New Methods for Using Cayley Graphs in Interconnection Networks

Gene Cooperman and Larry Finkelstein*

College of Computer Science
Northeastern University
360 Huntington Ave.
Boston, Mass. 02115

* The work of this author was supported in part by the National Science Foundation under grant number DCR-8603293.

New Methods for Using Cayley Graphs in Interconnection Networks

Gene Cooperman and Larry Finkelstein

Abstract. A number of researchers have proposed Cayley graphs and Cayley coset graphs as models for interconnection networks. New algorithms are presented for generating Cayley graphs in a more time-efficient manner than was previously possible. Alternatively, a second algorithm is provided for storing Cayley graphs in a space-efficient manner (1–1/2 bits per node), so that copies could be cheaply stored at each node of an interconnection network. The second algorithm is especially useful for providing a compact encoding of an optimal routing table (for example, a 12 kilobyte optimal table for 64,000 nodes). The algorithm relies on using a compact encoding of group elements known from computational group theory. Generalizations of all of the above are presented for Cayley coset graphs.

1. Introduction

There has been a strong interest recently in using Cayley graphs as a model for developing interconnection networks for large interacting arrays of CPU's. [1, 3, 7, 9, 10] Many of the traditional parallel network architectures such as cube-connected cycles (hypercubes) have descriptions as Cayley graphs. [9] Using Cayley graphs and Cayley coset graphs, researchers have also discovered new regular graphs with more nodes for a given diameter than were previously known.

Cayley graphs are directed graphs associated with a permutation group, G, and generators, Φ. The nodes of the graph are elements of G and the edges are labelled by generators in Φ. Cayley graphs are further defined so that if α and β are two nodes connected by a directed edge, (α, β), and the edge is labelled by $\phi \in \Phi$, then $\beta = \alpha\phi$. Finally, Cayley graphs have the property that they are *vertex-symmetric*. This means that for any two nodes, there is a graph isomorphism of the graph into itself which maps the first node into the second. In the case of Cayley graphs, this graph isomorphism also preserves the labels of the edges.

This paper describes a variety of new algorithms and data structures based on Cayley graphs, which are useful both for designing new interconnection networks, and developing routing strategies within those networks. Many of these techniques were developed using the methods of computational group theory. [18]

Two families of algorithms are of particular interest. The first shows how to generate Cayley graphs in a time which has empirically been observed to grow more slowly than the $O(|G|\Phi)$ that would be expected in a naive approach. Further, we find a theoretical characterization of the number of nodes examined in the new method, and show that it is never larger than in the naive approach. Among other uses, the method is ideal for finding the diameter of Cayley graphs efficiently in terms of time.

The second family of algorithms is concerned with space-efficient generation and storage of Cayley graphs. Our technique allows us to store a representation of a Cayley graph for the entire network, using only 1–1/2 bits of memory per node. This is important in two cases. First, if one wishes to use intelligent routing by local nodes in an operating parallel network, it is critical to store a map of the global network (or a large, local region) in as little memory as possible. Since this map would be stored at every node, the memory consumption would be the memory used by the map times the total number of nodes in the network. Second, for large groups the size of real memory, and ultimately the size of virtual memory, limit the ability to efficiently test the properties (such as diameter) of a sequence of candidate Cayley graphs. For this latter case, we have a variation requiring two bits per node to construct a Cayley graph, but needs temporary memory which is msall compared to the two bits per node.

Both of these algorithms implicitly produce a spanning tree for the Cayley graph which can be used as an optimal routing table. This occurs because both develop a spanning tree such that, following a path from a node to the identity, will trace out a minimum word for the permutation in terms of the generators. Since the graph is vertex-symmetric, for any two nodes g and g' the minimal word traced out from ${g'}^{-1}g$ to the identity will also trace out a minimal word from g to g'.

Certain of our techniques generalize to Cayley coset graphs. Many of the parallel network architectures which cannot be described as Cayley graphs, do in fact have natural descriptions as Cayley coset graphs. [1]

Our methods can save further space by using special, compact encodings of group elements. Section 2 discusses some group theoretic preliminaries and a compact encoding of general group elements for permutation groups. This is required for the space-efficient representation of Cayley graphs. Section 3 discusses our faster technique for generating spanning trees for Cayley graphs. In section 4, we present a space-efficient technique which allows the Cayley graph to be implicitly stored using only 1–1/2 bits per node. A new, compact encoding for the cosets of a subgroup of a permutation group is discussed in section 5 and applied in section 6 to generalizing the results of section 3 and 4 to Cayley coset graphs for permutation groups. These methods are especially efficient when the subgroup under consideration is normal.

It should be emphasized while many of the results of this paper are stated for permutation groups, the results of section 3 for fast generation of spanning trees for Cayley graphs only require that one must be able to efficiently compute products in the group. Hence our method works equally well for other representations. In particular, they work well for the representations of groups as matrices over finite fields used by some authors. [7]

2. Enumeration of Elements in Permutation Groups

Let G be a permutation group acting on an n-element set Ω and let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ be an ordering of Ω. The basic problem which we address is to devise a scheme for assigning to each element $g \in G$ a unique integer, denoted *count*(g), in the range 0 to $|G| - 1$, so that *count*(g) can be efficiently computed. This will allow us to store information about elements of a permutation group in an array so that the information associated with a given element $g \in G$ can be efficiently accessed after first computing the index *count*(g). This will have important ramifications in section 4. The method uses standard ideas from computational group theory, which we present in enough detail to make the section self-contained. This material will also be used in section 5 where we develop a more general scheme to enumerate cosets of a subgroup H of G.

Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ be an ordering of Ω. The point stabilizer sequence of G relative to α is the sequence of subgroups

$$G = G^{(1)} \supseteq G^{(2)} \supseteq \ldots \supseteq G^{(n)} = \{1\}$$

where $G^{(i)} = G_{\alpha_1 \alpha_2 \ldots \alpha_{i-1}}$ is the subgroup consisting of all permutations of G which fix each point of the set $\{\alpha_1, \ldots, \alpha_{i-1}\}$.

For $1 \leq i \leq n$, let $U^{(i)}$ be a complete set of right coset representatives for $G^{(i+1)}$ in $G^{(i)}$. We will always assume that the identity belongs to $U^{(i)}$. The set

$$U = \cup_{i=1}^{n-1} U^{(i)}$$

is called a *complete family of coset representatives* for the point stabilizer sequence of G relative to α. The associated cosets will be referred to as the *cosets of the point stabilizer sequence.* Each element of $U^{(i)}$ is in a 1-1 correspondence with the points in the orbit ${\alpha_i}^{G^{(i)}}$, denoted by $\Delta^{(i)}$, in the sense that each element of $U^{(i)}$ maps α_i to a distinct point of $\Delta^{(i)}$.

There are a variety of data structures which may be used to store U and $\Delta^{(i)}, 1 \le i \le n-1$. The choice of data structure depends on the tradeoff between time to access an element and space to store the data structure. A fairly detailed discussion on these matters is presented in [4, 11]. We base our work on the labelled branching data structure (relative to α) originally suggested by Jerrum, [14] but use a variation [12] which is more suitable for our applications. This data structure can be conveniently specified by three n-dimensional arrays, *parent*, τ and τ^{-1}. *parent* is an array of integers which can be viewed as defining a directed forest $\mathcal{B}$ on Ω. *parent*(j) is either `NIL`, in which case α_j is a root of $\mathcal{B}$, or else *parent*$(j) = i$, $i < j$, in which case there is an edge in $\mathcal{B}$ from α_i to α_j. τ is an array of elements of G such that if α_r is the unique root of α_i in G, then ${\alpha_r}^{\tau(i)} = \alpha_i$ and $\tau(i) \in G^{(r)}$. If *parent*$(i) =$ `NIL`, so that α_i is a root of $\mathcal{B}$, then we assume that $\tau(i)$ is the identity. τ^{-1} is a corresponding array with $\tau^{-1}(i) = [\tau(i)]^{-1}$, $1 \le i \le n$. $\Delta^{(i)}$ consists of α_i together with all points α_j such that there is a directed path in $\mathcal{B}$ from α_i to α_j. For $\alpha_j \in \Delta^{(i)}$, $\tau(i)\tau^{-1}(j)$ is the unique element of $U^{(i)}$ which moves α_i to α_j.

