Algorithmica (1991) 6: 554–564

Algorithmica
© 1991 Springer-Verlag New York Inc.

Parallel Sorting on Cayley Graphs

Daniel M. Gordon[1]

Abstract. This paper presents a parallel algorithm for sorting on any graph with a Hamiltonian path and 1-factorization. For an n-cube the algorithm is equivalent to the sequential balanced sorting network of Dowd, Perl, Rudolph, and Saks. The application of this algorithm to other networks is discussed.

Key Words. Sorting, Hypercube, Sorting network, Parallel algorithm, Complexity.

1. Introduction. The problem of sorting in parallel has been attacked for several different kinds of networks. The method of Ajtai *et al.* [1] sorts N elements in $O(\log N)$ time, but it is based on expander graphs, and is not a practical network. Various methods [10], [12] have been proposed for mesh networks, in which the processors form an $m \times n$ grid. The planarity of these networks make them ideal for VLSI implementation, but their high diameter means that the fastest time of any algorithm on a mesh to sort N elements is $O(\sqrt{N})$.

An n-dimensional hypercube, or n-cube for short, is another standard network for parallel processing, because of its regularity, the small number of connections between processors, and the low diameter (maximum distance between any pair of nodes) of the network. The diameter of an n-cube is $\log N$, for $N = 2^n$ the number of nodes, so in theory an algorithm on the n-cube could sort in $O(\log N)$ time. The best method currently known is Batcher's sort, which runs in $(\log^2 N)/2$ time.

In this paper we present Graphsort, a general algorithm suitable for many different kinds of networks. Applied to linearly connected processors, it produces the odd–even transposition sort. On an n-cube, it is equivalent to the sequential balanced sorting network of [4], which also sorts N elements in $(\log^2 N)/2$ time. The action of Graphsort on the cube makes its relation to Batcher's sort clear. In Section 4 we discuss the performance of Graphsort on other types of networks.

2. Definition of the Algorithm. Consider a network as a graph, with processors at each node containing one element of data, which may be exchanged with processors at adjacent nodes. In order to sort on a network, we need a linear ordering of the vertices. It is necessary that vertices adjacent in this ordering be adjacent in the graph, so that any inversions in the order of the data may be detected. Any such ordering $v_1, v_2, \ldots, v_N$ will form a Hamiltonian path for the graph. We only consider networks with Hamiltonian paths.

[1] Department of Computer Science, University of Georgia, Athens, GA 30602, USA.

Received December 10, 1988; revised July 5, 1989. Communicated by Robert Sedgewick.

Such a path is referred to as a *snake-like ordering* in [10] and [12], where a similar idea is used for sorting on a mesh. The path used in those papers starts at the upper-left corner of the mesh, going to the right along odd-numbered rows and to the left along even-numbered rows.

The other necessary ingredient to the algorithm is a *1-factorization* of the graph G. A *1-factor* is a set of disjoint edges which are incident on every vertex. A set of edge-disjoint 1-factors $F_1, F_2, \ldots, F_n$ for which $F_1 \cup F_2 \cup \cdots \cup F_n = G$ is a 1-factorization of G. For example, a 1-factorization of a path consists of two 1-factors: the odd edges and the even edges. The n-cube has a 1-factorization consisting of n sets of parallel edges in each of the dimensions.

Let G be any graph with a Hamiltonian path $H = v_1, v_2 \ldots, v_N$ and 1-factorization $F = F_1, F_2, \ldots, F_n$. Let x_i be the data element stored at v_i. Then the following algorithm will sort elements at the nodes of G.

Algorithm Graphsort. Repeat steps $1, 2, \ldots, n$ until the nodes are sorted in the order of H:

1 Compare the data entries in each pair of nodes connected by an edge (v_i, v_j) in F_1. If they are not in the same order as the order of the nodes in H (i.e., $i < j$ and $x_i > x_j$), switch the entries.
2. Compare each pair of nodes connected by an edge in F_2, and swap if the data is not in H-order.

⋮

n. Compare each pair of nodes connected by an edge in F_n, and swap if the data is not in H-order.

It is easy to see that Graphsort will eventually sort, since the odd–even transposition sort is a subset of the sorting network. In fact, any sorting network may be considered a special case of Graphsort, since the graph G may be chosen with its edges corresponding to all the comparisons done in the particular network.

