An Experimental Comparison of Rippling and Exhaustive Rewriting

Alan Bundy
A.Bundy@ed.ac.uk

Ian Green
I.Green@ed.ac.uk

4th December 1996

Abstract

We compare rippling and exhaustive rewriting using recursive path ordering, on a range of inductive proofs. We present statistics on success rates, branching rates and CPU times. We use these statistics to argue that rippling succeeds more often. However, these statistics also show that rippling and reduction are roughly the same in terms of average branching rate and that rippling often takes longer in terms of CPU time.

1 Introduction

Rippling has been advocated as a heuristic technique for guiding proofs by mathematical induction [5, 8, 11]. In particular, it is used to rewrite the induction conclusion into a form where a sub-expression of it matches the induction hypothesis. The main rival technique is the exhaustive application of rewrite rules. These rewrite rules are often oriented by a recursive path order or some similar simplification order. For brevity we will call this technique *reduction*.

On the basis of anecdotal evidence, rippling has been claimed to generate less search than reduction and to succeed in situations where reduction fails (*cf.* [6]). A more systematic experimental analysis is required to substantiate these claims and is long overdue. This paper fills that gap.

In this paper we report on an experimental comparison of these two techniques using the Clam proof planner. We compare them on three dimensions:

Success Rate: which technique more often leads to a proof?

Branching Rate: which technique requires the least search?

Timing: which technique is faster in CPU time?

We hypothesise an expected outcome of the experiments on each of these dimensions. We then give an extensive statistical analysis of the results to test each of each of these hypotheses.

Informally expressed, our null hypotheses are:

SUCC: Rippling is as equally likely to succeed as reduction (the alternative hypothesis is that rippling is more likely to succeed).

BR: The average branching rate of the proof search generated by rippling is equal to that generated by reduction (the alternative hypothesis is that rippling branches less than reduction).

CPU: Reduction and rippling are equally fast in CPU time (the alternative here is that reduction is faster).

We have tested both techniques on 111 inductive theorems drawn from an independent source: the Boyer/Moore corpus, as reported in [3]. We conducted two sets of tests: one with the minimal number of lemmas available, and one with the maximal number. The results were as follows: the BR null hypothesis could not be rejected; the CPU null hypothesis was rejected in the minimal lemma configuration, and the alternative accepted; the $SUCC$ null hypothesis was rejected in the maximal lemma configuration, and the alternative accepted.

§§2, 3 and 4 give the definitions of reduction and rippling and an overview of constructor-style induction. §5 describes the experiments and data gathering. §6 describes our results in detail. §§7 and 8 draw further conclusions concerning the relationship between the two techniques and suggest work for the future.

2 The Step Cases of Inductive Proofs

Inductive proofs are characterised by the application of induction rules, of which a simple example is Peano induction:

$$\frac{P(0), \quad \forall N{:}nat.\ (P(N) \to P(s(N)))}{\forall N{:}nat.\ P(N)}$$

here s is the successor function, *i.e.*, $s(N) = N + 1$. More generally, we will focus on *constructor style* induction rules of the general form (omitting types):

$$\frac{B \quad \forall \overrightarrow{X_1}.\ \bigwedge_{i=1}^{k_1} P(X_1^i) \to P(\overrightarrow{c_1(\overrightarrow{X_1})}), \quad \ldots, \quad \forall \overrightarrow{X_n}.\ \bigwedge_{i=1}^{k_n} P(X_n^i) \to P(\overrightarrow{c_n(\overrightarrow{X_n})})}{\forall X.\ P(X)}$$

where B is one or more base-cases and the c_i are step-case functions. Note that we follow the Prolog convention of using lower case for constants and upper case for variables.

In particular, we are interested in the step cases of such inductive proofs, *i.e.*, the proof of:

$$\forall X_1, \ldots, X_k.\ P(X_1) \wedge \ldots \wedge P(X_k) \to P(\overrightarrow{c(X_1, \ldots, X_k)}) \tag{1}$$

for a particular function c. In these proofs the $P(X_i)$ are called the *induction hypotheses* and $P(\overrightarrow{c(X_1, \ldots, X_k)})$ is called the *induction conclusion.*[1]

The usual goal of step cases is to rewrite the induction conclusion so that the induction hypotheses can be used to prove it. Typically, the rewriting will enable the induction hypotheses to be matched to subexpressions of the induction conclusion. These subexpressions can be replaced by *true* and the induction conclusion considerably simplified. Following Boyer and Moore, we call this process *strong fertilization.* An example is given in figure 1. Sometimes only one side of an equational induction conclusion can be rewritten to match the corresponding side of the induction hypothesis. The induction hypothesis can then be used to rewrite the induction conclusion. We call this process *weak fertilization.*

Note that when P contains universally quantified variables,[2] skolemisation will turn these into free variables in the induction hypotheses and skolem functions in the induction conclusion. During fertilization these free variables may be instantiated. This observation has important consequences for rippling (see §4).

