Synergistic Solutions on MultiSets*

Jérémy Barbay[1], Carlos Ochoa[2], and Srinivasa Rao Satti[3]

1 **Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile**
`jeremy@barbay.cl`
2 **Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile**
`cochoa@dcc.uchile.cl`
3 **Department of Computer Science and Engineering, Seoul National University, Seoul, South Korea**
`ssrao@cse.snu.ac.kr`

Abstract

Karp et al. (1988) described Deferred Data Structures for Multisets as "lazy" data structures which partially sort data to support online rank and select queries, with the minimum amount of work in the worst case over instances of size n and number of queries q fixed. Barbay et al. (2016) refined this approach to take advantage of the gaps between the positions hit by the queries (i.e., the structure in the queries). We develop new techniques in order to further refine this approach and take advantage all at once of the structure (i.e., the multiplicities of the elements), some notions of local order (i.e., the number and sizes of runs) and global order (i.e., the number and positions of existing pivots) in the input; and of the structure and order in the sequence of queries. Our main result is a synergistic deferred data structure which outperforms all solutions in the comparison model that take advantage of only a subset of these features. As intermediate results, we describe two new synergistic sorting algorithms, which take advantage of some notions of structure and order (local and global) in the input, improving upon previous results which take advantage only of the structure (Munro and Spira 1979) or of the local order (Takaoka 1997) in the input; and one new multiselection algorithm which takes advantage of not only the order and structure in the input, but also of the structure in the queries.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3.3 Information Search and Retrieval

Keywords and phrases deferred data structure, multivariate analysis, quick sort, select

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.31

1 Introduction

Consider a *multiset* $\mathcal{M}$ of size n. The *multiplicity* of an element x of $\mathcal{M}$ is the number m_x of occurrences of x in $\mathcal{M}$. We call the distribution of the multiplicities of the elements in $\mathcal{M}$ the *input structure*. As early as 1976, Munro and Spira [19] described a variant of the algorithm `MergeSort` using counters, which optimally takes advantage of the input structure when sorting a multiset $\mathcal{M}$ of n elements. Munro and Spira measure the "difficulty" of the instance in terms of the "input structure" by the entropy function $\mathcal{H}(m_1, \ldots, m_\sigma) = \sum_{i=1}^{\sigma} \frac{m_i}{n} \log \frac{n}{m_i}$,

* C. Ochoa is supported by CONICYT-PCHA/Doctorado Nacional/2013-63130161 (Chile). J. Barbay is supported by the project Fondecyt Regular no 1170366 from Conicyt and the Millennium Nucleus RC130003 "Information and Coordination in Networks" with code P10-024-F.

where σ is the number of distinct elements in $\mathcal{M}$ and $m_1, \ldots, m_\sigma$ are the multiplicities of the σ distinct elements in $\mathcal{M}$ (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. The time complexity of the algorithm is within $O(n(1 + \mathcal{H}(m_1, \ldots, m_\sigma))) \subseteq O(n(1+\log \sigma)) \subseteq O(n \log n)$.

Any array $\mathcal{A}$ representing a multiset lists its element in some order, which we call the *input order* and denote by a tuple. Maximal sorted subblocks in $\mathcal{A}$ are a local form of input order and are called *runs* [16]. As early as 1973, Knuth [16] described a variant of the algorithm `MergeSort` using a prepossessing step taking linear time to detect *runs* in the array $\mathcal{A}$. Takaoka [20] described a new sorting algorithm that optimally takes advantage of the distribution of the sizes of the runs in the array $\mathcal{A}$, which yields a time complexity within $O(n(1 + \mathcal{H}(r_1, \ldots, r_\rho))) \subseteq O(n(1+\log \rho)) \subseteq O(n \log n)$, where ρ is the number of runs in $\mathcal{A}$ and $r_1, \ldots, r_\rho$ are the sizes of the ρ *runs* in $\mathcal{A}$ (such that $\sum_{i=1}^{\rho} r_i = n$), respectively.

Given an element x of a multiset $\mathcal{M}$ and an integer $j \in [1..n]$, the *rank* $\texttt{rank}(x)$ of x is the number of elements smaller than x in $\mathcal{M}$, and *selecting* the j-th element in $\mathcal{M}$ corresponds to computing the value $\texttt{select}(j)$ of the j-th smallest element (counted with multiplicity) in $\mathcal{M}$. Those operations are central to the navigation of the Burrows-Wheeler transform [17] of a text when searching for occurrences of a pattern in it. As early as 1961, Hoare [12] showed how to support `rank` and `select` queries in average linear time, a result later improved to worst case linear time by Blum et al. [7]. Twenty years later, Dobkin and Munro [10] described a MultiSelection algorithm that supports several `select` queries and whose running time is optimal in the worst case over all multisets of size n and all sets of q queries hitting positions in the multisets separated by *gaps* (differences between consecutive `select` queries in sorted order) of sizes $g_0, \ldots, g_q$. Karp et al. [15] further extended Dobkin and Munro's result [10] to the online context, where the multiple `rank` and `select` queries arrive one by one. They called their solution a Deferred Data Structure and describe it as "lazy", as it partially sorts data, performing the minimum amount of work necessary in the worst case over all instances for a fixed n and q. Barbay et al. [2] refined this result by taking advantage of the gaps between the positions hit by the queries (i.e., the *query structure*).

This suggests the following questions:

1. Is there a sorting algorithm for multisets which takes the best advantage of both its *input order* and its *input structure* in a synergistic way, so that it performs as good as previously known solutions on all instances, and much better on instances where it can take advantage of both at the same time?
2. Is there a multiselection algorithm and/or a deferred data structure for answering `rank` and `select` queries which takes the best advantage not only of both of those notions of easiness in the input, but also of notions of easiness in the queries, such as the *query structure* and the *query order*?

