Rewrite heuristics and pair exploration
for automated theorem proving

Keeley Hoek (u5642917)

December 20, 2018

In this report we introduce an algorithm for proving equational lemmas by rewriting them within the ambient environment of an automated theorem prover. Our explanation is modelled on the dependent type theory language provided by *Lean 3*. The idea is to focus attention on reducing *interesting pairs* of left- and right-hand side expressions which we anticipate will be increasingly easy to prove are equal. We describe the algorithm in detail, before suggesting some basic extensions.

Beginning with an edit-distance metric defined on mathematical expressions and initially provided to the algorithm, we explain an augmentation using weights generated from a machine-learning SVM (support vector machine) classifier, continuously updating as the search for a proof progresses. Finally, a strategy for guiding the search towards more elegant proofs using an A*-like heuristic is introduced.

We then turn to demonstrating the function of the algorithm across several problem domains, selected in order to emphasise the impacts of the various extensions to the core algorithm which we proposed earlier. Namely, we solve problems in category theory, arithmetic, and a slightly more contrived theoretical example involving hypercube complete graphs. This is all done by invoking an full implementation of the procedures we described developed completely within Lean's metaprogramming language. We conclude by highlighting future areas for potential improvement.

1 Background

We will focus our attention solely on proving equalities of mathematical expressions, which we will often call *equational lemmas*. In principle, we allow arbitrary mathematical expressions (perhaps with some bound variables), and hence must resort to general methods and not restrict ourselves to any particular setting. In contrast, much work as been done on proving equations in specific areas such as for commutative rings (e.g. that of [3] for Coq) or first order logic (e.g that of [1]), and where a proof of the algorithm's correctness is possible.

We take a different approach—trying to emulate human strategies using more flexible machine-learning methods—inspecting the underlying expressions constituting an input equational lemma and rewriting it using known rules. This is done in the hope that these strategies combined with brute-force computing power can yield a proof—without relying on that possibility formally being guaranteed. We package these strategies into a tactic for a generic interactive/automated theorem prover, modelled on the formal language of Lean.

In the language of dependent type theory of Lean[1], every expression can be represented as some finite number of recursive applications of constructors of an inductive type `expr`. This means that for our purposes we can consider mathematical statements in Lean as essentially trees of repeated function applications, to be thought of themselves as repeated applications of a "constructor" function

$$\texttt{expr.app} : \texttt{expr} \to \texttt{expr} \to \texttt{expr}.$$

The leaf nodes of this tree of applications are then just some other kinds of expressions, perhaps modelled as applications of a function `expr.const` : `string` → `expr`. We have overlooked several of Lean's "embellishing" additional `expr.*` constructors here, most notably of the kind enabling the terms of an expression to be bound to free variables (λ-binders). Fortunately, within a tactic Lean is able to "unwrap" binders and replace references to bound variables by so-called "local constants" in the local state available during tactic execution (which can—and in our case will—just be thought of as another variant of `expr.const` above).

Thus, the equalities we will consider will amount to Lean expressions which have as their root vertex the application of the constructor `eq` : `expr` → `expr` → `expr`, accepting some left- and right-hand side expressions. As a shorthand for `eq`(e, f) we will write $e = f$. Our goal will then be to construct a term of the type represented by such an expression.

Notably, our constructions `expr.const` : `string` → `expr` have forgotten the type information of the constant which they represent. Type-soundness is enforced by Lean when we ask that a tree of expressions be *evaluated* (typically at the end of our rewriting procedure), so will generally not play directly into our considerations. It is at evaluation-time that our tactic's constructions are formally verified, and this permits us to develop tactic programs in a setting without having to meet any additional formal proof obligations.

We emphasise that the algorithm we describe below is more powerful than just an equation simplifier. We accept arbitrary lists of "known lemmas" and rewrite expressions potentially vastly increasing in complexity before once again decreasing. In particular, this system of known lemmas need not be confluent or satisfy a similar property.

2 The problem and preliminary notions

In order to prove equalities by rewriting their subexpressions, we necessarily must maintain a list of mathematical facts which may be used. For this purpose we will maintain a list L of `expr`s representing known proofs of equalities of the form $e = f$, for `expr`s e and f, perhaps each containing some number of variables with bound type (for example $n = n + 0$ with n any natural number). We will call the elements of the list L each *rewrite lemmas*, and throughout will fix such a list. In practice, the known rewrite lemmas in an interactive Lean session can be programmatically enumerated and inspected. We are actively exploring algorithms which can dynamically identify relevant rewrite lemmas (from an ambient library) given an equality $e = f$ to be proved.[2]

In this language our problem can now be stated precisely as follows.

[1] Note, though, that we restrict ourselves to discussing only those extremely general expected properties of `expr`s. A direct analogy with other theorem proving systems should be expected.

[2] This is an active area of research, and can go by the name *lemma discovery/synthesis* [4].

Problem 2.1. Let $e = f$ be an equality of `expr`s to be proved, and L a list of rewrite lemmas. Find a proof of $e = f$ (just a term of this type) by rewriting subexpressions of e and f until a tautology $x = x$ is obtained.

Building on Lean's expression manipulation facilities, we essentially define a list-of-`expr` valued function $\mathrm{rws}(e, L)$ accepting an `expr` e and a list of rewrite lemmas L, and returning a list of all possible expressions e' obtained from e by rewriting a subexpression of e using a lemma in L. We call any such expression e' a *rewrite* of e. Of course, in order to persuade the formal checker that such a replacement is justified, we should also return with each `expr` e' a proof r that $e = e'$. To clarify the description of our algorithm below we will suppress this information, and will essentially just explain how to determine that a proof of a given equality obtained by a sequence of rewrites actually exists. The way in which one should backtrack from this point, assembling the necessary proof facts which justify the performed manipulations, is entirely dependent on the formal prover in which the algorithm is being implemented (and in any case should be trivial and linear in the number of rewrites needed). It will be sufficient for our purposes to consider each e' returned by $\mathrm{rws}(e, L)$ to be "tagged" with a proof $e = e'$, and we will therefore not say anything further on this point.

Throughout we will fix a distance heuristic on `expr`s, i.e. a function

$$m : \texttt{expr} \times \texttt{expr} \to \mathbf{R}_{\geq 0}$$

with $\mathbf{R}_{\geq 0}$ the type of nonnegative real numbers, which estimates their rewrite distance and which we will call the *metric* (but will not demand that it satisfy any formal properties). In practice the set $\mathbf{R}_{\geq 0}$ will actually be a finite-precision approximation of the nonnegative real numbers. A useful working model for the metric $m(e, e')$, which we will see is (perhaps surprisingly) sufficient for many purposes, is the *token edit-distance* between e and e' (with each token having cost 1) calculated when each is *pretty-printed* by Lean into strings and tokenized at spaces. The Lean pretty-printer is intended to heuristically transform expressions into a "human-readable[3]" representation, omitting more technical features such as functions' various universe parameters or implicit type(class) arguments.

