Appl. Algebra, Alg. Algo. and Error-Correcting Codes (AAECC-8, 1990) Springer-Verlag Lecture Notes in Comp. Sci. **508**

APPLICATIONS OF CAYLEY GRAPHS

G. Cooperman, L. Finkelstein and N. Sarawagi
College of Computer Science
Northeastern University
Boston, Ma. 02115, U.S.A.

Abstract

This paper demonstrates the power of the Cayley graph approach to solve specific applications, such as rearrangement problems and the design of interconnection networks for parallel CPU's. Recent results of the authors for efficient use of Cayley graphs are used here in exploratory analysis to extend recent results of Babai et al. on a family of trivalent Cayley graphs associated with $PSL_2(p)$. This family and its subgroups are important as a model for interconnection networks of parallel CPU's. The methods have also been used to solve for the first time problems which were previously too large, such as the diameter of Rubik's $2 \times 2 \times 2$ cube. New results on how to generalize the methods to rearrangement problems without a natural group structure are also presented.

1. Introduction

Each finite group G, together with a generating set Φ, determines a directed graph called a Cayley graph. Once a Cayley graph has been constructed for G, it is possible to obtain algorithmic solutions to the following problems: describe a complete set of rewriting rules for G relative to some lexicographic plus length ordering on the words of Φ [9]; obtain a set of defining relations for G in terms of Φ [6]; and find a word in Φ of minimal length that represents a specified element of G. The last problem is called the *minimal word problem* for G. The solution of the minimal word problem provides an optimal strategy for many rearrangement problems, where the elements of the generating set have some physical significance. These include problems in communications, which can be viewed as token movements on graphs [11], as well as such popular puzzles as Rubik's cube.

There has been a great deal of interest recently in Cayley graphs and their generalization, Schreier coset graphs, for their exceptionally nice characteristics both as models for traditional parallel network architectures and as a potential source of new networks for parallel CPU's [1, 3, 5, 7, 8]. Using Cayley graphs, researchers have discovered new regular graphs with more nodes for a given diameter and for a given number of edges per node than were previously known. This allows construction of larger networks, while meeting design criteria of a fixed number of nearest neighbors and a fixed maximum communication time between arbitrary nodes.

This paper uses theoretical techniques originally developed for designing parallel networks of CPU's [9], and applies them to applications requiring a (sub)optimal solution to the minimal word problem. The purpose is as much to demonstrate the power of the Cayley graph approach, as to solve specific applications. Using these techniques, it has been possible to make empirical observations and computational progress which would not have been possible in a more traditional approach, such as the rewriting system approach described by the authors in an earlier work [4]. An immediate example is the computation of the full Cayley graph of Rubik's $2 \times 2 \times 2$ cube (3,674,160 nodes) on a SUN-3 with 8 megabytes of storage in less than 60 CPU hours.

In understanding the significance of this work, it is important to observe that in applications which involve large groups, one is most often limited by memory resources rather than time. The theoretical tools described in [9] provide both space-efficient data structures and CPU-efficient algorithms for computing with Cayley graphs and Schreier coset graphs. In particular, we are able to implicitly store a minimal spanning tree for a Cayley graph (or Schreier coset graph) using only $\log_2(3)$ bits per node, independant of the size of the generating set, plus additional storage which is small in comparison to the total. Routing depends on the nature of the representation of the underlying group, but can always be performed efficiently for the class of permutation groups. Similar ideas have appeared elsewhere, (see for example [12]), although this is the first time that these techniques have been successfully applied to Cayley and Schreier coset graphs.

These techniques greatly increase the range of problems that can be solved. To see this, it is necessary to first review the definition of a Cayley graph.

A *Cayley graph* $\mathcal{G}$ is a directed graph associated with a group G, and set of generators Φ. The nodes of $\mathcal{G}$ are the elements of G and the edges are labelled by generators in Φ. We will always assume that Φ is closed under inverses. If α and β are two nodes connected by a directed edge (α, β) and the edge is labelled by $\phi \in \Phi$, then $\beta = \alpha\phi$ as an element of G. A *Schreier coset graph* is similarly defined, but requires the additional specification of a subgroup H of G. A Schreier coset graph $\mathcal{G}_H$ is defined to be a directed graph whose nodes are the right cosets of H in G and whose edges are labelled by generators in Φ. If $H\alpha$ and $H\beta$ are two nodes connected by the directed edge $(H\alpha, H\beta)$ with label ϕ, then $H\alpha\phi = H\beta$.

Memory, rather than CPU time, represents the limiting resource for constructing both Cayley graphs and Schreier Coset graphs within the computer technology of today and the near future. This can be shown informally by examining the requirements for using simple breadth-first search to construct a spanning tree for a Cayley graph. This clearly requires space proportional to $|G|$, the order of the group G. The corresponding time requirements are proportional to $|\Phi||G|$, assuming hashing takes constant time. $|\Phi|$ is usually small for many group generating sets of interest, while $|G|$ is on the order of thousands, millions, or more. A unit memory operation may require 8 bytes (one 4-byte word to store a node representation, and one 4-byte word to store a pointer to a parent). The corresponding time to examine the $|\Phi|$ neighbors of a known node (to find new nodes) is usually significantly less than a CPU millisecond on a SUN-3 workstation.