We answer both questions affirmatively: In the context of Sorting, this improves upon both algorithms from Munro and Spira [19] and Takaoka [20]. In the context of MultiSelection and Deferred Data Structure for `rank` and `select` on Multisets, this improves upon Barbay et al.'s results [2] by adding three new measures of difficulty (input order, input structure and query order) to the single one previously considered (query structure). Additionally, we correct the analysis of the `Sorted Set Union` algorithm by Demaine et al. [9] (Section 2.2), and we define a simple yet new notion of "global" input order (Section 2.4), not mentioned in previous surveys [11, 18] nor extensions [3].

We present our results incrementally, each building on the previous one, such that **the most complete and complex result is in Section 4**. In Section 2 we describe how to measure the interaction of the order (local and global) with the structure in the input, and two new synergistic Sorting algorithms based on distinct paradigms (i.e., merging vs

splitting) which take advantage of both the input order and structure. We refine the second of those results in Section 3 with the analysis of a MultiSelection algorithm which takes advantage of not only the order and structure in the input, but also of the *query structure*, in the offline setting. In Section 4 we analyze an online Deferred Data Structure taking advantage of the order and structure in the input on one hand, and of the order and structure in the queries on the other hand. We conclude with a discussion of our results in Section 5.

2 Sorting Algorithms

We review in Section 2.1 the algorithms `MergeSort with Counters` described by Munro and Spira [19] and `Minimal MergeSort` described by Takaoka [20]: each takes advantage of distinct features in the input. In Sections 2.2 and 2.3, we describe two synergistic Sorting algorithms, which outperform both `MergeSort with Counters` and `Minimal MergeSort` by taking advantage of both the order (local and global) and the structure in the input, in a synergistic way.

2.1 Known Algorithms

The algorithm `MergeSort with Counters` described by Munro and Spira [19] is an adaptation of the traditional sorting algorithm `MergeSort` that optimally takes advantage of the input structure when sorting a multiset $\mathcal{M}$ of size n. The algorithm divides $\mathcal{M}$ into two parts of equal size, sorts both parts recursively, and then merges the two sorted lists. When two elements of same value v are found, one is discarded and a counter holding the number of occurrences of v is updated. Munro and Spira measure the "difficulty" of the instance in terms of the input structure by the entropy function $\mathcal{H}(m_1, \ldots, m_\sigma) = \sum_{i=1}^{\sigma} \frac{m_i}{n} \log \frac{n}{m_i}$, where σ is the number of distinct elements in $\mathcal{M}$ and $m_1, \ldots, m_\sigma$ are the multiplicities of the σ distinct elements in $\mathcal{M}$ (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. The time complexity of the algorithm is then within $O(n(1 + \mathcal{H}(m_1, \ldots, m_\sigma))) \subseteq O(n(1+\log \sigma)) \subseteq O(n \log n)$.

The algorithm `Minimal MergeSort` described by Takaoka [20] optimally takes advantage of the local input order, as measured by the decomposition into runs when sorting an array $\mathcal{A}$ of size n. The main idea is to detect the runs first and then merge them pairwise. The runs are detected in linear time. Merging the two shortest runs at each step further reduces the number of comparisons, making the running time of the merging process adaptive to the entropy of the sequence formed by the sizes of the runs. If the array $\mathcal{A}$ is formed by ρ runs and $r_1, \ldots, r_\rho$ are the sizes of the ρ runs (such that $\sum_{i=1}^{\rho} r_i = n$), then the algorithm sorts $\mathcal{A}$ in time within $O(n(1 + \mathcal{H}(r_1, \ldots, r_\rho))) \subseteq O(n(1+\log \rho)) \subseteq O(n \log n)$.

The algorithms `MergeSort with Counters` and `Minimal MergeSort` are incomparable, in the sense that neither one performs always better than the other. Simple modifications and combinations of these algorithms do not take full advantage of both the local input order and the input structure (see the extended version [5] for detailed counter examples).

In the following sections we describe two sorting algorithms that take the best advantage of both the order (local and global) and structure in the input all at once when sorting a multiset. The first one is a straightforward application of previous results, while the second one prepares the ground for the MultiSelection algorithm (Section 3) and the Deferred Data Structures (Section 4), which take advantage of the order (local and global) and structure in the input and of the order and structure in the queries.

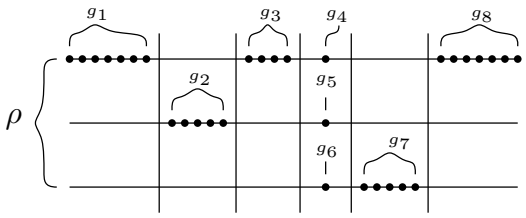

Figure 1 An instance of the Sorted Set Union problem with $\rho = 3$ sorted sets. In each sorted set $\mathcal{A}$, the entry $\mathcal{A}[i]$ is represented by a point of x-coordinate $\mathcal{A}[i]$. The sizes $(g_i)_{i\in[1..8]}$ of the blocks that form the sets are indicated. The sizes g_4, g_5 and g_6 are 1 because they correspond to elements of equal value and they determine the 4-th member of the partition π with value m_4 equals 3. The vertical bars separate the members of π.

