

 Open access • Book Chapter • DOI:10.1007/978-3-642-17511-4_10

Pairwise cardinality networks — Source link

Michael Codish, Moshe Zazon-Ivry





Institutions: Ben-Gurion University of the Negev

Published on: 25 Apr 2010 - International Conference on Logic Programming

Topics: Cardinality (SQL statements), Sorting network and Pairwise comparison

Related papers:

- [Translating Pseudo-Boolean Constraints into SAT](#)
- [Cardinality Networks: a theoretical and empirical study](#)
- [Efficient CNF encoding of Boolean cardinality constraints](#)
- [An Extensible SAT-solver](#)
- [Towards an optimal CNF encoding of Boolean cardinality constraints](#)

Share this paper:    

View more about this paper here: <https://typeset.io/papers/pairwise-cardinality-networks-sdblyz06>

Pairwise Cardinality Networks^{*}

Michael Codish and Moshe Zazon-Ivry

Department of Computer Science, Ben-Gurion University, Israel
{mcodish,moshezaz}@cs.bgu.ac.il

Abstract. We introduce pairwise cardinality networks, networks of comparators, derived from pairwise sorting networks, which express cardinality constraints. We show that pairwise cardinality networks are superior to the cardinality networks introduced in previous work which are derived from odd-even sorting networks. Our presentation identifies the precise relationship between odd-even and pairwise sorting networks. This relationship also clarifies why pairwise sorting networks have significantly better propagation properties for the application of cardinality constraints.

1 Introduction

Cardinality constraints take the form, $x_1 + x_2 + \dots + x_n \prec k$, where x_1, \dots, x_n are Boolean variables, k is a natural number, and \prec is one of $\{<, \leq, >, \geq, =\}$. Cardinality constraints are well studied and arise in many different contexts. One typical example is the Max-SAT problem where for a given propositional formula (in CNF) with clauses $\{C_1, \dots, C_n\}$, we seek an assignment that satisfies a maximal number of clauses. One approach is to add a fresh blocking variable to each clause giving $\varphi = \{C_1 \vee x_1, \dots, C_n \vee x_n\}$. Now we seek a minimal value k such that $\varphi \wedge (x_1 + x_2 + \dots + x_n < k)$ is satisfiable. We can do this by encoding the cardinality constraint to a propositional formula ψ_k and repeatedly applying a SAT solver to find the smallest k such that $\varphi \wedge \psi_k$ is satisfiable.

There are many works that describe techniques to encode cardinality constraints to propositional formulas. The starting points for this paper are the work by Asin *et al.*[1] and an earlier paper [4] which describes how pseudo Boolean constraints (which are more general than cardinality constraints) are translated to SAT in the MiniSAT solver. Both of these papers consider an encoding technique based on the use of sorting networks.

A (Boolean) sorting network is a circuit that receives n Boolean inputs x_1, \dots, x_n , and permutes them to obtain the sorted outputs y_1, \dots, y_n . The circuit consists of a network of comparators connected by “wires”. Each comparator has two inputs, u_1, u_2 and two outputs, v_1, v_2 . The “upper” output, v_1 , is the maximal value on the inputs, $u_1 \vee u_2$. The “lower” output, v_2 , is the minimal value, $u_1 \wedge u_2$. In brief, naming the wires between the comparators as

^{*} Supported by the G.I.F. grant 966-116.6 and by the Frankel Center for Computer Science at the Ben-Gurion University.

propositional variables, we can view a sorting network with inputs x_1, \dots, x_n and outputs y_1, \dots, y_n as a propositional formula, ψ , obtained as the conjunction of its comparators. In the context of the Max-SAT problem described above the formula ψ_k expressing the cardinality constraint $x_1 + x_2 + \dots + x_n < k$, is obtained by sorting the x_1, \dots, x_n (in decreasing order), and setting the k^{th} largest output to 0. As the outputs are sorted, this implies that all outputs from position k are zero and hence that there are less than k ones amongst the input values. If the outputs are y_1, \dots, y_n , the cardinality constraint is expressed as $\psi_k = \psi \wedge \neg y_k$.

Adding a clause $\neg y_k$ to the formula ψ that represents a sorting network with input wires x_1, \dots, x_n and output wires y_1, \dots, y_n , assigns a value (zero) to an output wire. This, in turn, imposes constraints on other input and output wires. So in some sense, we are running the network backwards. That sorting networks are bi-directional is well-understood, see for example [2]. However, this point is often overlooked. It is this bi-directionality that impacts the choice of sorting network construction best suited for application to cardinality constraints.

Sorting networks have been intensively studied since the mid 1960's. For an overview see for example, Knuth [5], or Parberry [6]. One of the best sorting network constructions, and possibly the one used most in applications, is due to Batcher as presented in [2]. Parberry [7] describes this network as follows:

For all practical values of $n > 16$, the best known sorting network is the odd-even sorting network of Batcher, which is constructed recursively and has depth $(\log n)(\log n + 1)/2$ and size $n(\log n)(\log n - 1)/4 + n - 1$.

The presentations in [1] and in [4] describe the use of odd-even sorting networks to encode cardinality constraints. In this paper we take a similar approach but apply the so called “*pairwise sorting network*” instead. Parberry [7], introduces the pairwise network:

It is the first sorting network to be competitive with the odd-even sort for all values of n . The value of the pairwise sorting network is not that it is superior to the odd-even sorting network in any sense, but that it is the first serious rival to appear in over 20 years.

In this paper, almost 20 years after the introduction of pairwise sorting networks, we are in the position to state that pairwise sorting networks are significantly superior to odd-even sorting networks, at least for encoding cardinality constraints. To obtain our results, we first observe that both types of networks are composed of two main components, which we term: *pairwise splitters* and *pairwise mergers*. Usually, the odd-even sorting network is viewed as a network of *odd-even mergers*. However, we show that each such odd-even merger is precisely equivalent to the composition of a pairwise splitter and a pairwise merger. Consequently, in the odd-even sorting network, pairwise splitters and mergers alternate, whilst in the pairwise network, the splitters and mergers occur in separate blocks, splitters before mergers.

The precise and clear presentation of the relationship between pairwise and odd-even sorting networks is new and is a contribution on its own right. It is also the basis for our results. As we illustrate in the rest of the paper, the splitters inhibit the propagation of data from the sorted outputs y_1, \dots, y_n towards the original inputs x_1, \dots, x_n occurring in the cardinality constraint. Hence, when encoding cardinality constraints, the splitters are best positioned closer to the x_i 's than to the y_i 's.

Cardinality networks, similar to sorting networks, are networks of comparators that express cardinality constraints. As described above, a cardinality network is easily constructed using a sorting network. However, both the odd-even as well as the pairwise sorting networks involve $O(n \log^2 n)$ comparators to sort n bits. Earlier work describes cardinality networks of size $O(n \log^2 k)$ for constraints of the form $x_1 + x_2 + \dots + x_n < k$ which is an improvement for the typical case where k is considerably smaller than n . For example, in [8], Wah and Chen illustrate a $O(n \log^2 k)$ construction for the (k, n) selection problem which is to select the k smallest (or largest) elements from a set of n numbers. Also in [1], Asin *et al.* define a simplified (odd-even) merge component and illustrate its application to construct a cardinality network of size $O(n \log^2 k)$.

In this paper we show that when expressing a cardinality constraint $x_1 + x_2 + \dots + x_n < k$ in terms of a pairwise or an odd-even sorting network, the network collapses automatically (by partial evaluation) to a cardinality network with $O(n \log^2 k)$ comparators. No further construction is required. Experimental evaluation indicates that the choice of a pairwise sorting network results in smaller encodings.

In Section 2 we present the classic odd-even and pairwise sorting networks. Section 3 clarifies a precise relationship between these two types of networks. This simplifies, and perhaps demystifies, the presentation of the pairwise network. In Section 4 we show that a cardinality network of size $O(n \log^2 k)$ is derived by partial evaluation from a pairwise sorting network. Section 5 presents a preliminary experimental evaluation and Section 6 concludes.