Under the assumptions of 8 bytes and 1 millisecond per node in G and assuming $|G| = 10,000,000$, the computation will require 80 megabytes of storage (excluding storage for the hash table) and approximately 14 CPU hours on a workstation. Since hashing of new nodes represents random accesses throughout data memory, efficient execution requires that all data be stored in semiconductor memory. Otherwise, frequent random disk accesses for virtual memory would make the program unacceptably slow. Hence, the requirements of 80 megabytes of semiconductor memory and 14 SUN-3 CPU hours clearly show the memory to be the limiting resource. In today's technology, the use of a supercomputer would show memory to be even more of a limiting resource. A future generation of parallel CPU's only strengthens further the argument that memory is the critical resource for constructing Cayley graphs.

There are many interesting applications that are outgrowths of the ability to compute with large examples. In section 3, these techniques are applied to an important family of trivalent Cayley graphs associated with the parametrized family of groups $PSL_2(p)$, for p a prime. Babai, Kantor and Lubotzky [2] describe an elegant routing algorithm for these graphs with the property that a path between any two nodes has length at most $45\log(|G|)$. Although this provides an upper bound on the diameter, these graphs are substantially more dense than indicated by the work of Babai et al., and therefore more interesting as possible interconnection networks. This has reduced the worst case estimate of the diameter to $22.5\log(|G|)$. Furthermore, in these groups short relations have been discovered that hold for all primes p.

In section 4, we present some results about the Cayley graph for Rubik's $2 \times 2 \times 2$ cube. To the authors' knowledge, this is the first time that the diameter of Rubik's $2 \times 2 \times 2$ cube has ever been computed. Yet, it was carried out in LISP on a SUN-3 with 8 megabyte of memory, and used included 1 megabyte for the main data structure. Estimates of space required to map out other well-known groups are presented. In particular, in the implementation of an interconnection network for parallel CPU's, a graph with 64,000 nodes could be mapped out and stored at each node of the network, yet consuming only 13 kilobytes for each instance.

In section 5, the methodology is extended to Schreier coset graphs. This allows one to model certain rearrangement problems in which the composition of two legal moves is not always possible. This is the case, for example, in the popular 15-puzzle or more generally to certain token movement problems on graphs [11]. In the token movement problem, labelled tokens are placed on the nodes of a graph. At least one token is designated a blank token. A *legal* move is one which interchanges a blank token with any token currently residing on a neighbor node. The object is to see if a goal configuration can be reached from some initial configuration through a sequence of legal moves.

This methodology has a wide range of applications including problems in operations research and the management of memory in totally distributed systems. Kornhauser et al. [11] developed an approach to this problem in which the question of whether the tokens can be arranged in a specified configuration is reduced to the group membership problem [13], which is solvable in polynomial time. Unfortunately their methodology for finding solutions will never yield an optimal length solution, because each of their group generators is composed of several token moves.

In order to remedy this, we describe a previously unpublished technique [10] and show how it can be extended so that many of these pebble moving problems can be solved optimally by finding a path of shortest length in a Schreier coset graph from the identity coset to a specified coset. This means that each generator, for a certain group associated with the graph, corresponds to a legal move and that the cosets for a suitably chosen subgroup are in 1-1 correspondence with the set of states of the problem. Using techniques in [9], we may now achieve substantially shorter solutions than those using a direct group theoretic approach. We illustrate these ideas in section 5, for the 8-puzzle.

2. Space-Efficient Data Structures for Cayley Graphs.

Given a finite group G and generating set Φ, there are two keys to our construction of a space-efficient data structure for the Cayley graph $\mathcal{G}$. First, an easily computable function *count* is used which assigns to each element $g \in G$, a unique integer in the range 0 to $|G| - 1$. This allows us to store information for a node of the graph in an array of length $|G|$ by using *count* as an index into the array, instead of storing an explicit node representation. Second, the distance from a node to the identity modulo some base is used instead of pointers to the parent or other neighboring nodes. Similarly for a Schreier coset graph $\mathcal{C}_H$ defined by the subgroup H of G, a function $count_H$ can be used which assigns to each right coset Hg of H, a unique integer in the range 0 to $[G : H] - 1$.

The functions *count* and $count_H$ assume the existence of some concrete representation for G, such as a group of permutations or matrices. In the case where G is an arbitrary permutation group, *count* is defined using standard ideas from computational group theory. The function $count_H$ is far more subtle and depends on a delicate counting argument given in [9]. The general description of *count* and $count_H$ is omitted, since this work does not depend on the details of computation of those functions. A method for defining *count* in the case where G is the set of unimodular 2×2 matrices over $GF(p)$, p a prime, is given in the next section.

In order to simplify the discussion, we restrict our attention to Cayley graphs. The case of Schreier coset graphs is a straightforward generalization. Given G, Φ and $\mathcal{G}$, allocate a bit vector D of length $2|G|$, and associate with each pair of bits a unique address from 0 to $|G| - 1$.

