International Journal of Foundations of Computer Science

BLOCK SORTING IS HARD

WOLFGANG W. BEIN*
Department of Computer Science,
University of Nevada, Las Vegas, NV 89154, USA.

LAWRENCE L. LARMORE†
Department of Computer Science
University of Nevada, Las Vegas, NV 89154, USA.

SHAHRAM LATIFI
Department of Electrical and Computer Engineering
University of Nevada, Las Vegas, NV 89154, USA.

and

I. HAL SUDBOROUGH
Department of Computer Science
University of Texas at Dallas, Richardson, TX 75083, USA.

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Block sorting is used in connection with Optical Character Recognition (OCR). Recent work has focused on finding good strategies which work in practice.
In this paper, we show that optimizing block sorting is $\mathcal{NP}$-hard. Along with this result, we give new non-trivial lower bounds. These bounds can be computed efficiently. We define the concept of "Local Property Algorithms" and show that several previously published block sorting algorithms fall into this class.

Keywords: Optimal Sorting, Block Sorting, Optical Character Recognition, $\mathcal{NP}$-Completeness

1. The Block Sorting Problem

The *block sorting* problem is the problem of minimizing the number of steps to sort a list of distinct items, where a sublist of items which are already in sorted order (called a *block*) can be moved together in one step. Blocks, once formed, are never broken up, but are only combined to form larger blocks; in the final step, there is just one block. For example, the list (d, b, e, a, c) can be block sorted in two

*Research supported by NSF grant CCR-9821009.
†Research supported by NSF grant CCR-9821009.

A F C B E D		How ? they did it do		
A C B E F D	`move F`	How they did	it ?	do
A C B E D	`and combine`	How they did	–E–	do
A B C E D	`move C`	How did they	–E–	do
A E D	`and combine`	———A———	–E–	do
A D E	`move D`	———A———	do	–E–
A	`and combine`	————————A————————		

Table 1: How to block sort "How ? they did it do"

steps: move b to obtain (d, e, a, b, c), then move the block d, e to obtain (a, b, c, d, e). We say that a list is *irreducible* if every block has just one item. Without loss of generality, the initial list is irreducible, since any block could be considered to be just one item.

The block sorting problem is equivalent to the following list deletion problem, which we call the *block deletion problem*. Given a list of distinct items, any item can be deleted in one step, and all but one item in any block can be deleted in zero steps. The problem is to minimize the number of steps needed to obtain a list of length one. Thus, for example, starting with the list (d, b, e, a, c), delete b to obtain (d, e, a, c). Since a, c and d, e are now blocks, c and e are deleted for free, obtaining (d, a). We write *step*(b) for the move which deletes the symbol b. In the second step, delete d to obtain the list (a). Thus two steps, *step*(b) and *step*(d) are required.

The block deletion problem is equivalent to the block sorting problem. Let each block be given a "name" equal to one of its elements. Then, the block sorting problem on an irreducible list is equivalent to the block deletion problem on the list of block names, where, whenever blocks are combined to form a new block, the new block is assigned the name of one of the old blocks which are combined. Further, we insist that the name of a block that was not moved be used as the new name. Without loss of generality, when a block is moved, it is always to a place where it can immediately be combined with another block, thus its name is simply deleted from the name list. Combining blocks where neither is moved is simply a free deletion from the name list. Henceforth, we shall refer to the block sorting problem and the block deletion problem interchangeably, treating them as different ways of looking at the same problem. Table 1 illustrates this concept, with an example inspired by an application in Optical Character Recognition (OCR). Here we assume that a list "How ? they did it do" has been recognized but the names are not in the desired order, "How did they do it ?" In the example, the deletion of F (which stands for the question mark) actually means moving block F to the position adjacent to block E, then combining E and F into one symbol, and renaming this new block E.

As in the example, we shall frequently omit commas and parentheses. The list (d, e, a, b, c) will be written simply as *deabc*. Thus, a block sorting problem instance can be given as a string w over an ordered alphabet Σ with ordering relation "$<$", where w has no duplicate symbols.

