In addition to analyzing reasons that both operators, op1 and op2, of the case succeeded with respect to the necessity goal, inter-solution learning analyzes reasons that one of them reached a better state than another with respect to the quality goal. First, it back-propagates the weakest conditions excluding the conditions in terms of predicates in the quality goal (quality predicate). The reasons for this exclusion are that the quality predicates are irrelevant to the necessity goal and the absolute value of quality is irrelevant to deciding which quality value is smaller between two solutions. Then it generates a preference rule to guide a search to a better solution based on the constructed sufficient conditions of the target concept, applying them to the template shown in Figure 5.

The learned control rule is *weak* compared to selection rules and rejection rules learned by other PRODIGY learning components, because "quality" is relatively defined between two solutions. The learned rule is also *too generic*, because the preference should be inferred from quality value implied from all the literals of a state, e.g. all the components of a circuit, but not from literals relevant to the proof. We deal with this problem by constructing a generalization hierarchy of learned control rules and by giving precedence to the most specific control rule among matched control rules.

Example

This section gives an example of a learning process. Figure 6 shows a part of the search tree for the case where the problem illustrated in Figure 4 is given. The difference between two solutions, S1 and S2, is caused by selecting difference operators, operator map-or2-inv1nor2 or operator map-or2-nand2inv1inv1, in achieving goal (not (unmapped-or2 e)) at node18.

The conditions of every applied operator including daemon, e.g. map-or2-inv1nor2, map-or2-nand2inv1, and del-inv1inv1, are back-propagated from each leaf node, node21 and node54, to the node18 as indicated by the dotted arrows in Figure 6 except for the conditions related to the quality, e.g. total-cell-num. Then the weakest conditions where the proof will hold is acquired. Next, the control rule illustrated in Fig-

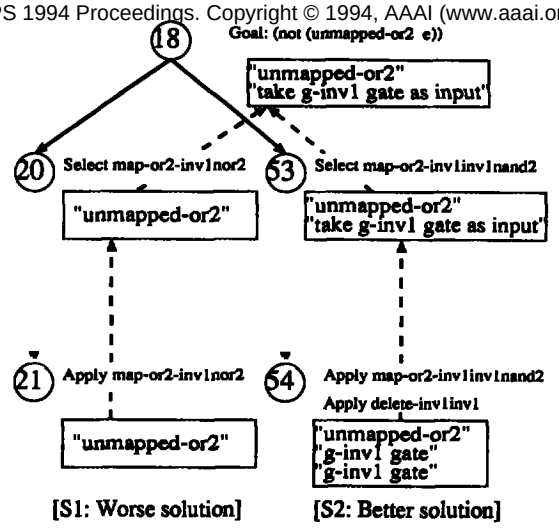

Figure 6: Learning Process.

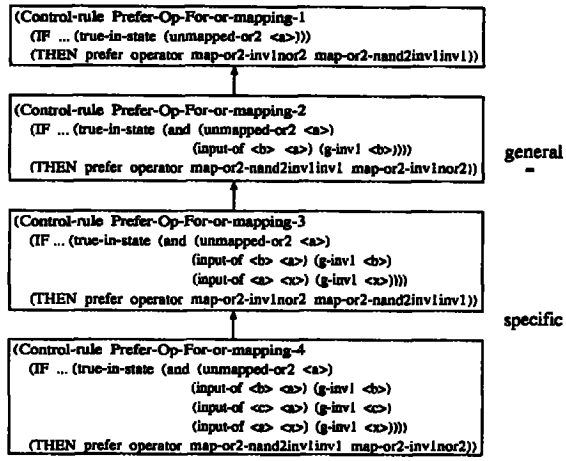

Figure 8: An example of control rule hierarchy.

```
(Control-Rule Prefer-Op-For-or-mapping-2
  (IF (and (current-goal
            (not (unmapped-or2 <a>)))
           (candidate-operator
                    map-or2-nand2inv1inv1)
           (candidate-operator
                     map-or2-inv1nor2)
           (true-in-state
            (and (unmapped-or2 <a>)
                 (input-of <b> <a>)
                 (g-inv1 <b>)))))
  (THEN prefer operator
        map-or2-nand2inv1inv1
        map-or2-inv1nor2))
```

Figure 7: Learned preference rule.

ure 7 is generated. The rule tells if an "unmapped-or" logic takes an output from an inverter as input, prefer operator map-or2-nand2inv1inv1 to operator map-or2-inv1nor2.

The precondition of learned control rule is compared with other control rules learned in advance, and control rule hierarchy is modified to include the newly learned control rule. Figure 8 shows an example of a part of the hierarchy. In this figure the upper rule is more general than the lower rule, e.g. Prefer-Op-For-or-mapping-1 is the most general, and Prefer-Op-For-or-mapping-2 is the most specific. If both Prefer-Op-For-or-mapping-1 and Prefer-Op-For-or-mapping-2 match to the current state, and Prefer-Op-For-or-mapping-3 does not match, the most specific rule among the matched control rules, Prefer-Op-For-or-mapping-2, is applied.

