A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks

Ian Parberry*
Department of Computer Science
Penn State University

Abstract

It is demonstrated, using a combination of theoretical and experimental computer science, that there is no nine-input sorting network of depth six. If a nine-input sorting network of depth six exists, then there exists one with very special structure. There is an efficient algorithm for constructing and testing comparator networks of this form. This algorithm was implemented and executed on a supercomputer.

1 Introduction

Oblivious comparison-based sorting received much attention early in the history of parallel computing, and has continued to be the subject of much research. The central problem, dubbed the *Bose-Nelson sorting problem* by Floyd and Knuth [6] (after Bose and Nelson [5]), is to devise the most efficient method of sorting n values using a fixed sequence of comparison-swap operations. Many popular sequential sorting algorithms such as *mergesort* are not oblivious, since the sequence of comparisons performed is not the same for all inputs of any given size. In contrast, *bubblesort* is oblivious. An oblivious comparison-based algorithm for sorting n values is called an n-input *sorting network*. One measure of the performance a sorting network is its *depth*, defined to be the number of parallel steps that the algorithm takes given that in one step any number of disjoint comparison-swap operations can take place simultaneously. Sorting networks have a particularly elegant implementation as circuits. Since the comparison-swap operations are oblivious, they can be hard-wired into place.

Optimal sorting networks for $n \leq 8$ and the most efficient sorting networks constructed to date for $n \leq 16$ can be found in Knuth [7] (see Table 1). There are several methods for recursively constructing sorting networks of depth $O(\log^2 n)$ (see, for example, Batcher [3], Parberry [8]), Their depth can be improved by a constant by interrupting the recursive construction early and substituting the best known sorting network on a small fixed number of inputs. For extremely large n the asymptotically optimal $O(\log n)$ depth sorting network of Ajtai, Komlós and Szemerédi [1, 2] has superior depth.

It was known that no nine-input sorting network of depth five can exist, but the best that had been obtained was depth seven. These bounds remained unimproved for over fifteen years. We undertook to determine by exhaustive search whether a nine-input sorting

*Research supported by NSF Grant CCR-8801659 and a Research Initiation Grant from the Pennsylvania State University. Author's current address: Department of Computer Sciences, University of North Texas, P.O. Box 13886, Denton, TX 76203-3886, U.S.A. Electronic mail: ian@dept.csci.unt.edu.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upper	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9
Lower	0	1	3	3	5	5	6	6	7*	7*	7	7	7	7	7	7

Table 1: Current upper and lower bounds on the depth of n-input sorting networks. The starred values are demonstrated here, and were previously equal to six.

network of depth six exists. It is reasonable to ask why we believed that such a naive approach could work now, as opposed to fifteen years ago. The answer is that the fruits of the last fifteen years of research in Computer Science have included impressive tools for the design and analysis of algorithms (theoretical computer science), the implementation of algorithms as programs (structured programming), the translation of programs into object code (compilers), and the execution of object code (supercomputers). Without any one of these tools, the search would not have been feasible. Even with these tools, its feasibility is not immediately apparent.

If our exhaustive search had found a sorting network of depth six, we would now conclude this paper by exhibiting it. However, no such sorting network was found. We can deduce (assuming that the program was correct) that no such sorting network exists. There is much contention in the mathematical community over whether such a computer-assisted "proof" of nonexistence really constitutes a proof in the formal sense. We prefer to regard this type of result as experimental computer science rather than mathematics. In the spirit of traditional experimental science, we will attempt to describe the experiment in sufficient detail for it to be believable and verifiable. Although it is clear that our program could not be used for sorting networks with more than ten inputs, it is hoped that the insights we have gained and techniques we have developed may serve as inspiration and tools for further research.

The remainder of the paper is divided into seven sections. We start with a formal definition of a sorting network (Section 1). It is easy to see that the search-space is far too large for a naive exhaustive search to be feasible using current technology (Section 3). Generating the candidates is fairly straightforward (Section 4), and the number of candidates can be substantially reduced by considering symmetries (Section 5). Testing of the candidates can be achieved quickly by using standard techniques, and information obtained during testing can further reduce the number of candidates (Section 6). A heuristic which rejects almost all and only non-sorting networks (Section 7) further reduces running time to the point where it can be implemented (Section 8). A preliminary version of this paper appeared in [10].

