A Transformational Analysis of Expensive Chunks

Jihie Kim and Paul S. Rosenbloom

Information Sciences Institute and Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, U.S.A.
jihie@isi.edu, rosenbloom@isi.edu
(310) 822-1510 (x769)
Fax: (310) 823-6714

Key words: machine learning, utility problem, expensive chunks, Soar, EBL

Abstract

Many learning systems must confront the problem of run time after learning being greater than run time before learning. This utility problem has been a particular focus of research in explanation-based learning (EBL). This paper shows how the cost increase of a learned rule in an EBL system can be analyzed by characterizing the learning process as a sequence of transformations from a problem solving episode to a learned rule. The analysis of how the cost changes through the transformations can be a useful tool for revealing the sources of cost increase in the learning system. Once all of the sources are revealed, by avoiding these sources, the learned rule will never be expensive. That is, the cost of the learned rule will be bounded by the problem solving. We performed such a transformational analysis of chunking in Soar. The chunking process has been decomposed into a sequence of transformations from the problem solving to a chunk. By analyzing these transformations, we have identified a set of sources which can make the output chunk expensive.

1 Introduction

Many learning systems must confront the problem of run time after learning being greater than run time before learning. This *utility problem* has been a particular focus of research in explanation-based learning (EBL). There have been approaches which are useful for producing cheaper rules [1, 2, 3, 4, 5, 6] or filtering out expensive rules [2, 7, 8, 9]. However, these approaches cannot generally guarantee that the cost of using the learned rules will always be bounded by the cost of the problem solving from which they are learned, given the same situation. One way of finding a solution which can guarantee such cost boundness is to analyze all the sources of cost increase in the learning process and then eliminate these sources. Here we propose to approach this task by decomposing the learning process into a sequence of transformations that go from a problem solving episode, through a sequence of intermediate problem-solving/rule hybrids, to a learned rule. Analyses of these transformations then point out where extra cost is being added, and guide the proposal of alternatives that do not introduce such added costs.

The focus of the analysis in this paper is *chunking* in Soar[10]. Soar is an architecture that combines general problem solving abilities with a chunking mechanism that is a variant of explanation-based learning [11]. In the context of characterizing learning systems as a sequence of transformations, our prior work has revealed one source of added expensiveness: in chunking (and other EBL systems which use search control in the problem solving), *eliminating search control in learning can increase the cost of the learned rules* [12]. The critical consequence of the elimination of search control is that the learned rules are not constrained by the path actually taken in the problem space, and thus can perform an exponential amount of search even when the original problem-space search was highly directed (by the control rules). This analysis was based on one step (removal of search control) among the whole sequence of transformations. To reveal all sources of additional cost, we need a complete analysis of the whole sequence of transformations.

This approach is similar in spirit to [13] in its use of a transformational analysis of the learning algorithm. However, the focus of their analysis and resulting algorithm development was on speedup rather than on boundedness, and on search-control-free EBL rather than on chunking.

Section 2 of this article briefly reviews chunking in Soar. Section 3 then describes and analyzes the sequence of transformations underlying chunking. The key results of this analysis — in addition to the identification of the transformational sequence itself — are: (1) the identification of the points in the transformations at which extra cost is added; and (2) proposed modifications that may eliminate the identified sources of extra cost. Section 4 presents preliminary experimental results backing up the analysis in Section 3. Finally, Section 5 summarizes and discusses issues for future work.

Figure 1: An example of chunking process.

2 Background

In Soar, productions comprise the *domain theory* for EBL [14, 15]. Each production consists of a set of conditions and a set of actions. Conditions test working memory for the presence or absence of patterns of tuples, where each tuple consists of an object identifier, an attribute and a value. Actions create *preferences*, each of which specifies the relative or absolute worth of a value for an attribute of a given object. Productions in Soar propose changes to working memory through these preferences, and do not actually make the changes themselves. Changes to working memory are made based on a synthesis of the preferences (by a fixed *decision procedure*). The cycle of production firings, creation of preferences, and creation of working memory elements (WMEs) underlies the problem solving.