If $count(g) = i$, then we store in $D[i]$, the distance modulo 3 in $\mathcal{G}$ from g to the identity node e. Note that the distance from g to e, is the minimal length of any word in Φ which represents g. D will sometimes be referred to as the *2-bit data structure* for $\mathcal{G}$. We define a *parent* of node g to be any neighbor which has distance to the identity one less than that of g. Note that g need not have a unique parent. Similarly, we define a *child* of g to be any neighbor which has distance one more than that of g. A *sibling* of g is any neighbor which has the same distance as g. Note that only the identity node does not have a parent.

Finding a minimal word representation for $g \in G$ is simple once D has been constructed. The idea is to create a path of minimal length from g to e by choosing an arbitrary parent node as the successor of each node along the path. Each time a new node is selected, the distance to e is diminished by one unit. Since e has no parent, the path eventually must terminate at e. The length of the path is equal to the length of a minimal word representation for G. In order to find a parent of g, it suffices to check the values of $D[count(g\phi)]$ for each $\phi \in \Phi$. If g is not equal to e, then there exists a parent node $g\phi_1$ of g. We then continue the process with g replaced by $g\phi_1$. If $\phi_1, \phi_2, \ldots, \phi_k$ is the sequence of edge labels along the path from g to e, then $g\phi_1 \cdots \phi_k = e$. Since Φ is closed under inverses, it then follows that $\phi_k^{-1} \cdots \phi_1^{-1}$ is a word of minimal length which represents g.

The time for computing a minimal word representation for $g \in G$ is $O(d|\Phi||count|)$, where d is the diameter of $\mathcal{G}$, and $|count|$ is the cost of invoking the function *count*. In the case where G is represented as a permutation group of degree n, then $|count| = O(m^2)$ where $m < n$ is the size of a *base* for G. A base is a subset of $\{1, 2, \ldots, n\}$ with the property that only the identity of G fixes every point of the set (see [13] for related concepts). Many interesting groups have bases with a small number of points. For these cases, the cost of computing *count* will be small in proportion to n. In the family of groups $PSL_2(p)$ of section 3, the cost of computing *count* is $O(1)$.

The above scheme leads to a simple routing algorithm for $\mathcal{G}$. Suppose a path of minimal length between two arbitrary nodes g and h is desired. If $\phi_1 \cdots \phi_k$ are the labels along a path of minimal length from e to $g^{-1}h$, then the path from g specified by the sequence $\phi_1, \ldots, \phi_k$ will terminate in h and have minimal length. (We identify the group element gh^{-1} with the name of the corresponding node.)

Storing the distance of each node to the identity modulo 3 clearly requires at least $\log_2(3) \approx 1.58$ bits per node. One can use 8 bits to store such data for 5 nodes, leading to 1.6 bits per node, which is close to the theoretical optimum. To efficiently compute the data structure initially requires additional temporary storage. In [9], a more complex scheme is described to compute the data structure using 2 bits per node of storage in time proportional to $d|G||\Phi||count|$. That scheme requires that $count^{-1}$ be computed within the same time bounds as *count*.

The construction from start of the 2-bit array D from the generators of a group follows. First, initialize D so that $D[i] = 3$, for $1 \leq i \leq |G|-1$ and $D[0] = 0$ (assuming that $count(e) = 0$). The value 3 for an entry in D is a marker which indicates that the final value, which must be 0, 1, or 2, has not yet been entered. Assume that for $\ell \geq 0$, we have filled in the value of $D[count(g)]$ for all nodes g with distance at most ℓ from e. The initialization takes care of the case when $\ell = 0$. To compute the correct D values for nodes at distance $\ell + 1$, for each j with $D[j] = 3$ check if $g = count^{-1}(j)$ has a parent h at distance ℓ from e. This is a necessary and sufficient condition for g to have distance $\ell + 1$ from e. To check this, compute $D[count(g\phi)]$ for each $\phi \in \Phi$. If any has value ℓ mod 3, then enter $\ell + 1$ mod 3 in $D[j]$.

The time to construct the 2-bit data structure can be reduced to $|G|(d + |\Phi||count|)$ by using $\log_2(5)$ bits per node [9]. The method also generalizes to Schreier coset graphs.

3. Exploring $PSL_2(p)$.

This section is concerned with an exploratory analysis of rewrite rules and short word algorithms for $PSL_2(p)$, p a prime, $p > 2$. The purpose of this section is to show the advantages of using our Cayley graph techniques to explore an interesting class of trivalent graphs discovered by Babai, Kantor and Lubotsky [2] associated with the groups $PSL_2(p)$, p a prime. We present some computational results on the actual diameters of these graphs together with the discovery of some new general relations for these groups, which lead to a shorter routing algorithm than the one described by the authors.

The trivalent graphs for $PSL_2(p)$ which we have studied are only a special case of a more general theory developed by Babai et. al. Their main result shows that every finite simple group G has a set of at most 7 generators so that every element $g \in G$ can be written as a word of length $O(\log_2(|G|))$ in these generators. Furthermore, they give an algorithm for finding a short, but not minimal, word representation for each element of G in terms of these generators. The implication for Cayley graphs is clear. The case where $G = PSL_2(p)$ is the core case and appears to be the one for which the corresponding Cayley graph will be the most dense.

The existence of a short word algorithm is very significant because it leads to a host of possible dense Cayley graphs as models for interconnection networks with good built in routing algorithms. Some very dense Cayley graphs have recently been discovered (see section 1). However, many of these do not have "natural" routing algorithms. The hypercube is an example of an interconnection network with a particularly nice routing algorithm, but which is relatively sparse. A node of the hypercube can be represented as a vector of bits (0's and 1's). A step to another node can be represented as toggling a single bit.