The following well-known result is easy to prove and will be used later on.

Lemma 2.1. *Let g and g' have the same values on the first j points $\{\alpha_1, \ldots, \alpha_j\}$. Then $g' \in G^{(j+1)}g$.*

In many of the algorithms in this paper, we have an element $g \in G$ represented as a word (or unevaluated product) in generators of G and need to compute the image of some point of Ω under the action of g. Clearly, this is trivial if we have an explicit representation for g, but we can often save considerable time by working only with the word representation for g. Thus, if g is represented by the word $g = g_1 g_2 \cdots g_k$ and α_i is the point in question, then α_i^g can be evaluated in time $O(k)$ by computing $(\ldots(((\alpha_i)^{g_1})^{g_2})^{\cdots})^{g_k}$. In this case, we say that we are *tracing out* α_i under the word $g_1 g_2 \cdots g_k$ and set $\alpha_i^g \leftarrow trace(\alpha_i, g_1 g_2 \cdots g_k)$.

Those points α_i for which $|\Delta^{(i)}| > 1$ are called *base points*, and the collection of all such points is called a *base*. For convenience, assume that the first m points $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ of Ω form a base for G. This can always be achieved by a base change. [5] The notion of a base for permutation groups is due to Sims. [18] It serves a role analogous to a base in linear algebra, in the sense that two elements of G are equal if and only if they are equal on each point of a base. Accordingly, each element $g \in G$ is uniquely determined by its *base image*, given by $(\omega_1, \ldots, \omega_m) = (\alpha_1^g, \ldots, \alpha_m^g)$.

In terms of the labelled branching data structure, the base points correspond to the interior nodes of $\mathcal{B}$. As a consequence, the length of the longest path in $\mathcal{B}$ from a node backwards to its root is at most m. As an immediate consequence, we obtain the following result.

Lemma 2.2. *Given a labelled branching data structure for G, it is possible to test in time $O(m)$ if a node $\alpha_j \in \Delta^{(i)}$, $i < j$, using the procedure, Pre-Image.*

Procedure Pre-image. *Input:* A possible base image $(\omega_1, \omega_2, \ldots, \omega_m)$ of an element of G. *Output:* Either a word representation in the non-identity elements of τ and τ^{-1} of length at most $2m$ for the unique $g \in G$ with this base image, or `false` if no such element of G exists. *Global Variables:* A labelled branching data structure for G relative to α.

Initialize g to the empty word, representing the identity.

For $j \leftarrow 1$ to m do
  Let k be such that $\alpha_k = \omega_j$
  Set $\alpha_\ell \leftarrow trace(\alpha_k, g^{-1})$
  If $\alpha_\ell \in \Delta^{(j)}$
    [$\tau^{-1}(j)\tau(k) \in G^{(j)}$ and moves α_j to α_k.]
    Set $g \leftarrow \tau^{-1}(j)\tau(k)g$
  Else return(FALSE)
return(g)

Proof: Assume that g has been constructed so that $\alpha_i^g = \omega_i$, $1 \leq i < j$, then any element which has $(\omega_1, \omega_2, \ldots, \omega_m)$ as its base image must in particular lie in the coset $G^{(j)}g$ by Lemma 2.1. In particular, ω_j must be an element of $(\Delta^{(j)})^g = \alpha_j^{G^{(j)}g}$. If this is not the case, then we return `false`. Otherwise, we multiply g on the left by a suitable element of $G^{(j)}$ which moves α_j to Ω_j. The resulting element has the same values on $\alpha_1, \ldots, \alpha_{j-1}$ as the previous element and the correct value on α_j. By induction, this proves correctness of the procedure *Pre-Image*.

In computing the time for *pre-image*, observe that the time for each iteration of the for loop is dominated by the call to *trace* and the check if $\alpha_k \in \Delta^{(j)}$. The latter takes time $O(m)$ by Lemma 2.2, and since the length of g never exceeds $2m$, *trace* takes time $O(m)$ as well. Note that the word representation of g^{-1} is readily available from the one for g, since both are words in the elements of τ and τ^{-1}. Thus *pre-image* takes time $O(m^2)$. □

Assume that the first m points $\{\alpha_1, \ldots, \alpha_m\}$ of Ω form a base for G. We can define a total ordering $\prec_\alpha$ on Ω by $a \prec_\alpha b$ if $a = \alpha_i$, $b = \alpha_j$, and $i < j$. This can be extended to a total ordering on base images of $g \in G$ of the form $(\alpha_1^g, \ldots, \alpha_m^g)$ under the lexicographic ordering induced by $\prec_\alpha$ on Ω.

We can define a function *count* by assigning to $g \in G$ the number of elements $h \in G$ such that $h \prec_\alpha g$. In section 5, we define a function $count_H$ for enumerating the left cosets of H in G, and give an algorithm for computing it. When H is the trivial group, $count_H$ reduces to the function, *count*. However, we can develop an alternative, more efficient method for computing *count*, based on the notion of factoring elements of G with respect to U.

For each i, $1 \leq i \leq m$, let $|\Delta^{(i)}| = r_i$ and number the elements in $\Delta^{(i)}$ from 0 to $r_i - 1$ according to the ordering α. Procedure *Factor* has as input the base image of some element of g, and returns a unique sequence $(b_1, \ldots, b_m)$ where $0 \leq b_i \leq r_i - 1$. If u_i is the unique element of $U^{(i)}$ which moves α_i to the b_i^{th} element of $\Delta^{(i)}$, then g is represented by the unique word $u_m u_{m-1} \cdots u_1$.

Procedure Factor. *Input:* An element $g \in G$. *Output:* The sequence $b = (b_1, b_2, \ldots, b_m)$ defined before. *Global Variables:* A labelled branching data structure for G.

Initialize u to the empty word
For $j \leftarrow 1$ to m do
  Set $\alpha_k \leftarrow trace(\omega_j, u)$
  [The element of G represented by the word gu is in $G^{(j)}$ and moves α_j to α_k.]
  Set b_j so that α_k is the b_j^{th} element of $\Delta^{(j)}$
  Set $u \leftarrow u\tau^{-1}(k)\tau(j)$
  [$\tau^{-1}(k)\tau(j) = (\tau^{-1}(j)\tau(k))^{-1}$, where $\tau^{-1}(j)\tau(k) \in U^{(j)} \subseteq G^{(j)}$.
  So, the element of G represented by gu is now in $G^{(j+1)}$.]
Return($(b_1, b_2, \ldots, b_m)$)

Procedure *Factor* is a simple adaptation of the standard membership algorithm for permutation groups, whose proof of correctness follows from noting that after the m^{th} iteration, the element represented by gu fixes the base points $\alpha_1, \ldots, \alpha_m$, and hence must be the identity. If u_j is a word corresponding to the $\tau(j)\tau(k)$ computed in the algorithm, then $u = u_1^{-1} \cdots u_m^{-1}$ and g may be represented by $u^{-1} = u_m \cdots u_1$. It follows, as in the discussion for *Pre-Image*, that *Factor* takes time $O(m^2)$.

We can now describe how to compute the function *count*. Given $g \in G$, if $factor(g) = (b_1, \ldots, b_m)$, then set

$$count(g) = b_1 + b_2 r_1 + \ldots + b_m r_1 \cdots r_{m-1}.$$

$0 \le b_i \le r_i - 1$ and the well-known group-theoretic result that $\prod_{1 \le i \le m} r_i = |G|$ together imply that *count* assigns to each element of G a unique integer in the range 0 to $|G| - 1$.