3 Reduction and Recursive Path Ordering

Rewriting works by repeated application of the following rule of inference:

$$\frac{exp[rhs\phi], \quad cond\phi, \quad cond \to lhs \Rightarrow rhs}{exp[lhs\phi]}$$

[1] $P(\overrightarrow{c(X_1, \ldots, X_k)})$ is intended to capture the fact that there may be multiple constructors in a single step-case.

[2] in positive polarity positions or existentially quantified variables in negative polarity positions.

Induction Hypothesis:

$$rev(t) <> L = qrev(t, L)$$

Induction Conclusion (and subsequent rewriting):

$$\begin{aligned} rev(h :: t) <> l &= qrev(h :: t, l) \\ (rev(t) <> h :: nil) <> l &= qrev(t, h :: l) \\ rev(t) <> (h :: nil) <> l &= qrev(t, h :: l) \\ rev(t) <> h :: l &= qrev(t, h :: l) \end{aligned}$$

Equations:

$$\begin{aligned} rev(X :: Y) &= rev(Y) <> X :: nil \\ qrev(X :: Y, Z) &= qrev(Y, X :: Z) \\ (X <> Y) <> Z &= X <> (Y <> Z) \\ X :: nil <> L &\Rightarrow X <> L \end{aligned}$$

rev and $qrev$ are list reversal functions; rev is the naive version and $qrev$ is tail recursive reverse. $::$ is the infix list constructor and $<>$ is infix list append. The Ls are free variables in the induction hypothesis. They correspond to the ls in the induction conclusion, which are sinks.

The proof works by applying the equations, left to right, to the induction conclusion. When this is complete the proof can be completed by fertilizing the induction conclusion with the induction hypothesis. Note that the free variable L is matched to the term $h :: l$.

Figure 1: The step case of an inductive proof

where $lhs\phi$ is a distinguished subexpression of exp and $lhs\phi$ is called the *redex*. The formula $cond \to lhs \Rightarrow rhs$ is called a *conditional rewrite rule* (or just a rewrite rule if $cond \equiv true$). $\Rightarrow$ is directed rewriting and $\to$ is implication.

To apply reduction, a set of (conditional) equations/implications must be oriented to form (conditional) rewrite rules, *i.e.*, $cond \to lhs \rightsquigarrow rhs$ must be oriented as either $cond \to lhs \Rightarrow rhs$ or $cond \to rhs \Rightarrow lhs$, where $\rightsquigarrow$ maybe $=$, $\leftrightarrow$ or $\to$. We cannot include both or the reduction process will loop.[3]

To prove that reduction with a set of rewrite rules will terminate we need to find a measure which maps each term to an element in a well-founded set.

A popular measure is *recursive path ordering with status*, (RPOS) a simplification ordering due to Dershowitz [9] and later Kamin and Lévy [10]; the rules defining RPO are shown in figure 2 (we omit the rules for RPOS to simplify the presentation). Recursive path orderings were first used in inductive completion systems, *e.g.*, RRL [12] and SPIKE [2]. They have now been adapted for use in explicit induction systems, *e.g.*, [13], where they have proved very successful.

4 Introduction to Rippling

Rippling works by the selective application of rewrite rules. The induction conclusion is annotated to show the similarities (called the *skeleton*) and the differences (called the *wave-fronts*) between it and the induction hypotheses. The rewrite rules (called *wave-rules*) are annotated to show the

[3] In fact, since rewriting w.r.t. implications may only occur at positions of appropriate polarity, one can be more generous and admit both directions, providing they are used at opposite polarity.

$$\frac{f \sim g, \quad \{s_1, \ldots, s_m\} \gg \{t_1, \ldots, t_n\}}{f(s_1, \ldots, s_m) \geq g(t_1, \ldots, t_n)} \qquad \frac{s_i \geq t}{f(s_1, \ldots, s_m) \geq t}$$

$$\frac{f \succ g, \quad \forall i.\ f(s_1, \ldots, s_m) > t_i}{f(s_1, \ldots, s_m) > g(t_1, \ldots, t_n)}$$

$>$ is the strict part of $\geq$. $\gg$ is the multi-set extension of $>$. $\succ$ is the quasi-precedence defined over function symbols: once this is fixed, $\geq$ is well defined.

Figure 2: Rules of recursive path ordering

similarities and differences between the left- and right-hand sides of the rule. When wave-rules are applied not only must they match a sub-expression in the induction conclusion at the object-level, but the wave-fronts in the wave-rule must also coincide with wave-fronts in the induction conclusion. Thus all ripples are rewrites but the reverse is not true. Rippling is also guaranteed to terminate [1]. A well founded, stable and monotonic measure is associated with the wave-fronts. A strict decrease in this measure is a defining characteristic of wave-rules.

Before giving the examples below we must introduce the notation of rippling. We describe this notation first with respect to induction hypotheses and conclusions and then with respect to wave-rules.