The group $PSL_2(p)$ is the quotient group of $SL_2(p)$ (the set of 2×2 matrices over $GF(p)$ of determinant 1) by its center which is the cyclic group generated by $-I$, where I is the 2×2 identity matrix. Our investigation begins with the generating set $S = \{x(1), x(1)^{-1}, r' \equiv h(1/2)r\}$, for $PSL_2(p)$, where

$$x(t) = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix},\ h(b) = \begin{pmatrix} b^{-1} & 0 \\ 0 & b \end{pmatrix} \text{ for } b \neq 0, \text{ and } r = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

It was shown in [2] that the diameter of the Cayley graph $\mathcal{G}(p)$ for $PSL_2(p)$ with respect to S is $O(\log_2 |PSL_2(p)|)$. In fact it was shown to be bounded by $45 \log_2(|PSL_2(p)|)$. This is a direct consequence of a clever short word algorithm. Since $|PSL_2(p)| = p(p^2 - 1)/2$, this leads to the upper bound on the diameter of $\mathcal{G}(p)$ of $\lceil 135 \log_2(p) - 45 \rceil$. A closer reading of [2] shows that the diameter of $\mathcal{G}(p)$ is actually bounded by $45 \log_2(p)$.

We found the true diameters by generating the 2-bit data structure for $\mathcal{G}(p)$ for various p up to $p = 131$. The generating set used was S in all cases. The encoding function *count* that was used for for the 2-bit data structure was defined from G to the set of positive integers $< p^3$. For any $g = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in PSL_2(p)$, since $ad - bc = 1$, $count(g) = bp^2 + cp + d$ if $d \neq 0$. Otherwise, $count(g) = ap^2 + cp$. The true diameters are compared with the theoretical estimate below. Most graphs $\mathcal{G}(p)$ were generated in a matter of minutes, and the largest ($p = 131$) required a few hours and half a megabyte of data, using the above data encoding resulting in a density of half the optimal density.

Group ($PSL_2(p)$)	Number of Nodes	Bound on Diameter ($45\log_2(p)$)	True Diameter
$PSL_2(3)$	12	72	3
$PSL_2(5)$	60	105	6
$PSL_2(7)$	168	127	9
$PSL_2(11)$	660	156	11
$PSL_2(13)$	1,092	167	12
$PSL_2(17)$	2,448	184	14
...		...	
$PSL_2(107)$	612,468	303	25
$PSL_2(109)$	647,460	305	28
$PSL_2(113)$	721,392	307	26
$PSL_2(127)$	1,024,128	315	27
$PSL_2(131)$	1,123,980	317	26

This large difference in the actual diameters of these Cayley graphs from the estimated diameters led to a closer study of the short word algorithm in [2]. Instead of the theoretical upper bound of $45\log_2(p)$, the table shows an empirical fit of the diameter to $4\log_2(p)$.

The original short word algorithm is reproduced here, followed by a discussion of the improvements that were made. Let $g = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in PSL_2(p)$ with $ad - bc = 1$ and $c \neq 0$ then

$$(A) \qquad g = x(c^{-1}(a-1))rx(-c)rx(c^{-1}(d-1))$$

$$(B) \qquad r = h(2)r'$$

Moreover if $0 \leq n < p$, $m+1 = \lceil \log_4(p-1) \rceil$ and $n = \sum_{i=0}^{m} a_i 4^i$ is the base 4 representation of n, then

$$(C) \qquad x(n) = h(2)^{-m}x(a_m)h(2)x(a_{m-1})h(2)\ldots x(a_1)h(2)x(a_0)$$

where $a_i \in \{0,1,2,3\}$. (A) implies that the length of g with respect to the generating set S is $3\max_{1\leq t<p}(length(x(t))) + 2\ length(r)$. If $c = 0$ then $rg = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ with $c' \neq 0$. Therefore the diameter of $\mathcal{G}(p)$ is at most $3\max_{1\leq t<p}(length(x(t))) + 3\ length(r)$. Since $r = h(2)r'$ and by (C) the length of any $x(t)$ and r with respect to S depends upon the length of $h(2)$, it is crucial to get a short word representation for $h(2)$. Babai et al. gave a word for $h(2)$ in S of length 13,

$$h(2) = x(1)^{-2}r'x(1)^2r'x(1)r'x(1)^{-4}r'.$$

This formula is true for all $PSL_2(p)$. Using the fact that the length of $h(2) \leq 13$, the diameter of $\mathcal{G}(p)$ was estimated to be $\leq 45\log_2(p)$.

Since the constant 45 in the estimate of the diameter depended on the length of the word representation of $h(2)$, it was natural to try to find a minimum length word representation for $h(2)$. Our approach was to use the 2-bit data structure for $\mathcal{G}(p)$ to compute the shortest word for $h(2)$ in each $PSL_2(p)$ as p was increased.

