

# Best-First Rippling

Moa Johansson, Alan Bundy and Lucas Dixon <sup>1</sup>

*School of Informatics  
University of Edinburgh  
Edinburgh, UK*

---

## Abstract

Rippling is a form of rewriting that guides search by only performing steps that reduce the syntactic differences between formulae. Termination is normally ensured by a measure that decreases with each rewrite step. Because of this restriction, rippling will fail to prove theorems about, for example, mutual recursion as steps that temporarily increase the differences are necessary. Best-first rippling is an extension to rippling where the restrictions have been recast as heuristic scores for use in best-first search. If nothing better is available, previously illegal steps can be considered, making best-first rippling more flexible than ordinary rippling. We have implemented best-first rippling in the IsaPlanner system together with a mechanism for caching proof-states that helps remove symmetries in the search space, and machinery to ensure termination based on term embeddings. Our experiments show that the implementation of best-first rippling is faster on average than IsaPlanner's version of traditional depth-first rippling, and solves a range of problems where ordinary rippling fails.

*Key words:* Proof-Planning, Theorem Proving, Rippling, Term Rewriting, Best-First Search

---

## 1 Introduction

Rippling is a heuristic used in automated theorem proving for reducing the differences between formulae [5]. It was originally designed for inductive proofs, where we aim to rewrite the inductive conclusion in such a way that we can apply the inductive hypothesis to advance the proof. Only rewrites that reduce differences and keep similarities are allowed. Rewrite rules can sometimes be applied both ways around and termination is guaranteed by defining a well-defined *ripple measure* that is required to decrease for each step of rewriting.

---

<sup>1</sup> Email: {moa.johansson, a.bundy, l.dixon}@ed.ac.uk

Rippling has been successfully used for automating proofs in a range of domains, for example, hardware verification [7], summing series [20], equation solving [12] and synthesis of higher-order programs [15].

If a rippling proof-attempt fails, *proof-planning critics* can use information from rippling to analyse the failure and suggest a patch such as a generalisation or conjecturing and proving a missing lemma [13]. Sometimes it may also be necessary to perform some rewrite that does not decrease the ripple measure or even temporarily increases it. This is necessary in, for example, many proofs involving mutually recursive functions [5] (§5.9). Ordinary rippling is not flexible enough to deal with this. Best-first rippling is suggested as a possible solution to these problems [5] (§5.14). The constraints of rippling are turned into a heuristic measure, allowing previously illegal steps if nothing better is available.

We have implemented best-first rippling in IsaPlanner [10], a proof-planner built on top of the interactive theorem-prover Isabelle [17]. IsaPlanner’s current implementation of higher-order rippling [11], has been extended to allow rewrites that normally would be regarded as illegal and discarded. Heuristic scores are assigned to the steps of rippling, and we use best-first search to pick the most promising new state (§4.1). Allowing previously illegal steps introduces a risk of non-termination, which is dealt with by introducing a check on term embeddings (§4.3). During development, we also discovered that the search space for best-first rippling often contained previously unobserved symmetries and developed methods for pruning such branches accordingly (§4.3). Using best-first search often caused the planner to conjecture and prove the same lemma several times. A new search strategy was developed which ensures each lemma is only attempted once (§4.4).

Our experiments show that best-first rippling can successfully solve a range of problems where the standard depth-first version of rippling fails. We do not find any problems that are solvable by ordinary rippling but not best-first rippling. Overall, the run-times for best-first rippling are also better than for ordinary depth-first rippling.

## 2 Rippling

Rippling works by identifying differences and similarities between two terms: the given and the goal. It then guides rewriting to reduce the differences, aiming to arrive at a sub-goal which can be justified by the given. Application of the given is called *fertilisation*.

The *skeleton* represents the parts of the goal that are similar to the given while *wave-fronts* represent the differences. A *wave-hole* denotes a sub-term inside a wave-front that belongs to the skeleton. In addition, if the given contains a universally quantified variable the corresponding position in the goal is called a *sink*. An example (from [11]) showing how the parts of a goal (here the inductive conclusion), can be annotated with respect to a given (the

inductive hypothesis) is shown below:

$$\begin{array}{l} \text{Given: } \forall b : \text{nat. } a + b = b + a \\ \text{Goal: } \boxed{\text{suc}(\underline{a})}^{\uparrow} + [b] = \boxed{\text{suc}(\underline{[b]} + a)}^{\downarrow} \end{array}$$