2 Sorting Networks - from Batcher to Parberry

Sorting networks were originally described as circuits, composed using comparators, which given values on their input positions compute (in parallel) the sorted values on their output positions. In the context of cardinality networks, it is beneficial to view sorting networks as relations between the “input” and “output” positions. Given such a relation, and values for some of the (input or output) positions, we seek values for the rest of the positions that satisfy the relation.

We represent a comparator as a relation $comparator(a, b, c, d)$ where intuitively a and b are two Boolean inputs and $\langle c, d \rangle$ a permutation of $\langle a, b \rangle$ with $c \geq d$. More formally, the relation is defined as follows:

$$comparator(a, b, c, d) \leftrightarrow (c \leftrightarrow a \vee b) \wedge (d \leftrightarrow a \wedge b)$$

A network of comparators is a conjunction of their corresponding relations. A sorting network is a relation on tuples of Boolean variables expressed in terms of

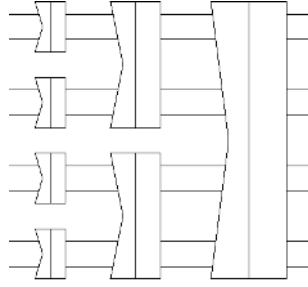


Fig. 1. An odd-even sorting network as a network of odd-even mergers

a network of comparators. To ease presentation, we will assume that the lengths of tuples are powers of 2.

The Odd-Even Sorting Network

The odd-even sorting network, due to Batcher [2], is based on the merge-sort approach: to sort a list of $2n$ values, first partition the list into two lists with n values each, then recursively sort these two lists, and finally merge the two sorted lists. The network is termed “odd-even” because of the way the merge component is defined.

We present the odd-even merge component as a ternary relation on tuples of Boolean values. The relation $\text{OEMerge}(A, B, C)$ is defined as a conjunction of comparators and expresses that merging sorted lists A and B , each of length n gives sorted list C , of length $2n$. The relation is defined for $n = 1$ and then recursively for $n > 1$ as follows:

$$\begin{aligned} \text{OEMerge}(\langle a_1 \rangle, \langle b_1 \rangle, \langle c_1, c_2 \rangle) &\leftrightarrow \text{comparator}(a_1, b_1, c_1, c_2). \\ \text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle d_1, c_2, \dots, c_{2n-1}, e_n \rangle) &\leftrightarrow \\ &\text{OEMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \wedge \\ &\text{OEMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_n \rangle) \wedge \\ &\bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}). \end{aligned}$$

The odd-even sorting network [2] is a binary relation on sequences of length 2^m and is defined as follows for $m = 0$ and recursively for $m > 0$. For $m > 0$ we denote the sequence length $2^m = 2n$.

$$\begin{aligned} \text{OESort}(\langle a_1 \rangle, \langle a_1 \rangle). \\ \text{OESort}(\langle a_1, \dots, a_{2n} \rangle, \langle c_1, \dots, c_{2n} \rangle) &\leftrightarrow \\ &\text{OESort}(\langle a_1, \dots, a_n \rangle, \langle d_1, \dots, d_n \rangle) \wedge \\ &\text{OESort}(\langle a_{n+1}, \dots, a_{2n} \rangle, \langle d'_1, \dots, d'_n \rangle) \wedge \\ &\text{OEMerge}(\langle d_1, \dots, d_n \rangle, \langle d'_1, \dots, d'_n \rangle, \langle c_1, \dots, c_{2n} \rangle). \end{aligned}$$

The recursive definition of an odd-even sorting network unravels to a network of merge components. Figure 1 illustrates the network that defines the relation

between 8 unsorted “inputs”, on the left, and their 8 sorted “outputs”, on the right. Each depicted odd-even merger represents a relation between two sorted sequences of length n (on its left side) and their sorted merge of length $2n$ (on its right side).

The Pairwise Sorting Network

The pairwise sorting network, due to Parberry [7], is also based on the merge-sort approach but with one simple, yet influential, difference in the first stage of the construction: To sort a list of $2n$ values, first split the list “pairwise” into two lists, the first with the n “larger” values from these pairs, and the second with the n smaller values. The resulting network is termed “pairwise” because of the way the elements to be sorted are pairwise split before recursively sorting the two parts and merging the resulting sorted lists.

$$\text{PWSplit}(\langle a_1, \dots, a_{2n} \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_n \rangle) \leftrightarrow \bigwedge_{1 \leq i \leq n} \text{comparator}(a_{2i-1}, a_{2i}, b_i, c_i).$$

The pairwise sorting network [7] is a binary relation on sequences of length 2^m and is defined as follows for $m = 0$ and recursively for $m > 0$. For $m > 0$ we denote the sequence length $2^m = 2n$.

$$\begin{aligned} & \text{PWSort}(\langle a_1 \rangle, \langle a_1 \rangle). \\ & \text{PWSort}(\langle a_1, \dots, a_{2n} \rangle, \langle d_1, \dots, d_{2n} \rangle) \leftrightarrow \\ & \quad \text{PWSplit}(\langle a_1, \dots, a_{2n} \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_n \rangle) \wedge \\ & \quad \text{PWSort}(\langle b_1, \dots, b_n \rangle, \langle b'_1, \dots, b'_n \rangle) \wedge \\ & \quad \text{PWSort}(\langle c_1, \dots, c_n \rangle, \langle c'_1, \dots, c'_n \rangle) \wedge \\ & \quad \text{PWMerge}(\langle b'_1, \dots, b'_n \rangle, \langle c'_1, \dots, c'_n \rangle, \langle d_1, \dots, d_{2n} \rangle). \end{aligned}$$

The description of the pairwise merger (PWMerge) given by Parberry in [7] is not straightforward to follow. We provide a simple specification of the PWMerge relation in the next section. For now, let us note a property of the pairwise merger by comparison to the odd-even merger. Consider the merging of two lists of bits, $\langle a_1, \dots, a_n \rangle$ and $\langle b_1, \dots, b_n \rangle$. The odd-even merger assumes only that the two lists are sorted. In contrast, the pairwise merger assumes also that each pair $\langle a_i, b_i \rangle$ is sorted “internally”. Namely, that $a_i \geq b_i$. Indeed, in [7], the pairwise merger is called: “sorting sorted pairs”.

The recursive definition of a pairwise sorting network unravels to a network of split and merge components. Figure 2 illustrates the network that defines the relation between 8 unsorted “inputs”, on the left, and their 8 sorted “outputs”, on the right. Scanning from left to right, we first have the network of splitters and then a network of mergers.

3 Sorting Networks - from Parberry back to Batcher

In this section we clarify a precise relationship between the odd-even and pairwise sorting networks. While very simple, this relationship is not to be found in the

literature. This relationship will provide the basis for our argument that pairwise sorting networks are significantly better than odd-even networks in the context of cardinality constraints. Parberry, in [7], provides little insight when stating that (1) “one can prove by induction on n that the n input pairwise sorting network is not isomorphic to the odd-even sorting network”; and (2) “it is also easy to prove that the pairwise sorting network has the same size and depth bounds as Batcher’s odd-even sorting network”.

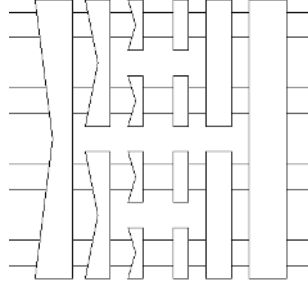


Fig. 2. A pairwise sorting network as a network of splitters and pairwise mergers

We first introduce a simple recursive definition for the pairwise merger and claim that it is indeed a suitable pairwise merger and that it has the same size and depth bounds as Parberry’s pairwise merger.