It was soon observed experimentally that for a number of values $p \geq 11$ a minimum word for $h(2)$ was given by the same word of length 9 in $x(1)$ and r'.

$$h(2) = x(1)r'x(1)^4r'x(1)r', \quad p \geq 11$$

This led to the hypothesis that the relation would be true for all p, $p > 2$. This fact can be directly verified by simply transforming the above elements to matrices over $GF(p)$ and performing the required multiplication.

The shorter formula for $h(2)$ lowered the estimate from $45\log_2(p)$ to $32\log_2(p)$. Further study of the algorithm then resulted in a decrease of the diameter to $22.5\log_2(p) - 33$, using symbolic manipulation.

The three identities

$$(i) \qquad r'h(2)^{-m} = h(2)^m r',$$

$$(ii) \qquad x(t)h(2)^m = h(2)^m x(4^m t) \ \forall m \geq 0, \text{ and}$$

$$(iii) \qquad rx(t)r = r'x(4t)r'$$

were employed to find a shorter word for g in equation (A) as follows.

Using identity (iii) in equation (A) we get,

$$(A') \qquad g = x(c^{-1}(a-1))r'x(-4c)r'x(c^{-1}(d-1))$$

If $c = 0$ then $r'g = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ with $c' \neq 0$. Since r' is a generator, the diameter of $PSL_2(p)$ is at most $1 + length(g)$, where g is as in equation (A').

Let $u = c^{-1}(a-1) = \sum_{i=0}^m u_i 4^i$, $t = -4c = \sum_{i=0}^m t_i 4^i$, and $v = c^{-1}(d-1) = \sum_{i=0}^m v_i 4^i$ be their base 4 representations, $(u_i, t_i, v_i \in \{0,1,2,3\})$. Then rewriting (A') we see, $g = x(u)r'x(t)r'x(v)$.

Using (C), (i) and (ii) successively in the above equation, we can obtain the following equation.

$$\begin{aligned} g = {} & h(2)^{-m}x(u_m)h(2)x(u_{m-1})h(2)x(u_{m-2})h(2)\ldots x(u_1)h(2)x(u_0) \\ & r'x(t'_m)h(2)x(t'_{m-1})h(2)x(t'_{m-2})h(2)\ldots x(t'_1)h(2)x(t'_0) \\ & r'x(v_m)h(2)x(v_{m-1})h(2)x(v_{m-2})h(2)\ldots x(v_1)h(2)x(v_0) \end{aligned}$$

By the above expansion, $length(g) \leq 4m \ length(h) + 3(m+1) \ length(x(3)) + 2$. Since $length(h) \leq 9$, and $length(x(3)) \leq 3$, so $length(g) \leq 45m + 11$. Moreover, $m + 1 = \lceil \log_4(p-1) \rceil$ implies that $length(g) \leq 22.5\log_2(p) - 34$. This proves the following result.

Proposition 3.1. *diameter*$(\mathcal{G}(p)) \leq 22.5\log_2(p) - 34$.

As a step to further improve the theoretical estimate of the diameter for $\mathcal{G}(p)$, the minimum words for $x(t)$ were studied experimentally, using the precomputed 2-bit data structure for $\mathcal{G}(p)$. Since $x(t)^{-1} = x(p-t)$ in $PSL_2(p)$, only the minimum words for $x(t)$ for $1 \leq t \leq (p-1)/2$ needed to be examined. This was done for $PSL_2(p)$ for all prime p from 3 to 131. The minimum words either fell into the following four patterns, or were well-determined concatenations of these four patterns. We would expect that new patterns would emerge for significantly larger p, but the range up to 131 seems to be sufficient for currently envisioned applications.

(a) For sufficiently small t depending on p, $x(t)$ had a minimal word of the form $x(1)^t$.

(b) For each $p > 33$, $x((p-3)/2)$ had the same minimum word of length 15,

$$x((p-3)/2) = r'x(1)^4 r'x(1)^{-1} r'x(1)^{-1} r'x(1)^4 r'.$$

When this word was multiplied out over $GF(p)$, it was found to be equal to $x(-3/2)$. This provided a word, $W_{3/2}$, in the elements of S for $x(-3/2)$ of length 15. Concatenating $W_{3/2}$ with $x(1)^i$ forms minimal words for $x(i + (p-3)/2)$ for some i. For example $x((p-1)/2)$ had minimal word $x(1)W_{3/2}$ of length 16, for $p > 33$.

(c) For many large p, $x(20)$ had the same minimal word W_{20} of length 19. For example $W_{20}x(1)$ and $W_{20}x(1)^2$ were the minimal words for $x(21)$ and $x(22)$ of lengths 20 and 21 respectively. We also observed that the minimal word for $X(23)$ was not $W_{20}x(1)^3$ of length 22, but was $W_{24}x(1)^{-1}$ of length 21, where W_{24} was the minimal word for $x(24)$ of length 20. This lead to a set of formulas for $x(4i) = W_{4i}$ where

$$W_{4i} = r'x(1)^{-1} r'x(1)^{-4} r'x(1)^{i-2} r'x(1)^{-4} r'x(1)^{-1} r'$$

is a word in S of length $14 + i$, for all $i > 2$. W_{4i} concatenated by $x(1)$, $x(1)x(1)$ or $x(1)^{-1}$ gives words for $x(4i+1)$, $x(4i+2)$ or $x(4i-1)$ of length 15+i, 16+i or 15+i respectively.

