U.S. NAVAL ACADEMY
COMPUTER SCIENCE DEPARTMENT
TECHNICAL REPORT

Algorithmic Reformulation of Polynomial Problems

Brown, Christopher W.

USNA-CS-TR-2007-01

June 13, 2007

USNA Computer Science Dept. ◦ 572M Holloway Rd Stop 9F ◦ Annapolis, MD 21403

Algorithmic Reformulation of Polynomial Problems

Christopher W. Brown
Department of Computer Science, Stop 9F
United States Naval Academy
Annapolis, MD 21402, USA
wcbrown@usna.edu

June 13, 2007

Abstract

This paper considers the problem of existential quantifier elimination for real algebra (QE). It introduces an algorithmic framework for exploring reformulations of QE problems, with the goal of finding reformulations that make difficult problems tractable for QE implementations, or for which these implementations find simpler solutions. The program `qfr` is introduced, which implements this approach, and its performance on some example problems is reported.

1 Introduction

"Why do it like a human when you can do it right." — Joel Moses

The above quote is indicative of a philosophy underlying computer algebra. We don't emulate human methods of solution in our algorithms for problems like integration, summation, factorization, solutions of systems of polynomial equations and a host of other problems. In fact, humans aren't very good at these problems, while computer algebra programs have proven to be remarkably effective. Why approach a mathematical problem like a person? Why not do it right?

This paper considers a very fundamental problem — quantifier elimination for the first order theory of real algebra. Roughly speaking, this means determining the satisfiability of systems of polynomial equalities and inequalities over the

reals, where the polynomials involved may have coefficients that are expressions in some parameters. When coefficients are rational functions of the parameters, this problem is solvable, and there has been a lot of work over many years on algorithms and software for doing it. The general approaches taken — Tarski's original approach [11], Collins' cylindrical algebraic decomposition [5], and the "root counting" approach of Weispfenning [15] and of Heintz, Roy and Salerno [9] — are typical of computer algebra in that they do not emulate the ways in which people solve such problems.

The thesis of the paper is that this subject would benefit from a bit of trying to "do it like a human" rather than just "doing it right". While it is true that humans are not good at solving these problems, neither are machines. The fundamental problem is computationally intractable in any traditional sense. What humans can do, however, that these algorithms cannot, is flexibly exploit problem-specific structure — the kind of structure that real-world problems have. The following problem, which comes from epidemiological modeling [12], is such an example.

$$\exists S, E, I, T \left[\begin{array}{l} d - dS - \beta_1 IS = 0 \wedge vE - (d + r_2)I = 0 \wedge \\ \beta_1 IS + \beta_2 IT - (d + v + r_1)E + (1 - q)r_2 I = 0 \wedge \\ -dT + r_1 E + qr_2 I - \beta_2 TI = 0 \wedge E > 0 \wedge I > 0 \wedge T > 0 \wedge S > 0 \end{array} \right]$$

where $d > 0 \wedge v > 0 \wedge r_1 > 0 \wedge r_2 > 0 \wedge q > 0 \wedge \beta_1 > \beta_2 > 0$

None of the software systems we have tried on this problem[1], as formulated, can solve it — at least not in the amount of time we were willing to wait. However, in the cited paper, the authors solve it by hand. Several [all?] of the above systems can be coaxed into solving the problem if we first do what any normal person would do: solve for linearly occurring variables and substitute. Doing this, which is a bit tedious, we can eliminate all but one of the variables. Software can take care of the last one.

Actually, there is another sense in which the way people solve problems like the above is better than that of our algorithms: people produce simple answers. In solving such problems we seek out steps that keep expressions simple, and we systematically exploit the problem's structure to rule out special cases, which has a lot to do with our ability to solve these problems while automated procedures fail.

This paper's major goal is to provide an efficient algorithm/data-structure foundation for attacking these problems in a human-like manner inside a program.

It is important to note that we are not suggesting that quantifier elimination problems be solved solely by mimicking people, rather that programs should try

[1] We tried: QEPCADB v1.45, Redlog v3.0, Mathematica v5.2 and the RS function of the Salsa Maple packages from INRIA.

to exploit problem-specific structure like people do. Typically, this will generate one or more simpler quantifier elimination problems that would then be solved by regular quantifier elimination programs. It is also important to note that this approach has nothing to offer for many kinds of QE problems. Higher degree problems, problems without equations, very generic problems ... problems like these that humans can't make much progress with will not benefit. However, we will exhibit a variety of application problems from diverse sources that our approach does apply to, and does succeed, in combination, with other programs, to give good quantifier free solutions in a reasonable amount of time.

1.1 A more precise formulation of the problem

The kind of human problem solving we are trying to emulate basically amounts to rewriting or reformulating existential quantifier elimination (QE) problems. Starting with an initial problem, we eliminate variables by substitution when they occur linearly in equations; we split into cases depending on, for example, whether a leading coefficient is zero or not; we remove redundancies like $0 < b$ in $a < 0 \wedge b < 0 \wedge a < b$; and much more in the same vein. We arrive at one or more simpler (we hope!) problems that we can't do any more with, and these will have to be solved by other methods. Thus, the fundamental problem is this: Given an existential QE problem P and a set of rewriting operators, find the "best" reformulation of P as one or more simpler QE problems.

Ultimately, one would be interested in finding the best rewriting — or even just a good rewriting — quickly. For this paper, however, we only consider the problem of generating all rewritings and finding the best. Our main result is an algorithm that efficiently searches the space of all rewritings of a given QE problem to find the one that is "best" with respect to a suitable metric. This algorithm provides a method for comparing fast, heuristic-based algorithms for rewriting in future work, and a foundation on which such algorithms can be based.

2 The "space of rewritings"

One of the basic concepts behind the AI search paradigm is that of the *state space*. An initial state and a set of operators that map states to potential successor states together generate a state space. In our case, states are formulas. The initial state is the input formula, and operators produce new formulas that are logically equivalent over the reals. One such operator, for example, rewrites formulas of the form $ax+b = 0 \wedge F$ as $\left(a \neq 0 \wedge F|_{x \leftarrow -b/a}\right) \vee (a = 0 \wedge b = 0 \wedge F)$. The basic idea is, starting with an input formula and a set of rewrite operators,

to explore the state space and return the rewriting that we deem to be "best". This differs from the usual AI search scenario, because there's no goal test — there's no way to look at a single state in isolation and determine that it is "the answer".

There are many rewritings that are completely uninteresting. For example, rewriting $f^5 > 0$ as $f^3 > 0$ isn't interesting. In fact, we have no reason to ever want to see either expression in any formula, since they are both equivalent to $f > 0$. As another example, consider $F \vee x^2 + 1 < 0 \wedge G$. Any rewritings of G are completely uninteresting, since the disjunct it is part of is equivalent to false anyway. Rewriting G is a waste of time and space. What these two examples point to is the need to distinguish between what we call "normalization" and the rewritings performed by operators. Normalization involves rewritings we will always want to carry out: e.g. $f^5 > 0 \longrightarrow f > 0$, or $x^2 + 1 < 0 \longrightarrow false$. Rewrite operators carry out rewritings that are not always desirable.

Another issue is the need recognize states that are the same as states that have already been processed. In different search contexts, "the same" might mean different things. In our context, all states are logically equivalent over the reals, so defining "the same" too deeply is counterproductive. We take "the same" to mean syntactically identical after normalization.