Theorem 1 (pairwise merge). *Consider the following specification of a pairwise merger for merging sequences of length n defined for $n = 1$ and then recursively for $n > 1$.*

$$\begin{aligned} & \text{PWMerge}(\langle a_1 \rangle, \langle b_1 \rangle, \langle a_1, b_1 \rangle). \\ & \text{PWMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle d_1, c_2, \dots, c_{2n-1}, e_n \rangle) \leftrightarrow \\ & \quad \text{PWMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \quad \wedge \\ & \quad \text{PWMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_n \rangle) \quad \wedge \\ & \quad \bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}). \end{aligned}$$

This specification is: (1) a correct merger for the pairwise sorting network; (2) a network of depth $\log n$ and size $n \log n - n + 1$; and (3) isomorphic to the iterative specification given by Parberry in [7] (pg.4).

Proof. (sketch) For (1) we need to show that if $\langle a_1, \dots, a_n \rangle$ and $\langle b_1, \dots, b_n \rangle$ are sorted sequences and for each position $a_i \geq b_i$, then $\langle d_1, c_2, \dots, c_{2n-1}, e_n \rangle$ is sorted (and has the same total number of 1’s and 0’s as in $\langle a_1, \dots, a_n \rangle$ and $\langle b_1, \dots, b_n \rangle$). This follows by a simple induction. Showing (2), is also straightforward, solving $S(1) = 0$ and $S(n) = 2S(n/2) + (n - 1)$ for the size, and $D(1) = 0$, $D(n) = D(n/2) + 1$ for the depth. (3) Follows by induction on n , which we assume to be a power of 2. Assuming that the two recursive calls to PWMerge

are isomorphic to their iterative specifications, we have that they each consist in $(\log_2 n - 1)$ layers generated by the iterations of Parberry's specification. Each respective pair of these layers (one from the odd call and one from the even call) are shown to combine to give a corresponding layer of the full network. The last iteration layer in the full network is precisely that introduced by the conjunction $\bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1})$.

We now proceed to observe the relationship between the pairwise and odd-even mergers.

Theorem 2 (odd-even merge). *An odd-even merger is equivalent to the composition of a pairwise split and a pairwise merger. For $n \geq 1$,*

$$\begin{aligned} \text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{2n} \rangle) &\leftrightarrow \\ \text{PWSplit}(\langle a_1, b_1, a_2, b_2, \dots, a_n, b_n \rangle, \langle a'_1, a'_2, \dots, a'_n \rangle, \langle b'_1, b'_2, \dots, b'_n \rangle) &\wedge \\ \text{PWMerge}(\langle a'_1, a'_2, \dots, a'_n \rangle, \langle b'_1, b'_2, \dots, b'_n \rangle, \langle c_1, \dots, c_{2n} \rangle). \end{aligned}$$

Proof. (brief description) The proof is by induction on n and follows the structure of the definition of OEMerge where the theorem holds for the recursive calls to OEMerge. (See Appendix.)

Figure 3 illustrates the relationship between the two types of sorting networks. There are three networks in the figure. On the left, we have a pairwise sorting network (of size 8). In the middle of this network, the column of four splitters (each of size 2) followed by the column of four pairwise mergers (also of size 2), form together a single column of four odd-even mergers, each of which is a 2×2 sorter. These 2×2 sorters are boxed in the left network. Now, the column of two splitters (each of size 4) can migrate to the right, because splitting before or after sorting has the same effect. This migration results in the middle network of the figure. In this middle network we now have, in the middle of the network, two 4×4 odd-even sorters. These 4×4 sorters are boxed in the middle network. The transition to the right network is once again by migrating a splitter over the two smaller sorting networks. This results in an 8×8 odd-even network. Figure 4 provides a high-level perspective on this transition. The left

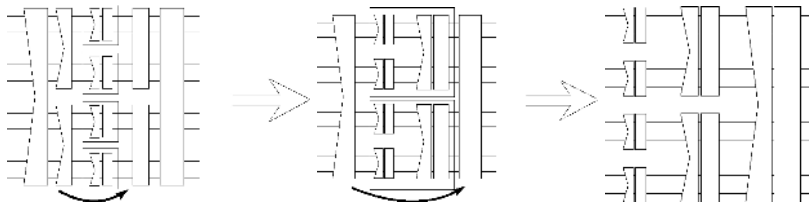


Fig. 3. Migration of splitters: from pairwise to odd-even

is a pairwise sorting network, composed of a split component followed by two recursive sorting networks and finishing with a pairwise merger. The splitter can

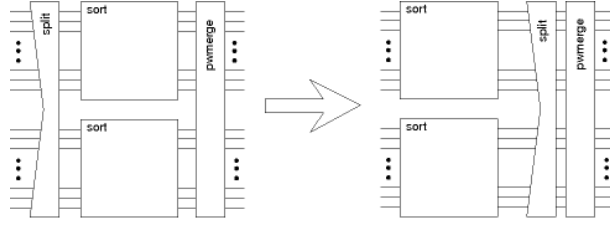


Fig. 4. High-level view on: from pairwise to odd-even

be migrated to the right, as splitting and sorting (twice) has the same effect as (twice) sorting and then splitting.

In [4] and in [1], the authors prove that for the CNF encodings of cardinality constraints using odd-even sorting networks, unit propagation preserves arc consistency. This means that for a constraint of the form $x_1 + x_2 + \dots + x_n < k$, as soon as $k - 1$ of the x_i variables have become true, then the rest will become false by unit propagation. If $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$ are the inputs and outputs of an odd-even sorting network, it means that setting any $k - 1$ variables from $\langle x_1, \dots, x_n \rangle$ the value 1, then by unit propagation the first $k - 1$ variables in $\langle y_1, \dots, y_n \rangle$ will be assigned the value 1. Moreover, if also $y_k = 0$ then the rest of the $n - k + 1$ variables from $\langle x_1, \dots, x_n \rangle$ will be assigned the value 0 by unit propagation. We note that pairwise sorting networks enjoy all of the same arc consistency properties as do the odd-even networks. The proofs of these claims are similar to those presented in [1]. We will see that pairwise networks enjoy one additional propagation property which does not hold for the odd-even networks.

4 The Pairwise Cardinality Network

In this section we show how the application of a pairwise sorting network to encode a cardinality constraint $x_1 + x_2 + \dots + x_n < k$ can be collapsed to a network with $O(n \log^2 k)$ comparators. To obtain this result we first enhance the definition of the pairwise merger, adding a linear number of clauses which express that the outputs of the merger are sorted. These clauses are redundant in the sense that in any satisfying assignment of the formula representing a pairwise sorting network, the outputs of the mergers are sorted anyway. Their purpose is to amplify propagation. We focus on the case when $<$ is the less-than relation. We assume that n is a power of 2 and that $k \leq n/2 + 1$. Otherwise we can encode the dual constraint (counting the number of empty seats is easier than counting the passengers in an almost full aircraft). In general, it is common that k is significantly smaller than n and in the worst case $k = n/2 + 1$. The following definition specifies the enhanced pairwise merger.

Definition 1 (enhanced pairwise merger). *The enhanced pairwise merger is defined for $n \geq 1$*

$$\begin{aligned} \text{PWMerge}'(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n} \rangle) &\leftrightarrow \\ \text{PWMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n} \rangle) &\wedge \\ \text{sorted}(\langle c_1, c_2, \dots, c_{2n} \rangle). \end{aligned}$$

where

$$\text{sorted}(\langle c_1, \dots, c_{2n} \rangle) \leftrightarrow \bigwedge_{i=1}^{2n-1} (c_i \vee \neg c_{i+1}).$$

The pairwise cardinality network for $x_1 + x_2 + \dots + x_n < k$ is obtained as a pairwise sorting network with inputs x_1, \dots, x_n and outputs y_1, \dots, y_n . We assume that the pairwise mergers in the network are enhanced and we set the k^{th} output y_k to zero (by adding the clause $\neg y_k$). The network then collapses to a smaller network by propagating the known value $y_k = 0$ (backwards) through the network. In particular, from the rightmost enhanced pairwise merger we have $\text{sorted}(\langle y_1, \dots, y_n \rangle)$ and obtain from $y_k = 0$ by unit propagation $y_i = 0$ for $k < i \leq n$.