Lemma 2.3. *Assume that G has a base of size m and that a labelled branching data structure has been constructed for G. Then there exists a 1-1 function $count: G \mapsto \{0, \ldots, |G| - 1\}$ so that if g is specified by a base image, then $count(g)$ can be computed in time $O(m^2)$.*

If a base is known for a permutation group, then the base image can be trivially computed in time $O(m)$. Hence, *count* could equally well be defined on permutations. The inverse function $count^{-1}$ is also easy to compute, but is not required in this paper. The use of functions similar to *count* are well-known in the literature. [8]

Many interesting permutation groups have small bases (small m), and so the use of *count* is in practice a very effective method for labelling elements of G. For example, every group $PSL(2, q)$ for q a prime power greater than 3 has a permutation representation acting on $q + 1$ points with a base of size 3. Thus, $PSL(2, 31)$ has order 14880, has a (faithful) permutation representation of degree 32, and has a base of size 3.

3. CPU-efficient Cayley graphs

Generating large Cayley graphs efficiently in terms of CPU time is a prime concern for network design. One would like to partially automate the design and simulation phase, by writing a program which will generate a series of Cayley graphs, test each one for properties such as diameter, and possibly store the results of a short simulation. When the number of nodes is less than a million, the amount of physical memory on the computer being used for design or simulation is not usually a limiting factor. In this scenario, the limitation will often be the time it takes to generate each Cayley graph.

In special situations there may also be an advantage for routing in fast generation of Cayley graphs. Fast generation of Cayley graphs allows a local node to dynamically rebuild any local portion of the Cayley graph in an efficient manner. This may have advantages for some intelligent routing schemes.

As a byproduct of the proposed approach for constructing Cayley graphs, we will also obtain both a *canonical rewriting system* [2, 16] for the underlying group, and a minimal word in the generators Φ for each element of the group. The rewriting system $\mathcal{R}$ consists of a set rules of the form $\ell \to r$ where ℓ and r are words in Φ which represent the same element of G and $r \prec \ell$ under a length reductive ordering $\prec$ (to be defined later). The rules in $\mathcal{R}$ can be used to efficiently reduce the word representation $w \in \Phi^*$ of an arbitrary element $g \in G$ to a word of minimal length which also represents g by successively replacing each occurrence in w of the left side of a rule of $\mathcal{R}$ by the right side of that rule. In this way, it can be used to generate optimal routes between arbitrary nodes in the Cayley graph. Unfortunately, except for some special groups, the groups tested in practice tend to have canonical rewriting systems where the number of rules is comparable to the number of elements of G. Hence, the storage requirements would usually be comparable to storing a routing table of all minimal words, which is also a byproduct of this algorithm.

Over the life of a naive algorithm for building a Cayley graph, $|\Phi||G|$ tests of newly discovered nodes would be required to determine if they corresponded to pre-existing ones in the partially generated Cayley graph. Our new algorithm requires only $|G| + |\mathcal{R}| - 1$ tests, where $\mathcal{R}$ is the number of rules in the canonical rewriting system. Although $|\mathcal{R}|$ can attain theoretical values up to $|G||\Phi| - |G| + 1$, in practice (when $|\Phi|$ is not too large) $|\mathcal{R}|$ is usually smaller than $|G|$. Hence, the new algorithm tests no more nodes than the naive algorithm, and usually many fewer nodes. Since the algorithm does not require one to store the rewrite rules, $\mathcal{R}$, the time in practice will still be dominated by the time to test if a newly discovered node already exists in the Cayley graph. $\mathcal{R}$ is discussed only because its size affects the number of such tests executed by the algorithm.

To fix notation, we assume that the nodes of a Cayley graph, a spanning tree of the Cayley graph, and the generators, Φ, are all three stored in some concrete representation. However, we will use the notation Φ^* for the set of words on letters such that the letters correspond to elements of Φ. The concrete representation must be such that each group element has a unique representation, and group multiplication may be computed effectively. Two common examples of such representations are groups of permutations and groups of matrices over a finite field.

We first briefly review the naive algorithm to generate a spanning tree $\mathcal{T}(G)$ for the Cayley graph with generators Φ. This tree will be built with the identity element as the root node. Let $\mathcal{T}_i(G)$ be the list of nodes in the tree $\mathcal{T}(G)$ at level i. For a node $\alpha \in \mathcal{T}(G)$, let $w(\alpha)$ be a minimal word for α in Φ^*. In implementations, we would expect $w(\alpha)$ to be stored as a pointer to a linked list of symbols for the elements of Φ. We will also use $w(\phi)$ for $\phi \in \Phi$ to represent the corresponding word of length one. $\mathcal{T}_0(G)$ is initialized to be a list containing only the identity, e, with $w(e)$ set to the empty word. We will write $\pi(w_1)$ for the concrete representation of a group element corresponding to a word, $w_1 \in \Phi^*$.

The algorithm can be completely described by indicating how to construct $\mathcal{T}_{i+1}(G)$ from $\mathcal{T}_i(G)$. In constructing $\mathcal{T}_{i+1}(G)$, we say that a word $v \in \Phi^*$ of length $i+1$ is *redundant* with respect to an ordering $\prec$, if for some node $\alpha \in \mathcal{T}_j(G)$ with $0 \leq j \leq i+1$, $u = w(\alpha)$, $\pi(v) = \pi(u)$, and $u \prec v$. We can test if v is redundant by using a hash table containing all previously encountered elements of $\mathcal{T}(G)$.

In the following algorithms, we assume a *length-reductive ordering*, $\prec$, on Φ^*. [16] Accordingly, $\prec$ satisfies the following properties:

(i) $\forall A, B, U, V, \quad A \prec B \Longrightarrow UAV \prec UBV$.

(ii) $\forall A, B, \quad |A| < |B| \Longrightarrow A \prec B$, where the notation $|W|$ means the length of W.

The lexicographic plus length ordering (induced by a fixed ordering of Φ) is an example of a length-reductive ordering.

Build-Spanning-Tree: Naive Version
Input: A level i, generators Φ for G and a list of nodes $\mathcal{T}_i(G)$ for level i, with associated words. Assume that $\mathcal{T}_i(G)$ and Φ are ordered (according to $\prec$). *Output:* An ordered list $\mathcal{T}_{i+1}(G)$ of nodes for level $i+1$, with associated words.

If $\mathcal{T}_i(G) = nil$ then return
Set $\mathcal{T}_{i+1}(G) = nil$
For each node $\alpha \in \mathcal{T}_i(G)$ do
  For each $\phi \in \Phi$ do

If $\alpha\phi$ is not redundant, then

Append a new node β to $\mathcal{T}_{i+1}(G)$ with

$\beta = \alpha\phi$ and $w(\beta) = w(\alpha)w(\phi)$

Building $\mathcal{T}(G)$ using this version of the algorithm requires examining $|\Phi||G|$ nodes to test redundancy. We could also form a rewrite rule for each redundant word $w(\alpha)w(\phi)$ by setting $w(\alpha)W(\phi) \to w(\beta)$ where $\beta = \alpha\phi$ for some node $\beta \in \mathcal{T}_j(G)$, $j \leq i+1$. Such a rule system reduces each word of Φ^* to minimal form, and is confluent, but is redundant in the sense that many left hand sides of rules will be reducible by other rules in the system. Hence such a set of rewrite rules is not canonical.

The key step to finding a more efficient algorithm is deriving a subset of candidate words $S \subseteq w(\mathcal{T}_i(G))w(\Phi)$, all of length $i+1$. Rather than construct all combinations, these candidates for $\mathcal{T}_{i+1}(G)$ are guaranteed to produce all of $\mathcal{T}_{i+1}(G)$ plus all rules of a canonical system $\mathcal{R}$ and nothing more.

We choose S to consist of all words $w \in \Phi^*$ of length $i+1$ with the property that $w = aUb$, where $aU = w(\alpha)$ and $Ub = w(\beta)$ for nodes $\alpha, \beta \in \mathcal{T}_i(G)$. One may think of the words of S as being formed from maximal *overlaps* from the words associated with nodes of $\mathcal{T}_i(G)$. In constructing $\mathcal{T}_1$, we allow empty overlaps to occur. This allows the following straightforward recursive version of the algorithm.