The straightforward approach to generating the space of rewritings then is given in Algorithm 1. If no operator introduces disjunctions, this kind of search

Algorithm 1 Naive exploration of the space of rewritings.

```
1: enqueue F in A
2: while Q not empty do
3:    G := dequeue Q
4:    for each rewriting operator application op do
5:       G' := result of applying op to G
6:       G'' := normalization of G'
7:       enqueue G'' in Q (unless G'' has already been generated)
```

might be reasonable. However, when disjunctions can be introduced (and this is the case with most interesting operators), this naive search suffers from two debilitating sources of inefficiency: Suppose $F_1 \vee F_2 \vee \cdots \vee F_k$ is a formula encountered during the search.

1. If $op_1, \ldots, op_k$ are operators such that op_i acts on disjunct F_i, there are 2^k distinct formulas generated by different orders of applying the operators (barring coincidental duplications). In other words, even though there is one "destination", namely $op_1(F_1) \vee op_2(F_2) \vee \cdots \vee op_k(F_k)$, there are 2^k steps if you traverse all possible paths getting there, because of all the different orders one can use to apply operators.

2. If, for each i, there are s_i operators that can be applied to F_i, then even forgetting about the orders in which operators are applied to disjuncts, there are $(1 + s_1)(1 + s_2) \cdots (1 + s_k)$ ways that a new formula can be generated by applying no more than one operator do each disjunct.

Of course, these two factors combine to produce an enormous number of possible rewritings. Moreover, in neither case have we considered applying a rewrite operator to the result of a rewrite operator — e.g. "substitute $x = -b/c$ into F then substitute $y = -d/e$ in the result".

As described in [3], Christian Gross implemented a heuristically guided version of above algorithm, which prunes away parts of the search tree based on a function for "grading" formulas. This program is able to find good rewritings for several interesting inputs, but only when they are relatively easy to find, since the space it has to search is so vast. In fact, it discovers very few distinct formulas, spending most of its time discovering duplicates. Moreover, the only operators it includes are factor splitting for equations and linear substitution. Presumably, incorporating more operators would exacerbate the problems associated with search space size.

Gross's implementation is intended to serve as a "preprocessor" for quantifier elimination program; primarily for QEPCADB, but also for Redlog or Mathematica. For several problems from different domains, it is able to find an input rewriting that make an intractable problem tractable for these systems, or improve the quality of answer. Both the successes of this program and its shortcomings motivate the search for a better approach to exploring the space of rewritings.

3 Searching more efficiently

The first idea for improving the search through the space of rewritings is to decouple disjuncts in a formula. The two major factors cited in the previous section can both be avoided by searching for rewritings of disjuncts independently. Given formula $F \vee G$, we generate the set S_F of rewritings of F and S_G of rewritings of G. These two sets represent the $|S_F| \cdot |S_G|$ rewritings obtained by forming the disjunction of an element of S_F and an element of S_G. Of course, this process is recursive, meaning that in searching for rewritings of F or G we might form new disjunctions, and we should treat the search for rewritings of each disjunct as independent problems.

The second idea for improving the search is that since we expect to generate the same disjuncts in different ways, and since we expect to discover that some disjuncts are unsatisfiable, we should be able to exploit of such information.

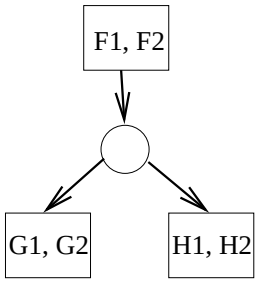

Figure 1: A simple rewrite graph representing the following formulas as equivalent: $F_1, F_2, G_1 \vee H_1, G_1 \vee H_2, G_2 \vee H_1, G_2 \vee H_2$.

In the next section we describe a data structure, the "rewrite graph", that allows us to implement the above ideas and thus to search much more effectively. Nodes in this graph are either Q-nodes, which represent a set of equivalent formula, or OR-nodes which represent disjunctions. The simple graph in Figure 1 represents the following formulas, all of which are equivalent: $F_1, F_2, G_1 \vee H_1, G_1 \vee H_2, G_2 \vee H_1, G_2 \vee H_2$. We also show how this data structure can be reorganized when a formula in a node is found to be false, or when two nodes are found to contain the same subformula. We will then reformulate our search as the process of choosing an unprocessed formula from some Q-node in the rewrite graph, applying operators to it and modifying the graph accordingly. An implementation of this approach is described later in the paper, along with some empirical results on its performance.

4 The primary data structure

If S is a set of equivalent formulas then, abusing notation a bit, we will allow S to appear as a formula in expressions, with the same meaning as using any one of its elements in its place.

Let F be an existentially quantified formula with free variables $x_1, \ldots, x_k$ and bound variables $x_{k+1}, \ldots, x_n$. We describe a data structure called a *rewrite graph* (RG).

Definition 1 *A Rewrite Graph (V_Q, V_{OR}, E, S) is a directed acyclic, bipartite graph (V_Q, V_{OR}, E) with function S that maps elements of V_Q to Tarski formulas, such that:*

- *For "Q-node" $v \in V_Q$, all elements of $S(v)$ (which we sometimes write S_v) are mutually equivalent.*

- *If "OR-node" $w \in V_{OR}$ is a child of Q-node v, and Q-nodes $u_1, \ldots, u_a$ are the children of w, then $S(v) \Leftrightarrow \bigvee_{i=1}^{a} S(u_i)$.*

Rewrite graphs give us a compact way to represent a very large number of rewritings of a formula. The following definition and theorem make that precise.

Definition 2 *Given rewrite graph $G = (V_Q, V_{OR}, E, S)$, define the function H mapping vertices into Tarski formulas as follows:*

1. *If v is a Q-node with children $w_1, \ldots, w_a$,*
   $H(v) = S(v) \cup H(w_1) \cup H(w_2) \cup \cdots \cup H(w_a)$.
2. *If w is an OR-node with Q-node children $u_1, \ldots, u_a$,*
   $H(w) = \{\bigwedge_{i=1}^{a} f_i \mid (f_1, \ldots, f_a) \in H(u_1) \times \cdots \times H(u_a)\}$.

Theorem 1 *Given rewrite graph $G = (V_Q, V_{OR}, E, S)$ and node $v \in V$, every formula in $H(v)$ is equivalent.*

PROOF. This is an obvious consequence of the definition of rewrite graphs. □

The general idea is that we start with some formula F and construct a rewrite graph G, rooted at node r, such that $F \in S(r)$. Then $H(r)$ gives the set of rewritings of F.

We will allow ourselves a further abuse of notation by using RG-nodes as formulas, e.g. $x_1 > 0 \wedge v$, where v is an RG-node. In this context, v has the same meaning as any element of $H(v)$.

RG's play a key role in our algorithm for exploring rewritings of an existentially quantified formula. Not only do they provide a compact representation of a large number of possible rewritings, they give a context to information that gets discovered during the rewriting process, which allows us to exploit such discoveries.