When a situation occurs so that a unique decision cannot be made because of either incomplete or inconsistent preferences, the system reaches an *impasse*. It creates a *subgoal* to deal with the impasse. In the subgoal created for the impasse, Soar tries to resolve the impasse. Whenever a supergoal object (called a *result*) is created in the subgoal, a new chunk is created. The chunk summarizes the problem solving (rule firings) that produced the result in the subgoal.

To create chunks, Soar maintains an instantiated trace of the rules which fired in the subgoal. The *operationality criterion* in chunking is that the conditions in the chunk should be generated from the supergoal objects. By extracting the part of the trace which participated in the result creation, the system collects the supergoal (operational) elements which are connected to the result. This process is called *backtracing*, and the instantiated trace is called a *backtrace*. It corresponds to the *proof tree* (or *explanation*) in EBL. The resulting supergoal elements are variabilized and reordered by a heuristic algorithm, and become the conditions of the chunk. The action of the chunk is the variabilization of the result. An example of chunking is shown schematically in Figure 1. The two striped vertical bars mark the beginning and the

Figure 2: Rete network of a rule.

end of the subgoal. The WMEs to the left of the first bar exist in the supergoal (prior to the creation of the subgoal). The objects (WMEs) between the two bars are internal to the subgoal. The object to the right of the second bar is the result of the subgoal. T1, T2, T3 and T4 are traces of the rule firings. For example, T1 records a rule firing which examined WMEs A and B and generated a preference suggesting WME G. The highlighted rule traces are those included in the backtrace; T2, T3, and T3 have participated in the result creation.

The chunking process can also be characterized in a different way — instead of simply considering it as a procedure which has problem solving episodes as input and learned rules (chunks) as output, it can be considered as a *sequence of transformations* from problem solving episodes, through intermediate *pseudo-chunks*, to chunks. The cost changes through the transformations can be estimated by analyzing each step.

Note that when we compare the cost of a problem solving episode to the cost of a chunk, by "cost" we will mean just the match cost of all of the rules that fired to generate the result (whether this be via multiple rules during the initial problem solving, or via a single chunk).[1] Because computing match cost is dependent on the match algorithm used, we briefly review the Rete algorithm [16] employed in Soar.

Rete is one of the most efficient rule-match algorithms presently known. Its efficiency stems primarily from two key optimizations: *sharing* and *state saving*. Sharing of common conditions in a production, or across a set of productions, reduces the number of tests performed during match. State saving preserves the previous (partial) matches for use in the future. Figure 2 illustrates a Rete network for a rule. Each WME consists of an object identifier, an attribute (indicated by an up-arrow(^)), and a value. Symbols enclosed

[1]Actually, the cost of a problem solving episode also includes the costs of firing rules and of making decisions. However, we will not explicitly focus on these factors here because they drop out during the transformational process.

Figure 3: A sequence of transformations from a problem solving to a chunk.

in angle brackets are variables. The conditions of the rule are compiled into a data flow network. The network has two parts. The alpha part performs constant tests on WMEs, such as tests for **at** and **yes**. The output of these tests are stored in alpha memories. Each alpha memory contains the set of WMEs which pass all of the constant tests of a condition (or more than one, if it is shared). The beta part of the network contains join nodes and beta memories.[2] Join nodes perform consistency tests on variables shared between conditions, such as <loc1>, which is shared between C1 and C2. Beta memories store partial instantiations of productions, that is, instantiations of initial subsequences of conditions. The partial instantiations are called *tokens*. Because match time per token is known to be approximately constant in Rete [17, 6] — and because counting tokens yields a measure that is independent of machines, optimizations, and implementation details — we will follow the standard practice established within the match-algorithm community and use the number of tokens, rather than time, as our comparative measure of match cost.

3 Transforming problem solving to a chunk

Figure 3 shows the sequence of transformations that convert a problem solv-

[2]There also are negative nodes, into which negative conditions are compiled. A negative node passes a partial instantiation when there are no consistent WMEs.