Suppose that Σ is an alphabet with two orderings, say "$<_1$" and "$<_2$." The

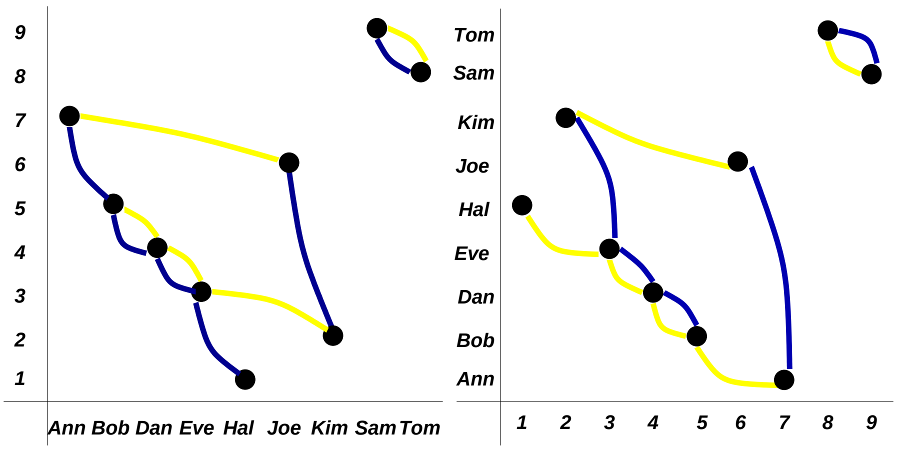

Figure 1: A Primal and Dual Block Sorting Problem Pair

primal biordering problem is, given w to be the string consisting of all the symbols of Σ in "$<_2$" order, block sort w under the "$<_1$" order. The *dual problem* is, given $\tilde{w}$ to be the string consisting of all the symbols of Σ in "$<_1$" order, block sort $\tilde{w}$ under the "$<_2$" order. Figure 1 gives an example, with children's names on one axis and ages on the other. Here "$<_1$" is the lexicographical order (for first names) and "$<_2$" is the usual order for the integers (for the ages). The primal sorting problem is, given the children in alphabetic order, block sort them into age order, while the dual problem is, given them in age order, block sort them into alphabetical order.

Any instance of the block sorting problem is a primal biordering problem. Consider an ordered alphabet Σ with ordering relation "$<$" and a block sorting instance w, consisting of a string of the distinct symbols of Σ. The problem has a natural dual, as follows. Let $\tilde{w}$ be the sorted string. Define $<_1=<$, and define $<_2$ to be that ordering (just on the symbols in w) so that w is already ordered under $<_2$. The dual problem is to block sort $\tilde{w}$ under $<_2$.

In our examples, below, we use the notation "$\ldots$" to denote the second ordering. Specifically, for a fixed string w over Σ, if symbol a has position i in w and symbol b has position j, for $i < j$, we write $a \ldots b$. (Thus the relation "$\ldots$" is an order relation which depends on w.) We call "$\ldots$" the *position* order. Two symbols a, b in w are called *consecutive* if a has position i in w and b has position $i+1$, and $a < b$, and there is no c in w with $a < c < b$. A string w is irreducible if and only if there are no consecutive elements a, b in w.

As mentioned earlier, block sorting is motivated by applications in Optical Character Recognition; see [6, 10, 11]. Text regions, referred to as *zones* are selected by drawing rectangles around them. Here the order of zones is significant, but in practice the output generated by a zoning procedure may be different from the correct text. Moving the pieces to the correct order corresponds to a block sorting problem. To evaluate the performance of a given zoning procedure it is of interest to find the minimum number of steps needed to obtain the correct string from the zones generated by the zoning procedure. Recently, Latifi *et al.* [6] have performed various experiments to test a number of strategies that seem to perform well in practice.

Unfortunately, as we shall see in this paper, block sorting is inherently hard. In fact, no better than a 3-competitive approximation algorithm is known. Sorting problems under various operations have been studied extensively. We mention work on sorting with prefix reversals [3, 5, 9], transpositions [1, 2] and block moves [7, 8]. In Section 2 we give a number of non-trivial lower bounds. Section 3 examines strategies that have been studied previously in connection with OCR [6], and show that these strategies can all be viewed as what we call "Local Property Algorithms" and that the run times of their implementations can be substantially improved. Finally, Section 4, shows that optimizing block sorting is $\mathcal{NP}$-hard.

2. Lower Bounds

As discussed in Section 1 let a string w and let $\tilde{w}$ be the string consisting of the same symbols, sorted under the order "$<$". We have the following lemma:

Lemma 1 *The minimum number of steps for block sorting w under order "$<$" is equal to the minimum number of steps to block sort $\tilde{w}$ under order "$\ldots$" of w.*

Proof. Consider the deletion of a symbol in w and $\tilde{w}$. Two symbols become consecutive in w as a result of such a deletion exactly when they become consecutive in $\tilde{w}$ □

We say there is a *blue edge* from a to b if $a > b$ and b is immediately after a in w, *i.e.*, ab is a substring of w. We also refer to a blue edge as a *reversal*. Alternatively, we say there is a *green edge* from b to a if $a \ldots b$ and $a > b$ and there is no c in w with $a > c > b$. We refer to a green edge as a *dual reversal.* Note that a green edge implies a blue edge in the dual string, and vice versa.

Figure 1 shows a primal dual pair, with green edges shaded lightly, and blue edges rendered in a darker shading. Blue and green edges are naturally directed; however, we illustrate them as undirected. It is somewhat difficult to see that block sorting requires the same number of steps in each instance, but it is easy to see that the block deletion problem requires the same number of steps for the primal and dual. Consider Figure 1. Items are adjacent if and only if one is exactly one space above and one space to the right of the other, and this rule holds for both diagrams. Deletions in each case consist of simply erasing a dot and collapsing the diagram. The dual diagram is the mirror image of the primal diagram, and a sequence of deletions in one can be seen as a sequence of deletions of the other by reflection. Since a mirror image of consecutive pair is a consecutive pair, the number of steps required is the same.