The wave-front is represented by a box, and the wave-hole by underlining. The skeleton, coming from the given,  $a + b = b + a$  corresponds to the parts of the goal that are either without annotation or underlined within the wave front. Note that the universally quantified variable  $b$  in the given becomes a sink in the goal, annotated by  $[b]$ . There are two strategies for making fertilisation possible, known as *rippling-in* and *rippling-out*. Rippling-out will try to remove the differences completely or move them out of the way. Wave-fronts are moved outwards until surrounding the entire term and the wave-hole contains an instance of the given. Rippling-in tries to move differences into sinks, corresponding to moving the wave-fronts inwards. The universally quantified variable in the given can then be instantiated to the contents of the sink and fertilisation is possible. The arrow of the wave-front indicates if the wave-front is to be rippled out ( $\uparrow$ ) or in ( $\downarrow$ ). In order to make the search space smaller, rippling-in is only allowed if there exists a sink or an outward wave-front inside the inward wave-front that eventually may absorb it. We lift this restriction for best-first rippling.

Rippling proceeds by applying rewrite-rules derived from definitions, theorems and axioms. To ensure that fertilisation will eventually be possible after rewriting, rippling requires the skeleton to be preserved by each step. Termination is guaranteed by defining a *ripple-measure*, based on the positions of the wave-fronts, which is required to decrease for each rewrite step. This also helps reduce the size of the search space, and makes it possible to allow equations to be applied in both directions, unlike traditional rewriting where only one direction is allowed. There are different implementations of ripple measures. Here, we will use a measure based on the sum of distances from each outward wave-front to the top of the term tree and from each inward wave-front to the nearest sink or nearest embedded outward wave-front. This measure will clearly decrease as outward wave-fronts are moved towards the top of the term-tree, and inward wave-fronts towards a sink further down.

*Example:*

As an example illustrating how rippling moves the wave-front outward to allow fertilisation, consider the step case goal of the inductive proof of the commutativity of addition, where the given is the inductive hypothesis. Note that the sinks have been omitted to reduce clutter, as the proof only uses the rippling-out strategy.

Given:  $\forall b : nat. a + b = b + a$

Goal:  $\boxed{suc(\underline{a})}^\uparrow + b = b + \boxed{suc(\underline{a})}^\uparrow$

with the rules <sup>2</sup>:

$$\begin{aligned} suc(X) + Y &\equiv suc(X + Y) \quad (1) & suc(X) = suc(Y) &\equiv X = Y \quad (3) \\ X + suc(Y) &\equiv suc(X + Y) \quad (2) \end{aligned}$$

where  $\equiv$  denotes the equality which can be used for rewriting. The proof of the step-case goal will then proceed as follows:

$$\begin{aligned} \boxed{suc(\underline{a})}^\uparrow + b &= b + \boxed{suc(\underline{a})}^\uparrow \\ &\Downarrow \text{by (1)} \\ \boxed{suc(\underline{a+b})}^\uparrow &= b + \boxed{suc(\underline{a})}^\uparrow \\ &\Downarrow \text{by (2)} \\ \boxed{suc(\underline{a+b})}^\uparrow &= \boxed{suc(\underline{b+a})}^\uparrow \\ &\Downarrow \text{by (3)} \\ &a + b = b + a \end{aligned}$$

Notice how each ripple-rewrite moves the wave-front outwards until we arrive at a state where the goal contains an instance of the given. We can now simply replace this instance with ‘True’ and conclude the proof. This is called *Strong fertilisation*.

If rule 3 was missing, there would have been no more rewrites possible after the state:  $\boxed{suc(\underline{a+b})}^\uparrow = \boxed{suc(\underline{b+a})}^\uparrow$ . We call such a state *blocked*.

However, in many cases it is still possible to apply the given using substitution. This is called *weak-fertilisation*. In the above example, this rewrites the blocked goal to  $suc(b + a) = suc(b + a)$ . The resulting goal is true by reflexivity. In situations where rippling is blocked but weak fertilisation is not possible, we can attempt to apply a critic [13].

<sup>2</sup> Following the convention for dynamic rippling (§2.1), the rules have not been annotated as wave-rules.

