A Non-Trivial Group-Theoretic Algebraic Manipulation Engine Toolkit Thing

Kyle Miller kmill@mit.edu
Scott Kovach kovach@mit.edu
Sagar Indurkhya indurks@mit.edu

8 May 2009

Abstract

We developed a framework to allow for deep group-theoretical manipulations. This paper will describe the design ideas and capabilities of our system.

1 Introduction

Group theoretical manipulations are very difficult. The orders of groups which are commonly useful are quite large. E_8, a simple Lie group of dimension 248, has nearly seven-hundred million elements. The Rubik's cube has about forty-three quintillion permutations. Needless to say, directly working with this many elements by hand is time-consuming.

Instead, a symbolic system to interact with groups is immediately useful. Others have developed similar systems such as the Magma computer algebra system by the Computational Algebra Group.

Our system can currently solve certain classes of equations on groups, and, in a limited manner, handle other parts of abstract algebra such as fields.

2 Groups

A group is a structure G composed of a set of elements and a binary operation between two elements in the set [1]. We say an element is in a group if it contained in the set of elements. The binary operation is usually denoted by $\circ : G \times G \to G$.

A group must satisfy four properties:

- Closure. If we have $a, b \in G$ then $a \circ b \in G$.
- Associativity. Given elements $a, b, c \in G$, then $a \circ (b \circ c) = (a \circ b) \circ c$.
- Identity. There is an element $e \in G$ such that $e \circ a = a \circ e = a$ for all $a \in G$.
- Inverse. Every element $a \in G$ has an inverse denoted by a^{-1} which is also in G such that $a \circ a^{-1} = a^{-1} \circ a = e$, where e is the group identity.

A group must have at least one element, and the set of elements may be infinitely large.

2.1 Examples

An example of a group is the set of isometric symmetries of a square which leave a square in exactly same position and orientation, such as reflections about the axes and 90° rotations. The symmetries may be composed, and each of these compositions are also in this group. The group of symmetries on an n-gon is known as the dihedral group D_n.

Another example of a group is the permutations of the ordered tuple $(1, \ldots, n)$. The group operation $a \circ b$ of tuples a and b is permuting the elements of a by the permutation b. For example, $(1, 3, 2) \circ (3, 1, 2) = (2, 1, 3)$. Note that $(1, \ldots, n)$ is the group identity. The group whose elements are all $n!$ permutations of $(1, \ldots, n)$ is known as the symmetric group S_n. Note that permutations will be written in one-line notation throughout this paper, and the computer representations of permutations are 0-indexed instead of 1-indexed for computational convenience.

Addition on the integers modulo n also constitutes a group known as the cyclic group C_n. The defining characteristic of a cyclic group is that there is an element $a \in C_n$ such that any $b \in C_n$ can be denoted by a a^i for some i, where exponentiation means repeatedly applying the group operation. This a is called

the group generator. For the addition modulo n example, 1 generates the group.

3 Group Interface

The abstract interface for a group is fairly straightforward. The following must be available for any group:

- `(group-operation` G`)` returns a procedure which takes two elements $a, b \in G$ and returns $a \circ b$.
- `(group-elements` G`)` returns a list of all of the elements in G. This is not guaranteed to complete, especially in the case of infinite G.
- `(is-element?` G e`)` is a predicate which expresses the validity of e being an element of G.

An interesting modification to examine in the future is making `group-elements` return a stream.

3.1 Finite Groups

One kind of group representation using this interface is a group with finitely many elements, all of which are known. The constructor for such groups takes a list of elements and a corresponding group operation.

The implementation of the interface for finite groups is a matter of directly using the information given to the constructor.

3.2 Generator Groups

A group may have infinite elements, and storing all of the elements of such a group is impractical. Another representation is storing the generators of the group, those elements which, when combined using the group operator, can yield all of the elements in the group. For instance, the integers can be generated using the element 1 and the operator $+$ (assuming we know the inverse to addition is negation, otherwise 1 and -1 suffice as generators).

However, this representation is also useful for such groups as the permutations of a Rubik's cube. Although there are finite permutations, it may not make sense to actually store all such permutations in memory to work with the group.

4 Tagging Information

It is useful to tag groups with hints. For instance, it may be computationally prohibitive to brute-force the inverse operation for the group of integers under addition which was mentioned in section 3.2. The inverse operation, then, may be tagged to the group.

Tagging inverses is among the more useful applications for tagging in the current system. Group cosets are also tagged.

The interface to tags is essentially the same as to hash tables, since tags are basically key/value pairs.

Weak hash tables are used to ensure tags are not stored if the object which they reference is garbage collected.

5 Threaded Solvers

Group-theoretical manipulations do not always have a simple means of solution, and, without considerable effort, it may also be difficult to determine which method among a set of methods is the most efficient.

The generic dispatch system `ghelper.scm` is partly able to do the selection of a solution method. However, the limitation with the generic dispatch system is that predicates must be able to guarantee returning a value. However, there are cases in which it is hard to determine whether or not a predicate will even halt, such as `is-finite?` on a generator group. And, `is-finite?` would necessarily be generic itself.

To remedy this, our development was to evaluate all known methods in parallel in the hope that at least one method will ultimately yield the correct answer by a concerted effort of uninspired decision-making, albeit with some loss in efficiency.

