Biased Tadpoles: a Fast Algorithm for Centralizers in Large Matrix Groups

Daniel Kunkle and Gene Cooperman
College of Computer and Information Science
Northeastern University
Boston, MA, USA
kunkle@ccs.neu.edu, gene@ccs.neu.edu

ABSTRACT

Centralizers are an important tool in in computational group theory. Yet for large matrix groups, they tend to be slow. We demonstrate a $O(\sqrt{|G|}\,(1/\log\varepsilon))$ black box randomized algorithm that produces a centralizer using space logarithmic in the order of the centralizer, even for typical matrix groups of order 10^{20}. An optimized version of this algorithm (larger space and no longer black box) typically runs in seconds for groups of order 10^{15} and minutes for groups of order 10^{20}. Further, the algorithm trivially parallelizes, and so linear speedup is achieved in an experiment on a computer with four CPU cores. The novelty lies in the use of a biased tadpole, which delivers an order of magnitude speedup as compared to the classical tadpole algorithm. The biased tadpole also allows a test for membership in a conjugacy class in a fraction of a second. Finally, the same methodology quickly finds the order of a matrix group via a vector stabilizer. This allows one to eliminate the already small possibility of error in the randomized centralizer algorithm.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic algorithms

General Terms: Algorithms, Experimentation

Keywords: centralizer, matrix groups, tadpole, conjugator, group order

1. INTRODUCTION

The tadpole algorithm in computational group theory is reminiscent of the Pollard rho algorithm [15] for integer factorization. Indeed, both the "tadpole" and the "rho" are meant to remind us geometrically of the shape of a certain trajectory. Tadpoles were originally invented by Richard Parker [14]. Our implementation will use appropriate biasing heuristics to create an algorithm for centralizers in large matrix groups. We also apply biased tadpoles to the problems of group order and conjugacy testing.

One defines the *conjugate* of a group element g by an element h as $g^h \stackrel{\text{def}}{=} h^{-1}gh$. This is analogous to the change of basis formula in linear algebra. Given a group G, the *centralizer subgroup of* $g \in G$ is defined as $\mathcal{C}_G(g) \stackrel{\text{def}}{=} \{h\colon h \in G, g^h = g\}$.

Note that the above definition of centralizer is well-defined independently of the representation of G. Even for permutation group representations, it is not known whether the centralizer problem can be solved in polynomial time. Nevertheless, in the permutation group case, Leon produced a *partition backtrack algorithm* [8, 9] that is highly efficient in practice both for centralizer problems and certain other problems also not known to be in polynomial time. Theißen produced a particularly efficient implementation of partition backtrack for GAP [17].

In the case of matrix groups, centralizers are well known to be computationally difficult for large groups. In this paper, we demonstrate a randomized centralizer algorithm. The algorithm can be thought of as a randomized Schreier-Sims algorithm under the conjugate action. We also make the assumption that a distribution of uniformly random group elements is available. However, the conjugate action implies that the group acts on a permutation domain of the same size as the group. For large groups, this makes computation of a Schreier generator computationally expensive.

The problem of computing a Schreier generator in the conjugate action reduces to the following problem:

PROBLEM 1.1 (CONJUGATOR). *Given two elements of a group G, g and h, find an element $r \in G$ such that $g^r = h$.*

A traditional method for computing a conjugator between g and h is by bidirectional search. However, this can require a lot of storage, and the use of such an out-of-CPU-cache algorithm makes it slow using today's technology. This is because of the large gap in speed between CPU and RAM. Further, as CPUs continue to gain additional cores, an in-cache algorithm can execute a different conjugator in each CPU core, unlike a bidirectional search algorithm.

The tadpole algorithm has the advantage of requiring space that is only constant in the size of a group element; i.e. storage proportional to a small constant times the memory required to store one group element. Further, a simple probability argument shows that the expected number of steps required to solve the Conjugator problem is $O(\sqrt{|G|}\log|G|)$ (under certain randomness assumptions). Further, for k such conjugacy tests with $k \geq |G|$, the amortized cost of one Conjugator is only $O(\sqrt{|G|})$. (See Section 4.2.) The obvious centralizer algorithm derived from this (Section 3) then operates in time $O(\sqrt{|G|}\,\log(1/\varepsilon)))$ with probability of error bounded above by ε, $0 < \varepsilon < 1$. (see Section 3.3).

ISSAC'09, July 28–31, 2009, Seoul, Republic of Korea.

The idea of a biased tadpole is to create a modified tadpole that, loosely speaking, acts on a smaller space than the original tadpole. In Section 6.3, we show a 9.6 times speedup by using biasing to compute the centralizer of an involution in an example for the group J_4.

The same tadpole idea can be used to more efficiently find a vector stabilizer subgroup of a matrix group.

PROBLEM 1.2 (VECTOR STABILIZER). *If a matrix group, G, acts on a vector space V, an element $g \in G$* stabilizes *a vector $v \in V$ if the product of v and the matrix representation of g is equal to v itself. A* vector stabilizer subgroup *of G is a subgroup H such that $vh = v$ for every $h \in H$.*

The rest of the paper has the following structure. First, we briefly conclude the introduction with related work. Section 2 describes the core idea of tadpoles walks. Section 3 presents our tadpole-based algorithm for the centralizer problem. Section 4 describes two additional tadpole-based algorithms: vector stabilizer and group order; and conjugacy testing. Section 5 describes the addition of biasing to tadpoles, and estimates the predicted speedup due to biasing. Finally, Section 6 provides a number of experimental results.