2 Sorting Networks

It will be convenient to adopt a formal notation for specifying sorting networks, in the interest of clarity and conciseness. Formally, a *comparator network* with n inputs is an ordered tuple $(L_1, L_2, \ldots, L_d, n)$. Each L_k is called a *level*, and is is a set of ordered pairs $\langle i, j \rangle$ where $1 \leq i < j \leq n$. Each $\langle i, j \rangle \in L_k$ is called a *comparator*, and is said to be *at level*

k. The values i and j are called *channels*. We insist that every channel is involved in at most one comparison at each level, that is,

$$|\{\langle i,j\rangle \in L_k \mid i < j \leq n\}| + |\{\langle j,i\rangle \in L_k \mid 1 \leq j < i\}| \leq 1,$$

for all $1 \leq i \leq n$ and $1 \leq k \leq d$.

Comparator networks are used to sort an input $x = \langle x_1, \ldots, x_n\rangle$ of n distinct values as follows. Let $C = (L_1, L_2, \ldots, L_d, n)$ be a comparator network. Define the *value* of C in channel i at level k on input x to be $V(i,k)$, where $V(i,0) = x_i$ for $1 \leq i \leq n$, and for $k > 0$,

$$V(i,k) = \begin{cases} \min(V(i,k-1), V(j,k-1)) & \text{if } \langle i,j\rangle \in L_k \\ \max(V(i,k-1), V(j,k-1)) & \text{if } \langle j,i\rangle \in L_k \\ V(i,k-1) & \text{otherwise.} \end{cases}$$

The *output* of C on input x is defined to be $\langle V(1,d), V(2,d), \ldots, V(n,d)\rangle$. If the output of C is in ascending order for all inputs x, then C is called a *sorting network*.

Each level of a comparator network consists of a set of independent comparisons which may be performed in parallel. The number of levels is thus a reasonable measure of parallel time. We will call this the *depth* of the network. We are also interested in the *size* of the network, which we define to be the total number of comparators used,

$$\sum_{i=1}^{d} |L_i|.$$

3 Exhaustive Search

Before investing a large amount of time in writing a program which implements a naive exhaustive search algorithm, it is prudent to estimate its running time. The size of the search space should be calculated, and some simple experiments should be performed to determine how much time is required to test each candidate. The product of these two values will be a reasonable estimate of the running time of the search program.

We say that a comparator network has *maximal size* if it has the maximum number of comparators for its depth. A maximal-size comparator network of depth d with n inputs has $d\lfloor n/2\rfloor$ comparators. Let $C(n,d)$ be the number of maximal-size comparator networks of depth d. Then $C(n,d) = M(n)^d$, where $M(n)$ is the number of different candidates for a single level.

If n is even, each level consists of n channels which are matched perfectly into pairs by $n/2$ comparators. Therefore there are as many different candidates for a single level as there are perfect matchings of n items. If n is even, there are $n-1$ choices for the channel to be matched to channel 1, $n-3$ choices for the channel to be matched to the next unmatched channel, etc. That is, $M(n)$ is the product of the odd numbers less than n. If n is odd, then there are as many different candidates for a single level as there are perfect matchings of $n+1$ items. Therefore,

$$M(n) = \prod_{i=1}^{\lfloor (n-1)/2\rfloor} (2i+1)$$

for all $n \geq 1$. The number of candidate comparator networks of maximal size is therefore

$$C(9,6) = 945^6 \approx 7 \times 10^{17}.$$

Improvement	Running Time
Naive Algorithm	4×10^{11} years
First level symmetries	4×10^{8} years
Second level symmetries	2×10^{7} years
Construction of last level	2×10^{4} years
Heuristic	20 years
CRAY-2	200 hours

Table 2: Estimated running time for the naive exhaustive search algorithm and for each of the successive improvements.

A maximal-size sorting network of depth six with nine inputs has 24 comparators. From Van Voorhis [12], all nine-input sorting networks must have at least 23 comparators. Therefore every non-maximal-size candidate can be generated from a maximal-size one by deleting a single comparator. Of the 24 comparators, only 16 are candidates for deletion since every sorting network which is missing a comparator from the first or last level still sorts when the comparator is replaced. (Note that, contrary to intuition, this is not the case for arbitrary comparators: inserting a comparator into the centre of one of the small but deep sorting networks in Knuth [7] will cause it to fail.) Thus there are $16C(9,6)$ candidate comparator networks which are not maximal-size, making a total of $17C(9,6) \approx 1.2 \times 10^{19}$.

Suppose that it is possible to generate and test 100 comparator networks per second (preliminary experiments indicated that this is a reasonable assumption for a program written in Berkeley UNIX[1] Pascal and executed on a VAX 11/780[2]). The running time of the naive exhaustive search program would then be 3.8×10^{11} years. We can conclude that a naive exhaustive search is not a feasible option.