Experimentation

Twelve circuits, each of which takes *two* inputs, were given to the inter-solution learning system[2] as training examples. Then the design system was tested with randomly selected 30 circuits, each of which takes *three* inputs. At this test run, learning routine was suspended.

In the experiment, MIN was set to 4 (the least size of three input circuits), and the number of node the planner can visit, i.e. resource limit, was varied from 44 to 1100 as shown in Figure 9. Both the systems with/without the learned control rules found the initial solutions for all problems within 44 nodes. However, the average size of the circuits synthesized by the system with learned control rules was about 16% smaller than that without the rules. It took more than 1,000 nodes for the system without the learned rules to find the high-quality solution that the system with the control rules found as initial solution.

By consuming more than 1,100 nodes, both the systems with/without learning control rules could find optimal solutions for all problems, but it took about 10 minutes on SparcStation2 to search the space. The system with the learned rules could find the optimal solution as initial solution for 27 of 30 circuits, and the average CPU time to find the optimal solution was 32.5 sec. as shown in Figure 9. On the other hand, the system without the learned rules could find the optimal solutions as initial solution for 11 of 30 circuits, and the average CPU time to find the optimal solution was 124.9 sec.

Even after learning, two circuits required more than 1,000 nodes to find optimal solutions, since backtracks occurred at a few points where decisions are made on which part of circuits should be synthesized first. The

[2]Intra-solution learning components like PRODIGY/EBL have not yet implemented on the latest PRODIGY planner, PRODIGY4.0, we are using.

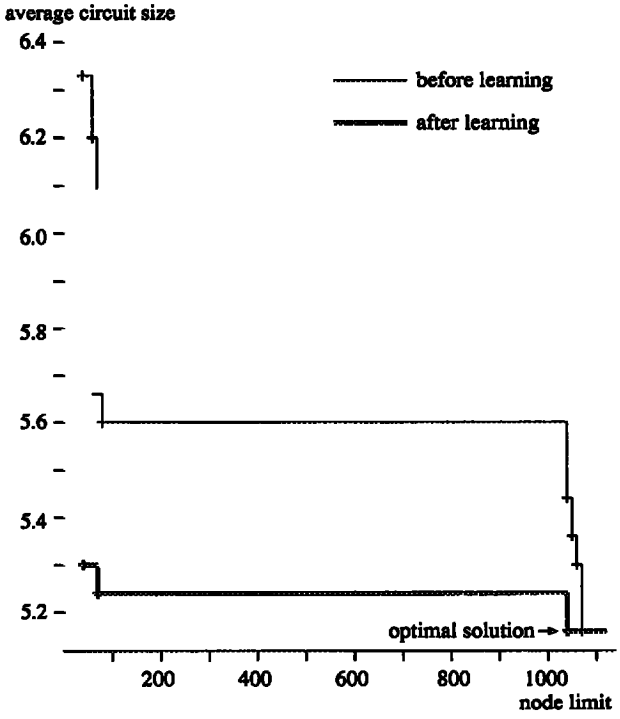

Figure 9: Solution quality when resource is limited.

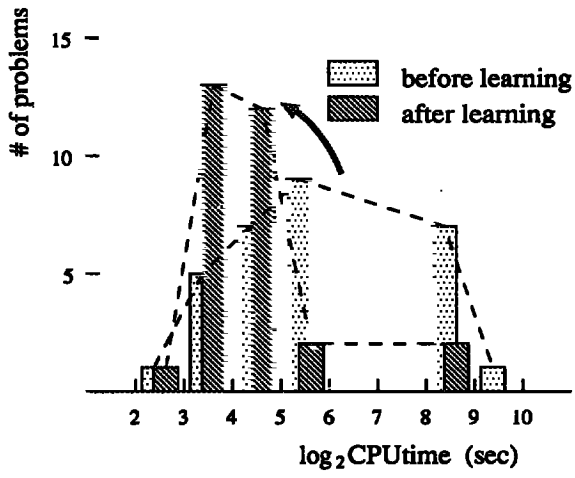

Figure 10: Time to find optimal solution.

backtracks will be reduced by learning goal control rules in addition to operator control rules.

Although the applied domain is not a toy, the problem tested are quite small. To apply the learning method to larger problems, i.e. real LSI design, we will have to solve utility problem, which is also a common problem among other learning methods.