In order that it is practical to execute the core loop of the algorithm in an interactive Lean session, it is necessary that we build-in some optimisations (as opposed to providing them as extensions later). For this reason we will need to introduce the notion of a *distance estimate*, modelled in Lean as some type `dist_est`. It will have two constructors "n" and "$\geq n$" (allowing $n \in \mathbf{R}_{\geq 0}$) representing knowledge that a distance (as measured by the metric m) between `expr`s is calculated to be precisely n (called an *exact* estimate), or bounded below by n (called a *partial* estimate), respectively. We also encode within partial distance estimates the information necessary to, as needed, strictly improve their lower bound (until they are rendered exact).

It will be important for us to compare two distance estimates, determining an element of the pair they form which, if both estimates were improved until they were exact, is represents a minimal distance. Formally given two distance estimates d and d', *comparing* them consists of strictly improving the bound represented by each in succession until one becomes exact and the other is bounded below by the former's

[3] For example, one desires that the natural number 4 be printed as the string containing the single character '4', and not four recursive applications of the natural number successor function to the constructor of the natural number zero.

exact value. If $d = n$ is exact and d' asserts boundedness from below by at least n, then we will write $d \leq d'$.

3 The core algorithm

In practical problems, it is extremely important to guide a search of the graph of all possible rewrites. For instance, in contrast with the examples given in later sections, in all but the most trivial cases a breadth-first-search of the rewrite graph has extremely poor performance. Calculating all of the possible rewrites of complex expressions can be very time-expensive, and can lead to highly inefficient methods of graph traversal. Also of note is the fact that the search graph can be infinite even in trivial arithmetic problems, for example whenever lemmas which introduce additional terms to an expression (e.g. $n = n + 0 = n + 0 + 0 = \cdots$) are considered.

Alleviating these disadvantages is especially important in an interactive session where responsiveness is critical. In this section we describe an algorithm for partially solving this problem. In later sections we entertain various optimisations and extensions to the procedure we present.

3.1 Description

Recall that we begin with a metric m : `expr` × `expr` → $\mathbf{R}_{\geq 0}$, a list L of rewrite lemmas, and a pair of expressions e and f for which we wish to prove an equality $e = f$.

The key idea of our solution to this problem is to proceed iteratively, maintaining a list P of currently *interesting pairs* each of two `expr`s, along with a `dist_est` bounding the distance (as measured by the metric m) between the elements of each pair. Every such pair[4] $p = (e_p, f_p, d_p)$ added to P will be guaranteed by construction to have had e_p obtained from e by performing some number of subexpression rewrites, and similarly f_p obtained from f by performing some number of rewrites. The purpose of maintaining such pairs is to systematically reduce the problem of showing $e = f$ to another problem $e_p = f_p$; by virtue of each coming from a pair in P, we ensure that solving the latter problem is equivalent.

We initialise P to contain only the "pair" $(e, f, \geq 0)$, indicating our desire to prove $e = f$. At each iteration step we select a pair $p = (e_p, f_p, d_p)$ of P which is guaranteed (by exhaustively examining the elements of P) to have a minimal distance estimate over all of P—we will call such such a p a *minimal pair*. We then check whether e_p has been *visited*; we maintain a(n initially empty) list V of the `expr`s which we say have been visited once already.

We then *visit* the expression e_p; we first calculate $R := \text{rws}(e_p, L)$. Then for each element e'_p of R we append the pair $(e'_p, f_p, \geq 0)$ to P if it has never been added to P before (the "≥ 0" represents a lack of current knowledge of the distance between e'_p and f_p).

There are then two possibilities; either e_p has already been visited or it has not. First consider the case where it has not. Then we add the original expression e_p to V (marking it as visited), and in addition record that e_p was obtained from a sequence of rewrites of e (as opposed to f).

The other possibility is that e_p has already been visited. When that occurred we recorded whether e_p had been obtained from rewriting e or f, and by virtue of the fact that e_p now appears as the first entry of the pair p, it has been obtained by

[4] This is a triple and not a pair of values, but nonetheless represents a pair of expressions e and f.

a sequence of rewrites of the expression e. We now check whether when we first visited e_p, it was also obtained by rewriting e. If this is the case then we immediately discard it and do nothing.

Otherwise, e_p has already been obtained by rewriting f as well, which necessarily implies that by some sequence of rewrites of each of e and f the claimed equality $e = f$ can be shown to be equivalent to $e_p = e_p$. Therefore $e = f$ can be proved in precisely this manner; we immediately abort the procedure and backtrack, reporting the sequence of rewrites which were used to obtain e_p in each case.

In any case, this completes the "visit" of e_p.

We now visit f_p, performing the completely analogous procedure to that which we have just described for the case of e_p—we essentially just interchange the labels e and f and e_p and f_p everywhere. The one necessary distinction is that when we calculate $R = \text{rws}(f_p, L)$, we append all possible pairs $(e_p, f_p', \geq 0)$ with f_p' in R (which have not already been added to P), to P—we replace the second entry of p, and not the first.

Finally, erase p from the list P, and repeat this process beginning with a new pair p' in P which has distance estimate $d_{p'}$ minimal in P. We give a summary of this process below.

Algorithm 3.1 ("Pair explore", core loop)**.**

Let metric m, the list L of rewrite lemmas, and `expr`s e and f be given.

Create a list P containing the single entry $(e, f, \geq 0)$. Create an empty list V. Then, repeat the following procedure forever.

(1). If the list P is empty then terminate, reporting failure (all interesting pairs have been exhausted, without success).

(2). Otherwise, let $p = (e_p, f_p, d_p)$ be the head element of P. For each successive element $p' = (e_{p'}, f_{p'}, d_{p'})$ of the tail of P, compare d_p and $d_{p'}$ improving each respective estimate if necessary. If $d_{p'} \leq d_p$ then replace p with p', continuing until all of the elements of P have been exhausted.

The result is that $p = (e_p, f_p, d_p)$ is an element of P with minimal d_p.

(3). Visit e_p. Visit f_p.

(4). If while visiting e_p or f_p we discovered a way to rewrite $e = f$ into a tautology then terminate, reporting the sequence of rewrites we identified which achieve this.

(5). Otherwise, erase p from P and return to (1).

The point is that the list of interesting pairs P maintains a list of our "hunches", where we suspect we may have success rewriting in the future. We always rewrite the pair with minimal metric distance, in the hope that the metric is guiding us towards pairs which are easier and easier to prove are equal. Nonetheless if a particular pair $p = (e_p, f_p, d_p)$ happens to be a "local minimum" of e_p-to-f_p distance, for example in the simplest case of having no further rewrites of each of e_p and f_p possible, this is overcome during execution since p is essentially just dropped and the process is again repeated. The stack of rewrites we have performed essentially just unwinds.

In the simplest case of there existing a sequence of rewrites of e and f toward a tautology with monotonely decreasing metric distance, and where no "false-positive" pairs—with smaller distance but larger number of total rewrites required to give a proof—can be added to P, pair-exploration immediately leads to a proof of equality without inspecting expressions obtained from any unnecessary rewrites.