Theorem 2 *Let $G = (V_Q, V_{OR}, E, S)$ be a rewrite graph.*

1. *For any i, $1 \leq i \leq k$, and $\alpha \in \mathbb{R}$, $G|_{x_i=\alpha} = (V_Q, V_{OR}, E, S')$, where $S'(v) = \{f|_{x_i=\alpha} \mid f \in S(v)\})$, is also a rewrite graph. Essentially this says that assigning a value to a free variable in a rewrite graph yields a rewrite graph.*
2. *If for Q-nodes $u, v \in V_Q$ we have $S_u \cap S_v \neq \emptyset$, $S_v \Leftrightarrow S_u$.*

3. *If for Q-node v any element of S_v is found to be equivalent to false, then v and all descendent nodes are equivalent to false.*

4. *If for Q-node v any element of S_v is found to be equivalent to true, then all nodes along any path to v are equivalent to true.*

5. *Let $v_1 \to w_1 \to v_2 \to \cdots \to w_s \to v_{s+1}$ be a path in G, where Q-nodes v_1 and v_{s+1} are known to be equivalent. Then $v_1, v_2, \ldots, v_s, v_{s+1}$ are all equivalent. (From which it trivially follows that $w_1, \ldots, w_s$ are all equivalent to $v_1, \ldots, v_{s+1}$.)*

PROOF. The first four points are easily seen to be true, so we only explicitly prove the last point. We will show that $v_1 \Leftrightarrow v_s$, and the conclusion follows by induction. In fact, it suffices to show that $v_1 \Leftrightarrow w_s$, since by our hypotheses $v_s \Leftrightarrow w_s$. So, let α be a point in free-variable space. If $v_1|_\alpha$ is true then $v_{s+1}|_\alpha$ is true, by hypothesis, and that in turn means $w_s|_\alpha$ is true, since v_{s+1} is a disjunct of w_s. Conversely, if $w_s|_\alpha$ is true, then $v_s|_\alpha$ is true, which means that, by point 1 and 4, v_1 is true. □

4.1 Reduced rewrite graphs: exploiting what we learn about formulas

In exploring the space of rewritings of a formula, two events can occur that provide information that we would like to exploit: we may discover that some formula is unsatisfiable, or we may discover that two formulas obtained through different rewritings are identical. The rewrite graph allows us to exploit this information.

Definition 3 *A Reduced Rewrite Graph (RRG) is a rewrite graph satisfying some additional restrictions:*

1. *There is no vertex $v \in V_Q$ such that $S(v)$ contains true or false.*

2. *There are no two vertices $u, v \in V_Q$ such that $S(v) \cap S(u) \neq \emptyset$.*

3. *There are no two vertices in V_{OR} with the same out-neighbor set.*

4. *Each vertex in V_{OR} has a unique in-neighbor.*

5. *Each OR-node has at least two children.*

The motivation for this definition is the goal of keeping the rewrite graph as simple as possible while still describing all of the same "interesting" rewritings.

Rewritings with constants, known redundancies, or inconsistencies are not "interesting". Point 1 of the definition means we don't allow constants, points 1 & 4 mean that we don't allow distinct but obviously equivalent Q-nodes, and point 3 means we don't allow distinct but obviously equivalent OR-nodes. Nicely enough, Theorem 2 provides us with the tools we need to algorithmically transform an arbitrary rewrite graph into a reduced rewrite graph. The principal work in that process is done by Algorithm 2.

Algorithm 2 takes as input a rewrite graph $G = (V_Q, V_{OR}, E, S)$ and a queue Q of assertions of the form (a, b), where $a, b \in V_Q \cup \{false\}$. Extending the function S by defining $S(false) = \{false\}$, assertion (a, b) is the statement $S(a) \Leftrightarrow S(b)$. The algorithm transforms G by deleting any nodes asserted to be false, and merging any nodes asserted to be equivalent. However, it accomplishes this in such a way that G remains a rewrite graph. Moreover, in this process no rewritings are "lost". More precisely, if a rewriting is represented by G, the same rewriting exists in the RG produced by the algorithm — except that some disjuncts may be removed if the assertions in Q and/or the structure of G determine that they are redundant.

One complicating issue in presenting Algorithm 2 is that as nodes are merged or deleted, the node names in Q may become stale. For example, if (a, b) is dequeued, where a and b are nodes, but node b has already been merged with other nodes, what do we do? The algorithm deals with this by keeping track of "aliases". If node a is deleted because a was asserted to be false, the name a is then considered to be an alias for false. If node b is merged with others into a new node c, then b is considered to be an alias for c. When a possibly stale name a is encountered, the algorithm simply asks for the true name of a before doing anything. The functions record and trueName provide this bookkeeping.

Theorem 3 *Let $G = (V_Q, V_{OR}, E, S)$ be a rewrite graph with root r, and suppose Q contains $(u_1, v_1), \ldots, (u_k, v_k)$. Let $G' = (V'_Q, V'_{OR}, E', S')$ be the graph resulting from applying Algorithm 2 to G and Q.*

1. *G' is a rewrite graph under the assumption $\bigwedge(u_i \Leftrightarrow v_i)$*
2. *For $1 \leq i \leq k$, trueName(u_i) = trueName(v_i) after Algorithm 2 finishes.*
3. *if $C_1 \vee \ldots \vee C_m \in H_G(r)$, then $H_{G'}$(trueName(r)) contains some formula $C_{i_1} \vee \cdots \vee C_{i_n}$, where $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, m\}$, such that*

$$\bigwedge(u_i \Leftrightarrow v_i) \implies (C_1 \vee \ldots \vee C_m) \Leftrightarrow (C_{i_1} \vee \cdots \vee C_{i_n})$$

Proof. First we observe that the algorithm terminates. At every iteration an element is dequeued from Q. In trivial iterations, i.e. when $u = v$ in line 4,

Algorithm 2 Reorganize $G = (V_Q, V_{OR}, E)$ based on Q, a queue information elements from $(V_Q \cup \{false\})^2$

```
1: while Q not empty do
2:    (u, v) := dequeue(Q)
3:    u := trueName(u), v := trueName(v)
4:    if u = v then continue
5:    if v = false then swap(u, v)
6:    if u = false then
7:       record(v, false), i.e. record that v is an alias for false
8:       for each child w of v do
9:          for each child x of w do
10:            enqueue((false, x), Q)
11:         for each parent v' ≠ v of w do
12:            enqueue((false, v'), Q)
13:      delete v
14:   else
15:      set W_Q to the set of all Q-nodes along paths from u to v or v to u
16:      set W_OR to the set of all OR-nodes along paths from u to v or v to u
17:      let x be a new Q-node and define S(x) = ⋃_{t∈{u,v}∪W_Q} S(t)
18:      for each s ∈ V_Q − W_Q such that (s, w) ∈ E, for some w ∈ W_OR do
19:         enqueue((s, x))
20:      delete all nodes in W_OR
21:      for each w ∈ W_Q ∪ {u, v} do
22:         record(w, x), i.e. record that w is an alias for x
23:      contract the elements of {u, v} ∪ W_Q into the new node x
24: delete any nodes that are unreachable from the original root
```

nothing is enqueued in Q. In each non-trivial iteration, the number of nodes in the graph decreases, thus the algorithm terminates.

Second we prove that G' is acyclic. Note that line 23 is the only place in the algorithm with the potential to introduce cycles, since the other lines that modify the graph only delete edges or nodes. If the contraction of the elements of W_Q into a single node were to create a cycle, it would have even length (since the contraction maintains the bipartite nature of the graph) and it would have to contain a vertex outside of $W_Q \cup W_{OR}$, since we've deleted the edges from W_Q into W_{OR}. If such a cycle exists, then contracting $W_Q \cup W_{OR}$ into a single node (and deleting edges from that node to itself) would produce a graph with a cycle (derived from the other cycle), which Lemma 1 proves is impossible.

