Lazy Partial Evaluation: An Integration of Explanation-Based Generalisation and Partial Evaluation

Peter Clark and Rob Holte
Ottawa Machine Learning Group
Computer Science, University of Ottawa
Ontario, CANADA K1N 6N5
{pclark,holte}@csi.uottawa.ca

Abstract

In this paper we present lazy partial evaluation (LPE), a new learning technique which is a hybrid of explanation-based generalisation (EBG) and partial evaluation (PE). LPE operates in a similar way to EBG, except that the work performed exploring failed proofs is also generalised and stored. The equivalence of the set of explored proofs to the original goal definition allows LPE to replace (rather than augment) the original definition with them, speeding up proof search for future examples. The resulting learning algorithm is significantly less computationally expensive than EBG, while also avoiding the potentially vast memory requirements of PE. It also removes the undesirable bias which EBG introduces, where EBG's preference for reusing operational proofs may result in a 'poor' proof being selected. We describe LPE and compare its performance with PE and EBG on two constraint satisfaction tasks. Finally, we analyse the conditions in which each of the learning techniques is most effective.

1 Introduction

Explanation-based generalisation (EBG) (Mitchell, Keller & Kedar-Cabelli, 1986) has become an established problem-solving technique, whose aim is to speed-up problem-solving by recording and reusing generalised solutions to previous problems It can be seen as an *example-based compilation technique* in which efficient or 'operational' versions of parts of a domain theory are synthesised and stored, examples being used to determine which parts of the theory to operationalise. It operates by proving a goal (eg. concept membership) for an example, and then synthesising an operational definition of the goal by collecting and generalising the operational subgoals in that proof. This new definition is added to the domain theory. While the new definition is only a reformulation rather than an extension of the original definition, it can be faster to evaluate.

EBG has been used extensively in many learning systems eg. Prodigy (Minton et al., 1987) and Soar (Laird, Rosenbloom & Newell, 1986), and sophisticated techniques for its application have been developed. However, our concern in this paper is with two limitations of the basic mechanism itself:

Repeated Computation: EBG can be a computationally expensive process – it explores the space of possible proofs to find one which applies to a training example. Although EBG learns an operational definition of the concept derived from a successful proof (ie. one which applies to the example), the work done exploring other proofs which turn out to be inapplicable to the example is lost. Given a new example for which the learned operational definitions do not apply, this search must be repeated again from scratch.

The Masking Effect: By preferring to reuse learned, operational definitions, an EBG system may miss alternative proofs of concept membership which it could have found by searching the original domain theory. When there is some cost associated with different proofs, an EBG system may miss a 'better' proof by preferring to reuse a learned proof. For example in the planning domain, where proofs correspond to plans, EBG may miss a shorter plan if an operational definition corresponding to a longer plan applies first. Here, learned knowledge has partially hidden or 'masked' the original domain knowledge, and obscured opportunities for further learning. Note that this problem is distinct from the utility problem of EBG, and concerns the 'quality' of the solution found rather than the time taken to find it.

An alternative to EBG is partial evaluation (PE), closely related to EBG (van Harmelen & Bundy, 1988, Prieditis, 1988). Rather than augmenting a domain

theory with operational versions of concept definitions, PE *replaces* a concept definition with all possible operational definitions of that concept. This is done in a once-and-for-all, preprocessing phase before problem-solving proper commences. The resulting domain theory is larger but may also be more efficient. It overcomes both of the above problems: first, the exploration of the original space of proofs to find (all) operational definitions is performed only once – in contrast, EBG must repeat this exploration from scratch each time learned, operational definitions do not apply to an example. Second, because PE makes the operational definitions in the domain explicit, these can then be ordered by their associated cost/quality to ensure that the 'best' is found first for any example. Despite these advantages, PE also has associated problems:

Space Requirement: The number of operational definitions can be very large, even infinite, and hence too costly in terms of memory to store.

Unnecessary Computation: Many of the definitions created by PE may not be required in practice and hence an unnecessary computational cost has been incurred in generating them.

To address the preceding problems, we have developed **lazy partial evaluation** (LPE), a hybrid of EBG and PE, which we now present. Following this, we compare LPE's performance with EBG and PE on two constraint satisfaction problems, and discuss the conditions in which each technique is the most effective learner.