A threaded solver T is a set of solution procedures. A solution procedure $t_i \in T$ is a procedure which takes the same arguments as T and either returns the value of T or, if it is unable to determine a value, it returns the special element $\varnothing$. All t_i which are able to return a value must return the same value, otherwise it is an error.

When T is executed, each of its procedures are executed in parallel as threads, and when one of the threads t_i returns a non-$\varnothing$ value, all of the threads are immediately stopped, and the value returned by t_i is the return value of T.

5.1 Solution Pipes

Obviously, there must be some mechanism to link together a thread which is requesting the result of a threaded solver and each of the threads within the threaded solver. This mechanism is a device like a pipe.

The solution pipe can be read from and written to by multiple processes. Only one valid answer may be written to a solution pipe, but a solution pipe may accept multiple $\varnothing$ elements. When a threaded solver is executed, a solution pipe is created. The solution pipe is notified of each of the new threads, and the outputs of the threads are attached to the input of the solution pipe. The threaded solver then waits on the output of the solution pipe.

Then, the pipe keeps track of all threads which have returned a value. If the threads all return $\varnothing$, then the threads which are waiting on the result of the solution thread are notified that no solution was found. However, if a thread returns a valid value, all of the threads piped into the solution thread are notified to stop.

5.2 Nested Threaded Solvers

The solvers in a threaded solver can in turn execute and wait on more threaded solvers. And, in turn, these too can execute and wait on more threaded solvers. Handling timesharing is relatively straightforward. In systems which require starting a special environment to begin multitasking (such as `conspire.scm`), the addition of a procedure such as `(maybe-with-time-sharing-conspiracy` *`thunk`*`)` will suffice, which only initializes a new time sharing environment if no other running time sharing environment currently exists.

However, an optimization can be made if two or more threads each want to run a threaded solver with the same arguments (by equality). By storing each solution pipe in a dictionary, indexed by both the threaded solver and the arguments, existing solution pipes may be retrieved and attached to threads requesting the same threaded solver with the same arguments.

Since this dictionary is storing each of the solutions pipes, we are essentially memoizing the results of the threaded solvers while ensuring each threaded solver with given arguments is only ever executed once. For space considerations, the dictionary may be a weak hash table so currently unused solution pipes may be garbage collected if necessary.

It was mentioned earlier that solver threads in a threaded solver may return $\varnothing$. There is a possibility that all of the threads in a threaded solver may return $\varnothing$, and in this case, the solution pipe sends a signal to the threaded solver which owned the threads that no solution was found. In the case of a single threaded solver, the solver simply stops the computation by reporting an error to the REPL.

However, in the case of nested solvers, the error does not need to stop the computation and destroy the threaded solver environment. For instance, let us have threaded solvers $T_1 = \{u_1, u_2\}$ and $T_2 = \{v_1\}$. Let the thread u_1 require the evaluation of T_2, the thread u_2 require significant time to execute, and the thread v_1 be guaranteed to return $\varnothing$. Then, computing $T_1(a)$ for some argument a will initialize a threaded solver with threads $u_1(a)$ and $u_2(a)$. The thread $u_2(a)$ will in turn initialize and wait on the threaded solver $T_2(b)$, with some other argument b, which then executes the thread $v_1(b)$. Now, we have threads $u_2(a)$ and $v_1(b)$ actively running with $u_1(a)$ waiting on the completion of $v_1(b)$. The thread $v_1(b)$ returns $\varnothing$, so the solution pipe connecting $v_1(b)$ to $T_2(b)$ sends a signal to $T_2(b)$ that no more solutions exist.

At this point, the entire computation could be stopped because we have a subproblem which has no solution. However, a more intelligent action is to make $u_2(a)$ send $\varnothing$ to its connect solution pipe. Thus, the solution pipe which T_1 is waiting on is sent $\varnothing$, and it notes that there is still one thread which able to return a value. And, after some time, $u_1(a)$ returns, and the solution pipe gives the value to $T_1(a)$, which then returns the value returned by $u_1(a)$.

Note that a call to $T_1(a)$ at this point will immediately return the value of $u_1(a)$ without starting a threaded solver because there is a solution pipe stored in the dictionary.

If $u_1(a)$ were to instead also return $\varnothing$, then, because the REPL is not waiting on any other threaded solvers, an error is raised.

If multiple threaded solvers are waiting on the same subproblem, the shared solution pipe retrieved from the dictionary blocks the threaded solvers until the computation is completed. However, if the computation fails with $\varnothing$, then each of the waiting threaded solvers must be notified. Thus, solution pipes also record which threads are awaiting a return value. And, so these threads are stopped correctly, the threads are forced to return $\varnothing$ before they are stopped. To do this, the thread record in `conspire.scm` was modified to also store the solution pipe waiting on the return value of a thread.

5.3 Generic Threaded Solvers

To use the threaded solver engine, there is an interface similar to `ghelper.scm`. The three important

parts of the interface are:

- `(make-threaded-solver` *`after-procedure arity`*`)` creates an object which is a procedure of *arity* arguments which, when evaluated, either binds to an existing solution pipe or creates a new solution pipe and starts attached solution threads. The procedure *after-procedure* is a procedure of *arity* + 1 arguments which takes the return value and the given arguments, which can be used to memoize the return value elsewhere, for example.
- `(defsolver` *`threaded-solver solution-thread`*`)` attaches a solution thread *solution-thread* of the correct arguments to the threaded solver *threaded-solver*.
- `unknown` is a unique element which represents the return value ∅. It is unique by `eq?`.