Next we prove that for each $v \in V'_Q$, all elements of $S'(v)$ are equivalent given $\bigwedge(u_i \Leftrightarrow v_i)$. This requires showing that our graph modifications are valid given the assertions in Q, and that any new assertions we add to Q are valid. For an assertion of the form "node v is $false$", our only action on the graph is to delete every edge into v. Since all such edges are from OR-nodes, this is equivalent to removing v from a disjunction, which is valid given the assertion. Any child w of v is an OR-node which, by definition, is equivalent to v and therefore is equivalent to $false$. This triggers two kinds of new assertions: first that any child of w is $false$, which is valid since a disjunction is false if and only if each disjunct is false, and second that any other parent of w is false, which is clear since an OR-node is equivalent to its parent. These are exactly the assertions added by the algorithm.

For an assertion of the form "nodes u and v are equivalent", we do more. Point 5 of Theorem 2 shows that all nodes that are on paths from u to v or v to u are equivalent — these are $W_Q \cup W_{OR}$. Any parent of an element of W_{OR} is clearly equivalent to the elements of V_Q, so the new assertions added in line 22 are justified. The algorithm deletes all nodes in W_{OR}, then contracts W_Q into a single new vertex x. Clearly contracting W_Q into a single new node is justified. Deleting W_{OR} is justified by the observation that any $w \in W_{OR}$ has a parent $a \in V_Q$ and a child $b \in V_Q$ and, by Theorem 2, $a \Leftrightarrow w \Leftrightarrow b$. Thus, we can eliminate all children other than b from w without changing its meaning, at which point w provides no information other than that $a \Leftrightarrow b$, which the contraction makes explicit, and that any other parent of w is equivalent to b which the contraction and/or the assertions previously added make explicit. This concludes the proof of Point 1.

Point 2 is clear from the fact that the algorithm only terminates when the queue is empty, which means that all the input assertions have been processed. Point 3 essentially asserts that all the rewritings represented in G are represented in G', except that some redundant disjuncts may have been deleted. This should be clear, though we point out that new rewritings may be represented in G'. □

Algorithm 3 uses Algorithm 2 to transform a rewrite graph into an equivalent rewrite graph. Essentially, all it has to do is detect violations of the conditions from the definition of reduced rewrite graph, convert them into assertions in a queue, and call Algorithm 2 to reorganize the graph based on the assertions.

Algorithm 3 Rewrite graph to reduced rewrite graph. **Input:** $G = (V_Q, V_{OR}, E, S)$, a rewrite graph with root node $r \in V_Q$. **Output:** G is transformed into an equivalent reduced rewrite graph. Note: Assume for all $v \in V_Q$, $true \notin S(v)$.

```
1:  while G is not reduced do
2:    set Q to an empty queue
3:    for all v ∈ V_Q such that false ∈ S(v) do
4:      enqueue((false, v)) in Q
5:    for all u, v ∈ V_Q such that S(u) ∩ S(v) ≠ ∅ do
6:      enqueue((u, v)) in Q
7:    for all w, x ∈ V_OR such that w and x have the same out-neighbor set do
8:      for each in-neighbor u of x, add edge (u, w)
9:      delete x
10:   for all u, v ∈ V_Q that share out-neighbor w ∈ V_OR do
11:     enqueue((u, v)) in Q
12:   for all w ∈ V_OR with no out-neighbors do
13:     for each u with an edge to w enqueue((false, u)) on Q
14:     delete w
15:   for all w ∈ V_OR with exactly one out-neighbor v do
16:     for each u with an edge to w enqueue((u, v)) on Q
17:     delete w
18:   Reorganize(G, Q)
```

4.2 Searching with rewrite graphs

With the machinery of the reduced rewrite graph in place, our view of search changes a bit. We will use a reduced rewrite graph to represent the formulas discovered by searching.

The search starts with an input formula f, and a rewrite graph consisting of a single Q-node r such that $S(r) = \{f\}$. As before, each search iteration consists of choosing a formula g and generating all the subformulas produced by applying rewrite operators to g. The difference is that the formula we choose is not, in general, equivalent to the input f. Instead, it is an element of $S(u)$ for some Q-node u in the rewrite graph. if a new formula h is generated that has no disjunctions, it simply gets added to $S(u)$. If a new formula $h_1 \vee \cdots \vee h_k$ is generated, a new OR-node child of u is created, and for each h_i, a new Q-node v_i, where $S(v_i) = \{h_i\}$, is created, all of which are children of the new OR-node.

Subformulas produced by rewrite operators may be normalized to $false$ (or $true$ if we're really lucky, since the entire search can be terminated and $true$ returned at that point), or may be normalized to a formula that has already been generated. Either event triggers a reorganization of the rewrite graph along the lines of Algorithm 3 in order to keep it reduced. This reorganization never creates new formulas, it merely moves them to different nodes, or removes them from the graph altogether.

Search terminates when there are no subformulas in nodes that have not already been rewritten using the rewrite operators. At this point, the rewrite graph implicitly represents all the rewritings generated by the given operators. From it, we can very easily pull out the rewriting that maximizes any grading function that distributes over disjunctions, e.g. a function satisfying $p(f \vee g) = p(f) + p(g)$.

Search based on rewrite graphs is a vast improvement over the generic AI-inspired search for two reasons: 1) we avoid the inefficiencies outlined in Section 2, and 2) we are able to expoit information discovered during the search to throw out subformulas — often before they are ever rewritten.

5 Implementation: qfr

We have implemented the approach described above in a program called `qfr` — quantified formula rewriting. The program makes some assumptions that are not required by the framework described above, most notably that polynomials are always kept in fully factored form, and that sets associated with Q-nodes contain no disjunctions. These design decisions simplified the system, but limit it to some extent. Additionally, our framework requires three components:

1. a normalizer,
2. a set of rewrite operators, and
3. a mechanism for choosing the next formula to process.

Several good normalization and/or rewrite operations come from the discussion of formula simplification in [7], and the descriptions of the method of quantifier elimination by virtual term substitution given in [14].

5.1 The normalizer

The role of the normalizer is two-fold: making formulas that are essentially identical syntactically identical (and thus easy to identify as "the same"), and discovering unsatisfiable formulas. In fact, it's probably desirable to separate these two activities, though qfr doesn't. We identified four different "levels" of simplification. Levels 1 and 2 assume that atomic formulas are the fully factored form $p_1 p_2 \cdots p_k \sigma 0$, but never examine the factors themselves. Levels 3 and 4 try to deduce information about the factors themselves. Normalization operates on a conjunction.

1. Level 1 normalization simplifies atomic formulas individually: removing content, eliminating exponents from $=$ and $\neq$, normalizing all exponents to 1 and 2 in $\geq$ and $\leq$, splitting $>$ and $<$ into even and odd factors (odds get exponent 1 and stay in the inequality, evens get exponent 1 as well, but broken into $\neq$ atoms), and breaking up atoms like $p_1 \cdots p_k \neq 0$ into $p_1 \neq 0 \wedge \cdots \wedge p_k \neq 0$.

2. Level 2 normalization assumes Level 1 normalization of atoms, and simplifies the conjunction by merging atomic formulas with the same left-hand side and, when possible, using (in)equalities on a single factor to simplify multi-factor (in)equalities — e.g. simplifying $x+1 > 0 \wedge (x+1)(x^2 - ax + b) < 0$ to $x + 1 > 0 \wedge x^2 - ax + b < 0$.

3. Level 3 simplification attempts to determine sign conditions of variables implied by the formula, and use those sign-conditions to determine whether sign-conditions on some factors are implied by the formula. For example, given $x - 1 > 0 \wedge x + y^2 + 1 < 0$, Level 3 normalization would deduce that $x - 1 > 0$ implies $x > 0$. It would then determine that with $x > 0$, $x + y^2 + 1$ must be positive, and thus that the formula is unsatisfiable.