2 Lazy Partial Evaluation

2.1 Definitions

We adopt the normal terminology of EBG as follows. A concept definition consists of a set of goals to be proved of an example, which, if proved, imply the example is a member of that concept. A domain theory consists of facts and rules (expressing how goals are defined in terms of subgoals and facts). Some goals are labelled as being operational. An operational concept definition is one expressed in terms of operational goals. Proof of concept membership involves repeatedly replacing non-operational goals in a concept definition with subgoals (which may still be non-operational), using rules in the domain theory. We term this process **(goal) expansion**.

2.2 The LPE Algorithm

Given an example, LPE behaves in a similar way to EBG: both techniques search for a proof of concept membership, then generalise the proof, collect the leaves of the proof to create a new concept definition, and save the new definition for future use. However LPE differs from EBG in that LPE also converts into concept definitions all the partially explored proofs which turned out not to apply to the example. In this way, none of the work done searching for proofs is wasted. After finding a successful proof, LPE (and EBG) can either continue to search for other (maybe 'better') proofs or simply abandon the remaining, unexplored lines of proof by treating them as if they had also failed (ie. generating concept definitions from them without expanding them further). It is important to note that LPE traverses the space of possible proofs in no more detail than EBG would. Because LPE generates definitions from all the traversed branches, the new definitions constitute a reformulation of the original concept definition which is *complete*, ie. capable of proving every example provable by the original domain theory and no more. Therefore, LPE can *replace* (rather than EBG's augmenting) the original concept definition with these simpler definitions, reducing the work required to find proofs for future examples. This is a key source of LPE's power. The result of applying LPE to a sequence of examples is a revised domain theory, partially evaluated *only as much as is necessary* to prove or refute the concept membership of the given examples (hence the name 'lazy partial evaluation'). If all parts of a domain theory are not required to determine the concept membership of the given examples, LPE will be both faster and more space efficient than PE.

The LPE algorithm is given in Figure 1 and in Prolog in Appendix A. It is similar to EBG, except that the failure of an operational goal no longer causes this line of proof to be abandoned (and another searched for). Instead, LPE returns a flag (`SatistfiedByExample`, set to **false**) which stops further goal expansion along this particular line of proof. The **foreach...while** clause (line 4 of Figure 1) implements this. Note that this can be converted easily into EBG and PE.

2.3 Example

As an example, consider the following domain theory defining the concept of a 'tiger':

```
tiger(X) :- striped(X), cat_family(X).

cat_family(X) :- carnivore(X), tail(X).
cat_family(X) :- fast_runner(X).

carnivore(X) :- eats_meat(X).
carnivore(X) :- teeth(X), mammal(X).

mammal(X) :- hairy(X).
mammal(X) :- gives_milk(X).
mammal(X) :- warm_blood(X).
```

Predicates describing basic properties of the animal are operational (ie. `striped`, `hairy`, `gives_milk`, `warm_blood`, `eats_meat`, `teeth`, `tail`, `fast_runner`). Given a training example, `joe`, described by the fol-

procedure LPE(Goals, Example) returning ProofLeaves and SatisfiedByExample:

```
1    let    SatisfiedByExample := true.
2    if     more than one Goal in Goals
3    then   let ProofLeaves := Goals.
4           foreach Goal_i in Goals while SatisfiedByExample = true:
5                expand Goal_i using LPE(Goal_i, Example) into ProofLeaves_i
6                                  and SatisfiedByExample_i,
7                replace Goal_i in ProofLeaves with ProofLeaves_i, and
8                let SatisfiedByExample := SatisfiedByExample_i.
9           endforeach.
10          return ProofLeaves and SatisfiedByExample.
11   elseif Goal is operational
12   then   if     Goal is satisfied by example[a]
13          then return Goal and true else return Goal and false.[b]
14   else   find a set of subgoals SubGoals which achieve Goal,
15          call LPE(SubGoals, Example) to find ProofLeaves and
16                           SatisfiedByExample, and
17          return ProofLeaves and SatisfiedByExample.
```

Notes for the above algorithm:

Goals is the concept to prove, Example is a training example, ProofLeaves is a reformulated definition of the concept (comprising of the leaves of the proof explored), and SatisfiedByExample is a flag, either **true**/**false**, indicating whether Example satisfies ProofLeaves or not.

This procedure explores just one line of proof: to explore all lines of proof, as LPE requires, standard backtracking techniques are used. The point of non-determinacy to which backtracking returns is line 14 ('**find** a set of subgoals...'), which may have 0 or more solutions (one for each domain theory rule concluding Goal).