Lemma 2 *Let w be an irreducible string with k blue edges and ℓ green edges. Let w' be the irreducible string obtained in one step. Then the number of blue edges k' and green edges ℓ' of the resulting string w' does not increase,* i.e. *$k' \leq k$ and $\ell' \leq \ell$. Furthermore, if $|w'| = n-2$, then $k' = k-1$ or $\ell' = \ell - 1$, and if $|w'| = n-3$, then $k' = k-1$ and $\ell' = \ell-1$.*

Proof. First consider the substring xyz in string w and assume that y is deleted. (If $|w| < 3$ then the lemma is trivial.) First we consider the case that $|w'| = n - 1$. There can be a blue edge from x to y or from y to z. If both edges are present, the deletion of y will cause k to decrease by 1. In the case only one of those edges is present the deletion of y will cause k to stay the same or decrease by 1, depending on whether $x > z$. The argument for green edges is analogous, using duality.

Consider now the case that the removal of y causes the length of the string to decrease by 2, *i.e.* $|w'| = n - 2$. In this case we have that either the symbols xz become consecutive, or that a substring uv of neighboring symbols in w become consecutive. In the first case, $k' = k - 1$, since there can only be one blue edge at y, because otherwise $x > z$. In the second case $\ell' = \ell - 1$, by an analogous argument using duality.

If both effects occur we have $|w'| = n - 3$, $k' = k - 1$ and $\ell' = \ell - 1$. □

Lemma 3 *Let w be a string of length n with k blue edges and ℓ green edges. Then $m \geq n - k - \ell - 1$, where m is the minimum number of steps needed to block sort w.*

Proof. We prove the statement of the lemma by induction on the number of steps. To this end, consider now an irreducible string w of length n. Suppose there exists an optimal sorting and let the symbol x in w be the first symbol removed by that sorting. The new string w' will either have length $n' = n - 1$, $n' = n - 2$, or $n' = n - 3$.

Case $n' = n - 1$: Due to lemma 2, we have $k' \leq k$ and $\ell' \leq \ell$. It follows that $m = m' + 1 \geq n' - k' - \ell' - 1 + 1 \geq (n - 1) - k - \ell - 1 + 1 = n - k - \ell - 1$

Case $n' = n - 2$: Due to lemma 2, we have $k' = k - 1$ or $\ell' = \ell - 1$. It follows that $m = m' + 1 \geq n' - k' - \ell' - 1 + 1 \geq (n - 2) - k - \ell + 1 - 1 + 1 = n - k - \ell - 1$.

Case $n' = n - 3$: Due to lemma 2, we have $k' = k - 1$ and $\ell' = \ell - 1$. It follows that $m = m' + 1 \geq n' - k' - \ell' - 1 + 1 \geq (n - 3) - (k - 1) - (\ell - 1) - 1 + 1 = n - k - \ell - 1$.

□

Combining lemma 2 and 3, we obtain the following lower bound.

Theorem 1 *Given a string w of length n with k reversals and ℓ dual reversals, Then the minimum number of steps m required to block sort w is bounded from below by*

$$m \geq \max \begin{cases} n - k - \ell - 1 \\ k \\ \ell \end{cases}$$

By averaging the three components in the inequality of theorem 1, we obtain:

Corollary 1 *For a string w of length n the the minimum number of steps m required to block sort w is bounded from below by $m \geq \left\lceil \frac{n-1}{3} \right\rceil$.*

We say that a sorting is *perfect* if the number of steps is equal to the number of blue edges. Note that Theorem 1 implies that a perfect sorting is optimal. Though the converse does not hold, it is useful to characterize perfect sortings. To this end, fix a specific block sorting of w.

We say that there is a *red edge* from a to b if

- $a < b$ and $a \dots b$
- a and b are joined before either is moved.
- If $a \dots c \dots b$, then c is moved before a and b are joined.

Red edges are naturally directed; however, for issues of connectivity and cyclicity (Lemma 5 and Theorem 2) we consider red, blue, and green edges to be undirected. The *red-blue graph* is the graph whose nodes are the symbols in w and whose edges are the red edges and the blue edges.

The following lemmas show that red edges are essentially "savings."

Lemma 4 *The number of steps plus the number of red edges is one less than the length of w.*

Proof. Tber of blocks is initially the length of w and decreases to 1. A step decreases the number of blocks by 1. Whenever two blocks, each containing a symbol that has never been moved, are joined, the number of blocks decreases by 1. There is one such joining for each red edge. □

Lemma 5 *The red-blue graph is acyclic.*

Proof. Let G be the red-blue graph for the block sorting of w. For a subsequence u of w consider the block sorting derived from the sorting of w by leaving out all sorting steps involving symbols not in u, then let H be the red-blue graph for this inherited block sorting of u. It is clear that for symbols a, b in u, if there is a blue (red) path in G then there is a blue (red) path in H.