4. Level 4 normalization uses the implied sign-conditions on variables along with sign-conditions on one other factor to try to deduce sign-conditions on a factor. For example, given $x + 1 > 0 \wedge 2x + y^2 + 2 < 0$, Level 4 normalization would deduce that $2x + y^2 + 2 - 2(x+1) < 0$, which implies $y^2 < 0$, and thus the input is unsatisfiable.

The system allows some control over what normalization gets done, and whether it is only used for determining unsatisfiability, or whether it is also used for simplification. All levels are very fast relative to polynomial factorization, which is the biggest bottleneck in the system. Obviously, Level 4 normalization depends quadratically on the number of factors in the conjunction, which makes it the most time consuming of the levels. On the other hand, discovering unsatisfiable subformulas, or strengthening an inequality into an equality can have an immense global impact. So it is worth expending some time.

There are several other operations that are probably worth incorporating, including breaking sums of squares equalities into separate equalities, and using more sophisticated tests for unsatisfiability.

5.2 The rewrite operators

Rewrite operators generate the space that we search, and thus determine what we can find. However, too many operators, or operators that generate too many rewritings can be a problem, as they will swamp the system. Ultimately, a heuristic guided search of the space should ameliorate such problems. For purposes of this paper, however, we always search the entire space of rewritings. The following rewrite operators are in the current version of qfr.

1. Splitting multi-factor equalities:

$$\left(\Pi_{i=1}^{k} p_i = 0\right) \wedge F \longrightarrow \vee_{i=1}^{k} \left(p_i = 0 \wedge F\right)$$

2. Linear substitution:

$$ax + b = 0 \wedge F \longrightarrow \left(a \neq 0 \wedge F|_{x \leftarrow -b/a}\right) \vee \left(a = 0 \wedge b = 0 \wedge F\right)$$

3. Linear substitution for x^k: if x only occurs in F to powers that are multiples of k, and $t = (k+1) \bmod 2$ then

$$ax^k + b = 0 \wedge F \longrightarrow \left(a \neq 0 \wedge F|_{x^k \leftarrow -b/a} \wedge tba < 0\right) \vee \left(a = 0 \wedge b = 0 \wedge F\right)$$

4. Linear S-polynomial reduction:

$$f = 0 \wedge g = 0 \wedge F \longrightarrow af - bg = 0 \wedge g = 0 \wedge F,$$

where $a, b \in \mathbb{Z} - \{0\}$, and $af - bg$ is linear in some quantified variable x, but neither f nor g are linear in x.

Linear substitution and factor splitting are the most obvious operators. Linear substitution for x^k has been identified as crucial by the implementors of Redlog. Linear S-polynomial reduction is there by virtue of the fact that it is something that we've used by hand in the past. It would be interesting to look at a more general idea of this kind, i.e. Gröbner style reduction by equations to produce linear polynomials that can be substituted. However, the complexity increases when more polynomials can be involved, and multiplying by non-constant factors generates more case distinctions. So adding such an operator would have to be carefully thought out.

5.3 Controlling the search

The basic mechanism behind the search in `qfr` is a priority queue of subformulas. The program is essentially no more than a loop consisting of 1) dequeuing a formula, 2) applying all relevant operators to that formula, 3) adding formulas the queue (if they are new) and to the rewrite graph, and 4) reorganizing the rewrite graph as needed.

The priority queue in the version of `qfr` described here simply follows a fewest-quantified-variables-first rule, with ties broken by the printed length of the formula. Once the entire space has been searched, we print out the "best" rewriting based on a simple grading scheme — although determining what is the "best" rewriting implicit in the graph is outside the scope of this paper.

6 Performance on example problems

The fundamental question for this report is whether or not the system can search the entire space of formula rewritings for interesting sized input formulas, and how large that space is. We will focus on a few example problems. and demonstrate that it can in many instances. Moreover, we'll see instances in which it does better than QEPCADB, Redlog or Mathematica individually. All timings given are on a 1.6 GHz Pentium with 512MB of memory.

6.1 The SEIT Problem

First we consider the example from the introduction: the system of equations arising from finding the equilibrium points of the SEIT model described in the introduction. As a disclaimer, it should be pointed out that the inability of QE software to handle this problem motivated the present work. So the fact that `qfr` does a good job with it shouldn't be too surprising. Hopefully it is an instructive illustration none the less. Recall that this quantifier elimination problem is:

$$\exists S, E, I, T \left[\begin{array}{l} d - dS - \beta_1 IS = 0 \wedge vE - (d + r_2)I = 0 \wedge \\ \beta_1 IS + \beta_2 IT - (d + v + r_1)E + (1 - q)r_2 I = 0 \wedge \\ -dT + r_1 E + qr_2 I - \beta_2 TI = 0 \wedge E > 0 \wedge I > 0 \wedge T > 0 \wedge S > 0 \end{array} \right]$$

where $d > 0 \wedge v > 0 \wedge r_1 > 0 \wedge r_2 > 0 \wedge q > 0 \wedge \beta_1 > \beta_2 > 0$

`qfr` is able to search the entire space of rewritings of this problem with the given operators in 87 seconds. The rewrite graph grows and contracts in the process,

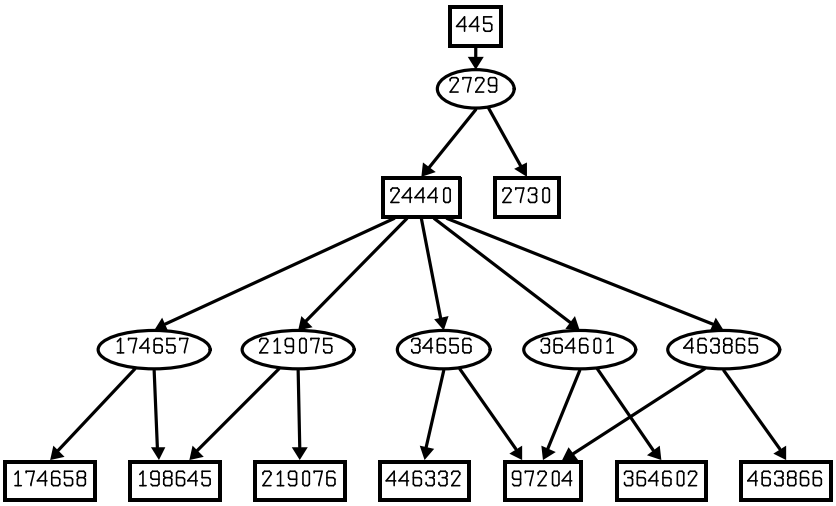

Figure 2: Rewrite graph resulting from searching the full space of rewritings for the SEIT problem.

as Q-nodes are discovered to be false or distinct Q-nodes are discovered to be equivalent. In the end, the graph (see Figure 2) contains just 10 Q-nodes and six OR-nodes, with 759 conjunctions distributed across the Q-nodes. It represents 118,535 distinct rewritings — and that is, of course, after having thrown away many disjuncts that it discovered to be unsatisfiable or redundant. The "best" formulation found is

$$\exists s \left[\begin{array}{l} s > 0 \wedge s - 1 < 0 \wedge \beta_1 sqr_2 v + \beta_1 s^2 \beta_2 v - ds\beta_2 v - \beta_1 s\beta_2 v + d\beta_2 v - \beta_1^2 s^2 v \\ +\beta_1 dsv + \beta_1 sr_1 r_2 - ds\beta_2 r_2 + d\beta_2 r_2 + \beta_1 dsr_2 + \beta_1 dsr_1 - d^2 s\beta_2 \\ +d^2\beta_2 + \beta_1 d^2 s = 0 \wedge \\ \beta_1 > 0 \wedge \beta_2 > 0 \wedge d > 0 \wedge v > 0 \wedge r_1 > 0 \wedge r_2 > 0 \wedge q > 0 \wedge \beta_2 - \beta_1 < 0 \end{array} \right],$$

and, in fact, `qfr` finds that rewriting less than a second into its search. QEPCADB is able to solve this formulation quite easily and with the simplest possible answer, provided the conditions on the free variables are passed along as "assumptions". Redlog and Mathematica solve the above QE problem instantly as well, but with larger formulas.

