UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SYNERGISTIC (ANALYSIS OF) ALGORITHMS AND DATA STRUCTURES

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCIÓN COMPUTACIÓN

CARLOS ERNESTO OCHOA MÉNDEZ

PROFESOR GUÍA:
JÉRÉMY BARBAY

MIEMBROS DE LA COMISIÓN:
DIEGO ARROYUELO BILLIARDI
TRAVIS GAGIE
RAJEEV RAMAN

Este trabajo fue financiado por CONICYT-PCHA/Doctorado Nacional/2013-63130161, y los proyectos CONICYT Fondecyt/Regular nos 1120054 y 1170366

SANTIAGO DE CHILE
2019

Resumen

Los refinamientos actuales del análisis del peor caso sobre instancias con tamaño de entrada fijo consideran el orden de la entrada (por ejemplo, las subsecuencias ordenadas en una secuencia de números y las cadenas poligonales simples en las que puede dividirse una secuencia de puntos) o la estructura de la entrada (por ejemplo, la multiplicidad de los elementos en un multiconjunto y las posiciones relativas entre un conjunto de puntos en el plano), pero nunca, hasta donde sabemos, ambos al mismo tiempo. En esta tesis se proponen nuevas técnicas que combinan soluciones que se aprovechan del orden y la estructura de la entrada en una sola solución sinérgica para ordenar multiconjuntos, y para calcular la eficiencia de Pareto y la envoltura convexa de un conjunto de puntos en el plano. Estas soluciones sinérgicas se aprovechan del orden y la estructura de la entrada de tal forma que asintóticamente superan cualquier solución comparable que se aproveche solo de una de estas características. Como resultados intermedios, se describen y analizan varios algoritmos de mezcla: un algoritmo para mezclar secuencias ordenadas que es óptimo para cada instancia del problema; el primer algoritmo adaptativo para mezclar eficiencias de Pareto; y un algoritmo adaptativo para mezclar envolturas convexas en el plano. Estos tres algoritmos se basan en un paradigma donde las estructuras se dividen antes de ser mezcladas. Este paradigma es conveniente para extenderlo al contexto donde se responden consultas.

Karp et al. (1998) describieron estructuras de datos diferidas como estructuras "perezosas" que procesan la entrada gradualmente a medida que responden consultas sobre los datos, trabajando la menor cantidad posible en el peor caso sobre instancias de tamaño n fijo y número de consultas q fijas. En esta tesis se desarrollan nuevas técnicas para refinar aún más estos resultados y aprovechar al mismo tiempo el orden y la estructura de la entrada y el orden y la estructura de la secuencia de consultas en tres problemas distintos: calcular el rango y la posición de un elemento en un multiconjunto, determinar si un punto está dominado por la eficiencia de Pareto de un conjunto de puntos en el plano y determinar si un punto pertenece a la envoltura convexa de un conjunto de puntos en el plano. Las estructuras de datos diferidas que se obtienen superan todas las soluciones previas que solo se aprovechan de un subconjunto de estas características.

Como una extensión natural a los resultados sinérgicos obtenidos en este trabajo para ordenar un multiconjunto, se describen estructuras de datos comprimidas que se aprovechan del orden y la estructura de la entrada para representar un multiconjunto, mientras se responden consultas del rango y la posición de elementos en el multiconjunto.

Abstract

Refinements of the worst case complexity over instances of fixed input size consider the input order (e.g., the runs in an array of numbers, the simple polygonal chains into which a sequence of points can be partitioned) or the input structure (e.g., the multiplicities of the elements in a multiset, the relative positions between a set of planar points), but rarely (never until now, as far as we know) both at the same time. We propose new techniques that combine solutions taking advantage both of the input order and the input structure into a single synergistic solution for sorting multisets, and for computing the maxima set and the convex hull in the plane. Such synergistic solutions take advantage both of the input order and the input structure, so as to asymptotically outperform any comparable solution which takes advantage only of one of those features. As intermediate results, we describe and analyze an instance optimal algorithm for merging sorted sets, the first adaptive algorithm for merging maxima sets, and an adaptive algorithm for merging convex hulls in the plane. Those three new merging solutions are based on a splitting paradigm, which is suitable to extend to supporting online queries.

Karp et al. (1988) described deferred data structures as "lazy" data structures which gradually preprocess the input to support online queries, with the minimum amount of work in the worst case over instances of fixed size n and fixed number of queries q (i.e., the query size). We develop new techniques in order to further refine this approach and take advantage all at once of the input order and the input structure, and of the order and structure in the sequence of queries for answering rank and select queries on a multiset, domination queries on a set of planar points, and point membership queries on the convex hull of a set of planar points. The resulting deferred data structures outperform all previous solutions that take advantage of only a subset of those features in the decision tree model and the algebraic decision tree model.

As a natural extension of the synergistic sorting results, we describe compressed data structures to represent a multiset taking advantage both of the input order and the input structure, while supporting the operators rank and select on the multiset.

Contents

List of Figures

Chapter 1

Introduction

The computational complexity of most problems is often studied in the worst case over instances of fixed size n, for n asymptotically tending to infinity. This approach has been refined in many ways, but most notably for NP-hard problems under the term "Parameterized Complexity" [33], for polynomial problems under the term "Adaptive (Analysis of) Algorithms" [28, 54], and more simply for data encodings under the term "Data Compression" [7], for a wide range of problems and data types. Such a variety of results has motivated various classifications: in the context of NP-hard problems with a theory of Fixed Parameter Tractability [33], and in the context of sorting in the decision tree model with a theory of reduction between measures of disorder in permutations and multisets [59].

In the context of the adaptive (analysis of) algorithms, we introduce an additional perspective from which to classify algorithms and data structures: those taking advantage of some measures of the input order (e.g., disorder measures for sorting permutations [54, 28], decomposition into simple polygonal chains for the computation of the convex hull [50]) (Section 1.1), those taking advantage of the input structure (e.g., the multiplicities of the elements for sorting multisets [56], output sensitive algorithms [48] and input-order oblivious instance optimality [1], both for the computation of the maxima set and the convex hull) (Section 1.2), and those taking advantage both of some measures of the input order and the input structure in a synergistic way (Section 1.3), a class which to our knowledge was not considered before. For each class, we exemplify some algorithms in it.

In the context of the competitive analysis of data structures for supporting online queries, we introduce an additional perspective related to the queries, from which to classify data structures: those taking advantage of some measures of the query order (e.g., measures that capture how close the positions of two consecutive queries are [16, 37]), those taking advantage of the query structure (e.g., differences between the ranks of consecutive select queries in sorted order on a multiset [27, 44, 6]), and those taking advantage both of some measures of the query order and the query structure in a synergistic way (Chapter 5).

1.1 Input Order

The input of every algorithm can be represented as a sequence of elements (e.g., streaming algorithms [4]). By input order, we mean to consider features of the input that depend on the order of the sequence that represents the input. Concerning the sorting of multisets, we review algorithms taking advantage of the order in the sequence of numbers that represents the multiset. Concerning the computation of the convex hull in the plane, we review algorithms that take advantage of the order into which the points are listed in a polygonal chain. As far as we know, the notion of input order has never been considered for the computation of the maxima set of planar points until now. In Section 4.2.1, we describe an algorithm that computes the maxima set of planar points taking advantage of some measure of the input order.

Any array A representing a multiset lists its elements in some order, denoted by a tuple (e.g., $A = (2, 3, 1, 3, 7, 8, 9, 4, 5, 6)$). Maximal sorted subblocks in A are called *runs* [49] (e.g., $\{(2, 3),\ (1, 3, 7, 8, 9),\ (4, 5, 6)\}$ in A). As early as 1973, Knuth [49] described a variant of the algorithm `MergeSort` [49, Section 5.2.4] using a preprocessing step taking linear time to detect runs in the array A, which he named `Natural MergeSort`. Mannila [52] refined the analysis of the `Natural MergeSort` algorithm to yield a time complexity for sorting an array A of size n in time within $O(n(1 + \log \rho)) \subseteq O(n \log n)$, where ρ is the number of runs in A. Takaoka [63] described a new sorting algorithm that takes advantage of the distribution of the sizes of the runs in A[1]. Takaoka measured the difficulty of the instance in terms of the distribution of the sizes of the runs in A by the *entropy function* $\mathcal{H}(r_1, \ldots, r_\rho) = \sum_{i=1}^{\rho} \frac{r_i}{n} \log \frac{n}{r_i}$, where ρ is the number of runs in A and $r_1, \ldots, r_\rho$ are the sizes of the ρ runs (such that $\sum_{i=1}^{\rho} r_i = n$), respectively. The time complexity of this algorithm is within $O(n(1 + \mathcal{H}(r_1, \ldots, r_\rho))) \subseteq O(n(1+ \log \rho)) \subseteq O(n \log n)$. Takaoka proved that this time complexity is worst-case optimal over instances of size n formed by ρ runs of sizes $r_1, \ldots, r_\rho$, respectively, in the decision tree model. All these algorithms take advantage of the order of the elements in the tuple A that represents the multiset (i.e., the input order).

One of the ways to take advantage of the input order of a set of points in the plane is captured by the concept of polygonal chain. A *polygonal chain* is specified by a sequence of planar points, and consists of the line segments connecting the pairs of consecutive points. Any set of points in the plane can be represented by a polygonal chain, which is determined by the order into which the points are listed. A polygonal chain is simple if it does not have a self-intersection. In 2002, Levcopoulos et al. [50] described an algorithm that computes the convex hull of planar points by decomposing the set of points into simple polygonal chains. They proved that the time complexity of this algorithm is within $O(n(1+ \log \kappa)) \subseteq O(n \log n)$, where κ is the minimum number of simple polygonal chains into which the set of n points can be partitioned. They showed that this time complexity is worst-case optimal over instances of n points that can be partitioned into κ simple polygonal chains, in the algebraic decision tree model. Note that κ depends only on the order of the points: by reordering them, one can always reduce κ to one, or increase it to within $\Theta(n)$.