Assume now that G have a cycle K of length ℓ, then let u be the subsequence of w consisting precisely of those symbols used in K, *i.e.* $|u| = \ell$. Because of the preservation of blue and red paths there is a cycle in the red blue graph for the inherited block sorting of u, which has length at least ℓ. Thus, that graph has at least ℓ edges, contradicting Theorem 1 and Lemma 4. □

From Lemmas 4 and 5 we immediately obtain:

Theorem 2 *The sorting is perfect if and only if the red-blue graph is a tree.*

Proof. Immediate from Lemmas 4 and 5. □

We now give a number of results which limit the number of red edges in the red-blue graph and which are useful later.

Lemma 6 *For any given node x, there can be at most one $y < x$ for which there is a red edge from y to x. Furthermore, there can be at most one $z > x$ for which there is a red edge from x to z.*

Proof. Assume there was a red edge from $y' < y$ to x. Then y would have to move before y' contradicting the existence of the red edge from y to x. A similar argument applies to z. □

Lemma 7 *If $a \dots c \dots b \dots d$, then there cannot be both a red edge from a to b and a red edge from c to d.*

Proof. Since there is a red edge from a to b, c is moved before b. If there is also a red edge from c to d, then c is moved before b, contradiction. □

The following lemma follows directly from duality:

Lemma 8 *If $a < c < b < d$, then there cannot be both a red edge from a to b and a red edge from c to d.*

Lemma 9 *If $a < c < d < b$ and $c \dots a \dots b \dots d$, then there cannot be both a red edge from a to b and a red edge from c to d.*

Proof. Because $a < c < b$, and there is a red edge between a and b, c has to move before a. On the other hand, because of the red edge from c to d, a is moved before c is moved, contradiction. □

We conclude this section with an alternative lower bound, which is based on counting alternating cycles in a certain directed graph. We call w *anchored* if the first symbol of w is also the smallest, and the last symbol is the largest. If not, we can simply "anchor" w by adding the prefix 0, defined to be smaller than any symbol of w, and appending "∞," defined to be smaller than any symbol of w. Trivially, the number of steps required for block sorting does not change. Henceforth in this section, we assume that w is anchored. As before, let n be the length of w.

We now define *black* and *white* edges. If ab is a substring of w, we say that there is a white edge from b to a. If $a < b$ and there is no symbol of w which is larger than a and smaller than b, we say that there is a black edge from a to b. We define the *breakpoint* graph G to be the directed graph whose nodes are the symbols of w and whose edges are the black and white edges. We define an *alternating cycle* (or just "*cycle*" if the context is clear) to be a directed cycle whose edges are alternately white and black. Note that every symbol except the first and last has black out-degree 1, white out-degree 1, black in-degree 1, and white in-degree 1. The first symbol has black out-degree 1 and white in-degree 1, while he last symbol has black in-degree 1 and white out-degree 1. Thus, each edge of G lies in precisely one alternating cycle.

We define the *black length* of a cycle to be its number of black edges. We say that a cycle is *odd* if it has odd black length, otherwise, it is *even*. Let $odd\_cycles(w)$ be the number of odd cycles in G. We note that the black length of any cycle is at least 2, since we assume there are no consecutive symbols in w.

Theorem 3 *Let w be an anchored string of length n. The number of steps required to sort w is at least $\frac{1}{2}(n - 1 - odd\_cycles(w))$.*

Proof. If C is any alternating cycle of the breakpoint graph G of w, and the black length of C is c, we define $\Phi(C) = \lfloor \frac{c}{2} \rfloor$. Thus, $\Phi(C) = 1$ if $c = 3$, while $\Phi(C) = 2$ if $c = 4$ or $c = 5$, and so forth. We define $\Phi(w) = \sum_C \Phi(C)$, where C ranges over all alternating cycles of G. Note that $\Phi(w) = \frac{1}{2}(n - 1 - odd\_cycles(w))$.

We can show that if x is any symbol other than the first or last symbol, and if w' is the resulting string after $step(x)$, then $\Phi(w) - \Phi(w') \leq 1$. Therefore, the number of steps needed to sort w is at least $\Phi(w)$.

Cycles are edge disjoint, but not node disjoint. We define a Type I symbol to be a symbol which appears as a node twice in one cycle, and a Type II symbol to be a symbol which appears in two different cycles. Each symbol, other than the first and last symbols, is either Type I or Type II. Recall that the move $step(x)$ is defined by choosing a symbol x of w, deleting x, then combining all consecutive symbols, if any.