Of more interest is to take well-known graphs, and see for which choices of H and F Graphsort sorts any input in a reasonable amount of time. For G a path, with F_1 consisting of the odd edges and F_2 the even edges, Graphsort is the odd–even transposition sort.

Shearsort [12] and its modifications [10], [13] are similar to Graphsort. Shearsort also uses a Hamiltonian path in a mesh, but it sorts using the horizontal edges until the rows are sorted, and only then sorts along vertical edges repeatedly until the columns are sorted. This process is repeated until the network is sorted. Rotatesort, the version of Marberg and Gafni [10], sorts an $m \times n$ mesh in $O(m + n)$ steps.

3. Graphsort on the *n*-Cube.

An n-cube has $N = 2^n$ vertices corresponding to the elements of $\{0, 1\}^n$, with edges between vertices which differ in a single entry. The *natural labeling* of a vertex $i_1, i_2, \ldots, i_n$ will be the binary number

$$i = i_1 i_2 \cdots i_n = \sum_{k=1}^{n} i_k 2^{n-k}.$$

A Hamilton path on the n-cube is also called a *Gray code*. There are many Hamiltonian paths on an n-cube, and in principle any one would work, but it is convenient to use the standard Gray code.

We write $G(n)$, the n-dimensional Gray code, as a list of n-bit binary numbers representing the vertices of the n-cube. Let $\overline{G(n)}$ be the Gray code written in reverse order, and define $G(1)$ as the sequence: $\langle 0, 1\rangle$. Then we can define the Gray code recursively by

$$G(n+1) = \langle 0 \oplus G(n), 1 \oplus \overline{G(n)}\rangle. \tag{1}$$

In other words, the $(n+1)$st Gray code is the nth with an extra zero added in front, followed by the same code in reverse order with a one in front. In the language of Hamiltonian paths, the $G(n)$ first traverses the $(n-1)$-dimensional subcube with first coordinate zero, then crosses to the subcube with first coordinate one, and traverses it in the reverse order.

For the 1-factorization of the n-cube, let F_k be the set of edges between nodes differing in the kth coordinate. Then $F = F_1 \cup F_2 \cup \cdots \cup F_n$ is clearly a 1-factorization of the n-cube, since the sets are disjoint and each edge of the cube is in some F_k.

Then Graphsort in this case becomes:

Algorithm n-Cubesort. Repeat steps $1, 2, \ldots, n$ until the nodes are sorted in Gray code order:

1. Compare the data entries in each pair of nodes differing only in the most significant bit. If they are not in the same order as the Gray code labels of the nodes, switch the entries.
2. Compare each pair of nodes differing in the second most significant bit, and swap if the data is not in Gray code order.

⋮

n. Compare each pair of nodes differing in the least significant bit, and swap if the data is not in Gray code order.

Following the terminology of [4], the $N/2$ parallel comparisons in each step is called a *phase*, and each group of n phases is called a *block*. Figure 1 shows a block

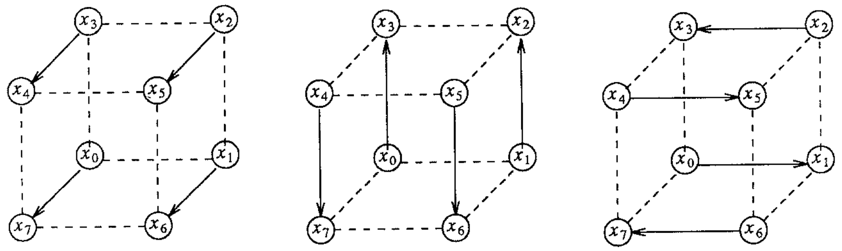

Fig. 1. One round of Graphsort on the 3-cube.

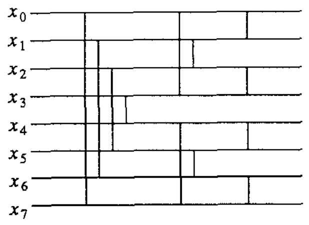

Fig. 2. Standard sorting network diagram of Cubesort.

for $n = 3$ on the cube. Figure 2 shows the same sequence of comparisons drawn as a standard sorting network, where the top horizontal line is the node with Gray code 0, the next is the node with Gray code 1, and so on. A vertical line between two horizontal line represents a comparison between the corresponding nodes, with the larger element going to the bottom of the vertical line, and the smaller one to the top.