2.2 "Kind-of-new" Sorting Algorithm `DLM Sort`

In 2000, Demaine et al. [9] described the algorithm `DLM Union`, an instance optimal algorithm that computes the union of ρ sorted sets. The algorithm scans the sets from left to right identifying *blocks* of consecutive elements in the sets that are also consecutive in the sorted union (see Figure 1 for a graphical representation of such a decomposition on a particular instance of the Sorted Set Union problem). In a minor way we refine their analysis as follows:

These blocks determine a partition π of the output into intervals such that any singleton corresponds to a value that has multiplicity greater than 1 in the input, and each other interval corresponds to a block as defined above. Each member i of π has a value m_i associated with it: if the member i of π is a block, then m_i is 1, otherwise, if the member i of π is a singleton corresponding to a value of multiplicity q, then m_i is q. If the instance is formed by δ blocks of sizes $g_1, \ldots, g_\delta$ such that these blocks determine a partition π of size χ whose members have values $m_1, \ldots, m_\chi$, we express the time complexity of `DLM Union` as within $\Theta(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$. This time complexity is within a constant factor of the complexity of any other algorithm computing the union of these sorted sets (i.e., the algorithm is instance optimal).

We adapt the `DLM Union` algorithm for sorting a multiset. The algorithm `DLM Sort` detects the runs first through a linear scan and then applies the algorithm `DLM Union`. After that, transforming the output of the union algorithm to yield the sorted multiset takes only linear time. The following corollary follows from our refined analysis above:

▶ **Corollary 1.** *Given a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks of sizes $g_1, \ldots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \ldots, m_\chi$, the algorithm* `DLM Sort` *performs within $n+O(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$ data comparisons. This number of comparisons is optimal in the worst case over multisets of size n formed by ρ runs and δ blocks of sizes $g_1, \ldots, g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \ldots, m_\chi$.*

While the algorithm `DLM Sort` answers the Question 1 from Section 1, it does not yield a MultiSelection algorithm nor a Deferred Data Structure answering Question 2. In the following section we describe another sorting algorithm that also optimally takes advantage of the local order and structure in the input, but which is based on a distinct paradigm, more suitable to such extensions.

Algorithm 1 `Quick Synergy Sort`

Input: A multiset $\mathcal{M}$ of size n
Output: A sorted sequence of $\mathcal{M}$

1: Compute the ρ runs of respective sizes $(r_i)_{i\in[1..\rho]}$ in $\mathcal{M}$ such that $\sum_{i=1}^{\rho} r_i = n$;
2: Compute the median μ of the middles of the runs, note $j \in [1..\rho]$ the run containing μ;
3: Perform doubling searches for the value μ in all runs except the j-th, starting at both ends of the runs in parallel;
4: Find the maximum $\max_\ell$ (minimum $\min_r$) among the elements smaller (resp., greater) than μ in all runs except the j-th;
5: Perform doubling searches for the values $\max_\ell$ and $\min_r$ in the j-th run, starting at the position of μ;
6: Recurse on the elements smaller than or equal to $\max_\ell$ and on the elements greater than or equal to $\min_r$.

2.3 New Sorting Algorithm `Quick Synergy Sort`

Given a multiset $\mathcal{M}$, the algorithm `Quick Synergy Sort` identifies the *runs* in linear time through a scanning process. It computes a pivot μ, which is the median of the set formed by the middle elements of each run, and partitions each *run* by μ. This partitioning process takes advantage of the fact that the elements in each *run* are already sorted. The insertion ranks of the pivots in the runs are identified by doubling searches [6]. It then recurses on the elements smaller than μ and on the elements greater than μ. (See Algorithm 1 for a more formal description).

▶ **Definition 2** (Median of the middles)**.** Given a multiset $\mathcal{M}$ formed by runs, the "*median of the middles*" is the median element of the set formed by the middle elements of each run.

The number of data comparisons performed by the algorithm `Quick Synergy Sort` is asymptotically the same as the number of data comparisons performed by the algorithm `DLM Sort` described in the previous section. We divide the proof into two lemmas. We first bound the number of data comparisons performed by all the doubling searches of the algorithm `Quick Synergy Sort` (i.e., steps 3 and 5 of the Algorithm 1).

▶ **Lemma 3.** *Let $g_1, \ldots, g_k$ be the sizes of the k blocks that form the r-th run. The overall number of data comparisons performed by the doubling searches of the algorithm* ***`Quick Synergy Sort`*** *to find the values of the medians of the middles in the r-th run is within $O(\sum_{i=1}^{k} \log g_i)$.*

Proof. Every time the algorithm finds the insertion rank of one of the medians of the middles in the r-th run, it partitions the run by a position separating two blocks. The doubling search steps can be represented as a tree. Each node of the tree corresponds to a step. Each internal node has two children, which correspond to the two subproblems into which the step partitions the run. The cost of the step is less than four times the logarithm of the size of the child subproblem with smaller size, because of the two doubling searches in parallel. The leaves of the tree correspond to the blocks themselves.

We prove that at each step the total cost is bounded by eight times the sum of the logarithms of the sizes of the leaf subproblems. This is done by induction over the number of steps. If the number of steps is zero then there is no cost. For the inductive step, if the number of steps increases by one, a new doubling search step is done and a leaf subproblem is partitioned into two new subproblems. At this step, a leaf of the tree is transformed into an

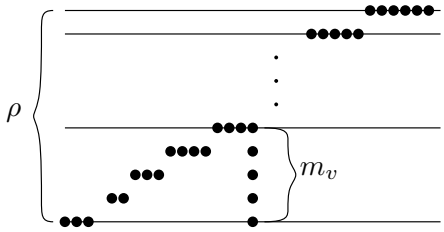

Figure 2 A multiset $\mathcal{M}$ formed by ρ runs. Each entry $\mathcal{M}[i]$ is represented by a point of x-coordinate $\mathcal{M}[i]$. There is an element of multiplicity m_v present in the last m_v runs and the rest of the runs are formed by only one block.

internal node and two new leaves are created. Let a and b such that $a \leq b$ be the sizes of the new leaves created. The cost of this step is less than $4 \log a$. The cost of all the steps then increases by $4 \lg a$, and hence the sum of the logarithms of the sizes of the leaves increases by $8(\lg a + \lg b) - 8 \lg(a + b)$. But if $a \geq 4$ and $b \geq a$, then $2 \lg(a + b) \leq \lg a + 2 \lg b$. The result follows. ◀

As shown in the following lemma, the overall number of data comparisons performed during the computation of the medians of the middles (i.e., step 2 of the Algorithm 1) is within $O(\sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$, where $m_1, \ldots, m_\chi$ are the values of the member of the partition π (see Section 2.2 for the definition of π) and ρ is the number of runs in $\mathcal{M}$.