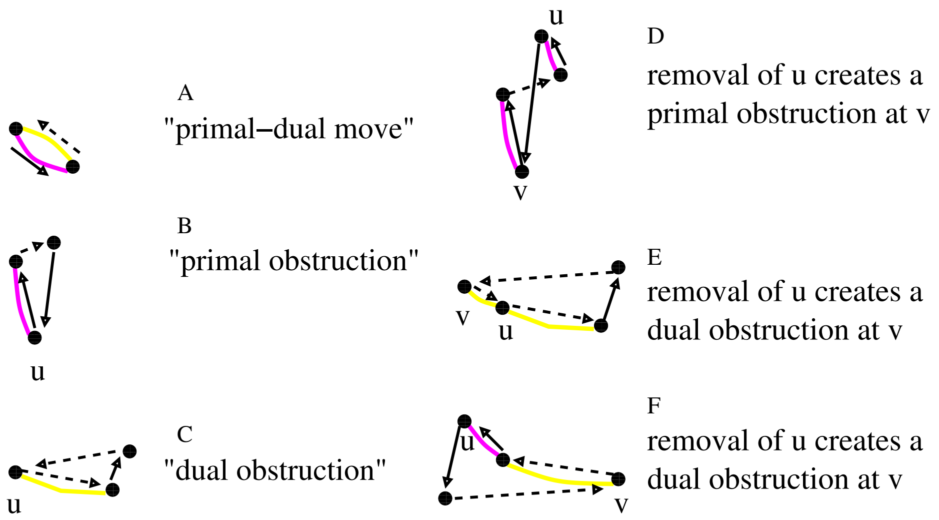

Figure 2: Useful Local Properties.

We summarize the effects of $step(x)$ on the numbers of even and odd cycles, as follows:

1. Suppose x is a Type II symbol. Let P and Q be the two cycles that contain x. Suppose P has black length p and Q has black length q, The effect of $step(x)$ on the alternating cycles is: (a) P and Q are removed. (b) There is a new cycle T whose black length is $p+q-1$. and (c) All other cycles remain.

2. Suppose x is a Type I symbol, which appears twice in a cycle T, of black length t. (Then the actual length of the T is $2t$.) Then T is the union of two cycles (in this case, not alternating cycles) which meet at x, of lengths u and v, where u and v are both odd and $u+v=2t$. Note, also, that $u, v \geq 3$. The effect of $step(x)$ on the alternating cycles is as follows: (a) T is removed. (b) If $u \geq 5$, then there is a new cycle U of black length $\frac{u-1}{2}$. (c) If $v \geq 5$, then there is a new cycle V of black length $\frac{v-1}{2}$. (d) All other cycles remain.

Checking all the cases, we can verify that the value of Φ cannot decrease by more than 1, and we are done. $\square$

3. Local Property Algorithms

A local property algorithm attempts to minimize the number of steps to block sort a string by deciding locally whether to delete an element. For example, an algorithm that seeks to eliminate blue edges must remember whether any given symbol is the left end point of a blue edge. A local property algorithm could look for other properties, but we require that the algorithm maintain the information at a symbol in constant time per node update.

Figure 2 illustrates examples of such properties. Here blue edges are shaded dark, green edges light. Arrows denote the "order edges" from Theorem 3 with inverse white edges depicted as solid arrows, and black edges rendered as broken arrows. For example, if a node is simultaneously the left end point of a blue and a green edge, then any optimal algorithm can without loss of optimality delete such an element first. (We call this a primal-dual move; in the figure this is situation A.) Lemma 2 suggests looking for nodes the deletion of which causes the string length to decrease by up to three. Properties B and C ("primal" and "dual obstructions") identify such nodes. If there are no nodes of type B and C then a possible strategy is to identify nodes the deletion of which will cause situations B and C in the next step; properties D, E , and F identify such nodes. Formally properties A though F are defined later in Table 3.

Specifically, a local property algorithm maintains a data structure called a *quad list*. If the string to be sorted is $w = w_1, \ldots, w_n$ then the quad list is a doubly linked list consisting of the elements $(w_1, 1), \ldots, (w_n, n)$ with a head pointer pointing to $(w_1, 1)$. Additionally there are bidirectional pointers for the dual order, *i.e.* there is a bidirectional pointer between (u, i) and (v, j) if $u < v$ and there exists no y in w such that $u < y < v$. We call these additional pointers the *value pointers*, the former *positional pointers*. Each node therefore stores four pointers. We call the four associated pointer fields, `left`, `right` for the positional pointers and `up`, `down` for value pointers. Thus for node `u` the previous or next elements in respective orderings are `u.left`, `u.right`, `u.down`, `u.up`, where by a slight abuse of notation we use the pointer to mean also the node under the pointer. Finally, we assume that we have a constant number of extra boolean fields to store local information at each node of the list as needed. For example, there could be an extra field `u.blue_edge` whose value is true if the symbol in `u` is the left end point of a blue edge. We require that such properties are local in the sense that the field can be calculated and maintained in $O(1)$ time per node. The quad list can be set up in $O(n \log n)$ time. Thus, a local property algorithm take $O(n \log n)$ setup-time and $O(n)$ for all subsequent steps. Table 3 shows local properties A through F in terms of our data structure.