Figure 4: An example Soar task.

ing episode into a chunk. Each transformation (except for the last) creates an intermediate structure, called a *pseudo-chunk*. As the sequence progresses, the pseudo-chunks become more like chunks and less like problem solving. Each pseudo-chunk can itself be matched and fired (given an appropriate interpreter) and thus independently create the result. The cost of a pseudo-chunk can be determined by counting the number of tokens generated during the match to produce the result. By analyzing how the transformations alter these costs, the sources of added expensiveness are revealed.

The following subsections discuss each transformation shown in Figure 3, including their resulting (pseudo-)chunks and their effects on cost. These discussions are presented in the context of a simple illustrative task — that of evaluating which mode of transportation is best in particular situations (Figure 4). There are four rules and fifteen WMEs to begin with in this task. In the figure, a number in front of a rule condition denotes the number

Figure 5: Problem solving episode excluding unnecessary rule firings. This structure embodies PS-chunk.

of tokens generated by in the problem solving episode shown in Figure 5 by joining the tokens passed on from the previous conditions with the WMEs in the condition's alpha memory. Figure 5 shows how the sequence of rule firings during the problem solving episode creates the result (G1 ^object Q1 BEST)[3], given the rules and the WMEs. A connection from one rule to another rule through a decision means that preferences created by the former rule participated in the decision for the WME which is matched to a condition of the latter rule. The trivial decision steps — creation of one candidate and the following creation of a WME from the candidate — are not shown for brevity. Actual problem solving normally includes other rule firings which are not linked to the result creation; however, those are omitted here.

3.1 Filtering out unnecessary rule firings (⇒ PS-chunk)

As a first step toward producing a chunk, we can filter out the unnecessary rule firings which did not participate in the result creation. For the given example, this transformation eliminates all other rule firings, if there were any, beyond those shown in Figure 5. The resulting pseudo-chunk — called a *PS-chunk* (Problem-Solving-like chunk) — looks very similar to the original problem solving, aside from the missing unnecessary parts.[4] However, its

[3]This preference means that Q1 is the best alternative among the candidate values, given the identifier G1 and the attribute object.

[4]In addition, architectural actions that occurred during the problem solving episode are replaced in the PS-rule by dummy rules that have the same effect, much in the way that architectural axioms are used in Prodigy/EBL[2].

Figure 6: E-chunk: results from eliminating search control in the PS-chunk.

processing differs significantly from the initial problem solving by being closed off from intermediate WMEs generated outside of this structure.[5] For example, the link between R3 and R4 through W22 means that no other WMEs except for those created by R3 are matched to the condition of R4. The only parts of a PS-chunk that are exposed to the full set of WMEs are the conditions matched to the supergoal elements, and the result creation. The key difference between a PS-chunk and a normal chunk is that matching a PS-chunk requires replaying (part of) problem solving, while matching a normal chunk requires just one rule match. Either can create the result in a similar circumstance.

The cost (number of tokens) of a PS-chunk is bounded by the cost of problem solving. If there were unnecessary rule firings in the problem solving (as is usually the case), the cost of a PS-chunk is strictly less than the cost of the problem solving. If not, the cost is the same as the problem solving.

3.2 Removing search control (⇒ E-chunk)

PS-chunks contain all rules involved in the result creation. However, chunking employs only traces from *task-definition rules*; that is, rules that directly propose values of WMEs. *Search-control rules*, as distinguished from task-definition rules, suggest the relative worth of the proposed values. The search-control rules are missing in chunking [18, 10] (and other EBL systems [19]) based on the assumption that they only affect the efficiency, not the correctness of learned rules. This omission is intended to increase the generality of the learned rules — reducing the number of conditions by leaving out search control rules means less restriction on the test of applicability of the rules,

[5]It is not different in how it uses such optimizations as sharing and state saving; for example, the tokens from the first two conditions of R4 are shared with the tokens from the first two conditions of R3.