Consider the instance depicted in Figure 2 for an example illustrating from where the term $\log \binom{\rho}{m_v}$ comes. In this instance, there is a value v that has multiplicity $m_v > 1$ in $\mathcal{M}$ and the rest of the values have multiplicity 1. The elements with value v are present at the end of the last m_v runs and the rest of the runs are formed by only one block. The elements of the i-th run are greater than the elements of the $(i+1)$-th run. During the computation of the medians of the middles, the number of data comparisons that involve elements of value v is within $O(\log \binom{\rho}{m_v})$. The algorithm computes the median μ of the middles and partitions the runs by the value of μ. In the recursive call that involves elements of value v, the number of runs is reduced by half. This is repeated until one occurrence of μ belongs to one of the last m_v runs. The number of data comparisons that involve elements of value v up to this step is within $O(m_v \log \frac{\rho}{m_v}) = O(\log \binom{\rho}{m_v})$, where $\log \frac{\rho}{m_v}$ corresponds to the number of steps where μ does not belong to the last m_v runs. The next recursive call will necessarily choose one element of value v as the median of the middles.

▶ **Lemma 4.** *Let $\mathcal{M}$ be a multiset formed by ρ runs and δ blocks such that these blocks determine a partition π of size χ of the output whose members have values $m_1, \ldots, m_\chi$. Consider the steps that compute the medians of the middles and the steps that find the elements $\max_\ell$ and $\min_r$ in the algorithm* `Quick Synergy Sort`*, the overall number of data comparisons performed during these steps is within $O(\sum_{i=1}^{\chi} \log \binom{\rho}{m_i})$.*

Proof. We prove this lemma by induction over the size χ of π and the number of runs ρ. The number of data comparisons performed by one of these steps is linear in the number of runs in the sub-instance (i.e., ignoring all the empty sets of this sub-instance). Let $\mathcal{T}(\pi, \rho)$ be the overall number of data comparisons performed during the steps 2 and 4 of the algorithm `Quick Synergy Sort`. We prove that $\mathcal{T}(\pi, \rho) \leq \sum_{i=1}^{\chi} m_i \log \frac{\rho}{m_i} - \rho$. Let μ be the first median of the middles computed by the algorithm. Let ℓ and r be the number of runs that are completely to the left and to the right of μ, respectively. Let b be the number of runs that are split in the doubling searches for the value of μ

in all runs. Let π_ℓ and π_r be the partitions determined by the blocks yielded to the left and to the right of μ, respectively. Then, $\mathcal{T}(\pi,\rho) = \mathcal{T}(\pi_\ell,\ell+b) + \mathcal{T}(\pi_r,r+b) + \rho$ because of the two recursive calls and the step that computes μ. By Induction Hypothesis, $\mathcal{T}(\pi_\ell,\ell+b) \leq \sum_{i=1}^{\chi_\ell} m_i \log \frac{\ell+b}{m_i} - \ell - b$ and $\mathcal{T}(\pi_r,r+b) \leq \sum_{i=1}^{\chi_r} m_i \log \frac{r+b}{m_i} - r - b$. Hence, we need to prove that $\ell + r \leq \sum_{i=1}^{\chi_\ell} m_i \log\left(1+\frac{r}{\ell+b}\right) + \sum_{i=1}^{\chi_r} m_i \log\left(1+\frac{\ell}{r+b}\right)$, but this is a consequence of $\sum_{i=1}^{\chi_\ell} m_i \geq \ell+b, \sum_{i=1}^{\chi_r} m_i \geq r+b$ (the number of blocks is greater than or equal to the number of runs); $\ell \leq r+b, r \leq \ell+b$ (at least $\frac{\rho}{2}$ runs are left to the left and to the right of μ); and $\log\left(1+\frac{y}{x}\right)^x \geq y$ for $y \leq x$. ◀

Consider the step that performs doubling searches for the values $\max_\ell$ and $\min_r$ in the run that contains the median μ of the middles, this step results in the finding of the block g that contains μ in at most $4\log|g|$ data comparisons, where $|g|$ is the size of g. Combining Lemma 3 and Lemma 4 yields an upper bound on the number of data comparisons performed by the algorithm `Quick Synergy Sort`:

▶ **Theorem 5.** *Let $\mathcal{M}$ be a multiset of size n formed by ρ runs and δ blocks of sizes $g_1,\ldots,g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1,\ldots,m_\chi$. The algorithm `Quick Synergy Sort` performs within $n + O(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log\binom{\rho}{m_i})$ data comparisons on $\mathcal{M}$. This number of comparisons is optimal in the worst case over multisets of size n formed by ρ runs and δ blocks of sizes $g_1,\ldots,g_\delta$ such that these blocks determine a partition π of size χ of the output whose members have values $m_1,\ldots,m_\chi$.*

We extend these results to take advantage of the global order of the multiset in a way that can be combined with the notion of runs (local order).

2.4 Taking Advantage of Global Order

Given a multiset $\mathcal{M}$, a *pivot position* is a position p in $\mathcal{M}$ such that all elements in previous position are smaller than or equal to all elements at p or in the following positions. In 1962, Iverson [13] described an improved version of `BubbleSort` [16] that identifies such pivot positions (as pair of consecutive elements that the algorithm have placed at their final positions and on which it does not make further comparisons). We show that detecting such positions also yields an improved version of `QuickSort` in general, and of our `QuickSort`-inspired solutions in particular. More formally:

▶ **Definition 6** (Pivot positions)**.** Given a multiset $\mathcal{M} = (x_1,\ldots,x_n)$ of size n, the "pivot positions" are the positions p such that $x_a \leq x_b$ for all a,b such that $a \in [1..p-1]$ and $b \in [p..n]$.