The following definition specifies how comparators may be eliminated due to partially known inputs. It works like this (we focus on the propagation of zero's): For each comparator, if the upper output value is zero then we can remove the comparator, setting the other output bit to zero as well as the two input bits; and if either one of the input bits is zero, then we can remove the comparator setting the lower output bit to zero while the upper output bit is identified with the other input bit. The elimination of comparators is a simple form of partial evaluation and applied at the comparator level, before representing the network as a CNF formula. Except for the identification of an output bit with an input bit (e.g. $b = c$ in the definition below), it could also be performed as unit propagation at the CNF level.

Definition 2 (partial evaluation of comparators).

$$\begin{aligned} \text{comparator}(a, b, c, d) \wedge \neg c &\models_{pe} \neg a \wedge \neg b \wedge \neg d \\ \text{comparator}(a, b, c, d) \wedge \neg a &\models_{pe} \neg d \wedge (b = c) \\ \text{comparator}(a, b, c, d) \wedge \neg b &\models_{pe} \neg d \wedge (a = c) \end{aligned}$$

Figure 5 illustrates an 8 by 8 pairwise sorter and details the comparators in each of the components (as vertical lines between the wires). The figure highlights the rightmost merger

$$\text{PWMerge}(\langle a_1, \dots, a_4 \rangle, \langle b_1, \dots, b_4 \rangle, \langle c_1, \dots, c_8 \rangle)$$

To express the cardinality constraint $x_1 + x_2 + \dots + x_8 < 4$, we set the output c_4 to 0. In the figure, applying partial evaluation, five of the nineteen comparators consequently become redundant (indicated by dashed lines) and can be removed. The reader is encouraged to check that in the corresponding odd-even network only 4 comparators can be eliminated.

The next two theorems express a propagation property of the pairwise merger. First, consider the relation $\text{PWMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n} \rangle)$

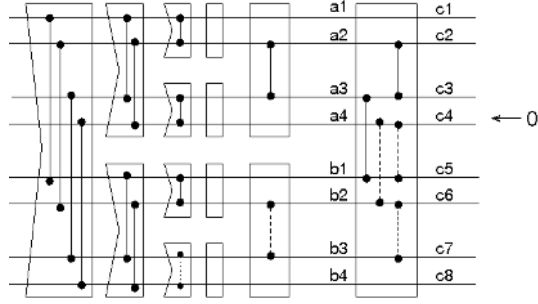


Fig. 5. A size 8 pairwise cardinality network (dashed comparators are redundant)

specified in Theorem 1. In the context of the pairwise sorting network, the inputs, upper $\langle a_1, \dots, a_n \rangle$ and lower $\langle b_1, \dots, b_n \rangle$ to the merger, are both sorted as sequences as well as sorted pairwise. It means that (in any satisfying assignment) there are at least as many ones in the upper inputs as in the lower inputs. Hence, if there are less than k ones in the output sequence (namely, $c_k = 0$), then there must be less than $\lceil k/2 \rceil$ ones in the lower input sequence (namely $b_{\lceil k/2 \rceil} = 0$). This facilitates constraint propagation in the design of the pairwise cardinality network. For example, in Figure 5, The zero on output c_4 of the rightmost merger propagates to a zero on its lower input, b_2 . This process continues as b_2 is also an output of the next merger (to the left).

The next theorem states that if there are less than k ones in the outputs of the merger, then it follows from the definition of the merger and partial evaluation alone that there are less than $\lceil k/2 \rceil$ ones in the lower set of inputs. The consequence itself is obvious for each merger in the context of the sorting network (because in any satisfying assignment there are more ones in the upper set of inputs than in the lower). That it follows (independent of the context of the merger) by partial evaluation is not obvious and is useful below to simplify a pairwise network. Note that the claim is with regards to the non-enhanced pairwise merger. Namely it does not rely on the fact that the outputs of the merger are sorted.

Theorem 3. For $k \leq n$,

$$\text{PWMerge} \left(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle \right) \wedge \left(\bigwedge_{i=k}^{2n} \bar{c}_i \right) \models_{pe} \left(\bigwedge_{j=\lceil k/2 \rceil}^n \bar{b}_j \right)$$

Proof. (brief description) The proof is by induction on n . It follows from the recursive definition of PWMerge focusing on the parity of k and of $\lceil k/2 \rceil$. (See Appendix.)

The next theorem is similar. It states that if there are less than k ones in the outputs of the merger, then it follows from partial evaluation that for the smallest $k' \geq k$ which is a power of 2, the k' -th bit in the upper set of inputs is

a zero. Once again, it is obvious that in the actual context of the merger in the sorting network, all of the inputs a_k, \dots, a_n will be set to zero. But the weaker result (for k') follows from the definition of the merger and partial evaluation alone.

Theorem 4. *Let $k \leq n$ and let k' be the smallest power of 2 that is greater or equal to k . Then*

$$\text{PWMerge} \left(\begin{array}{l} \langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \\ \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle \end{array} \right) \wedge \left(\bigwedge_{i=k}^{2n} \bar{c}_i \right) \models_{pe} \bar{a}_{k'}$$

The proof is similar to that of Theorem 3 but with fewer cases due to the fact that k' is a power of 2.

Proof. (by induction on n) The base case for $n = 1$ is trivial. For $n > 1$ assume that the statement holds for all $n' < n$ and denote $k' = 2p$. We have

$$\left(\bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \right) \wedge \left(\bigwedge_{i=k}^{2n} \bar{c}_i \right) \models_{pe} \bigwedge_{i=p}^n \bar{e}_i \quad (1)$$

The inductive hypothesis implies that the p^{th} element, $a_{2p} = a_{k'}$, of the sequence $\langle a_2, a_4, \dots, a_n \rangle$ is zero. So we get $\bar{a}_{k'}$ as claimed.

It is important to note that Theorem 4 does not claim that all of the a_j with $j \geq k'$ become negated by partial evaluation. This does not hold. However, when using enhanced pairwise mergers we have $\text{sorted}(\langle a_1, \dots, a_n \rangle)$ from the corresponding merger where $\langle a_1, \dots, a_n \rangle$ are outputs. This gives by unit propagation that $a_j = 0$ for $k' \leq j \leq n$.

The next theorem is inspired by the work presented in [1] where the authors observe, in the context of the $n \times n \rightarrow 2n$ odd-even sorting network, that if we are only interested in the $n+1$ largest elements of the output, the merger can be simplified to a network with two inputs, each of length n and an output of length $n+1$. We show a similar result for the pairwise merger but emphasize that if we are only interested in the $n+1$ largest outputs because $c_{n+1}=0, \dots, c_{2n}=0$ (as in the context of a cardinality constraint with $k \leq n+1$), then we obtain by partial evaluation a simplified merge network with $n+1$ outputs in which the last output is surely a zero. We keep it in the presentation to simplify the proofs.