3.2 Basic extensions

The algorithm we have described above admits several simple optimisations which are seen to have significant effects on performance in practice. By design, the value of the function $\mathrm{rws}(x, L)$ for the same expression x may be required in multiple iterations of the core loop—it can therefore be cached, to the benefit of situations where computing all rewrites is very expensive (for example, when the list L is very large). Also, as we mentioned above when describing comparing distance estimates, storing references to a table of distance estimates which can be updated from iteration to iteration can greatly improve responsiveness in an interactive session. This is a consequence of the fact that computing edit-distances can also be quite expensive for long sequences of tokens, and so doing this in pieces and only where required can avoid unnecessary computations where we very quickly establish that the edit-distance we are calculating is "large enough to be ignored for now".

It is also natural to consider the introduction of some tuning parameters, motivated by some characteristic behaviours of the algorithm. For example, observe that during an iteration in which we select a pair $p = (e_p, f_p, d_p)$, we compute (or recall) the entire lists $\mathrm{rws}(e_p, L)$ and $\mathrm{rws}(f_p, L)$ and then potentially add a new pair to P for each element of both of these lists (depending on whether each new pair has already been seen at some point). It can be the case that for many pairs p the expressions e_p and f_p have associated sets of rewrites which are very large—a simple example is the case of a long but trivial arithmetic expression, where multiplication by 1 and addition by 0, and associativity and commutativity can be applied in many places. It is convenient to package the information of the pairs obtained from the rewrites of both $\mathrm{rws}(e_p, L)$ and $\mathrm{rws}(f_p, L)$ into a single list[5] L_p associated to p.

In the operation of the algorithm as described the contents of L_p are immediately appended to the list P, deleting pairs which have already been added to P at some point. Then p is deleted from P. It can be desirable to instead introduce a parameter N_p called the *pop-size* and equal to some positive natural number or ∞. Then when a minimal pair p is selected from P and L_p is calculated, we instead append only the first N_p elements of L_p to P and remove them from L_p (or fewer if L_p contains fewer than N_p elements). Then, instead of deleting p from P we re-add it, being careful to record of the remaining elements of L_p. If P is then ever again a minimal pair, we examine the next N_p elements of L_p and so on until every element of L_p has been exhausted, at which point p will actually be erased from P. The advantage of doing this is that often not all of the possible rewrites available need be considered in order to make progress, and the disadvantage is that very small N_p values can "blind" the search algorithm to potentially very important rewrites for some time (we consider the order of a list $\mathrm{rws}(x, L)$ essentially random).

[5] There is also a subtle question which arises here as to how this pair of lists should be combined—by concatenation, by 1-1 splicing, or something else—which can lead to differing search characteristics where rewrites for one expression is pursued first before the other, or rewrites for both are pursued in alternating fashion, etc. For the purposes of later sections, in implementing the algorithm ourselves we performed 1-1 splicing.

4 Integrating machine-learning

While executing an instance of pair-exploration typically many subexpression rewrites are calculated, but the expression that each rewrite yields is often never considered in the form of being given a "visit". We have described using the pop-size parameter N_p to manage the effect this has on the number of new pairs added to P each iteration, but we did not actually change the number of calculated rewrites. In this section we explore how this additional information regarding known, "reachable" `expr`s, can be leveraged to improve the token edit-distance metric on expressions.

4.1 Weighted edit-distance metrics

When a human attempts to prove an equational lemma, for instance a trigonometric identity such as

$$\sin^4 x - \cos^4 x = 1 - 2\cos^2 x,$$

they employ a number of strategies. In this example, suppose we start with the left-hand side $\sin^4 x - \cos^4 x$. One such human-like strategy could proceed as follows:

1. Observe that if we are ever to perform manipulations of the left-hand side in order to eventually obtain the right-hand side, we will need to eliminate the $\sin^4 x$ term since it does not appear in the right-hand side at all.
2. Search our "working knowledge of mathematics", consisting of our known rewrite lemmas, for ones which can rewrite $\sin^4 x$ into an expression not containing "$\sin x$".
3. Note that $\sin^4 x = (\sin^2 x)^2$ and that the rewrite lemma $\sin^2 x = 1 - \cos^2 x$ successfully rewrites it, removing the instance of $\sin x$.
4. Rewrite the left-hand side, obtaining a proof that $\sin^4 x - \cos^4 x = (1 - \cos^2 x)^2 - \cos^4 x$.
5. Simplify[6] the result, obtaining $\sin^4 x - \cos^4 x = 1 - 2\cos^2 x$.
6. Realise that this is precisely the equality to be proved. (If we were not yet finished, then repeat.)

Of course, this strategy is far from flawless; it obviously does not succeed in general, and for example offers no help when both the left-hand and right-hand sides of a given equality are simple rearrangements of the same tokens. Nonetheless, recognising which tokens are important to eliminate during a manipulation of an equality can provide important information to a proof search, making it much more efficient than a brute-force approach.

[6] We intentionally leave the notion of *simplification* of an expression imprecise here. For our purposes, simplification should roughly correspond to the "mindless" expression manipulations humans might perform before reanalysing the result of their calculation—expanding squared parenthesised subterms and then cancelling terms in the result, for example. One model of this procedure, which is essentially provided by Lean, is to keep track of an ambient list S of directed rewrite lemmas (rewrite lemmas which should only applied when a particular side of the equality they represent is matched), and when asked to simplify an expression just mindlessly apply these lemmas until it is not possible any further. In our case the set S could contain $(a+b)^2 = a^2 + 2ab + b^2$ (among many other lemmas), for example.

One way attempt to exploit these observations is to generalise the idea of the tokenized edit-distance metric on `expr`s to weight the insertion/deletion/substitution of tokens depending on how important we suspect their manipulation will be to the proof.

Thus let T be the set of all tokens which can be obtained from tokenizing `expr`s and let $\ell = (\ell_i)$ be a finite sequence in T. Suppose that a weight function $w : T \to \mathbf{R}_{\geq 0}$ is given. We allow the elements (ℓ_i) to be manipulated by some sequence of operations each with an associated $\mathbf{R}_{\geq 0}$-valued cost, with each operation being

- a deletion of a token $t = \ell_i$ from ℓ for a cost of $w(t)$, or
- an insertion of a token t into ℓ for a cost of $w(t)$, or
- the substitution of a token $t = \ell_i$ in ℓ with a token t' for a cost of $\max\{w(t), w(t')\}$.

The cost of a sequence of operations is just the sum of each operation in the sequence.

The w-weighted edit-distance between two finite sequences of tokens ℓ and ℓ' is then the minimum of the costs of all sequences of operations taking ℓ to ℓ' (the Wagner–Fischer algorithm [6] implementing this calculation is easily adapted cache partial progress in a `dist_est` structure). By converting `expr`s into strings of tokens, for example by pretty-printing them and tokenizing at spaces, such a weight function w defines a metric m_w : `expr` × `expr` $\to \mathbf{R}_{\geq 0}$ (and the case of the $w(t) = 1$ weighting recovers our original edit-distance metric).

4.2 Token classification

By computing appropriate weight functions, we can steer a pair-exploration search towards the most important rewriting steps. Ideally, we obtain the benefits of the human-like strategy seen in the example above without the disadvantages of directly applying this strategy and potentially getting stuck after making a wrong-turn rewrite.