There is another characterization of the Gray code which is useful. Let $g_n(i)$ denote the natural labeling of the ith element in $G(n)$. Let $Q(m)$ be the bit which is different in $g_n(m)$ and $g_n(m+1)$. Then it is well known (see, for example, [11]) that

$$Q(m) = \max\{k : 2^k | m\}. \tag{2}$$

Thus, changes are made in the top k bits only every 2^{n-k} steps in the Gray code. If we divide the Gray code into 2^k blocks of 2^{n-k}, then the first k bits in each block will be constant, and the last $n-k$ bits will run through $G(n-k)$ or $\overline{G(n-k)}$:

$$\begin{array}{rl|l}
 & \overbrace{}^{\text{first } k \text{ bits}} & \overbrace{}^{\text{last } n-k \text{ bits}} \\
g_k(0) = & 00\ldots00 & G(n-k), \\
g_k(1) = & 00\ldots01 & \overline{G(n-k)}, \\
g_k(2) = & 00\ldots11 & G(n-k), \\
g_k(3) = & 00\ldots10 & \overline{G(n-k)}, \\
 & \vdots & \\
g_k(2^k-2) = & 10\ldots01 & G(n-k), \\
g_k(2^k-1) = & 10\ldots00 & \overline{G(n-k)}.
\end{array} \tag{3}$$

THEOREM 1. *Cubesort is equivalent to the sequential balanced sorting network.*

PROOF. In [4] the sequential balanced sorting network is defined recursively. Let $\mathbf{SB}_n$ denote the sequential balanced sorting network on $N = 2^n$ inputs x_0, $x_1, \ldots, x_{N-1}$. Then $\mathbf{SB}_1$ consists of a single comparator. The first phase of $\mathbf{SB}_n$ contains the comparisons

$$x_0 : x_{N-1}, x_1 : x_{N-2}, \ldots, x_{N/2-1} : x_{N/2}. \tag{4}$$

The later phases are two copies of $\mathbf{SB}_{n-1}$, applied in parallel to the first $N/2$ inputs and the last $N/2$ inputs. Figure 2 is equivalent to $\mathbf{SB}_3$.

Theorem 1 will be shown by demonstrating that the same definition works for Cubesort. 1-Cubesort consists of a single comparison of the two nodes of a 1-cube.

Phase 1 of n-Cubesort consist of comparisons between nodes differing only in the high-order bit. Since the Gray code $G(n)$ first traverses the $(n-1)$-cube with leading bit 0, and then traverses the $(n-1)$-cube with leading bit 1 in reverse order, its first phase compares the first and last inputs, the second and second-to-last, and so on, as in (4). The remaining phases involve comparisons between nodes differing in lower-order bits, and so are confined to the two subcubes. By (1), the parts of the Gray code in each subcube are just $(n-1)$-dimensional Gray codes, so the comparisons in each subcube will be $(n-1)$-*Cubesort*. □

In [4] it is shown that $\log N$ blocks of the sequential balanced sorting network are sufficient to sort. However, the proof is rather involved, and uses constructs such as *chains* and *cochains* which are important but not clearly motivated. In the setting of the n-cube, the proof becomes simpler, and the relation of this algorithm to the Batcher odd-even merge sort becomes clear. This proof is based on a simplified version of the Dowd, Perl, Rudolph, and Saks proof by Williamson [14].

THEOREM 2. *n-Cubesort sorts in* $\log^2 N$ *time.*

PROOF. We do not need to consider arbitrary data elements. Because of the well-known Zero-One Principle [8], we may assume that the data at each node is a zero or a one:

ZERO-ONE PRINCIPLE. *If a network sorts all sequences of zeros and ones, then it will sort any sequence.*

Let C_i^k be the k-dimensional subcube of the n-cube with the last k bits of each vertex (in the natural labeling) equal to i, where i is any integer between 0 and 2^k-1. Two C^k*'s* will be *neighboring* if their vertices differ only in the $(k+1)$st coordinate. Figure 3 shows the C^0, C^1, and C^2 subcubes of a 3-cube. These subcubes are the *cochains* of [4]. The proof will show that Cubesort works by