5.4 Example

The following is an example of the nested threaded solver described in section 5.2.

```
(define t1 (make-threaded-solver #f 1))

(defsolver t1
  (lambda (a)
    (display "u1")
    (let lp ((i 1000000))
      (if (= i a)
          #t
          (lp (- i 1))))))
(defsolver t1
  (lambda (a)
    (display "u2")
    (t2 (- a 1))))

(define t2 (make-threaded-solver #f 1))

(defsolver t2
  (lambda (b)
    (display "v1")
    unknown))
```

The `#f` in `make-threaded-solver` represents that we do not wish to do anything special with the return value of the threaded solver.

We could use this argument for `t1` if, for example, we wish to know what is being returned by the threaded solver for debugging purposes,

```
(define t1
  (make-threaded-solver
    (lambda (ret a)
      (printf "t1 returned ~a given ~a"
              ret a))
    1))
```

Although contrived, these examples illustrate the general mechanism of a threaded solver.

5.5 Real Example

A real example of the use of threaded solvers is to check the group axioms for a group.

```
(define group?
  (make-threaded-solver #f 1))

(defsolver group?
  (lambda (g)
    (if (cyclic? g)
        #t
        unknown)))

(defsolver group?
  (lambda (g)
    (and (group-closed? g)
         (group-associativity? g)
         (not (eq? no-inverse
                   (group-identity g)))
         (group-invertible? g))))
```

If a group G has the property of being cyclic, then we know G is a group without further checking. Otherwise, we need to check the axioms themselves. And, we aren't quite sure procedure will finish first with a non-`unknown` value.

The procedure `cyclic?`, too, is a threaded solver. It checks to see if G either was generated with `(make-cyclic-group` *`n`*`)` or there is an element $a \in G$ such that for all $b \in G$, $b = a^i$ for some i.

5.6 Difficulties

There are still some unsolved difficulties and directions for improvement with the threaded solver system.

For instance, it is not trivial to denote the equivalence of procedures. And so, if two threads simultaneously return equivalent procedures, the solution pipe is unable to determine the equivalence, and therefore

raises an error. Currently, the system presumes procedures returned by threads are equivalent without further checking.

An example of a place which requires this behavior is in the derivation of the group inverse operation.

```
(define group-inverse-operation
  (make-threaded-solver #f 1))

(defsolver group-inverse-operation
  (lambda (g)
    (get-tag g 'inverse unknown)))

(defsolver group-inverse-operation
  (lambda (g)
    (let ((inverses
           (map (lambda (a)
                  (group-inverse g a))
                (group-elements g))))
      (lambda (a)
        (let lp ((elts (group-elements g))
                 (inv inverses))
          (cond ((null? elts) no-inverse)
                ((equal? a (car elts))
                 (car inv))
                (else (lp (cdr elts)
                          (cdr inv)))))))))
```

In the first case, the group may have been tagged with an inverse operation. A valid example of this is to tag the the infinite group of integers under addition with the inverse `(lambda (a) (- a))`. However, if a group has not been tagged, we must brute-force the group inverse operation and construct such a procedure.

Alternatively, `group-inverse-operation` could return a procedure which requests group inverses via `group-inverse`. However, the system then may return such a function even if the inverse tag on the group exists. Although, in this case there is no problem because `group-inverse` can explicitly call `get-tag` to see if a group inverse has been tagged.

6 Solving Equations

A simple equation to solve is $a = b \circ c$ where $a, c \in G$ are known and we want the element b. Since every element has an inverse, $b = a \circ c^{-1}$, so to solve the equation we can compute `((group-operation g) a (group-inverse g c))`.

A fancier manipulation, which also takes advantage of the framework, is the problem of solving an equation of the form $b = c_n \circ \ldots \circ c_1 \circ a$ for unknown elements $c_i \in G$, where $a, b \in G$. The c_i are taken from a given set of allowable elements.

There are two motivations for being able to solve equations of this form. The first is the Chicken Nugget problem, and the second is solving the Rubik's cube.

6.1 Chicken Nugget Problem

Stated simply, the problem is that we wish to purchase n chicken nuggets, and we are only allowed to order chicken nuggets in boxes of 5 and 8. First, how many of each kind of box must we order? Second, what is the minimum number of nuggets such that for any greater number of chicken nuggets we may guarantee being able to purchase exactly that many nuggets?

We can look at this problem as finding $c_i \in 5, 8$ such that G is the group of integers under addition. Since the system is not yet smart enough in handling infinite groups, however, we will instead use a cyclic group with of much greater order than the number of nuggets we wish to purchase. Although a series of c_i will be found, if their integer sum exceeds that of the number of nuggets we wish to purchase, then the solution will be thrown out.

Using the equation solver, we can write the following:

```
(define g (make-cyclic-group 1000))
(define (solve-chicken-nuggets n)
  (solve-to-element g n '(5 8) 0))
```

where `n` is the number of nuggets we wish to purchase. The third line represents solving the equation $n \equiv 5a + 8b \pmod{1000}$ for a and b. The return value is a list of 5 and 8 which sum to `n` if a solution actually exists, otherwise the sum will greatly exceed `n` (but still be equivalent modulo 1000).

