Best-First Rippling

Moa Johansson, Alan Bundy, and Lucas Dixon

School of Informatics, University of Edinburgh,
Appleton Tower, Crichton St, Edinburgh EH8 9LE, UK
{moa.johansson, a.bundy, lucas.dixon}@ed.ac.uk

Abstract. Rippling is a form of rewriting that guides search by only performing steps that reduce the differences between formulae. Termination is normally ensured by a defined measure that is required to decrease with each step. Because of these restrictions, rippling will fail to prove theorems about, for example, mutual recursion where 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.

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 be applied both ways around and termination is guaranteed by defining a *ripple measure* that is required to decrease for each step of rewriting. Rippling has been successfully used for automating proofs in a range of domains, for example, hardware verification [8], summing series [21], equation solving [13] and synthesis of higher-order programs [16].

Rippling is however not guaranteed to succeed. *Proof-planning critics* has been proposed as a solution. Critics analyse failed proof attempts to suggest patches such as a generalisation or conjecturing and proving a missing lemma [14]. Sometimes it may also be necessary to perform a rewrite that does not decrease the ripple measure or temporarily increases the differences between given and goal. This is necessary in, for example, 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]

O. Stock and M. Schaerf (Eds.): Aiello Festschrift, LNAI 4155, pp. 81–98, 2006.

(§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 [11], a proof-planner built on top of the interactive theorem-prover Isabelle [18]. IsaPlanner's current implementation of higher-order rippling [12], has been expanded 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 (§4.2). 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 rippling often contained 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 delays steps waiting for the same lemma (§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, on average, better than for ordinary depth-first rippling, despite the potentially larger search space.

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 [12]) 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[1]:

$$\begin{aligned} &\mathit{Given}: \quad \forall b : \mathit{nat}.\ a + b = b + a \\ &\mathit{Goal}: \quad \boxed{\mathit{suc}(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \boxed{\mathit{suc}(\underline{\lfloor b \rfloor + a})}^{\downarrow} \end{aligned}$$

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 $\lfloor b \rfloor$. There are two strategies for making fertilisation possible, known

[1] Note this is *one* way of annotating this goal; in general a goal may be annotated in several different ways.

as *rippling-in* and *rippling-out*. Rippling-out will try to remove the differences completely or move them out of the way, so that the wave-front surrounds the entire term and the wave-hole contains an instance of the given. Rippling-in tries to move differences into sinks. 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 equations, definitions, theorems and lemmas. To ensure that fertilisation will eventually be possible after rewriting, rippling requires the skeleton to be preserved between 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 make it possible to allow rewrite-rules 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. 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.

$$\begin{aligned} &\mathit{Given}: \quad \forall b : \mathit{nat}.\ a + b = b + a \\ &\mathit{Goal}: \quad \boxed{\mathit{suc}(\underline{a})}^{\uparrow} + b = b + \boxed{\mathit{suc}(\underline{a})}^{\uparrow} \end{aligned}$$

with the rules [2]:

$$\mathit{suc}(X) + Y \equiv \mathit{suc}(X + Y) \tag{1}$$

$$X + \mathit{suc}(Y) \equiv \mathit{suc}(X + Y) \tag{2}$$

$$\mathit{suc}(X) = \mathit{suc}(Y) \equiv X = Y \tag{3}$$

The proof of the step-case goal will proceed as follows:

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

$$\Downarrow \mathit{by\ rule}\ 1$$

[2] Following the convention for dynamic rippling (§2.1), the rules have not been annotated as wave-rules in static rippling.

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

$$\Downarrow \textit{ by rule } 2$$

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

$$\Downarrow \textit{ by rule } 3$$

$$a + b = b + a$$

$$\Downarrow \textit{ Fertilise}$$

$$True$$

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.*

In the case that rule 3 were 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 say that the state is *blocked.* It is however still possible to apply the given using substitution, which rewrites the blocked goal to $suc(b+a) = suc(b+a)$. This is called *weak-fertilisation.* 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 [14].