[a] (lines 12 & 13) replacing this '**if-then-else**' test with '**return** Goal and **true**' converts LPE into PE.

[b] (line 13) replacing "**return** Goal and **false**." with "**fail**." (ie. backtrack) converts LPE to EBG.

Figure 1: The LPE Algorithm.

lowing:

```
gives_milk(joe).      eats_meat(joe).
tail(joe).            striped(joe).
```

the successful proof of concept membership is:

```
tiger(joe) :-
   striped(joe),
   cat_family(joe) :-
      carnivore(joe) :-
         eats_meat(joe).
      tail(joe).
```

In addition, there are two failed proofs:

```
tiger(joe) :-
   striped(joe),
   cat_family(joe) :-
      carnivore(joe) :-
        teeth(joe),       <=== FAILS!
        mammal(joe).
      tail(joe).

tiger(joe) :-
   striped(joe),
   cat_family(joe) :-
      fast_runner(joe).   <=== FAILS!
```

The following concept definitions result (for PE, the example is not needed):

EBG

```
tiger(X):-striped(X),eats_meat(X),tail(X).
tiger(X):-striped(X),cat_family(X).
```

PE

```
tiger(X):-striped(X),eats_meat(X),tail(X).
tiger(X):-striped(X),teeth(X),hairy(X),tail(X).
tiger(X):-striped(X),teeth(X),gives_milk(X),tail(X).
tiger(X):-striped(X),teeth(X),warm_blood(X),tail(X).
tiger(X):-striped(X),fast_runner(X).
```

LPE

```
tiger(X):-striped(X),eats_meat(X),tail(X).
tiger(X):-striped(X),teeth(X),mammal(X),tail(X).
tiger(X):-striped(X),fast_runner(X).
```

For EBG, the single reformulation:

```
tiger(X):-striped(X),eats_meat(X),tail(X).
```

is extracted from the successful proof and added to the additional definition. In contrast, LPE also stores two

other definitions:

```
striped(X),teeth(X),mammal(X),tail(X)
striped(X),fast_runner(X)
```

constructed from the two (generalised) failed proofs. Unlike PE, LPE stops further exploration of proofs as soon as it is known that they do not apply to the training example. Hence LPE does not waste effort expanding `mammal(joe)`, because it is already known this line of proof is not needed for the example.

Note that LPE (and PE) have both replaced the original concept definition with a set of simpler equivalents, whereas EBG has augmented the original definition.

2.4 Avoiding the Masking Effect

The masking effect occurs when a system adopts a previously learned solution to a problem in preference to searching for a new solution, and, in doing so, misses a 'better' solution. We illustrate later how this effect can seriously degrade performance of EBG.

It should be carefully noted that this problem is distinct from the utility problem of EBG (Minton, 1988). The utility problem occurs when the cost (in speed) of testing a concept definition outweights the (speed) benefit it provides when it applies, and is a problem for all compilation learning techniques. The masking problem, however, concerns the 'quality' of the solution found (independent of the time taken to find it). It only occurs when a proof provides additional information besides yes/no concept membership, for example when the proof corresponds to a plan for use by a planner. The problem occurs when a proof corresponding to a 'bad' solution (eg. a long plan) is adopted when a better solution (eg. a shorter plan) exists. If all solutions are equally 'good', then the masking effect does not occur. In the experiments described in the next section, the masking effect was by far the dominating problem. Further work is required to assess the utility issue for LPE in more detail.

LPE, like PE, can avoid the masking effect by ordering the (original and learned) concept definitions by an upper bound on their associated quality. Unlike EBG, where learned definitions are tried before resorting to the original domain knowledge, LPE interleaves original and learned definitions together[1]. Given a problem to solve (ie. an example to establish concept membership of), each definition is tried in turn. If the definition is operational and the example satisfies it, we exit with success. If the definition is non-operational and is expanded as we try and prove the example satisfies it, then new concept definitions are generated from the successful and failed proofs as described in Section 2.2.

[1] We assume that the domain theory can be reordered arbitrarily without invalidating it. In Prolog, this requires the domain theory to be 'pure' Prolog (no cut).

The old definition is then deleted, the new definitions added (positioned according to their qualities in the list of definitions) and the search continues. In this way LPE gradually replaces the original definitions with repeatedly simpler, and eventually operational, definitions as demanded by the problems encountered. This concept definition ordering algorithm is shown in Appendix B.