1.1 Related Work

Butler describes a backtracking algorithm for centralizers of matrix groups based on examining the action of the matrix group on vectors of the vector space [2]. Eugene Luks showed how to find centralizers in polynomial time for the special case of solvable matrix groups [11].

In the domain of permutation groups, Leon invented the partition backtrack algorithm [8] for centralizer and other permutation group problems not known to be in polynomial time. Theißen added an efficient implementation of this algorithm into GAP [17].

Luks described a polynomial time centralizer algorithm for matrix groups [11]. Bray describes a fast algorithm specialized for centralizers of an involution in certain groups [1]. Murray and O'Brien demonstrated how to use a random Schreier-Sims algorithm [7] for the matrix group viewed as a permutation group on vectors or subspaces [12].

This work assumes the ability to generate uniformly random group elements. Celler and Leedham-Green [3] developed the product replacement algorithm, which is extremely efficient, and widely used in GAP. Although the best theoretical guarantee of randomness to date requires that the algorithm use $O(n^9)$ group multiplications [13], a small number of group operations suffice in practice to produce elements that pass all current tests of randomness.

2. TADPOLE WALKS

The core idea of the tadpole algorithm [14] is to create a dynamical system on any orbit of a permutation group. This dynamical system yields a tadpole walk. A *tadpole walk* on a permutation domain and group generators is a construction in which one takes a pseudo-random, but deterministic function f (such as a hash function) that acts on the permutation domain and returns an element of the group. (In Richard Parker's original description, it returned one of the group generators.) One then views this as a dynamical system on the points of the permutation domain. Given a point x_0, one computes the new point $x_1 = x_0^{f(x_0)}$ (the application of the generator $f(x_0)$ to the point x_0). One iterates to produce a sequence $x_0, x_1, \ldots, x_j, \ldots, x_i$ for which $x_i = x_j$. Clearly, once one of the x_i is equal to an earlier x_j, the series will then continue cycle among the points x_j, x_{j+1}, ..., x_{i-1}. This cycle is sometimes called an *attractor cycle*.

It remains to detect when the x_i have arrived on the attractor cycle. This is done using *landmarks*. A landmark is a point x_i whose representation as a point in the permutation domain satisfies a chosen property. In our applications, a landmark is a state for which the k initial bits are all zero. The constant k can be tuned. It is assumed that the first k bits are pseudo-random. Where that is not true, alternatives are possible such as to hash the bits, or to use a related algorithm of Sedgewick and Szymanski [16].

Upon arriving at a state x_i that is a landmark, one looks it up in a hash table. If it is already in the hash table, then one has already made a circuit around the attractor cycle. Otherwise, one stores x_i in the hash array. One adjusts k to simultaneously ensure that the overhead of checking landmarks is small, and that at least one landmark appears on every attractor cycle.

It can easily be shown that the number of steps until a tadpole walk makes one revolution around an attractor cycle is $O(\sqrt{n})$, for n the size of the permutation domain on which it acts. The space required to store the landmarks is then $O(\sqrt{n}/2^k)$. A second critical property of tadpoles is that the expected number of attractor cycles is only $O(\log |n|)$, for n the size of the permutation domain on which the tadpole acts [5]. Hence, there are relatively few large tadpoles. This will be critical in Algorithm 1 (Centralizer).

One empirically observes further that in practice, the number of *large* attractor cycles tends to be less than five. While smaller attractor cycles may exist, our computations almost never encounter such smaller attractor cycles, and so they do not greatly affect the running time. (See Section 6.)

Tadpole Walks Using Constant Space.

The above method uses a hash table of landmark elements to detect the attractor cycle of the tadpole walk, which requires $O(\sqrt{n}/2^k)$ space. An alternate approach uses the *two-finger trick*, which reduces the space required to $O(1)$, at the cost of requiring about three times as many generator applications. This trick has a further advantage in that Algorithm 1 (centralizer) becomes a black box algorithm.

In this method, two *fingers* follow the tadpole walk. The first finger uses the standard definition of the walk. The second finger follows the walk at twice the speed, applying two generators for each one generator the first finger applies. The walk has found an attractor when the two fingers meet.

3. TADPOLE ALGORITHM FOR CENTRALIZER

Suppose we wish to find the centralizer of an element $g \in G$. We will do so by performing tadpole walks of group elements acting on group elements by conjugation.

Start with the element to be centralized. Conjugate it by a random group element $r_1 \in G$. Then, do a tadpole walk on g^{r_1} until some element is repeated, and an attractor cycle is found. We keep track of the word used to conjugate, starting with that random group element, and continuing with the generators chosen by the tadpole walk. If the tadpole walk produces a word w_1, then we have a path, $g^{r_1 w_1}$, that leads from g to a landmark on the attractor cycle.

We then repeat the process, using a different initial random group element for conjugation, $r_2 \in G$. If this second tadpole walk leads to the same attractor cycle, then we produce a second word $r_2 w_2$ leading to the same landmark. Hence $g^{r_1 w_1} = g^{r_2 w_2}$. This implies that $g^{r_1 w_1 w_2^{-1} r_2^{-1}} = g$, and therefore that $r_1 w_1 w_2^{-1} r_2^{-1}$ is in the centralizer of g.