6.2 Rubik's Cube

The Rubik's cube is a permutation group with well-defined operations such as the rotations of the faces. These rotations represent the group operation of permutation.

And, because all relevant permutations of the cube are reachable by face rotations, the rotations are generators of the Rubik's group.

We will be working with a $2 \times 2 \times 2$ cube because such cubes are simpler to work with. We are assum-

ing one of the cubies remains stationary throughout manipulation.

The group can be represented in the system by a generator group with

```
(define r (make-generator-group
            rubiks-moves apply-perm))
```

The moves defined in `rubiks-moves` are the basic face rotation permutations, each of which is given one of the names `f fi l li u ui`. The permutations themselves are not illuminating to include here.

This representation is useful because we do not actually require knowing all of the elements of the group to solve the equation.

Thus, we can do such manipulations as

```
(map rubiks-move->name
     (solve-to-element
       r rubiks-identity
       rubiks-moves (rubiks-randomized)))

;Value 11: (l f ui l fi u fi l fi u)
```

to solve a cube.

Or, to understand the solver better,

```
(map rubiks-move->name
     (solve-to-element
       r rubiks-identity
       rubiks-moves
       (apply-perm
         (rubiks-name->move 'Ui)
         (rubiks-name->move 'F))))

;Value 16: (fi u)
```

6.3 The Solver

The solver a breadth-first graph searcher. The vertices of the graph are elements of the group, and edges represent left-multiplying elements of the group. Starting from the vertex a, we try edges c_i until we end up at b. The running time is roughly exponential with respect to the length of the solution sequence.

However, if we happen to know the group inverse (as in, the group inverse operation has been tagged to the group), then there is a trick to quicken the search. Two simultaneous breadth-first searches are conducted from both a and b, left-multiplying c_i from a and right-multiplying c_i^{-1} from b. When the searches meet in the middle, we have a solution. While still roughly exponential, the running time is now with respect to half the length of the final solution sequence.

This double-sided search is necessary to handle solving the Rubik's cube. The maximum distance between any two permutations on the $2 \times 2 \times 2$ cube is 14 moves. Only requiring searching a depth of 7 is much more efficient.

The solver is also useful for determining `is-element?` on generator groups. One course of action taken by this threaded solver is to try to find the elements of the group by `generate-group`. However, the group may be infinite, and so if the solver can solve the equation $a = c_1 \circ \ldots \circ c_k$ for c_i in the set of generators, then a is in the group. There is currently no way to determine whether an element is not in a group of infinitely many elements.

```
(define g (make-generator-group '(1) +))
(is-element? g 5)
;Value: #t
```

7 Group Algorithms

For group-theoretic manipulations, we include a number of basic group algorithms to compute various structures related to groups, such as subgroups, conjugacy classes, and alternative representations, as the Todd-Coxeter algorithm allows. In more difficult problems related to classifying groups and computing groups given partial information, these are often useful.

7.1 Basic Algorithms

Our toolkit includes basic algorithms for building up new groups from old ones and computing certain elementary group structures.

We implement product group and group generation from generators as abstractly as possible. The product algorithm takes two groups and creates elements formed from every pairing of one element from each group. It then associates with this a new group operation, which applies the old group operations separately to each component. This can be iterated arbitrarily:

```
(product-group
 (make-cyclic-group 2)
 (make-cyclic-group 2))

; Value 1:
#(group ((1 1) (1 0) (0 1) (0 0))
```

```
 #[compound-procedure 16])

(product-group
 (make-cyclic-group 2)
 (make-cyclic-group 2)
 (make-cyclic-group 2))

; Value 2:
#(group ((1 (0 0)) (1 (0 1)) (1 (1 0))
        (1 (1 1)) (0 (0 0)) (0 (0 1))
        (0 (1 0)) (0 (1 1)))
 #[compound-procedure 18])
```

It's also interesting to test group predicates on higher-level groups from the outputs of these procedures.

```
(cyclic? (product-group
           (make-symmetric-group 2)
           (make-cyclic-group 2)))
;Value: #f

(cyclic? (product-group
           (make-symmetric-group 2)
           (make-cyclic-group 3)))
;Value: #t
```

The engine can also be given a group in the form of generators. If all the group elements are explicitly needed, the (`generate-group` *`generators operation`*) procedure is used. This function performs a breadth-first search to visit all vertices of the corresponding Cayley graph. An optimization is made to use combinations of generators as they are found as generators to visit all vertices more quickly.

The procedure (`group-elements` G) for G which are generator groups will use `generate-group` to find the elements. If this procedure is able to return the elements before other threads in the threaded solver, the group data structure is modified and the elements are stored in the structure so that future requests for elements do not need to re-generate the group:

```
(define g (make-generator-group
           '((1 2 3 0) (1 0 2 3))
           apply-perm))
; Value 5:
#(generator-group ((1 2 3 0) (1 0 2 3))
 #perm)

(group-elements g)
; Value 6:
((0 3 2 1) (2 0 3 1) (3 2 0 1) (1 0 3 2)
                ...
 (2 3 1 0) (3 0 2 1) (0 2 1 3) (0 1 3 2)
 (3 2 1 0) (2 1 0 3) (0 3 1 2) (2 0 1 3))

g
; Value 7:
#(group
 ((0 3 2 1) (2 0 3 1) (3 2 0 1) (1 0 3 2)
                ...
  (2 3 1 0) (3 0 2 1) (0 2 1 3) (0 1 3 2)
  (3 2 1 0) (2 1 0 3) (0 3 1 2) (2 0 1 3))
 #perm)
```

where `#perm` stands in place of the cryptic `compound-procedure` representation to make clear that the operation of the group is `apply-perm`.