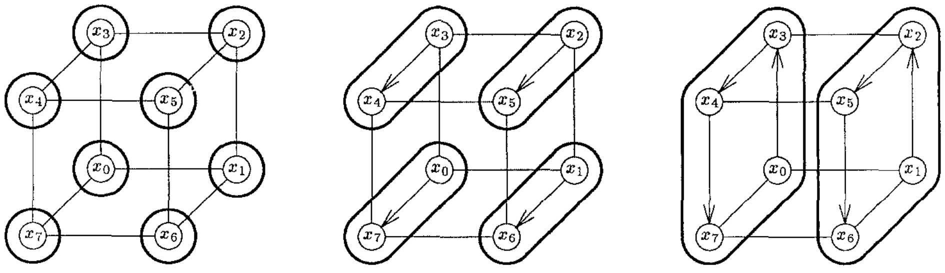

Fig. 3. C^0's, C^1's, and C^2's in the 3-cube.

sorting progressively bigger C^k's, and then merging neighboring pairs of them to form C^{k+1}'s.

The proof results from a series of lemmas:

LEMMA 1. *Phases* 1, 2, ..., k *of n-Cubesort applied to* C_i^k *form one block of k-Cubesort.*

PROOF. It suffices to show that the subsequence of the Gray code $G(n)$ in C_i^k is the Gray code $G(k)$ of C_i^k. Then, since the comparisons of phases 1, 2, ..., k of n-Cubesort are within C_i^k in the same order as in k-Cubesort, the lemma follows.

Consider the nth Gray code, $G(n)$. Its subsequence in C_i^k consists of all the nodes with their last $n - k$ bits equal to i. By (3) the first 2^{n-k} nodes will have their k leading bits equal to zero, while cycling through a Gray code on the $(n - k)$-subcube. One of these nodes has its last $n - k$ bits equal to i, and so is the first node in Gray code order of C_i^k.

The next 2^{n-k} nodes will have $k - 1$ leading zeros followed by a one, and so one of these will be the second node in the Gray code on C_i^k. Continuing, each block of 2^{n-k} nodes will have one node from C_i^k, and by the left-hand side of (3), these nodes will occur in the proper order. □

In the following lemmas we encounter the situation where a k-subcube is not sorted, but all of its even members $x_0, x_2, \ldots, x_{2^k-2}$ are sorted, and all of its odd members $x_1, x_3, \ldots, x_{2^k-1}$ are sorted as well. Such a cube is called *shuffled.*

LEMMA 2. *Suppose* C_i^k *and* C_j^k *are neighboring k-subcubes, say* $j = i + 2^{n-k}$, *and both cubes are sorted. Then if phase* $k + 1$ *is applied, the resulting cube* C_i^{k+1} *will be shuffled.*

PROOF. Denote the nodes of C_i^k by $y_0, y_1, \ldots, y_{2^k-1}$, and the nodes of C_j^k by z_0, $z_1, \ldots, z_{2^k-1}$. Then in (3), y_l and z_l both occur in the lth block. Their order will depend on the parity of l, because of the alternating $G(n - k)$'s and $\overline{G(n - k)}$'s: y_0 comes first in the 0th block, z_1 in the next, and so on. Therefore the order of the nodes in C_i^{k+1} will be

$$y_0, z_0, z_1, y_1, y_2, z_2, z_3, \ldots, z_{2^k-2}, z_{2^k-1}, y_{2^k-1}. \tag{5}$$

Then at phase $k + 1$, y_l is compared with z_l, since each natural labeling of z_l is 2^{n-k} bigger than that of y_l. For l even, the larger data element goes to y_l, while for l odd the larger data goes to z_l. This is because in (3) y_l occurs first in all the $G(n - k)$'s, and last in the $\overline{G(n - k)}$'s.

To show that this sequence is shuffled, consider y_l, for l even. Then $y_l \leq y_{l+1}$, since the y's are sorted, and in the $(k + 1)$st phase, z_{l+1} gets the larger of the two values z_{l+1} and y_{l+1}. Therefore y_l is no bigger than z_{l+1}, the element two nodes after it in C_i^{k+1}.

For l odd, phase $k + 1$ results in y_l getting the smaller of y_l and z_l, and z_{l+1} getting the smaller of y_{l+1} and z_{l+1}. Since y_l and z_l were smaller than y_{l+1} and

z_{l+1}, respectively, the minimum of the first two must be smaller than the minimum of the second two, and so y_l is less than z_{l+1}.