6.2 The Joswig–Witte Problem

In [10], the truth of a certain conjecture in shown to depend upon the satisfiability of the following system of equations and constraints:

$$\exists s, x_1, x_2, x_3, x_4 \left[\begin{array}{l} 1 + s^2 x_1 x_3 + s^8 x_2 x_3 + s^{19} x_1 x_2 x_4 = 0 \\ \wedge \\ x_1 + s^8 x_1 x_2 x_3 + s^{19} x_2 x_4 = 0 \\ \wedge \\ x_2 + s^{11} x_1 x_4 + s^{10} x_3 x_4 = 0 \\ \wedge \\ s^4 x_1 x_2 + x_3 + s^{19} x_1 x_3 x_4 + s^{24} x_2 x_3 x_4 = 0 \\ \wedge \\ x_4 + s^{31} x_1 x_2 x_3 x_4 = 0 \\ \wedge \\ 0 < s < 1 \end{array} \right].$$

The high degrees of s in this system makes it difficult to solve. Even computing a Gröbner basis requires many hours and the right program. `qfr` is able to search the entire space of rewritings for this problem in less than 5 1/2 minutes. The "best" rewriting it discovers isXS

$$\exists s, x_1 \left[\begin{array}{l} s > 0 \wedge s - 1 < 0 \wedge x_1 \neq 0 \wedge x_1 + 1 \neq 0 \wedge x_1 - 1 \neq 0 \\ \wedge s^{23} x_1 - 1 \neq 0 \wedge P = 0 \wedge Q = 0 \end{array} \right],$$

where P and Q are relatively large polynomials in s and x_1. This two variable system is found to be unsatisfiable in 135 seconds by `cad2d`, a special version of QEPCADB that is optimized for 2D CAD construction. For technical reasons[2], we must add the constraint $res_{x_1}(P, Q) = 0$ to the system, otherwise `cad2d` doesn't restrict its lifting to the points at which $res_{x_1}(P, Q)$ vanishes, as it obviously should. Essentially, the 2-variable system is solved by: 1) isolating roots of the resultants, 2) throwing away those that fall outside s-interval $(0, 1)$, 3) for each remaining root α constructing interval polynomials containing $P(\alpha, x_1)$ and $Q(\alpha, x_1)$, 4) isolating the roots of these interval polynomials, and 5) verifying that the isolating intervals for the two polynomials are disjoint. The interval Descartes root isolation method described in [6], for example, is able to do this quite quickly.

Figure 3 shows the final rewrite graph for this problem. It consists of 20 OR-nodes and 41 Q-nodes containing 1841 conjunctions. In this case the graph is a tree, it represents 12,191 distinct rewritings of the input. The tree is less than half the size if we don't use the "linear S-polynomial" rewriting operator, and the "best" rewriting is unchanged. This gives some indication that ramping up the number of operators available is likely to make a heuristic approach to

[2]Equational constraints are not implemented in `cad2d`, which is why the propagated constraint $res_{x_1}(P, Q)$ must be added to the formula explicitly by hand.

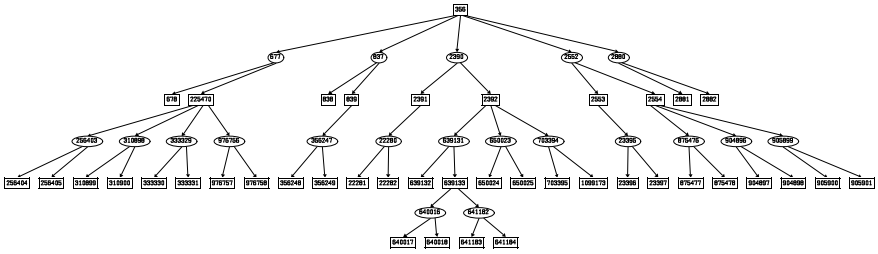

Figure 3: Rewrite graph resulting from searching the full space of rewritings for the Joswig–Witte problem.

limiting the search more necessary. It's also worth pointing out, that much of the work of the search went into exploring cases that were eventually found to be unsatisfiable, and those portions of the graph were then removed.

6.3 The Wang–Xia Problem

In [13], Wang & Xia analyze a system of five equations in five variables, with inequality side constraints, and a single parameter, v. The system describes the equilibrium points of a biological model. The system was "solved" in the following sense: the positive v axis was decomposed into intervals, inside of which the number of solutions to the system was constant as v varied. The approach was based on computing "border polynomials" [16]. No timing information was given.

The system they considered is the following:

$$\left[\begin{array}{l} 150000vz_3 + 750vz_3x - 599999x + 200 = 0 \\ \wedge \\ 625y_1^2 + 750000y_3 + 625y_3y_1 + 19200xy_1 - 8xy_1^2 - 8xy_1y_3 - 900000000 = 0 \\ \wedge \\ -11520000x + 9600xy_1 + 8xy_1y_3 + 8xy_3^2 + 1500000y_3 - 625y_3y_1 - 625y_3^2 = 0 \\ \wedge \\ 250z_1^2 + 75000z_3 + 250z_3z_1 + 1800y_3z_1 - 3y_3z_1^2 - 3y_3z_1z_3 - 22500000 = 0 \\ \wedge \\ -270000y_3 + 900y_3z_1 + 3y_3z_1z_3 + 3y_3z_3^2 + 150000z_3 - 250z_3z_1 - 250z_3^2 = 0 \\ \wedge \\ x \geq 0 \wedge y_1 \geq 0 \wedge y_3 \geq 0 \wedge z_1 \geq 0 \wedge z_3 \geq 0 \wedge 1200 - y_1 - y_3 \geq 0 \wedge 300 - z_1 - z_3 \geq 0 \end{array}\right]$$

They were interested in the solutions to this system in the parameter v. By keeping track of the operators applied during search, `qfr` is actually able to produce solutions to the original problem from solutions to the simplified problem, even when its rewriting steps eliminate quantifiers — at least for the set of rewrite operators considered here. Quantifying all variables but v in the above

system, `qfr` searches the whole space in 64 seconds, producing as its "best" formula