Similarly, in 2011, Ahn and Okamoto [2] described various algorithms reordering the points in a certificate of the convex hull, which can be checked in linear time, thus extending to the computation of the convex hull various measures of disorder studied in the context of sorting permutations [54].

1.2 Input Structure

By input structure, we mean to consider features of the input that depend on the structure of the instance and that are independent from the order of the input. Concerning the sorting

[1]Barbay and Navarro [7] rediscovered this principle later and extended it to compressed data structures for permutations.

of multisets, we review an algorithm taking advantage of the multiplicities of the elements in the multiset. Concerning the computation of the maxima set and the convex hull in the plane, we review algorithms taking advantage of the relative positions of the points.

Consider a multiset M of size n (e.g., $M = \{4, 4, 3, 3, 4, 5, 6, 7, 1, 2\}$ of size $n = 10$). The *multiplicity* of an element x of M is the number m_x of occurrences of x in M (e.g., $m_3 = 2$). The distribution of the multiplicities of the elements in M is the set of pairs (x, m_x) (e.g., $\{(1,1), (2,1), (3,2), (4,3), (5,1), (6,1), (7,1)\}$ in M). As early as 1976, Munro and Spira [56] described a variant of the algorithm `MergeSort` [49, Section 5.2.4] using counters, which takes advantage of the distribution of the multiplicities of the elements in M when sorting it. The time complexity of the algorithm is within $O(n(1 + \mathcal{H}(m_1, \ldots, m_\sigma))) \subseteq O(n(1+\log\sigma)) \subseteq O(n \log n)$, where σ is the number of distinct elements in M and $m_1, \ldots, m_\sigma$ are the multiplicities of the σ distinct elements (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. They proved that this time complexity is worst-case optimal over instances of size n with σ distinct elements of multiplicities $m_1, \ldots, m_\sigma$, respectively, in the decision tree model. The distribution of the multiplicities of the elements in M (i.e., the input structure) is independent of the order into which the elements are listed.

Similar results exist in two dimensions: in 1985, Kirkpatrick and Seidel [47] described an algorithm that computes the maxima set of a set P of planar points[2]. They showed that the time complexity of this algorithm is within $O(n(1 + \log h)) \subseteq O(n \log n)$, where h is the number of points in the maxima set (i.e., the output size), and n is the number of points of P (i.e., the input size). They showed that this time complexity is worst-case optimal over instances of input size n and output size h, in the decision tree model. This algorithm is *output sensitive*, in the sense that the time complexity depends on the size of the output, and can be classified as adaptive to the input structure, as the positions of the points clearly determine the output (and its size).

Considering the problem of computing the convex hull of planar points, in 1970, Chand and Kapur [18] proposed the `Gift Wrapping` algorithm (rediscovered by Jarvis [42] in 1973 under the name of `Jarvis's walk`). It was presented as running in time within $O(n^2)$, where n is the number of points (i.e., the input size), but is adaptive to the number h of vertices of the convex hull (i.e., the output size), with a running time within $O(nh) \subseteq O(n^2)$. In 1986, Kirkpatrick and Seidel [48], generalizing their previous results on the output sensitive computation of maxima sets [47] (described above), described an algorithm that computes the convex hull in time within $O(n \log h) \subseteq O(n \log n)$. They proved that this time complexity is worst-case optimal over instances of input size n and output size at most h, in the algebraic decision tree model. The algorithm relies on a variation of the divide-and-conquer paradigm, which they call the "marriage-before-conquest" principle.

Both results described by Kirkpatrick and Seidel [47, 48] for computing the maxima set and the convex hull of planar points, respectively, were further improved when Afshani et al. [1] observed that minor variants of such algorithms take optimal advantage of the positions of the points. Afshani et al. [1] proved that such variants are instance optimal[3] among algorithms ignoring the order of the input in the decision tree model and the algebraic decision tree model where the tests involve only multilinear functions with a constant number of arguments, respectively. These results take advantage of the relative positions of the points

[2]They described algorithms for computing the maxima set of points in any dimension. We describe only the version in the plane.

[3]For any instance I and a class $\mathcal{A}$ of algorithms, instance optimal algorithms performs within a constant factor of the performance of any algorithm $A \in \mathcal{A}$ on I [29].

(i.e., the input structure), which are independent of the order into which the points are listed.

In the following section, we consider the design of algorithms taking advantage both of some measures of the input order and the input structure, which is the main contribution of this work.

1.3 Synergistic Solutions

In the context of sorting multisets and computing the convex hull of planar points, any dovetailing combination[4] of the algorithms described by Munro and Spira [56] and Takaoka [63], and of the algorithms described by Kirkpatrick and Seidel [48] and Levcopoulos et al. [50], respectively, takes advantage both of some measures of the input order and the input structure. But such solutions are inefficient: many operations are repeated, and some opportunities to quickly solve the instance are lost due to the lack of communication between the two parallel branches of the dovetailing combination of the algorithms.

HYPOTHESIS: It is possible to improve upon both categories of techniques with a single algorithm which optimally takes advantage both of the input order and the input structure, in a synergistic way.

We confirm the hypothesis by describing synergistic algorithms for sorting multisets, and for computing the maxima set and the convex hull, the two latter in the plane. These synergistic algorithms take advantage both of some measures of the input order and the input structure, synergistically, in the sense that they never perform asymptotically worse than previous algorithms [47, 48, 50, 56, 63, 7]. Moreover, on some large classes of instances, where they can take advantage both of some measures of the input order and the input structure, they asymptotically outperform previous solutions by more than a constant factor (see Chapter 4). As far as we know, every known approach described so far takes advantage of either the input order or the input structure, but not of both.

In the context of supporting offline and online queries, we obtain algorithms and data structures for answering rank and select queries on a multiset, domination queries on a set of planar points, and point membership queries on the convex hull of planar points. These algorithms and data structures take advantage of some measures of the input order and the input structure as well as notions of difficulty in the queries, such as the query structure and some measures of the query order, in a synergistic way (see Chapter 5). In the following section, we outline the integrality of our results.

1.4 Thesis Structure and Contributions

After revisiting previous concepts and results on sorting multiset, computing maxima sets and convex hulls in the plane, and on related problems, we present new synergistic solutions for sorting multisets, for computing maxima sets and convex hulls in the plane. Our approach to present synergistic algorithms for those three different problems follows the same pattern:

[4]A dovetailing combination of k algorithms executes the k algorithms in parallel and stops as soon as one of the algorithms finishes.

1. Define techniques that take advantage of some measures of the input order by partitioning the input sequence of elements (i.e., numbers for sorting multisets, points for computing maxima sets and convex hulls) into "easy" subsequences, such that in each "easy" subsequence, the computation can be done in time linear in the number of elements in the subsequence (Chapter 4).

2. Merge the resulting structures (i.e., sorted sets for sorting multisets, maxima sets for the computation of the maxima set, and convex hulls for the computation of the convex hull) by taking advantage of the input structure (Chapter 3).

We present first the merging algorithms in Chapter 3, which are interesting on their own right [24, 5], and are the most technical part of our synergistic approach. We consider the extension of the synergistic solutions to supporting queries, and the extension of some of them to compressed data structures in Chapter 5. In the following, we give a detailed description of the content of each chapter.

Chapter 2: *Background*, we review previous concepts and results on sorting multisets, and on computing the maxima set and the convex hull of planar points, taking advantage of either the input structure or some measures of the input order, and on related problems: searching, multiselection, deferred data structures, and compressed data structures. We give a brief description of the models of computation used in the lower bounds reviewed and obtained in this thesis.

Chapter 3: *Adaptive Merging*, we describe adaptive (analysis of) algorithms for merging sorted sets, merging maxima sets, and merging convex hulls, all which solve the second point of our approach.

We slightly correct the analysis of the algorithm `Sorted Set Union` for merging sorted sets described by Demaine et al. [24] (Section 3.1.1). We also describe a new instance optimal algorithm, which we name `Quick Set Union`, for merging sorted sets (Section 3.1.2), using a distinct paradigm to the one used by the algorithm `Sorted Set Union` (splitting vs merging), more suitable to extend to solutions for the multiselection problem and deferred data structures (described in Chapter 5).

We generalize the `Quick Set Union` algorithm and the corresponding analysis from the merging of sorted sets to the merging of maxima sets (Section 3.2) and of convex hulls (Section 3.3). As far as we know, this is the first adaptive algorithms for merging maxima sets (Section 3.2.1). In particular, we analyze the algorithm for merging maxima sets and prove its optimality in the worst case over instances of measure δ fixed, where δ depends not only on the number of maxima sets and its sizes, but also on the relative positions between the maxima sets. We extend the difficulty measure in the analysis of the algorithm for merging maxima sets to the analysis of the algorithm for merging convex hulls (Section 3.3.2). Even though we cannot prove that the algorithm for merging convex hulls is optimal in terms of this refined measure, our analysis suffices to prove that the synergistic algorithm based on it never performs asymptotically worse than the algorithms described by Kirkpatrick and Seidel [48] and Levcopoulos et al. [50], and, on some large classes of instances, asymptotically outperforms both by more than a constant factor.

We summarize the main results described in this chapter in the following table:

Problems	Previous work	This thesis
Sorted Sets	$\Theta\left(\sum_{i=1}^{\delta}\log g_i + \sum_{i=1}^{\chi}\log\binom{\rho}{m_i}\right)$ ([24])	$\Theta\left(\sum_{i=1}^{\delta}\log g_i + \sum_{i=1}^{\chi}\log\binom{\rho}{m_i}\right)$ (Thm. 1)
Maxima Sets	–	$\Theta\left(\sum_{j=1}^{\beta}\log s_j + \sum_{i=1}^{\delta}\log\binom{\zeta}{m_i}\right)$ (Thm. 2)
Convex Hulls	–	$O\left(\kappa\delta\log\frac{h}{\delta} + \delta\sum_{i=1}^{\kappa}\log\frac{r_i}{\delta}\right)$ (Thm. 4)

Table 1.1: Results for merging sorted sets, merging maxima sets, and merging convex hulls.