The argument to show that $z_l \le y_{l+1}$, for l odd and l even, proceeds in the same manner. □

LEMMA 3. *Suppose C_i^k is shuffled. Then if phases* 1, 2, ..., *k are applied, C_i^k will be sorted.*

PROOF. The proof is by induction on k. For $k = 1$, any 1-cube is shuffled, and is sorted by the single comparison at phase 1.

Suppose that the lemma is true for 1, 2, ..., k, and let C_i^{k+1} be shuffled. Let C_i^k and C_j^k be as in Lemma 2. By the Zero-One Principle, we may assume that the inputs are all zeros and ones. Since C_i^{k+1} is shuffled, the even nodes of the cube, y_0, z_1, y_2, $z_3, \ldots, y_{2^k-2}$, z_{2^k-1}, consist of a zeros followed by $2^k - a$ ones. The odd nodes, $z_0, y_1, z_2, y_3, \ldots, z_{2^k-2}, y_{2^k-1}$, will be b zeros followed by $2^k - b$ ones. Thus the number of zeros in C_i^k will be $\lceil a/2 \rceil + \lfloor b/2 \rfloor$, and the number of zeros in C_j^k will be $\lfloor a/b \rfloor + \lceil b/2 \rceil$. The difference between the number of zeros (and ones) in the two subcubes is at most one.

Note that the subcubes C_i^k and C_j^k are also shuffled, since in (5) consecutive y's and z's alternate in parity. Therefore, after phases 1, 2, ..., k, C_i^k and C_j^k will be sorted by the induction hypothesis. The number of zeros and ones in each subcube is still within one of the other, since the comparisons in these phases are all within the subcubes. Thus, the nodes in C_i^{k+1} will have at most one inversion, which is removed by the comparisons in phase $k + 1$. □

PROOF OF THEOREM 2. By the above lemmas, we have that all the C^k's are sorted after the kth phase of the kth block, and are merged into shuffled C^{k+1}'s in the $(k + 1)$st phase. The remaining phases are not needed, and may be skipped.

Because of the symmetry of the comparisons, the subcubes remain shuffled through these extra phases. At phase l, C_i^k is compared with $C_{i+2^{n-l}}^k$, which is also shuffled. Moreover, all the comparisons of the even nodes go in the same direction, and all the comparisons of the odd nodes go in the other direction. Since both subsequences were sorted, the resulting subsequences are the maxima and minima of two sorted sequences, which must also be sorted. □

This shows that n-Cubesort sorts in $\log^2 N$ time, which is twice the time of Batcher's sort. However, the action of the final $n - k - 1$ phases in the kth block, for $k = 1, 2, \ldots, n - 2$, are not used, and may be left out without affecting the proof that the algorithm sorts. This modified algorithm is as fast as Batcher's odd-even merge sort, and is in fact a variation of Batcher's sort, with subcubes being sorted and merged.

4. Cayley Graphs.

While Graphsort can be applied to any graph, it is natural to place some restrictions on them. For instance, the diameter of the graph should be small, since trivially the time necessary to sort cannot be smaller than the diameter.

If the graph is to represent a realistic parallel network, the degree should not be too high, and a certain amount of regularity is desirable to make it easier to design other algorithms for.

Let Γ be a group and let $\tau_1, \tau_2, \ldots, \tau_k$ be a set of generators for Γ. Then the *Cayley graph* $G(\Gamma; \tau_1, \ldots, \tau_k)$ is a graph with vertices V corresponding to the elements of Γ. Two vertices γ_i and γ_j are connected by an edge if and only if there is some k such that $\gamma_i \tau_k = \gamma_j$. For example, the n-cube is a Cayley graph. Let $\Gamma = (\mathbf{F}_2)^n$ and let the τ's be the generators

$$(1, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, \ldots, 0, 1).$$

In [3] it has been suggested that Cayley graphs make good choices for networks for parallel architectures. They are vertex symmetric, and most standard networks can be formulated in terms of Cayley graphs.

There are also advantages for applying Graphsort. Cayley graphs have a natural 1-factorization: let F_i be the set of all edges corresponding to τ_i. Then it is easy to see that $F = F_1, F_2, \ldots, F_k$ form a 1-factorization of G.