Theorem 5. *Consider the following specification for a simplified pairwise merger.*

$$\begin{aligned} & \text{SMerge}(\langle a_1 \rangle, \langle b_1 \rangle, \langle a_1, b_1 \rangle). \\ & \text{SMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle d_1, c_2, \dots, c_{n+1} \rangle) \leftrightarrow \\ & \quad \text{SMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_{n/2+1} \rangle) \wedge \\ & \quad \text{SMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_{n/2+1} \rangle) \wedge \\ & \quad \bar{e}_{n/2+1} \wedge \bigwedge_{i=1}^{n/2} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}). \end{aligned}$$

Then,

$$\text{PWMerge} \left(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle \right) \wedge \bigwedge_{i=n+1}^{2n} \bar{c}_i \models_{pe} \text{SMerge} \left(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{n+1} \rangle \right)$$

Proof. (by induction on n) For $n = 1$ there is nothing to prove. Let us observe the case, $n = 2$. We have from the definitions

$$\begin{aligned} \text{PWMerge}(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \langle a_1, c_2, c_3, b_2 \rangle) &= \text{comparator}(a_2, b_1, c_2, c_3) \\ \text{SMerge}(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \langle a_1, c_2, c_3 \rangle) &= \bar{b}_2 \wedge \text{comparator}(a_2, b_1, c_2, c_3) \end{aligned}$$

And the result holds:

$$\text{PWMerge}(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \langle a_1, c_2, c_3, b_2 \rangle) \wedge \bar{b}_2 \models_{pe} \text{SMerge} \left(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \langle a_1, c_2, c_3 \rangle \right)$$

For the general case $n > 2$. Assume that $\bigwedge_{i=n+1}^{2n} \bar{c}_i$. Partial evaluation of the $\text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1})$ from the definition of PWMerge (for $n/2 + 1 \leq i \leq n - 1$) gives $\bigwedge_{i=n/2+2}^n \bar{d}_i$ and $\bigwedge_{j=n/2+1}^n \bar{e}_j$, and from the same definition we obtain that $c_{2n} = e_n$. The remaining comparators from this part of the definition are

$$S_1 = \bigwedge_{i=1}^{i=n/2} \{ \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \}$$

Now, applying the inductive hypothesis on the odd and the even cases we get:

$$S_2 = \left(\text{SMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_{n/2+1} \rangle) \wedge \text{SMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_{n/2+1} \rangle) \wedge \bar{e}_{n/2+1} \right)$$

The result follows because $\text{SMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{n+1} \rangle) \leftrightarrow S_1 \wedge S_2$.

The next theorem complements the previous one. Consider

$$\text{PWMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle)$$

When negating the outputs $\langle c_k, \dots, c_{2n} \rangle$ and assuming that $\langle a_1, \dots, a_n \rangle$ are constrained to be sorted, the pairwise merger reduces by partial evaluation to a simplified merger, the size and depth of which depend exclusively on k . This is the key property that enables the construction of a cardinality network of size $O(n \log^2 k)$. Looking over a pairwise sorting network from outputs to inputs, we have a series of mergers followed by a series of splitters. If the first merger (from the outputs) has zeros from its k^{th} position then at the next level there are zeros from the k^{th} and from the $(k/2)^{th}$ positions. After $\log k$ levels, some of the mergers become trivial (zeros on all inputs and outputs).

Theorem 6. *Let $k \leq n$ and let k' be the smallest power of 2 that is greater or equal to k . Then*

$$\begin{aligned} \text{PWMerge} \left(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{2n} \rangle \right) \wedge \text{sorted}(\langle a_1, \dots, a_n \rangle) \wedge \bigwedge_{i=k}^{2n} \bar{c}_i \models_{pe} \\ \text{SMerge}(\langle a_1, \dots, a_{k'} \rangle, \langle b_1, \dots, b_{k'} \rangle, \langle c_1, \dots, c_{k'+1} \rangle) \end{aligned}$$

n	full	Method	$k=4$	$k=8$	$k=16$	$k=32$	$k=n/2$
128	1471	pw	258	416	644	955	1248
		oe	315	572	879	1148	1335
256	3839	pw	515	812	1226	1841	3288
		oe	635	1164	1851	2532	3510
1024	24063	pw	2053	3143	4475	6425	20933
		oe	2555	4716	7683	11268	22260
2048	58367	pw	4102	6230	8680	12056	51130
		oe	5115	9452	15459	23048	54259

Table 1. # of comparators for cardinality networks obtained via partial evaluation

Proof. (See Appendix.)

We are now in position to state the main theorem of the paper.

Theorem 7. *The pairwise cardinality network encoding a cardinality constraint $x_1 + x_2 + \dots + x_n \prec k$ collapses by partial evaluation to a network with $O(n \log^2 k)$ comparators.*

Proof. (sketch) Construct an $n \times n$ pairwise sorting network. For simplicity, assume that k and n are powers of 2 and that $k \leq n/2$.

1. View the network like this: in the middle we have n/k sorting networks of size $k \times k$. These give a total size of $O(n/k * k \log^2 k) = O(n \log^2 k)$.
2. On the “right” of these $k \times k$ networks we have $1 + 2 + \dots + \frac{n}{2k} = \frac{n}{k} - 1$ pairwise mergers, each of size $O(k \log k)$ after partial evaluation. This gives another $O(n \log k)$.
3. Now view the full sorting network. Let $c(n, k)$ denote the number of comparators in the split components of the network after partial evaluation originating from setting the k^{th} output to zero. That $c(n, k)$ is in $O(n \log^2 k)$ comes from the recurrence

$$c(n, k) = \begin{cases} 0 & \text{if } k = 1 \\ k \log^2 k & \text{if } n = k \text{ and } k > 1 \\ c(n/2, k) + c(n/2, k/2) + n/2 & \text{otherwise.} \end{cases}$$

That the pairwise cardinality network for n variables and bound k reduces by partial evaluation to a network with $O(n \log^2 k)$ comparators is theoretically pleasing. However, note that it is also possible to build the same reduced network directly using simplified pairwise mergers and propagating the bound k which is halved in each step for the lower recursively defined pairwise sorter.

Finally, we note that Theorem 7 holds also when encoding a cardinality network using an odd-even sorting network. The proof is based on the observation that an odd-even merger $\text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{2n} \rangle)$ where $c_k = 0$ simplifies to a network of size $O(k \log k)$.

5 A Preliminary Evaluation

This section describes a preliminary comparison of the use of odd-even and pairwise sorting networks for the applications involving cardinality constraints.

Table 1 shows some statistics regarding the size of the networks obtained from sorting networks after application of partial evaluation. Here size is measured counting number of comparators. Note that each comparator can be encoded using 6 CNF clauses, or alternatively, based on the technique proposed in [1] as a “half comparator” and encoded using only 3 clauses.

The first column in Table 1 indicates the size of the network, the second column indicates the number of comparators before application of unit propagation. The third column indicates the type of network considered, pairwise (**pw**) or odd-even (**oe**). The next columns indicate the size of the network after unit propagation for various values of k . The last column considers the worst case with $k = n/2$. The table indicates that cardinality networks expressed using pairwise sorting networks are more amenable to partial evaluation.

Table 2 describes results when solving a Boolean cardinality matrix problem encoded using three techniques. An instance, (n, k_1, k_2) , is to find values for the elements of an $n \times n$ matrix of Boolean variables where the cardinality of each row and column is between values k_1 and k_2 . The table summarizes results for $n = 100$, for various values of k_1 with $k_2 = k_1 + 2$ and $k_2 = k_1 + 3$.

The first column indicates the value of k_2 in terms of k_1 : $k_2 = k_1 + 2$ or $k_2 = k_1 + 3$. The second column indicates what is being measured: CNF size after partial evaluation (in 1000’s of clauses) or CPU time for solving the problem (in seconds). We are running MiniSAT version 2 through its Prolog interface as described in [3]. The machine is a laptop running Linux with 2 Genuine Intel(R) CPUs, each 2GHz with 1GB RAM. The third column indicates the encoding method: using a pairwise cardinality network (based on a pairwise sorting network) (**pw**), or using an odd-even sorting network (**oe**), or using the cardinality network described in [1] (based on a construction built from a cascade of odd-even sorting networks). The next columns provide the data for various values of k_1 . The results indicate a clear advantage for the use of pairwise sorting networks.

6 Summary and Conclusion

Sorting networks are often applied when encoding cardinality constraints. We argue the advantage in basing such encodings on the pairwise sorting network instead of on the odd-even sorting network as typically chosen, for example in [4] and in [1].

Our presentation clarifies the precise relationship between the pairwise network introduced in 1992 and the odd-even network from 1968. The simplicity of this connection is surprising and perhaps demystifies the intuition underlying the pairwise network, which from 1992 is not referred to at all in the literature.