Chapter 4: *Synergistic Computation Analysis*, we develop new techniques to take advantage of some measures of the input order when sorting multisets (Section 4.1.1), and when computing the maxima set (Section 4.2.1) and the convex hull (Section 4.3.1) of planar points. Such techniques, in combination with the merging algorithms described in Chapter 3 yield synergistic algorithms for those three problems.

Concerning the sorting of multisets, we define a simple yet new[5] notion of "global" input order, formed by the number of pre-existing pivot positions in the input (i.e., positions p such that all the elements to the left of p are smaller than or equal to all the elements to the right of p, for example, $(3, 2, 1, 6, 5, 4)$ has one pre-existing pivot position in the middle) (Section 4.1.1). We describe examples of the difficulty of combining both measures of the input order and the input structure into a single synergistic algorithmic technique for sorting multisets (Section 4.1.2). We describe how to measure the interaction of some measures of the input order (i.e., runs and pivot positions) with the input structure (i.e., the distribution of the multiplicities of the distinct elements), and two new synergistic sorting algorithms based on distinct paradigms (merging vs splitting), which take advantage both of some measures of the input order and the input structure to sort the multiset in less time than traditional solutions based on one of those, at most [56, 63, 7] (Section 4.1.2).

Concerning the computation of the maxima set of planar points, we define a new algorithm that takes advantage of some measure of the input order (Section 4.2.1). Such an algorithm partitions the input polygonal chain, determined by the order in which the points are listed, into smooth subsequences (see Definition 14 page 50), such that each of the maxima sets of the smooth subsequences can be computed in time linear in the size of the subsequence. As far as we know, this is the first algorithm that computes the maxima set of planar points taking advantage of some measure of the input order.

Concerning the computation of the convex hull of a set of planar points, we improve previous techniques to analyze the time complexity of algorithms computing the convex hull in function of some measure of the input order (i.e., considering not only the minimal number of simple chains into which the input sequence of points can be partitioned, but also the distribution of their sizes). We present an algorithm that partitions a sequence S of n points into simple polygonal chains, which is faster than the one described by Levcopoulos et al. [50] (Section 4.3.1). The new partitioning algorithm runs in time within $O(n)$, while the partitioning algorithm described by Levcopoulos et al. [50] runs in time within $O(n(1+\log\kappa))$, where κ is the minimum number of simple polygonal chains into which S can be partitioned. We refine Levcopoulos et al.'s measure of difficulty and analysis of the computation of the convex hull of such chains (Section 4.3.1).

The combination of the algorithms for merging maxima sets and merging convex hulls (described in Chapter 3) with the input order results yields synergistic algorithms that com-

[5]The notion of "global" input order was not mentioned in previous surveys [28, 54] nor extensions [7].

pute the maxima set (Section 4.2.2) and the convex hull (Section 4.3.2), both in the plane. We prove that on large classes of instances, such synergistic algorithms asymptotically outperform the best previous solutions: both those that take optimal advantage of the positions of the points [47, 48, 1] (i.e., the input structure), and those that take advantage of the simple polygonal chains into which the sequence of points can be partitioned [50] (i.e., some form of input order).

We summarize the main results described in this chapter in the following table:

Problems	Previous work	This thesis
Sorting	$O(n(1+\mathcal{H}(m_1,\dots,m_\sigma))$ ([56]) $O(n(1+\mathcal{H}(r_1,\dots,r_\rho)))$ ([63])	$n+O\left(\sum_{i=1}^{\delta}\log g_i+\sum_{i=1}^{\chi}\log\binom{\rho}{m_i}\right)$ (Thm. 5)
Maxima Set	$O(n(1+\mathcal{H}(n_1,\dots,n_h)))$ ([1])	$2n+O\left(\sum_{j=1}^{\beta}\log s_j+\sum_{i=1}^{\delta}\log\binom{\zeta}{m_i}\right)$ (Thm. 6)
Convex Hull	$O(n(1+\mathcal{H}(n_1,\dots,n_h)))$ ([1])	$O\left(n+\sum_{j=1}^{\delta}w_j\log s_j+\sum_{i=1}^{\delta}\log\binom{\kappa}{m_i}\right)$ (Thm. 8)

Table 1.2: Results for sorting, and computing maxima sets and convex hulls.

Chapter 5: *Synergistic Data Structures*, we consider the extension of the synergistic algorithms to support offline and online queries. We obtain algorithms and data structures for supporting offline and online queries, which take advantage not only of some measures of the input order and the input structure, but also of notions of difficulty in the queries, such as the query structure (introduced by Dobkin and Munro [27][6]) and some measures of the query order. We describe two compressed data structures to represent a multiset taking advantage both of some measures of the input order and the input structure, while supporting the operators rank and select on it.

We extend our synergistic results about sorting multisets to supporting rank and select queries on multisets, offline and online, according to the potential "difficulty" in both the order and the values in the queries themselves, in addition to the potential difficulty in the data being queried in order and structure (Section 5.1). In the offline setting, this improves upon the algorithms described by Dobkin and Munro [27] and Kaligosi et al. [44]. In the online setting, this improves upon the results described by Barbay et al. [6] by adding three new measures of difficulty (i.e., measuring some form of input order, the input structure, and some form of query order) to the single one previously considered (i.e., measuring the query structure).

Related to computing the maxima set of planar points, we define the problem of supporting *domination* queries on a set of planar points, i.e., given a set of data points and a set of query points, determine which of the query points are below the staircase represented by the maxima set of the data points. In the offline setting, we describe an algorithm that answers several domination queries, taking advantage of a decomposition of the data points into smooth sequences (i.e., a form of input order), of the relative positions of the data points (i.e., the input structure), and of the relative positions between the query points and the data points (i.e., the query structure) (Section 5.2.1). We extend such result to the online setting by describing a deferred data structure that answers domination queries taking advantage of the order in which the queries are given (i.e., the query order) in addition to the other three measures considered (Section 5.2.2).

We study the support of *point membership* queries on the convex hull of a set of planar

[6]Barbay et al. [6] were the first to name this measure *query structure*.

points as defined by Karp et al. [45], i.e. given a set of data points and a set of query points, determine which of the query points are included in the convex hull of the data points. Similarly to the support of domination queries, in the offline setting, we describe an algorithm that answers several point membership queries taking advantage of a decomposition of the data points into simple polygonal chains (i.e., a form of input order), of the relative positions of the data points (i.e., the input structure), and of the relative positions between the query points and the data points (i.e., the query structure) (Section 5.3.1). We extend such a result to the online setting by describing a deferred data structure that answers point membership queries taking advantage of some measure of the query order in addition to the other three measures considered (Section 5.3.2). This improves upon the deferred data structure described by Karp et al. [45], which takes advantage only of the number of data points and the number of query points.

We summarize the main results described in this chapter in the following tables:

Problems	Previous work	This thesis
Rank and Select	$O(\mathcal{B}(M_q))$ ([27]) $\mathcal{B}(M_q) + o(\mathcal{B}(M_q)) + O(n)$ ([44])	$n + O(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\xi} \rho_i \log \rho_i)$ (Thm. 9)
Domination	–	$O\left(n + \beta \mathcal{H}_B(I) + q^{1/2} \log \beta\right)$ (Thm. 14)
Point Membership	–	$O\left(n(\mathcal{H}_B(L) + \mathcal{H}_B(U)) + q \log \frac{b}{q}\right)$ (Thm. 16)

Table 1.3: Results for supporting offline rank and select queries on a multiset, supporting offline domination queries on a set of planar points, and supporting offline point membership queries on the convex hull of a set of planar points.

Problems	Previous work	This thesis
Rank and Select	$\mathcal{B}(M_q) + o(\mathcal{B}(M_q)) + O(n)$ ([6])	$n + O(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\gamma} \rho_i \log \rho_i + \sum_{i=1}^{q-1} \log d_i + \sum_{i=1}^{u} \log g_i)$ (Thm. 11)
Domination	–	$O(n + \beta \mathcal{H}_B(I) + \sum_{i=1}^{q-1} \log d_i + \sum_{j=1}^{q} \log g_j$ (Thm. 15)
Point Membership	$O((n+q) \log(\min\{n, q\})$ ([45])	$O(n(\mathcal{H}_B(U) + \mathcal{H}_B(L)) + \sum_{i=1}^{q-1} (\log d_i + \log e_i) + \sum_{j=1}^{q} (\log f_j + \log g_j))$ (Thm. 17)

Table 1.4: Results for supporting online rank and select queries on a multiset, supporting online domination queries on a set of planar points, and supporting online point membership queries on the convex hull of a set of planar points.

Chapter 6: *Discussion*, we conclude with a discussion of our results and with a partial list of potential issues left open for improvement.

Our results about merging sorted sets (described in Section 3.1), synergistic sorting multisets (described in Section 4.1.2), and algorithms and deferred data structures supporting rank and select queries on multisets (described in Section 5.1.1 and 5.1.2) were developed in collaboration with Jérémy Barbay and Srinivasa Rao Satti, and presented at the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), July 4-6, 2017, Warsaw, Poland, under the title "Synergistic Solutions on MultiSets" [10].

Our results about merging convex hulls (described in Section 3.3), and synergistic computation of the convex hull of planar points (described in Section 4.3) were developed in collaboration with Jérémy Barbay, and presented at the 24th International Computing and

Combinatorics Conference (COCOON'18), July 2-4, 2018, Qingdao, China, under the title "Synergistic Solutions for Merging and Computing Planar Convex Hulls" [8].