Furthermore, it has been conjectured that all undirected Cayley graphs are Hamiltonian. This conjecture is not true for all vertex-symmetric graphs, since the Petersen graph has a Hamiltonian path but no Hamiltonian cycle. Neither is it true if the Cayley graph is directed (which happens if some τ is in the set of generators and its inverse is not); see Exercise 6 in [11]. The conjecture is open, and does not provide any way of finding a Hamiltonian path in a Cayley graph, but it does suggest that Cayley graphs are a well-behaved family to apply this algorithm to.

It makes sense to look for other Cayley graphs of "nice" groups. One reasonable choice would be S_n, the symmetric group. There are two well-known sets of generators of S_n which result in a graph of small diameter. One is the *star graph*, with generators (1 2), (1 3), ..., (1 n). This graph has many properties such as fault tolerance, an easy routing algorithm, and diameter $\lfloor 3(n-1)/2 \rfloor$ [2], which make it a practical network.

The *pancake graph* can be defined by thinking of a stack of n pancakes, which may be permuted by flipping the top k pancakes, for $k = 2, 3, \ldots, n$. Denote these operations by f_k. The exact diameter of this graph is unknown for $n > 9$, but is at most $(5n + 5)/3$ [7]. The pancake graph has a simple Hamiltonian cycle [15]: call it $H(n)$. Then $H(2) = f_2$, $H(3) = f_2, f_3, f_2, f_3, f_2$, and

$$H(n) = H(n-1), f_n, H(n-1), f_n, \ldots, f_n, H(n-1).$$

While it is difficult to prove an upper bound for the time complexity of Graphsort on a given graph, it is easy to get good lower bounds, by trying random permutations on the network. The results of such tests for the star and pancake graphs are given in Table 1. For each n, the table gives a lower bound on the number of phases needed. Also given, as a comparison to the performance of Cubesort, is $\log_2 N$.

The performance of the two graphs is disappointing. Possible explanations are the pancake graph's lack of symmetries compared with the cube and star graphs.

Table 1. Lower bounds for sorting on star and pancake graphs.

n	N	Star	Pancake	$(\log_2 N)^2$
3	6	6	6	6.7
4	24	30	21	21.0
5	120	64	60	47.7
6	720	200	135	90.1
7	5040	546	450	151.3

and the Hamiltonian path used for the star graph. No natural Hamiltonian cycle with properties similar to the Gray code is known for the star graph. Such a cycle would be interesting in its own right, and would presumably lead to better performance of Graphsort on the star graph.

Faber and Fellows [5] have been working on finding Cayley graphs with low degree and diameter and a large number of vertices. These qualities make these graphs seem ideal for Graphsort, but the *ad hoc* constructions of the graphs makes it difficult to prove any bounds for the performance of Graphsort on these graphs. The proof of Cubesort's performance depended strongly on the structure of the n-cube and the Gray code.

Most of the groups in the graphs of [5] are $GL(n, \mathbf{Z}/q\mathbf{Z})$ or $SL(n, \mathbf{Z}/q\mathbf{Z})$, for various values of n and q, but the generators are empirically determined. A regular family of these groups and generators would be easier to prove bounds for.

One very interesting possibility is the *Ramanujan graphs* of Lubotzky *et al.* [9]. These are Cayley graphs of $PSL(2, \mathbf{Z}/q\mathbf{Z})$ or $PGL(2, \mathbf{Z}/q\mathbf{Z})$ for a prime $q \equiv 1 \pmod 4$, with generators corresponding to representations of another prime p as sums of four squares. These graphs are the best explicitly known expander graphs, which are the basis for the AKS sorting network, so they are also good candidates for fast implementations of Graphsort.

All known generators for $PSL(2, \mathbf{Z}/q\mathbf{Z})$ or $SL(2, \mathbf{Z}/q\mathbf{Z})$ result in Cayley graphs with diameter $O(\log N)$, where N is the order of the group. If a Hamiltonian path for these graphs with any set of generators is found, it could be used to test Graphsort on these groups.

The expander property may be an important feature for a graph to work well with Graphsort. It can be shown that each block of n-Cubesort functions as an ε-nearsorter on some $(n-1)$-dimensional subcube; after each block the number of entries in the subcube which do not belong there is at most $\varepsilon 2^{n-1}$, for some ε. This may be used for an alternate proof that Cubesort works in $O(\log^2 N)$ time. The problem is that each such nearsorter takes $O(\log N)$ time, while nearsorters in the AKS network, because of the network's expanding properties, take only a constant number of steps.