However, computing a useful weight function is an entire sub-problem in and of itself. We propose a strategy based on the number of occurrences of tokens in expressions obtained from rewriting the original expressions e and f. Let $\omega_x(t)$ be the number of times which the token t has been encountered in tokenizing the `expr` x (accounting for a multiplicity of occurrences in the same expression).

Note that it is enough during execution of our algorithm to partially define the metric m_w on only pairs of those expressions which we have already obtained by rewriting. Thus, we only require a definition of $w : T \to \mathbf{R}_{\geq 0}$ replacing T with the set of tokens which have been obtained by pretty-printing `expr`s that we have seen so-far (and we redefine T thusly).

Label the elements of the (necessarily finite) set T by $t_1, t_2, \ldots, t_n$. For each expression x we can thus form the vector $\omega_x = (\omega_x(t_1), \ldots, \omega_x(t_n))$. We can then form the set C_e of all those ω_xs for which x is an expression which has been obtained during a pair-exploration iteration from rewriting f. Define C_f for f similarly. Then execute a linear SVM (support vector machine) algorithm[7] in order to calculate a hyperplane

$$\mathbf{a} \cdot \mathbf{v} - b = 0$$

[7] Such as that provided by *libSVM* [2], which we used and integrated into Lean via native foreign library bindings.

in $\mathbf{R}^n$ classifying the sets C_e and C_f, with $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbf{R}^n$, $b \in \mathbf{R}$, and $\mathbf{v} \in \mathbf{R}^n$ free. The point is that this hyperplane divides into $\mathbf{R}^n$ two regions, classifying (by its vector ω_x) whether an expression x is more likely to have been obtained by rewriting e, or f.

The component a_i of $\mathbf{a}$ gives the slope of this classifying plane in the ith coordinate, and therefore quantifies the extent to which possessing the token $t_i \in T$ determines whether an expression x obtained during pair-exploration was likely to have been obtained from rewriting the original expression e, or f. Tokens t_i with a large component $|a_i|$ are thus precisely those which we desire to rewrite-away or rewrite-and-add in order to transform e into f (or more likely reconcile these two expressions as equal to a common third expression).

A token t_i appearing equally frequently in expressions obtained from e and f will necessarily have $a_i = 0$. However, for the purposes of obtaining a proof of an equality, a misplaced token t_i is still undesirable. Therefore, it is undesirable to permit the token weights given by w to actually be equal to zero—we would like them to retain some positive edit-distance cost. We therefore suggest the SVM weight function

$$w(t_i) = 1 + \log(1 + |a_i|).$$

We will call the induced metric m_w the *SVM-weighted (edit-distance) metric.*

Another possible weight function w can be obtained by computing the sums Ω_e and Ω_f of the sets of vectors ω_x obtained from rewriting e and f respectively. The ith component of the difference

$$\mathrm{abs}(\Omega_e - \Omega_f) = (|(\Omega_e)_1 - (\Omega_f)_n|, \ldots, |(\Omega_e)_n - (\Omega_f)_n|) =: (c_1, \ldots, c_n)$$

then gives a different measure of how much the token t_i accounts for expressions obtained from e, or f. We analogously suggest the weight function

$$w(t_i) = 1 + \log(1 + |c_i|)$$

in this case, giving a metric m_w which we will call the (unnormalised) *centre-of-mass-weighted (edit-distance) metric.*

Finally we note that it is not necessary, and can be quite expensive in some situations, to recompute the weight functions w after each iteration of pair-exploration in each case. Note that while recalculating the values of a weight function is typically not very expensive, there is the much more significant associated cost of having to reset the in-progress metric calculations stored in all of the `dist_est`s because all of the distance estimates they give may no longer be correct for the new metric. As a consequence, it is desirable to perform a weight recalculation after some fixed number N_r of pair-exploration iterations, as opposed to every iteration. In the meantime if we see a token we have not seen before, we simply assign it some default weight which we will take to be unity.

In fact, this strategy "deferred refreshing" can have additional function benefits over a setting of $N_\mathrm{r} = 1$. For example, consider a setting of $N_\mathrm{p} < \infty$ and $N_\mathrm{r} = 1$. It can happen that in a single iteration step, some number of rewrites are all found rewriting an expression obtained from e and giving rise to a token t which has not been seen before, but which would have also been seen from rewriting f if the pop-size parameter N_p was not finite. Alternatively, it could happen instead that no direct rewrites of f give rise to expressions containing t, but many second-order rewrites of the resulting expressions do. In any case the pair-exploration algorithm with $N_\mathrm{r} = 1$ may erroneously pursue rewrites which insert or remove the token t without this behaviour being of material importance to the proof.

5 Further extensions

Before turning to the demonstration examples, we briefly explain some further extensions to the variants of an edit-distance metric which we have just described. First note that all of the major pieces we have described—the pair exploration algorithm, the general notion of a metric, the idea of edit-distance weight functions—all fit together as modular objects which can be plugged-into one another and swapped as needed. Our underlying implementation of all of these pieces has been structured with the aid of Lean's powerful metaprogramming language in much the same way. Extensible modules wrapping, iterating, or replacing the components we have just described are all easily developed and invoked from an interactive session.

One natural wrapper for the metrics we have described essentially implements an A*-like heuristic. The heuristic measures distances on the search graph which is constructed as rewrites are computed, and can be defined as follows. Observe that each time we computed an element e'' of $\text{rws}(e', L)$ for some expression e', we essentially determined an edge $e' \to e''$. This graph structure has been effectively suppressed thus far, but it now comes to the fore. Fix a metric $m : \texttt{expr} \times \texttt{expr} \to \mathbf{R}_{\geq 0}$ and let $p = (e_p, f_p, d_p)$ be a "pair" obtained during pair-exploration. Then using this graph structure we can compute shortest distances $\text{dist}(e, e_p)$ and $\text{dist}(f, f_p)$ using the currently known graph having rewritten expressions as vertices and edges $e' \to e''$ whenever there is a rewrite lemma taking e' to e'' in one step. It is then interesting to define the *A*-metric associated to m*

$$m_\alpha^{\text{A}^*}(e_p, f_p) = \alpha\,\text{dist}(e, e_p) + m(e_p, f_p) + \alpha\,\text{dist}(f_p, f)$$

for some constant $\alpha \in \mathbf{R}_{\geq 0}$.

This metric is interesting because the quantities $\text{dist}(e, e_p)$ and $\text{dist}(f_p, f)$ are the known shortest[8] number of rewrites required to obtain e_p from e and f_p from f, respectively. They therefore represent an estimate of the complexity of a proof of $e = f$ if a proof is found via showing $e_p = f_p$. Increasing the parameter α increases the bias away from potential proofs which appear to be high in complexity—the pair-exploration algorithm attempts to better emulate a shortest-path search. This has obvious positive consequences in the context of attempting to find the most concise, elegant, and human-readable proofs of equational lemmas.

6 Demonstration and sample problem domains

We have no intention of giving a guided tutorial on the Lean theorem prover's language, nor of giving exhaustive documentation describing the operation of the Lean meta-programs which we have developed to implement the algorithms that have just been described.