2.1 Static and Dynamic Rippling

There are two main approaches for implementing rippling: static and dynamic rippling. 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. Note that a single theorem or definition may give rise to several wave-rules. Basin and Walsh give a formal calculus for static rippling in first-order logic and provide a proof of termination [1]. They represent annotations as function-symbols at the object level of the goal. The object level annotations require a special notion of substitution as standard substitution may produce illegal annotations. Another problem with static rippling in a higher-order setting, as pointed out by Smaill and Green [20], is that the object level annotations are not stable over β-reduction. This makes it impossible to pre-annotate higher-order rewrite rules as they may turn out to be non-skeleton preserving. To overcome these problems, the use of *dynamic rippling* [9,12], and *term embeddings*, for representing annotations [20,9], 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.

The *Clam* proof-planner [7], written in Prolog, and the higher-order λ*Clam* [19], written in λProlog, both implement rippling. The Clam-family of proof planners uses a set of *methods* and *methodicals*. Methods specify what conditions have to be true for the method to be applicable to a goal and what will be true after the method has been applied. They also carry a reference to the corresponding tactic that will be used in the theorem-prover when the proof plan is executed. Methodicals combine several atomic methods into larger compound methods.

3.1 IsaPlanner

Recently, a higher-order version of dynamic rippling has been implemented in IsaPlanner [11,10], a proof planner written in Standard ML for the interactive theorem-prover Isabelle [18]. 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 [22], executable in Isabelle and argued to be more readable than the output from earlier proof-planners such as λ*Clam*. Rippling in IsaPlanner has also been shown to be considerably faster than in λ*Clam* [12].

As opposed to the Clam-family of proof planners, IsaPlanner plans the proof through a series of *reasoning states*. Each reasoning state contains the partial proof plan constructed so far, the next reasoning technique to be applied and contextual information. The reasoning techniques are defined to be functions from a reasoning state to a sequence of new reasoning states. This sequence represents all the ways the technique can be applied to its input state. The contextual information contains knowledge acquired during proof planning, including information about rippling-annotations and skeletons.

IsaPlanner supports several search strategies, including a generic best-first search. Search strategies can be applied globally or locally over a reasoning technique.

IsaPlanner's implementation of rippling is designed in a modular fashion to be easily extendable and can support different versions of rippling with different notions of annotations and ripple measures simultaneously. Our best-first rippling implementation is a module defined in terms of the module for ordinary rippling, thereby making best-first rippling available for any of IsaPlanner's versions of rippling.

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 smaller 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 heuristics scores and compare them with all other open states in the search. The state with the lowest score is 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.

Example: Breaking the Skeleton. As an example, consider the following problem with mutually recursive definitions of even and odd (here called $evenM$ and $oddM$).

$$\begin{aligned} &\mathit{Given}: \quad evenM(n) \vee oddM(n) \\ &\mathit{Goal}: \quad evenM(\boxed{suc(suc(\underline{n}))}^{\uparrow}) \vee oddM(\boxed{suc(suc(\underline{n}))}^{\uparrow}) \end{aligned}$$

with the rules:

$$evenM(0) \equiv True \tag{4}$$

$$evenM(suc(X)) \equiv oddM(X) \tag{5}$$

$$oddM(0) \equiv False \tag{6}$$

$$oddM(suc(X)) \equiv evenM(X) \tag{7}$$

This gives the following best-first rippling proof using two-step induction:

$$evenM(\boxed{suc(suc(\underline{n}))}^{\uparrow}) \vee oddM(\boxed{suc(suc(\underline{n}))}^{\uparrow})$$

$$\Downarrow \textit{ by rule } 5$$

$$oddM(suc(n)) \vee oddM(suc(suc(n)) \tag{8}$$

$$\Downarrow \textit{ by rule } 7$$

$$evenM(n) \vee oddM(\boxed{suc(suc(\underline{n}))}^{\uparrow})$$

$$\Downarrow \textit{ by rule } 7$$

$$evenM(n) \vee evenM(suc(n)) \tag{9}$$

$$\Downarrow \textit{ by rule } 5$$

$$evenM(n) \vee oddM(n)$$

Fertilisation is now possible. Note that the skeleton is disrupted in steps 8 and 9 (the subgoals are therefore not annotated), but restored in the following step. These steps are necessary for the completion of this proof but would not be allowed in ordinary rippling.