However, we will see in the remainder of the paper that the running time can be reduced by the use of slightly more sophisticated techniques. In Section 5 we will learn that the number of candidates for the first level can be reduced to 1, and that the number of candidates for the second level can be reduced to 57. In Section 6 we will see how to construct the last level on-the-fly during testing, given the first five levels. In Section 7 we will develop a heuristic which can be applied to the first four levels of the network. It will be proved that the heuristic, which in preliminary experiments rejected more than 99.9% of the bad candidates (which cannot be completed to give a sorting network), does not reject any good candidates. Finally, by implementing the algorithm on a CRAY-2[3] supercomputer (which is approximately 1000 times faster than the VAX 11/780 used for the preliminary experiments), the run-time can be brought down to a practical level. Table 2 shows the effect of each of these successive improvements.

[1]UNIX is a trademark of AT&T Bell Labs.
[2]VAX is a trademark of Digital Equipment Corp.
[3]CRAY is a trademark of Cray Research.

4 Generating Comparator Networks

The problem of generating all n-input comparator networks of depth d can be reduced to generating all possible candidates for a single level. We omit the details of this standard reduction, which is a simple exercise using recursion. We can restrict ourselves to generating maximal-size comparator networks, since we observed in Section 3 that non-maximal-size networks can be generated by the selective deletion of comparators from a network of maximal size. We also saw in Section 3 that this problem is reducible to the problem of generating all perfect matchings of an even number of items.

Suppose n is even. Consider the problem of generating the perfect matchings of the values $a_1, a_2, \ldots, a_n$ which are initially stored in an array named item, with item$[i] = a_i$ for $1 \leq i \leq n$. Procedure match will permute the contents of item in such a manner that all perfect matchings are generated, where the value in item$[2i-1]$ is to be matched with the value in item$[2i]$, for $1 \leq i \leq n/2$. It would be sufficient to generate all permutations, but our aim is to keep the running time to a minimum by generating only permutations which are perfect matchings. We assume the existence of a procedure called process, which takes an array as a parameter and processes the perfect matching contained therein.

```
1.   procedure match(item,n);
2.     begin
3.       if n = 2 then process(item)
4.         else
5.           begin
6.             match(item,n − 2);
7.             for i := n − 2 downto 1 do
8.               begin
9.                 swap item[i] with item[n − 1];
10.                match(item,n − 2)
11.              end;
12.            temp := item[n − 1];
13.            for i := n − 1 downto 2 do
14.              item[i] := item[i − 1];
15.            item[1] := temp
16.          end
17.    end;
```

The perfect matchings of the n values in item$[1]$, item$[2]$, ..., item$[n]$ are processed with a call to match(item,n). The correctness of the procedure can be verified by induction on even n. The hypothesis is certainly true for $n = 2$. Now suppose that the hypothesis is true for match(item,$n-2$). In line 6, all perfect matchings of n items with a_n matched to a_{n-1} are processed. In lines 7 through 11, each iteration of the for-loop processes the perfect matchings of n items with a_n successively matched with a_{n-2}, a_{n-3},..., a_1. Upon termination of the for-loop, item$[n] = a_n$, item$[1] = a_{n-1}$, and for $2 \leq i \leq n-1$, item$[i] = a_{i-1}$. This permutation (a cyclic shift of item$[1]$ through item$[n-1]$) is undone in lines 12 through 15.

Let $T(n)$ be the running-time of procedure match with second parameter n, where n is even. We claim that $T(n) = O(M(n))$; that is, the amount of time required to process all perfect matchings of n items is linear in the number of perfect matchings. We know by inspection of the algorithm that $T(2) = c$ and for $n \geq 4$, $T(n) = (n-1)T(n-2) + dn$, for

some natural numbers c, d.

Lemma 4.1 *If $2 \leq i \leq n/2 - 1$, and $n \geq 4$ is even, then:*

$$\sum_{k=1}^{i} \prod_{j=1}^{k} (n - 2j + 1) \leq 2 \prod_{j=1}^{i} (n - 2j + 1). \tag{1}$$

PROOF: The proof follows by induction on i. The hypothesis holds for $i = 2$, since $(n-1) + (n-1)(n-3) \leq 2(n-1)(n-3)$ for $n \geq 4$. Now suppose that

$$\sum_{k=1}^{i-1} \prod_{j=1}^{k} (n - 2j + 1) \leq 2 \prod_{j=1}^{i-1} (n - 2j + 1). \tag{2}$$

From inequality (1), it is sufficient to prove that

$$\sum_{k=1}^{i-1} \prod_{j=1}^{k} (n - 2j + 1) \leq \prod_{j=1}^{i} (n - 2j + 1). \tag{3}$$