Nonetheless, we devote this section to giving some real examples of interactively proving theorems in Lean using pair-exploration and related methods—which together form a Lean tactic we will call "`rewrite_search`[9]". We do this both to give some indication of the practical problems which can be solved via the methods

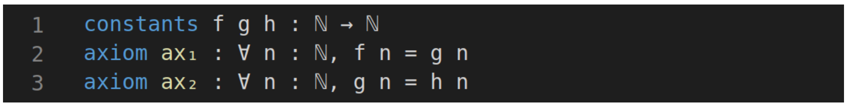
[8] Though, it need not be the case that during our search a calculated $\text{dist}(e, e_p)$ value is actually minimal over *all possible* ways of rewriting e and the resulting expressions. We merely examine our current knowledge in order to calculate $\text{dist}(e, e_p)$.

[9] The source code for this bundle of utilities can be found at `https://github.com/semorrison/lean-tidy`, along with continually improving associated documentation.

which have described, and also to illustrate the convenient way in which the metaprograms we have developed can be invoked in a Lean session.

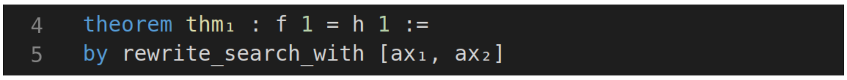
6.1 Interacting with Lean

We begin by very briefly introducing the interactive Lean session in which we will be working, by solving a trivial example problem using `rewrite_search`. We first enter the following three-line code snippet.



```
1   constants f g h : ℕ → ℕ
2   axiom ax₁ : ∀ n : ℕ, f n = g n
3   axiom ax₂ : ∀ n : ℕ, g n = h n
```

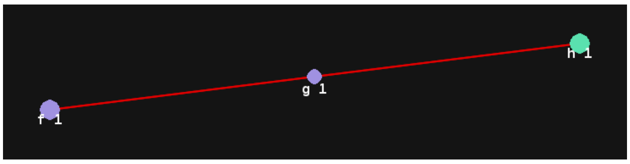
The first line (1) declares the existence of three functions f, g and h, each sending natural numbers to natural numbers. Line (2) then asserts (under the name $\texttt{ax}_1$) that it is known that $f(n) = g(n)$ for every $n \in \mathbf{N}$. Line (3) asserts the analogous fact $\texttt{ax}_2$ for the pointwise equality of g and h. We now introduce a theorem to be proved:



```
4   theorem thm₁ : f 1 = h 1 :=
5   by rewrite_search_with [ax₁, ax₂]
```

Line (4) declares that we intend to prove a theorem $\texttt{thm}_1$ which claims that $f(1) = h(1)$. On line (5) we invoke our general machinery, instructing Lean to perform pair-exploration using the rewrite lemma list $L = [\texttt{ax}_1, \texttt{ax}_2]$.

This code sample compiles and executes without error, and this indicates that `rewrite_search` successfully obtains a proof. Upon request, the interactive editor panel reports the rewrite steps which were performed (written in valid Lean language itself), and this is accompanied by the following interactive force-directed graph (of which we provide only an image capture) depicting the entire rewrite graph which was explored.



The two larger vertices represent the original left- and right-hand expressions, and the colour of intermediary vertices (in this case of which there is one) represents which of the two larger vertices they were first obtained from by rewriting. Of course, in this case the actual proof obtained is entirely trivial, requiring (as shown) a single rewrite of the left-hand side $f(1)$ by $\texttt{ax}_1$, and then again by $\texttt{ax}_2$. We now turn to some real examples.

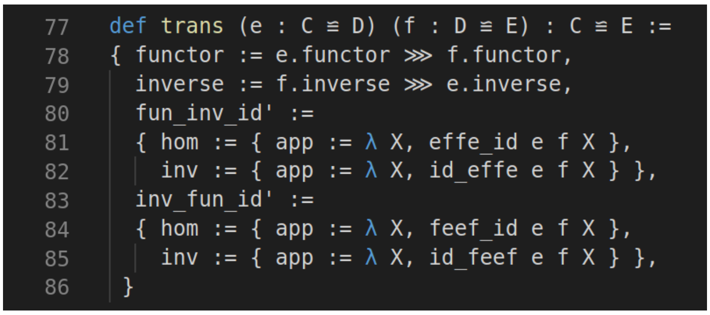
Throughout all of the following examples, a set L of rewrite lemmas from the ambient interactive Lean session have been implicitly tagged. However, we stress that these collections of lemmas were not crafted specifically for the purpose of being sufficient to solve the examples we provide, but instead are defined globally throughout the projects they have been extracted from and have been assembled for the sole purpose of developing the Lean library to which they belong efficiently and concisely.

6.2 Category theory

Invocations of `rewrite_search` have been found to be particularly convenient when developing a from-first-principles category theory library[10] in Lean. In this context, one often wants to make a construction via specifying the necessary data, where formally a list of conditions which this data must satisfy should also be verified. For example, when defining a functor between categories it might be desirable to specify only what the functor should do on morphisms, leaving well-definedness on the objects and actual functoriality as “self-evident”—as would certainly be done when communicating to other human mathematicians. Of course when higher-level constructions such as even just monoidal categories or symmetric monoidal functors are to be specified, the list of necessary conditions to be formally checked can grow very rapidly.

Lean’s metaprogramming features allow in many situations these extraneous additional proof obligations to be omitted in their entirety, with a request to generate them silently being made a (user-specified) tactic such as `rewrite_search`. In this setting, `rewrite_search` has proved very convenient in practise.

A typical example is that of proving that the composition of two equivalences of categories is an equivalence—an equivalence $\mathscr{C} \cong \mathscr{D}$ being the data of a functor $F : \mathscr{C} \to \mathscr{D}$, a functor $G : \mathscr{D} \to \mathscr{C}$, and natural isomorphisms $\alpha : FG \to \mathrm{id}$ and $\beta : GF \to \mathrm{id}$. The Lean code required to define such a composition and generate a formal proof that all of the necessary conditions are satisfied (that each natural transformation α and β is an isomorphism and actually natural, for example), is specified fully[11] in the following code snippet.



```
def trans (e : C ≅ D) (f : D ≅ E) : C ≅ E :=
{ functor := e.functor ⋙ f.functor,
  inverse := f.inverse ⋙ e.inverse,
  fun_inv_id' :=
  { hom := { app := λ X, effe_id e f X },
    inv := { app := λ X, id_effe e f X } },
  inv_fun_id' :=
  { hom := { app := λ X, feef_id e f X },
    inv := { app := λ X, id_feef e f X } },
 }
```

Note here that no explicit call to `rewrite_search` has been made, since the generic mechanism for determining omitted proofs has been invoked.

This is obviously much more concise than explicitly specifying the eight separate proofs of sub-facts which were necessary in order for the Lean-kernel to accept this definition—in fact, with “trace-mode” activated and the default parameter values $N_\mathrm{p} = \infty$ and $N_\mathrm{r} = 10$, the following data is reported:

[10] This library is available on-line at `https://github.com/semorrison/lean-category-theory`.