Example: Non-measure Decrease Required. The proof of the theorem $evenR(suc(suc(0)) * n)$, taken from [4], requires us to modify the argument to the even-function before we can apply fertilisation. As a result, the rewrites will not move the wave-front, only rearrange the terms inside it, and the measure will stay the same over a number of steps. Note that this is the two-step recursively defined even-function, (referred to as $evenR$) as opposed to the mutually recursive version defined above.

The proof uses the following rules from the definitions of addition, multiplication and for the two-step recursive version of $evenR$:

$$evenR(suc(suc\ X)) \equiv evenR(X) \tag{10}$$

$$X + 0 \equiv 0 \tag{11}$$

$$X + suc(Y) \equiv suc(X + Y) \tag{12}$$

$$X * suc(Y) \equiv (X * Y) + suc(Y) \tag{13}$$

Our given and goal gives the following rippling sequence (with the sum-of distance ripple measure given for each step):

$$Given:\ evenR(suc(suc(0)) * n)$$

$$Goal:\ evenR(suc(suc(0)) * \boxed{suc(\underline{n})}^{\uparrow})\quad Measure:\ 2$$

$$\Downarrow\ by\ rule\ 13$$

$$evenR(\boxed{\underline{suc(suc(0)) * n} + suc(suc(0))}^{\uparrow})\quad Measure:\ 1$$

$$\Downarrow\ by\ rule\ 12$$

$$evenR(\boxed{suc(\underline{suc(suc(0)) * n} + suc(0))}^{\uparrow})\quad Measure:\ 1$$

$$\Downarrow\ by\ rule\ 12$$

$$evenR(\boxed{suc(suc(\underline{suc(suc(0)) * n} + 0))}^{\uparrow})\quad Measure:\ 1$$

$$\Downarrow\ by\ rule\ 10$$

$$evenR(\boxed{\underline{suc(suc(0)) * n} + 0}^{\uparrow})\qquad Measure:\ 1$$

$$\Downarrow\ by\ rule\ 11$$

$$evenR(suc(suc(0)) * n)\qquad Measure:\ 0$$

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.

4.1 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 loose the guarantee for termination, as it is possible to become stuck in a loop by applying the same rewrite-rule in opposite directions.

Mutually recursive functions are another source of potential non-termination as it is possible to apply a non-skeleton preserving rewrite rule in a direction such that the subgoal gets larger and larger. Recall the previous example:

$$evenM(\boxed{suc(suc(\underline{n}))}^{\uparrow}) \vee oddM(\boxed{suc(suc(\underline{n}))}^{\uparrow})$$

Here we can rewrite $evenM(\boxed{suc(suc(\underline{n}))}^{\uparrow})$ in two ways, neither of which preserves the skeleton. We can either apply rewrite-rule 5 from left to right or, as rewrites are allowed in both directions, rule 7 from right to left. The latter would give the result

$$oddM(suc(suc(suc(n)))) \vee oddM(suc(suc(n))$$

where $evenM$ has been transformed into $oddM$ by adding a successor-function rather than removing one. Consider now applying rule 5 from right to left, which produces a state that does embed the skeleton but adds yet another successor function:

$$evenM(\boxed{suc(suc(suc(suc(\underline{n}))))}^{\uparrow}) \vee oddM(\boxed{suc(suc(\underline{n}))}^{\uparrow})$$

Subsequent bad applications could keep alternating between $evenM$ and $oddM$, each time adding another successor-function and hence never terminating. Our solution to these problems uses caching of the visited states and is discussed in §4.3.

4.2 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 w 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 [10]. 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 the best-first search takes two reasoning states and compares them. A state is regarded as *less* than another state if its heuristic score is better, thus placing it closer to the front of the agenda.