Other basic operations include calculating centralizers, conjugacy classes, and subgroups:

```
(define s3 (make-symmetric-group 3))
;Value 31: s3

(centralizer '(1 2 0) s3)
;Value 32:
#(group ((0 1 2) (1 2 0) (2 0 1)) #perm)

(conjugacy-class '(1 0 2) s3)
;Value 30: ((0 2 1) (2 1 0) (1 0 2))

(pp (subgroups s3))
(#(group ((0 1 2)) #perm)
#(group ((0 1 2) (0 2 1)) #perm)
#(group ((0 1 2) (1 0 2)) #perm)
#(group ((0 1 2) (1 2 0) (2 0 1)) #perm)
#(group ((0 1 2) (2 1 0)) #perm)
#(group ((0 1 2) (0 2 1) (1 0 2)
         (1 2 0) (2 0 1) (2 1 0)) #perm))
;Unspecified return value
```

In addition to these, many more operations could be added as needed and built up from previous ones.

7.2 Quotient Groups

Another fundamental way to construct new groups from old is through the computation of quotient groups. Given a group and a normal subgroup, our procedure first constructs the cosets. This procedure multiplies the first element of the subgroup by each element of the group.

Since cosets partition the group, each time a new element is found in this way, we have a new coset, and the algorithm then multiplies the entirety of the

subgroup by this element and remembers them in a hash table. This runs efficiently.

The quotient group procedure then performs a similar computation; it multiplies the first element of each coset by each element of the group to see how the cosets are permuted. This permutation is determined by tags assigned to the group by the coset procedure. It finally takes the list of permutations and returns it as a permutation group:

```
(quotient s3 (generate-group '((1 2 0))
                             apply-perm))
;Value 35: #(group ((1 0) (0 1)) #perm)
```

This computes S_3/C_3, since C_3 is a normal subgroup of the symmetric group S_3.

7.3 Todd-Coxeter Algorithm

The most powerful group-theoretic algorithm included in the toolkit thing is the Todd-Coxeter algorithm for coset enumeration. We use it as a convenient way for a user to input many types of groups and a method of generating groups, potentially in search methods.

A fundamental way to represent a group is through generators and relations. The group is presented as a set of products that equal the identity. A cyclic group, for instance, has one generator, with the relation $x^n = 1$. Every group can be written in this way, and it often provides a simple, intuitive way to describe groups. The tetrahedral group, describing the symmetries of a tetrahedron, as another example, has relations $x^3 = y^2 = z^2 = xyz = 1$.

Although this is notationally compact, such a representation is computationally difficult. The Todd-Coxeter algorithm, however, is a simple and efficient algorithm for converting a set of relations into a permutation representation of the group, if it is finite.

We provide the algorithm with a set of relations and the generators of an arbitrary subgroup. It then numbers the cosets of this subgroup and analyzes how the generators permute them. This gives a set of permutations which represent the generators and can be used to generate the entire group. The representation thus obtained is the permutation representation induced by the subgroup; passing in the trivial subgroup yields the regular representation, which is faithful.

The algorithm keeps track of three sets of tables: the coset table, the relation tables, and the subgroup tables. The coset table keeps track of the final answer; it has a row for each coset, and a column for each generator and its inverse. The entries c_{ij} is the action of the generator j on the coset i. The table for a relation has a column for each term in the product; for instance, the relation x^3 would have three x columns. It has a row for each coset, which contains the orbit of that coset under the action of the columns. It contains the additional information that the final coset must equal the initial, since the product of all the elements of the relation is the identity. Finally, there is a subgroup table for each generator of the subgroup. It is similar to the relation table, but it contains only one row for the coset equal to the subgroup. The action of a subgroup generator on the subgroup is trivial, so this gives additional information.

Figure 1 is an example using the dihedral group D_3. We will use the subgroup $\langle y \rangle = C_2$. For simple subgroups, the subgroup table is unnecessary, and in this case we will omit it along with the coset table.

Thus, $y = (2\ 3)$ and $x = (1\ 2\ 3)$, written in cyclic notation.

Note that we only need three cosets to completely fill the tables, which follows from the fact that the index of C_2 in D_3 is three. The algorithm simply adds new cosets and fills in as much of the table as possible before adding more.

x	x	x	
1	2	3	1
2	3	1	2
3	1	2	3

y	y
1	1
2	3
3	2

y	x	y^{-1}	x	
1	1	2	3	1
2	3	1	1	2
3	2	3	2	3

Figure 1: Relations: $x^3 = y^2 = yxy^{-1}x = 1$

7.4 Example

Using an extension to make generator relation input simple for a person, an example of executing the algorithm is

```
(tc '("x^3" "y^2" "yxy^-1x") '("y"))
;Value 37: ((0 2 1) (1 2 0))
```

The first permutation is y: it sends 0 to 0, 1 to 2, and 2 to 1. The next is x: it sends 0 to 1, 1 to 2, and 2 to 0.