Existing pivot positions in the input order of $\mathcal{M}$ divide the input into subsequences of consecutive elements such that the range of positions of the elements at each subsequence coincide with the range of positions of the same elements in the sorted sequence of $\mathcal{M}$: the more there are of such positions, the more "global" order there is in the input. Detecting such positions takes only a linear number of comparisons by applying the first phase of the algorithm `BubbleSort` [16], which sequentially compares the elements, from left to right in a first phase and then from right to left in a second phase. The positions of the elements that do not interchange their values during both executions are the pivot positions in $\mathcal{M}$.

When there are ϕ such positions, they simply divide the input of size n into $\phi+1$ sub-instances of sizes $n_0,\ldots,n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Each sub-instance I_i for $i \in [0..\phi]$ then

has its own number of runs r_i and alphabet size σ_i, on which the synergistic solutions described in this work can be applied, from mere Sorting (Section 2) to supporting MultiSelection (Section 3) and the more sophisticated Deferred Data Structures (Section 4).

▶ **Corollary 7.** *Let $\mathcal{M}$ be a multiset of size n with ϕ pivot positions. Let $n_0, \ldots, n_\phi$ be integers such that the ϕ pivot positions divide $\mathcal{M}$ into $\phi+1$ sub-instances of sizes $n_0, \ldots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Let ρ_i and δ_i be such that each sub-instance I_i of size n_i is formed by ρ_i runs and δ_i blocks of sizes $g_{i1}, \ldots, g_{i\delta_i}$ such that these blocks determine a partition π_i of size χ_i of the output whose members have values $m_{i1}, \ldots, m_{i\chi_i}$ for $i \in [0..\phi]$. There exists an algorithm that performs within $3n + O(\sum_{i=0}^{\phi} \left\{ \sum_{j=1}^{\delta_i} \log g_{ij} + \sum_{j=1}^{\chi_i} \log \binom{\rho_i}{m_{ij}} \right\})$ data comparisons for sorting $\mathcal{M}$. This number of comparisons is optimal in the worst case over multisets of size n with ϕ pivot positions which divide the multiset into $\phi+1$ sub-instances of sizes $n_0, \ldots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$) and each sub-instance I_i of size n_i is formed by ρ_i runs and δ_i blocks of sizes $g_{i1}, \ldots, g_{i\delta_i}$ such that these blocks determine a partition π_i of size χ_i of the output whose members have values $m_{i1}, \ldots, m_{i\chi_i}$ for $i \in [0..\phi]$.*

Next, we generalize the algorithm `Quick Synergy Sort` to an offline multiselection algorithm that partially sorts a multiset according to the set of `select` queries given as input. This serves as a pedagogical introduction to the online Deferred Data Structures for answering `rank` and `select` queries presented in Section 4.

3 MultiSelection Algorithm

Given a linearly ordered multiset $\mathcal{M}$ and a sequence of ranks $r_1, \ldots, r_q$, a multiselection algorithm must answer the queries `select`(r_1), …, `select`(r_q) in $\mathcal{M}$, hence partially sorting $\mathcal{M}$. We describe a MultiSelection algorithm based on the sorting algorithm `Quick Synergy Sort` introduced in Section 2.3. This algorithm is an intermediate result leading to the Deferred Data Structure described in Section 4.

Given a multiset $\mathcal{M}$ and a set of q `select` queries, the algorithm `Quick Synergy MultiSelection` follows the same first steps as the algorithm `Quick Synergy Sort`. But once it has computed the ranks of all elements in the block that contains the pivot μ, it determines which `select` queries correspond to elements smaller than or equal to $\max_\ell$ and which ones correspond to elements greater than or equal to $\min_r$ (see Algorithm 1 for the definitions of $\max_\ell$ and $\min_r$). It then recurses on both sides.

We extend the notion of blocks to the context of partial sorting. Next, we introduce the definitions of *pivot blocks* and *selection blocks* (see Figure 3 for a graphical representation of these definitions).

▶ **Definition 8** (Pivot Blocks)**.** Given a multiset $\mathcal{M}$ formed by ρ runs and δ blocks. The "*pivot blocks*" are the blocks of $\mathcal{M}$ that contain the pivots and the elements of value equals to the pivots during the steps of the algorithm `Quick Synergy MultiSelection`.

In each run, between the pivot blocks and the insertion ranks of the pivots, there are consecutive blocks that the algorithm `Quick Synergy MultiSelection` has not identified as separated blocks, because no doubling searches occurred inside them.

▶ **Definition 9** (Selection Blocks)**.** Given the i-th run, formed of various blocks, and q `select` queries, the algorithm `Quick Synergy MultiSelection` computes ξ pivots in the process of answering the q queries. During the doubling searches, the algorithm `Quick Synergy MultiSelection` finds the insertion ranks of the ξ pivots inside the i-th run. These positions