Figure 7: I-chunk: created by constraining variables (by instantiations) in an E-chunk. The structure remains the same as in the E-chunk (Figure 6) for this example.

and thus implies increased generality. An *E-chunk* (Explanation-structure-like chunk) is the intermediate structure which is formed by removing search control from a PS-chunk. Figure 6 shows the E-chunk created from the PS-chunk in Figure 5. The search-control rule R2 is gone, and all proposed candidates become WMEs without filtering through the search control in the decision process. This structure can be mapped onto the normal backtrace in chunking. The only difference between an E-chunk and a backtrace is that a backtrace consists of instantiations while an E-chunk consists of rules. By replacing the rules in the trace, a backtrace can be directly mapped to an E-chunk. An E-chunk is similar to an EBL explanation structure.

The consequence of eliminating search control is that the E-chunk is not constrained by the path actually taken in the problem space, and thus can perform an exponential amount of search even when the original problem-space search was highly directed (by the control rules), as analyzed in [12]. In the above example, without constraining eval-operator to the best candidate — which has priority 1 — the number of tokens in the match of rule R3 increases from 7 to 14. Overall, the total number of tokens increases from 17 to 20. This is thus one of the star-marked (i.e., cost increasing) transformations in Figure 3.

One promising way of avoiding this problem is to *incorporate search control in chunking*[12]. By incorporating search control in the explanation structure, the match process for the learned rule can focus on the path that was actually followed.

3.3 Constraining variables by instantiations(⇒I-chunk)

The variabilization step in chunking is performed by examining the backtrace (explanation). All constants are left alone; they are never replaced by variables. All object identifiers in the instantiations are replaced by variables; and in particular, all occurrences of the same identifiers are replaced by the same variable. Since E-chunks consist of rules rather than instantiations, we need to model chunking's variabilization step as the strengthening of constraints on the match rather than as the weakening of them. If a variable is instantiated as a constant, it is replaced by that constant. If a variable is instantiated by an identifier, it remains as a variable, but may possibly

Figure 8: U-chunk: results from eliminating intermediate rule firings in the I-chunk.

undergo a name change; in particular, all occurrences of variables which are instantiated by the same identifier are replaced by the same variable. For example, the variables in Figure 6 can be constrained as shown in Figure 7. The pseudo-chunk generated by this step is called an *I-chunk* (instantiation-based chunk).

This transformation can *overspecialize* learned rules when distinct variables in the original rules accidentally happen to match the same identifier; for example, although variable <g2> in R1 and variable <g3> in R4 (Figure 6) is instantiated by same the identifier G1, and changed to the same variable <g1>, they can correctly be generalized as different variables. However, from the perspective of cost, this transformation doesn't increase the number of tokens. The number of tokens generated should remain the same, or be reduced by the introduced constraints.

3.4 Eliminating intermediate rule firings (⇒ U-chunk)

This step unifies the separate rules in an I-chunk into a single rule, called a *U-chunk* (unified chunk). Figure 8 shows the result of unifying the example problem I-chunk into the corresponding U-chunk. Although R1, R3, and R4 still have their own identifiable conditions in the U-chunk, there are now no intermediate rule firings. The boundaries between the rules are eliminated by removing the intermediate processes of rule firing and WME creation. In lieu of these processes, the instantiations generated by matching the earlier rules in the firing sequence (i.e., the tokens produced by their final conditions) are passed directly to the match of the later rules. In effect, this step replaces the intermediate WMEs with the instantiations which created the WMEs. For example, one of R4's conditions receives the instantiations of R3 directly as intermediate tokens, rather than receiving WMEs created from the instantiations. Thus, R1, R3, and R4 are no longer (separate) rules.

Figure 9: Number of tokens can increase in a U-chunk.