Wave-fronts: are the differences between the induction conclusion and the induction hypotheses. We denote them by putting the expression in a rectangular box with a directional arrow. The arrow indicates the intended direction of movement of the wave-front. An upwards arrow indicates that the wave-front should move outwards (or up the parse tree); a downwards arrow indicates that the wave-front should move inwards (or down the parse tree). Wave-fronts usually have one or more bits of skeleton inside them. These are called *wave-holes* and are underlined. For instance, the abstract induction conclusion from (1) would be annotated as $P(\boxed{\overrightarrow{c(\underline{X_1}, \ldots, \underline{X_k})}}^{\uparrow})$.

Skeletons: are the common expressions in the induction conclusion and the induction hypotheses. The skeleton corresponds to any expression outside all wave-fronts and any underlined expression inside a wave-front. When wave-fronts have multiple holes the expression may contain several skeletons. Then the expressions outside the wave-fronts will be shared by several different skeletons. For instance, the skeletons in the abstract induction conclusion above are: $P(X_1), \ldots, P(X_k)$, one for each induction hypothesis in (1).

Free-variables: When the induction hypothesis is skolemized, universally quantified variables become free variables. These free variables have a special status because they can match any sub-expression in the induction conclusion, including wave-fronts. To denote this special status they are written in calligraphic font.

Sinks: The part of the induction conclusion corresponding to free variables is called a *sink* because it is one of the places that wave-fronts can be moved to and absorbed in the fertilization match. They thus have a status which is ambiguous between wave-front and skeleton. Sinks are denoted by the marks $\lfloor \ldots \rfloor$, which looks a little like a kitchen sink with a plug hole in the middle.

A formal definition of this notation can be found in [1]. Figure 3 gives some more examples of wave annotation in use.

The notation for wave-rules is essentially the same, except that there are no sinks and free variables are not written in calligraphic font. The skeleton denotes the parts of the wave-front

Induction Hypothesis:

$$qrev(qrev(tl, \mathcal{L}), [\,]) = rev(\mathcal{L}) <> rev(rev(tl))$$

Induction Conclusion (and subsequent ripple):

$$
\begin{aligned}
qrev(qrev(\boxed{[hd|\underline{tl}]}^{\uparrow}, \lfloor l \rfloor), [\,]) &= rev(\lfloor l \rfloor) <> rev(rev(\boxed{[hd|\underline{tl}]}^{\uparrow})) \\
qrev(qrev(tl, \lfloor [hd|l] \rfloor), [\,]) &= rev(\lfloor l \rfloor) <> rev(\boxed{\underline{rev(tl)} <> [hd]}^{\uparrow})) && (2) \\
qrev(qrev(tl, \lfloor [hd|l] \rfloor), [\,]) &= rev(\lfloor l \rfloor) <> \boxed{rev([hd]) <> \underline{rev(rev(tl))}}^{\uparrow} \\
qrev(qrev(tl, \lfloor [hd|l] \rfloor), [\,]) &= \boxed{\underline{rev(\lfloor l \rfloor)} <> rev([hd])}^{\downarrow} <> rev(rev(tl)) && (3) \\
qrev(qrev(tl, \lfloor [hd|l] \rfloor), [\,]) &= rev(\lfloor [hd] <> l \rfloor) <> rev(rev(tl)) \\
qrev(qrev(tl, \lfloor [hd|l] \rfloor), [\,]) &= rev(\lfloor [hd|l] \rfloor) <> rev(rev(tl))
\end{aligned}
$$

The wave-fronts are in boxes and the wave-holes underlined. The $\mathcal{L}$s are free variables in the induction hypothesis. They correspond to the sinks $\lfloor l \rfloor$s in the induction conclusion.

The ripple starts with the applications of the recursive definitions of qrev and rev (wave-rules (6) and (4) from figure 4) to the left- and right-hand sides, respectively. The lemmas (10), (9) and (11) are then applied, in turn, to the right-hand side. Finally, the sink on the right-hand side is simplified to make it equal to the one on the left-hand side.

Note that this proof requires the same equation to be used in opposite directions, namely (11) and (10) applied at steps (3) and (2), respectively. This would cause problems for reduction since both rules could not be included without causing looping.

The wave-rules used in this proof are shown in figure 4.

Figure 3: An example of ripple notation

that are common between the left- and right-hand sides of the rule; the wave-fronts denote the differences. Figure 4 gives some example wave-rules using this notation.

Generally there are many ways to annotate a given formula as a wave-rule. For instance, (4) and (5) are two different wave-rules corresponding to the same equation. In Clam annotations are computed 'on-the-fly' to avoid generating wave-rules that are never used.

5 The Experimental Methodology

To compare two rival computational techniques experimentally it is necessary to build two computer systems which differ *only* in those two techniques and run them on a *representative* set of examples.

We first considered building stand-alone rippling and reduction engines and testing them on pairs of induction hypotheses and conclusions. We rejected this set-up because the step cases of inductive proofs consist of more than just rewriting. They also contain, for instance, case splits and nested inductions. This simple experimental set-up would not allow comparisons in such situations.