### 2.1 *Static and Dynamic Rippling*

There are two main approaches for implementing rippling: *static* and *dynamic*. They represent and handle annotations in different ways. Rippling as described by Bundy et al. [5] will be referred to as *static rippling*. In static rippling, the rewrite-rules are annotated before rippling starts in such a way that they will ensure measure decrease and skeleton preservation. The annotated rules are called *wave-rules* and can be applied to any goal with matching annotations. Basin and Walsh provide a proof of termination for first-order static rippling [1]. They represent annotations as function-symbols at the object level of the goal. This requires a special notion of substitution that erases annotations outside the skeleton when replacing terms, otherwise illegal annotations may be produced. Furthermore, object level annotations are not stable over  $\beta$ -reduction in a higher-order setting [19]. To overcome these problems, the use of *dynamic rippling* [8,11], and *term embeddings*, for representing annotations [19,8], have been introduced. In dynamic rippling, annotations are stored separately from the goal and rewrite rules are not annotated in advance. Instead, all ways of rewriting the goal with a particular rule are generated after which the annotations are re-computed and measure decrease and skeleton preservation checked. This means that no specialised version of substitution is needed.

Dynamic rippling is more suitable as a starting point for our best-first rippling implementation because it initially generates all possible rewrites, including new subgoals that are non-skeleton preserving and non-measure decreasing. These would normally be discarded, but we will adapt rippling to instead assign them heuristic scores.

## 3 Proof-Planning

Rippling has been implemented and used within the context of *proof-planning* [3,6]. Proof planning is a technique for guiding the search for a proof in automated theorem proving by exploiting that ‘families’ of proofs, for example inductive proofs, share a similar structure. Instead of searching the large space of an underlying theorem-prover, the proof-planner can reason about the applicable methods for a conjecture and construct a *proof-plan* consisting of a tree of *tactics*. A tactic is some sequence of steps, known to be sound, that are used for solving a particular problem in a theorem-prover, such as simplification, induction etc.

Recently, a higher-order version of dynamic rippling has been implemented in IsaPlanner [10,9], a proof planner written in Standard ML for the interactive theorem-prover Isabelle [17]. In IsaPlanner, proof planning is interleaved with execution of the proof in Isabelle giving IsaPlanner access to Isabelle’s powerful tactics. The resulting proof-plan is then represented as a proof script in the Isar language [21], executable in Isabelle and argued to be more readable

than the output from earlier proof-planners such as  $\lambda Clam$  [18]. Rippling in IsaPlanner has also been shown to be considerably faster than in  $\lambda Clam$  [11].

## 4 Best-First Rippling

Ordinary rippling requires each step in the rippling-process to satisfy the restrictions of measure decrease and skeleton preservation, otherwise the step is regarded as invalid. There are however a number of occasions where these ‘invalid’ ripple-steps would be useful or necessary for the success of rippling. In proofs involving mutually recursive functions, the skeleton might be temporarily disrupted but restored in a later step (see for example [5], §5.9). Another example is a proof where it is necessary to ‘unblock’ rippling by performing rewrites inside the wave front [4], which might lead to a temporary increase in the ripple-measure.

In best-first rippling, the measure decrease and skeleton preservation requirements are, instead of being strictly enforced, reflected in a heuristic score. The heuristic prefers decreasing ripple measures and skeleton preservation but previously invalid steps can then be considered if nothing better is available.

To realise best-first rippling we need dynamic rippling and best-first search. We must consider all rewrites at any given state, evaluate their heuristic scores and compare them with all other open states in the search space. The open states are stored in an *agenda*, a list of states in increasing order of heuristic score. The state with the lowest score is thus the most promising one from which to continue rippling. IsaPlanner implements dynamic rippling and has a generic version of best-first search, making it a suitable platform for implementing best-first rippling. We follow IsaPlanner’s modular implementation of rippling, allowing support for different versions of best-first rippling with different notions of annotations and ripple measures simultaneously.

### *Example*

As an example illustrating the need for best-first rippling consider the proof of the theorem  $even(suc(suc(0)) * n)$ , which will require both non-measure decreasing and non-skeleton preserving steps. The proof uses the following rules from the definitions of addition, multiplication and for the mutually recursive even and odd functions:

$$even(suc(X)) \equiv odd(X) \quad (4) \qquad X + suc(Y) \equiv suc(X + Y) \quad (7)$$

$$odd(suc(X)) \equiv even(X) \quad (5) \qquad X * suc(Y) \equiv (X * Y) + suc(X) \quad (8)$$

$$X + 0 \equiv X \quad (6)$$

Our given and goal gives the following rippling sequence:

$$\begin{array}{l}
 \text{Given: } \text{even}(\text{suc}(\text{suc}(0)) * n) \\
 \text{Goal: } \text{even}(\text{suc}(\text{suc}(0)) * \boxed{\text{suc}(n)}^\uparrow) \quad \text{Measure: 2} \\
 \Downarrow \text{by (8)} \\
 \text{even}(\boxed{\text{suc}(\text{suc}(0)) * n + \text{suc}(\text{suc}(0))}^\uparrow) \quad \text{Measure: 1} \\
 \Downarrow \text{by (7)} \\
 \text{even}(\boxed{\text{suc}(\text{suc}(\text{suc}(0)) * n + \text{suc}(0))}^\uparrow) \quad \text{Measure: 1} \\
 \Downarrow \text{by (7)} \\
 \text{even}(\boxed{\text{suc}(\text{suc}(\text{suc}(\text{suc}(0)) * n + 0))}^\uparrow) \quad \text{Measure: 1} \\
 \Downarrow \text{by (6)} \\
 \text{even}(\boxed{\text{suc}(\text{suc}(\text{suc}(\text{suc}(0)) * n))}^\uparrow) \quad \text{Measure: 1} \\
 \Downarrow \text{by (4)} \\
 \text{odd}(\text{suc}(\text{suc}(\text{suc}(0)) * n)) \quad \text{No Skeleton} \\
 \Downarrow \text{by (5)} \\
 \text{even}(\text{suc}(\text{suc}(0)) * n) \quad \text{Measure: 0}
 \end{array}$$

Strong fertilisation is now applicable as the wave-front has been fully rippled out leaving the ripple measure 0. All but the first and last step in the rippling proof do not change the ripple-measure as the rewrites are applied to terms inside the wave-front. In the second last step it is necessary to break the skeleton as *even* and *odd* are defined in terms of each other.

#### 4.1 Best-First Heuristic

Best-first rippling requires a heuristic evaluation function for deciding which state is the most promising to evaluate next in the rippling process. Valid ripples should be considered before non-measure decreasing or non-skeleton preserving steps. The ripple measure gives an indication of how far we are from being able to apply fertilisation and conclude the proof.

We have used IsaPlanner's sum-of-distance ripple measure during development and testing. Rippling with this measure has been shown to perform better than with other kinds of measures [9]. As mentioned earlier, best-first rippling has however been implemented in a modular fashion, allowing use of any type of ripple measure.

IsaPlanner’s best-first search function expects to be supplied with a heuristic order function used for keeping the agenda sorted in increasing order. It is therefore not necessary to compute and store explicit numerical scores for the states, just determine their relative ordering. Our heuristic function for comparing reasoning states can be summarised as follows:

- States to which strong fertilisation can be applied are always preferred over continued rippling.
- Skeleton preserving states are always given a better score than non-skeleton preserving states.
- When both states preserve the skeleton, the state with the best ripple measure is given the lower score. If the states have the same ripple measure, they are given equal heuristic scores.
- If neither state preserves the skeleton, the reasoning state with the smallest goal-term scores better.

Strong fertilisation should be preferred over everything else as it applies the inductive hypothesis and typically concludes the proof. Skeleton preservation. States in which the skeleton embeds are ordered based on the ripple-measures. In comparing ripple-measures, we need to take into account that, as IsaPlanner employs dynamic rippling, each reasoning state might have several ripple measures, one for each way a skeleton embeds. IsaPlanner also supports rippling with multiple skeletons, each of which may embed in different ways. To compare such states, the best ripple-measure of each one is given to the heuristic function for best first search.

The heuristic also handles non-rippling states, such as setting up a new rippling attempt or applying fertilisation. If the current state becomes blocked, fertilisation or critics are attempted before more rippling elsewhere in the search space. This because these non-rippling steps will either result in a solution, if fertilisation is successful, or if a critic begins proving a lemma, a new ripple-state will be created to which our standard heuristic is applicable.

Because best-first rippling does not become blocked as often as ordinary rippling does, we experimented with introducing some heuristic measure allowing the application of weak fertilisation and critics before we run out of applicable rules. We developed a variant of best-first rippling where weak-fertilisation and IsaPlanner’s lemma calculation critics were applied eagerly to states where none of the children were skeleton-preserving, i.e. the state would have been blocked in ordinary rippling. The non-skeleton preserving children are also kept in the agenda, but given a worse heuristic score than to weak fertilise and/or conjecture a lemma. Results from these experiments are discussed in §5.

## 4.2 Complications with Best-First Rippling

The price for the greater flexibility of best-first rippling is that the search space is considerably larger. The increased number of possibilities to continue rippling also means that rippling will rarely become blocked, which is when applying fertilisation or critics would normally be considered. Furthermore, allowing non-measure decreasing and non-skeleton preserving steps means that best-first rippling will lose the guarantee for termination, as it is possible to become stuck in a loop by applying the same rewrite-rule in opposite directions.