Algorithm 1 Centralizer

Input: Group G, $g \in G$, pseudo-random function $f : G \mapsto G$
Output: random elements, RandomCentralizerElts, for $\mathcal{C}_G(g)$

1: Initialize a hash array of AttractorCycleReps to be empty.
2: Initialize RandomCentralizerElts to empty.
3: **repeat**
4: Initialize empty Landmark hash array, and LandmarkPtr list
5: Let r be a uniformly random element of G
6: Let $x = g^r$
7: [Begin tadpole walk at random conjugate of g]
8: **repeat**
9: Let $r = rf(x)$
10: Let $x = x^{f(x)}$
11: **if** x is a landmark **then**
12: If x is not in Landmark array, hash it, and append it to LandmarkPtr
13: **end if**
14: **until** x is a landmark and x is in Landmark array
15: Use LandmarkPtr list to determine all landmarks occurring at x or later in Landmark array
16: Let m be the lexically least such landmark
17: **if** m is in AttractorCycleReps **then**
18: Get the value w_m, and Add rw_m^{-1} to RandomCentralizerElts
19: **else**
20: Set $w_m = r$, and add the key-value pair (m, w_m) to AttractorCycleReps
21: **end if**
22: **until** stopping condition satisfied (see Section 3.3)
23: Return RandomCentralizerElts

This tadpole-based method for centralizer is described by Algorithm 1. See Section 2 for a description of the landmarks used in the algorithm.

If the hash array AttractorCycleReps in Algorithm 1 becomes too large, we can remove the less frequently occurring elements. However, [5, Appendix], shows the expected number of attractor cycles to be bounded above by $(1/2)\log_2 |G|$. Also, the algorithm can use a number of different stopping conditions, which are discussed further in Section 3.3.

In fact, it is not required that $g \in G$, in which case the above algorithm produces an external centralizer subgroup. More generally, although we identify g^r as defined by the conjugate action, one could take $g \in \Omega$ for any permutation domain Ω. The pseudo-random function f must then be defined as mapping Ω to G.

THEOREM 1. *If r_2 is a random group element of G, then $r_1 w_1 w_2^{-1} r_2^{-1}$ of the previous discussion is a random element of the centralizer subgroup $\mathcal{C}_G(g)$.*

PROOF. The group elements r_1 and w_1 are independent of the random element r_2. While the element w_2 depends on the choice of r_2, in fact, w_2 depends only on g^{r_2}. This is because w_2 is a tadpole walk with starting point g^{r_2}.

Let us temporarily fix r_2 and choose $r_2' = hr_2$, where h is chosen randomly from $\mathcal{C}_G(g)$. Note that $g^{r_2'} = g^{hr_2} = g^{r_2}$, since $h \in \mathcal{C}_G(g)$. So a tadpole walk starting at $g^{r_2'}$ depends on r_2, but is independent of the choice of h for $r_2' = hr_2 \in \mathcal{C}_G(g)r_2$. The random choice of r_2' implies a random choice of $h \in \mathcal{C}_G(g)$ such that $r_2' = hr_2$. Since $r_1 w_1 w_2^{-1} r_2^{-1} \in \mathcal{C}_G(g)$ and h is random in $\mathcal{C}_G(g)$, $r_1 w_1 w_2^{-1} r_2'^{-1}$ is a random element of $\mathcal{C}_G(g)$.

We have shown that for a fixed r_2, a random choice of $r_2' \in \mathcal{C}_G(g)r_2$ produces a random element of $\mathcal{C}_G(g)$ using Algorithm 1. Since the algorithm produces a random element of $\mathcal{C}_G(g)$, for a random choice of r_2' restricted to $\mathcal{C}_G(g)r_2$, and since this is true independently of the choice of $r_2 \in G$, it suffices to choose a random $r_2 \in G$, and then a random $r_2' \in \mathcal{C}_G(g)r_2$, to ensure $r_1 w_1 w_2 - 1 r_2'^{-1} \in \mathcal{C}_G(g)$. Since this produces a random $r_2' \in G$, it suffices to choose a random $r_2 \in G$ to ensure $r_1 w_1 w_2^{-1} r_2'^{-1} \in \mathcal{C}_G(g)$. Therefore, any random choice of r_2 produces a random element of $\mathcal{C}_G(g)$. □

3.1 Early Termination of Long Walks

As will be seen in Section 6, there can be several common attractor cycles with widely varying lengths. If a particular tadpole walk is excessively long, one gains evidence that one may be in a long attractor cycle. Under these circumstances, it is tempting to terminate the tadpole walk, and begin a new walk using a new random element that will hopefully lead to a shorter attractor cycle. Corollary 2 implies that as long as the decision to terminate is based solely on the number of steps or time passed, the resulting centralizers computed from completed tadpole walks will still be uniformly random. They will not be biased by such a decision procedure.

COROLLARY 2. *In producing the random elements of the centralizer subgroup according to Theorem 1, one may terminate a particular tadpole walk based on the number of steps executed so far, and any properties of previous tadpole walks. The early termination of a particular tadpole may remove one element of the centralizer subgroup, but it will not bias the uniform distribution of the other random elements of the centralizer that are generated.*

PROOF. The tadpole walk to be terminated has already produced an initial random starting position g^r for $r \in G$ random. If one fixes r and chooses $r' \in \mathcal{C}_G(g)r$ uniformly at random, then $g^{r'} = g^r$. If r produces a centralizer element $h \in \mathcal{C}_G(g)$, then r' produces an element $r'r^{-1}h \in \mathcal{C}_G(g)$. Since $r'r^{-1}$ is a uniformly random element of $\mathcal{C}_G(g)$ *independently of when the tadpole walk is terminated*, $r'r^{-1}h$ is also uniformly random in $\mathcal{C}_G(g)$ independently of when the tadpole walk is terminated. So, termination of the tadpole walk causes removal of this uniformly random element $r'r^{-1}h$, which does not bias the uniform distribution of the resulting centralizer elements produced by Algorithm 1. □

There are several potential heuristics for early termination of a tadpole walk. Such heuristics are not employed in our experiments, since the primary concern is evaluating the core algorithm. However, one such heuristic is to set a variable *cutoff* to be the median number of steps for all previous tadpole walks. The median means that half of the walks were longer and half were shorter. One then terminates any tadpole walk that has already used two times *cutoff* steps.