The 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 concludes the proof. Skeleton preservation is always preferred over non-preservation as we only want to apply non-skeleton preserving steps when there are no other options. States that do embed the skeleton 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. For comparisons, we use the best ripple-measure of each state.

The heuristic also handles non-rippling states, such as setting up a rippling attempt or applying fertilisation. Non-rippling steps are simply preferred before more rippling as a fixed number of non-rippling steps will either result in a solution (if fertilisation is successful) or a new ripple-state to which our standard heuristic is applicable if we have to prove a lemma. Little or no search is needed.

Because best-first rippling does not become blocked as often as ordinary rippling does, we considered 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.

4.3 Termination and Reduction of Search Space Size

As mentioned before, allowing rippling with non-measure decreasing wave-rules means that best-first rippling is no longer guaranteed to terminate. The same wave-rule now can be applied in opposite directions, causing loops, and it is possible to apply rewrites that just blow up the size of the goal-term as described in §4.1. Another source of inefficiency is the many symmetric branches in the search tree.

To deal with these problems, the best-first implementation caches the visited states of a ripple sequence. We filter out any new subgoals that are identical to subgoals previously seen anywhere in the search tree, thereby pruning symmetric branches. The termination and looping problem is dealt with by introducing an *embedding check*, as used in IsaPlanner's lemma conjecturing ([10] Chapter 9). If a previous goal-term embeds into the new sub-goal it is removed, which filters rewrites that would otherwise cause divergence. Kruskal's Theorem [15], 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. We have

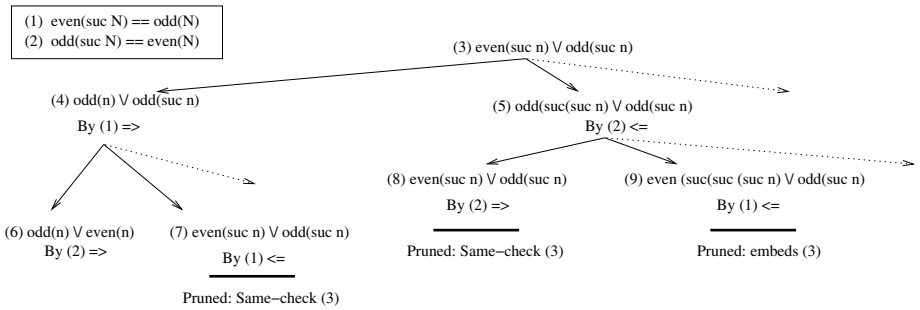

Fig. 1. Partial search tree for best-first rippling showing how branches are pruned to avoid loops and redundant rewrites. Note that the two rules are allowed to be applied in both directions.

chosen to only check embeddings against other states on the same branch, if checked against states on alternative OR-branches we could potentially prune useful states. This approach appears to work well in practice. Figure 1 illustrates how unproductive branches are pruned to reduce the size of the search space.

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. When using best-first search, it may be the case that after a lemma has been conjectured and a proof attempt begun, some state in the original proof attempt has a better heuristic score so rippling is continued from there. If this second ripple also becomes blocked and requires a lemma which we already have started a proof of elsewhere, 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. Initially, the problem was 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. Experiments also suggested that this approach may miss solutions to some problems and generally lead to longer run-times. 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 marked as delayed. Similarly, the strategy checks if the current state wishes to resume some delayed states, which are then returned to the agenda. IsaPlanner's lemma conjecturing machinery was augmented with a cache for lemmas-in-progress in addition to the existing caching of completed proof attempts. The lemma conjecturing critic inspects the cache and if an attempt is already in progress, the reasoning state is marked as delayed and not evaluated further until the proof attempt of the relevant lemma is finished.

4.5 Storing Skeletons

Ordinary rippling will discard any skeletons that cannot be embedded in the current goal term as they are not needed any more. Best-first rippling on the other hand, needs to keep all skeletons. After applying a non-skeleton preserving step, the previous skeleton must be kept so we can keep track of whether or not the skeleton is restored in subsequent steps.