Build-Spanning-Tree: New Version

Input: A level i, and a list of nodes $\mathcal{T}_i(G)$ for level i, with associated words. Assume that $\mathcal{T}_i(G)$ and Φ are ordered (according to $\prec$). *Output:* An ordered list $\mathcal{T}_{i+1}(G)$ of nodes for level $i+1$, with associated words, and a set of final rules $\mathcal{R}_{i+1}$ whose left hand side has length $i+1$.

If $\mathcal{T}_i(G) = nil$ then return

Let $S = \{aUb\colon \quad aU = w(\alpha),\, Ub = w(\beta),\, \alpha, \beta \in \mathcal{T}_i(G),\ a, b \in w(\Phi)\}$

For each $w \in S$ do

If $\pi(w) = \pi(v)$, where $v = w(\alpha)$, $\alpha \in \mathcal{T}_j(G)$, $j \leq i$ then

[(Optional) Create a new rule.]

Append $w \to v$ to $\mathcal{R}_i$

Else

[Create a new node.]

Append a new node β to $\mathcal{T}_{i+1}(G)$ with

$\beta = \pi(aU)\pi(b)$ and $w(\beta) = aUb$

Let $\mathcal{R}$ be the final set of rules formed from applying the Build-Spanning-Tree routine.

Theorem 3.1. *The new version of Build-Spanning-Tree computes $\mathcal{T}(G)$ and a final (canonical) set of rules $\mathcal{R}$ for G by examining $|G| - 1 + |\mathcal{R}|$ words of Φ^*.*

Proof: It suffices to prove correctness of the algorithm. It is clear from construction, that for each node $\alpha \in \mathcal{T}(G)$, $w(\alpha)$ is the smallest word under $\prec$ which represents α. Hence, the words of $\mathcal{T}(G)$ (i.e. words of the form $w(\alpha)$ for $\alpha \in \mathcal{T}(G)$) are irreducible under the rules of $\mathcal{R}$ and represent distinct elements of G.

In order to show that $\mathcal{T}(G)$ is complete, i.e., contains $|G|$ elements, it suffices to show that each word $v \in \Phi^*$ is reducible, under $\mathcal{R}$, to a word in $\mathcal{T}(G)$. If this is not the case, let v be the smallest word of Φ^* under $\prec$ which is not reducible under $\mathcal{R}$ to a word in $\mathcal{T}(G)$. Let $v = aUb$ where $a, b \in w(\Phi)$. Since v is irreducible under $\mathcal{R}$, so are aU and Ub. By the assumption, aU and Ub are then of the form $aU = w(\alpha)$ and $Ub = w(\beta)$, $\alpha, \beta \in \mathcal{T}(G)$. At this point, if $\pi(v) = \gamma$ for some node $\gamma \in \mathcal{T}(G)$ with $w(\gamma) \prec v$, then the algorithm would introduce a rule into $\mathcal{R}$ of the form $v \to w(\gamma)$, which contradicts the assumption about v. Otherwise, the algorithm would create a node $\delta \in \mathcal{T}(G)$ with $v = w(\delta)$. This again leads to a contradiction.

It remains to show that the rules of $\mathcal{R}$ are interreduced. However, the left side of each rule is a word in $\mathcal{T}(G)$, hence irreducible under $\mathcal{R}$, and the left side of a rule is a word of the form aUb where aU and Ub are words in $\mathcal{T}(G)$, $a, b \in W(\Phi)$. If aUb is reducible under $\mathcal{R}$ (excluding the current rule), then it must be reduced by a rule with left side having length at most $|aUb| - 1$. But then either aU or Ub is reducible by the same rule and this gives a contradiction. □

An interesting corollary to Theorem 3.1 is the fact that if $\ell \to r \in \mathcal{R}$, then $|\ell| - |r| \leq 2$. To see this note that ℓ has the form aUb where aU and Ub are elements of $\mathcal{T}_i(G)$ for some $i \geq 1$. Let $r \in \mathcal{T}_j(G)$, $j \leq i$. If $j < i - 1$, then aU and rb^{-1} represent the same element of G, with $|rb^{-1}| < |aU|$. But then aU is redundant, which is a contradiction, and so $i - 1 \leq |r| \leq i + 1$. This fact is already noted in [16].

We next sketch an efficient algorithm for the construction of the candidate set of words, S, in the second line of Build-Spanning-Tree. Assume that $\mathcal{T}(G)$ is constructed as a tree with each node α having 3 fields: a backwards pointer to its parent, a list of pointers to its children (ordered from left to right according to $\prec$) and a data field which holds an element of $\phi \in \Phi$ so that $w(\alpha) = w(parent(\alpha))w(\phi)$. Then it is clear how to construct S for level $i + 1$ in Build-Spanning-Tree. Let $\mathcal{T}(G)_i$ be the nodes at level i listed in order. For each node $\alpha \in \mathcal{T}(G)_i$, at level i compute $w(\alpha) = aU$ by following α back to the root. To find if there exists a node $\beta \in \mathcal{T}(G)$ with $w(\beta) = Ub$, for some $b \in w(\Phi)$, simply follow $\mathcal{T}(G)$ down from the root using U as the guide for selecting the next child, if any. Either a node γ at level $i - 1$ will be found with $w(\gamma) = U$ or else the search will fail. In the former case, any child of γ will serve as β. By a clever use of data structures, possibly at the expense of additional storage, one can make the timing of this part of the algorithm independent of the size of Φ.

For the next theorem concerning time complexity, we now restrict our attention to the specific representation of a group as permutation. Recall that the definition of a *base* for a permutation group in section 2 allows us to multiply a permutation by a generator on the right in time proportional to the size of the base. This assumes that a full action of each of the generators is stored, but that for other permutations, only the action on base points is stored.

Theorem 3.2. *The time complexity for the new version of Build-Spanning-Tree is $O(|G|(h + 2d + m) + |\mathcal{R}|(h + 1)$, where $|\mathcal{R}|$ is the number of canonical rules, $|G|$ is the size of the group, m is the size of the base for the permutation group, h is the time to hash on the permutation, and d is the depth of the Cayley tree searched (or the diameter of the corresponding Cayley graph).*

Proof: Theorem 1 shows that only $|G| + |\mathcal{R}|$ nodes are examined. Each time we examine a new node, we have to test for redundancy and this accounts for $(|G| + |\mathcal{R}|)h$ work. In constructing the candidate words S, we have to travel backwards from a node in the tree to the root to discover its word representation (aU in the previous algorithm) and find all nodes of the tree with word representation Ub. At most $2d - 1$ work is is required to find if there is a node with word representation U. The total amount of additional work over all nodes in the tree is bounded by the number of nodes examined, $|G| + |\mathcal{R}|$. The time to perform this phase, is proportional to $|G|2d + |\mathcal{R}|$. Finally, one permutation multiplication is required for each node of $\mathcal{T}(G)$. For each node α, the multiplication is always of the form $\alpha\phi$ for $\phi \in \Phi$. If only the permutation action on the base points is stored for the nodes of $\mathcal{T}(G)$, then the multiplication will cost m per node. The result now follows by adding up the individual terms. □

Experiments were carried out to determine the number of nodes typically examined in constructing a Cayley graph for A_7 (the group of even numbers of transpositions on seven elements) as the number of random generators increased. Both the random generators and their inverses were included in Φ. For example, 100 generators represents 50 random generators and their inverses. CPU times for the new version were never more than a few minutes using LISP on a SUN-3.

number of generators	diameter	nodes examined (new version)	nodes examined (naive version)
6	7	6,379	15,120
10	5	9,897	25,200
20	4	18,625	50,400
100	3	65,853	252,000

In the preceding discussion, we have described redundancy checks in terms of hashing on the permutation of a newly discovered node. There is an alternative to hashing which is much more space-efficient, and can be executed reasonably fast. This is the use of bases to assign a unique integer from 0 to $|G|-1$ for each permutation of G as described in section 2. Its CPU time will be dominated by m additions and multiplications of integers, where m is the size of the base for G. The space required would be just one bit per node of the Cayley graph.