In contrast to previous works, such as [8] and [1], which encode cardinality constraints by application of specially constructed networks of comparators, our

	Measure	Method	$k_1=5$	$k_1=10$	$k_1=15$	$k_1=20$	$k_1=25$
$k_2 = k_1 + 2$	cnf size ($\times 1000$)	pw	350	542	763	761	782
		oe	467	683	883	896	920
		card	473	665	760	878	914
	cpu time (sec.)	pw	6	19	27	89	270
		oe	23	1152	2395	946	1250
		card	41	382	2916	1110	832
$k_2 = k_1 + 3$	cnf size ($\times 1000$)	pw	350	554	763	766	782
		oe	467	697	883	901	920
		card	509	684	840	904	924
	cpu time (sec.)	pw	5	8	46	100	280
		oe	37	257	2583	1122	1546
		card	27	621	1798	1317	1158

Table 2. Results for Boolean cardinality matrix problems (n, k_1, k_2) with $n = 100$.

contribution is based directly on the (automatic) simplification of a sorting network. It is straightforward to apply this kind of simplification directly on the procedure that generates the network for $x_1 + x_2 + \dots + x_n < k$, instead of first generating the $O(n \log^2 n)$ sorting network and then simplifying it to a network of size $O(n \log^2 k)$.

Acknowledgement We thank the anonymous reviewers for useful comments on the earlier version of this paper.

References

1. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009.
2. Kenneth E. Batchier. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.
3. Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.
4. Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.
5. Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
6. Ian Parberry. *Parallel Complexity Theory*. Research Notes in Theoretical Computer Science. Pitman Publishing London, 1987.
7. Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.
8. Benjamin W. Wah and Kuo-Liang Chen. A partitioning approach to the design of selection networks. *IEEE Trans. Computers*, 33(3):261–268, 1984.

A Appendix: Proof Sketches

Proof. (of Theorem 2) By induction on n . The base case, $n = 1$, follows directly from the definitions. For $n > 1$, assume that the theorem holds for all $n' < n$. By definition,

$$\begin{aligned} & \text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle d_1, c_2, \dots, c_{2n-1}, e_n \rangle) \leftrightarrow \\ & \quad \text{OEMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \wedge \\ & \quad \text{OEMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_n \rangle) \wedge \\ & \quad \bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \end{aligned}$$

and the induction hypothesis holds for the recursive odd and even cases giving:

$$\begin{aligned} & \text{OEMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \leftrightarrow \\ & \quad \text{PWSplit} \left(\begin{array}{l} \langle a_1, b_1, a_3, b_3, \dots, a_{n-1}, b_{n-1} \rangle \\ \langle a'_1, a'_3, \dots, a'_{n-1} \rangle, \langle b'_1, b'_3, \dots, b'_{n-1} \rangle \end{array} \right) \wedge \\ & \quad \text{PWMerge}(\langle a'_1, a'_3, \dots, a'_{n-1} \rangle, \langle b'_1, b'_3, \dots, b'_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \\ & \text{OEMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_n \rangle) \leftrightarrow \\ & \quad \text{PWSplit} \left(\begin{array}{l} \langle a_2, b_2, a_4, b_4, \dots, a_n, b_n \rangle, \\ \langle a'_2, a'_4, \dots, a'_n \rangle, \langle b'_2, b'_4, \dots, b'_n \rangle \end{array} \right) \wedge \\ & \quad \text{PWMerge}(\langle a'_2, a'_4, \dots, a'_n \rangle, \langle b'_2, b'_4, \dots, b'_n \rangle, \langle e_1, \dots, e_n \rangle) \end{aligned}$$

substituting this in the definition (and rearranging the conjuncts) gives:

$$\begin{aligned} & \text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{2n} \rangle) \leftrightarrow \\ & \quad \text{PWSplit} \left(\begin{array}{l} \langle a_1, b_1, a_3, b_3, \dots, a_{n-1}, b_{n-1} \rangle \\ \langle a'_1, a'_3, \dots, a'_{n-1} \rangle, \langle b'_1, b'_3, \dots, b'_{n-1} \rangle \end{array} \right) \wedge \\ & \quad \text{PWSplit} \left(\begin{array}{l} \langle a_2, b_2, a_4, b_4, \dots, a_n, b_n \rangle, \\ \langle a'_2, a'_4, \dots, a'_n \rangle, \langle b'_2, b'_4, \dots, b'_n \rangle \end{array} \right) \wedge \\ & \quad \text{PWMerge}(\langle a'_1, a'_3, \dots, a'_{n-1} \rangle, \langle b'_1, b'_3, \dots, b'_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \wedge \\ & \quad \text{PWMerge}(\langle a'_2, a'_4, \dots, a'_n \rangle, \langle b'_2, b'_4, \dots, b'_n \rangle, \langle e_1, \dots, e_n \rangle) \wedge \\ & \quad \bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \end{aligned}$$

which by definitions of PWSplit and PWMerge gives the required result.

$$\begin{aligned} & \text{OEMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle c_1, \dots, c_{2n} \rangle) \leftrightarrow \\ & \quad \text{PWSplit}(\langle a_1, b_1, a_2, b_2, \dots, a_n, b_n \rangle, \langle a'_1, a'_2, \dots, a'_n \rangle, \langle b'_1, b'_2, \dots, b'_n \rangle) \wedge \\ & \quad \text{PWMerge}(\langle a'_1, a'_2, \dots, a'_n \rangle, \langle b'_1, b'_2, \dots, b'_n \rangle, \langle c_1, \dots, c_{2n} \rangle) \end{aligned}$$

Proof. (of Theorem 3). By induction on n . The base case, $n = 1$, holds vacuously. For $n > 1$, assume that the statement holds for all $n' < n$ and consider the following cases according to the parities of k and $\lceil k/2 \rceil$. Recall that,

$$\begin{aligned} & \text{PWMerge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \langle d_1, c_2, \dots, c_{2n-1}, e_n \rangle) \leftrightarrow \\ & \quad \text{PWMerge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle, \langle d_1, \dots, d_n \rangle) \wedge \\ & \quad \text{PWMerge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle, \langle e_1, \dots, e_n \rangle) \wedge \\ & \quad \bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}). \end{aligned}$$

Let $\{b'_1, b'_2, \dots, b'_{n/2}\}$, and $\{b''_1, b''_2, \dots, b''_{n/2}\}$ be the odd and the even subsequences of $\{b_1, b_2, \dots, b_n\}$ respectively. We consider two cases depending on the parity of k .

k is even: Denote $k = 2p$. We have

$$\left(\bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \right) \wedge \left(\bigwedge_{i=k}^{2n} \bar{c}_i \right) \models_{pe} \left(\bigwedge_{i=p+1}^n \bar{d}_i \right) \wedge \left(\bigwedge_{j=p}^n \bar{e}_j \right) \quad (2)$$

Now consider two subcases depending on the parity of p : (1) Assume $p = 2q$. From Equation (2) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \bar{b}'_i$ and $\bigwedge_{j=q}^{n/2} \bar{b}''_j$ which together imply that $\bigwedge_{i=k/2}^n \bar{b}_i$. (2) Assume $p = 2q + 1$. From Equation (2) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \bar{b}'_i$ and $\bigwedge_{j=q+1}^{n/2} \bar{b}''_j$ which together imply that $\bigwedge_{i=p}^n \bar{b}_i$ or in terms of k that $\bigwedge_{i=k/2}^n \bar{b}_i$.
 k is odd: Denote $k = 2p + 1$. We have

$$\left(\bigwedge_{i=1}^{n-1} \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \right) \wedge \left(\bigwedge_{i=k}^{2n} \bar{c}_i \right) \models_{pe} \left(\bigwedge_{i=p+2}^n \bar{d}_i \right) \wedge \left(\bigwedge_{j=p+1}^n \bar{e}_j \right). \quad (3)$$

Two subcases are considered for the parity of p : (1) Assume $p = 2q$. From Equation (3) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \bar{b}'_i$ and $\bigwedge_{j=q+1}^{n/2} \bar{b}''_j$. Therefore, $\bigwedge_{i=p+1}^n \bar{b}_i$ or $\bigwedge_{i=\lceil k/2 \rceil}^n \bar{b}_i$. (2) Assume $p = 2q + 1$. From Equation (3) and the inductive hypothesis we get $\bigwedge_{i=q+2}^{n/2} \bar{b}'_i$ and $\bigwedge_{j=q+1}^{n/2} \bar{b}''_j$. Therefore, $\bigwedge_{i=p+1}^n \bar{b}_i$ or $\bigwedge_{i=\lceil k/2 \rceil}^n \bar{b}_i$. In all four subcases we have $\bigwedge_{i=\lceil k/2 \rceil}^n \bar{b}_i$ which proves the theorem.