In computing the median, any terminated tadpole walk is considered to have taken an infinite number of steps.

3.2 Complexity: Number of Tadpoles Required

The following theorem is well known. It is based on the fact that in any subgroup chain, the order of a subgroup in its parent subgroup is at most 1/2. So, the longest possible chain is $\log_2 |G|$. It has been observed that in practice, one can usually generate G using many fewer group elements than would be indicated by $O(\log n)$. Elementary abelian 2-groups are a worst case, and many algorithms for those groups reduce to linear algebra. Most groups encountered in practice are far from this worst case.

THEOREM 3. *$O(\log |G| - \log \varepsilon)$ random elements of a group G suffice to generate G with probability of error bounded above by ε, $0 < \varepsilon < 1$.*

THEOREM 4. *For a group G, the expected number of tadpole walks needed to generate $\mathcal{C}_G(g)$ is $O(\log |\mathcal{C}_G(g)|)$.*

PROOF. Theorem 3 states that one needs $O(\log |\mathcal{C}_G(g)|)$ random centralizer elements. The Appendix of [5] shows that the expected number of attractors is $(1/2)\log_2 |\mathcal{C}_G(g)|$. Algorithm 1 produces one random centralizer element per tadpole walk, except those tadpole walks that are the first to reach a new attractor cycle. Since at most $(1/2)\log_2 |\mathcal{C}_G(g)|$ can reach a new attractor cycle, $O(\log |\mathcal{C}_G(g)|)$ tadpole walks suffice to generate G. □

A standard argument demonstrates the refined estimate of $O(\log |\mathcal{C}_G(g)| - \log \varepsilon)$ tadpole walks to generate $\mathcal{C}_G(g)$ with upper bound ε on probability of error.

Let $n < |G|$ be the size of the conjugacy class of the element g to be centralized. One can Combine this with the expected length of a tadpole walk, $\sqrt{n}$, and assume $c \log |\mathcal{C}_G(g)|$ tadpole walks for some fixed c chosen in advance. Noting that $n|\mathcal{C}_G(g)| = |G|$, it is clear that Algorithm 1 executes in $O(\sqrt{n} \log |\mathcal{C}_G(g)|) = O(\sqrt{|G|})$ multiplications, and it succeeds with probability of error at most $1/|\mathcal{C}_G(g)|^c$. The next section describes a stopping criterion independent of knowledge of $|\mathcal{C}_G(g)|$ or $|G|$.

3.3 Stopping Criteria

Algorithm 1 is mostly a black box algorithm, in that it does not depend on the representation of x, g, or r, except for purposes of landmarks. Even that restriction could be removed by using the cycle-finding algorithm of Sedgewick and Szymanski [16]. As with all such randomized black box algorithms, one must rely on a randomized stopping criterion. Let the maximum length of a chain of subgroups in G be L.

THEOREM 5. *If L is an upper bound on the length of a subgroup chain in $|\mathcal{C}_G(g)|$, and if the stopping criterion of Algorithm 1 is to stop after k centralizer elements are produced, for $k > L$, then the probability of error before producing the full centralizer is at most $1/2^{k-L}$.*

The proof is clear. Note that if the group order of the centralizer, $|\mathcal{C}_G(g)|$, or the order $|G|$, is known, then the number of prime factors of that order (including repeated prime factors) can be used as the upper bound L.

If $|\mathcal{C}_G(g)|$ and $|G|$ are not known a priori, then the next stopping criterion can be used. This yields the revised estimate $O(\sqrt{|G|} - \sqrt{n} \log \varepsilon)) = O(\sqrt{|G|}\,(1 - \log \varepsilon / |\mathcal{C}_G(g)|))$, or the coarser estimate of $O(\sqrt{|G|} \log(1/\varepsilon))$ for $0 < \varepsilon < 1$ specifying the desired upper bound on the error probability.

THEOREM 6. *If the stopping criterion of Algorithm 1 is to stop after k centralizer elements in a row fail to cause the candidate centralizer subgroup to grow, then the probability of error is at most $1/2^{k-1}$.*

PROOF. We will show the probability of error to be bounded above by $1/2^k + 1/4^k + 1/8^k + \cdots = 1/2^{k-1}$. To see this, assume that Algorithm 1 produces an infinite number of generators for the centralizer. Consider the last element that caused the candidate centralizer subgroup to grow. If the k immediately preceding steps had not increased the subgroup size, Algorithm 1 would have stopped before that element. The probability of that happening is at most $1/2^k$, since the subgroup size at that point was at most 1/2 the full group size. Then look at the second to last generator that caused the candidate centralizer subgroup to grow. If there were k immediately preceding steps that did not produce a new generator, then we would have stopped at this point. The probability of that is $1/4^k$. Continuing in the same way produced the series upper bound, which yields $1/2^{k-1}$. □

4. OTHER TADPOLE-BASED ALGORITHMS

Here, we present two additional tadpole-based algorithms, both of which are closely related to the algorithm for centralizer presented above. They are: computing vector stabilizers and group order; and conjugacy testing.

4.1 Vector Stabilizer and Group Order

Suppose we wish to compute the order of a matrix group G with generators $\langle g_1, g_2, \ldots, g_k \rangle = G$. We will do so using a method that makes use of tadpoles to compute vector stabilizers. The overall algorithm is closely related to the Schreier-Sims algorithm for permutation groups.