Instead, we decided to incorporate both techniques into the same theorem prover for inductive proof. We already had such a prover in the Oyster-Clam system [7]. This system embodies rippling, but its modular structure readily supports the replacement of rippling with reduction. In fact, it already contained a module for reduction with rules oriented by RPOS. This module is used in non-inductive parts of the proofs, *e.g.,* base cases. It was, thus, a simple matter to replace rippling by reduction. The only other changes required were to make both modules calculate the branching rate at each step and to insert statistics gathering procedures, *e.g.,* for calculating CPU times and branching rates.

We ran the examples only on the Clam proof planning part of the Oyster-Clam system. This carries out rippling and reduction in detail, so was sufficient for our purposes. Running these plans on the Oyster proof checker would have produced no new information.

The Boyer/Moore corpus is widely used as a representative and independently generated test set for inductive proofs. For instance, it has been adopted as a basis for the induction competition being organised by McAllester in conjunction with CADE-14.[4] We already had a part of this corpus in a form suitable for input to Clam, so it seemed an obvious basis for comparison. In one sense, however, the Boyer/Moore corpus is not representative: the Boyer/Moore theorem prover, Nqthm [4], proves them all.[5] So we added some test examples which are known to cause difficulty for Nqthm.

The number of lemmas available to an inductive theorem prover has a significant effect on both its search space and its success rate. Redundant lemmas will increase the branching rate and, hence, the search space. This may prevent a proof being found within a pre-set time limit. However, a shorter proof may be found and cause the prover not to explore regions of higher branching rate.

On the other hand, the absence of key lemmas may prevent a proof being found. In our experiments we tested two extreme situations. In the *minimal* configuration, only the definitions of functions were available as rewrite rules. In the *maximal* configuration all theorems of the test set were available as rewrite rules (excepting the theorem being proved). This required both their automatic orientation as reduction rules and their automatic annotation as wave-rules.

Rewriting search spaces contain both AND and OR branches. AND branches occur when two redexes are disjoint. Like the selected literal restriction in resolution, one choice can be made and the other delayed until later. OR choices occur when one redex is nested within another. Both choices must be considered since once one is chosen the other may no longer be available later. We would have preferred to have counted only OR choices when calculating branching rates. Unfortunately, a deeper analysis convinced us there was no clean division between the two kinds of choice. For instance, two disjoint redexes can be nested within a third redex. Reluctantly, we were forced to use both AND and OR choices to calculate branching rate. This gives branching rates which are misleadingly high, but comparison between rippling and reduction is still possible since they are affected equally. It does, unfortunately, dilute the ratio between the branching rates, so that the contrast is not so great as it would be if we were able to restrict attention to OR branching alone.

This point is relevant when comparing the findings in this paper with the claims made in our earlier papers. Where we have claimed that rippling usually generates a branching rate of 0 or 1, this refers to OR branching alone. For instance, a combined branching rate of 2 often describes two AND choices, *e.g.,* one on each side of an equation. This corresponds to an OR branching rate of 1. We did not count zero branching when calculating branching rates. This had the effect of artificially increasing the measured branching rate of rippling in many cases. However, including zero branching would have biased the experimental results in the opposite direction.

[4] See `http://www.ai.mit.edu/people/dam/induction/contest.html`

[5] Albeit with some hints to guide the proof.

6 Results

Both rippling and reduction versions of Clam were run on 111 example theorems drawn from the Boyer/Moore corpus [3] and various other sources. The experiments were performed on a Sun SPARC Ultra I, with 128Mb of RAM, running Clam 2.5 under Quintus Prolog version 3. Data were collected on success rate, average branching rate and CPU time for both the minimal and maximal lemma configurations. The full experimental data are available on application to the second author.

Branching rate and CPU time data were collated from both fruitful and fruitless parts of the space, but only for *successful proof searches.* The space was searched depth-first, terminating either when a proof was found or a time-limit of 5 minutes was exceeded.

The overall and average statistics are given in table 1. Since we have sufficient data (i.e., population sizes larger than 30) we used a normal analysis to test all hypotheses. In the case of branching rate and CPU times, we compared the differences between means using paired observations (*i.e.*, theorem by theorem). In the case of success rate, we compared the difference between the probabilities.

Statistic	*Rippling*		*Reduction*	
	Min	*Max*	*Min*	*Max*
Overall success (%)	91.9	88.3	87.4	74.8
Average branching rate	1.38	1.51	1.37	1.44
Average CPU time (ms)	2047	6876	1309	3260

Table 1: Overall and average statistics

In the subsections below we provide more detail, formally restate our hypotheses and discuss whether they were confirmed or rejected.

6.1 Success Rate

The success rates given in table 2 show results across the test set. A more formal version of the hypothesis from §1 is $SUCC$ that $p_{rip} = p_{red}$, where p_{rip} and p_{red} are the probabilities that rippling and reduction, respectively, are able to prove a theorem. $SUCC_{min}$ and $SUCC_{max}$ indicate this hypothesis in the minimal and maximal, respectively, lemma configurations.