To match U-chunks, an extension is required to the Rete algorithm. The traditional form of the algorithm, as shown in Figure 2, requires a *linear* match network, in the sense that a total ordering must be imposed on the conditions to be matched; such as C1, then C2, and then C3. In (linear) Rete, each join node checks the consistency of a token (a partial instantiation) and a WME, with each token itself being a sequence of WMEs, each of which matches one condition. However, U-chunks require the ability to perform *non-linear* matches, in which conditions are matched hierarchically via join nodes that compare pairs of tokens, rather than just a single token and a WME. They also require the ability to create hierarchically structured tokens (when pairs of incoming tokens are consistent); that is, a token must now be a sequence of WMEs or tokens (instantiations of a rule).

One benefit of going with U-chunks, rather than I-chunks, is that they enable equality tests across sub-structures which previously represented separate rules. For example, we can now test equality between the instantiations of <q1> in R1 and R3. However, cost problems are introduced in going to U-chunks because the number of instantiations of a rule can be greater than the number of WMEs created from those instantiations. For example, given the rule and WMEs in Figure 9-(a), two instantiations — (1 ^x 3) (3 ^y 5) and (1 ^x 4) (4 ^y 5) — are created. Because these two instantiations generate the same bindings for variables <a> and <c>, only one tuple (WME) is generated in the problem solving.[6] In this case, the number of tokens is increased after we replace the WMEs with the instantiations. This really happens in our example. While the two instantiations of R3 are collapsed into one WME and supplied to the fourth condition of R4 in the I-chunk, the two instantiations are directly used in the U-chunk, and create one more token. A worst case can arise when the working memory is structured as in Figure 9-(b). While the number of instantiations is exponential in the number of conditions, the number of WMEs is only one.

Our proposed solution to this problem is to *preprocess instantiations before they are used* so that the number of tokens passed from a substructure of a U-chunk is no greater than the number of WMEs passed in the corresponding

[6]Working memory is a set in Soar (and other Ops-like languages), and does not include duplicate elements.

Figure 11: Linearization can increase the number of tokens.

I-chunk. This could potentially be done either be grouping instantiations that generate the same WME or by selecting one of them as a representative.

3.5 Linearizing (⇒ Chunk)

A U-chunk can be linearized to become a chunk. The hierarchical structure of U-chunks is flattened into a single layer, and the conditions are totally ordered. For example, the non-linear structure in Figure 8 can be flattened to the structure in Figure 10. After the flattening, chunking uses a heuristic condition-ordering algorithm to further optimize the resulting match.

The linearization transformation turns out to introduce three ways in which match costs can increase. The first way arises directly from the flattening of the U-chunk's hierarchical structure. In a U-chunk, the conditions in a sub-hierarchy (e.g. the conditions in R1) are matched independently from the other parts of the structure before its created instantiations are joined with the others. By combining these sub-hierarchies together — through linearization — some of the previously independent conditions get joined with other parts

	Number of tokens
Problem Solving	52
PS-chunk	42
E-chunk	108
I-chunk	108
U-chunk	198
L-chunk	215

Table 1: Number of tokens of each step in a Grid Task.

of the structure before they finish their sub-hierarchy match. This change can increase the number of tokens. For example, after linearizing the U-chunk in Figure 11-(a), the number of tokens increases no matter what condition ordering is used. In the worst case, the increase can be exponential in the number of hierarchical levels in the U-chunk.

The second way arises from the impact flattening can have on sharing. As long as the implementation of Rete cannot capture the sharing from the non-linear structure (of the U-chunk), the number of tokens can increase. For example, in Figure 11-(b), sharing of sub-tokens from R1 for C2 and C3 in R2 cannot be realized in a linearized structure.

The third way arises because the heuristic condition-ordering algorithm cannot guarantee optimal orderings. Whenever this algorithm creates a non-optimal ordering, additional cost may be occurred.

Our proposed solution to this set of problems is to *eliminate the linearization step.* By keeping the hierarchical structure — that is, by replacing chunks with U-chunks — all three causes of cost increase can be avoided. The key thing that this requires is an efficient generalization of Rete for non-linear match.