First, choose a vector x such that $xg \neq x$ for some generator $g \in [g_1, g_2, \ldots, g_k]$. Compute the transversal of G with respect to x. The transversal is defined as $\mathcal{T}_G(x) \stackrel{\text{def}}{=} \{y \colon g \in G, y = xg\}$. We compute the transversal by breadth-first search, beginning with the vector x, and enumerating all vectors reachable by some word in the generators of G.

Then, find a set of generators of the vector stabilizer subgroup of G with respect to x. The vector stabilizer subgroup is defined as $\mathcal{S}_G(x) \stackrel{\text{def}}{=} \{g \colon g \in G, xg = x\}$. To do this, we use a slightly modified version of Algorithm 1 (Centralizer). Instead of the domain consisting of elements of G and the action being conjugation, the domain consists of vectors and the action is vector-matrix multiplication.

The order of G is the product of the order of the transversal and the order of the vector stabilizer subgroup: $|G| = |\mathcal{T}_G(x)||\mathcal{S}_G(x)|$. If the vector stabilizer is trivial, we return the order of the transversal as the order of the group. Otherwise, we recursively compute the order of the vector stabilizer subgroup, using the elements of the vector stabilizer discovered using tadpoles as generators of the group.

4.2 Conjugacy Testing

We first consider the case in which one wants to do many conjugacy tests. In this limit, the time is $O(\sqrt{|G|})$ steps, the number of steps in a tadpole walk. Suppose we are given an element $g \in G$ of order k, and we wish to determine which of the conjugacy classes of order k that element belongs to.

First, we will complete a small number of tadpoles for random elements in each of the conjugacy classes of order k. This will provide us with a few *landmarks*, each associating a tadpole attractor cycle with a specific conjugacy class.

Then, we simply compute one more tadpole walk starting from element g, and look up which conjugacy class the attractor cycle landmark is associated with. As argued in Section 2, and seen empirically in the experimental results of Section 6, there are only a few large attractor cycles that are visited with high probability. So, we need only a small number of landmarks in each of the conjugacy classes to find a match with high probability. In the unlikely event that the landmark has not been seen yet, we can compute another tadpole starting at g^r, where r is a random element of G.

This method is particularly efficient when there are many elements for which one wants to perform a conjugacy test, as the initial cost of associating landmarks with each conjugacy class is amortized over the tadpole computations for each of the elements.

Next, we consider the case of a single conjugacy test. In this case, the number of steps is $O(\sqrt{|G|\log|G|})$. In this case, we consider two elements $g, h \in G$, and we wish to construct an element $r \in G$ such that $g^r = h$. The algorithm consists of forming $c\sqrt{|G|}$ random conjugates of g, where the constant c can be chosen as described at the end of the paragraph, and then executing a tadpole walk for each of these random conjugates. The tadpole walk terminates when the lexically least landmark of its attractor cycle has been identified. This produces at most $c\sqrt{|G|}$ distinct attractor cycles. One similarly finds $c\sqrt{|G|}$ random conjugates of h, and then executes a tadpole walk for each one. The probability of a tadpole walk from g and one from h reaching a common attractor cycle is then a constant. The constant c can be tuned to increase the probability of two tadpole walks from g and h meeting. This shows that the expected number of steps is $O(\sqrt{|G|\log|G|})$.

5. TADPOLE BIASING

By *biasing* a tadpole walk, we seek to constrain that walk to visiting elements in a subset of the permutation domain. As observed in Section 2, the expected length of a tadpole walk before a duplicate element is reached (i.e., before an attractor is found) is $O(\sqrt{n})$, where n is the number of elements in the permutation domain. Therefore, if the walk is constrained to a subset of the permutation domain of size n/k, the expected length of the walk is reduced to $O(\sqrt{n/k})$. If the additional time spent computing the bias function is not prohibitive, we have reduced the total time to finish a tadpole walk. We next demonstrate a method for adding biasing to the tadpoles used in Algorithm 1 (Centralizer).

5.1 Biasing Conjugation in Matrix Groups

First, we briefly describe a normal tadpole walk. We then describe the addition of biasing to decrease the expected length of the walk.

Standard Tadpole.

We are given generators of a group $\langle g_1, g_2, \ldots, g_k\rangle = G$ and a *start element* $x_0 \in G$. We are also given a hash function $g_i = f(h)$ that deterministically, and pseudo-randomly, maps any element $h \in G$ to a generator g_i of G.

The tadpole iteratively conjugates the current element in the walk by the generator specified by the hash function, until some element is repeated. So, one *step* in the walk is defined as

$$\begin{aligned} g &= f(x_i) \\ x_{i+1} &= g^{-1}x_i g \end{aligned}$$

The walk continues until some $x_i = x_j$, for $j < i$.

The time to compute one step is dominated by the two matrix multiplication operations required by the conjugation. For matrices of dimension n, this requires $2n^3$ operations.

Biasing Function.

To add a bias, we define a *prefix* function $[e_1, e_2, \ldots, e_m] = p(x, g)$, where $x \in G$, $g \in G$ is one of the generators, and $[e_1, e_2, \ldots, e_m]$ is a vector containing the first m elements of the matrix resulting from the conjugation $g^{-1}xg$. In other words, we are computing the first m out of n^2 entries of the result of one step.