We assumed that these probabilities are binomial parameters and applied the normal test. In the case of minimal lemmas we cannot reject the null hypothesis at confidence level 95%.[6] In the case of maximal lemmas, we can reject the null hypothesis with 97.5% confidence and accept the alternative, that rippling is more likely to succeed than reduction.

The two histograms in figure 5 show how the success rate of reduction decreases with increasing difficulty of theorem. As a measure of theorem difficulty we took the CPU time taken for the rippling proof. This is not an ideal, objective measure of difficulty, but we could not think of a better one. The height of each bar shows the percentage of theorems that reduction could prove for each level of difficulty. It can be seen that this steadily decreases as the theorem difficulty increases. The effect is much more pronounced in the maximal lemma configuration.

We do not give the corresponding histograms for rippling success rates because all the bars would be 100%. This is because rippling proves all the theorems that reduction proves.

6.2 Branching Rate

The branching rate data are presented as scattergrams in figure 6. This data is restricted to the theorems that both rippling and reduction can prove.

[6] This is the confidence level usually required in statistical experiments to reject the null hypothesis.

Technique	*Percentage proved*	
	Minimal lemmas	*Maximal lemmas*
Rippling & Reduction	87.4	74.8
Rippling alone	4.5	13.5
Reduction alone	0	0
Neither	8.1	11.7
TOTAL	100	100

The table above shows the ratios of theorems proved (in the minimal and maximal lemma configuration) by: both rippling and reduction; rippling alone; reduction alone; and by neither rippling nor reduction.

Table 2: A comparison of success rates

A more formal version of the BR hypothesis from §1 is that $b_{rip} = b_{red}$, where b_{rip} and b_{red} are the average branching rates for rippling and reduction, respectively. BR_{min} and BR_{max} indicate this hypothesis in the minimal and maximal, respectively, lemma configurations.

We used the normal test to test these hypotheses. We found that we cannot reject the null hypothesis in either the maximal or minimal configuration and so must conclude that rippling and reduction do not differ significantly in branching rate.

6.3 CPU Time

The CPU time data are presented as scattergrams in figure 7. These data are restricted to the theorems that both rippling and reduction can prove.

A more formal version of the CPU null hypothesis from §1 is that $t_{rip} = t_{red}$, where t_{rip} and t_{red} are the average CPU times for rippling and reduction, respectively. CPU_{min} and CPU_{max} indicate this hypothesis in the minimal and maximal, respectively, lemma configurations.

We used the normal test to test these hypotheses. In the case of maximal lemmas we cannot reject the null hypothesis at confidence level 95%. In the case of minimal lemmas, we can reject the null hypothesis at the 95% confidence level, and accept the alternative that reduction is faster than rippling.

6.4 Discussion of Results

A summary of the results of the hypothesis testing is shown in table 3. We discuss each of these results in turn.

6.4.1 Success Rate

In the maximal lemma configuration the success rate data are much better than we expected. We knew that when bi-directional rewriting was required, rippling might succeed, but reduction was bound to fail. We expected such situations to be rare, and were surprised to discover that they were not. In the minimal lemma configuration the results are suggestive, but not statistically significant. In retrospect, this is not surprising since the opportunities for bi-directional rewriting are much less when the rules are restricted to definitions.

It is surprising that the success rates of both rippling and reduction *drop* when more rules are available, *i.e.* are less in the maximal lemma configuration than the minimal one. There seem to be two causes of this. One cause is that with the bigger search space more of the searches exceed the 5 minutes limit. The other cause is that the search space is not complete. For instance, a successful fertilization cuts the step case choice points. Particularly if this was only a weak fertilization, then the resulting subgoal may fail. The maximal lemma configuration gives more

Hypothesis	*Decision*	*Confidence*
$SUCC_{min}$	not reject	95%
$SUCC_{max}$	reject	97.5%
BR_{min}	not reject	95%
BR_{max}	not reject	95%
CPU_{min}	reject	95%
CPU_{max}	not reject	95%

The first column shows our null hypotheses, the second column shows whether this was rejected or not, and the third column shows the confidence level at which this decision was taken. SUCC is the hypothesis that rippling and reduction are equally successful. BR is the hypothesis that rippling has the same branching rate as reduction. CPU is the hypothesis that reduction has the same CPU time as rippling. The subscripts min and max indicate whether the results are from the minimal or maximal lemma configurations.

Table 3: Results of hypotheses testing

opportunities for such incompleteness. We were pleased to see that rippling suffered from these problems far less often than reduction.

Both the analysis above and the histograms in figure 5 suggest that rippling has its best relative performance in more difficult cases, *i.e.* harder theorems and more redundant lemmas. This is very encouraging and suggests that tests on even harder cases might show rippling in an even better light.

6.4.2 Branching Rate

We were disappointed that rippling did not have a significantly lower branching rate than reduction. Closer analysis of the data suggests that this is because the problem set is too easy. There are just not enough redexes in the induction conclusions to present a significant amount of search. We need to retry the experiments with much bigger theorems to see what happens when more choice is available.