8 Many-dimensional Iteration

This part of the project (which falls under the long-term basic research sub-team of our group) is referred to as the multi-dimensional-iterator (MDI). Imagine that we want to find a group with i elements and j maximal order. Then one way to do it is to simply iterate through all the possible values of i and j respectively in such a manner that for any pair (i, j), the program examines that pair after a finite number of computations. Such a search is traditionally done using the following circular walk through the first quadrant:

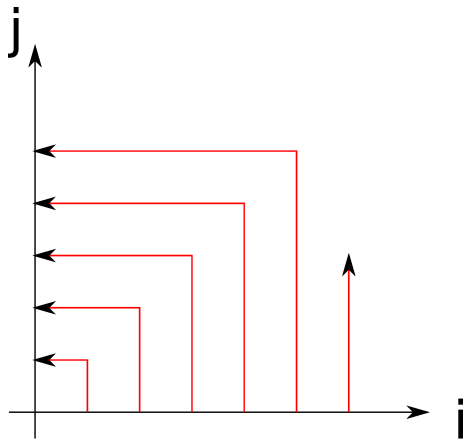

Figure 2: A basic two dimensional iteration

Of course in real problems we have constraints to limit the space we are searching. Going with the example above, the order of the group i must be greater than or equal to the maximal order of the group ($i \geq j$). Now imagine that the user of our program would like to search some n dimensional space, and supplies us with a series of predicates that tell us which areas to avoid. Now suppose that n is large, say $10,000$, but the predicates prune out the vast majority of the search space. Then, our circular walking approach fails because it is effectively going to be doing a depth-first-search requiring back-tracking. Efficient back-tracking for DFS requires memoization, and we clearly cannot afford to memoize in a high dimensional space. We can imagine scenarios where perhaps the space is a giant maze and this type of approach becomes computationally intractable very quickly. As illustrated in figure 3, we instead grow a boundary starting at the origin that grows outward wrapping around predicate regions (dark gray in the figure). In fact we use two boundaries, a new boundary (black) and a old boundary (light gray). The old boundary prevents the new boundary from examining locations it has already examined, and each iteration the previous new boundary becomes the current old boundary.

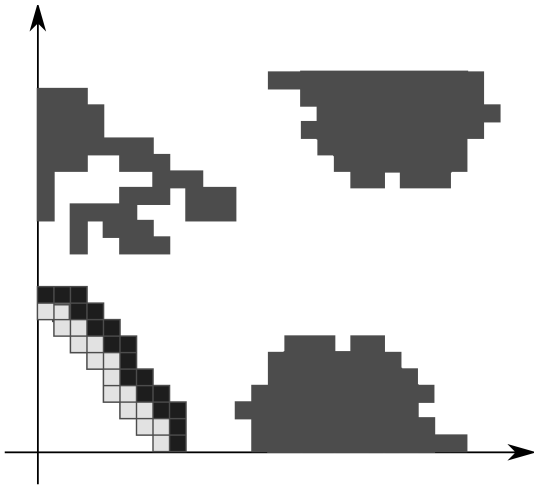

Figure 3: Boundary expansion with predicate regions

Suppose now that $n = 10,000$. Then any point in space has $3^{10,000} - 1$ neighbors (assuming in each dimension we can move either forwards, backwards, or stay put), a number that is computationally intractable under almost any circumstances. Thus almost every traditional data-structure-oriented operation we can imagine has to be carefully re-examined, since we cannot at any point afford to enumerate non-trivial subsets of the neighbors of a point in space without running out of memory or never finishing our computation.

And so we study the following two problems: given a point in space, if we have a series of predicates, each of which takes as input some subset of the position vector, how can we generate and store all of the different valid neighbors to which we can walk. Specifically, we are interested in the fact that we can use one predicate's abilities to prune the set of valid neighbors to reduce the number of neighboring positions another predicate must examine. Secondly, we are interested in solving a network-dynamic-programming problem, that is, we would like to share the computations that a point does with it's neighbors since a non-trivial number of their

neighborhoods overlap. In this paper we will spend a lot of time developing the first problem before coming to the unfortunate conclusion that we are simply trying to solve 3-SAT in disguise (dooming us to failure since 3-SAT is NP-Complete and it is likely that $P \neq NP$). We will primarily be using the Trie data structure created by Fredkin.

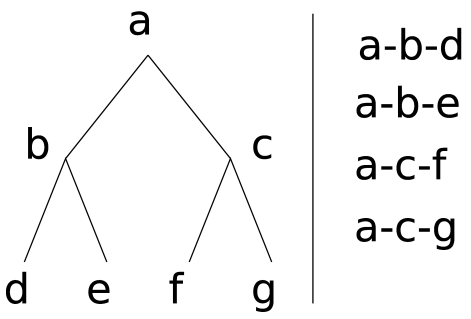

Figure 4: A trie structure storing four sequences

Initially our point may look something like the following:

$$\left\langle \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} a \\ b \\ c \end{bmatrix} \right\rangle$$

where $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ may be expressed using the ambivalence operator as `(amb a b c)`.