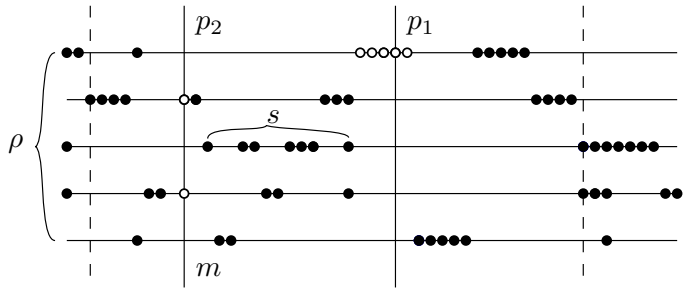

Figure 3 An instance of the MultiSelection problem where the multiset $\mathcal{M}$ is formed by $\rho = 5$ runs. In each run $\mathcal{R}$, the entry $\mathcal{R}[i]$ is represented by a point of x-coordinate $\mathcal{R}[i]$. The dash lines represent the answers of the two `select` queries. The solid vertical lines represent the positions p_1 and p_2 of the first two pivots computed by the `Quick Synergy MultiSelection` algorithm. The pivot blocks corresponding to the pivots p_1 and p_2 are marked by contiguous open disks. The algorithm divides the runs into selection blocks. $s = 7$ is the size of the second selection block, from left to right, into which the third run is divided by the algorithm. $m = 2$ is the number of pivot blocks of size 1 corresponding to the pivot p_2.

determine a partition of size $\xi + 1$ of the i-th run where each element of the partition is formed by consecutive blocks or is empty. We call the elements of this partition "*selection blocks*". The set of all selection blocks contains the set of all pivot blocks.

Using these definitions, we generalize the results proven in Section 2.3 to the more general problem of MultiSelection.

▶ **Theorem 10.** *Given a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks; and q offline* `select` *queries over $\mathcal{M}$ corresponding to elements of* `ranks` $r_1, \ldots, r_q$. *Let ξ be the number of pivots computed by the algorithm* `Quick Synergy MultiSelection` *in the process of answering the q queries. Let $s_1, \ldots, s_\beta$ be the sizes of the β selection blocks determined by these ξ pivots in all runs. Let $m_1, \ldots, m_\lambda$ be the numbers of pivot blocks corresponding to the values of the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \ldots, \rho_\xi$ be the sequence where ρ_i is the number of runs that have elements with values between the pivots i and $i+1$ sorted by* `ranks`, *for $i \in [1..\xi]$. The algorithm* `Quick Synergy MultiSelection` *answers the q* `select` *queries performing within $n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\xi} \rho_i \log \rho_i\right) \subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i\right)$ data comparisons, where $\Delta_i = r_{i+1} - r_i$, $r_0 = 0$ and $r_{q+1} = n$.*

Proof. The pivots computed by the algorithm `Quick Synergy MultiSelection` for answering the queries are a subset of the pivots computed by the algorithm `Quick Synergy Sort` for sorting the whole multiset. Suppose that the selection blocks determined by every two consecutive pivots form a multiset $\mathcal{M}_j$ such that for every pair of selection blocks in $\mathcal{M}_j$ the elements of one are smaller than the elements of the other one. The algorithm `Quick Synergy Sort` would perform within $n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i\right)$ data comparisons in this supposed instance (see the proof of Lemmas 3 and 4 analyzing the algorithm `Quick Synergy Sort` for details). The number of comparisons needed to sort the multisets $\mathcal{M}_j$ is within $\Theta(\sum_{i=0}^{\xi} \rho_i \log \rho_i)$. The result follows. ◀

The process of detecting the ϕ pre-existing pivot positions seen in Section 2.4 can be applied as the first step of the multiselection algorithm. The ϕ pivot positions divide the input of size n into $\phi + 1$ sub-instances of sizes $n_0, \ldots, n_\phi$. For each sub-instance I_i for $i \in [0..\phi]$, the multiselection algorithm determines which `select` queries correspond to I_i

and applies then the steps of the algorithm `Quick Synergy MultiSelection` inside I_i in order to answer these queries.

The `Quick Synergy MultiSelection` algorithm takes advantage of the number and sizes of the runs (i.e., the local input order), the number and positions of the pre-existing pivot positions (i.e., the global order), the multiplicities of the elements in the multiset (i.e., the input structure) and the differences between consecutive `select` queries in sorted order (i.e., the query structure).

In the result above, the queries are given all at the same time (i.e., offline). In the context where they arrive one at the time (i.e., online), we define a Deferred Data Structure for answering online `rank` and `select` queries, inspired by the algorithm `Quick Synergy MultiSelection`.

4 `Rank` and `Select` Deferred Data Structures

We describe the Full-Synergistic Deferred Data Structure that answers a set of `rank` and `select` queries, arriving one at the time, over a multiset $\mathcal{M}$, progressively sorting $\mathcal{M}$. This deferred data structure is based in the `Quick Synergy MultiSelection` algorithm described in the previous section. This data structure takes advantage of the order (local and global) and structure in the input, and of the order and structure in the queries.

By "query order", we mean to consider the "distances" between consecutive queries. To take advantage of the query order, we introduce a data structure that finds the nearest pivots to the left and to the right of a position $p \in [1..n]$, while taking advantage of the *distance* between the position of the last computed pivot and p, as measured by the number of computed pivot blocks between the two positions. For that we use a *finger search tree* [8] maintaining *fingers* (i.e., pointers) to elements in the search tree and supporting efficient updates and searches in the vicinity of those. Brodal [8] described an implementation of finger search trees that searches for an element x, starting the search at the element given by the finger f in time within $O(\log d)$, where d is the distance between x and f in the set (i.e, the difference between $\texttt{rank}(x)$ and $\texttt{rank}(f)$ in the set). This operation returns a finger to x if x is contained in the set, otherwise a finger to the largest element smaller than x in the set. This implementation supports the insertion of an element x immediately to the left or to the right of a finger in worst-case constant time.

Given a multiset $\mathcal{M}$ of size n, the Full-Synergistic Deferred Data Structure includes a finger search tree $\mathcal{F}_{\texttt{select}}$, in which it marks the elements in $\mathcal{M}$ that have been computed as pivots when it answers the online queries. For each pivot p in $\mathcal{F}_{\texttt{select}}$, the data structure stores pointers to the insertion ranks of p in each run, to the beginning and to the end of the block g to which p belongs, and to the position of p inside g. This finger search tree is also used to find the two successive pivots between which the query fits.