Another source of potential non-termination is rewrite rules that introduce additional term-structure and do not decrease the ripple measure. As an example, consider the annotated term:  $even(\boxed{suc(suc(\underline{n}))})^\uparrow$ ). We can either apply rewrite-rule 4 from left to right or, as rewrites are allowed in both directions, rule 5 from right to left. The latter would give the result  $odd(suc(suc(suc(n))))$  where  $even$  has been transformed into  $odd$  by adding a successor-function rather than removing one. Consider now applying rule 4 from right to left, which produces a goal that adds another successor function:  $even(\boxed{suc(suc(suc(suc(\underline{n}))))})^\uparrow$ ). Subsequent bad applications could keep alternating between  $even$  and  $odd$ , each time adding another successor-function and hence never terminating. Our solution to these problems uses caching of the visited states as well as an embedding check, as discussed in §4.3.

## 4.3 Termination and Reduction of Search Space Size

As mentioned above, allowing rippling with non-measure decreasing wave-rules means that best-first rippling is no longer guaranteed to terminate. This is dealt with by introducing an *embedding check*, as used in IsaPlanner’s lemma conjecturing ([9] Chapter 9). If a previous goal-term embeds into the new sub-goal, on the same branch of the search space, it is removed. This filters rewrites that would otherwise cause divergence. Kruskal’s Theorem [14], states that there exists no infinite sequence of trees such that an earlier tree does not embed into a later tree. Therefore, the embedding check will restore termination, which was lost as we relaxed the restriction of ripple-measure decrease. Following the approach in [11] that caches intermediate subgoals, we also filter out any new subgoals that are identical to ones previously seen anywhere in the search tree, thereby pruning symmetric branches. As can be seen from the results in §5, the combination of these techniques appears to work well in practice. Fig. 1 presents an example that illustrates how unproductive branches are pruned to reduce the size of the search space.

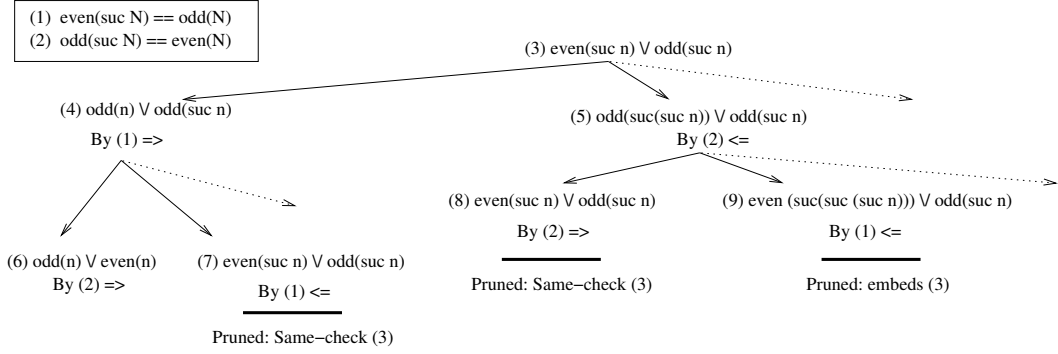


Fig. 1. A partial search tree using best-first rippling that shows how branches are pruned to avoid loops and redundant rewrites. Note that equations can be applied in both directions, indicated above by  $\leq$  and  $\geq$ .

#### 4.4 Delaying Parts of the Search

We discovered that a common problem arising when using best-first or breadth-first search for rippling is that the same lemma might be conjectured independently at different places in the search space, causing the planner to pursue several simultaneous attempts on the same lemma.

In IsaPlanner’s standard depth-first rippling this is not an issue. When a blocked state is encountered, a lemma is conjectured and proved before backtracking to try more rippling in the original proof attempt. Lemmas that have already been proved to be true (or failed) are cached, allowing later blocked states requiring the same lemma to use the previous result, thus saving time by avoiding symmetric parts of the search space. This happens frequently when a rippling attempt becomes blocked and requires a lemma which we already have started a proof of elsewhere. In such situations, we want to prevent beginning a second attempt. Instead, the second reasoning state should be suspended until the lemma has been proved. After the lemma is proved, not only the state from which it was originally conjectured, but also any other states waiting for that particular lemma, should be resumed.