In many theorems rippling had a higher branching rate than reduction. In some cases this is because rippling allows the same lemma to be used in both orientations. Additionally since we only count situations where the number of redexes is non-zero, the rippling branching rate can be misleadingly high. Proofs where many goals do not contain ripple-redexes fail to contribute to a *decrease* in the branching rate. A more careful analysis would have to account for proof length, and so allow branching rate to be a more accurate estimator of the size of the search space.

6.4.3 Timing

We expected rippling to be slower than reduction because of the overhead of inserting and matching annotation. We were pleased that rippling was only about twice as slow as reduction. Given the higher success rate of rippling this slightly slower speed seems a price well worth paying.

Eyeballing the scattergrams in figure 7, rippling seems to be consistently slower than reduction in both the minimal and maximal configurations. We were, therefore, surprised that the CPU_{max} hypothesis was not rejected. Closer analysis of the data reveals this to be an artifact. If we remove the three outliers, where rippling is faster than reduction, then CPU_{max} is rejected with 99% confidence. These three outliers happen because reduction requires a significant amount of backtracking to find a proof.

7 Further Work

The major problem with the test set of examples used so far is that they are too easy. Both techniques generate relatively low branching rates and prove a high number of theorems in a relatively short amount of time. This generates insufficient contrast to give a definitive comparison between the techniques. We intend to repeat the experiments with a more difficult set of theorems.[7] In particular, we need longer theorems containing more redexes, so that there are more rewriting choices. We have several such theorems available from our work on hardware verification and program synthesis. The difficulty is in ensuring that these examples are *representative* enough to provide a fair comparison.

One problem with this plan is that with a lower success rate we will be missing a lot of data, especially on proof CPU times. It may be necessary to revert to our previously rejected plan (see §5) to compare only the step cases of inductive proofs, rather than overall proofs. This might, in any case, be another useful set of data to collect and compare.

A possible criticism of our conclusions is that RPOS is a straw man and more powerful termination orders might give better results. We doubt this, since most simplification orders would orient our experimental rule set in the same way as RPOS.[8] In any case, it is the fixed orientation of rewrite rules that creates problem for reduction rather than the particular orientation. However, we could readily check this claim by running the same tests using several alternative simplification orders.

A more interesting experiment would be to use a dynamically generated simplification order. Recursive path orderings can be generated dynamically in this way as shown by Lescanne (see Forgaard [10]). In fact, Clam computes the quasi-precedence incrementally in just this way. We have a variant of the rewriting method which orients rules *lazily* during the course of proof search. Different branches of the proof can have the same rule oriented in different directions. Thus we would only encounter orientation problems if the same reduction rule was required to be oriented in different directions on the same branch of the proof.[9] Unfortunately, our initial experience with this dynamic strategy in depth-first proof search is that the success rate drops dramatically because it too often picks inappropriate orderings first. It might, however, be worth experimenting with this strategy further to see if these initial problems can be overcome.

It would be interesting to remove the second cause of failure in the maximal lemma configuration by allowing Clam to back-track into rippling after a successful fertilization. We could then see whether this increased the maximal lemma success rate.

We could also experiment with variations of rippling to see where its power is coming from. For instance, we could not insist on measure decrease during rippling, but only insist on skeleton preservation. This will lose termination, so we could try reimposing it with a fixed orientation and alternatively consider methods of searching an infinite search space.

8 Conclusions

Although not all of our hypotheses were confirmed, we were very encouraged by these experimental results. Although the branching rate of rippling was not significantly better than that of reduction, the success rates were. Moreover, the differences in success rates were most pronounced in the most difficult cases, *i.e.*, harder theorems and more redundant lemmas. As expected, rippling was slower than reduction, but by an acceptably small factor.

These results strongly suggest trying the same experiments on a much tougher set of theorems. We then hope rippling will exhibit a lower branching rate and even higher success rate than reduction for an acceptably small increase in time. Work on these new experiments is about to start.

[7] We hope to have these experiments completed by the CADE-14 deadline for camera-ready copy and to include them in a revised version of this paper.

[8] RPOS is strong enough to orient *all* the rules in the test set.

[9] See figure 3 for an example of such a proof.

Acknowledgements

The research reported in this paper was supported by EPSRC grant GR/L/11724. We would like to thank John Hallam and Helen Lowe for advice on statistical processing of the data.

References

[1] David Basin and Toby Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[2] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *Proceedings of the 13th IJCAI.* International Joint Conference on Artificial Intelligence, 1993.

[3] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[5] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[6] Alan Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[7] Alan Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[8] Alan Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[9] Nachum Dershowitz. Orderings for term rewriting systems. *Journal of Theoretical Computer Science*, 17(3), 1982.

[10] Randy Forgaard. A program for generating and analyzing term rewriting systems. Master's thesis, Laboratory for Computer Science, MIT, USA, September 1984.

[11] D. Hutter. Guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[12] Deepak Kapur and H. Zhang. A rewrite rule laboratory—User's manual. In *Proc. of 3rd Intl. Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, 1989.