By inequality (2), inequality (3) holds if

$$2 \prod_{j=1}^{i-1} (n - 2j + 1) \leq \prod_{j=1}^{i} (n - 2j + 1),$$

that is, $n - 2i + 1 \geq 2$, which certainly holds for $i \leq n/2 - 1$. □

It can be demonstrated by induction on $i \leq n/2 - 1$ that

$$T(n) \leq \prod_{j=1}^{i} (n - 2j + 1) T(n - 2i) + 2d \sum_{k=1}^{i} \prod_{j=1}^{k} (n - 2j + 1).$$

Therefore by Lemma 4.1 with $i = n/2 - 1$,

$$T(n) \leq c \prod_{j=1}^{n/2-1} (n - 2j + 1) T(n - 2i) + 4d \prod_{j=1}^{n/2-1} (n - 2j + 1).$$

That is, $T(n) \leq (c + 4d) M(n)$.

The recursion can be removed from procedure match without asymptotic time loss using standard techniques. Both the recursive and the non-recursive versions of this algorithm generate the perfect matchings in the same order. We will number them consecutively, starting at one. We call this the *lexicographic number* of the perfect matching.

5 Symmetries

We have chosen to build our sorting networks using *min-max* comparators, which send the smallest value to the left and the largest to the right, as opposed to *max-min* comparators, which send the largest to the left and the smallest to the right. Formally, a *generalized*

comparator network is an ordered tuple $(L_1, L_2, \ldots, L_d, n)$, where each L_k is a set of ordered pairs $\langle i, j\rangle$, such that $1 \leq i, j \leq n$, $i \neq j$ and for all $1 \leq i \leq n$, $1 \leq k \leq d$,

$$|\{j \mid \langle i, j\rangle \in L_k\}| + |\{j \mid \langle j, i\rangle \in L_k\}| \leq 1.$$

Each ordered pair $\langle i, j\rangle \in L_k$ is called a *min-max comparator* if $i < j$, and a *max-min comparator* if $i > j$. Intuitively, each comparator is an ordered pair of channels. The first is the channel which receives the minimum of the two inputs, and the second is the channel which receives the maximum of the two inputs. Let C be a generalized comparator network. Suppose that the output of C on input $x = \langle x_1, x_2, \ldots, x_n\rangle$ is $\langle y_1, y_2, \ldots, y_n\rangle$. C is said to be a *generalized sorting network* if there exists a permutation $\Pi\colon\{1, \ldots, n\} \to \{1, \ldots, n\}$ such that for all inputs x, the output is sorted after application of the permutation Π, that is, $\langle y_{\Pi(1)}, y_{\Pi(2)}, \ldots, y_{\Pi(n)}\rangle$ is in sorted order. Generalized sorting networks are no more powerful than sorting networks:

Theorem 5.1 *(Floyd-Knuth) For every generalized sorting network there is a sorting network with the same size and depth. If the former has only min-max comparators in the first k levels, then the latter is identical in the first k levels.*

PROOF: See Knuth [7] or Parberry [9]. □

Let $\Pi\colon\{1, \ldots, n\} \to \{1, \ldots, n\}$ be a permutation. If $\langle i, j\rangle$ is a comparator, define $\Pi(\langle i, j\rangle) = \langle \Pi(i), \Pi(j)\rangle$. If $L = \{C_1, \ldots, C_r\}$, where each C_i is a comparator for $1 \leq i \leq r$, define $\Pi(L) = \{\Pi(C_1), \ldots, \Pi(C_r)\}$. If $C = (L_1, \ldots, L_d, n)$ is a generalized comparator network, define $\Pi(C) = (\Pi(L_1), \ldots, \Pi(L_d), n)$.

Intuitively, $\Pi(C)$ is the generalized comparator network obtained from C by permuting the channels according to the permutation Π, changing min-max comparators to max-min comparators (and vice-versa) as appropriate.

Lemma 5.2 *If C is a generalized sorting network, then $\Pi(C)$ is a generalized sorting network for every permutation Π.*

PROOF: If the output of C is always in sorted order after it is permuted according to the permutation Π_C, then the output of $\Pi(C)$ is in sorted order after application of the permutation Π^{-1} followed by application of Π_C. □

5.1 The First Level

There is no need to check all comparator networks to see if they sort. In fact, due to the following result we can fix the first level. We say that a comparator network is in *first normal form* if the first level has comparators between channel $2i - 1$ and channel $2i$ for $1 \leq i \leq \lfloor n/2\rfloor$.