The rest of the results described in this thesis have not yet been submitted for peer review consideration: our results about merging of maxima sets (described in Section 3.2), our results about synergistic computation of maxima sets (described in Section 4.2), our results about compressed data structures supporting rank and select queries on multisets (described in Section 5.1.3), our results about supporting domination queries on a set of planar points (described in Section 5.2), and our results about supporting point membership queries on a set of planar points (described in Section 5.3).

Chapter 2

Background

In this thesis, we are mostly concerned with how to design synergistic algorithms that take advantage of "orthogonal" measures of difficulty: some of the measures depending on the input order, and others depending on the input structure. We study the effectiveness of the synergistic techniques in three fundamental problems in Computer Science: sorting multisets, computing maxima sets, and computing convex hulls, the two latter in the plane. We consider the extension of these techniques to supporting (offline and online) related queries.

To start this chapter, we offer a few notes on the model of computations used in the lower bound reviewed and obtained in this thesis: the decision tree model [3] and the algebraic decision tree model [62] (Section 2.1).

We review a classic algorithm that searches for the insertion rank r of an element e in a sorted unbounded array A, whose running time is within $O(\log r)$ (Section 2.2). The computational complexity analysis of the merging algorithms (described in Chapter 3), and in consequence the synergistic results (described in Chapter 4) rely on such an algorithm and its analysis. We review some classic results concerning adaptive (analysis of) algorithms for sorting multisets (Section 2.3), and computing maxima sets and convex hulls (Section 2.4), taking advantage of either some measures of the input order or the input structure. In the context of supporting queries in the offline setting, we review algorithms that answer rank and select queries on a multiset (Section 2.5.1). In the online setting, we review deferred data structures supporting membership queries on a multiset, rank and select queries on a multiset, and point membership queries on the convex hull of a set of planar points (Section 2.5.2). In a similar context, where the focus is to efficiently answer queries using small space, we review compressed data structures for encoding permutations while supporting direct access and inverse access operations (Section 2.5.3). Those results will serve as a base for comparison in Chapter 3 and 4, where we describe synergistic algorithms and data structures that, in large classes of instances, outperform all the solutions we review in this chapter.

2.1 Models of Computation

We briefly describe the models of computation used in the lower bounds reviewed and obtained in this thesis.

2.1.1 Decision Tree Model

In the decision tree model [3], every decision is based on the comparison of two elements. The usual representation of an algorithm that obtains information only through comparisons of values in the input is a decision tree. A decision tree is a rooted tree. In such a representation, each interior node represents a decision and has two outgoing edges. The leaves represent all possible outputs of the algorithm. Given an instance I, the execution of the algorithm starts at the root and follows the edge that represents the outcome of the decision, and continues recursively. Therefore, the execution of the algorithm on I yields a path from the root to some leaf. Given two distinct inputs, the execution of the algorithm on those distinct instances will reach distinct leaves. The number of comparisons performed by the algorithm on I is the length of such a path. Given a decision tree that represents an algorithm, the height of the tree is a lower bound on the number of comparisons performed by the algorithm in the worst case.

The lower bounds that we review for sorting multisets (Section 2.3), and computing the maxima set of planar points (Section 2.4.1), and the one that we obtain for merging maxima sets (Section 3.2), assume the decision tree model of computation.

In the lower bounds for sorting multisets (Section 2.3), the elements to compare are the elements of the multisets. In the lower bounds for computing the maxima set of planar points (Section 2.4.1) and for merging maxima sets (Section 3.2), the elements to compare are the coordinates of the points in the input (i.e., coordinate comparisons).

2.1.2 Algebraic Decision Tree Model

The algebraic decision tree model is a generalization of the decision tree model [62]. Similar to the decision tree model, the algorithm is represented by a rooted tree. The difference is that a polynomial P on the input I is associated with each internal node, in contrast to a decision tree where a comparison is associated with each internal node. In an algebraic decision tree, each internal node has three outgoing edges, and every decision is based on the sign of evaluating such polynomial (i.e., $P(I) < 0$, $P(I) = 0$, and $P(I) > 0$). The leaves represent all possible output of the algorithm. Differently to a decision tree, in an algebraic decision tree several inputs could reach the same leaf (i.e., those inputs that satisfy a set of polynomial equations). Therefore, the evaluation of the complexity is more difficult, and is based on the topology of the input space [62].

The lower bounds that we review for the computation of the convex hull of a set of planar points (Section 2.4.1 and 2.4.2) assume the algebraic decision tree model of computation.

2.2 Searching

An important building block towards the merging algorithms (described in Chapter 3), and as consequence towards the synergistic algorithms (described in Chapter 4), is a solution to the unbounded search problem [13]:

Unbounded Search: given a function $F : \mathbb{N}^+ \rightarrow \{X, Y\}$ such that

$$F(j) = \begin{cases} X & \text{if } j < k \\ Y & \text{if } j \geq k \end{cases}$$

where k is an integer that uniquely defines F, determine k.

In 1976, Bentley and Yao [13] described an infinite family of unbounded search algorithms. Those algorithms are used in searching a large sorted array A, paying a search cost that depends on the insertion rank r of the searched element e (i.e., the number of elements in A that are less than e). We describe only the *doubling search* algorithm. We extensively use doubling search to yield synergistic algorithms for sorting multisets, and for computing maxima sets and convex hulls of planar points. The doubling search algorithm first determines $m = \lfloor \log_2 k \rfloor + 1$ by successively evaluating $F(2^i - 1)$ for $i = 1, 2, 3, \dots$ until $F(2^m - 1) = Y$, which yields a range $[2^{m-1}, 2^m - 1]$ for k. Then a binary search on those 2^{m-1} elements yields the exact value of k.

Given a sorted array A, and an element e of A, the time complexity of finding e using doubling search is at most $2 \log_2 r$, where r is the insertion rank of e in A. Note that, the time complexity for finding e in A using binary search is at most $\log_2 n$, where n is the size of A. For $r \in o(n^{1/2})$, doubling search outperforms binary search, and doubling search never performs asymptotically worse by more than a constant factor than binary search.

2.3 Sorting

We describe here some relevant results for sorting multisets along two axis: one taking advantage of the distribution of the multiplicities of the distinct elements in the multiset (i.e., the input structure) (Section 2.3.1), and others taking advantage of the distribution of the sizes of the runs in the multiset (i.e., one form of input order) (Section 2.3.2). For completeness, we review the algorithmic techniques behind these results and show that the worst-case performances of such algorithms are incomparable, in the sense that neither one is asymptotically better than the other in the worst case over instances formed by n elements (Section 2.3.3).

2.3.1 Input Structure

In 1976, Munro and Spira [56] described the algorithm `MergeSort with Counters` that sorts a multiset M of size n taking advantage of the input structure. The algorithm `MergeSort with Counters` is an adaptation of the traditional sorting algorithm `MergeSort` [49]: it divides M into two parts whose sizes differ at most by one, sorts both parts recursively, and then merges the two sorted parts. The algorithm `MergeSort with Counters` differs from the classical version in that in the merging step, when two elements of the same value v are found, one is discarded and a counter holding the number of occurrences of v is updated. Munro and Spira measured the "difficulty" of the instance in terms of the distribution of the multiplicities of the σ distinct elements in the multiset (i.e., the input structure) by the *entropy function* $\mathcal{H}(m_1, \dots, m_\sigma) = \sum_{i=1}^{\sigma} \frac{m_i}{n} \log \frac{n}{m_i}$, where $m_1, \dots, m_\sigma$ are the multiplicities of the σ distinct elements in M (such that $\sum_{i=1}^{\sigma} m_i = n$), respectively. The time complexity of the algorithm is within $O(n(1 + \mathcal{H}(m_1, \dots, m_\sigma))) \subseteq O(n(1 + \log \sigma)) \subseteq O(n \log n)$. They proved that this time complexity is worst-case optimal over instances of size n with σ distinct elements of multiplicities $m_1, \dots, m_\sigma$, respectively, in the decision tree model.

2.3.2 Input Order

As early as 1973, Knuth [49] described a variant of the algorithm `MergeSort` [49, Section 5.2.4] that takes advantage of some measure of the input order, which he named `Natural MergeSort`. The algorithm `Natural MergeSort` uses a preprocessing step taking linear time to detect runs in the array A. Mannila [52] refined the analysis of the `Natural MergeSort` algorithm to yield a time complexity for sorting an array A of size n in time within $O(n(1+\log\rho)) \subseteq O(n\log n)$, where ρ is the number of runs in A.

Takaoka [63] described the `Minimal MergeSort` algorithm[1] that sorts the array A that represents a multiset of size n. The main idea is to detect the runs first, and then merge them pairwise prioritizing the shortest ones, thus taking advantage of the potential disequilibrium in the distribution of the sizes of the runs. The runs are detected in linear time through a scanning process identifying the positions $i \in [1..n-1]$ in A such that $A[i] > A[i+1]$. Merging the two shortest runs at each step makes the number of comparisons performed by the merging process adaptive to the entropy of the sequence formed by the sizes of the runs. Indeed, the time complexity of the `Minimal MergeSort` algorithm is 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 A and $r_1,\ldots,r_\rho$ are the sizes of the ρ runs in A (such that $\sum_{i=1}^{\rho} r_i = n$), respectively. Takaoka [63] proved that this time complexity is worst-case optimal over instances of size n formed by ρ runs of sizes $r_1,\ldots,r_\rho$, respectively, in the decision tree model.

2.3.3 Comparison Between Sorting Algorithms

The worst-case performances of the algorithms `MergeSort with Counters` (described in Section 2.3.1) and `Minimal MergeSort` (described in Section 2.3.2) are incomparable, in the sense that neither one is asymptotically better than the other in the worst case over instances formed by n numbers. We describe examples of families of instances where the algorithm `MergeSort with Counters` sorts the instance asymptotically faster than the algorithm `Minimal MergeSort` (Example 1), and some other examples of families of instances where the algorithm `Minimal MergeSort` sorts the instance asymptotically faster than the algorithm `MergeSort with Counters` (Example 2).