This prefix computation can be done by two vector-matrix multiplication operations: the first multiplying a $1\times n$ vector of g^{-1} by the full $n\times n$ matrix x; the second multiplying the resulting $1\times n$ vector by a $n\times m$ portion of g. The resulting cost is $n^2 + nm$ operations. Typically, $m < n$, so a prefix computation is at least a factor of n less expensive than a full conjugation computation.

To implement the bias, we remove the use of the hash function and decide which generator to use by computing the prefix of $g^{-1}xg$ for all generators g and choose the one that yields the lexically least result. In doing so, the tadpole walk will visit those states with lower prefixes more often, thereby effectively reducing the size of the domain.

Note that the quality of the biasing depends in part on the number of generators. The larger the number of generators, the lower the expected value of the minimum prefix. So, in cases where very few generators are given, we expand the set of generators with additional elements.

5.2 Other Biasing Functions

In certain cases, there may be biasing functions even more effective than the prefix computation presented above.

For example, in the general linear group $\mathrm{GL}(n, q)$, one can construct an arbitrary change-of-basis matrix. By conjugating by such a change-of-basis matrix, one can imitate the standard algorithm for Gaussian elimination. Thus, one can extend the function $f(x)$ of Algorithm 1 to $g(x^{f(x)})$, where $g()$ maps its argument to a new matrix that is as close to diagonal form as possible. This produces large biasing, since the range of $g()$ will be relatively small.

There exist much stronger biasing functions. For example, a biased tadpole for conjugacy testing can converge in a single step if the biasing function returns the unique lexically least element in the conjugacy class. A less extreme and less expensive version of this approach is to return an element with a small base image under a lexical ordering. While we do not pause to describe such a heuristic in detail, this is clearly motivated by similar results in which lexically least elements can be found in other settings [4, 10, 6].

Specifically, Cooperman and Finkelstein [4, Section 5] show that for a subgroup $H < G$, one can assign a unique integer in $[1, \ldots, |G|/|H|]$ for each distinct coset Hg for $g \in G$. Linton [10] shows how to find a lexicographically least image of a set of points under the action of a permutation group. Hulpke and Linton [6] demonstrate a lexicographic ordering for subgroups of groups.

Table 1: Results of computing the order of twelve different matrix groups (* indicates that GAP exceeded its maximum of 4 GB of RAM without completing).

Group	Order	Dim	Size of 1^{st} Transversal	Chain Length	Our Time (s)	GAP's Time (s)
J_3	5.0×10^7	80	50232960	1	750.37	*
McL	9.0×10^8	22	22275	3	1.27	19.89
He	4.0×10^9	51	8330	4	1.25	1161.58
A_{14}	4.3×10^{10}	12	3003	8	2.17	0.50
$2^{1+8}.O_8^+(2)$	8.9×10^{10}	24	4147200	3	55.03	*
Ru	1.5×10^{11}	28	417600	3	5.66	*
Co_3	5.0×10^{11}	22	37950	6	1.82	10.90
Co_2	4.2×10^{13}	22	46575	5	2.03	1108.76
Fi_{22}	6.4×10^{13}	78	142155	6	3.98	*
$F_4(2)$	3.3×10^{15}	26	17821440	3	257.91	*
Co_1	4.2×10^{18}	24	8386560	4	124.63	*
$E_6(2)$	2.1×10^{23}	27	69193488	5	1219.85	*

These three algorithms, centered around lexicographic orderings, provide the basis for important heuristics for tadpoles acting: on cosets; on sets; and on subgroups. Finding lexicographic orderings on sets leads to biases that are useful for finding set stabilizers for permutation groups, while lexicographic orderings on on subgroups can lead to finding new heuristics for finding normalizer subgroups.

Last, note the following important implementation detail: If a biasing function uses a randomized algorithm as a subroutine, then the seed of the corresponding random number generator must be re-initialized to a fixed value at the beginning of the function. Otherwise, the biasing function would not be deterministic (but pseudo-random), as required.

6. EXPERIMENTAL RESULTS

All of the experimental results given below were produced using a computer with: two 2.00 GHz dual-core Intel Xeon CPUs; 16 GB of RAM; and Linux version 2.6.9. All of our algorithms were implemented in C and compiled with GCC version 3.4.5. All matrix representations and involution words in standard generators were taken from version 3.0 of the ATLAS of Finite Group Representations [18].

In cases where our algorithm requires the use of a random group element, we produce that element using a naive random walk of length at least 100 (and length 10000 for the larger groups). A more sophisticated generator, such as product replacement [3], would have produced the same timings or better.

For all biased tadpoles, we used 100 random group elements as generators and a prefix of size 8 (see Section 5.1 for the definition of prefix).

6.1 Group Order and Centralizer for Matrix Groups

Here, we provide a comparison of our tadpole-based methods for matrix groups to the corresponding algorithms implemented in GAP, including computing group order and the centralizer of an involution. We choose to focus on involutions because they are often useful in other algorithms, and because their centralizers are usually large. Although GAP's strategy appears to be a naive enumeration of all group elements, we compare with GAP as an implementation of the only other general matrix centralizer algorithm in the literature.

Tables 1 and 2 show experimental results for the following twelve groups: Janko group J_3; McLaughlin group McL; Held group He; Alternating group A_{14}; Conway groups Co_3, Co_2, and Co_1; $2^{1+8}.O_8^+(2)$; Rudvalis group Ru; Fischer group Fi_{22}; and untwisted groups of exceptional Lie type $F_4(2)$ and $E_6(2)$. Note that $2^{1+8}.O_8^+(2)$ appears as a subgroup of Co_1 and is of interest because it is far from simple.