Theorem 5.3 *If there is an n-input sorting network of depth d then there is an n-input sorting network of depth d that is in first normal form.*

PROOF: Let C be a sorting network of depth d. Without loss of generality we can assume that the first layer of C is maximal-size. Clearly, there exists a permutation Π such that $\Pi(C)$ is a generalized comparator network in first normal form. By Lemma 5.2, $\Pi(C)$ is a

n	R(n)	n	R(n)
1	1	6	5
2	1	7	23
3	3	8	12
4	3	9	58
5	9	10	20

Table 3: The value of $R(n)$, the number of representative second levels with n inputs, for $n \leq 10$.

generalized sorting network. Therefore, by Theorem 5.1, there exists a sorting network of depth d that is in first normal form. □

5.2 The Second Level

The number of candidates for the second level of the comparator network can be reduced significantly. A permutation $\Pi: \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ is called *first-normal-form preserving* if for all n-input comparator networks C in first normal form, $\Pi(C)$ is also in first normal form. Let $\rho(n)$ be the set of first-normal-form preserving permutations. Then $\rho(n)$ partitions the set of candidates for the second level of a comparator network into equivalence classes; two candidates L_1 and L_2 are in the same equivalence class iff there exists $\Pi \in \rho(n)$ such that $\Pi(L_1) = \Pi(L_2)$. A level L is called *representative* if L has minimal lexicographic number within its equivalence class. A comparator network is said to be in *second normal form* if it is in first normal form and its second level is representative.

Theorem 5.4 *If there is an n-input sorting network of depth d, then there is an n-input sorting network of depth d that is in second normal form.*

PROOF: Let C be a sorting network of depth d. Without loss of generality we can assume that C is in first normal form (by Theorem 5.3). Clearly, there exists a permutation $\Pi \in \rho(n)$ such that $\Pi(C)$ is a generalized comparator network in second normal form. By Lemma 5.2, $\Pi(C)$ is a generalized sorting network. Therefore, by Theorem 5.1, there exists a sorting network of depth d that is in second normal form. □

The representative perfect matchings which are candidates for the second level of a second normal form sorting network with n inputs can be pre-computed by generating equivalence classes by brute force. The number of representatives, denoted $R(n)$, is easily computed for $n \leq 10$ (see Table 3). We are particularly interested in $R(9) = 58$. This includes the lexicographically first perfect matching (which is in an equivalence class by itself). This is not a candidate for the second level of a first normal form sorting network since it is the same as the first level and thus could be deleted to give a sorting network of depth 5, which is known to be impossible (see Table 1).

It may at first seem puzzling that for large enough odd n, $R(n) > R(n+1)$. Although the number of matchings is the same in both cases, there are more equivalence classes in the

former case than the latter, since the equivalence classes in the former case are in general smaller. This is due to the fact that first-normal-form preserving permutations must leave the last channel (which is the one without a comparator on the first level in first normal form) invariant when the number of inputs is odd.

6 Testing Sorting Networks

We store the current comparator network $C = (L_1, \ldots, L_d, n)$ using two two-dimensional arrays max and min, in order to make the testing process as efficient as possible. For $1 \leq i \leq n$ and $1 \leq k \leq d$,

$$\mathsf{min}[i,k] = \begin{cases} j & \text{if there exists } j < i \text{ such that } \langle j, i \rangle \in L_k \\ i & \text{otherwise} \end{cases}$$

and $\mathsf{max}[i,k]$ is defined similarly. Intuitively, if there is a comparator on channel i at level k then $\mathsf{min}[i,k]$ is the channel on which the minimum output of the comparator emerges, and $\mathsf{max}[i,k]$ is the channel on which the maximum output of the comparator emerges. It is a fairly straightforward task to translate from the perfect matching representation used in Section 5 to this new representation, or rather, to update the new representation of the current comparator network as changes are required.

6.1 The Zero-one Principle

One obvious way to verify that a comparator network sorts is to try all permutations of the numbers one through n as inputs. Fortunately, many fewer trials are sufficient. The following result is often called the *zero-one principle*.

Theorem 6.1 *(Bouricious) An n-input comparator network is a sorting network iff it sorts all sequences of n zeros and ones.*

PROOF: See Knuth [7] or Parberry [9]. □

By Theorem 6.1, we only need to check whether a comparator network sorts all 2^n zero-one inputs. Suppose we check them in *binary reflected Gray code* order. This ordering of the inputs has the useful property that each input differs from the previous one in exactly one bit. Bitner, Ehrlich and Reingold [4] provide algorithms for generating the n-bit binary reflected Gray code in time $O(2^n)$ (that is, linear in the number of inputs generated). We will assume the existence of three sub-algorithms; procedure firstinput which initializes the appropriate global variables to represent the all-zero input, function finished which returns true if the global variables indicate that all inputs have been generated, and procedure nextinput which updates the appropriate global variables to represent the next input in binary reflected Gray code order. In particular, a call to nextinput(channel,delta,zeros) will set variable channel to be the index of the bit which was changed (in the range 1 through n), delta to be the value that it was changed to, and zeros to be the number of zeros in the current input. The details of these sub-algorithms are left to the interested reader. The following Boolean function sorts returns true if the comparator network represented by the arrays min and max is a sorting network.

```
1.   function sorts(min,max);
2.     begin
3.       for i := 1 to n do
4.         for k := 1 to d do
5.           ones[i, k] := 0;
6.       firstinput;
7.       nextinput(channel,delta,zeros);
8.       failed := false;
9.       while not failed and not finished do
10.        begin
11.          failed := not sortscurrent(channel,delta,zeros,min,max,ones);
12.          nextinput(channel,delta,zeros)
13.        end
14.      return(not failed)
15.    end;
```