The skeletons and their possible embeddings are stored in IsaPlanner's contextual information for rippling. Previously, only the list of possible embeddings of a skeleton was stored. When a skeleton failed to embed, all references to the skeleton were removed, making it impossible to later check if the skeleton could embed into some new state. For best-first rippling, the contextual information for rippling has been modified to store a list of pairs consisting of both the skeleton and a list of embeddings of that skeleton (as opposed to only the embeddings list). A skeleton not embedded in the current subgoal will have an empty list of embeddings, but will still be kept.

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, consisting of 55 theorems in Peano arithmetic and about lists, to test the performance of best-first rippling compared to ordinary rippling on standard problems. Best-first rippling has a larger search space and 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,

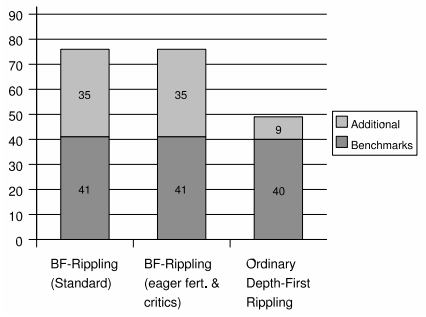

Fig. 2. Number of successes on the 94 theorems in the test set (55 benchmarks and 39 additional)

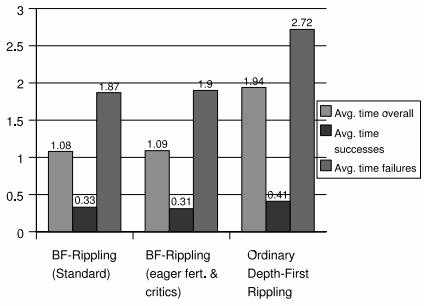

Fig. 3. Average run-times in seconds

including proofs about mutually recursive functions, proofs involving destructor-style functions (such as the predecessor function in 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 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.

The number of successful proofs for the three versions of rippling are displayed in figure 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. Ordinary depth-first rippling succeeded to find 49 proofs, 40 from IsaPlanner's benchmarks compared to 41 for best-first rippling. The additional benchmark problem solved by best-first rippling was $rev(l) = qrev(l, [\,])$, a problem expected to fail as IsaPlanner lacks a generalisation critic for accumulator variables, but here solved as a side effect of our caching mechanism[3]. 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.

[3] The interested reader can find the proof on the project website: `http://dream.inf.ed.ac.uk/projects/bfrippling`.

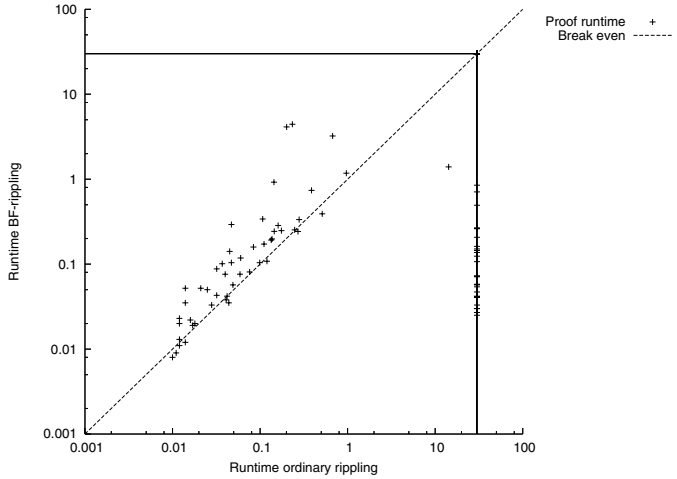

Fig. 4. Each scatter-plot represents a conjecture, with the x-value being the runtime for ordinary rippling and the y-value the runtime of standard best-first rippling. The vertical and horizontal lines marks the timeout limit of 30 seconds. Failed proof-attempts have also been plotted along these lines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

Figure 3 shows the average run-times for proof-attempts while figure 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 result, best-first rippling is faster than depth-first rippling overall, and also spends less time on conjectures it cannot prove thanks to the caching and embedding-check.