Type	Condition	Effect of Deletion
A	`u.right.up = u`	$n' \leq n - 1$
B	`u.left.up.left = u`	$n' \leq n - 2$
C	`u.down.right.down = u`	$n' \leq n - 2$
D	`u.left.left.up.left = u`	$n' \leq n - 1$, `u.right` becomes type B
E	`u.down.right.down.down = u`	$n' \leq n - 1$, `u.up` becomes type C
F	`u.left.up.up.left = u`	$n' \leq n - 1$, `u.left.up` becomes type C

Latifi *et al.* [6] give an $O(n^2)$ algorithm, called the 1-lookahead algorithm, for which they show that computer experimentation gives relatively good results in practice. Their algorithm is the local property algorithm with the properties of Table 3, where properties A ... F have alphabetical priority. If no property is available at any step the 1-lookahead algorithm simply arbitrarily removes the last node. Thus, our data structure improves their run time to $O(n \log n)$.

4. The Block Sorting Problem is $\mathcal{NP}$-Complete

The 0-1 block sorting problem is, given string w, can w be block sorted in k steps? Henceforth, when we say the "block sorting problem," or just "block sorting," we mean the 0-1 block sorting problem. We show now that this problem is $\mathcal{NP}$-complete. Trivially, block sorting is in $\mathcal{NP}$. To prove $\mathcal{NP}$-completeness, we reduce *boolean 3-satisfiability*, a well-known [4] $\mathcal{NP}$-complete problem, to block sorting.

Suppose $\mathcal{B} = \mathcal{C}^1\mathcal{C}^2\ldots\mathcal{C}^m$ is a boolean expression consisting of the conjunction of $m > 0$ clauses, each of which is the disjunction of exactly 3 terms, each of which is either a variable or the negation of a variable. Assume there are n variables. We introduce an ordered alphabet $\Sigma_{n,m}$ consisting of $4nm + 2m + 4n + 1$ distinct symbols. We then construct a string $\mathcal{X}$ of length $8m + 4n + 1$ over $\Sigma_{n,m}$ in which no symbol is repeated. Finally, we show that $\mathcal{B}$ is satisfiable if and only if $\mathcal{X}$ can be block sorted in $6m + 2n - 1$ steps.

$\Sigma_{n,m}$ consists of the following symbols:

- p_i^j,$\bar{p}_i^j$,q_i^j,$\bar{q}_i^j$, for all $1 \le i \le n$ and $1 \le j \le m$. We call these *term symbols*. We call p_i^j and $\bar{p}_i^j$ *left term symbols*, and we call q_i^j and $\bar{q}_i^j$ *right term symbols*.
- ℓ^j, r^j for all $1 \le j \le m + n$. We call these *clause control symbols*
- u_i, v_i for all $1 \le i \le n$. We call these *variable control symbols*
- s. We call this the *separator*.

The ordering on $\Sigma_{n,m}$ is generated by the following rules:

- $u_i < p_i^k < p_i^j < \bar{p}_i^k < \bar{p}_i^j < q_i^j < q_i^k < \bar{q}_i^j < \bar{q}_i^k < v_i < s$ for all $1 \le i \le n$ and $1 \le j < k \le m$.
- $v_{i-1} < u_i$ for all $1 < i \le n$
- $s < \ell^k < r^k < \ell^j < r^j$ for all $1 \le j < k \le m + n$.

Assume now that for an instance of the boolean 3-satisfiability problem the names of the variables are x_i for $1 \le i \le n$. A clause must be of the form $z_a + z_b + z_c$, where each z_i is either x_i or $\bar{x}_i$. Without loss of generality, $a > b > c$. We first define the *simple encoding* of a clause, using symbols which have no superscripts. The simple encoding of x_i uses the two symbols p_i and q_i, while the simple encoding of $\bar{x}_i$ uses the two symbols $\bar{p}_i$ and $\bar{q}_i$, The simple encoding of a clause is eight symbols, beginning with ℓ and ending with r. The remaining six symbols are those needed for the terms, but interleaved; first the p's, then the q's. For example, the simple encoding of the clause $(x_5 + x_3 + \bar{x}_2)$ is $\ell p_5 p_3 \bar{p}_2 q_5 q_3 \bar{q}_2 r$. For the true encoding of a clause, insert the index of that clause as a superscript for every symbol. For example, if $\mathcal{C}^4 = (x_5 + x_3 + \bar{x}_2)$, then its encoding is $\ell^4 p_5^4 p_3^4 \bar{p}_2^4 q_5^4 q_3^4 \bar{q}_2^4 r^4$.

After the true encoding of the clauses we add "control sequences" $\ell^{m+i} u_i v_i r^{m+i}$, for $1 \le i \le n$, $\ell^{m+i} u_i v_i r^{m+i}$. Finally, we finish off the encoding: $\mathcal{X}$ consists of s, followed by the encodings of the clauses in order, followed by the control sequences for each i from 1 to n.