It should also be remarked that the algorithm for generating the Cayley graph does not depend on the use of inverses. Hence, it would work equally well for the more general analogue of a Cayley graph, based on monoids rather than groups.

Theorem 3.3. *A group G with generators Φ and length reducing ordering $\prec$ has a canonical rewriting system $\mathcal{R}$ with at most $|G||\Phi| - |G| + 1$ rules.*

Proof: It is clear that Build-Spanning-Tree examines at most $|G||\Phi| + 1$ words. (The "+1" comes from examining the initial node.) Since Theorem 1 states that it examines $|G| - 1 + |\mathcal{R}|$ nodes, we have that $|G| + |\mathcal{R}| < |G||\Phi| + 1$, from which the upper bound follows immediately. □

Remark. As a byproduct of our result, we found an upper bound of $|G||\Phi| - |G| + 1$ for the number of rules, $\mathcal{R}$, of a canonical system, which is a sharper bound than those previously known. Le Chenadec [16] reports $2|G||\Phi_{\mathrm{LeChenadec}}|$ where G is a group, and $\Phi = 2\Phi_{\mathrm{LeChenadec}}$. Our Φ is twice as large because we consider monoids in which ϕ and ϕ^{-1} are considered distinct generators. In our notation, Le Chenadec's formula would be $|G||\Phi|$. Le Chenadec's proof was given for finite groups, while our formula also applies to finite monoids.

The Build-Spanning-Tree algorithm provides routing information in the form of both minimal words. There has been much previous work on the minimum word problem. Two of the more important approaches for groups are the methods of Knuth-Bendix [15] for general term algebras and Todd-Coxeter coset enumeration [2] for groups and cosets, but the new method is independent of knowing a set of defining equations, and appears to be more efficient for the common case when $\mathcal{R}$ is large.

4. Space-Efficient Cayley graphs

Storing and accessing Cayley graphs efficiently in terms of space is a prime concern in two situations. First, it is important if one intends to store a representation of the entire Cayley graph at each local node of a parallel network of CPU's. In an intelligent routing scheme in which the local node chooses routes based on current congestion or failed links, the ability to examine the entire Cayley graph is a valuable feature. Second, for large designs the amount of physical memory on the computer used for design or simulation is often a constraint. This is the case when building networks using Cayley graphs with several million nodes. More physical memory is expensive, and the use of virtual memory would greatly slow down the simulation. A benefit of this technique is that it is trivial to compute optimal routes between arbitrary nodes.

We describe a method for storing a directed Cayley tree (a minimum spanning tree for the Cayley graph) using only 2 bits per node. 2 bits per node appears to be the optimal space requirement for generating the Cayley graph. Once generated, however, the graph can be stored using 1–1/2 bits per node. The data structure depends on the existence of a permtuation representation for G on an n-element set Ω, in which the first m points form a base for G, together with a labelled branching data structure for G. We can then use the function *count* constructed in section 2 define an encoding of G into the numbers 0 to $|G|-1$. The 2-bit data structure can then be described as follows. Allocate a bit vector of length $2|G|$, referred to as the 2-bit data structure and associate with each pair of bits a unique address from 0 to $|G|-1$. If $count(g) = i$, then we store in address i, the distance modulo 4 in the Cayley graph from g to the identity node. This distance can be readily computed from a spnning tree for the Cayley graph with *root* the identity element, which we will refer to as the Cayley tree.

The next result proves that the standard operations on the Cayley graph can be efficiently performed with this data structure.

Lemma 4.1. *Assume that we have constructed both a labelled branching for G and the two bit data structure. Let $g \in G$ be specified by a base image. Then we can perform the following computations.*

(i) Find all neighbors of g in the Cayley graph in time $O(m^2|\Phi|)$. In particular, this includes finding the *parent* node and all children nodes in the Cayley tree.

(ii) Find the minimum word representation for g in time $O(m^2|\Phi|d)$, where d is the diameter of the Cayley graph.

(iii) Find a shortest route between nodes g and h in the Cayley graph in time $O(m^2|\Phi|d)$.

Proof: Given node g of the Cayley graph, let g have base image $\omega = (\alpha_1^g, \dots, \alpha_m^g)$. Then the neighbors of g have base images $\omega^\phi, \phi \in \Phi$, and hence are uniquely determined. Now, since *count* has input the base image of an element of G, we can compute *count* for g and all its neighbors. *count* provides an index into the 2-bit data structure, and thus for g and all its neighbors, we can compute the distance modulo 4 from the root node. In particular, if g has distance s modulo 4, then the parent of g in the Cayley tree is the unique neighbor with distance $s-1$ modulo 4 and the children are the unique neighbors with distances $s+1$ modulo 4. The remaining neighbors are siblings at the same distance form the root node. The time for the computation is dominated by the $|\Phi|$ calls to *count*. Since each call takes time $O(m^2)$ by Lemma 2.3, the time to compute the neighbors, including the parent and children nodes of g in the Cayley tree is $O(m^2|\Phi|)$.

Part (ii) essentially follows from (i). Given node g, we can find in time $O(m^2|\Phi|)$ the parent h of g in the Cayley tree. If $h\phi = g$, $\phi \in \Phi$, then by recursion, if w is the word along the path from the root of the Cayley tree to h, then $w\phi$ is a word of minimal length representing g. If node g has distance s from the root, then it takes $O(m^2|\Phi||s|)$ invocations of *count* to find the parent of each node in the path from the *root* to g. Since $s \leq d$, the proof of (ii) follos immediately.

Suppose now that we are given nodes g and h and want to compute the minimal path between g and h. Assume that we have base images for both g and h. Using *pre-image*, we may obtain in time $O(m^2)$, a word of length at most $2m$ in the non identity elements of τ and τ^{-1} which represents h. This immediately gives a word of length $2m$ over the same set of elements which represents h^{-1}.

At any rate, in time $O(m^2)$, we can find the base image of gh^{-1}. A shortest path between nodes g and h may then be obtained by finding a word of minimal length in Φ which represents gh^{-1}. This, in turn can be found by part (ii) in the required time bound. ∎

Theorem 4.1. *The 2-bit data structure described above can be constructed within the space limitations of 2 bits per node.*

We construct the data structure using a repeated backtracking algorithm with a cutoff depth, while setting the depth successively higher. We will refer to the level of a node as the distance from the node to the identity node. At each level, we will need to make 4 passes over the nodes at that level. If the graph has been filled in to level i, then we can use a backtracking algorithm to access all nodes that are exactly at level i. Expand the initial identity node to find all of its neighboring nodes. Keep only those nodes at the next level. This can be determined since the 2-bit pattern for the next level will be exactly 1 more than the last, modulo 4. Continue by expanding a node at the next level, until we reach level i. Then use a standard backtracking algorithm to examine the remaining nodes. We will refer to this process as a depth-first search with cutoff depth i.

The construction can be summarized by the following graph.

nodes at	2-bit pattern stored (modulo 4)				
level:	Pass 0	Pass 1	Pass 2	Pass 3	Pass 4
$i-1$	$i-1$	i	$i-1$	$i-1$	$i-1$
i	i	i	i	$i+1$	i
$i+1$	junk	i	i	$i+1$	$i+1$

During each of the four passes, we carry out a depth-first search with a cutoff depth of $i-1$ or i. On the first pass, we search to depth i and set all nodes at level $i-1$ and i, and all neighbors of nodes at level i to the 2-bit pattern i mod 4. On the second pass, we search to depth $i-1$, and set all nodes at level $i-1$ to the pattern $i-1$ mod 4. (On this pass, nodes at level $i-1$ will be recognized by the initial pattern i mod 4 instead of $i-1$ mod 4.) On the third pass, we search to depth i and set all nodes at level i and all neighbors with a pattern of i mod 4 to the new pattern $i+1$ mod 4. Finally, on the fourth pass, we search to depth i, and set the nodes at level i to the pattern i mod 4. (Nodes at level i will be recognized by the pattern $i+1$ mode 4, rather than i mod 4.) ∎

The algorithm can take up to $4|G||\Phi|(m^2+m)d$ time. m determines the time to compute the m base points of the next permutation by a permutation multiplication. m^2 is the time to compute the compact encoding of a permutation, which is used to index into the array of 2-bit patterns. The factor of d is due to the d levels to be examined. Because of the typically exponential increase in the widths of Cayley graphs for the early levels, usually we will have $d < c \log |G| / \log |\Phi|$, where c is a small constant of the order of 2 or 3.