3 Application to Constraint Satisfaction

3.1 Introduction

In this section we evaluate LPE's application to constraint satisfaction problems (CSPs). A basic operation of all constraint satisfaction algorithms (CSAs) is to test whether a hypothesis satisfies a constraint. We wish to speed up this operation using some compilation learning technique. Note that we are not presenting a new CSA, but instead wish to improve the performance of existing CSAs by speeding up constraint testing by learning.

Other researchers have developed alternative learning techniques specifically for CSPs. For example, Eskey and Zweben (1990) have developed plausible explanation-based learning (PEBL) for learning improved search heuristics and Dechter (1990) has developed several learning techniques for reducing search of constraint graphs. These projects aim to improve the CSAs themselves by reducing the number of constraint tests they need perform – they are thus complementary to our goal, which is to work within some CSA to speed up the basic operation of constraint testing itself through learning.

3.2 Definitions

CSPs can be formulated as a search for an assignment of *values* to *variables* such that a set of *constraints* between variables is satisfied. A *hypothesis* is an assignment of values to all variables, and a *solution* is a hypothesis which satisfies the constraints.

3.3 Speeding up Constraint Tests

To apply constraints in CSAs, a (typically small) program is required to test whether the constraints are satisfied or not by a hypothesis. This program constitutes a computational definition of the constraints, and is the "domain theory" for CSPs. Executing this program tests whether the constraints hold on a hypothesis. It is the execution of parts of this program we wish to speed up by 'operationalising' the constraint definitions.

However, for efficient constraint satisfaction, we require more than simply knowledge of whether a hy-

pothesis fails the constraints – we also need to know *why* the failure occurred, ie. which variables were involved in the failure. This allows us to return immediately to an assignment responsible for a failure, skipping over irrelevant variable assignments. This form of dependency-directed backtracking is called *backjumping* (Gaschnig, 1979).

In order to explain why a hypothesis fails some constraints, it is useful to re-express the constraints as conditions for failure. (eg. from Appendix C, constraint: "don't visit berlin more than once" becomes failure-test: "visit berlin more than once"). This 'trick' was used by Mostow and Bhatnagar (1987) in their system Failsafe to allow EBG to be applied. By using negated constraints, we can now produce a proof or 'explanation' for why the hypothesis failed a constraint (the target concept is "a failed hypothesis"), which also returns the variables involved in the failure. A 'good' explanation is one which blames the earlier variable assignments made, as this allows greater backjumping by the CSA. Hence there is a measure of 'quality' associated with each definition, and the possibility of a masking effect occurring.

As an example, consider the 'cities' CSP (Appendix C) used in the experiments later. The predicate `not_okay` encodes the ten (negated) constraints, taking as input a hypothesis `H` and succeeding if `H` fails a constraint. If so, it also returns the variables involved in that failure. If more than one constraint is violated, there will be more than one solution. Consider that the CSA has generated the hypothesis route `oslo` → `berlin` → `paris` → `berlin` → `athens` → `paris` → `oslo`. To see if the hypothesis (represented as a list of the seven values for the seven variables) fails a constraint, we call (using LPE):

```
lpe_call(   not_okay([o,b,p,b,a,p,o], Vars)    )
```

One solution to this is to return `Vars` = `[var2,var4]`, representing that the values of `var2` and `var4` violate a constraint (berlin cannot be visited more than once). It is the computation of `not_okay` which we wish LPE to speed up. In addition we wish LPE to find the 'best' solution, ie. the one blaming the earliest variables, allowing the CSA to backjump furthest.

To illustrate LPE on this example, the (generalised) successful proof using the original domain theory looks:

```
not_okay(H, [var2,var4]) :-     % Hypo fails if...
  visited_more_than_once(b,H) :-
    visited(b,H,day2,var2) :-   % at berlin on day2
      denotes(var2,day2),       % (var2 is day2, &
      valof(var2,H,b),          % var2 has value b)
    visited(b,H,day4,var4) :-   % and at b on day4
      denotes(var4,day4),
      valof(var4,H,b),
    day2 \= day4.               % & different days
```

An operational definition of `not_okay` is derived from this (just as in EBG) by collecting the operational subgoals (`valof` is the only operational predicate):

```
not_okay(H, [var2,var4]) :-
     valof(var2,H,b), valof(var4,H,b).
```

In addition, LPE generates six other definitions for this constraint from the six other failed proofs explored. The first of these definitions is:

```
not_okay(H, [var1,VarX]):-
   valof(var1,H,b), visited(b,H,D,VarX), D\=day1.
```

("if berlin is visited on day1 and some other day then `H` is `not_okay`"). Note that `visited(b,H,Day,VarX)` is not expanded further as we already know the definition does not apply to our example (the preceding goal `valof` failed). As described earlier, the equivalence of these simpler definitions to the original definition allows LPE to replace the original definition with them, speeding up proof search for future examples.