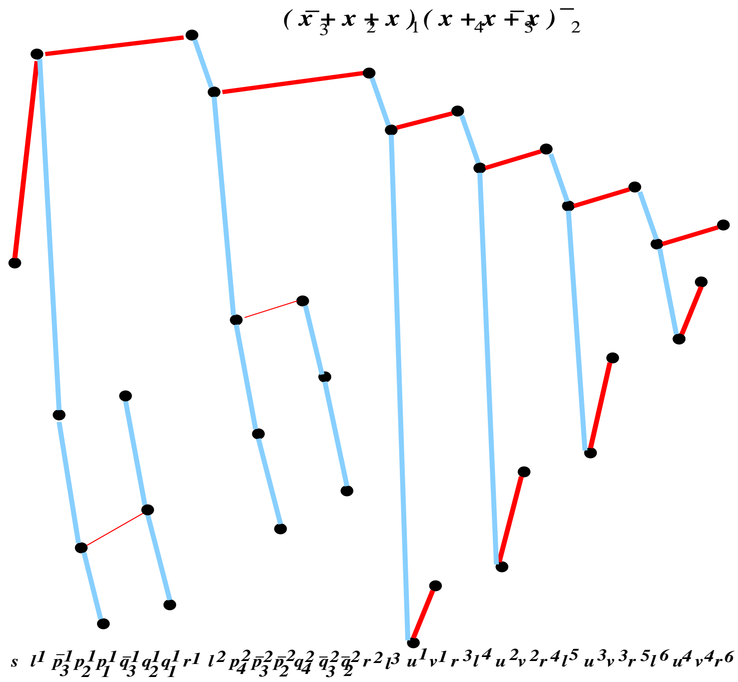

Figure 3: Reduction of the Expression $(\bar{x}_3 + x_2 + x_1)(x_4 + \bar{x}_3 + \bar{x}_2)$

Figure 3 shows a complete reduction of the expression $(\bar{x}_3+x_2+x_1)(x_4+\bar{x}_3+\bar{x}_2)$. Red edges are dark, and blue edges are shown in the lighter shading. Note that the existence of the two thin red edges, which complete the tree, is due to the fact that in the first clause x_2 and in the second clause x_4 can be chosen as validating terms to satisfy the expression.

We are now ready to prove the result: We first prove that satisfiability of a clause implies a perfect sorting.

Lemma 10 *If $\mathcal{B}$ is satisfiable, there exists a perfect sorting of $\mathcal{X}$.*

Proof. Given $\mathcal{B}$, and given a satisfying assignment of the variables, we now show how to block sort $\mathcal{X}$ in exactly $6m + 2n - 1$ steps.

For each $1 \leq j \leq m$, designate one term of $\mathcal{C}^j$ which is true under the assignment. Call this the *relevant* term of $\mathcal{C}^j$. The steps are as follows:

1. Delete all symbols which form the encodings of terms which are not relevant. This takes $4m$ steps.

2. For all j, starting with 1 and ending with m, in that order, join and the delete the two symbols which form the encoding of the relevant term of $\mathcal{C}^j$. These symbols will be either $p_i^j q_i^j$ or $\bar{p}_i^j \bar{q}_i^j$ for some i. This takes m steps. Note that elements become available for joining due to the order in which they are processed.

3. Join all pairs of the form $u_i v_i$ and delete them. This takes n steps.

4. For all $2 \leq j \leq m+i$, join the symbols $\ell^j r^j$ and delete them. This takes $m+n-1$ steps.

The remaining symbols are sorted. □

We now show that the existence of a perfect sorting implies satisfiability.

Lemma 11 *If there is a perfect sorting of $\mathcal{X}$, then $\mathcal{B}$ is satisfiable.*

Proof. If there is a perfect sorting, then by Theorem 2 the red-blue graph is connected, and acyclic, *i.e.* a tree. It has exactly $6m+2n-1$ blue edges. All blue edges connect symbols which are adjacent in $\mathcal{X}$. The components of the blue graph, which we call blue components, are thus substrings of $\mathcal{X}$. There are $2m+2n+2$ blue components. We refer to the space between two blue components as a *gap*. The $2m+2n+1$ red edges must connect those components, and must not violate the conditions given by Lemmas 8, 7, and 9. We shall show that these requirements imply the needed properties.

Claim A: There is no red edge from s to r^j. Instead, for some $1 \leq \hat{\jmath} \leq m$, there is a red edge from s to $\ell^{\hat{\jmath}}$. Proof: If there is a red edge from s to r^j, there can be no other red edge across the gap to the left of r^j, by Lemmas 6 and 8. Thus, the graph is disconnected, a contradiction. On the other hand, there must be a red edge from s to some x, else the graph is disconnected. Since $s < x$, x must be either ℓ^j or r^j, for some j, by Lemma 7. Since there is no red edge from s to r^j, the claim follows.

Claim B: For all j, there is a red edge from ℓ^j to r^j. Proof: First consider $j > \hat{\jmath}$: If $\hat{\jmath} = n+m$, there is nothing to prove. Otherwise, we proceed by induction on j, beginning with $j = n+m$. Since r^j is the first symbol of its component (*i.e.*, there is no connection via a blue edge) there must be a red edge from x to y where $x \ldots r^j$ and either $y = r^j$ or $r^j \ldots y$. The case $r^j \ldots y$ is ruled out by the inductive hypothesis, using Lemma 8. Thus there is a red edge from x to r^j. By Claim A and Lemma 7, $s < x < r^j$. The only choice for x is ℓ^j.