The data structure can be constructed more quickly at the cost of temporarily using $|G| \log_2 d$ space. The construction is simple. We allocate a vector with $\log_2 d$ bits per node, and store the true diameter. We initialize all nodes to 0, and then use iterative deepening. This eliminates the factor of 4. In addition, more powerful heuristics for cutting down the factor d can then be made available.

Description of 1–1/2 Bit Data Structure. The construction of the 2-bit data structure can be modified by storing all entries, mod 3, instead of mod 4. All of the properties of the 2-bit data structure are satisfied by the 1–1/2 bit data structure. The 1–1/2 bit data structure can be constructed by first constructing the 2-bit data structure, and then searching the Cayley tree to replace all numbers, mod 4, by numbers, mod 3. An obvious data compression algorithm can then be applied such as storing 3 adjacent nodes containing the numbers mod 3 by using 5 bits.

5. Enumeration of Cosets in Permutation Groups.

In section 2, we described a method for constructing a function *count* on a permutation group G which assigns to each $g \in G$ a unique integer in the range 0 to $|G| - 1$ such that $count(g)$ can be efficiently computed. This method was exploited in the algorithms presented for constructing Cayley graphs. In this section, we are primarily concerned with a more general problem. For a subgroup $H \subseteq G$, a *right coset* of H in G is a set of the form $Hg = \{hg\colon\ h \in H\}$ for some $g \in G$. Our problem is to construct a function, $count_H$, which will assign to each right coset of H a unique integer in the range 0 to $[G : H] - 1$ with the property that $count_H$ can be efficiently computed. A *representative* of a coset Hg is a group element $g' \in Hg$. The argument of $count_H$ will be specified by an arbitrary representative of the coset of H.

We will define the function $count_H$, below, on left cosets. A *left coset* of H in G is a set of the form $gH = \{gh\colon\ h \in H\}$ for some $g \in G$. We can then uniquely extend $count_H$ to a function on right cosets, using the formula $count_H(Hg) = count_H(g^{-1}H)$. This is well-defined because for $g' \in Hg$, $(g')^{-1} \in (Hg)^{-1} = g^{-1}H^{-1} = g^{-1}H$. So, if $Hg = Hg'$, then $count_H(Hg) = count_H(Hg')$.

As in section 2, assume that the first m points $\{\alpha_1, \ldots, \alpha_m\}$ of Ω form a base for G. Recall that $\prec_\alpha$ is the total ordering on Ω defined by $a \prec_\alpha b$ if $a = \alpha_i$, $b = \alpha_j$, and $i < j$. This can be extended to a total ordering on base images of $g \in G$ of the form $(\alpha_1^g, \ldots, \alpha_m^g)$ under the lexicographic ordering induced by $\prec_\alpha$ on Ω.

We begin by giving a brief description of Richardson's method [17] for obtaining a canonical base image for a right coset Hg. This is given by defining a function $\rho(Hg)$ which assigns to each coset Hg the minimal base image for all elements in Hg under $\prec_\alpha$.

Function ρ**.** *Input:* A coset Hg of H in G specified by g. *Output:* The minimal base image of an element of Hg under $\prec_\alpha$. *Global Variables:* Data structures to hold the cosets for the point stabilizer sequence for G and H relative to α. *Local Variables:* A sequence $(\omega_1, \ldots, \omega_m)$ to hold the base image.

Initialize γ to g
For $i \leftarrow 1$ to m do
  Let $k = \min\{j\colon\ \alpha_j \in \alpha_i^{H^{(i)}\gamma}\}$ and set $\omega_i = \alpha_k$
  Let $h \in H^{(i)}$ such that $\omega_i = \alpha_i^{h\gamma}$
  Set $\gamma \leftarrow h\gamma$
Return$(\omega_1, \ldots, \omega_m)$

It is straightforward to show that the computation of ρ takes time $O(mn)$. Since each right coset Hg can be naturally identified with the left coset $g^{-1}H$, this provides a method for enumerating both right and left cosets. The use of ρ to enumerate (right) cosets of H, appears to be most practical when there is sufficient memory. This method has been successfully exploited by several authors [6,8], and we refer the reader to these articles for a discussion of implementation issues. In order to exploit the compact scheme described in section 3, for storing spanning trees for Cayley graphs and Cayley coset graphs, it is necessary be able to efficiently assign to each coset of H a unique integer in the range 0 to $[G : H] - 1$. A natural way to try to extend the method just described is to define a function $count_H$ which assigns to each coset Hg, the number of cosets Hg' with the property that $\rho(Hg') \prec_\alpha \rho(Hg)$. Since this number does not appear easy to compute, we shall concentrate instead on an alternative method for assigning a canonical base image to cosets of H which is more expensive to compute than ρ but allows for fast computation of $count_H$.

For $g \in G$, we define $\lambda(gH)$ to be the base image of the element $g' \in gH$ which is minimal under $\prec_\alpha$. The following algorithm can be used to compute λ. A key step is the use of a *change of base algorithm* to obtain $H_{\omega_1,\ldots,\omega_i}$ from $H_{\omega_1,\ldots,\omega_{i-1}}$, starting with $i = 1$. This may be accomplished using a fast new base change algorithm [5, 13] which substantially contributes to the practicality of this method.

Function λ. *Input:* A left coset gH specified by $g \in G$. *Output:* The minimal base image, $(\omega_1, \ldots, \omega_m)$, of an element of gH under $\prec_\alpha$. *Global Variables:* Data structures to hold the cosets for the point stabilizer sequence for G and H relative to α. *Local Variables:* A sequence $(\omega_1, \ldots, \omega_m)$ to hold the base image.

Set $\gamma \leftarrow g$
For $i \leftarrow 1$ to m do
  Let $k = \min\{j\colon\ \alpha_j \in \alpha_i^{\gamma H_{\omega_1,\ldots,\omega_{i-1}}}\}$ and set $\omega_i = \alpha_k$
  [Computing $H_{\omega_1,\ldots,\omega_{i-1}}$ requires the use of a change of base algorithm.]
  Let $h \in H_{\omega_1,\ldots,\omega_{i-1}}$ such that $\omega_i = \alpha_i^{\gamma h}$.
  Set $\gamma \leftarrow \gamma h$
Return$(\omega_1, \ldots, \omega_m)$

Lemma 5.1. *It costs $O(mn^2)$ time to compute $\lambda(g)$.*

Proof: It costs $O(n^2)$ time to compute $H_{\omega_1,\ldots,\omega_{i-1}}$ from $H_{\omega_1,\ldots,\omega_{i-2}}$, $i \geq 2$, using a cyclic base change [5]. This dominates the running time for each iteration of the "for" loop and so the computation of λ takes $O(mn^2)$ time. □

We are now ready to present the construction of $count_H$ as described earlier. As before, set $count_H(gH)$ to be the number of cosets $g'H$ of H in G, such that $\lambda(g'H) \prec_\alpha \lambda(gH)$. We can compute $count_H$ using the following algorithm.

Function $count_H$. *Input:* An element $g \in G$. *Output:* $count_H(gH)$. *Local Variables:* A sequence $\omega = (\omega_1, \ldots, \omega_m)$ to hold the base image of $\lambda(g)$.