$$
\left[\begin{array}{l}
y_3 > 0 \wedge z_3 > 0 \\
\wedge \\
(750vz_3 - 599999)(66750vz_3y_3 - 14999911y_3 + 57600000vz_3 + 76800) < 0 \\
\wedge \\
27z_3^3y_3^3 + 8100z_3^2y_3^3 - 2430000z_3y_3^3 - 729000000y_3^3 \\
-2250z_3^3y_3^2 + 675000z_3^2y_3^2 + 405000000z_3y_3^2 \\
-187500z_3^3y_3 + 56250000z_3^2y_3 + 33750000000z_3y_3 \\
+15625000z_3^3 - 14062500000z_3^2 = 0 \\
\wedge \\
3z_3^2y_3 - 270000y_3 - 250z_3^2 + 150000z_3 < 0 \\
\wedge \\
3z_3y_3 + 900y_3 - 250z_3 > 0 \\
\wedge \\
(66750vz_3y_3^2 - 14999911y_3^2 - 45000000vz_3y_3 \\
+35999940000y_3 - 69120000000vz_3 - 92160000) \\
(66750vz_3y_3 - 14999911y_3 + 57600000vz_3 + 76800) < 0 \\
\wedge \\
43441734375000v^3z_3^3y_3^3 + 2753611266937500v^2z_3^2y_3^3 \\
-781874265726831075 0vz_3y_3^3 + 1124989575004895102973y_3^3 \\
+41616956250000000v^3z_3^3y_3^2 \\
-15960208532175000000v^2z_3^2y_3^2 \\
+19507457439221957100000vz_3y_3^2 \\
-40499739900283739013524 00y_3^2 \\
-7835616000000000000v^3z_3^3y_3 \\
+7257286575360000000000v^2z_3^2y_3 + \\
10368019353182100480000000vz_3y_3 \\
+13824012902214266880000y_3 - 6370099200000000000000v^3z_3^3 \\
-2548039680000000000v^2z_3^2 \\
-339738624000000000vz_3 - 150994944000000 = 0
\end{array}\right]
$$

This result was obtained by making the following substitutions:

$$
\begin{aligned}
x &= \frac{625y_3(-2400+y_3+y_1)}{8(y_3+1200)(y_3+y_1-1200)} \\
z_1 &= -\frac{-270000y_3-250z_3^2+150000z_3+3z_3^2y_3}{3z_3y_3+900y_3-250z_3} \\
y_1 &= -\frac{66750vz_3y_3^2-14999911y_3^2-45000000vz_3y_3+35999940000y_3-69120000000vz_3-92160000}{66750vz_3y_3-14999911y_3+57600000vz_3+76800}
\end{aligned}
$$

It is important to note that during this search process `qfr` deduced that the original system implies that the denominators in each of these substitution expressions are non-zero.

QEPCADB can be used to "solve" the system in the same sense as Wang–Xia's solution: namely that the v-axis is decomposed into open intervals in which the number of real solutions is constant (In fact, on each interval the solutions are

defined by a finite set of real-valued functions of the interval.), and the system is solved at a sample value for v from each interval. This requires less than 90 seconds.

To understand the value of the rewriting, we first note that QEPCADB is unable to solve the original formulation of the problem — even when given the `measure-zero-error` option, which corresponds to Wang–Xia's ignoring finitely many values of v. Redlog 3.1 gives an "Arithmetic exception" error and fails — using `rlgqe`, the "generic" quantifier elimination option. Mathematica fails to give an answer after more than 4 hours — although, in fairness, we are unaware of any way to indicate to it that finitely many v values may be ignored. QEPCADB, as mentioned, can solve the rewritten system, and Mathematica doubtless could as well if there were a way to tell it to solve the problem "generically", i.e. not to lift over section cells in v-space.

To demonstrate that finding this rewriting is not trivial, we consider asking Redlog and Mathematica to eliminate $\{y_1, z_1, x\}$ from the original input. Both require about one second to compute a result. Redlog eliminates x and z_1, but not y_1, and returns a formula consisting of 50729 characters (not counting whitespace). Mathematica eliminates all three variables, returning a formula consisting of 26683 characters. The formula returned by `qfr` consists of 875 characters.

6.4 Computing possible branch cuts

This example stems from the approach developed in [2, 1] for simplification of expressions containing inverse elementary functions. The question is to determine potential branch cuts for the function

$$\sqrt{\sqrt{p} - \sqrt{q}}$$

where $p, q \in \mathbb{C}$. "Potential branch cuts" consist of the union of the cuts for $\sqrt{p}$ and $\sqrt{q}$ along with the points in pq-space that get mapped by $\sqrt{p} - \sqrt{q}$ to the negative real axis, or zero. The later set is characterized by the following formula, which represents a complex number as the ordered pair given by its real and imaginary parts. Note: variable u represents $\sqrt{p}$, and v represents $\sqrt{q}$.

$$\exists R_u, I_u, R_v, I_v \left[\begin{array}{l} R_p = R_u^2 - I_u^2 \wedge I_p = 2R_uI_u \wedge [R_u > 0 \vee R_u = 0 \wedge I_u >= 0] \\ \wedge \\ R_q = R_v^2 - I_v^2 \wedge I_q = 2R_vI_v \wedge [R_v > 0 \vee R_v = 0 \wedge I_v >= 0] \\ \wedge \\ I_u - I_v = 0 \wedge R_u - R_v <= 0 \end{array} \right]$$

After 3 seconds, `qfr` completes the search of the entire space of rewritings. The rewrite graph consists of two OR-nodes and six Q-nodes containing 185

conjunctions. The "best" rewriting is given by the disjunction of four quantified formulas:

$$
\begin{array}{l}
\exists R_u, R_v \left[\begin{array}{l} R_u > 0 \wedge R_v > 0 \wedge R_v - R_u >= 0 \wedge I_p = 0 \wedge I_q = 0 \wedge \\ R_v^2 - R_q = 0 \wedge R_u^2 - R_p = 0 \end{array} \right] \\
\vee \\
\exists I_v \left[\begin{array}{l} (I_v)(I_p) > 0 \wedge (I_v)(I_q) > 0 \wedge (I_v)(I_q - I_p) >= 0 \wedge \\ I_q^2 - 4I_v^2 R_q - 4I_v^4 = 0 \wedge 4I_v^4 + 4R_p I_v^2 - I_p^2 = 0 \end{array} \right] \\
\vee \\
\exists I_v [I_v > 0 \wedge I_p = 0 \wedge I_q > 0 \wedge I_q^2 - 4I_v^2 R_q - 4I_v^4 = 0 \wedge I_v^2 + R_p = 0] \\
\vee \\
\exists I_v \left[\begin{array}{l} I_v > 0 \wedge I_p = 0 \wedge I_q = 0 \wedge I_q^2 - 4I_v^2 R_q - 4I_v^4 = 0 \wedge \\ 4I_v^4 + 4R_p I_v^2 - I_p^2 = 0 \wedge I_q - I_p = 0 \end{array} \right]
\end{array}
$$

At first blush, this might not seem like much of an improvement, but QEPCADB solves each piece of this extremely quickly, providing a simple solution; Mathematica quickly gives an answer answer that's about half the size of what it produces from the original input; and Redlog quickly produces a formula consisting of two simple quantifier-free pieces and two simple pieces with one quantified variable, as opposed to the formula consisting of six pieces, each with one quantified variable.

6.5 The REMIS-Patterson problem: poor performance

When `qfr`'s rewrite operators simply don't apply, for example formulas without equalities, the whole approached described here has nothing to offer. On the other hand, it also takes no time to run it and discover that fact. Poor performance is when the program runs for a long time without discovering interesting reformulations of the problem. In this paper, we are not so much aiming at finding good formulations quickly, as at exploring the entire space of reformulations. It is fully expected that the incorporation of some heuristics to rule out unpromising regions of the search space would drastically reduce the search time, without significantly degrading the quality of rewritings discovered. However, without such heuristics, we must regard a problem for which the time to search the whole space is "unreasonably large" as providing an example on which the algorithm performs poorly.