(d) Finally there were several values of t for which the shortest word, when multiplied out over the rationals yielded formulas for $x(1/4)$, $x(7/4)$, $x(9/4)$, $x(3/8)$, $x(13/8)$, $x(11/8)$, $x(9/8)$.

Using the above four patterns and the formulas discovered, the shortest word for any $x(t) \in PSL_2(p)$ can easily be found for $p \leq 131$. Disregarding the minimal words of type (d), a simple short word algorithm for $x(t)$ is described below.

Short-Word-Algorithm *Input:* An arbitrary t, $1 \leq t < p$, where $PSL_2(p)$. *Output:* A short word for $x(t)$.

```
If $t \leq (p-1)/2$
    If $t = (p-1)/2$
        Then if $t < 16$ Output $x(1)^t$
        Else Output $x(1)W_{3/2}$
    Else write $t = 4q + r$ , where $0 \leq r < 4$
    $d \leftarrow$ minimum $\{t, 15 + (p-3)/2 - t, 14 + q + r, 14 + q + 2\}$
        if $d = t$ output $x(1)^t$
        if $d = 15 + ((p-3)/2) - t$ output $W_{3/2}x(1)^{t-(p-3)/2}$
        if $d = 14 + q + r$ and $r < 3$ output $W_{4q}x(1)^r$
        if $d = 14 + q + 2$ and $r = 3$ output $W_{4(q+1)}x(1)^{-1}$
Else $t > (p-1)/2$ output the inverse word of Short-Word-Algorithm$(p-t)$
```

Using the above algorithm to find a short word for $x(t)$, and using the factorization of an arbitrary $g \in G$ in terms of the $x(t)$'s and r', we can show that the diameter of $\mathcal{G}(p)$ is at most $3(p/10 + 16.3) + 3$. For $p < 349$ this leads to a better estimate of the diameter than the best known asymptotic theoretical bound, $22.5\log_2(p) - 33$.

Another interesting application of the 2-bit data structure, is its use to find all possible rewrite rules in the elements of S which are true for all $PSL_2(p)$. It was hoped to find a family of rewrite rules which would reduce an algorithmically derived short word for a given group element into a still shorter word. The idea is as follows. If an element $g \in G$ has two different minimal word representations, say w_1 and w_2, then both w_1 and w_2 can be found easily from 2-bit data structure. Therefore $w_1 w_2^{-1} = e$ is a relation of even length in G. Further, if an element $g \in G$ and the element sg where $s \in S$, have respective minimal words w_1 and w_2 of the same length then again w_1 and w_2 can be found easily, and $sw_1w_2^{-1} = e$ is a relation of odd length in G. In this way all such relations of odd and even length in G can be found.

All such relations in $PSL_2(131)$ up to length 20 were generated as described above, and then checked to see which relations were also true for all p. Three such universal relations were found of length 2, 9, and 21.

(i) $$(r')^2 = 1$$

(ii) $$(r'x(1)^2)^3 = 1$$

(iii) $$r'x(1)^{-1}r'x(1)^4r'x(1)r'x(1)^4r'x(1)^{-1}r'x(1)^{-4} = 1$$

4. Large Problems.

In addition to efficiently exploring many smaller cases, as in section 3, the space-efficient version of Cayley graphs is especially important in solving larger problems which would formerly have either been infeasible or required much larger computers. As an example, we find that on a SUN-3 workstation using LISP we are able to process 1,000,000 nodes of a Cayley graph for $PSL_2(p)$ every 4 CPU hours. The rate is roughly independent of p.

Our largest example to date is finding the diameter of the Cayley graph for the group `Rubik2` associated with Rubik's $2 \times 2 \times 2$ cube. Singmaster [14, p. 60] poses this as an outstanding problem. This cube consists of the corners of the traditional Rubik's $3 \times 3 \times 3$ cube, while ignoring all other sub-blocks. The entire Cayley graph for `Rubik2` was mapped out in place and stored, using $\log_2(5)$ bits/node $\times 3,674,160$ nodes $= 1$ megabyte of space. The task used the more general function *count* instead of the PSL-specific encoding and required 60 CPU hours on a SUN-3 workstation.

The more traditional method of breadth-first search, as described in the introduction, would have required 8 bytes/node $\times 3,674,160$ nodes $= 30$ megabytes of memory for data, or a ratio of 30 times more data storage. If a hashing scheme was used the ratio would have been proportionately increased due to overhead of the hash table. While this represents only an argument with respect to order of magnitude, the ratio of 30 illustrates the power of the proposed methodology.

The traditional generating set for `Rubik2` consists of nine elements. If an orientation of the $2 \times 2 \times 2$ cube is fixed, there are three basic moves: $u =$ rotation of the upper face by 90 degrees clockwise; $f =$ rotation of the front face by 90 degrees clockwise; and $l =$ rotation of the left face by 90 degrees clockwise. This, along with their inverses and their squares form the nine element generating set.

The diameter of the Cayley graph for `Rubik2` with these nine generators is 11. This was found as a result of constructing the 2-bit data structure for the Cayley graph. If the generating set is restricted to the six independent generators and inverses $\{u, f, l, u^{-1}, f^{-1}, l^{-1}\}$, then the diameter of the Cayley graph is 14. This computation took 100 CPU hours.