Example 1. *Consider the family of instances* $\langle 1,2,1,2,\ldots,1,2\rangle$

Each instance of this family is formed by $\frac{n}{2}$ sequences of $\langle 1,2\rangle$. On each instance I of this family, the algorithm `Minimal MergeSort` detects $\frac{n}{2}$ runs and then merges them two at a time. The time complexity of `Minimal MergeSort` on I is within $\Theta(n\log n)$. On the other hand, the algorithm `MergeSort with Counters` takes advantage of the large multiplicities of the elements with values 1 and 2. In every merging step, the resulting set is always $\{1,2\}$. Therefore, the number of elements is reduced by half at each step, which yields a time complexity linear in the size n of I.

[1]It was rediscovered by Barbay and Navarro [7], who generalized it into a compressed data structure for permutations.

Example 2. *Consider the family of instances $\langle 1, 2, \ldots, n \rangle$*

Each instance of this family is formed by n distinct numbers in sorted order. On each instance I of this family, the algorithm `Minimal MergeSort` detects in time linear in the size n of I that the sequence is already sorted, and in turn it finishes. On the other hand, the time complexity of the algorithm `MergeSort with Counters` on I is within $\Theta(n \log n)$, because this algorithm would perform at least as many operations as `MergeSort`, given that all the elements are distinct.

Furthermore in Section 4.1 (page 48), we show that simple modifications and combinations of the algorithms `MergeSort with Counters` and `Minimal MergeSort` do not take full advantage at the same time of the number and sizes of the runs (i.e., a form of input order), and of the multiplicities of the distinct elements in the multiset (i.e., the input structure). To address this problem, we describe two synergistic algorithms that sort a multiset taking advantage both of some measure of the input order and the input structure, such that on large classes of instances, they outperform algorithms `MergeSort with Counters` and `Minimal MergeSort` by more than a constant factor (Section 4.1). In the following sections, we review similar results about the computation of the maxima set and the convex hull of planar points, which can be seen as generalizations of the sorting of multisets to two dimensions.

2.4 Computation of Maxima Sets and Convex Hulls

The computation of the maxima set and the convex hull are fundamental problems in Computational Geometry [60, 23]. The histories of those problems are strongly correlated: most of the results on one also generalize to the other. We formally define the maxima set and the convex hull of planar points.

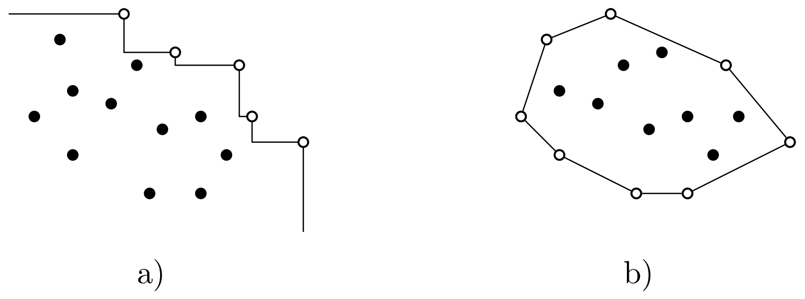


Figure 2.1: A set P of planar points. a) The maxima set of P marked by open disks, and b) the convex hull of P marked by open disks.

Definition 1. *Let p_x and p_y denote the x- and y-coordinates of the point $p \in \mathbb{R}^2$, respectively. Given two points p and q, p* dominates *q if $p_x \geq q_x$ and $p_y \geq q_y$. Given a set P of points in the plane, a point p from P is called* maximal *if none of the other points of P dominates p. The* maxima set *of such a set P is the uniquely defined set of all maximal points.*

Definition 2. *Given a set P of planar points, the* convex hull *of P is the (unique) minimal convex set containing P.*

The vertices of the convex hull of P are a subset of P. In two dimensions, the convex hull is partitioned into two polygonal chains, the upper hull and the lower hull, stretching between the leftmost and rightmost vertices of the convex hull.

A graphical representation of those definitions is given in Figure 2.1. We describe some relevant algorithms that compute the maxima set and the convex hull of planar points along two axes: ones taking advantage of the relative positions of the points (i.e., the input structure) (Section 2.4.1), and others taking advantage of the simple polygonal chains into which the input sequence of points can be partitioned (i.e., one form of input order) (Section 2.4.2). Similarly to sorting multisets, the worst-case computational complexities of such "orthogonal" algorithms are incomparable in the sense that neither one is asymptotically better than the other in the worst case over instances formed by n planar points (Section 2.4.3).

2.4.1 Input Structure

In 1985, Kirkpatrick and Seidel [47] described an algorithm that takes advantage of the input structure to compute the maxima set of a set P of planar points[2]. The algorithm partitions the points of P into two sets P_1 and P_2 by the median μ of the x-coordinates of the points of P (such that all the points of P_1 are to the left of all the points of P_2). The median μ is computed in time linear in the number of points of P using the algorithm described by Blum et al. [15]. The algorithm then computes the maxima set M of P_2 recursively, and computes the set $Q \subseteq P_1$ not dominated by the points of M. Finally, it computes the maxima set of Q, that in addition to M, form the maxima set of P. The time complexity of such an algorithm is within $O(n(1 + \log h)) \subseteq O(n \log n)$, where h is the number of points in the maxima set (i.e., the output size), and n is the number of points of P (i.e., the input size). They showed that this time complexity is worst-case optimal over instances of input size n and output size h, in the decision tree model.

In 1986, Kirkpatrick and Seidel [48], following a similar approach to the one they used to compute the maxima set, described a slightly more complex algorithm that computes the upper hull of a set P of points in the plane (the computation of the lower hull is symmetric, and complete it into the computation of the convex hull). The algorithm relies on a variation of the divide-and-conquer paradigm, which they call the "marriage-before-conquest" principle. So, we name it the `Marriage-before-conquest Hull` algorithm.

Before describing the `Marriage-before-conquest Hull` algorithm, we define the concept of *supporting line*, which is key to the description of it:

Definition 3. *A* supporting line *of a set P of planar points is a straight line that contains a point p of P and that leaves all the points of P in the same half-plane (i.e., p is a vertex of the convex hull of P).*

The `Marriage-before-conquest Hull` algorithm partitions P into two sets P_1 and P_2, whose sizes differ at most by one, using the vertical line ℓ of equation $x = \mu$, where μ is the median of the x-coordinates of the points of P. The algorithm finds the edge e of the upper hull of P that intersects ℓ: for that, it pairs the points of P, and computes the median m of the slopes of the lines through the pairs. It computes then the point q that lies in the supporting line of P of slope m. If there is a pair of points of P to the left of the vertical line of equation $x = q_x$ such that the line through the pair has slope less than m (resp., greater than m), then the rightmost (resp., leftmost) point in the pair cannot be a vertex of e. A symmetric situation arises if the pair of points is to the right of the vertical

[2]They described algorithms for computing the maxima set of points in any dimension. We describe only the version in the plane.

line of equation $x = q_x$ and the slope of the line through the pair is greater than (resp., less than) m: the leftmost (resp., rightmost) point in the pair cannot be a vertex of e. So, the number of candidate points to vertices of e is reduced by half, yielding a linear time procedure to compute e. The algorithm discards all the points below e, and recurses then on the non-discarded points of P_1 and P_2. The time complexity of this algorithm is within $O(n \log h) \subseteq O(n \log n)$, where h is the number of vertices in the convex hull of P (i.e., the output size), and n is the number of points of P (i.e., the input size). They showed that this time complexity is worst-case optimal over instances of input size n and output size h, in the algebraic decision tree model [48].

Both results described above are *output sensitive*, in the sense that the time complexity depends on the size of the output, and can be classified as adaptive to the input structure, as the positions of the points clearly determine the output (and its size).

Sen and Gupta [61] refined the analyses of the algorithms described by Kirkpatrick and Seidel that compute the maxima set [47] and the upper hull [48] of planar points. Their analyses consider a partition of the points determined by the vertical lines through the points in the maxima set and the upper hull of the points, respectively. Formally, they proved that, given a set S of n planar points, such that the points in S are partitioned into h vertical strips of sizes $n_1, \ldots, n_h$ (such that $\sum_{i=1}^{h} n_i = n$), respectively, determined by the vertical lines through the h points in the maxima set (resp., the upper hull) of S, the time complexity of the Kirkpatrick and Seidel's algorithm that computes the maxima set (resp., the upper hull) of S is within $O(n(1 + \mathcal{H}(n_1, \ldots, n_h)) \subseteq O(n(1 + \log h))$. These results subsume the concept of *output sensitive* algorithms. They name it *distribution sensitive* algorithms, as the time complexity depends on a distribution of the points determined by the points in the output. Inspired by the results described by Sen and Gupta [61], in Section 5.3, we analyze synergistic algorithms and data structures supporting point membership queries on a set of planar points.

In the following, we define formally the concepts of *instance optimal* algorithms and *instance optimal* algorithms in the *order oblivious* setting. *Instance optimal* algorithms in the *order oblivious* setting simultaneously subsumes *output sensitive* algorithms, *distribution sensitive* algorithms, and all algorithms that do not take advantage of the order in which the points are given or that assume that the input is given in a random order.