Once a pivot block g is computed, every element in g is a valid pivot for the rest of the elements in $\mathcal{M}$. In order to capture this idea, we modify the finger search tree $\mathcal{F}_{\texttt{select}}$ so that it contains the pivot blocks (i.e., a sequence of consecutive values) instead of singleton pivots. This modification allows the Full-Synergistic Deferred Data Structure to answer `select` queries, taking advantage of the structure and order in the queries and of the structure and order in the input. But in order to answer `rank` queries taking advantage of the features in the queries and the input, the data structure needs another finger search tree $\mathcal{F}_{\texttt{rank}}$. In $\mathcal{F}_{\texttt{rank}}$ the data structure stores, for each block g identified, the value of one of the elements in g, and pointers in $\mathcal{M}$ to the beginning and to the end of g, and in each run to the position where the elements of g partition the run.

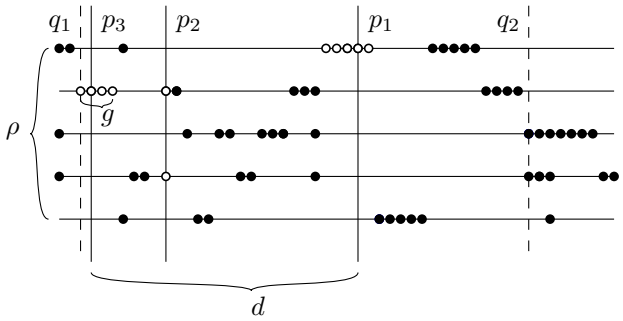

Figure 4 The state of the Full-Synergistic Deferred Data Structure on an instance where the multiset $\mathcal{M}$ is formed by $\rho = 5$ runs. In each run, the entry $\mathcal{M}[i]$ is represented by a point of x-coordinate $\mathcal{M}[i]$. The dash lines represent the positions q_1 and q_2 of the answers of the first two queries. The solid vertical lines represent the positions p_1, p_2 and p_3 of the first three pivots computed by the Full-Synergistic Deferred Data Structure. The pivot blocks corresponding to the pivots p_1, p_2 and p_3 are marked by contiguous open disks. $d = 4$ is the distance (i.e., the number of computed pivot blocks) between the queries q_1 and q_2. If q_1 is a `rank` query, then $g = 4$ is the size of the identified block that contains the answer of the query q_1.

▶ **Theorem 11.** *Consider a multiset $\mathcal{M}$ of size n formed by ρ runs and δ blocks. Let γ and $r_1, \ldots, r_q$ be the number of pivot blocks computed by the* Full-Synergistic Deferred Data Structure *in the process of answering q online* **`rank`** *and* **`select`** *queries over $\mathcal{M}$, and the* **`ranks`** *of the elements corresponding to these queries, respectively. Let $s_1, \ldots, s_\beta$ be the sizes of the β selection blocks determined by the pivots in the γ blocks in all runs. Let $m_1, \ldots, m_\lambda$ be the numbers of pivot blocks corresponding to the values of the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \ldots, \rho_\gamma$ be the sequence where ρ_i is the number of runs that have elements with values between the elements in the blocks i and $i+1$ sorted by* **`ranks`**, *for $i \in [1..\gamma]$. Let $d_1, \ldots, d_{q-1}$ be the sequence where d_j is the number of computed pivot blocks between the block that answers the $(j-1)$-th query and the one that answers the j-th query before starting the steps to answer the j-th query, for $j \in [2..q]$. Let u and $g_1, \ldots, g_u$ be the number of* **`rank`** *queries and the sizes of the computed and searched pivot blocks in the process of answering the u* **`rank`** *queries, respectively. The* Full-Synergistic Deferred Data Structure *answers the q online queries by performing within $n + O(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\gamma} \rho_i \log \rho_i + \sum_{i=1}^{q-1} \log d_i + \sum_{i=1}^{u} \log g_i) \subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i + q \log n\right)$ data comparisons, where $r_0 = 0$, $r_{q+1} = n$, and $\Delta_i = r_{i+1} - r_i$, for all $i \in [1..n]$.*

Proof. The algorithm answers a new `select`(i) query by searching in $\mathcal{F}_{\texttt{select}}$ for the nearest pivots to the left and right of the query position i. If i is contained in an element of $\mathcal{F}_{\texttt{select}}$, then the block g that contains the element in the position i has already been computed. If i is not contained in an element of $\mathcal{F}_{\texttt{select}}$, then the returned finger f points the nearest block b to the left of i. The block that follows f in $\mathcal{F}_{\texttt{select}}$ is the nearest block to the right of i. It then applies the same steps as the algorithm `Quick Synergy MultiSelection` in order to answer the query. Given f, the algorithm inserts in $\mathcal{F}_{\texttt{select}}$ each pivot block computed in the process of answering the query in constant time, and stores the respective pointers to positions in $\mathcal{M}$. In $\mathcal{F}_{\texttt{rank}}$ the algorithm searches for the value of one of the elements in b. Once the algorithm obtains the finger returned by this search, the algorithm inserts in $\mathcal{F}_{\texttt{rank}}$ the value of one of the elements of each pivot block in constant time, and stores the respective pointers to positions in $\mathcal{M}$ (see Figure 4 for a graphical representation of some of the parameters used in the analysis).