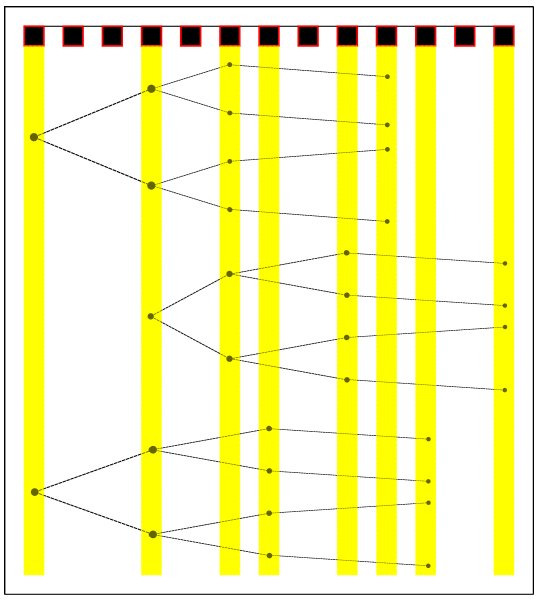

Figure 5: Fusing multiple trie structures together

We are given a set of predicates $P = \{p_1, \ldots, p_n\}$, and what we will call predicate-targets. We represent the valid strings of variables that each predicate accepts by a trie-structure. For example, in figure 5 we have three predicates. Variables are denoted by the solid black blocks. Note that some of the variables are used by more than one predicate. Now, the key operation is to fuse these three trie-structures together into a larger trie-structure. First however, we introduce the *canteen* data structure. The canteen data-structure is effectively a pair, with the first element being a list of variables that a trie-structure is covering, and the second being the trie-structure. Our code base has the following major functions described briefly. While they may seem not-too-difficult to code, note that we must code every one of them to be nearly algorithmically optimal or the code simply will not work.

The procedure `(Create-Trie list-of-points)` creates a Trie data structure given a list of points. For example in figure 4, the strings on the right would fed in, and a trie-structure like the one on the left would be output.

The procedure `(Create-Canteen variables list-of-values-for-each-variable)` creates a canteen.

In the procedure `(get-sub-canteen desired-vars canteen)`, you basically specify a subset of variables (`desired-vars`) and it extracts from the provided canteen a new canteen that has a trie structure which only covers the desired variables. This procedure is really hard to code in a manner that doesn't result in combinatorical explosions.

The procedure `(blockerize alpha)` takes an `alpha` of the form:

```
((a baz1) (a baz2) (a baz3) (b baz4) (b baz5))
```

into

```
((a (baz1 baz2 baz3)) (b (baz4 baz5)))
```

`(block-matcher blocksA blocksB operator)` is a procedure which, given two lists of blocks, finds all matching blocks (by header) and applies `operator` on them, returning the list of results of the operator calls.

The procedure `(block-multiply blockA blockB varsA varsB)` multiplies all the elements of `blockA` with all the elements of `blockB`.

`(Join-Canteens canteenA canteenB predicateA)` joins two canteens together across their common

variables in an algorithmically near-optimal manner. Note that `canteenA` is initially all the possible values that can be fed into `predicateA`, but `canteenB` represents all the possible values that returned true for `predicateB`, and thus `canteenB` is used to prune the values of `canteenA` before they are fed into `predicateA`.

The actual procedure used to prune the points is `(prune-points init-list predicate-pairs)`. The end result is algorithmic sub-optimality because there are too many combinatorical explosions preventing better than exponential performance. We realized that this is just a really bad way to solve Circuit Satisfiability. Note, however, that this is trying to solve 3-SAT without knowing what the predicates actually are (as in, it treats the predicates like blackboxes, whereas 3-SAT knows exactly what the predicates are).

Here are some examples of the system:

```
(define (pred1 x y z)
  (<= (+ x y z) 5))

(define (pred2 x y z)
  (<= (+ (* x x) (* y y) (* z z)) 10))

(define myInitList
  '((1 2 3 4) (1 2 3 4) (1 2 3 4)))

(pp (prune-points
      myInitList
      (list (cons (list 0 1 2)
                  (list pred1)))))
((0 1 2)
 ((1 ((1 ((1)
          (2)
          (3)))
      (2 ((1)
          (2)))
      (3 ((1)))))
  (2 ((1 ((1)
          (2)))
      (2 ((1)))))))
```

The interpretation of the result is that the first entry are the variables represented, which are `x`, `y`, and `z` in this case. The second part is the trie structure. This particular trie can be expanded out to $\{111, 112, 113, 121, 122, 131, 211, 212, 221\}$

Here is another example, testing two predicates, both operating on the same set of variables.

```
(pp (prune-points
      myInitList
      (list (cons (list 0 1 2)
                  (list pred1))
            (cons (list 0 1 2)
                  (list pred2)))))
((0 1 2)
 ((1 ((1 ((1)
          (2)))
      (2 ((1)
          (2)))))
  (2 ((1 ((1)
          (2)))))))
```

A second example is testing two predicates as before, but they now only share two variable in common.

```
(define myInitList2
'((1 2 3 4)
          (1 2 3 4)
          (1 2 3 4)
          (1 2 3 4)))

(pp (prune-points
      myInitList2
      (list (cons (list 0 1 2)
                  (list pred1))
            (cons (list 1 2 3)
                  (list pred2)))))
((1 2 3 4)
 ((1 ((1 ((1 ((1)
              (2)))
          (2 ((1)
              (2)))))
      (2 ((1 ((1)
              (2)))))))
  (2 ((1 ((1 ((1)
              (2)))
          (2 ((1)
              (2)))))
      (2 ((1 ((1)
              (2)))))))
  (3 ((1 ((1 ((1)
              (2)))))))))
```

Something interesting to note is that this is not exactly circuit-satisfiability since the prune points predicates given are treated like black boxes, whereas in circuit-satisfiability the predicates are transparent. However, as we saw in the second example, it is not necessary that the traditional ordering is the most compact way to store trie structures after they are

fused together. However, it is not clear how to do the reordering in less than exponential time.