Definitions are ordered according to the constraint variables which they explicitly name, the earlier the latest variable they mention being considered 'better' (allowing greater backjumping). Ties are broken by looking at the next latest variable they mention. Some non-operational definitions may not have been expanded enough to explicitly name all the constraint variables they involve (eg. `[var1,VarX]` below). These are placed according to those which they do explicitly mention, if any (eg. `var1`). Thus the following definitions are ordered (variables starting with an upper case letter, '...' denoting the definition body):

```
not_okay(H, Vars) :- ...
not_okay(H, [var1,VarX]) :- ...
not_okay(H, [var2]) :- ...
not_okay(H, [var1,var4]) :- ...
not_okay(H, [var2,var4]) :- ...
```

For a non-operational definition, given a new example to prove, it will either not apply, or be LPE expanded, the expanded definitions inserted below it in their appropriate places, the definition deleted, and the search for a definition continued. This ensures that the first operational definition which is located (if any) is the 'best', avoiding the masking effect (Section 2.4 and Appendix B).

Note that many CSAs assume all pairwise (or n-wise) constraints are explicitly known, eg. by applying full PE to identify them from the original domain theory. Here, however, we do not make this assumption; in fact our concern is exactly with identifying possible sets of conflicting variable assignments from the initial constraint definitions.

Cities	No Learning	EBG	PE	LPE
hypotheses generated:	165	1029	165	165
final no. concept defs:	10	47	1011	314
goal expansions:	54223	14639	10886	1520
cpu time (sec):	331.9	92.2	78.1	33.6
Zebras	**No Learning**	**EBG**	**PE**	**LPE**
hypotheses generated:	7626	19433	7626	7626
final no. concept defs:	15	179	818	806
goal expansions:	$o(2 \times 10^7)$	453812	3782	3599
cpu time (sec):	$o(4 \times 10^5)$	13722.1	111.5	155.4

Table 1: Comparative performance of algorithms in two CSPs

3.4 Comparative Experiments

3.4.1 Algorithms

To evaluate LPE, we compare it with three other approaches to constraint testing:

No learning: For each hypothesis the CSA produces, generate *all* explanations of its failure from scratch using the domain theory, and select the 'best' (ie. blaming earliest variables).

EBG: In our experiments, EBG was applied (when no operational definition succeeded) by searching the domain theory for the best (rather than the first) explanation of failure and the new definition added at the end (rather than the start) of the list of operational definitions. These two decisions reduced the masking effect and produced the best results for EBG.

PE: The operational definitions produced by PE are ordered according to their quality (blaming earliest variables), to ensure the first definition which applies is the 'best'.

3.4.2 Method and Evaluation Criteria

We compared these methods using a particular CSA called 'generalised backjumping' (Clark & Holte, 1992), an enhancement of Gaschnig's (1979) backjumping algorithm. This algorithm is a simple but effective tree-searcher. A hypothesis is gradually constructed by assigning values to variables until a constraint fails: at this point, the algorithm immediately backtracks ('backjumps') to redo the assignment of the most recent variable involved in the failure (ie. skipping over irrelevant variables). For present purposes, its key feature is that finding the earliest cause of failure ('the most useful proof of failure') results in the biggest backjumps and hence the smallest number of hypotheses which need to be tested. It should be noted that the above methods could also have been applied with other CSAs.

The methods were compared on two constraint satisfaction problems: the 'cities' problem (designed by us for this comparison) and the well-known 'zebras' problem, described in Appendices C and D. The two problems were chosen to explore the strengths and weaknesses of LPE. The 'cities' problem offers several opportunities for being 'lazy', arising when the constraints do not need to be fully expanded to solve the problem. Conversely, the famous 'zebra' problem requires almost all the constraints in full.