The concept of *instance optimal* algorithms was introduced by Fagin et al. [29]:

Definition 4. *Let $\mathcal{A}$ be a class of algorithms, and let $\mathcal{I}$ be a class of legal inputs to the algorithms. An algorithm $A \in \mathcal{A}$ is* instance optimal *over $\mathcal{A}$ and $\mathcal{I}$ if for every $B \in \mathcal{A}$ and every $I \in \mathcal{I}$, the cost of A on I is at most a constant factor times the cost of B on I.*

In Chapter 3, we describe instance optimal algorithms for merging sorted sets. Afshani et al. [1] introduced the concept of *instance optimal* algorithms in the *order oblivious* setting:

Definition 5. *Let A be an algorithm in a certain class $\mathcal{A}$ such that for every input sequence P, and for every algorithm $A' \in \mathcal{A}$, the maximum cost of A on P is at most a constant factor times the maximum cost of A' on P, where the maximum is taken over all permutations of P. They called algorithms satisfying this property* instance optimal *in the* order oblivious *setting.*

Afshani et al. [1] showed that minor variants of the algorithms described by Kirkpatrick and Seidel for computing the maxima set [47] and the upper hull [48] of planar points are instance optimal in the order oblivious setting, in the decision tree model and the algebraic decision tree model where the tests involve only multilinear functions with a constant number of arguments, respectively. They showed that the time complexity of the variant that computes the maxima set (resp., the upper hull) is within $O(n(1+\mathcal{H}(n_1,\dots,n_h))) \subseteq O(n(1+\log h)) \subseteq O(n\log n)$, where $n_1,\dots,n_h$ (such that $\sum_{i=1}^{h} n_i = n$) are the sizes of a partition of the input points by enclosing rectangles (resp., triangles), such that every rectangle (resp., triangle) is completely below the maxima set (resp., the upper hull) of the points, with the minimum possible value for $\mathcal{H}(n_1,\dots,n_h) = \sum_{i=1}^{h} \frac{n_i}{n}\log\frac{n}{n_i}$.

In the following section, we describe algorithms taking advantage of the order in which the points are given, as opposed to the algorithms described above.

2.4.2 Input Order

For points in the plane, a natural way to consider the input order is through the sequence specified by the order into which the points are listed. A *polygonal chain* is a curve specified by a sequence of points. The curve itself consists of the line segments connecting the pairs of consecutive points. A polygonal chain P is *simple* if any two edges of P that are not adjacent are disjoint, or if the intersection point is a vertex of P; and any two adjacent edges share only their common vertex. See Figure 2.2 for examples of these definitions. Melkman [53] described an algorithm that computes the convex hull of a simple polygonal chain in time linear in the number of points in the polygonal chain, and Chazelle [19] described an algorithm that tests whether a polygonal chain is simple in time linear in number of points in the polygonal chain.

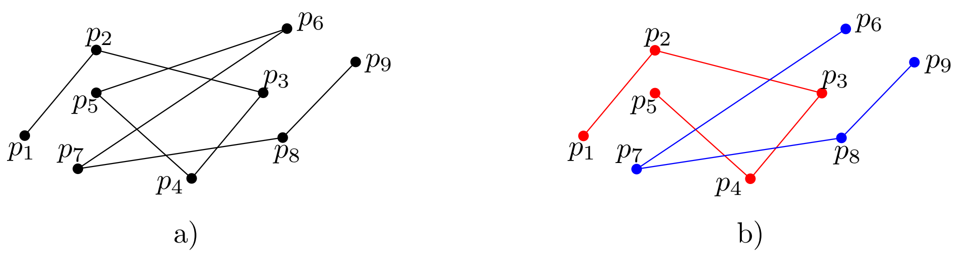


Figure 2.2: a) A polygonal chain P specified by a sequence of 9 points. b) The decomposition of P into two simple polygonal chains.

In 2002, inspired by previous results on sorting permutations [51], Levcopoulos et al. [50] combined the algorithms described by Melkman [53] and Chazelle [19] into an algorithm that computes the convex hull of a polygonal chain P taking advantage of some measure of the input order. The algorithm tests if P is simple, using Chazelle's algorithm [19]: if it is, the algorithm computes the convex hull of P in linear time, using Melkman's algorithm [53]. Otherwise, if P is not simple, it partitions P into two polygonal chains P' and P'', whose sizes differ at most by one; recurses on each of them; and merges the resulting convex hulls using a linear time merging algorithm described by Preparata and Shamos [60]. Levcopoulos et al. [50] measured the time complexity of their algorithm in terms of the minimum number of simple polygonal chains κ into which the polygonal chain P of n points can be partitioned. Let $t(n,\kappa)$ be the worst-case time complexity taken by such algorithm on a polygonal chain of n points that can be partitioned into κ simple polygonal chains. They showed that $t(n,\kappa)$ satisfies the following recursion relation: $t(n,\kappa) \in t(\lceil\frac{n}{2}\rceil,\kappa_1) + t(\lfloor\frac{n}{2}\rfloor,\kappa_2) + O(n)$, such that $\kappa_1+\kappa_2 \le \kappa+1$. The solution to this recursion yields $t(n,\kappa) \in O(n(1+\log\kappa)) \subseteq O(n\log n)$.

They showed that this time complexity is worst-case optimal over instances of n points that can be partitioned into κ simple polygonal chains, in the algebraic decision tree model.

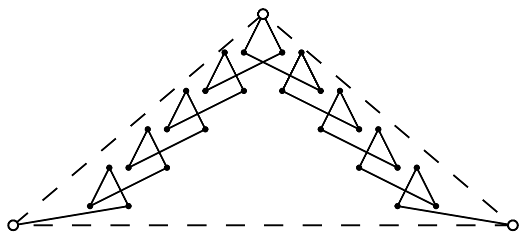
In 2011, Ahn and Okamoto [2] followed a distinct approach for computing the convex hull, also based on some notions of input order. They considered a variant of the problem where the output is the same size of the input, but such that the convex hull can be checked and extracted in linear time from this output. In this context, they described adaptive results that are directly inspired by disorder measures introduced through the study of adaptive algorithms for sorting permutations, such as `Runs` and `Inv` [54, 28]. Inspired by Ahn and Okamoto's definition [2], we define some simple measure of the input order for the computation of the maxima set in Section 4.2.1.

As far as we know, the notion of input order has never been considered for the computation of the maxima set of planar points. The case is different when the computation of the convex hull is considered, where several algorithms that take advantage of either some measures of the input order or the input structure have been described. We compare such algorithms in the following section.

2.4.3 Comparison Between Convex Hull Algorithms

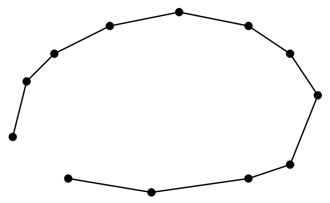
Similarly to the situation of the sorting algorithms described in Section 2.3.3, the worst-case performances of the algorithms described by Kirkpatrick and Seidel [48] (even when analyzed by Afshani et al. [1] such as described in Section 2.4.1) and Levcopoulos et al. [50] (described in Section 2.4.2) are incomparable, in the sense that neither one is asymptotically better than the other in the worst case over instances formed by n points. We describe below examples of families of instances where the algorithm described by Kirkpatrick and Seidel [48] computes the convex hull asymptotically faster than the algorithm described by Levcopoulos et al. [50] (Example 3), and some other examples of families of instances where the algorithm described by Levcopoulos et al. [50] computes the convex hull asymptotically faster than the algorithm described by Kirkpatrick and Seidel [48] (Example 4).

Example 3. *Consider a family of instances such that each is formed by a sequence P of n planar points enclosed by a triangle (noted in dashes in the illustration below) such that every polygonal chain formed by six consecutive points of P has a self-intersection.*



The convex hull of every instance of this family is formed by only three vertices (marked by open disks in the illustration). Hence, the time complexity of the algorithm described by Kirkpatrick and Seidel [48] to compute the convex hull of P is linear in the number of points of P. On the other hand, the minimal number of simple polygonal chains into which P can be partitioned is $\Theta(n)$. Therefore, the time complexity of the algorithm described by Levcopoulos et al. [50] to compute the convex hull of P is within $\Theta(n \log n)$.

Example 4. *Consider a family of instances such that each is formed by a sequence P of n planar points such that each point in P is a vertex of the convex hull of P, and the polygonal chain specified by P is simple (see the illustration below).*



All the points of P are vertices of the convex hull of P. Hence, the time complexity of the algorithm described by Kirkpatrick and Seidel [48] that computes the convex hull of P is within $\Theta(n \log n)$. On the other hand, the algorithm described by Levcopoulos et al. [50] identifies that the whole sequence is a simple polygonal chain, so it computes the convex hull of P in time linear in the number of points of P.

In Chapter 4, we describe synergistic algorithms that compute the maxima set and the convex hull of planar points taking advantage both of some measures of the input order and the input structure, so that they never perform asymptotically worse than previous algorithms [47, 48, 1, 50]. Moreover, on some large classes of instances, asymptotically outperform those previous algorithms by more than a constant factor. In the following section, we review some extensions of the results described in previous sections for supporting offline and online queries.

2.5 Algorithms and Data Structures Supporting Queries

In the context of supporting rank and select queries on a multiset, we review some results in the offline setting, where the queries arrive all at once (Section 2.5.1). In the online setting, where the queries arrive one by one, we review the concept of deferred data structures supporting membership queries on a multiset, rank and select queries on a multiset, and their extension to geometric problems: supporting point membership queries on the convex hull of a set of planar points and supporting dominance search queries on a set of planar points (Section 2.5.2). Such deferred data structures partially reorder the data while answering the queries. In a similar context, we review a data structure that encodes a permutation using small space and efficiently supports direct access and inverse access operations on the permutation (Section 2.5.3).

2.5.1 Offline Rank and Select Operations

Given an element x of a multiset M and an integer $j \in [1..n]$ (e.g., $M = \{1, 2, 3, 3, 4, 5, 6, 7, 8\}$, $x = 3$, and $j = 4$), the rank $\texttt{rank}(x)$ of x is the number of elements smaller than x in M (e.g., $\texttt{rank}(3) = 2$), and selecting the j-th element in M corresponds to computing the value $\texttt{select}(j)$ of the j-th smallest element (counted with multiplicity) in M (e.g., $\texttt{select}(4) = 3$). The support of rank and select queries on M is related to the sorting problem, as sorting the array A representing M makes it possible to support each select query in constant time and each rank query in $O(\log n)$ comparisons. What makes sorting and supporting rank and select queries distinct is that, when there are only a few queries (or when the multiset is dynamically updated), sorting the whole multiset is "overkill", for there are better solutions.