The REMIS data-base[3] includes many real quantifier elimination problems. One of these is the "Patterson Problem" [8]. The existential formulation of this

[3] `http://www.algebra.fim.uni-passau.de/~redlog/remis/`

problem is:

$$\exists x_1, x_2, x_3, x_4 \left[\begin{array}{l} (y - u_3)x_2 + (-x + u_2)x_1 - u_2 y + u_3 x = 0 \wedge 2u_1 x_2 - u_1^2 = 0 \\ \wedge \\ yx_4 + (-x + u_1)x_3 - u_1 y = 0 \wedge 2u_2 x_4 + 2u_3 x_3 - u_3^2 - u_2^2 = 0 \\ \wedge \\ (u_1 u_3 x_2 + u_1 u_2 x_1)x_4 + (-u_1 u_2 x_2 + u_1 u_3 x_1)x_3 \\ +(-u_1 u_3^2 - u_1 u_2^2)x_1 \neq 0 \end{array} \right].$$

This formula is in some sense set up for `qfr` to perform poorly. It models a geometric configuration, but leaves out all the non-degeneracy conditions — this was purposely done to demonstrate how Redlog's "generic" quantifier elimination discovers non-degeneracy conditions for itself. If we add those conditions ($u_1 u_2 - u_2 x - u_3 y \neq 0 \wedge u_2 - x \neq 0 \wedge y \neq 0$), `qfr` performs acceptably. It searches the entire space of rewritings in 27 seconds, producing a search tree with 23 OR-nodes and 1176 conjunctions distributed across 27 Q-nodes.

However, without the non-degeneracy conditions, `qfr` takes about 17 minutes to complete its search — spending all that extra time exploring portions of the space of rewritings that correspond to cases that are simply not of interest. Moreover, because the only operator `qfr` has for eliminating quantified variables applies only to equations, it returns a large formula in which many disjuncts have quantified variables that can be eliminated trivially — for example:

$$\exists x_3 [x - u_2 \neq 0 \wedge u_1 - x = 0 \wedge y = 0 \wedge u_3 = 0 \wedge u_2 \neq 0 \wedge x \neq 0 \wedge x_3 \neq 0]$$

For humans, "$\exists x_3 [x_3 \neq 0]$" is pretty easily recognized as true! Since the basic approach of this work has been to ensure that the program can at least do the things that people can easily do, we should add a rewrite operator that does quantifier elimination for subformulas of the form $\exists x [ax + b \ \sigma \ 0]$ even when σ is not "=".

7 Conclusions and Future Work

This paper paper considers the problem of finding rewritings of quantified input formulas that make good inputs to quantifier elimination programs. The essential motivation for this is the observation that quantifier elimination algorithms do not do a good job of exploiting problem structure, and that by rephrasing QE problems, experts are often able to make more effective use of QE software. We have presented a data structure, the "rewrite graph", and an algorithm based on that data structure that explores the space of rewritings of the input formula induced by a set of rewrite operators. We have shown that even for some non-trivial problems, the method is able to explore the entire space of rewritings and discover good rewritings.

There are many directions for future work — first and foremost is to investigate heuristic methods for guiding the search for rewritings. Ultimately we would like to find good answers without having to search the entire space of rewritings. A heuristically guided approach is likely to become increasingly necessary as the number of rewrite operators available to the system grows.

Another direction for work concerns the "next step" — i.e. the passing on to one or more QE systems of pieces of the reformulated problem. It may be better not to keep the search for rewritings and the use of QE algorithms separate. Instead, as `qfr` recognizes that a node contains a quantified formula that would be particularly easy for an available system, it could simply call the system directly on that node. The result of the QE — particularly if that node is found to be unsatisfiable — could have a big impact on the subsequent search.

References

[1] Beaumont, J., Bradford, R., Davenport, J., and Phisanbut, N. Adherence is Better Than Adjacency. In *Proceedings ISSAC 2005* (2005), M. Kauers, Ed., pp. 37–44.

[2] Bradford, R., and Davenport, J. H. Towards better simplification of elementary functions. In *Proc. of the 2002 international symposium on Symbolic and algebraic computation* (New York, NY, USA, 2002), ACM Press, pp. 16–22.

[3] Brown, C. W., and Gross, C. Efficient preprocessing methods for quantifier elimination. In *CASC* (2006), pp. 89–100.

[4] Caviness, B., and Johnson, J. R., Eds. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and Monographs in Symbolic Computation. Springer-Verlag, 1998.

[5] Collins, G. E. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Lecture Notes In Computer Science* (1975), vol. Vol. 33, Springer-Verlag, Berlin, pp. 134–183. Reprinted in [4].

[6] Collins, G. E., Johnson, J. R., and Krandick, W. Interval arithmetic in cylindrical algebraic decomposition. *Journal of Symbolic Computation 34*, 2 (2002), 145–157.

[7] Dolzmann, A., and Sturm, T. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation 24*, 2 (Aug. 1997), 209–231. Special Issue on Applications of Quantifier Elimination.

[8] Dolzmann, A., Sturm, T., and Weispfenning, V. A new approach for automatic theorem proving in real geometry. *Journal of Automated Reasoning 21*, 3 (1998), 357–380.

[9] Heintz, J., Roy, M., and Solernó, P. Sur la complexité du principe de Tarski-Seidenberg. *Bull. Soc. Math. France 118* (1990), 101–126.

[10] Joswig, M., and Witte, N. Products of foldable triangulations. to appear in Adv. Math., 2005.

[11] Tarski, A. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, 1951. second ed., rev. Reprinted in [4].

[12] van den Driessche, P., and Watmough, J. Reproduction numbers and sub-threshold endemic equilibria for compartmental models of disease transmission. *Mathematical Biosciences 180* (2002), 29–48.

[13] Wang, D., and Xia, B. Stability analysis of biological systems with real solution classification. In *Proc. International Symposium on Symbolic and Algebraic Computation* (2005), pp. 354–361.

[14] Weispfenning, V. Quantifier elimination for real algebra — the quadratic case and beyond. *AAECC 8* (1997), 85–101.

[15] Weispfenning, V. A new approach to quantifier elimination for real algebra. In *Quantifier Elimination and Cylindrical Algebraic Decomposition* (1998), B. Caviness and J. Johnson, Eds., Texts and Monographs in Symbolic Computation, Springer-Verlag, Vienna.

[16] Yang, L., and Xia, B. Real solution classifications of semi-algebraic systems. In *Algorithmic Algebra and Logic — Proceedings of the A3L 2005* (2005), A. Dolzmann, A. Seidl, and T. Sturm, Eds., Herstellung und Verlag, Norderstedt, pp. 281–289.

8 Appendix

Lemma 1 *Let $G = (V, E)$ be a directed acyclic graph. Let u and v be vertices such that there is no path from v to u. If S is the set of all vertices along paths from v to u, then the graph G' obtained from G by contracting $V' \cup \{u, v\}$ into a single new vertex x (cutting out edges from x back into x) is a DAG.*

Proof. Suppose G' has a cycle $w_1, \ldots, w_k$. The cycle must contain vertex x since, otherwise, the same vertices would form a cycle in G. Let $w_i = x$. Thus, there exist vertices $a, b \in V' \cup \{u, v\}$ such that $w_1, \ldots, w_{i-1}, a$ and and

$b, w_{i+1}, \ldots, w_k, w_1$ are paths in G. Let P_b be the path from u to b in G, and let P_a be the path from a to v in G. Then $P_b \to w_{i+1}, \ldots, w_k, w_1, w_2, \ldots, w_{i-1} \to P_a$ is a path from u to v in G. Thus, $w_1 = w_2 = \cdots = w_k = x$, i.e. the cycle is an edge from x back into x, which is a contradiction. $\square$