The second part of the system, solving the network dynamic programming, eluded us for the following reason: suppose we have point A with a neighbor point B. Suppose we now prune the valid neighbors of A, and would like to share this information with B. However, as we saw above, it is easy to concisely represent all of B's points using the `amb` operator. However, once we start using just trie data structures, what used to take just a few words in memory now requires a huge amount of space. Thus, it is clear that some type of combinatorical data structure is required for representing the set of valid neighbors for a point in space.

Primarily, since Scheme doesn't give unrestricted access to pointers, it is difficult to do high-performance programming, or even very tricky programming in which data structures have to be manipulated very carefully and very precisely with almost no excess memory requirements allowed. Another problem with this part of the project that we realized much later is that the number of dimensions is not dynamic, which makes it difficult to encode a great number of problems.

Once we solve the second part of the project (the network-dynamic-programming problem), we would be able to solve much more complex analogues to the following types of problems: Suppose we have a group G such that $6 \leq o(G) \leq 10$ with elements $a_1, \ldots, a_k$ for $1 \leq k \leq 10$ such that $a_1^{\gamma_0} = a_2^{\gamma_1} = ... = a_k^{\gamma_k} = 1$. Then find all valid sets of values for $\gamma_0, \gamma_1, ..., \gamma_k$.

Encoded in predicates,

```
(define (is-group-valid?
          . list-of-gamma-values)
  (define (get-TC-format
            remaining-gamma-vals
            cur-variable-index)
    (if (null? remaining-gamma-vals)
        '()
        (cons
          (make-list
            (car remaining-gamma-vals)
            cur-variable-index)
          (get-TC-format
            (cdr remaining-gamma-vals)
            (+ 1 cur-variable-index)))))
  ; using todd-coxeter which fails
  ; via time-out (and then returns #f)
  (todd-cox (get-TC-format
              list-of-gamma-values
              0)))

; order should be >= than the number of
; non-zero gamma-vals and the gamma-values
; should be sorted (descending) so we don't
; solve the same problem twice.
(define (valid-gamma-values?
          order gamma-vals)
  (and (>= order
       (reduce (lambda (x y)
                 (if (> y 0)
                     (+ x 1)
                     x))
               0 gamma-vals)
       (< 0
          (reduce (lambda (x y)
                    (list
                      y
                      (* (cadr x)
                         (if (> (car x) y)
                             0
                             y))))
                  (list (car gamma-vals) 1)
                  gamma-vals)))))
(let ((starting-point
       '(1 1 1 1 1 1 1 1 1 1))
      (pred-pairs
       (list (list
              '(1 2 3 4 5 6 7 8 9 10)
              is-group-valid?)
             (list
              '(0 1 2 3 4 5 6 7 8 9 10)
              valid-gamma-values?))))
  (MDI starting-point pred-pairs))
```

which would return all sets of values of the $o(G)$ and γ_i that satisfy our problem in a reasonably efficient manner.

9 Extensions

The group system can be used to implement higher-order objects which depend on groups and to implement other interesting procedures.

9.1 Fields

Our system has limited support of fields. A field F is a set of elements with two associated binary operators which we will denote by $+$ and $\cdot$. The $+$ operator forms an abelian group (a group with the additional property of commutativity) with the elements in F,

and the $\cdot$ operator forms an abelian group with the elements in F less the identity of the $+$ group. Also, the distributive law of $\cdot$ over $+$ must hold.

Thus, we can create a predicate to check a field:

```
(defsolver field?
  (lambda (f)
    (and (abelian? (field-add-group f))
         (abelian? (field-prod-group f))
         (distributive? f))))
```

The procedures `field-add-group` and `field-prod-group` return the groups associated with $+$ and $\cdot$, respectively.

9.2 Group Naming

An unfinished system is the naming of a group by to which of C_n, S_n, D_n, etc. the group is isomorphic.

It can currently identify cyclic groups. It is hoped that the procedure `find-generators` will be able to aid in identification. This procedure finds the minimal set of elements required to generate a given group,

```
(name-group (make-cyclic-group 4))
;Value 11: (cyclic 4)

(name-group (make-generator-group
              '((1 2 3 0))
              apply-perm))
;Value 12: (cyclic 4)

(name-group (make-generator-group
              '((2 3 0 1))
              apply-perm))
;Value 13: (cyclic 2)
```

It is not yet obvious how to handle being able to name a group multiple things. For instance, a group of two elements may be isomorphic to both S_2 and C_2.

References

[1] Rowland, T., and Weisstein, E. W. *Group.* MathWorld, 2009.