[13] U. S. Reddy. Term rewriting induction. In *Proc. of Tenth International Conference on Automated Deduction.* Springer-Verlag, 1990.

$$rev(\boxed{X :: \underline{Y}}^{\uparrow}) \Rightarrow \boxed{\underline{rev(Y)} <> X :: nil}^{\uparrow} \tag{4}$$

$$\boxed{\underline{rev(Y)} <> X :: nil}^{\downarrow} \Rightarrow rev(\boxed{X :: \underline{Y}}^{\downarrow}) \tag{5}$$

$$qrev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) \Rightarrow qrev(Y, \boxed{X :: \underline{Z}}^{\downarrow}) \tag{6}$$

$$qrev(Y, \boxed{X :: \underline{Z}}^{\uparrow}) \Rightarrow qrev(\boxed{X :: \underline{Y}}^{\downarrow}, Z) \tag{7}$$

$$(\boxed{\underline{X} <> Y}^{\uparrow}) <> Z \Rightarrow X <> (\boxed{Y <> \underline{Z}}^{\downarrow}) \tag{8}$$

$$X <> \boxed{(Y <> \underline{Z})}^{\uparrow} \Rightarrow \boxed{(\underline{X} <> Y)}^{\downarrow} <> Z \tag{9}$$

$$rev(\boxed{\underline{L} <> \underline{K}}^{\uparrow}) \Rightarrow \boxed{\underline{rev(K)} <> \underline{rev(L)}}^{\uparrow} \tag{10}$$

$$\boxed{\underline{rev(K)} <> \underline{rev(L)}}^{\downarrow} \Rightarrow rev(\boxed{\underline{L} <> \underline{K}}^{\downarrow}) \tag{11}$$

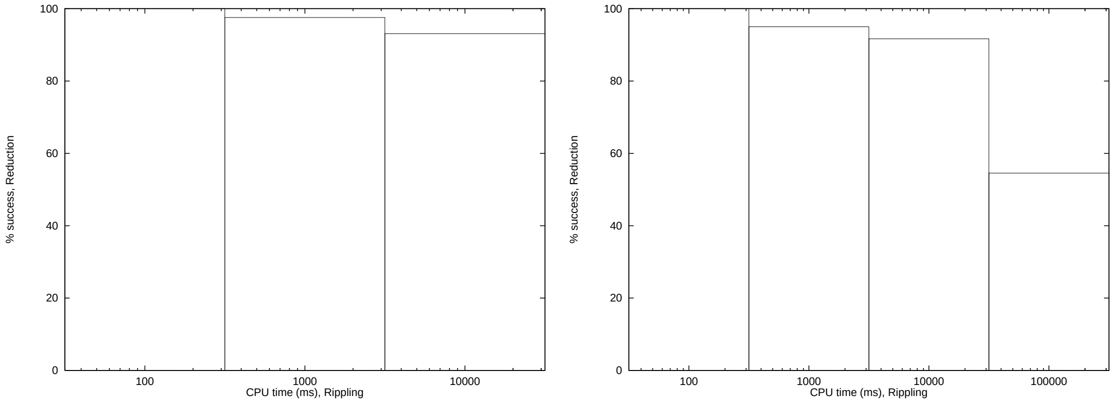
(4) and (5) come from the recursive definitions of rev. (6) and (7) come from the recursive definitions of qrev. (8) and (9) come from the associative law of $<>$*. (10) and (11) come from the distributive law of rev over* $<>$*.*

(4) and (10) are longitudinal *wave-rules used to ripple wave-fronts outwards until the skeletons are totally contained in their wave-holes. (6), (7), (8) and (9) are* transverse *wave-rules used to ripple wave-fronts sideways above sinks. Transverse wave-rules cannot be used unless they place wave-fronts above sinks. (5) and (11) are inwards directed wave-rules used to ripple wave-fronts inwards towards sinks.*

(10) and (11) contain wave-fronts with multiple wave-holes. Each of these wave-holes define a different skeleton; the expression outside the wave-fronts being shared between them.

Notice that, unlike conventional rewrite rules, a set of wave-rules can contain equations oriented in both directions. The pairs: (4) and (5); (6) and (7); (8) and (9); and (10) and (11) all provide examples. The termination proof of rippling, [1], ensures that this does not cause looping. Such pairs of rules can even be used within the same proof, cf. *figure 3.*