When choosing a set of groups to test on, we restricted ourselves to groups with an available matrix representation over $GF(2)$ of dimension 128 or less. This is because our implementation of matrix operations was originally developed for J_4, which has a representation of dimension 112 over $GF(2)$ (so we represent each row of the matrix with two 64-bit words). Because we have optimized for this case, matrix operations of smaller dimension are suboptimal. In the future, we plan to generalize our implementation to yield maximum performance for arbitrary dimensions, possibly by incorporating the MeatAxe library.

6.1.1 Group Order

We tested our tadpole-based algorithm for computing group order on the twelve matrix groups listed in Table 1. The groups range in size from 5.0×10^7 to 2.1×10^{23}, with dimension ranging from 12 to 80.

Recall that the algorithm for computing group order (Section 4.1) works by recursively computing a series of vector stabilizers and transversals. Table 1 lists the size of the first (and largest) transversal, and the total number of transversals computed (i.e., the chain length).

Finally, the table gives the total time required by our algorithm and by the corresponding group order algorithm in GAP. In cases where GAP ran out of memory before completing (max 4 GB), no time is shown. Of the five out of twelve cases where GAP produced an answer, GAP was faster in one (A_{14}), and our algorithm was between 5 and 1000 times faster in the other four. With the exception of J_3 (which has a trivial vector stabilizer), our algorithm completes in seconds for groups of order 10^{13} and less, and in minutes for groups up to order 10^{23}.

Table 2: Results of computing the centralizer of an involution in twelve different matrix groups (* indicates that GAP exceeded its maximum of 4 GB of RAM without completing).

Group	Involution Word	Centralizer Size	Our Centralizer Time (s)	Our Order Time (s)	GAP Centralizer and Order Time (s)
J_3	a	1920	17.78	0.90	*
McL	a	40320	4.64	0.69	18.12
He	a	161280	7.11	1.17	1010.06
A_{14}	$(\mathrm{ba})^6$	46080	13.39	1.14	1.15
$2^{1+8}.O_8^+(2)$	a	49152	15.11	2.26	*
Ru	a	116480	7.78	1.16	*
Co_3	bb	2903040	8.05	1.32	11.86
Co_2	a	743178240	6.91	1.66	1011.21
Fi_{22}	a	18393661440	15.74	5.69	*
$F_4(2)$	a	754974720	20.49	66.84	*
Co_1	a	2012774400	86.56	43.20	*
$E_6(2)$	a	135291469824	1643.73	11.87	*

6.1.2 Centralizers of Involutions

Next, we tested our algorithm for computing centralizers using the same twelve groups. In each case, we chose to compute the centralizer of an *involution* (element of order 2). Table 2 shows the results of these computations. In that table, the chosen involutions are given as short words in the two generators listed for that group in version 3 of the ATLAS of Finite Group Representations [18]. The centralizers range in order from 1920 to approximately 1.35×10^{11}. In each case, we give the time for our algorithm in two pieces: the time to compute the centralizer (as a set of generators of the subgroup); and the time to compute the order of the centralizer (using our algorithm for group order).

For GAP we list only a single time, because GAP computes the order of a centralizer as that centralizer is produced. Note that the times listed for GAP in Table 2 are nearly identical to those in Table 1, suggesting that GAP is using the same strategy for both group order and centralizer computations.

Again, of the five cases where GAP completes, GAP is faster in one (A_{14}), and our method is faster in the other four, with a maximum speedup of over 100 times. For groups of order 10^{15} or less, we compute the centralizer in 20 seconds or less. For the largest case, group order approximately 10^{23}, the algorithm completes in less than half an hour.

6.2 Case Study of a Large Group: J_4

Because our initial motivation for developing biased tadpoles was the problem of conjugacy testing in Janko's group J_4, we report here results for the closely related problem of finding centralizers in that group.

We examine two extreme cases: computing the centralizer of an involution; and of an element of order 37. For involutions, the centralizer is relatively large, and the conjugacy class relatively small. For elements of order 37, the reverse is true. Our element of order 2 is the product $(abababbb)^6$, and our element of order 37 is the product ab (generators as given in v3 of the ATLAS [18]).

Centralizer of an Involution.

In this case, the centralizer is of order $1816657920 \approx 1.8 \times 10^9$ and the conjugacy class (the domain of the tadpole walk) is of order $47766599364 \approx 4.8 \times 10^{10}$.

Using biased tadpoles, we were able to produce one element of the centralizer in an average of 11.17 seconds. We produced 100 such elements, for a total time less than 20 minutes. For these 100 biased tadpoles, an attractor was found after an average of 27500 steps, and there were a total of 4 attractors discovered. Additional analysis of these tadpoles, and a comparison between the biased and unbiased versions, is presented in Section 6.3.1.

Because we expect $O(\log n)$ elements of a group to generate that group, these 100 elements are sufficient to generate the centralizer with very high probability. Theorem 5 produces a bound on the probability of error of 2^{-65}, where $k = 100$ and $L = 34$ with L being the number of prime factors, including repetition, for $|J_4|$. We could not directly confirm this by computation because the size of the first transversal in the centralizer exceeded available memory.

Centralizer of an Element of Order 37.

In this case, the centralizer is of order 37 and the conjugacy class (the domain of the tadpole walk) is of order $2345285703948042240 \approx 2.3 \times 10^{18}$. In this large space, the biased tadpoles required an average of 23.5 hours to complete. We computed 20 such elements in parallel on 20 nodes of a compute cluster. The longest cases, and therefore the entire computation, were completed in about 1.5 days. For these 20 tadpoles, an attractor was found after an average of 2.0×10^8 steps, and a total of 3 attractors were found.