[11] Admittedly, we have used a shorthand notation defined in the ambient category theory library to specify the compositions which give the components of the needed natural isomorphisms.

Proof number	`expr`s seen	`expr`s visited	Rewrites used
1	22	10	7
2	91	29	16
3	46	18	10
4	14	6	4
5	22	10	7
6	91	29	16
7	46	18	10
8	14	6	4

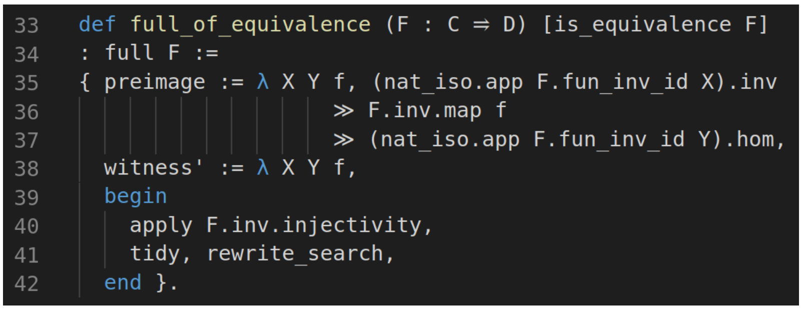
The "rewrites used" column gives the number of rewrites which were actually used in the final proof. Thus explicitly specifying each of these rewrite lemmas and their order of use would have very significantly—and essentially pointlessly—obfuscated an otherwise routine mathematical definition (especially considering the fact that proofs 1-to-4 and 5-to-8 were completely analogous symmetrical versions of each other).

The different performance impacts of centre-of-mass, SVM, and the unweighted edit-distance metrics are very marginal in this example, and the following one. Distinguishing between the centre-of-mass and SVM weightings on either side of a slash "/", we have the following similar results to those given above:

Proof number	`expr`s seen	`expr`s visited	Rewrites used
1	21/22	9/10	7/7
2	86/93	28/30	17/17
3	41/43	16/16	10/10
4	14/14	6/6	4/4
5	21/22	9/10	7/7
6	86/93	28/30	17/17
7	41/43	16/16	10/10
8	14/14	6/6	4/4

This can happen for example when a complex composition of morphisms is to be shown equal to the identity, wherein very little can be gleaned by comparing other derived complicated compositions of morphisms with the single identity morphism. The distinction between the weighting strategies is much more prominent in other areas of the category theory library, and (as we shall see) in other problem domains.

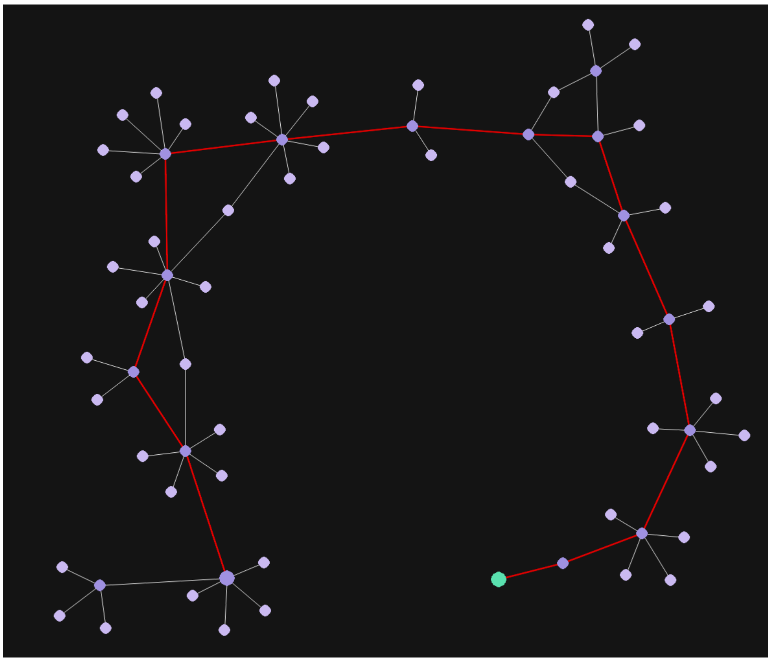
We conclude by examining a proof in Lean that a functor $F : \mathscr{C} \to \mathscr{D}$ which is part of an equivalence is necessarily full. This result is encoded in the following code snippet (which is excerpted from a file which includes some other facts about equivalences).



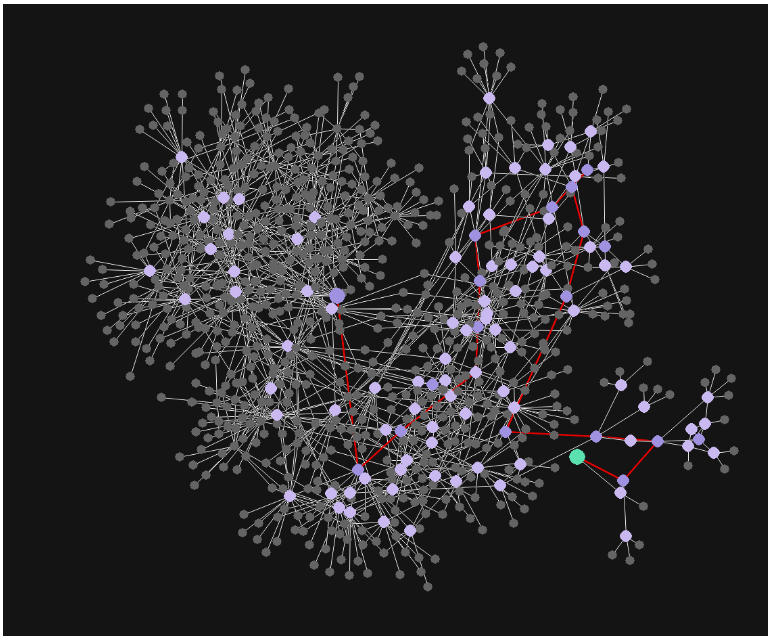
```
33  def full_of_equivalence (F : C ⇒ D) [is_equivalence F]
34  : full F :=
35  { preimage := λ X Y f, (nat_iso.app F.fun_inv_id X).inv
36                      ≫ F.inv.map f
37                      ≫ (nat_iso.app F.fun_inv_id Y).hom,
38    witness' := λ X Y f,
39    begin
40      apply F.inv.injectivity,
41      tidy, rewrite_search,
42    end }.
```

Notably, on line (40) we have inserted a call to `apply, ...` in order to hint that faithfulness of one of the functors involved in the equivalence will be critical in the proof (greatly reducing the duration of the search). On the following line we proceed an invocation of `rewrite_search` by a meta-tactic `tidy`, which by unfolding definitions just simplifies our goal into a single equality to be solved. We have configured `rewrite_search` in this case to use an SVM classifier metric with $N_\mathrm{p} = 3$ and $N_\mathrm{r} = 10$.

Below we give a visualisation of the search graph built throughout execution of the pair-exploration algorithm, where it is easily seen that the route taken by `rewrite_search` is very efficient (the actual rewrites which were used are given in red—and efficient at least in the sense that very few "dead-end" rewrites have been considered).