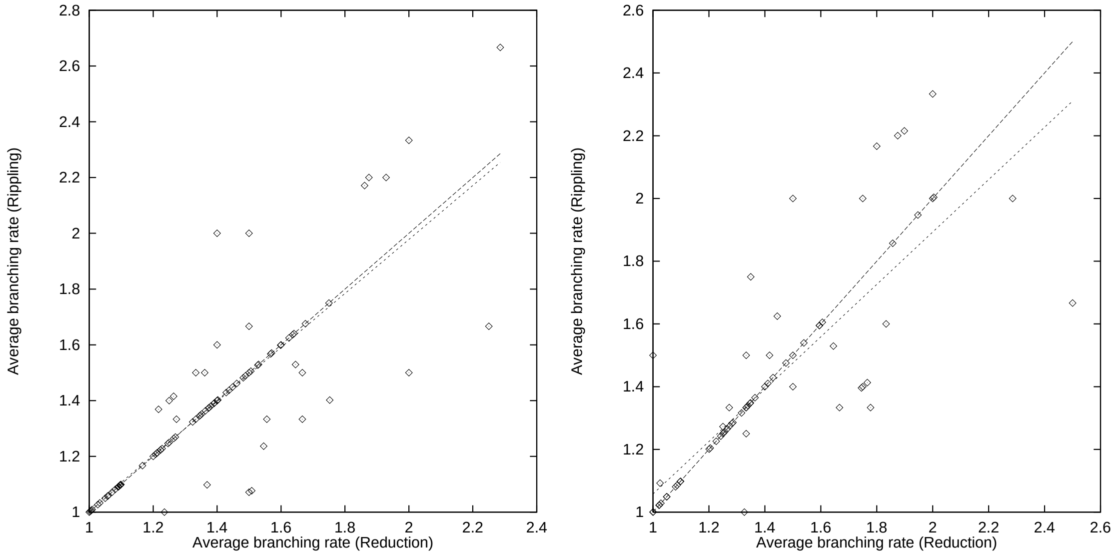
Figure 4: Some example wave-rules



The histograms above show the percentage of theorems proved by reduction for each class of theorems. The classes are defined by the CPU time of the rippling proof. These CPU times are plotted logarithmically. For instance, the column marked 100 on the x axis contains all theorems which took from 0–100 CPU milli-seconds to prove.

The left-hand histogram is for the minimal lemma configuration and the right-hand histogram is for the maximal lemma configuration.

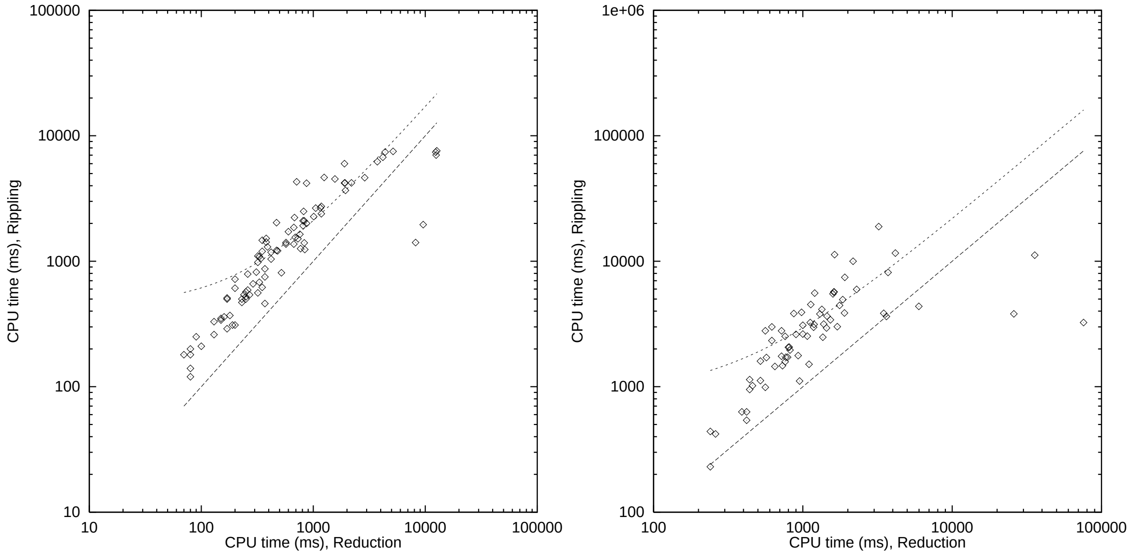
Figure 5: Success rate of reduction against theorem difficulty



Each point in the scattergrams above is a pair $\langle x, y \rangle$ where x is the average branching rate using reduction, and y is the average branching rate using rippling. There is one point for each theorem proved both by reduction and rippling. If these branching rates were the same for each proof then every point would lie on the main diagonal, given here, for reference, as a $45°$ line. Points above this line indicate proofs whose rippling branching rate was higher than the reduction one, and vice versa for points below the line. A regression line (dotted), giving the best fit to these points, is also shown.

The left-hand scattergram is for the minimal lemma configuration and the right-hand scattergram is for the maximal lemma configuration.

Figure 6: Comparison of branching rates



Each point in the scattergrams above is a pair $\langle x, y\rangle$ where x is the CPU time using reduction, and y is the CPU time using rippling. If these CPU times were the same for each proof then every point would lie on the main diagonal, given here, for reference, as a 45° line. Points above this line indicate proofs whose rippling CPU time was higher than the reduction one, and vice versa for points below the line.

The curved line is a best-fit regression line, ignoring the outlying points.

The left-hand scattergram is for the minimal lemma configuration and the right-hand scattergram is for the maximal lemma configuration.

Figure 7: Comparison of CPU times