Now we consider the case $j < \hat{\jmath}$: For any $j < \hat{\jmath}$, there is a red edge from ℓ^j to r^j. If $\hat{\jmath} = 1$, there is nothing to prove. Otherwise, we use induction on j, beginning with $j = \hat{\jmath} - 1$. Suppose $j = \hat{\jmath} - 1$. There must be a path from ℓ^j to r^j. By Lemmas 7 and 8, and by Claim A, the only way such a path can exist is for there to be a red edge from ℓ^j to r^j. Now suppose $j < \hat{\jmath} - 1$. There must be a path from ℓ^j to r^j. By the inductive hypothesis, there are red edges from ℓ^k to r^k for all $j < k < \hat{\jmath}$. By Lemmas 8 and 9, the only way such a path can exist is if there is a red edge from ℓ^j to r^j.

For the case $j = \hat{\jmath}$, we need to show the following claim first:

Claim C: There is a red edge from u_i to v_i. Proof: There must be a red edge connecting v_i to some other symbol. By the previous arguments in Claim B and Lemma 7, the only possibility for that other symbol is u_i. This concludes the proof of Claim C.

Returning to Claim B, we now can establish that there is a red edge from $\ell^{\hat{\jmath}}$ to $r^{\hat{\jmath}}$: Since $r^{\hat{\jmath}}$ is the first symbol of its component, there must be a red edge from x to y where $x \ldots r^{\hat{\jmath}}$ and either $y = r^j$ or $r^j \ldots y$. The case $r^j \ldots y$ is ruled out by Claim A and Lemma 8. Claim A rules out the possibility that $x \ldots \ell^{\hat{\jmath}}$. By Claim

C and Lemma 7, x cannot be a term symbol. Thus, $x = \ell^{\hat{j}}$, the only remaining possibility.

Claim D: For each $1 \leq j \leq m$, there is some i for which there is a red edge from p_i^j to q_i^j, in which case x_i is a term of $\mathcal{C}^j$, or from $\bar{p}_i^j$ to $\bar{q}_i^j$, in which case $\bar{x}_i$ is a term of $\mathcal{C}^j$. Furthermore, if there is a red edge from p_i^j to q_i^j, then there is no red edge from $\bar{p}_i^k$ to $\bar{q}_i^k$. Proof: We first show that for each $1 \leq j \leq m$, there is a red edge from x to y where $\ell^j \ldots x \ldots y \ldots r^j$, and where x is a left term symbols and y is a right term symbol. This is because the blue component containing the three right term symbols between ℓ^j and r^j must have a red edge. All other possibilities for x are ruled out by Claim B.

To prove that $\mathcal{B}$ is satisfiable we select a validating term for each clause such that no two validating terms are contradictory. Let the validating term of clause $\mathcal{C}^j$ be x_i if the red edge connects p_i^j and q_i^j, and $\bar{x}_i$ if the red edge connects $\bar{p}_i^j$ to $\bar{q}_i^j$. Claim D guarantees no that two of the validating terms contradict each other. □

Lemma 10 and 11 now establish our result:

Theorem 4 *Block sorting is $\mathcal{NP}$-complete.*

References

1. V. Bafna and P.A. Pevzner. Sorting by transposition. *SIAM Journal on Discrete Mathematics*, 11:224–240, 1998.
2. V. Bafna and P.A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25:272–289, 1999.
3. A. Caprara. Sorting by reversals is difficult. In *Proceedings 1st Conference on Computational Molecular Biology*, pages 75–83. ACM, 1997.
4. M. R Garey and D. S. Johnson. *Computers and intractability – A guide to the theory of NP-completeness*. Freeman, San Francisco, 1979.
5. W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.
6. R. Gobi, S. Latifi, and W. W. Bein. Adaptive sorting algorithms for evaluation of automatic zoning employed in OCR devices. In *Proceedings of the 2000 International Conference on Imaging Science, Systems, and Technology*, pages 253–259. CSREA Press, 2000.
7. L.S. Heath and J.P.C. Vergara. Sorting by bounded block-moves. *Discrete Applied Mathematics*, 88:181–206, 1998.
8. L.S. Heath and J.P.C. Vergara. Sorting by short block-moves. *Algorithmics*, 28 (3):323–354, 2000.
9. H. Heydari and I. H. Sudborough. On the diameter of the pancake network. *Journal of Algorithms*, 25(1):67–94, 1997.
10. J. Kanai, S. V. Rice, and T.A. Nartker. Automatic evaluation of OCR zoning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17 (1):86–90, 1995.
11. S. Latifi. How can permutations be used in the evaluation of automatic zoning evaluation? In *ICEE 1993*. Amir Kabir University, 1993.