5. Extension to Schreier Coset Graphs.

Many applied rearrangement problems have invertible state transitions (legal moves), but do not have a natural group structure. This prevents one from applying the Cayley graph methodology. An interesting (and difficult) example of a rearrangement problem without a natural group structure is the classic 15-puzzle. This has 15 tiles and a blank arranged in a 4×4 rectangle. A legal move consists of interchanging a tile and the blank. The goal is to achieve a specified configuration of the tiles.

One would like to impose a group structure on the 15-puzzle in which the generators are the legal moves and the binary operator is composition. Where legal moves exist, associativity holds and there is an inverse. The identity is the null move. However, arbitrary moves cannot be composed, and so there is not a well-defined binary operator for the group. For example, a legal move interchanging a tile in position 7 with a blank in position 8 cannot be followed by a legal move interchanging a tile in position 2 with a tile in position 3.

A new method is given for re-formulating the 15-puzzle as a coset problem, in which each legal move corresponds to a generator of a group, and the goal state corresponds to a certain coset of a specified subgroup of the group. This allows the use of Schreier coset graphs as described in section 2.

Kornhauser et al. [11] described a formulation which has the defect of mapping a product of legal moves into a generator for a specified group. They also show that any state accessible as a product of arbitrary legal moves in the original formulation will also be accessible as a product of generators in the new formulation. Thus the existence problem is formulated as a group membership problem [13] in a group-theoretic setting. However, a word in the group-theoretic setting will in general correspond to a longer word in terms of the original legal moves, and so their approach cannot be used to find optimal solutions. The approach described here does not suffer that defect.

For purposes of exposition, the 8-puzzle is discussed. This is the 3×3 analogue of the 15-puzzle above. The extension of the technique to the 15-puzzle and other problems will be obvious.

The Eight Puzzle. The 3×3 board of the 8-Puzzle can be thought of as being in one of three *configurations* based on the location of the blank space. Each of the configurations has a set of four *orientations*: up, right, down, and left. We label the cells in each configuration and orientation as shown below. The configuration appears above the *configuration label* (A, B, or C), and four cells for the orientation appear below. The blank space of a configuration is shown in square brackets.

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & [9] \\ & A & \\ \\ \multicolumn{3}{c}{28\ 29\ 30\ 31} \end{array} \qquad \begin{array}{ccc} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & [17] & 18 \\ & B & \\ \\ \multicolumn{3}{c}{32\ 33\ 34\ 35} \end{array} \qquad \begin{array}{ccc} 19 & 20 & 21 \\ 22 & [23] & 24 \\ 25 & 26 & 27 \\ & C & \\ \\ \multicolumn{3}{c}{36\ 37\ 38\ 39} \end{array}$$

Given a state of the 8-puzzle in configuration A and orientation 28, there are two possible legal moves: sliding the tile in cell 8 into cell 9, and sliding the tile in cell 6 into cell 9. Both of these result in a board position in Configuration B, although the latter move requires a built-in rotation of the board by 90 degrees so that the blank ends up in the center cell of the bottom row.

For the moment, we ignore the orientation cells, and describe the action on the 9 tiles of a given configuration. Consider the move $(6 \to 9)$. This is represented by mapping the cells of configuration A into configuration B with a 90 degree rotation. Thus, applying $(6 \to 9)$ in configuration A followed by a 90 degree rotation yields the following configuration.

$$\begin{array}{ccc} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 6 & [9] & 3 \end{array}$$

This arrangement must then be mapped into the cells of configuration B. Doing so yields the permutation given below. (Cells of configuration C are fixed).

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \\ 12 & 15 & 18 & 11 & 14 & 16 & 10 & 13 & 17 & 7 & 4 & 1 & 8 & 5 & 2 & 6 & 9 & 3 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \end{pmatrix}$$

Finally, the orientation cells must be mapped. Since the previous move involved a 90 degree rotation, the four orientation cells of configuration A would be mapped by a cyclic rotation into the orientation cells of configuration B.

In this manner we can represent each of the nine basic moves of the 8-puzzle. Three examples are given below. The right arrow indicates which tile is being moved into which blank square. The two configurations that are interchanged are also indicated, along with the mapping of cells. Those cells which are mapped to themselves are not shown. For the 8-puzzle the generators are transpositions, although this is not a requirement of this methodology.

$(8 \rightarrow 9)$ $(A \leftrightarrow B)$ (0 degree rotation)

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 18 & 17 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 9 & 8 & 32 & 33 & 34 & 35 & 28 & 29 & 30 & 31 \end{pmatrix}$$

$(14 \rightarrow 17)$ $(B \leftrightarrow C)$ (0 degree rotation)

$$\begin{pmatrix} 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 19 & 20 & 21 & 22 & 26 & 24 & 25 & 23 & 27 & 10 & 11 & 12 & 13 & 17 & 15 & 16 & 14 & 18 & 36 & 37 & 38 & 39 & 32 & 33 & 34 & 35 \end{pmatrix}$$

$(24 \rightarrow 23)$ $(B \leftrightarrow C)$ $(-90$ degree rotation)

$$\begin{pmatrix} 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 25 & 22 & 19 & 26 & 24 & 20 & 27 & 23 & 21 & 12 & 15 & 18 & 11 & 17 & 14 & 10 & 13 & 16 & 39 & 36 & 37 & 38 & 33 & 34 & 35 & 32 \end{pmatrix}$$