To compare the methods, we measure the number of hypotheses the CSA algorithm generates, the total number of final concept definitions, the number of expansions of non-operational goals each algorithm performs, and the total CPU time used for constraint testing (ie. ignoring the overhead of the CSA itself).

3.5 Results and Analysis

The results are shown in Table 1. We first summarise the main conclusions: in the 'cities' CSP, substantially less CPU time is required to test the constraints with LPE than with either EBG and PE. This is a significant result, showing that example-based learning can outperform PE, and at the same time also have a reduced memory requirement. In the 'zebras' problem, LPE does the fewest goal expansions, but requires more CPU time than PE. On this problem, EBG is extremely inefficient. We now discuss the results in more detail.

3.5.1 Hypotheses Generated

Each hypothesis generated by the CSA is a training example for which membership of the concept 'fails a constraint' is to be proved. For example in 'cities', 164 generated hypotheses failed a constraint, hence caused backjumping, before the solution (hypothesis 165) was found.

No learning: For each hypothesis, the domain theory is searched to find the 'best' proof of failure (ie. identifying the earliest choice point the CSA can backjump to). Thus the number of hypotheses tested here is *optimal* (ie. minimal): at no point could the CSA have backjumped further

given the domain theory provided. Recall that our aim is not to reduce the number of hypotheses tested, but to speed up those tests.

PE and LPE: As both of these algorithms completely avoid the masking effect (Section 2.4), the 'best' proofs of failure are always found, hence the number of hypotheses generated is minimal.

EBG: Here the masking effect is evident. EBG's preference for using its learned, operational definitions meant that sometimes a 'better' explanation for failure was missed. As a result, the CSA did not backjump as far as it could have, and many hypotheses were unnecessarily generated.

It is worth noting that some of the 'good' explanations which were missed by EBG earlier in the search are sometimes learned later, when hypotheses arise for which earlier explanations do not apply. To see this, if we re-run the CSA retaining EBG's learned definitions, only 241 (cities) and 14965 (zebras) hypotheses are generated.

3.5.2 Final Number of Concept Definitions

We compare how each algorithm extends/reformulates the original concept definitions:

No learning: The 10 original definitions for the 'cities' problem (15 for zebras) remain unchanged after executing the CSA.

EBG: For 'cities', EBG ends up with 47 definitions, namely the 10 in the original domain theory plus 37 learned, operational definitions. These 37 are adequate to explain all the 1029 hypotheses, which include the 165 hypotheses generated by the other algorithms. For 'zebras', EBG adds 164 definitions to the original domain theory.

LPE: For 'cities', LPE has reformulated the 10 original definitions as 314 definitions. These include the 37 EBG definitions, plus 277 others. 121 of these 277 were operational, corresponding to 'good' explanations of hypotheses that EBG missed (due to masking), and to explanations that apply to none of the hypotheses, but which had to be expanded by LPE to be fully operational before their inapplicability could be recognised. The remaining 156 were non-operational, corresponding to branches of proofs which were not fully expanded. For 'zebras', only 4 of the 806 learned definitions were non-operational, showing there is little opportunity for 'being lazy' in this domain.

PE: The 1011 (cities) definitions include the 158 operational definitions of LPE, plus the full expansion of LPE's remaining 156 definitions. This figure represents an upper bound on the number of definitions which EBG and LPE can produce.

3.5.3 Goal Expansions

Here, we count the number of times a non-operational goal in a concept definition is replaced by its subgoals (which may still non-operational). This action is performed to prove concept membership (no learning), to operationalise the domain theory (PE), or both (EBG, LPE). This is the factor which learning is supposed to reduce.

As in Section 1, we use the adjective 'repeated' to refer to any goal expansion that is performed more than once and 'unnecessary' to refer to any goal expansion performed but not actually required to solve the given problems.

No learning: Here the *maximum* number of goal expansions needed for the examples is performed, because the domain theory is used from scratch each time. Thus a substantial amount of computation is repeated.

PE: PE does all possible goal expansions once and only once. However, although no expansions are repeated, many may be unnecessary if the resulting definitions are not needed for the examples encountered in practice.

LPE: Performs the *minimum* number of goal expansions needed for the examples seen. No repeated or unnecessary expansions are performed, and hence this figure will be always less (or equal to) those for no learning and PE.

EBG: EBG, like LPE, also avoids unnecessary expansions. However, for examples where a learned operational definition does not apply, the original domain theory is searched from scratch to find a proof of membership, repeating expansions performed for similar examples earlier. Hence the number of goal expansions for EBG will be between that for LPE and no learning.