It is very important to stress in this instance that the true rewrite graph is infinite, and the true degree of many vertices is approximately 10 (a reduced pop-size parameter limited this number during the actual search). Below we give a continuation of the previous graph, including additional grey vertices which were not even found during the search (but nonetheless were identified by chance after forcing a more complete examination of the rewrite graph for a few minutes) being included.



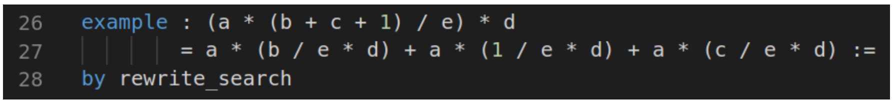
6.3 Arithmetic

Our pair-exploration algorithm for solving general equational lemmas is automatically suited (or at least applicable) to simplifying equations in arithmetic. For example, the `rewrite_search` tactic complements "equation normalisation" tactics present in Lean's standard library, which are able to solve certain classes of problems quickly but cannot handle arbitrary division, for example—`rewrite_search` is necessarily much more robust by design. If `rewrite_search` were able to prove an equational lemma in roughly the same time as it would take to determine that the lemma belongs to a class which can solved by a more efficient algorithm, then the utility of `rewrite_search` as a general purpose interactive Lean tactic would obviously be further improved.

Consider the example of proving the (completely arbitrary) equational lemma

$$(a \times (b + c + 1) / e) \times d = a \times (b / e \times d) + a \times (1 / e \times d) + a \times (c / e \times d)$$

for a, b, c, d, e all rational numbers (and e nonzero). In Lean we would specify this lemma via:



```
example : (a * (b + c + 1) / e) * d
        = a * (b / e * d) + a * (1 / e * d) + a * (c / e * d) :=
by rewrite_search
```

The setting of arithmetic provides an excellent example of the utility of centre-of-mass and SVM weighted edit-distance metrics over the standard unweighted metric.

Fix the parameters $N_p = 10$ and $N_r = 10$ (we increase N_p compared to the previous section to eliminated the "blinder" effect at low setting has on performance in typical arithmetic problems). Then an unweighted search sees a total of 556 `expr`s and visits 89, in order to obtain a proof using 19 rewrites. On the other hand a centre-of-mass weighted search sees 180 `expr`s, visits 25, and finds a proof using 17 rewrites, while an SVM weighted search has the very similar performance of 187 `expr`s seen, 25 visited, and a proof of length 18. There is a corresponding factor-of-ten reduction in execution time as a consequence. This makes use of an unweighted pair-exploration search for this purpose quite impractical, while the weighted variants enjoy convenient interactive behaviour.

6.4 Hypercube complete graphs

Finally, we will examine a much more contrived problem domain compared to those we have just seen, but which will allow us to demonstrate the advantages of an A*-like metric in a simple (and extreme) case. In this example we will not provide the Lean code required to explicitly construct[12] the interactive context in which we will invoke `rewrite_search`—we will be content to just describe it.

We define three functions f, g, and h, all maps $\mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$. We then introduce axioms of the form

$$\begin{gathered}\forall a, b, c \in \mathbf{N} : f(a,b,c) = f(\alpha, b, c)\\ \forall a, b, c \in \mathbf{N} : f(a,b,c) = f(a, \beta, c)\\ \forall a, b, c \in \mathbf{N} : f(a,b,c) = f(a, b, \gamma)\end{gathered}$$

for all possible $\alpha, \beta, \gamma \in X = \{1,2,3\}$. We proceed reproducing the resulting set of axioms twice more; replacing f everywhere with g, and then again with h.

The result is that these axioms specify three disjoint complete graphs each on $|X|^3 = 27$ vertices, with one graph corresponding to each of f, g, and h (the first of which has vertices $f(a,b,c)$ for $(a,b,c) \in X^3$, and analogously for g and h). We then additionally impose the axioms

$$f(2,2,2) = g(3,3,3) \quad \text{and} \quad g(1,1,1) = h(2,2,2).$$

This creates a path from any vertex in any of the complete graphs to any vertex of any other—i.e. combined with the previous axioms asserts that the functions f, g, and h agree on X^3. Our goal will be to prove that $f(1,1,1) = h(1,1,1)$.

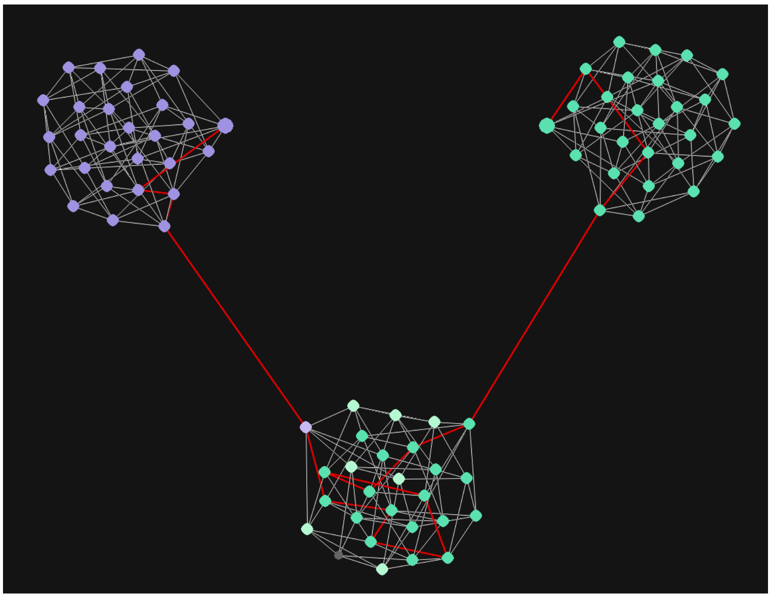
This is an interesting toy environment to consider when testing an A*-like metric, since it is possible to perform several rewrites pointlessly meandering within a single complete graph, before finally moving to a vertex in that graph for which one of the two special axioms above permit rewriting into a different complete graph. The purpose of our A*-metric is of course to try to minimise this behaviour and approximate optimality.

In order to maximise the chance of a "meandering" route being found due to poor knowledge of the total search graph, we set $N_\mathrm{p} = 1$ and choose a default value of $N_\mathrm{r} = 10$. A direct invocation of `rewrite_search` using the unweighted edit-distance metric yields the performance 80 `expr`s seen, 72 `expr`s visited, 17 rewrites used.