To prove Theorem 6 we use the following lemmata

Lemma 1. For $n > 1$,

$$\text{PWMerge} \left(\begin{pmatrix} \langle a_1, \dots, a_n \rangle, \\ \langle b_1, \dots, b_n \rangle, \\ \langle c_1, \dots, c_{2n} \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n/2+1}^n (\bar{a}_i \wedge \bar{b}_i) \models_{pe} \text{PWMerge} \left(\begin{pmatrix} \langle a_1, \dots, a_{n/2} \rangle, \\ \langle b_1, \dots, b_{n/2} \rangle, \\ \langle c_1, \dots, c_n \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n+1}^{2n} \bar{c}_i$$

Proof. (by induction on n). For the base case $n = 2$ we have (by definition)

$$\text{PWMerge}(\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle, \langle a_1, c_2, c_3, b_2 \rangle) \leftrightarrow \text{comparator}(a_2, b_1, c_2, c_3)$$

$$\text{PWMerge}(\langle a_1 \rangle, \langle b_1 \rangle, \langle a_1, b_1 \rangle) \leftrightarrow \text{true}$$

We need to show that if a_2 and b_2 are negated then c_3 and b_2 become negated, and that $\langle a_1, b_1 \rangle = \langle a_1, c_2 \rangle$. Both facts follow because when $a_2 = 0$ the comparator gives by partial evaluation that $c_3 = 0$ and $b_1 = c_2$.

For the general case, we apply the inductive hypothesis (*) to the odd and the even cases in the definition of the pairwise merger. Giving respectively:

$$\begin{aligned} & \text{PWMerge} \left(\begin{pmatrix} \langle a_1, a_3, \dots, a_{n-1} \rangle, \\ \langle b_1, b_3, \dots, b_{n-1} \rangle, \\ \langle d_1, \dots, d_n \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n/4+1}^{n/2} (\bar{a}_{2i-1} \wedge \bar{b}_{2i-1}) \models_{pe} \\ & \quad \text{PWMerge} \left(\begin{pmatrix} \langle a_1, a_3, \dots, a_{n/2-1} \rangle, \\ \langle b_1, b_3, \dots, b_{n/2-1} \rangle, \\ \langle d_1, \dots, d_{n/2} \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n/2+1}^n \bar{d}_i \\ & \text{PWMerge} \left(\begin{pmatrix} \langle a_2, a_4, \dots, a_n \rangle, \\ \langle b_2, b_4, \dots, b_n \rangle, \\ \langle e_1, \dots, e_n \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n/4+1}^{n/2} (\bar{a}_{2i} \wedge \bar{b}_{2i}) \models_{pe} \\ & \quad \text{PWMerge} \left(\begin{pmatrix} \langle a_2, a_4, \dots, a_{n/2} \rangle, \\ \langle b_2, b_4, \dots, b_{n/2} \rangle, \\ \langle e_1, \dots, e_{n/2} \rangle \end{pmatrix} \right) \wedge \bigwedge_{i=n/2+1}^n \bar{e}_i \end{aligned}$$

Several of the comparators from this application (*) are reduced by partial evaluation as follows:

$$\bar{e}_n \wedge \bigwedge_{i=n/2+1}^{n-1} \bar{e}_i \wedge \bar{d}_{i+1} \wedge \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \models_{pe} \bigwedge_{i=n+2}^{2n} \bar{c}_i$$

$$e_{n/2} \wedge \bar{d}_{n/2+1} \wedge \text{comparator}(e_{n/2}, d_{n/2+1}, c_n, c_{n+1}) \models_{pe} \bar{c}_{n+1}$$

Together this gives $\bigwedge_{i=n+1}^{2n} \bar{c}_i$, and now the required result follows directly from the definition of the pairwise merger.

Lemma 2. *Let $k \leq n$ and let k' be the smallest power of 2 which is greater or equal to k . Then,*

$$\text{PWMerge} \left(\begin{array}{l} \langle a_1, \dots, a_n \rangle, \\ \langle b_1, \dots, b_n \rangle, \\ \langle c_1, \dots, c_{2n} \rangle \end{array} \right) \wedge \bigwedge_{i=k'+1}^n \bar{a}_i \wedge \bar{b}_i \models_{pe} \text{PWMerge} \left(\begin{array}{l} \langle a_1, \dots, a_{k'} \rangle, \\ \langle b_1, \dots, b_{k'} \rangle, \\ \langle c_1, \dots, c_{2k'} \rangle \end{array} \right) \wedge \bigwedge_{i=2k'+1}^{2n} \bar{c}_i$$

Proof. The proof is by induction on n . For $n = k'$ it is trivial. For the general case, $n > k'$, we apply the induction hypothesis (*) to the odd and even cases in the definition of the pairwise merger. Note that each such application gives a pairwise merger of size k' :

$$\begin{aligned} & \text{PWMerge} \left(\begin{array}{l} \langle a_1, a_3, \dots, a_{n-1} \rangle, \\ \langle b_1, b_3, \dots, b_{n-1} \rangle, \\ \langle d_1, \dots, d_n \rangle \end{array} \right) \wedge \bigwedge_{i=k'/2+1}^{n/2} (\bar{a}_{2i-1} \wedge \bar{b}_{2i-1}) \models_{pe} \\ & \quad \text{PWMerge} \left(\begin{array}{l} \langle a_1, a_3, \dots, a_{2k'-1} \rangle, \\ \langle b_1, b_3, \dots, b_{2k'-1} \rangle, \\ \langle d_1, \dots, d_{2k'} \rangle \end{array} \right) \wedge \bigwedge_{i=2k'+1}^n \bar{d}_i \\ & \text{PWMerge} \left(\begin{array}{l} \langle a_2, a_4, \dots, a_n \rangle, \\ \langle b_2, b_4, \dots, b_n \rangle, \\ \langle e_1, \dots, e_n \rangle \end{array} \right) \wedge \bigwedge_{i=k'/2+1}^{n/2} \bar{a}_{2i} \wedge \bar{b}_{2i} \models_{pe} \\ & \quad \text{PWMerge} \left(\begin{array}{l} \langle a_2, a_4, \dots, a_{2k'} \rangle, \\ \langle b_2, b_4, \dots, b_{2k'} \rangle, \\ \langle e_1, \dots, e_{2k'} \rangle \end{array} \right) \wedge \bigwedge_{i=2k'+1}^n \bar{e}_i \end{aligned}$$

The application (*) of the pairwise merger definition introduces comparators which are reduced by partial evaluation:

$$\begin{aligned} \bar{e}_n \wedge \bigwedge_{i=2k'+1}^{n-1} \bar{e}_i \wedge \bar{d}_{i+1} \wedge \text{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) & \models_{pe} \bigwedge_{i=4k'+2}^{2n} \bar{c}_i \\ e_{2k'} \wedge \bar{d}_{2k'+1} \wedge \text{comparator}(e_{2k'}, d_{2k'+1}, c_{4k'}, c_{4k'+1}) & \models_{pe} \bar{c}_{4k'+1} \end{aligned}$$

Together this gives $\bigwedge_{i=4k'+1}^{2n} \bar{c}_i$ and from the definition of the pairwise merger we get $\text{PWMerge}(\langle a_1, \dots, a_{2k'} \rangle, \langle b_1, \dots, b_{2k'} \rangle, \langle c_1, \dots, c_{4k'} \rangle) \wedge \bigwedge_{i=4k'+1}^{2n} \bar{c}_i$ which together with the lemma statement and application of the previous lemma gives:

$$\begin{aligned} & \text{PWMerge}(\langle a_1, \dots, a_{2k'} \rangle, \langle b_1, \dots, b_{2k'} \rangle, \langle c_1, \dots, c_{4k'} \rangle) \wedge \bigwedge_{i=k'+1}^{2k'} \bar{a}_i \wedge \bar{b}_i \models_{pe} \\ & \text{PWMerge}(\langle a_1, \dots, a_{k'} \rangle, \langle b_1, \dots, b_{k'} \rangle, \langle c_1, \dots, c_{2k'} \rangle) \wedge \bigwedge_{i=2k'+1}^{4k} \bar{c}_i. \end{aligned}$$

These two give us the required $\text{PWMerge}(\langle a_1, \dots, a_{k'} \rangle, \langle b_1, \dots, b_{k'} \rangle, \langle c_1, \dots, c_{2k'} \rangle) \wedge \bigwedge_{i=2k'+1}^{2n} \bar{c}_i$.