Beginning several attempts of the same lemma was one of the major sources for inefficiencies in our initial implementation of best-first rippling. The problem was first tackled by giving rippling in a lemma attempt a better heuristic score than rippling the original conjecture. This is however not always desirable: if a bad lemma is conjectured, we do want the option to abandon it and explore other possibilities. We chose to instead create a new generic search strategy in IsaPlanner. This strategy inspects all new states and may temporarily remove them from the agenda if they are marked to be delayed. Similarly, the strategy checks if the current state wishes to resume some delayed states, which are then returned to the agenda. To implement this, IsaPlanner’s lemma conjecturing machinery was augmented with a cache for lemmas-in-progress in addition to the existing caching of results for proof attempts of lemmas [9] (Chapter 9). The lemma conjecturing critic inspects

the cache and if an attempt is already in progress then the reasoning state is marked as delayed and not evaluated further until the proof attempt of the relevant lemma is finished.

## 5 Evaluation and Results

Best-first rippling has been evaluated by comparing it to IsaPlanner’s implementation of ordinary rippling, which uses depth-first search. We measured the number of successfully solved problems as well as run-times on both successful and failed proof attempts. Our test-problems included a set of benchmarks for IsaPlanner that consists of 55 theorems in Peano arithmetic and about lists. These are typical of the shared test problems of the inductive theorem proving community and most are also found in the Boyer-Moore corpus. The experiments were conducted on a standard 2 GHz Intel Pentium4 PC with 512 MB of memory running Isabelle2005. Each problem had a timeout limit of 30 seconds.

Best-first rippling performs some extra work computing heuristic scores, so we expected it to be slower than ordinary depth-first rippling. The benchmarks also included a range of non-theorems, allowing us to test the robustness of best-first rippling. Ideally, we would like to exhaust the search space quickly when no solution can be found, rather than see non-termination. In addition to IsaPlanner’s benchmarks, we also tested a set of 39 problems where we would expect to see the full benefits of best-first rippling, including proofs about mutually recursive functions, proofs involving destructor-style functions (such as a predecessor function for Peano arithmetic) and proofs where measure increasing steps are required. The mutually recursive problems typically require induction schemes reflecting the depth of the nested recursive function definitions. As an example, recall the mutually recursive definition of *even* and *odd* from §4. The two functions are defined in terms of each other so we use two-step induction. In these cases, the induction scheme was supplied manually to IsaPlanner/Isabelle, as inference of induction schemes is currently limited to standard recursively defined data-types.

We also compared a version of best-first rippling that applies critics when it is blocked, with a version that eagerly tries to apply critics or weak-fertilisation when no more skeleton-preserving steps are available. This was expected to indicate whether applying critics is more efficient than searching the larger space arising from allowing non-skeleton preserving steps.

The number of successful proofs for the three versions of rippling are displayed in Fig. 2. Both best-first rippling with eager application of critics and the variant without it, managed to find proofs for 76 of the 94 theorems. On IsaPlanner’s benchmark set, ordinary depth-first rippling succeeded in finding 40 proofs, compared to 41 for best-first rippling. The extra theorem proved by

best-first rippling was  $rev(l) = qrev(l, [])$ <sup>3</sup>, a problem expected to fail as Isa-Planner lacks a generalisation critic for accumulator variables, but here solved as a side effect of our caching mechanism<sup>4</sup>. On the additional set, ordinary rippling proved only 10 theorems compared to 35 for best-first, which was expected as these were chosen from classes of problems known to be difficult for ordinary rippling.

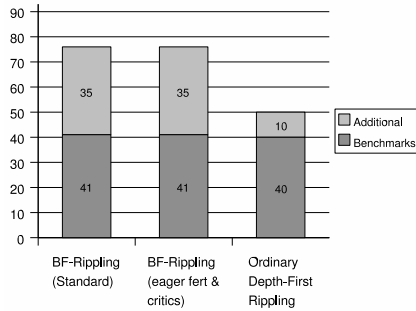


Fig. 2. Number of successes on the 94 theorems in the test set (55 benchmarks and 39 additional)

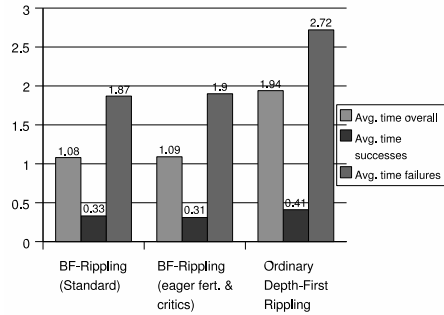


Fig. 3. Average run-times in seconds.