Set $count_H \leftarrow 0$, $\gamma \leftarrow g$ and $(\omega_1, \ldots, \omega_m) \leftarrow \lambda(gH)$
For $i \leftarrow 1$ to m do
  Let $\mathcal{O}$ be the set of $H_{\omega_1,\ldots,\omega_{i-1}}$ orbits on $\alpha_i^{G^{(i)}\gamma}$,
    $\mathcal{O}' = \{\Gamma \in \mathcal{O}\colon\ \exists \alpha_j \in \Gamma, \alpha_j \prec_\alpha \omega_i\}$, and
    $\Delta = \cup_{\Gamma \in \mathcal{O}'} \Gamma$
      [Since $G^{(i)}\gamma H_{\omega_1,\ldots,\omega_{i-1}} = G^{(i)}\gamma$, $H_{\omega_1,\ldots,\omega_{i-1}}$ setwise stabilizes $\alpha_i^{G^{(i)}\gamma}$.]
  Add $|\Delta||G^{(i+1)}|/|H_{\omega_1,\ldots,\omega_{i-1}}|$ to $count_H$
  Let $h \in H_{\omega_1,\ldots,\omega_{i-1}}$ such that $\omega_i = \alpha_i^{\gamma h}$.
  Set $\gamma \leftarrow \gamma h$
Return$(count_H)$

Lemma 5.2. *The procedure for computing $count_H$ is correct and takes time $O(mn^2)$.*

Proof: If $\lambda(gH) = (\omega_1, \ldots, \omega_m)$, then let $\lambda_i(gH) = \omega_i$. In order to prove correctness, it suffices to show that for i in the range $1 \leq i \leq m$ and $g \in G$, if

$$\mathcal{K} = \{g'H\colon\ g' \in G, \lambda_k(g'H) = \omega_k, 1 \leq k \leq i-1, \text{ and } \lambda_i(g'H) \prec_\alpha \omega_i\},$$

then $|\mathcal{K}| = |\Delta||G^{(i)}|/|H_{\omega_1,\ldots,\omega_{i-1}}|$, with Δ and $(\omega_1, \ldots, \omega_{i-1})$ as in the algorithm.

Assume that $\gamma \in gH$ such that $\alpha_k^\gamma = \omega_k$, $1 \le k \le i$. This corresponds to the value of γ in the algorithm during the i^{th} iteration. Clearly, $\mathcal{K}$ is in 1-1 correspondence with the set

$$\mathcal{L} = \{g' H_{\omega_1,\ldots,\omega_{i-1}} \colon\ g' \in G^{(i)}\gamma, \alpha_k^{g'} = \omega_k, 1 \le k < i, \text{ and } \alpha_i^{g'} \prec_\alpha \omega_i\}.$$

For each $\delta \in \Delta$, let g_δ be an element of $G^{(i)}\gamma$ which moves i to δ and let

$$\mathcal{M} = \cup_{\delta\in\Delta}\ G^{(i+1)} g_\delta.$$

It is easy to verify that

$$\mathcal{M} = \cup\{g' H_{\omega_1,\ldots,\omega_{i-1}} \colon g' H_{\omega_1,\ldots,\omega_{i-1}} \in \mathcal{L}\}.$$

So, $|\mathcal{M}| = |\Delta||G^{(i+1)}|$ and $|\mathcal{K}| = |\mathcal{L}| = |\mathcal{M}|/|H_{\omega_1,\ldots,\omega_{i-1}}|$, which yields the correct count.

The time for computing $count_H$ is dominated by the time to compute $H_{\omega_1,\ldots,\omega_{i-1}}$ from $H_{\omega_1,\ldots,\omega_{i-2}}$, $2 \le i \le m$. This requires $m-1$ cyclic base changes, each of which takes time $O(n^2)$ time. Thus $count_H$ takes time $O(mn^2)$. □

Corollary 5.3. *$count_H$ can be uniquely extended to right cosets, and it can be computed in time $O(mn^2)$.*

The proof is by use of the formula $count_H(Hg) = count_H(g^{-1}H)$, which was shown to be well-defined at the beginning of the section.

Remark. The functions λ and $count_H$ can be combined into one single function. We have presented separate versions for clarity.

A subgroup H of G is *normal* in G if $Hg = gH$ for all $g \in G$. In this case, we can substantially improve the computation of $count_H$ due to the following observations.

First, If H is normal in G, then $\rho(Hg) = \lambda(gH)$, and so $\lambda(Hg)$ can be computed in time $O(mn)$. Second, for the computations of this section to be effective, it has been implicitly assumed the groups G and H are described by strong generating sets with respect to the ordering of Ω, $(\alpha_1, \ldots, \alpha_n)$. (See [18] for definition of strong generating set.) Strong generators for an ordering can always be computed from any set of generators for a group, and since the computation is done only once, the time will be small compared to repeated calls to $count_H$. If $\gamma \in gH$ has a base image $w = \{\omega_1, \ldots, \omega_m\}$, then conjugating by g the strong generators for H with respect to the ordering $(\alpha_1, \ldots, \alpha_n)$ will yield a strong generating set for H with respect to the ordering $(\alpha_1^\gamma, \ldots, \alpha_n^\gamma) = (\omega_1, \ldots, \omega_m, \alpha_{m+1}^g, \ldots, \alpha_n^g)$. One can always construct a strong generating set of size at most $O(|G|) \le O(m \log n)$. [11] Hence the time to conjugate the generators is at most $O(m \log n)$. From the new strong generating set, a complete family of coset representatives for the point stabilizer chain of H can be constructed in time $O(|S|n) \le O(nm \log n)$ using a recent result. [11] Thus all coset representatives h and orbits $\mathcal{O}$ to compute $count_H$ can be computed in time $O(nm \log n)$, and $count_H$ can be computed in time $O(nm \log n)$ for normal subgroups, H.

6. Generalizations to Cayley Coset Graphs

Some authors have worked with the more general Cayley coset graphs. [1] It is well-known that multiplication of a right coset by a group element on the right is well-defined, since $(Hg_1)g_2 = H(g_1 g_2)$. Hence, one may define a *Cayley coset graph* $\mathcal{C}_H$, as a directed graph associated with a group G, a subgroup H, and generating set Φ for G. The nodes of the graph are right cosets of H in G, and the edges are labelled by generators in Φ. There is a directed edge between two nodes Hk and $H\ell$ of $\mathcal{C}_H$, labelled by $\phi \in \Phi$, if $Hk\phi = H\ell$.

As an example of a Cayley coset graph, suppose the ordering α on Ω is the trivial ordering and set $G = S_n$, $H = G^{(2)}$ and $\Phi = \{(1,2),(2,3),\ldots,(n-1,n)\}$. By Lemma 2.1, the coset $G^{(2)}g$ consists of all points of Ω which move 1 to 1^g. In this way, each coset can be identified with a unique point of Ω. Clearly, if $\phi \in \Phi$, then $G^{(2)}g\phi$ is identified with the point $1^{g\phi}$. In particular, if ϕ fixes 1^g, then ϕ is the label of a directed edge taking $G^{(2)}g = Hg$ to itself. In this case, the Cayley coset graph can be visualized as a straight line graph on n points with each node having $n-3$ loops. An immediate observation is that Cayley coset graphs are not in general vertex symmetric.

It is easy to show that when H is normal in G, then $\mathcal{C}_H$ is vertex symmetric. In fact, we can define a homomorphism $\mu: G \mapsto Aut(\mathcal{C}_H)$ where the image of Hk under $\mu(g)$, denoted by $Hk^{\mu(g)}$, is the coset $g^{-1}Hk = Hg-^{-1}k$. If there is a directed edge between Hk and $H\ell$ labelled by ϕ (so that $Hk\phi = H\ell$), then $g^{-1}Hk\phi = Hg^{-1}k\phi = Hg^{-1}\ell = g^{-1}H\ell$ implies that there is a directed edge between $Hk^{\mu(g)}$ and $H\ell^{\mu(g)}$ labelled by ϕ. Hence $\mu(g)$ is an automorphism of $\mathcal{C}_H$. Furthermore, μ is a homomorphism on G whose kernel is H.