3.5.4 CPU Time

The CPU times refer to the time spent proving concept membership for the hypotheses generated by the CSA ie. the (small) overheads of the CSA itself have been subtracted.

No Learning: The CPU time is high because the search for an explanation of failure needs to be generated from scratch for each hypothesis.

EBG: While EBG is fast to test examples for which a learned, operational definition applies, it pays a high time penalty when no learned definitions apply and a proof has to be generated from scratch.

PE, LPE: One would expect LPE to always be faster than PE, because it generates exactly the same examples, has fewer concept definitions to try, and avoids unnecessary goal expansions. However, LPE differs from PE in a way that affects

the *speed of implementation* of the reformulated domain theory. PE does all learning at the outset and therefore the PE'd domain theory can be optimised for execution. LPE, on the other hand, interleaves the execution and reformulation of the domain theory, and hence must store definitions in a data structure supporting arbitrary insertion and deletion operations (Section 2.4). This makes execution slower[2] to evaluate, and hence incurs a time penalty that LPE's benefits (reduced number of definitions and goal expansions) must overcome to make LPE worthwhile.

4 Discussion

The most important finding of our analysis and evaluation is that LPE can significantly outperform both EBG and PE in certain problems. In particular, LPE will always locate better (or the same) explanations and require fewer (or equal) goal expansions than EBG. Thus EBG should be used only when the domain theory, as reformulated by LPE, grows unacceptably large.

The choice between PE and LPE hinges on the tradeoff between the speed with which goal expansions are executed and the number of goal expansions. Goals can be expanded faster in PE than in LPE, but LPE always performs fewer (or equal) expansions. If LPE does fewer than half the goal expansions of PE, as in domains in which the PE expansion is infinite, LPE is the method of choice. There are two kinds of domain in which the slight speed advantage of PE is significant. The first, exemplified by the 'zebra' CSP, is that in which almost all proofs will be required at some point during problem-solving. We conjecture that most real-world problems are not of this kind. The second kind of domain in which LPE does almost as many goal expansions as PE are domains in which goals must be expanded to an operational level in order to determine if they apply to a hypothesis. In this kind of domain there are no opportunities for 'laziness': LPE is forced to be 'eager' and behave just like PE.

5 Conclusion

We have presented LPE, a new compilation learning technique. LPE is based on a similar learning principle to EBG, namely that storing the results of search can speed up problem-solving. LPE's strength comes from the equivalence of the set of (successful and failed) proofs explored to the original domain theory. This allows LPE to repeatedly replace rather than augment the original theory with simpler equivalents, making proofs faster to find in future. In addition the bias or 'masking effect' of EBG can be avoided. In domains where full PE is intractable or unnecessary, LPE can be a more effective 'compilation learning' method. LPE is faster than EBG unless the reformulation of the domain theory grows unacceptably large. This analysis has been illustrated in a constraint satisfaction task.

[2](approximately by a factor of 2.5 in this implementation)

Acknowledgements and Availability

Thanks to Stan Matwin, David Aha and the anonymous referees for their valuable comments on an earlier draft of this paper.

A Prolog implementation of LPE is available from the authors ({pclark,holte}@csi.uottawa.ca) on request.

References

Clark, P. and Holte, R. (1992). Generalised backjumping. Technical report, Univ. Ottawa, Canada.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312.

Eskey, M. and Zweben, M. (1990). Learning search control for constraint-based scheduling. In *AAAI-90*, pages 908–915.

Gaschnig, J. (1979). *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon Univ., PA.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *AAAI-88*, pages 564–569. Kaufmann.

Minton, S., Carbonell, J., Etzioni, O., Knoblock, C., and Kuokka, D. (1987). Acquiring effective search control rules: Explanation-based learning in the prodigy system. In *Proc. Fourth Int. Machine Learning Workshop*, pages 122–133, CA, Kaufmann.

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

Mostow, J. and Bhatnagar, N.(1987). Failsafe – a floor planner that uses ebg to learn from its failures. In *IJCAI-87*, pages 249–255.