Other graphs are also good candidates. DeBruijn graphs are not Cayley graphs, but they have a recursive structure that makes them suitable for use as practical networks. In addition, they have very low diameter, and many constructions of Hamiltonian cycles are known. See [6] for an extensive survey of these constructions.

5. Further Directions.

It is still an open question whether there is a family of graphs for which Graphsort sorts N data elements in $O(\log N)$ time. One avenue of research is to determine its performance on other graphs. In particular, find a family of expander graphs with Hamiltonian paths and 1-factorizations, and examine Graphsort's performance on them.

Another open questioin is: Is there any sorting algorithm on the n-cube which can sort in $O(n)$ time? Perhaps some modification of Cubesort could improve its performance. Shearsort [12] took $\lceil \log m \rceil + 1$ rounds on an $m \times n$ mesh, where each round consisted of row sorting followed by column sorting, similar to Cubesort. Marberg and Gafni [10] added a few macros to the algorithm, and reduced the number of rounds needed to a constant. It is possible that similar modifications might reduce the number of rounds needed by Cubesort, ideally also to a constant number of rounds.

Another interesting area to study would be other choices of Gray codes. It seems likely that this Gray code and sequence of comparisons in each round is optimal, but perhaps not. In $G(n)$, x_0 and x_{N-1} are next to each other, but intuitively it seems more sensible to have them far apart. A Hamiltonian path which did this might work better than Cubesort. The proofs given above depend strongly on the structure of $G(n)$, but a sufficiently regular alternate Gray code might be usable. Empirical results suggest that random Gray codes do worse than $G(n)$.

Finally, what are the conditions on a graph that make it work well for Graphsort? One necessary condition is a small diameter, but is that sufficient? Further tests with other graphs, especially Cayley graphs for $PSL(2, \mathbf{Z}/q\mathbf{Z})$ or $SL(2, \mathbf{Z}/q\mathbf{Z})$, might shed light on these questions, as well as the optimal performance of Graphsort.

Acknowledgments. I would like to thank Gill Williamson for showing me a simplified proof that the sequential balanced sorting network works, which made the action of Cubesort on the subcubes much clearer. Also, thanks to Rodica Simion for bringing [4] to my attention.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, in *Proceedings of the 15th ACM Symposium on Theory of Computing*, 1983, pp. 1-9.

[2] S. B. Akers, D. Harel, and B. Krishnamurthy, The star graph: an attractive alternative to the n-cube, in *Proceedings of the International Conference on Parallel Processing*, 1987, pp. 393-400.

[3] S. B. Akers and B. Krishnamurthy, A group-theoretic model for symmetric interconnection networks, *IEEE Trans. Comput.*, **38** (1989), 555-566.

[4] M. Dowd, Y. Perl, L. Rudolph, and M. Saks, The sequential balanced sorting network, in *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, Montreal, August 1983, pp. 161-172.

[5] V. Faber and M. Fellows, Unpublished communication.

[6] H. Fredricksen, A survey of full length nonlinear shift register cycle algorithms, *SIAM Rev.*, **24** (1982), 195-221.

[7] W. H. Gates and C. H. Papadimitriou, Bounds for sorting by prefix reversal, *Discrete Math.* **27** (1979), 47-57.

[8] D. E. Knuth, *The Art of Computer Programming*, vol. 3, 2nd printing, Addison-Wesley, Reading, MA, 1975.
[9] A. Lubotzky, R. Phillips, and P. Sarnak, Ramanujan graphs, *Combinatorica*, **8** (1988), 261–277.
[10] J. M. Marberg and E. Gafni, Sorting in constant number of row and column phases on a mesh, *Algorithmica*, **3** (1988), 561–572.
[11] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, 2nd edition, Academic Press, New York, 1978.
[12] I. D. Scherson, S. Sen, and A. Shamir, Shear sort: a true two-dimensional sorting technique for VLSI networks, in *Proceedings of the 1986 International Conference on Parallel Processing*, 1986, pp. 903–908.
[13] C. P. Schorr and A. Shamir, An optimal sorting algorithm for mesh connected computers, in *Proceedings of the 18th ACM Symposium on Theory of Computing*, 1986, pp. 255–261.
[14] S. G. Williamson, Unpublished communication.
[15] S. Zaks, A new algorithm for generation of permutations, *BIT*, **24** (1984), 196–204.