The algorithm answers a new `rank`(x) query by finding the *selection block* s_j in the j-th run such that x is between the smallest and the greatest value of s_j for all $j \in [1..\rho]$. For that the algorithm searches for the value x in $\mathcal{F}_{\texttt{rank}}$. The number of data comparisons performed by this searching process is within $O(\log d)$, where d is the number of blocks in $\mathcal{F}_{\texttt{rank}}$ between the last inserted or searched block and the returned finger f. Given the finger f, there are three possibilities for the `rank` r of x: (i) r is between the `ranks` of the elements at the beginning and at the end of the block pointed by f, (ii) r is between the `ranks` of the elements at the beginning and at the end of the block pointed by the finger following f, or (iii) r is between the `ranks` of the elements in the selection blocks determined by f and the finger following f. In the cases (i) and (ii), a binary search inside the block yields the answer of the query. In the case (iii), the algorithm applies the same steps as the algorithm `Quick Synergy MultiSelection` in order to compute the median μ of the middles, and partitions the selection blocks by μ. The algorithm then decides to which side x belongs. Similar to the algorithm for answering a `select` query, the data structure inserts in $\mathcal{F}_{\texttt{select}}$ every block computed in the process of answering the `rank` query. ◀

The process of detecting the ϕ pivot positions seen in Section 2.4 allows the Full-Synergistic Deferred Data Structure to insert these pivots in $\mathcal{F}_{\texttt{select}}$ and $\mathcal{F}_{\texttt{rank}}$. For each pivot position p in $\mathcal{F}_{\texttt{select}}$ and $\mathcal{F}_{\texttt{rank}}$, the data structure stores pointers to the end of the runs detected on the left of p; to the beginning of the runs detected on the right of p; and to the position of p in the multiset. This concludes the description of our synergistic results. In the next section, we discuss how these results relate to various past results and future work.

5 Discussion

Kaligosi et al.'s multiselection algorithm [14] and Barbay et al's deferred data structure [2] use the very same concept of *runs* as the one described in this work. The difference is, we describe algorithms that *detect* the existing runs in the input in order to take advantage of them, while the algorithms described by those previous works do not take into consideration any pre-existing runs in the input, and rather build and maintain such runs as a strategy to minimize the number of comparisons performed while partially sorting the multiset. We leave the combination of both approaches as a topic for future work, which could probably shave a constant factor off the number of comparisons performed by the Sorting and MultiSelection algorithms and by the Deferred Data Structures supporting `rank` and `select` queries on Multisets.

Barbay and Navarro [3] described how any Sorting algorithm taking advantage of specificities in the input, directly implies a compressed encoding for permutations. By using the similarity of the execution tree of the algorithm `MergeSort` with the `Wavelet Tree` data structure, they described a compressed data structure for permutations taking advantage of the local order, i.e., using space proportional to $\mathcal{H}(r_1, \ldots, r_\rho)$ and supporting `direct access` (i.e. $\pi()$) and `inverse access` (i.e. $\pi^{-1}()$) in worst time within $O(1 + \lg \rho)$ and average time within $O(1 + \mathcal{H}(r_1, \ldots, r_\rho))$. We leave as future work the extension of our work into a compressed data structure for multisets taking advantage of both its structure and (local and global) order.

Another perspective is to generalize the synergistic results to related problems in computational geometry: Karp et al. [15] defined the first deferred data structure not only to support `rank` and `select` queries on multisets, but also to support online queries in a deferred way on Point Membership in a Convex Hull in two dimensions and online Dominance

queries on sets of multi-dimensional vectors. Preliminary results [4] show that one can refine the results from Karp et al. [15] to take advantage of the blocks between queries (i.e., the structure in the queries) as Barbay et al. [2] did for multisets; but also of the relative position of the points (i.e., the structure in the input) as Afshani et al. [1] did for Convex Hulls and Maxima; of the order in the points (i.e., the order in the input), as computing the convex hull in two dimensions takes linear time if the points are sorted; and potentially of the order in the queries.

References

1. Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. *J. ACM*, 64(1):3:1–3:38, March 2017. doi:10.1145/3046673.
2. Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jonathan Sorenson. Near-optimal online multiselection in internal and external memory. *J. Discrete Algorithms*, 36:3–17, 2016. doi:10.1016/j.jda.2015.11.001.
3. Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013. doi:10.1016/j.tcs.2013.10.019.
4. Jérémy Barbay and Carlos Ochoa. Synergistic computation of planar maxima and convex hull, 2017. arXiv:1702.08545.
5. Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti. Synergistic sorting and deferred data structures on multisets, August 2016. arXiv:1608.06666.
6. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.
7. Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.
8. Gerth S. Brodal. Finger search trees with constant insertion time. In Howard J. Karloff, editor, *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 540–549. ACM/SIAM, 1998. URL: http://dl.acm.org/citation.cfm?id=314613.314842.
9. Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In David B. Shmoys, editor, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752. ACM/SIAM, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338634.
10. David P. Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *J. ACM*, 28(3):454–461, 1981. doi:10.1145/322261.322264.
11. Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992. doi:10.1145/146370.146381.
12. Charles A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, 1961. doi:10.1145/366622.366647.
13. Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. URL: http://www.softwarepreservation.org/projects/apl/Books/APROGRAMMING%20LANGUAGE.
14. Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming (ICALP 2005)*, volume 3580 of *LNCS*, pages 103–114. Springer, 2005. doi:10.1007/11523468_9.
15. Richard Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM J. Comput.*, 17(5):883–902, 1988. doi:10.1137/0217055.

16 Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

17 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

18 Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Aust. Comput. J.*, 24(2):70–77, 1992. URL: http://50years.acs.org.au/__data/assets/pdf_file/0017/111464/ACJ-V24-N02-199205.pdf.

19 J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976. doi:10.1137/0205001.

20 Tadao Takaoka. Partial solution and entropy. In Rastislav Královic and Damian Niwiński, editors, *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS 2009)*, volume 5734 of *LNCS*, pages 700–711. Springer, 2009. doi:10.1007/978-3-642-03816-7_59.