Function sorts uses a Boolean function sortscurrent which returns true if the current comparator network sorts the current input. In particular, sortscurrent(channel, delta,zeros,min,max,ones) assumes that:

1. The comparator network represented by the arrays min and max (which we will call the "current comparator network") sorts the previous input.

2. The array ones represents the state of the current comparator network on the previous input in the sense that ones$[i, k]$ contains the number of ones which are input to the comparator on channel i at level k if such a comparator exists, and is undefined otherwise.

3. The current input differs from the previous one in the bit whose index is given as the parameter channel. The modified bit was changed to the value given as the parameter delta. The number of zeros in the current input is given as the parameter zeros.

If $C = (L_1, L_2, \ldots, L_d, n)$ is a comparator network of size s and depth d, the *graph* of C, denoted $G(C)$, is defined as follows. It has a vertex for each channel at every level, and edges which indicate possible routes for the values being sorted. More formally, $G(C) = (V, E)$ where

$$V = \{(i, j) \mid 1 \leq i \leq n,\ 0 \leq j \leq d\},$$

and $((i_1, j_1), (i_2, j_2)) \in E$ iff $i_2 = i_1 + 1$ and either $j_1 = j_2$ or $\langle j_1, j_2 \rangle \in L_{i_2}$ or $\langle j_2, j_1 \rangle \in L_{i_2}$. If $x = \langle x_1, x_2, \ldots, x_n \rangle \in \{0, 1\}^n$, define the *value* of a vertex $(i, j) \in V$ on input x, denoted $v_x(i, j)$, to be the value of C in channel i at level j on input x (see Section 2). Let $G(C, x)$ denote the labelled graph obtained by labelling the vertices of $G(C)$ with their respective values on input x.

Lemma 6.2 *Let $g(x)$ be the input which follows x in the binary reflected Gray code ordering. Then $G(C, g(x))$ differs from $G(C, x)$ only in the labels of a sequence of vertices $(0, j_0), (1, j_1), \ldots, (d, j_d)$ which form a path in $G(C)$.*

PROOF: The result follows by induction on d. Suppose that $d = 1$. Suppose that $g(x)$ differs from x in the ith bit. There are three cases to consider.

Case 1. $\langle i,k\rangle \in L_1$ for some $i < k \leq n$. Let $v_i = v_x(\langle i,0\rangle)$ and $v_k = v_x(\langle k,0\rangle)$. There are four sub-cases to consider.

1. $v_i = 0$, $v_k = 0$. Then $v_{g(x)}(\langle i,1\rangle) = 0$ and $v_{g(x)}(\langle k,1\rangle) = 1$.
2. $v_i = 0$, $v_k = 1$. Then $v_{g(x)}(\langle i,1\rangle) = 1$ and $v_{g(x)}(\langle k,1\rangle) = 1$.
3. $v_i = 1$, $v_k = 0$. Then $v_{g(x)}(\langle i,1\rangle) = 0$ and $v_{g(x)}(\langle k,1\rangle) = 0$.
4. $v_i = 1$, $v_k = 1$. Then $v_{g(x)}(\langle i,1\rangle) = 0$ and $v_{g(x)}(\langle k,1\rangle) = 1$.

The required result follows since $(\langle i,0\rangle, \langle i,1\rangle), (\langle i,0\rangle, \langle k,1\rangle) \in L_1$.
Case 2. $\langle k,i\rangle \in L_1$ for some $1 \leq k < i$. The proof is similar to Case 1.
Case 3. $\langle i,k\rangle, \langle k,i\rangle \notin E$ for all $k \neq i$. The required result follows since $(\langle i,0\rangle, \langle i,1\rangle) \in L_1$.