As early as 1961, Hoare [39] 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. [15], based on a linear time median finding algorithm.

In 1980, Frederickson and Johnson [34] described two algorithms that support rank and select queries, respectively, in ρ sorted sets. The algorithms take advantage of the sizes of the sorted sets. The key idea of the selection algorithm is to rule out a constant percentage of the remaining elements on each iteration. The time complexities of both algorithms to answer one query are within $O(\rho + \sum_{i=1}^{\rho} \log n_i)$, where $n_1, n_2, \ldots, n_\rho$ are the sizes of the ρ sorted sets. They proved that those time complexities are worst-case optimal over instances formed by ρ sorted sets of sizes $n_1, \ldots, n_\rho$, respectively, and one query, in the decision tree model. Note that this time complexity is sublinear when $\rho \in o(n)$.

Given a linearly ordered multiset M and a sequence of ranks $r_1, \ldots, r_q$, a multiselection algorithm must answer the queries $\texttt{select}(r_1)$, ..., $\texttt{select}(r_q)$ in M, hence partially sorting M. In 1981, Dobkin and Munro [27] described a multiselection algorithm that supports q select queries at once on a multiset M. The algorithm computes the median μ of M, and partitions M into those elements less than μ, those equals to μ, and those greater than μ. Depending on the sizes of these three sets, the algorithm partitions the select queries, and recurses in each set using the corresponding queries. They showed that the time complexity of the algorithm is within $O(n \log n - \sum_{i=0}^{q} g_i \log g_i)$, where n is the size of M, and the q queries hit positions in M separated by *gaps* (i.e., differences between the ranks of consecutive select queries in sorted order) of sizes $g_0, \ldots, g_q$[3] (i.e., the *query structure*). They proved that this time complexity is worst-case optimal over all multisets of size n and all sets of q queries hitting positions in the multisets separated by *gaps* of sizes $g_0, \ldots, g_q$. The quantity $\mathcal{B}(M_q) = n \log n - \sum_{i=0}^{q} g_i \log g_i$ is the information-theoretic lower bound on the number of comparisons required to answer q selection queries on M, in the decision tree model [27]. Kaligosi et al. [44] later described a variant of this algorithm which performs a number of comparisons within $\mathcal{B}(M_q) + o(\mathcal{B}(M_q)) + O(n)$, which is within an asymptotically negligible additional term of the optimal (as opposed to a constant factor of the optimal as in Dobkin and Munro's result [27]).

In Chapter 5, based on the ideas of Frederickson and Johnson [34], Dobkin and Munro [27], and the synergistic algorithm for sorting multisets described in Section 4.1.2, we describe a multiselection algorithm which takes advantage, at the same time, of the number and sizes of the runs in the data (i.e., a form of input order), of the multiplicities of the elements in the multiset (i.e., the input structure), and of the gaps between the positions hit by the queries (i.e., the query structure).

All the results described in this section assume that the queries are given at the same time (i.e., offline). In the following section, we review results supporting queries assuming that the queries arrive one by one (i.e., online).

[3]Assuming that there are two fictitious queries hitting positions 0 and $n+1$, respectively.

2.5.2 Deferred Data Structures

The usual approach to support queries consists of preprocessing the data, thus building up a data structure that enables queries to be answered efficiently. But, the preprocessing time can considerably exceed the time needed for answering a few queries without preprocessing. Karp et al. [45] offered a new approach, in which the data structure is built up while the queries are answered. They called their solution a deferred data structure and describe it as "lazy", as it partially reorders the data.

Karp et al. [45] studied the support of online *membership queries* on a totally ordered set U, i.e. given a set S of elements drawn from U and a set of queries, where each query is an element of U; for each query, determine whether it is present in S. They described a deferred data structure that given a set of n elements, answers a set of q online membership queries by performing a number of comparison within $O((n+q)\log(\min\{n,q\})$. They proved that this number of comparison is worst-case optimal over instances of n elements and q queries, in the decision tree model.

Barbay et al. [6] extended the result described by Karp et al. [45] to supporting online rank and select queries on a multiset, and refined it by taking advantage of the gaps between the positions hit by the queries (i.e., the query structure). The deferred data structure described by Barbay et al. [6] is 1-competitive with the offline algorithm described by Kaligosi et al. [44] (introduced in the previous section) in the number of comparisons performed. Karp et al.'s approach [45] is based on `MergeSort` [49] while Barbay et al.'s approach [6] is based on `QuickSort` [38]. None of those approaches take into consideration any pre-existing runs in the input, and rather Barbay et al.'s approach [6] builds and maintains such runs as a strategy to minimize the number of comparisons performed.

Karp et al. [45] were most ambitious, and defined the general paradigm for deferred data structures. They described some features essential for a problem that supports queries to be amenable to this approach: the deferred data structure can be adopted for a problem on a set S of n data if: (i) it is possible to answer a query in $O(n)$ time; and (ii) the set S can be partitioned in $O(n)$ time into two subsets S_1 and S_2, whose sizes differ at most by one, such that the answer to the query in S is equal to the answer to the query on either S_1 or S_2, and while the partitioning is computed it is possible to know if the answer is in S_1 or S_2 in constant time. They described deferred data structures for problems in Computational Geometry: supporting point membership queries on the convex hull of a set of planar points, supporting dominance search queries, half plane intersection queries, and fixed-constraint multi-objective linear programming.

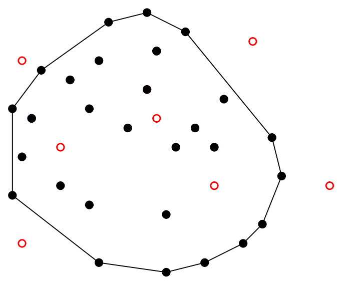


Figure 2.3: An instance of the problem of supporting point membership queries on the convex hull of a set of data points. The black solid disks mark the data points, and the red open disks mark the query points. The convex hull of the data points is marked.

Karp et al. [45] defined the problem of supporting *point membership queries* on the convex hull of a set of data points as follows: given a set of data points and a set of query points in the plane, determine which of the query points are included in the convex hull of the data points (see Figure 2.3 for a graphical representation of this definition). They proved that such a problem fits into their paradigm for deferred data structures. (i) A single query point q can be solved in $O(n)$ time by computing the polar angles from q to all data points. If the range of angles is greater than $180°$, then q is included in the convex hull of the data points. Otherwise, it is not. (ii) Given a set S of planar points, the points of S can be partitioned into two subsets S_1 and S_2 by the vertical line ℓ of equation $x = \mu$, where μ is the median of the x-coordinates of the points in S, in $O(n)$ time [48]. The edges e_u and e_l of the upper and lower hulls of the points of S, respectively, that intersect ℓ can be computed in $O(n)$ time [48]. If the point q is inside the quadrilateral determined by e_u and e_l, then q is included in the convex hull of the data points. Otherwise, it is possible to know whether the answer to q is in S_1 or S_2 in constant time by testing if q is to the right or to the left of ℓ. Such deferred data structure progressively computes the convex hull of the set of data points while answering the queries. Their approach is based on the `Marriage-before-conquest Hull` algorithm (described in Section 2.4.1).

Karp et al. [45] also described a deferred data structure that supports dominance search queries: given a set S of points in a k-dimensional space and a query point q, report the number of points in S dominated by q. This problem does not fit directly into their paradigm as they used a multidimensional divide-and-conquer strategy to solve it.

In Section 5.2, we defined the related problem of supporting *domination queries* on a set of planar points: given a set of data points and a set of query points, determine which of the query points are above the staircase defined by the maxima set of the data points (i.e., which of the query points are not dominated by any of the data points). This formulation is closer to the one of supporting point membership queries than the formulation of supporting dominance search queries.

In Chapter 5, we describe synergistic deferred data structures that support rank and select queries on a multiset taking advantage of the potential easiness in the data (i.e., some measures of the input order and the input structure), as well as, potential easiness in the queries (i.e., some measure of the query order and the query structure) (Section 5.1). We define the query order by a measure that captures how close the positions of two consecutive queries are in terms of the number of pivots computed by the deferred data structure (i.e., the number of nodes in the path between the nodes that represent consecutive queries in the recursion tree of the structure).[4] We also extend these results to problems in Computational Geometry by describing algorithms and deferred data structures supporting domination queries on a set of planar points and point membership queries on the convex hull of a set of planar points (Section 5.2 and 5.3).

In the following section, we lay the foundations for two compressed data structures encoding a multiset M taking advantage both of some measure of the input order and the input structure, while supporting rank and select queries on M (Section 5.3).

[4]See Section 5.1.2 for a more formal description.

2.5.3 Compressed Data Structures

In 2013, Barbay and Navarro [7] described how sorting algorithms in the decision tree model, taking advantage of specificities of the input, directly imply compressed encodings of permutations. By using the similarity of the execution tree of the algorithm `MergeSort` with the `Wavelet Tree` data structure [57], they described a compressed data structure for encoding permutations supporting direct access (i.e. $\pi()$) and inverse access (i.e. $\pi^{-1}()$). They showed that if the permutation π over $[1..n]$ is formed by ρ runs of sizes $r_1, \dots, r_\rho$, then the compressed data structure encodes π in $n(1 + \mathcal{H}(r_1, \dots, r_\rho)) \leq n(1 + \log_2 \rho)$ bits, and supports $\pi()$ and $\pi^{-1}()$ in worst-case time within $O(1 + \log \rho / \log \log n)$, and average time within $O(1 + \mathcal{H}(r_1, \dots, r_\rho) / \log \log n)$, when the argument is uniformly distributed over $[1..n]$.