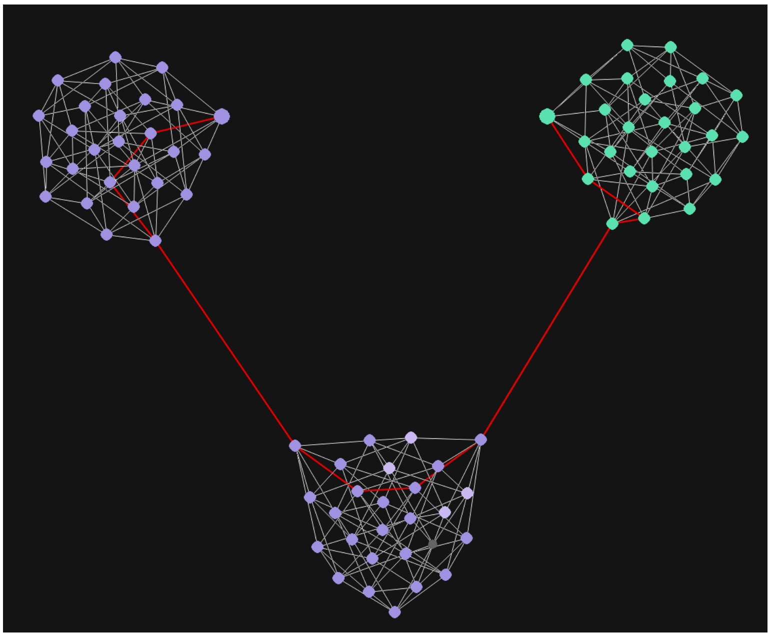
Taking $\alpha = 5$ (an arbitrary, large quantity compared to the standard edit-distance cost of 1) in the definition of the A*-like unweighted edit-distance metric $m_\alpha^{\mathrm{A}^*}$, we

[12] It being too unwieldy and also computer generated. The *Mathematica*™ 11.0 notebook used to generate the example can be found at `https://github.com/semorrison/lean-tidy/blob/master/test/rewrite_search_cube_axioms.nb`.

instead see the much improved result 80 `expr`s seen, 76 `expr`s visited, and only 11 rewrites used in the final proof (this is optimal). Visualising these results, we improved the extremely-meandering path



into the optimal path



.

Because of the relatively high performance impact associated with introducing additional `expr` vertices to the search graph (and hence the desire to introduce the pop-size parameter N_p), it appears promising to set $N_p = 1$ and counteract the "graph blindness" which is incurred by only revealing a single edge at time by using an A*-like approach such as that which we just gave. The A*-approach also exhibits better performance in terms of fewer total `expr`s seen for smaller values of α, say $\alpha = 1$, at the cost of not producing a precisely-optimal length 11 path (we we find a length 14 path in this case). Performance is seen to improve additionally when a classifier-weighting such as centre-of-mass or SVM is introduced.

Finally, note that the long and meandering 17-rewrite path initially obtained could be reduced to a closer-to-optimal length 14 path by running a shortest-path algorithm on the known rewrite graph when any path whatever connecting the left- and right-hand sides is found—as opposed to simple naive backtracking (the former of which we have implemented in our Lean module and recommend). While overlapping with the advantages of the A*-like metric, we expect that A*-like metrics should also decrease path-lengths in situations where the total rewrite graph is too large and sparse to possibly be fully known—a situation where finding a shortest-path using the resulting rewrite graph at the end of a search would likely be of little benefit.

7 Future work

There are many possible directions for future extensions of this work—given its modular nature, each piece can essentially be individually examined and improved on its own. We conclude by giving some recommended directions of further enhancement primarily based on observations made when the algorithm was used in practice.

The fact that our edit-distance calculations are performed on space-tokenized pretty-printed strings is quite a crude procedure, though nonetheless possesses significant power in practical applications (perhaps surprisingly so). Since expressions in Lean are essentially repeated function applications, they possess a natural tree structure which could potentially be exploited for additional information. One possibility is to replace the string-tokenized edit-distance metric with a labelled-tree edit-distance metric, such as that of [5].

In practice, it desirable that a search is able to be left to run for some period of time, especially in the case where it is not even known if a rewrite-path between the two input expressions exists. It is therefore desirable that searches lasting a very large number of pair-exploration iterations remain efficient over time. One factor which has been observed to interfere with this efficiency is a build-up of "stale" interesting pairs which were found earlier in the search, but where additional progress on the problem had been made since and the pairs would never be revisited. Nonetheless, these pairs remain in the global interesting pair list P and can consume calculation time as each new pair selected from P is compared with each of P's its elements. This problem is particular pertinent in situations where a classifier such as SVM refreshes the weights associated to each token, and therefore all distance estimates as well. Each of the distance estimates for the potentially many "stale" pairs must be recalculated in this case, which can be very expensive, in practice even completely pausing the search for a moment. We thus propose that in the interest of performance a strategy for pruning "stale" interesting pairs from the list P—migrating them to some kind of "back-up" list, or just dropping them altogether—be investigated.

An implementation detail of the `rewrite_search` tactic is that if a maximum number of pair-exploration iterations are exceeded, then the tactic aborts returning an error message. This is necessary to prevent situations where an impossible problem is posed to `rewrite_search` and leads to an infinite-loop. However, it is often the case that a rewrite-problem is impossible not because the equality it represents is actually false, but because `rewrite_search` has not been explicitly made aware of a needed rewrite lemma (or this lemma has not yet been proved). It would instead be more desirable if once a maximum number of iterations were reached, the "goal state"—representing what remains of the theorem to be proved—is replaced with a human-readable simplified version of the equality (i.e. measured by the metric to have smallest distance between its two sides) which was original given as input. This would allow missing lemmas to be more easily identified, and also aid in interactively proving theorems where the complete proof strategy is not already fully known.

Finally, we suggest a direct extension to the pair-exploration algorithm itself which is inspired by human behaviour. Often humans are not only able to recognise that particular terms of one side of an equality should be eliminated (as we attempted to motivate using weighted edit-distances), but they go an additional step to conclude that it suffices to show some number of other equalities of subterms, or of intermediary expressions. These "conjectured sub-equalities" are often obtained just by pattern matching, and are frequently those terms most significantly contributing to the edit-distance cost of changing the tokens of one side of the equality to to the other. A general strategy for determining such sub-equalities would be very valuable, both from the perspective of guiding the algorithm and improving performance, and also proving more helpful feedback to an interactive user of the tactic. Tactic users could be informed of the sub-equalities which were determined but could not be discharged automatically, and also the ambient interactive environment could be automatically searched for rewrite lemmas which were not explicitly provided but closely match the conjectured equalities.

References

[1] Leo Bachmair and Harald Ganzinger. "Rewrite-based equational theorem proving with selection and simplification". *Journal of Logic and Computation* 4.3 (1994), pp. 217–247. DOI: 10.1093/logcom/4.3.217.

[2] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: a library for support vector machines". *ACM transactions on intelligent systems and technology (TIST)* 2.3 (2011), p. 27. DOI: 10.1145/1961189.1961199.

[3] Benjamin Grégoire and Assia Mahboubi. "Proving equalities in a commutative ring done right in Coq". *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 98–113. DOI: 10.1007/11541868_7.

[4] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. "Proof-pattern recognition and lemma discovery in ACL2". *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2013, pp. 389–406. DOI: 10.1007/978-3-642-45221-5_27.

[5] Mateusz Pawlik and Nikolaus Augsten. "RTED: a robust algorithm for the tree edit distance". *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 334–345. DOI: 10.14778/2095686.2095692.

[6] Robert A Wagner and Michael J Fischer. "The string-to-string correction problem". *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173. DOI: 10.1145/321796.321811.