The algorithm, *Build-Spanning-Tree*, of section 3 can be generalized to Cayley coset graphs, providing that H is normal in G. Using the notation of that section, if $\mathcal{T}(\bar{G})$ is the tree which is constructed and α is a node of $\mathcal{T}(\bar{G})$, then $w(\alpha)$ will be a minimal word in Φ under $\prec$ which represents an element of the coset $H\alpha$. In addition, we can construct a canonical set of rules $\bar{\mathcal{R}}$ for reducing any word of Φ^* to a word in $\mathcal{T}(\bar{G})$. This is equivalent to obtaining a canonical set of rules in the generating set $\bar{\Phi} = \{H\phi : \phi \in \Phi\}$ for $\bar{G}$ with length-reductive ordering inherited from $\prec$. The reason that these rules work (and why Theorem 3.1 admits such a generalization) is that if $\ell \to r \in \bar{\mathcal{R}}$, so that $Hw(\ell) = Hw(\alpha)$ and $v = u_1\ell u_2$ is an arbitrary word, then $Hw(v) = Hw(u_1)Hw(\ell)Hw(u_2) = Hw(u_1)Hw(r)Hw(u_2) = Hw(u_1ru_2)$. That is, we can reduce word representations for coset representatives of H in the usual manner. We can use the functions ρ defined in section 5 to build a hashing scheme for testing redundancy as in section 3.

The space-efficient Cayley graphs of section 4 have a clear generalization to Cayley coset graphs. The only distinctions are that a node is denoted by the action of a representative of the coset on the base points of G, and *count* must be replaced by $count_H$ when using the compact encoding. The formal result is as follows.

Theorem 6.1. *Assume that we have constructed labelled branchings for both G and $H \subseteq G$, and the two bit data structure for $\mathcal{C}_H$. Assume further that H is normal in G. Let $g \in G$ be specified by a base image. Then we can:*

(i) Find all neighbors of Hg in time $O(mn(\log n)|\Phi|)$. In particular, this includes finding the *parent* node and all children nodes.

(ii) Find the minimum word representation for an element of Hg in time $O(mn(\log n)|\Phi|d)$, where d is the diameter of the Cayley graph.

(iii) Find an minimal length path between nodes Hg and Hh in time $O(mn(\log n)|\Phi|d)$.

For non-normal subgroups, we are forced to have a more restricted analogue of Theorem 4.1(iii) due to the fact that $\mathcal{C}_H$ is not in general vertex symmetric. Only for normal subgroups H can we compute an optimal route.

Theorm 6.2. *Under the assumptions of Theorem 6.1, except that H is a general subgroup of G (not necessarily normal), we can:*

(i) Find all neighbors of Hg in time $O(mn^2|\Phi|)$. In particular, this includes finding the *parent* node and all children nodes.

(ii) Find the minimum word representation for an element of Hg in time $O(mn^2|\Phi|d)$, where d is the diameter of the Cayley graph.

(iii) Find a path between Hg and Hh, which is the sum of the lengths of the minimal paths from each of these nodes to the identity node.

It is worth remarking that H is generated by a subset Φ' of Φ, then it is possible to give a suboptimal (but probably very close to optimal) word which represents an element $g \in G$. This method assumes the existence of spanning trees for $\mathcal{C}_H$ and the Cayley graph for G. The idea is simply to find a minimum word w_1 in Φ which represents an element of Hg and then find a minimum word w_2 in Φ' which represents $\pi(w_1)g^{-1} \in H$. We intend to study further the implications of this strategy and its practicality in routing in the Cayley graph for G.

7. Conclusion.

Two distinct algorithms have been given for manipulating Cayley graphs. The first enables Cayley graphs to be generated faster than before. It is expected to be most useful in the design phase of interconnection networks. The second algorithm stores Cayley graphs compactly, using 1–1/2 bits per node. This is expected to be most useful for storing the Cayley graph at each local node, where memory will be most expensive. Generalizations of these algorithms to certain Cayley coset graphs have also been presented.

The first algorithm may provide promise for dynamically and efficiently recreating the routing table, without storing a pre-computed data structure. Dynamically computing optimal routes may be especially beneficial for intelligent routing, in which the local node already has information about congested or failed nodes, and attempts to route around them. The first algorithm has the added advantage of producing a canonical set of rewrite rules for the group. These rewrite rules reduce an arbitrary word to its minimal form, and thus provide an alternative method for quickly discovering optimal routes. Unfortunately, except for special groups, the number of canonical rewrite rules is usually comparable to the number of nodes in the graph.

As networks grow, it will be desirable to run simulations on networks containing millions of nodes. In that case, the avoidance of virtual memory is crucial for doing the simulations in reasonable time. The second algorithm can be used to generate Cayley graphs space-efficiently, using 2 bits per node. Further, optimally compact encodings of the nodes of a Cayley graph or Cayley coset graph are useful for minimizing the storage needed to check if a new node has already been encountered. A standard encoding for groups was reviewed, and a new encoding for cosets was described.

Acknowledgements. The authors gratefully acknowledge Namita Sarawagi for discussions on faster variations of base change algorithms for special cases.

References.

1. F. Annexstein, M. Baumslag, A.L. Rosenberg, "Group Action Graphs and Parallel Architectures", COINS Technical Report 87-133, Computer and Information Science Department, University of Massachusetts (1987).

2. B. Benninghofen, S. Kemmerich, and M.M. Richter, *Systems of Reductions*, Lecture Notes in Computer Science **277**, Springer-Verlag, Berlin, 1987.

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.

4. C.A. Brown, G. Cooperman, and L. Finkelstein, “Solving Permutation Problems Using Rewriting Rules”, *Proc. of the International Symposium on Symbolic and Algebraic Computation* (ISSAC 88, Rome, Italy), Springer Verlag Lecture Notes in Computer Science.

5. C.A. Brown, L. Finkelstein, and P.W. Purdom, “A New Base Change Algorithm for Permutation Groups”, *SIAM J. Computing*, to appear.

6. G. Butler, “Computing in Permutation and Matrix Groups II: Backtrack Algorithm”, *Math. Comp.* **39** (1982), pp. 671–680.

7. 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.

8. J.J. Cannon, “Construction of Defining Relators for Finite Groups”, *Discrete Math.* **5** (1973), pp. 105-129.

9. 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.

10. D.V Chudnovsky, G.V. Chudnovsky and M.M. Denneau, “Regular Graphs with Small Diameter as Models for Interconnection Networks”, *I.E.E.E. Trans. Comp.* (1988), pp. 232–239.

11. G. Cooperman and L.A. Finkelstein, “A Strong Generating Test and Short Presentations for Permutation Groups”, *J. Symbolic Algebra*, submitted for publication.

12. G. Cooperman, L. Finkelstein and P. Purdom, “Fast Group Membership Using a Strong Generating Test”, to appear in *Proc. of 1989 Computers and Mathematics Conference.*

13. G. Cooperman, L. Finkelstein and N. Sarawagi, “Efficient Computation of a Base Change”, Northeastern University Technical Report, in progress.

14. M. Jerrum, “A Compact Representation for Permutation Groups”, *J. of Algorithms* **7** (1986), pp. 60–78.

15. D.E. Knuth and P.B. Bendix, “Simple Word Problems in Universal Algebras” in *Computational Problems in Abstract Algebras*, J. Leech, Ed., Pergamon Press, 1970, pp. 263–297.

16. P. Le Chenadec, *Canonical Forms in Finitely Presented Algebras*, Pitman Publishing, London, 1986 (in U.S.A., J. Wiley & Sons, New York).

17. D. Richardson, *GROUP: A Computer System for Group-theoretical Calculations*, M.Sc. Thesis, Univerity of Sydney, 1973.

18. C.C. Sims, “Computation with Permutation Groups”, in *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, edited by S.R. Petrick, ACM, New York, 1971.