4 Experimental Results

In order to supplement the abstract analysis provided in the previous section with experimental evidence, we have implemented a set of learning algorithms that correspond to the set of initial subsequences of the overall transformation sequence; that is, each learning algorithm in the set starts with the problem solving episode and generates a distinct type of (pseudo-)chunk. We have also implemented the extensions to the Rete algorithm necessary to allow all of the types of pseudo-chunks to match and fire. At each stage from problem solving to chunks, match cost is evaluated by counting the number of tokens required during the match to generate the result.

So far, the resulting experimental system has been applied to a simple Grid-task problem[6] which creates one subgoal to break a tie (impasse) among the candidate operators and creates a chunk. The results of this experiment are shown in table 1. The pattern of cost increases matches the expectations generated from the earlier analysis in that a transformation led to increased cost on this task if and only if it was identified by the analysis as a cost increasing transformation.

5 Summary and Future Work

We have performed an analysis of the chunking process as a sequence of transformations from a problem solving episode to a chunk. By analyzing these transformations, we have identified a set of sources which can make the output chunk expensive. We conjecture there are no other sources of cost increase, but cannot yet prove this.

Based on the above analysis and the proposed potential solutions to the sources of expensiveness, we are currently working towards the specification and implementation of a variant of chunking which does not introduce any of these sources. If it works, the cost of using a chunk should always be bounded by the cost of the corresponding problem solving.

A similar transformational analysis can also be performed for EBL. As with the analysis of chunking, this analysis should identify sources of expensiveness in EBL, and help guide the design of safer EBL mechanisms. In addition, a parallel analysis of EBL and chunking should further clarify the relationship between the two. An earlier comparison related the four basic structural components (goal concept, domain theory, training example, operationality criterion) of the two systems[11]; however, a transformational analysis should allow us to go beyond this to a deeper analysis of the processes underlying the two algorithms. A preliminary analysis of EBL shows that the sequence of transformations underlying EBL is very similar to that in chunking, except for the regression process, where chunking uses instantiations. In addition, EBL systems seem to suffer from cost problems similar to those that show up for chunking.

Acknowledgments

This research was supported under subcontract to the University of Southern California Information Sciences Institute from the University of Michigan as part of contract N00014-92-K-2015 from the Advanced Research Projects Agency (ARPA) and the Naval Research Laboratory (NRL). We would like to thank Milind Tambe and Bob Doorenbos for helpful comments on this work.

References

[1] A. E. Prieditis and J. Mostow. Prolearn: Towards a prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 494–498, 1987.

[2] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569, 1988.

[3] P. Shell and J. Carbonell. Empirical and analytical performance of iterative operators. In *The 13th Annual Conference of The Cognitive Science Society*, pages 898–902. Lawrence Erlbaum Associates, 1991.

[4] Jude W. Shavlik. Aquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–70, 1990.

[5] O. Etzioni. Why prodigy/ebl works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 916–922, 1990.

[6] M. Tambe. *Eliminating combinatorics from production match*. PhD thesis, Carnegie-Mellon University, 1991.

[7] R. Greiner and I. Jurisica. A statistical approach to solving the ebl utility problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 241–248, 1992.

[8] J. Gratch and G. Dejong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Aritificial Intelligence*, pages 235–240, 1992.

[9] S. Markovitch and P. D. Scott. Information filtering : Selection mechanism in learning systems. *Machine Learning*, 10(2):113–151, 1993.

[10] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325, 1991.

[11] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, 1986. AAAI.

[12] J. Kim and P. S. Rosenbloom. Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 174–181, 1993.

[13] A. Segre and C. Elkan. A high-performance explanation-based learning algorithm. *Artificial Intelligence*, 69:1–50, 1994.

[14] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization – a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[15] G. F. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[16] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[17] M. Tambe, D. Kalp, A. Gupta, C. L. Forgy, B. G. Milnes, and A. Newell. Soar/psm-e: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146–160, 1988.

[18] J. E. Laird, P. S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57, 1986.

[19] S. Minton. Personal communication. 1993.