We confirmed that the order of the computed centralizer was 37. The time for this verification was trivial in comparison the tadpole computations.

6.3 Tadpole Accelerators

Here, we present experimental results for two tadpole accelerators: biasing and parallelism. The results show a significant speedup of 9.6 times for biasing in the case computing centralizing elements in J_4, and a full linear speedup of 4.0 times on a machine with 4 CPU cores.

6.3.1 Biasing Effectiveness

For this experiment we used the same 100 tadpole computations that were used to compute the centralizer of an involution in J_4 (Section 6.2). For each of the 100 cases, we ran one biased and one unbiased tadpole.

Table 3: Comparison of biased and unbiased tadpoles (averages over 100 trials).

	Biased		Unbiased	
Tail Length	23751		169497	
Attr Length	3832		224153	
Total Steps	27583		393650	
Steps per sec	2463		3679	
Time (s)	11.2		107.0	
	Attr Size	Visits	Attr Size	Visits
List	1454	75	35350	7
of	7921	11	238364	93
Attrs	11218	6		
	14961	8		

Table 3 presents the results of these 100 trials. The statistics include an average of: the length of the tail (number of steps until an attractor cycle is reached); the length of the attractor cycle; the total number of steps; the time to complete the walk; and the number of steps per second. Also listed are the lengths of the attractor cycles discovered, and the number of trials out of 100 that visited each attractor.

These results show that biasing can significantly decrease the length of a tadpole walk. In this case, biasing yields a 14 times reduction in the total walk length, mostly due to the almost 60 times reduction in the average attractor length. As predicted in Section 5.1, the cost of computing the bias increases the time to complete one step in the tadpole by 50%. Overall in this case, biasing yields an almost 10 times speedup over unbiased tadpoles.

6.3.2 Parallel Speedup on Multicore Architectures

For this experiment, we simply chose one of the biased tadpole walks from the previous section and computed that walk four times. We tested three cases: one process computing all four walks serially; two processes, each computing two walks in parallel; and four processes, each computing one walk in parallel. The three cases completed in 49.23 seconds, 24.64 seconds, and 12.25 seconds, respectively. This test shows the expected four times speedup on a computer with four CPU cores.

7. CONCLUSION AND FUTURE WORK

Biased tadpoles demonstrate a $O(\sqrt{|G|}\log(1/\varepsilon)))$ centralizer algorithm for matrix groups for ε an upper bound on the probability of error. The method is orthogonal to another general method for matrix groups from Magma: the use of the Murray-O'Brien random Schreier-Sims algorithm [12] in combination with Leon's backtracking [8]. We thank the reviewer for pointing out this other method. A combination of our method with that one is planned for the future.

8. ACKNOWLEDGMENT

We acknowledge helpful discussions with Jürgen Müller concerning conjugacy testing, that led us later to consider the more general problem of centralizers. We also thank the reviewers for helpful comments.

9. REFERENCES

[1] J. Bray. An improved method for generating the centraliser of an involution. *Arch. Math. (Basel)*, 74:241–âĂŞ245, 2000.

[2] G. Butler. Computing in permutation and matrix groups II: Backtrack algorithm. *Mathematics of Computation*, 39(160):671–680, Oct. 1982.

[3] F. Celler, C. Leedham-Green, S. Murray, A. Niemeyer, and E. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23:4931–4948, 1995.

[4] G. Cooperman and L. Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992.

[5] G. Cooperman and M. Tselman. Using tadpoles to reduce memory and communication requirements for exhaustive, breadth-first search using distributed computers. In *Proc. of ACM Symposium on Parallel Architectures and Algorithms (SPAA-97)*, pages 231–238. ACM Press, 1997.

[6] A. Hulpke and S. Linton. Total ordering on subgroups and cosets. In *Proc. of Int. Symp. on Symbolic and Algebraic Comp. (ISSAC-03)*, pages 156–160, 2003.

[7] J. Leon. On an algorithm for finding a base and strong generating set for a group given by a set of generating permutations. *Math. Comp.*, 35:941–974, 1980.

[8] J. S. Leon. Permutation group algorithms based on partitions, I: Theory and algorithms. *J. Symbolic Computation*, 12:533–583, 1991.

[9] J. S. Leon. Partitions, refinements, and permutation group computation. In *Groups and computation II, 1995*, volume 28 of *DIMACS (Discrete Math. Theor. Comp. Sci.)*, pages 123–158. Amer. Math. Soc., 1997.

[10] S. Linton. Finding the smallest image of a set. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation (ISSAC-04)*, pages 229–234, New York, NY, USA, 2004. ACM.

[11] E. M. Luks. Computing in solvable matrix groups. In *33rd Annual Symposium on foundations of Computer Science*, pages 111–120. IEEE, 1992.

[12] S. H. Murray and E. A. O'Brien. Selecting base points for the schreier-sims algorithm for matrix groups. *J. Symb. Comput.*, 19:577–584, 1995.

[13] I. Pak. The product replacement algorithm is polynomial. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 476–485. IEEE Press, 2000.

[14] R. Parker. Tadpole, 1986. oral communication.

[15] J. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[16] R. Sedgewick and T. G. Szymanski. The complexity of finding periods. In *STOC '79: Proc. of the 11th annual ACM Symposium on Theory of Computing*, pages 74–80, New York, NY, USA, 1979. ACM.

[17] H. Theißen. *Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen*. PhD thesis, Rheinisch-Westfalische Technische Hochschule, Aachen, Germany, 1997.

[18] R. Wilson and et al. ATLAS of finite group representations – version 3. `http://brauer.maths.qmul.ac.uk/Atlas/v3/`.