Let G be the group generated by the obvious 9 generators corresponding to legal moves of the 8-puzzle, and let H be the point stabilizer subgroup of G which stabilizes all the cells of configuration A (1 through 9 and 28 through 31). A scrambled puzzle is then represented by a permutation, where the cells of configuration A are mapped according to how the puzzle is scrambled, with the cells of the configuration in which the scrambled puzzle lies. In fact a coset of G in H corresponds to a unique scrambled puzzle. Solving this scrambled puzzle, in group theoretic terms, is to find a word in the 9 nine generators for the coset representative of H in G that contains the permutation representing the scrambled puzzle.

We generated the 2-bit data structure for the for the Schreier coset graph of the 8-Puzzle and found that the diameter of the graph was 31. This signifies that given any scrambled 8-puzzle it can be unscrambled in less than 32 moves. In the construction of the 2-bit data structure for this Schreier coset graph, we used a special purpose encoding function, assigning to each coset a unique integer in the range $[0, 12 * 8^7 = 25165824]$. This is 100 times larger than optimal, but it can be computed very efficiently.

Once the 2-bit data structure is constructed for the cosets of H in G, it is used to to find a minimal word representing any coset in terms of the generators (moves of the 8-puzzle) of G. Hence to solve a scrambled 8-Puzzle, first represent the given board X as a permutation σ, which interchanges the cells of configuration A (i.e. $A_1 \cup A_2$) with the cells of the configuration in which X lies, according to the scrambled board X, then find a minimal word w in the generators for the coset $H\sigma$. Applying the moves in the word w^R (the reverse of the word w), in order, to the scrambled puzzle will unscramble it.

Pebble Motion on Graphs. The above representation of the 8-puzzle as a group in which a legal move corresponds to one generator of the group, can be generalized to the pebble coordination problem [11]. If the graph has n nodes and if k is the number of configurations of the graph based on the position(s) of the blank node(s). Label the nodes of each configuration consecutively $(1 \ldots, nk)$ starting with the nodes of the configuration (A) corresponding to the goal state. Then each legal move and each scrambled state can be represented as a permutation of $\{1, \ldots, nk\}$. We can find a word in the generators, which represents the initial (scrambled) state by finding an appropriate coset representative of a coset of the group representing the problem in an appropriate point stabilizer subgroup (the subgroup that stabilizes all the cells of configuration A).

References

1. F. Annexstein, M. Baumslag, A.L. Rosenberg, "Group Action Graphs and Parallel Architectures", SIAM J. Computing **19** (1990), pp. 544–569.

4. L. Babai, G. Cooperman, L. Finkelstein, and Á. Seress, "Nearly Linear Time Algorithms for Permutation Groups with a Small Base", *Proc. of the 1991 International Symposium on Symbolic and Algebraic Computation* (ISSAC '91), AMC Press, pp. 200–209, July, 1991.

2. L. Babai, W.M. Kantor and A. Lubotsky, "Small diameter Cayley graphs for finite simple groups", *European Journal of Combinatorics* **10** (1989), pp. 507–522.

3. J-C. Bermond, C. Delome, and J-J. Quisquater, "Strategies for interconnection networks: Some methods from graph theory", *Journal of Parallel and Distributed Computing* **3** (1986), pp. 433–449.

5. L. Campbell, G.E. Carlsson, V. Faber, M.R. Fellows, M.A. Langston, J.W. Moore, A.P. Mullhaupt, and H.B. Sexton, "Dense Symmetric Networks from Linear Groups", preprint.

6. J.J. Cannon, "Construction of Defining Relators for Finite Groups", *Discrete Math.* **5** (1973), pp. 105–129.

7. G.E. Carlson, J.E. Cruthirds, H.B. Sexton, and C.G. Wright, "Interconnection networks based on a generalization of cube-connected cycles", *I.E.E.E. Trans. Comp.* **C-34** (1985), pp. 769–777.

8. D.V Chudnovsky, G.V. Chudnovsky and M.M. Denneau, "Regular Graphs with Small Diameter as Models for Interconnection Networks", *3rd Int. Conf. on Supercomputing*, Boston, May, 1988, pp. 232–239.

9. G. Cooperman and L. Finkelstein, "New Methods for Using Cayley Graphs in Interconnection Networks", *Discrete Applied Mathematics* **37/38** (special issue on Interconnection Networks), 1992, pp. 95–118.

10. M. Frydenberg, A. Riel, N. Sarawagi, Unpublished manuscript.

11. D. Kornhauser, G. Miller and P. Spirakis, "Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups and Applications", *Proc. 25th IEEE FOCS* (1984), pp. 241–250.

12. T. Ohtsuki, "Maze-Running and Line-Search Algorithms", article in *Advances in CAD for VLSI*, **4**, North Holland, Amsterdam (1986).

13. C.C. Sims, "Computation with Permutation Groups", in *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, edited by S.R. Petrick, ACM Press, New York, 1971, pp. 23–28.

14. D. Singmaster, *Notes on Rubik's Magic Cube*, Enslow Publishers, Hillside, N.J., 1981.