This completes the proof for $d = 1$. Now suppose that $d > 1$ and the hypothesis holds for all sorting networks of depth $d-1$. By an argument similar to the above, the sequence

$$\langle v_{g(x)}(\langle 1,1\rangle), v_{g(x)}(\langle 2,1\rangle), \ldots, v_{g(x)}(\langle n,1\rangle)\rangle$$

differs from

$$\langle v_x(\langle 1,1\rangle), v_x(\langle 2,1\rangle), \ldots, v_x(\langle n,1\rangle)\rangle$$

in exactly one bit. The required result then holds by the induction hypothesis. □

Thus in procedure sortscurrent(channel,delta,zeros,min,max,ones), changing the bit numbered channel in the current input to the value delta will cause $2d$ values in ones to change in a pattern which corresponds to a path in the graph of the current comparator network.
Observation 1. Note that (by a case analysis) the changed bit affects the max output of the comparator on its channel iff the number of ones input to that comparator on input x, plus one if the changed bit was changed to a one, equals one (and affects the min-output of that comparator otherwise).
Observation 2. It is particularly easy to check whether the current comparator network sorts the current input since we need only verify that the value on channel zeros + delta changes at the outputs.

Thus procedure sortscurrent can be implemented as follows. A variable current contains the channel whose value is changed at the current level; the current level is contained in variable level. By Lemma 6.2, there will be only one such channel at each level.

```
1.   function sortscurrent(channel,delta,zeros,min,max,ones);
2.     begin
3.       current := channel;
4.       for level := 1 to d do
5.         if there is a comparator on the current channel at this level then
6.           begin
7.             minimum := min[level,current];
8.             maximum := max[level,current];
9.             newones := ones[level,current] +2 · delta − 1;
10.            if ones[level,current] + delta = 1
11.              then current := maximum
12.              else current := minimum;
13.            ones[level,minimum] := newones;
14.            ones[level,maximum] := newones;
15.          end
16.      return(current = zeros + delta)
17.    end;
```

Line 3 initializes the current channel to be the channel that has its value changed at level 0. Lines 4-15 set the current channel to the channel whose value is changed at levels 1 through d in turn. Lines 7 and 8 compute the channels on which the min-output and max-output of the comparator on the current channel at the current level emerge. Line 9 computes the new number of ones which are input to that comparator (one more than before if delta = 1 and one less than before if delta = 0). Lines 10-12 update the current channel (see Observation 1 above), and lines 13 and 14 update the number of ones input to the current comparator using the value computed in line 9. Line 16 computes the final result (see Observation 2).

The correctness of procedure sortscurrent can be proved by induction on the number of iterations of the for-loop. Clearly it runs in time $O(d)$. Therefore procedure sorts runs in time $O(2^n d)$. The sorting network verification problem is co-NP complete even for shallow sorting networks (Parberry [11]), hence it seems unlikely that a sub-exponential time algorithm for checking optimal sorting networks is possible. However, a significant savings in time can be made by noticing that we are checking only comparator networks in first normal form. It is sufficient to check only the $3^{n/2}$ strings of $n/2$ bit-pairs with each pair either 00, 01, or 11, since the first level of such a comparator network permutes all bit-strings into this form. Generation of the appropriate inputs can be achieved by generating the $3^{n/2}$ strings of $n/2$ ternary digits (the ternary digit 0 represents the binary string 00, the ternary digit 1 represents the binary string 01 and the ternary digit 2 represents the binary string 11) in *ternary reflected Gray code order* in linear time by modifying the algorithms of Bitner, Ehrlich and Reingold [4]. Thus the time to check each first normal form comparator network is $O(3^{n/2}d)$.

6.2 The Last Level

The last level of a sorting network can be constructed on-the-fly during the testing process if we use the algorithms of Section 6.1. We test each comparator network of depth $d-1$ to see if one more level can be added to make it a sorting network, as follows. Given a comparator network of depth $d-1$, check whether it sorts the current input. If so, then go

on to check the next input. Note that the subsequent addition of more comparators at level d will not change its ability to sort this input. If it does not sort the current input, that is, current $\neq$ zeros + delta in line 16 of function sortscurrent (see Section 6.1), then insert a comparator between channels current and zeros + delta on level d. If this is acceptable, then go on to check the next input. If it is not acceptable (that is, there is already a comparator on one or both of the channels at level d, inserted while testing a previous input), then reject the current comparator network. In this case the current comparator network of depth $d-1$ cannot be completed to give a sorting network of depth d.