Fig. 3 shows the average run-times for proof-attempts while Fig. 4 shows the time spent on each proof for best-first and ordinary rippling. Ordinary rippling is slightly faster on most problems both techniques can solve but the differences are small. Best-first rippling is however faster on average, due to a few outliers for ordinary rippling. Ordinary rippling fails or times out more often than best-first rippling. As a consequence, best-first rippling is faster than depth-first rippling overall, and also spends less time on conjectures it cannot prove, including non-theorems, thanks to the caching and embedding-check.

Between the two variants of best-first rippling, the differences in runtime appears to be small. Conjecturing lemmas eagerly when no skeleton-preserving steps are available appears to make little difference to the run-times of the mutually recursive problems in our test set.

To summarise the results; best-first rippling proves a number of theorems where ordinary rippling is too restricted to succeed, as expected. Despite the larger search-space of best-first rippling the differences in run-times compared to ordinary rippling are small. Best-first rippling appears to be more robust when presented with non-theorems, less time is spent on failed proof-attempts compared to ordinary rippling. The full collection of test problems, results and

<sup>3</sup>  $rev$  and  $qrev$  are the recursive and tail recursive list reversal functions. We wish to prove that they produce the same result.

<sup>4</sup> The interested reader can find the proof on the project website: <http://dream.inf.ed.ac.uk/projects/bfripping>.

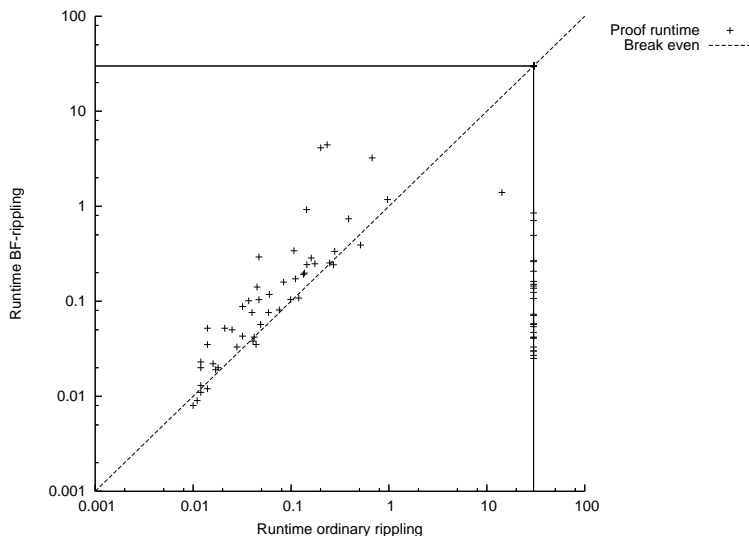


Fig. 4. Each point in the scatter-plot represents a conjecture, with the x-value being the runtime for ordinary rippling and the y-value the runtime of standard best-first rippling. The vertical and horizontal lines marks the timeout limit of 30 seconds. Failed proof-attempts have also been plotted along these lines for clarity. A logarithmic scale is used for better visualisation.

function definitions can be found on the project website<sup>5</sup>. The source code is available from the IsaPlanner website<sup>6</sup>.

## 6 Related Work

The main difference between our work and IsaPlanner’s previous implementations of ordinary depth-first rippling [9], is that best-first rippling relaxes the requirements that each state must preserve the skeleton and decrease the ripple measure. These requirements guarantee the termination of ordinary rippling, something that would normally be lost for best-first rippling. Our implementation instead uses mechanisms for caching visited states to prune the search space of many symmetric branches and a check on term embeddings to restore termination and avoid loops.

Brotherston implemented a greedy best-first methodical in the  $\lambda$ Clam proof planner [2] that supports dynamic re-ordering of child nodes in the search space during depth first search<sup>7</sup>. Applied to rippling, this strategy suffers from problems as it does not allow switching focus to the most promising area of the search. Our best-first search strategy is not depth-first which supports switching the focus to different parts of the search tree.

Manning et al. present an implementation of best-first proof-planning in *Clam* [16]. A best-first heuristic is employed to make choices between three

<sup>5</sup> <http://dream.inf.ed.ac.uk/projects/bfripping>

<sup>6</sup> <http://sourceforge.net/projects/isaplanner/>

<sup>7</sup> Personal communication: internal Blue Book Note series, numbers 1405, 1409, 1425

different proof planning methods; generalisation, simplification and induction. Our work differs from that of Manning as we are applying best-first search *within* the rippling technique.