Related Work

PRODIGY/EBL(Minton *et al.* 1989) is also capable of learning a rejection rule and/or a preference rule at the same decision point as the proposed system. For example, by assuming operator map-or2-nand2inv1inv1 (better operator) succeeded and operator map-or2-inv1nor2 (worse operator) failed, a control rule which reject map-or2-inv1nor2 can be learned. However, since it does not distinguish quality goal from necessity goal, it generates a overly specific and useless rule that includes conditions about absolute value of the quality, e.g. total-cell-num, at that decision point. Other PRODIGY learning components, STATIC(Etzioni 1991) and DYNAMIC(Pérez & Etzioni 1992), which analyze non-recursive parts of domains and ignore recursive parts, are incapable of learning the control rules that the proposed system learned. This is because of the domain is recursive, meaning quality predicate, e.g. "total-cell-num", occurs in the precondition and the postcondition of the identical operator.

FS2(Bhatnagar 1992), which searches forward and learns by building incomplete explanations of failures, performs well in recursive domains. The proposed system also searches forward for quality goal, and builds incomplete explanations since it learns control rules based on local optimality. But it is different from FS2 in such aspect as; (1)it learns rules for better solutions, (2)it learns preference rules in contrast to rejection rules, (3) it avoids over-generalization by making a control rule hierarchy in contrast to by specializing control rules when incorrect result is given, and (4) it still utilizes backward reasoning to find a solution which satisfies necessity goal.

(Minton *et al.* 1989) and (Gratch & DeJong 1992) pointed out that the utility of preference rules is fairly limited in such domains as STRIPS domain and blocks world domain. Our experiment presented a counterexample, and demonstrated the effectiveness of preference rules in a domain where solution quality is required. Gratch(Gratch, Chien, & DeJong 1993) proposed a learning method for optimization problem based on statistics. As our method are based on analysis of a single example, the learning cost is very low. But we may need to incorporate a kind of statistical approach to solve larger problems.

CPS/SCALE(Tong & Franklin 1989) improves circuit design performance by compiling initial knowledge base, and by giving priority to the most specific compiled rule. On the other hand, the proposed system improves the performance by learning control rules, and

is domain independent.

(Pérez & Carbonell 1994) also proposes a learning algorithm to improve the quality of a solution by explaining why one solution is better than the other. Although its algorithm is similar to ours, it defines the "quality" as a function of plan while our method defines it as a function of state, and the sufficient quality condition is explicitly defined as a goal.

Conclusions

We have proposed an extension of PRODIGY, that is, an extension of planner architecture and an application of EBL to optimization problems. To solve optimization problems a new type of goal, quality goal, was added, and a method which learns control rules for an optimal or near-optimal solution by comparing quality of solutions was incorporated. The proposed method was applied to a LSI design domain, a typical domain where optimality of the final state is required. The resultant system synthesized optimal circuits almost four times faster than that without control knowledge.

Acknowledgments

The main part of this work was done while the author was at Carnegie Mellon University from 1991 to 1992. We would like to thank Jaime Carbonell, Manuela Veloso, Yolanda Gil, Alicia Pérez, Xuemei Wang and other PRODIGY group members for their useful comments on earlier drafts of this paper. Thanks also to Satoru Fujita and Motohide Otsubo for several helpful discussions, Futaba Iwamoto for her help proofreading this paper, and Masahiro Yamamoto, Takeshi Yoshimura, and Masanobu Watanabe for providing the opportunity and environment for this work.

References

Bhatnagar, N. 1992. Learning by incomplete explanations of failures in recursive domains. In *Proc. of ML92*, 30–36.

Blythe, J.; Etzioni, O.; Gil, Y.; Joseph, R.; Kahn, D.; Knoblock, C.; Minton, S.; Pérez, A.; Reilly, S.; Veloso, M.; and Wang, X. 1992. PRODIGY4.0: The manual and tutorial. Technical Report 150, CMU-CS.

DeJong, G., and Mooney, R. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1:145–176.

Etzioni, O. 1991. STATIC: A problem-Space compiler for PRODIGY. In *Proc. of AAAI91*, 533–540.

Gratch, J., and DeJong, G. 1992. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proc. of AAAI92*, 235–240.

Gratch, J.; Chien, S.; and DeJong, G. 1993. Learning search control knowledge for deep space network scheduling. In *Proc. of ML93*, 135–142.

Minton, S.; Carbonell, J.; Knoblock, C.; Kuokka, D.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem solving perspective. *Artificial Intelligence* 40:63–118.

Mitchell, T.; Keller, R.; and Kedar-Cabelli, S. 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1:47–80.

Pérez, M. A., and Carbonell, J. G. 1994. Control knowledge to improve plan quality. In *Proc. of AIPS94*.

Pérez, M. A., and Etzioni, O. 1992. DYNAMIC: A new role for training problems in EBL. In *Proc. of ML92*, 367–372.

Tong, C., and Franklin, P. 1989. Tuning a knowledge base of refinement rules to create good circuit designs. In *Proc. of IJCAI89*, 1439–1445.