7 The Heuristic

If c is a channel, let $to(c)$ be the set of channels which can send a value to channel c in a two-level comparator network. Similarly, let $from(c)$ be the set of channels which can receive a value from channel c in a two-level comparator network. More formally, if (L_1, L_2, n) is a two-level comparator network, and $S \subseteq \{1, 2, \ldots, n\}$, define for $i = 1, 2$,

$$connect_i(S) = S \cup \{t \mid \text{there exists } s \in S \text{ such that } \langle s, t\rangle \in L_i \text{ or } \langle t, s\rangle \in L_i\}.$$

Then $to(c) = connect_1(connect_2(\{c\}))$, and $from(c) = connect_2(connect_1(\{c\}))$. Clearly $|to(c)| \leq 4$ and $|from(c)| \leq 4$. Furthermore:

Lemma 7.1 *For any channel c, $|to(c) \cup from(c)| \leq 5$.*

Proof: Let $C = (L_1, L_2, n)$ be a two-level comparator network, and $1 \leq c \leq n$. Without loss of generality, we will assume that C has maximal size. Suppose $to(c) = \{c, c_1, c_2, c_3\}$, where there is a comparator between channels c_3 and c in L_1, there is a comparator between channels c_1 and c_2 in L_1, and there is a comparator between channels c_2 and c in L_2. Then $c, c_2, c_3 \in from(c)$. Therefore $|to(c) \cup from(c)| \leq 5$. □

The bound of Lemma 7.1 is easily seen to be tight.

When generating depth d comparator networks, we pause after generation of level $d-2$ and apply a further test to the first $d-2$ levels of the network. We call the latter a *stub*. If the stub fails the test, we move on to the next stub. Otherwise, we continue as before (generating all possible candidates for level $d-1$ in turn and testing whether the network can be completed by the construction of level d to give a sorting network). This test prunes branches of the search tree at bad stubs. We must ensure that our test rejects only bad stubs, that is, stubs that cannot be extended to sorting networks by the addition of two levels of comparators. Stubs are tested by using the algorithms of Section 6 to construct for each channel c a set $reach(c)$ which consists of those channels that must either send a value to channel c in the last two levels or receive a value from channel c in the last two levels. That is, $reach(c)$ consists of those channels d such that for some input, in line 16 of function sortscurrent, either $c =$ current and $d =$ zeros + delta or vice-versa. By Lemma 7.1 we can reject stubs that have a channel c with $|reach(c)| > 5$. Although the test does not reject good stubs, it will not make a useful heuristic unless very few bad stubs pass. Preliminary experiments indicated that this would be the case.

8 Conclusion

The algorithms described in this paper were initially coded in Berkeley Unix Pascal and tested on a VAX 11/780. A moderate increase in speed (around 20%) was obtained by using the programming language C.

Some further small optimizations were implemented in the final version, for example, the search tree is pruned slightly by keeping track of redundant comparators. If a comparator is duplicated on two consecutive levels, then the second occurrence can be removed. As observed in Section 3, a 9-input sorting network of depth 6 can have only one redundant comparator. The search tree can be pruned at comparator networks which have more redundant comparators.

The final version of the program was run at low priority on a CRAY-2 supercomputer, and terminated correctly without finding any nine-input sorting networks of depth six. We have shown that if any such sorting network existed, then the program would have found a canonical version in the candidates that it tested. Therefore we conclude that no such sorting network exists. The program has also been used to enumerate the ten-input sorting networks of depth seven. A total of 87 sorting networks were found, including the previously known network listed by Knuth [7].

9 Acknowledgements

The author wishes to thank Bruce Parker for assistance with the implementation of the improvements to the sorting network testing routine using the ternary reflected Gray code (see Section 6.1), construction of the last level of the candidate comparator network on-the-fly (see Section 6.2), and for the translation of the source code from Pascal to C.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pages 1–9, April 1983.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–48, 1983.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, April 1968.

[4] J. R. Bitner, G. Ehrlich, and E. M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Commun. ACM*, 19(9):517–521, September 1976.

[5] R. C. Bose and R. J. Nelson. A sorting problem. *J. Assoc. Comput. Mach.*, 9:282–296, 1962.

[6] R. W. Floyd and D. E. Knuth. The Bose-Nelson sorting problem. In J. N. Srivastava, editor, *A Survey of Combinatorial Theory*. North-Holland, 1973.

[7] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[8] I. Parberry. The alternating sorting network. Technical Report CS-87-26, Dept. of Computer Science, Penn. State Univ., September 1987.

[9] I. Parberry. *Parallel Complexity Theory.* Research Notes in Theoretical Computer Science. Pitman Publishing, London, 1987.

[10] I. Parberry. A computer-assisted optimal depth lower bound for sorting networks with nine inputs. In *Proceedings of Supercomputing '89*, pages 152–161, 1989.

[11] I. Parberry. Single-exception sorting networks and the computational complexity of optimal sorting network verification. *Mathematical Systems Theory*, 23:81–93, 1990.

[12] D. C. Van Voorhis. An improved lower bound for sorting networks. *IEEE Trans. Comput.*, C-21:612–613, June 1972.