Proof. (of Theorem 6) From Theorem 3 we have:

$$\text{PWMerge} \left(\begin{array}{l} \langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \\ \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle \end{array} \right) \wedge \left(\bigwedge_{i=k'}^{2n} \bar{c}_i \right) \models_{pe} \left(\bigwedge_{j=k'/2}^n \bar{b}_j \right)$$

In particular we have $\bigwedge_{i=k'+1}^n \bar{b}_i$. In addition, from Theorem 4 we have:

$$\text{PWMerge} \left(\begin{array}{l} \langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle, \\ \langle c_1, c_2, \dots, c_{2n-1}, c_{2n} \rangle \end{array} \right) \wedge \left(\bigwedge_{i=k'}^{2n} \bar{c}_i \right) \models_{pe} \bar{a}_{k'}$$

and consequently that $\bar{a}_{k'} \wedge \text{sorted}(\langle a_1, \dots, a_n \rangle) \models_{pe} \bigwedge_{i=k'+1}^n \bar{a}_i$. From lemma 2, and the above we obtain $\text{PWMerge}(\langle a_1, \dots, a_{k'} \rangle, \langle b_1, \dots, b_{k'} \rangle, \langle c_1, \dots, c_{2k'} \rangle)$. We are given that $\bigwedge_{i=k'+1}^{2k'} \bar{c}_i$. Hence, according to Theorem 5 we get $\text{SMerge}(\langle a_1, \dots, a_{k'} \rangle, \langle b_1, \dots, b_{k'} \rangle, \langle c_1, \dots, c_{k'+1} \rangle)$.

B Appendix: Specifying Sorting Networks in Prolog

This appendix illustrates working Prolog specifications for the sorting network constructions presented in the paper. The code is simplified assuming that input/output sizes are powers of 2. Networks are represented as lists of comparators. Comparators are atoms of the form `comparator(A,B,C,D)` where A,B are the inputs and C,D are the outputs.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a Batcher odd-even sorting network %
% (see 4th page of article) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

oe_sort(As,Cs,Comparators) :-
    oe_sort(As,Cs,Comparators-[]).

oe_sort([A],[A],Cmp-Cmp) :- !.
oe_sort(As,Cs,Cmp1-Cmp4) :-
    split(As,As1,As2),
    oe_sort(As1,Ds1,Cmp1-Cmp2),
    oe_sort(As2,Ds2,Cmp2-Cmp3),
    oe_merge(Ds1,Ds2,Cs,Cmp3-Cmp4).

% merge two sorted sequences to a sorted sequence
oe_merge([A],[B],[C1,C2],
         [comparator(A,B,C1,C2)|Cmp]-Cmp) :- !.
oe_merge(As,Bs,[D|Cs],Cmp1-Cmp4) :-
    oddEven(As,AsOdd,AsEven),
    oddEven(Bs,BsOdd,BsEven),
    pw_merge(AsOdd,BsOdd,[D|Ds],Cmp1-Cmp2),
    pw_merge(AsEven,BsEven,Es,Cmp2-Cmp3),
    combine(Ds,Es,Cs,Cmp3-Cmp4).

% split down the middle
split(Xs,As,Bs) :-
    length(Xs,N), N1 is ceil(N/2),
    length(As,N1), append(As,Bs,Xs).

% split to odd and even
oddEven([Odd,Even|As],[Odd|Odds],[Even|Evens]) :-
    oddEven(As,Odds,Evens).
oddEven([],[],[]).

% combines the even and odd sorted elements
combine([],[],[],Cmp-Cmp).
combine([A|As],[B|Bs],[C1,C2|Cs],
        [comparator(A,B,C1,C2)|Cmp1]-Cmp2) :-
    combine(As,Bs,Cs,Cmp1-Cmp2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct an (alternative) odd-even merger. %
% It is specified as the combination of a %
% pairwise split and a pairwise merge %
% (see Theorem 2) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alternative_oe_merge(As,Bs,Cs,Cmp1-Cmp3) :-
    interleave(As,Bs,ABs),
    pw_split(ABs,ABs1,ABs2,Cmp1-Cmp2),
    pw_merge(ABs1,ABs2,Cs,Cmp2-Cmp3).

interleave([],[],[]).
interleave([A|As],[B|Bs],[A,B|ABs]) :-
    interleave(As,Bs,ABs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise sorting network %
% (see 5th page of article) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_sort(As,Cs,Comparators) :-
    pw_sort(As,Cs,Comparators-[]).

pw_sort([A],[A],Cmp-Cmp) :- !.
pw_sort(As,Cs,Cmp1-Cmp5) :-
    pw_split(As,As1,As2,Cmp1-Cmp2),
    pw_sort(As1,Ds1,Cmp2-Cmp3),
    pw_sort(As2,Ds2,Cmp3-Cmp4),
    pw_merge(Ds1,Ds2,Cs,Cmp4-Cmp5).

% split pairs from a sequence to their larger and smaller elements
pw_split([],[],[],Cmp-Cmp).
pw_split([A1,A2|As],[B|Bs],[C|Cs],
         [comparator(A1,A2,B,C)|Cmp1]-Cmp2) :-
    pw_split(As,Bs,Cs,Cmp1-Cmp2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise merger. It merges %
% two sorted sequences of sorted pairs %
% (see Theorem 1, page 5) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_merge([A],[B],[A,B],Cmp-Cmp) :- !.
pw_merge(As,Bs,[D|Cs],Cmp1-Cmp4) :-
    oddEven(As,AsOdd,AsEven),
    oddEven(Bs,BsOdd,BsEven),
    pw_merge(AsOdd,BsOdd,[D|Ds],Cmp1-Cmp2),
    pw_merge(AsEven,BsEven,Es,Cmp2-Cmp3),
    combine(Ds,Es,Cs,Cmp3-Cmp4).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise merger following the %
% description from page 4 of [Parberry92]. %
% This is the network referred to in the %
% proof of Theorem 2. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

parberry_pw_merge(As,Bs,Cs,Ds,Cmp1-Cmp2) :-
    length(As,K),
    parberry_pw_merge_by_level(K,As,Bs,Cs,Ds,Cmp1-Cmp2).

parberry_pw_merge_by_level(1,As,Bs,As,Bs,Cmp-Cmp).
parberry_pw_merge_by_level(M,As,Bs,Cs,Ds,Cmp1-Cmp3) :-
    M>1, !, M1 is M//2,
    length(NewAs1,M1), append(NewAs1,As2,As),
    length(NewBs2,M1), append(Bs1,NewBs2,Bs),
    compare(Bs1,As2,NewBs1,NewAs2,Cmp1-Cmp2),
    append(NewAs1,NewAs2,NewAs), append(NewBs1,NewBs2,NewBs),
    parberry_pw_merge_by_level(M1,NewAs,NewBs,Cs,Ds,Cmp2-Cmp3).

compare([],[],[],Cmp-Cmp).
compare([A|As],[B|Bs],[C|Cs],[D|Ds],
        [comparator(A,B,C,D)|Cmp1]-Cmp2) :-
    compare(As,Bs,Cs,Ds,Cmp1-Cmp2).

```