Prieditis, A. E. (1988). Environment-guided program transformation. In G. DeJong, editor, *AAAI Spring Symposium on Explanation-Based Learning*, pages 201–209, Ca, AAAI.

van Harmelen, F. and Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36:401–412.

```
%    Usage: lpe(+Goal, +GenGoal, -Leaves, -GenLeaves, -TFFlag).
1    lpe((A,B), (GenA,GenB), (LeavesA,LeavesB), (GenLeavesA,GenLeavesB), TF) :- !,
2            lpe(A, GenA, LeavesA, GenLeavesA, TFA),
3            (TFA = true -> lpe(B, GenB, LeavesB, GenLeavesB, TF)
4                        ;  LeavesB = B, GenLeavesB = GenB, TF = false ).
5[a] lpe(   A, GenA, false, GenA, false) :- operational(GenA), \+ call(A), !.
6[b] lpe(   A, GenA,     A, GenA,  true) :- operational(GenA), !, call(A).
7[c] lpe(Goal, GenGoal, Leaves, GenLeaves, TF) :-
8            clause(GenGoal, GenBody),
9[d]         copy((GenGoal:-GenBody), (SpecGoal:-GoalBody)),
10           UnificationGoal = (Goal=SpecGoal),
11           lpe((UnificationGoal,GoalBody), (true,GenBody), Leaves, GenLeaves, TF).
```

[a] Deleting line 5 converts LPE into EBG.
[b] Deleting line 5 and `call(A)` in line 6 converts LPE into PE.
[c] To see the whole proof, 7 is: `lpe(Goal,GenGoal,(Goal:-Leaves),(GenGoal:-GenLeaves),TF):-`
[d] `copy(A,CopyA) :- assert(copy_of(A)), retract(copy_of(CopyA)).` (`A` and `CopyA` don't share vars)

Example usage: `| ?- lpe(tiger(joe), tiger(X), Ls, GenLs, TF).`

```
GenLs = striped(X), eats_meat(X), tail(X) ;
GenLs = striped(X), teeth(X), mammal(X), tail(X) ;
GenLs = striped(X), fast_runner(X) ;
no
```

Figure 2: The Prolog LPE algorithm

procedure `find_best_proof(Example,Concept)` returning `Proof`:
foreach definition `D` of `Concept` in domain theory:
  if `D` is operational and `Example` satisfies `D` then **exit**, returning `D`.
  if `D` is not operational
  then **repeat call** `LPE(D,Example)` to get an $\texttt{ExpandedD}_\texttt{i}$
    until no more $\texttt{ExpandedD}_\texttt{i}$ can be found
    if `LPE` made some expansions (the set of $\texttt{ExpandedD}_\texttt{i} \neq \{\texttt{D}\}$)
    then delete `D` and add all $\texttt{ExpandedD}_\texttt{i}$ to domain theory, ordered
      by (upper bound on) each $\texttt{ExpandedD}_\texttt{i}$'s 'quality' (Section 2.4)
exit with failure (`Example` doesn't satisfy any definition of `Concept`).

Figure 3: The Concept Ordering/Execution Algorithm

Appendix A: The Prolog LPE Algorithm

A Prolog implementation of LPE is shown in Figure 2.

Appendix B: Concept Ordering Algorithm

The concept ordering/execution algorithm is down in Figure 3, which in turn calls the basic `LPE` mechanism.

Appendix C: The Cities Problem

A city (one of {athens,paris,berlin,oslo,london}) must be visited on each of the seven days of the week, and:

- a city is not visited on two consecutive days.
- paris is visited exactly three times.
- athens is visited at least once.
- berlin is not visited more than once.
- athens is not visited after any visits to paris.
- oslo is visited directly after any trip to london.
- oslo is not visited on day 2.
- paris is the last place visited.
- if oslo is visited, then berlin mustn't be visited.
- the city visited first must not also be visited on day 6.

We formulate the problem using seven variables, one for each day of the week, each taking the name of a city as its value. Note there are several solutions to this problem.

Appendix D: The Zebra Problem

- There are five houses, each of a different colour and inhabited by men of different nationalities, with different pets, drinks and cigarettes.
- The Englishman lives in the red house.
- The Spaniard owns a dog.

- Coffee is drunk in the green house.
- The Ukranian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- The Chesterfield smoker lives next to the fox owner.
- Kools are smoked next to the house where the horse is kept.
- The Lucky-Strike smoker drinks orange juice.
- The Japanese smokes Parliament.
- The Norwegian lives next to the blue house.

We formulate the problem using twenty five variables, the first five for the colours of houses 1,...,5, the second five for the nationalities, and the third, fourth and fifth five for the pets, cigarettes and drinks respectively.