The differences in runtime appears to be small between the two variants of best-first rippling. 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. We also notice that best-first rippling spent less time on failed proof attempts, including the non-theorems in the test set, despite the larger number of allowed rewrites.

The full collection of test problems, results and function definitions can be found on-line at `http://dream.inf.ed.ac.uk/projects/bfrippling`. The source code is available from the IsaPlanner website: `http://sourceforge.net/projects/isaplanner/`.

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.

6 Related Work

6.1 Depth-First Rippling

The main difference between our work and IsaPlanner's previous implementations of ordinary depth-first rippling [10], is that best-first rippling relaxes the requirements that each state must preserve the skeleton and decrease the ripple measure. These requirements guarantees the termination of ordinary rippling, something that is lost for best-first rippling. Our implementation instead uses mechanisms for caching of visited states to avoid loops and a check on term embeddings to restore termination (see Kruskal's Theorem [15]). This works well in practise and has the additional advantage of pruning the search space of symmetric branches.

6.2 Best-First Rippling in λClam

James Brotherston implemented a best-first methodical in the λClam proof planner [2]. The best-first methodical use a greedy search strategy, considering only the best option at the current node, not previous nodes higher up in the tree. Higher branches in the search tree are only investigated on backtracking. Applied to rippling[4], Brotherston identifies this as a problem as it does not allow switching focus to the most promising area of the search. Our best-first search strategy is not greedy and we can easily switch focus to different parts of the search tree as IsaPlanner's reasoning states, held in the agenda, contains the necessary local contextual information about the proof-plan and next reasoning technique.

6.3 Best-First Proof-Planning

Manning et al. presents an implementation of best-first proof-planning in *Clam* [17]. A best-first heuristic is employed to make choices between three different proof planning methods; generalisation, simplification and induction, as a fixed ordering sometimes causes unnecessarily complicated proofs or even causes failure. Our work differs from that of Manning as we are applying best-first search *within* the rippling technique. IsaPlanner applies induction and rippling first, then attempts simplification or generalisation if the ripple becomes blocked. Despite this, all proofs in [17] are solvable by best-first rippling, although perhaps not in the most efficient way.

7 Further Work

As a side-effect of the caching mechanism, best-first rippling manages to prove the conjecture $rev(l) = qrev(l, [\,])$ where we would expect rippling to fail without a generalisation critic that can introduce an accumulator variable before induction and rippling is attempted. Best-first rippling does however fail to prove more

[4] Personal communication: internal Blue Book Note series, numbers 1405, 1409, 1425.

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 if an induction scheme other than 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 [14]. IsaPlanner has currently only one critic, for lemma calculation. We plan to implement additional critics in IsaPlanner. This is expected to allow a larger number of problems to be solved automatically, including many of the problems from the test set where both best-first and ordinary depth-first rippling currently fail.

The 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. In allowing these steps we do however lose the guarantee of termination for rippling. Our solution to this problem introduces an embedding check (§4.3), where new subgoals in which we can embed previously seen cached 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 §4.1 and restores termination. We also found that the search space often would contain symmetries, where the same state occurs in several different places. To improve efficiency, any goal identical to a cached goal is simply pruned.

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 runtimes 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. On our test set, we did however find that applying critics eagerly when no more skeleton preserving states were available, 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. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *10th International Conference on Automated Deduction*, number 449 in LNAI, pages 647–648, 1990.
8. 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.
9. 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.
10. L. Dixon. *A proof-planning framework for Isabelle*. PhD thesis, School of Informatics, University of Edinburgh, 2005.
11. L. Dixon and J. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, pages 279–283, 2003.
12. L. Dixon and J. Fleuriot. Higher-order rippling in IsaPlanner. In *Proceedings of TPHOLs'04*, pages 83–98, 2004.
13. D. Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997.
14. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.
15. J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 1960.
16. D. Lacey, J. Richardson, and A. Smaill. Logic program synthesis in a higher-order setting. *Computational Logic*, 1861:87–100, 2000.
17. 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.
18. 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.

19. 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.
20. 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.
21. 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.
22. 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.