## 7 Further Work

Best-first rippling managed to prove the conjecture  $rev(l) = grev(l, [])$  as a side effect of the caching mechanism, but fails to prove more complicated theorems involving similar tail-recursive functions. Such problems can be solved using a critic to analyse the failed proof attempt in order to suggest a generalisation. Another limitation of the current implementation is that the user is required to specify the induction scheme if a non-standard one is required. The *Clam* proof-planner had a number of critics for finding lemmas, forming generalisations, case-splits and revising the induction scheme [13]. IsaPlanner has currently only one critic, namely lemma calculation. We plan to implement additional critics in IsaPlanner, further increasing the number of theorems that can be proved automatically.

The embedding and caching techniques we have discussed could also benefit ordinary rippling. In particular, pruning states already seen from the search space removes symmetric branches which would potentially improve run-times.

Our test-set mainly consisted of relatively easy theorems. Further experiments will evaluate best-first rippling on harder problems. We also plan to undertake a larger comparison between rippling and regular rewriting.

## 8 Conclusions

We have shown that our implementation of best-first rippling is able to automatically prove a number of theorems where IsaPlanner’s previous implementation of depth-first rippling fails, for example, proofs about mutually recursive functions and proofs requiring a temporary increase in the ripple measure. Rippling has been allowed more flexibility by recasting the measure decrease and skeleton preservation requirements into heuristic scores. However, in allowing these steps we do lose the guarantee of termination for rippling. Our solution is to introduce an embedding check (§4.3), where new subgoals in which we can embed previously seen goals on the same branch are pruned. This cuts out branches where subsequent applications of non-skeleton preserving rewrites leads to divergence as described in §4.2 and restores termination.

Using best-first search rather than depth-first search means that it is possible to switch between rippling in a lemma attempt and rippling in the original proof, depending on which seems more promising. This often gave rise to the same lemma being conjectured from different blocked states. Our new search strategy suspends any states requiring a lemma for which a proof is already in progress. When a lemma is proved, all states waiting for it are resumed.

Our test results show that best-first rippling not only is capable of solving

a range of problems not solvable by ordinary rippling, but also has faster run-times overall thanks to the combination of efficiency measures described above and the guidance from best-first search. We also compared two versions of best-first rippling to verify if it is beneficial to apply critics before best-first rippling is blocked, as best-first rippling might not become blocked as often as ordinary rippling due to the larger search space. However, we found that applying the currently available critics in this manner made little difference.

## References

- [1] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 1-2(16):147–180, 1996.
- [2] J. Brotherston and L. Dennis. *LambdaClam v.4.0.1 User/Developer’s manual*. Available online: <http://dream.inf.ed.ac.uk/software/lambda-clam/>.
- [3] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction*, pages 111–120, 1988.
- [4] A. Bundy. The termination of rippling + unblocking. Informatics research paper 880, University of Edinburgh, 1998.
- [5] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
- [6] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1992.
- [7] F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In *First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 94–108. Springer Verlag, 1996.
- [8] L. A. Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. Technical Report Computer Science No. NOTTCS-WP-SUB-0503230955-5470, University of Nottingham, 2005.
- [9] L. Dixon. *A proof-planning framework for Isabelle*. PhD thesis, School of Informatics, University of Edinburgh, 2005.
- [10] L. Dixon and J. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE’03*, pages 279–283, 2003.
- [11] L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *Proceedings of TPHOLs’04*, pages 83–98, 2004.
- [12] D. Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997.
- [13] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.

- [14] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 1960.
- [15] D. Lacey, J. Richardson, and A. Smaill. Logic program synthesis in a higher-order setting. *Computational Logic*, 1861:87–100, 2000.
- [16] A. Manning, A. Ireland, and A. Bundy. Increasing the versatility of heuristic based theorem provers. In A. Voronkov, editor, *International conference on Logic Programming and Automated Reasoning LPAR'93*, number 698 in Lecture Notes in Artificial Intelligence, pages 194–204. Springer Verlag, 1993.
- [17] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A proof assistant for higher-order logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.
- [18] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with Lambda-Clam. In *15th International Conference on Automated Deduction*, number 1421 in LNAI, pages 129–133, 1998.
- [19] A. Smaill and I. Green. Higher-order annotated terms for proof search. In *Theorem Proving in higher-order logics: 9th international conference*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–413. Springer Verlag, 1996.
- [20] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In *11th Conference on Automated Deduction*, number 607 in LNCS, pages 325–339. Springer Verlag, 1992.
- [21] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer Verlag, 1999.