The main idea behind the construction of the data structure is the same as the adaptive sorting algorithm they described (independently described by Takaoka [63] and that we reviewed in Section 2.3.2), which takes advantage of the number of runs and its sizes in the permutation. The algorithm detects runs first, and then merges them pairwise, using a `MergeSort`-like step. The detection of ascending runs can be done in linear time by a scanning process identifying the positions i in π such that $\pi(i) > \pi(i+1)$. 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 of the sizes of the runs (i.e., a form of input order). The merging process is then represented by a tree with the shape of a Huffman tree [40] for the distribution of the sizes of the runs. Recording in a bit vector the results of each comparison performed by the algorithm in each node of the Huffman-shaped tree is enough to encode the permutation, and to support $\pi()$ and $\pi^{-1}()$ on individual values of π without decompressing the whole encoding. We draw inspiration from this result to design compressed data structures encoding a multiset that take advantage of the input structure, in addition to the measure of input order considered by Barbay and Navarro [7], while supporting rank and select on it (Section 5.1.3).

This concludes the description of the results relevant to this work. In the following chapter, we describe the contributions of this work by designing algorithms for merging sorted sets and merging maxima sets and convex hulls, the latter two in the plane.

Chapter 3

Adaptive Merging

A cornerstone of our approach to present synergistic solutions for sorting multisets, computing maxima sets, and computing convex hulls, is the design of merging algorithms for those structures. We devote this chapter to the study of algorithms for merging sorted sets, maxima sets, and convex hulls. Those merging problems are interesting in their own right, both for their computational complexity and for their applications [24, 5].

Demaine et al. [24] described an instance optimal algorithm for merging sorted sets, based on the sorting algorithm `MergeSort` [49, Section 5.2.4]. We correct their analysis in a minor way (Section 3.1.1). We describe a new instance optimal algorithm `Quick Set Union` for merging sorted sets (Section 3.1.2), which is more suitable to extend to supporting rank and select queries on a multiset. As its name indicates, it is based on the sorting algorithm `QuickSort` [38].

We generalize the algorithm `Quick Set Union`, and its corresponding analysis to merging maxima sets in the plane by describing and analyzing the algorithm `Quick Maxima Union` (Section 3.2). To the best of our knowledge, this is the first algorithm that merges maxima sets.

We also generalize the algorithms `Quick Set Union` and `Quick Maxima Union` to merging convex hulls in the plane by describing and analyzing the algorithm `Quick Hull Union` (Section 3.3), which requires more advanced techniques. Concerning the merging of convex hulls in the plane, Barbay and Chen [5] described an algorithm for merging $k \geq 2$ convex hulls. Barbay and Chen's algorithm is directly inspired by the algorithm described by Demaine et al. [24] for merging sorted sets. Both algorithms compute the merged structure (a sorted multiset in the algorithm described by Demaine et al. [24], and a convex hull in the algorithm described by Barbay and Chen [5]) from left to right.

In Section 3.4, we discuss some issues related to the measures of difficulty we defined to evaluate our algorithms.

3.1 Merging Sorted Sets

We formally define the merging sorted sets problem as follows:

Merging Sorted Sets: Given ρ sorted sets $A_1, \dots, A_\rho$, compute the sorted union $S = A_1 \cup \cdots \cup A_\rho$ of the ρ sorted sets. We refer to the output S as the *merged set*.

We review an instance optimal algorithm[1] described by Demaine et al. [24] for merging sorted sets (Section 3.1.1), which we name `Sorted Set Union`[2]. We describe a new instance optimal algorithm for merging sorted sets (Section 3.1.2), which prepares the ground for the multiselection algorithm (Section 5.1.1) and the deferred data structures that answer rank and select queries on a multiset (Section 5.1.2).

3.1.1 Algorithm Sorted Set Union

In 2000, Demaine et al. [24] described the algorithm `Sorted Set Union`, an instance optimal algorithm for merging sorted sets. The algorithm scans the ρ sorted sets from left to right identifying *blocks* of consecutive elements in the sorted sets that are also consecutive in the merged set S (i.e., the output). See Figure 3.1 for a graphical representation of such a decomposition, on a particular instance of the merging sorted sets problem. It inserts the smallest element of each sorted set in a heap. At each step, it deletes from the heap all the elements whose values are equal to the minimum value v of the heap. If more than one element is deleted, it knows the multiplicity of v in the merged set S, and outputs v and its multiplicity. The blocks corresponding to elements of value with multiplicity greater than 1 contain only this element. For each sorted set that contains v, the algorithm adds to the heap the element immediately following v. On the other hand, if there is only one element of minimum value v, the algorithm extracts from the heap an element whose value u is the second minimum in the heap, and executes a doubling search[3] for the value u in the sorted set A that contains v. Once the algorithm finds the insertion rank r of u in A (i.e., the number of elements whose values are less than u), it identifies a block in A of consecutive positions where the multiplicities of all the elements in the block are 1 in the merged set S (i.e, in A, all the elements whose positions are to the left of r form the block). This block is output, and its elements are discarded from future iterations of the algorithm. The process is repeated until all the elements are discarded. In a minor way, we correct the analysis of the algorithm `Sorted Set Union` as follows:

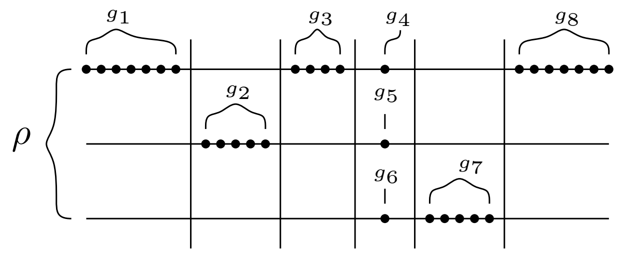


Figure 3.1: An instance I of the merging sorted sets problem computing the union of $\rho = 3$ sorted sets, and a partition certificate π of I. In each sorted set A, the entry $A[i]$ is represented by a point of x-coordinate $A[i]$. The sizes of the blocks that form the sets are noted. The sizes g_4, g_5 and g_6 are equal to 1 because they correspond to elements of equal value, and they induce the 4-th member of π with value m_4 equal to 3. The vertical bars separate the interval members of π.

[1]Demaine et al. [24] did not claim that this algorithm is instance optimal, though Afshani et al. [1] did.

[2]They originally named it "Adaptive" and later [25] introduced another one as "Small Adaptive". We opted for a more descriptive name.

[3]Doubling search is a technique for searching sorted unbounded arrays in which an element of rank r is found by performing $2 \log r$ comparisons [13]. See Section 2.2 for details.

The `Sorted Set Union` algorithm partitions the sorted sets into blocks. Those blocks induce a partition certificate π of the merged set S into intervals such that any singleton corresponds to a value that has multiplicity greater than 1 in the sorted sets, and each other interval corresponds to a block, contained in a single sorted set, whose elements have values with multiplicities equal to 1 in the sorted sets. Each interval member i of π has a value m_i associated with it: if the member i of π is a block whose elements have values with multiplicities equal to 1, then m_i is 1, otherwise, if the member i of π is a singleton corresponding to a value of multiplicity $q > 1$, then m_i is q. If the sorted sets are formed by δ blocks of sizes $g_1, \ldots, g_\delta$ such that these blocks induce a partition certificate π of size χ whose interval members have values $m_1, \ldots, m_\chi$, then we express the number of comparisons performed by algorithm `Sorted Set Union` as within

$$\Theta\left(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i}\right).$$

This number of comparisons is within a constant factor of the number of comparisons performed by any other algorithm computing the merged set of those sorted sets (i.e., the algorithm is instance optimal). Any algorithm for merging sorted sets must certify that the elements of some blocks have multiplicities 1 in the merged set, and potentially that some other elements have multiplicities greater than 1 in the merged set. Demaine et al. [24] described the concept of *proof* that captures such certifications, and using theirs arguments, one can prove that the length of the shortest binary encoding of such a proof for an instance I is the quantity $\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i}$ (the quantity claimed in their work [24] is incorrect, as they considered the second term as the sum over all the distinct values of the output). Hence, given an instance I of such a problem, this quantity is a lower bound on the number of comparisons performed by any correct algorithms on I, in the decision tree model. In the following section, we describe a new instance optimal algorithm for merging sorted sets, which is based on a distinct paradigm, more suitable for extending to supporting queries.

3.1.2 Algorithm Quick Set Union

Given ρ sorted sets, the algorithm `Quick Set Union` computes the merged set S of the ρ sorted sets. As its name indicates, the algorithm is directly inspired by the `QuickSort` [38] algorithm. It computes a pivot μ, which is the median of the set formed by the elements at the middle positions of the sorted sets. It partitions each sorted set into potentially three sorted subsets: a subset formed by those elements less than μ, a singleton subset formed by the element equal to μ, and a subset formed by those elements greater than μ. The partition is done by performing doubling searches for the value μ in all sorted sets, starting at both ends of each sorted set in parallel. If the multiplicity of the value μ is equal to 1 in the sorted sets, then in the sorted set A that contains μ, the algorithm identifies the block B (containing μ) of consecutive positions such that the value of each element of B has multiplicity 1 in S. The block B is identified using doubling searches in A for the element $l \notin A$ of maximum value to the left of μ, and for the element $r \notin A$ of minimum value to the right of μ. Once the sorted sets have been partitioned by the value μ, the algorithm computes the elements l and r in time linear in the number of sorted sets. The algorithm outputs B, and discards its elements from future iterations. The algorithm recurses on the sorted subsets whose elements are smaller than l and on the sorted subsets whose elements are greater than r. (See Algorithm 1 for a more formal description).

Definition 6. *Given ρ sorted sets, the* median of the middles *is the median of the set M formed by the elements at the middle positions of the sorted set. Each element of M is the median of the sorted set that contains it.*

Algorithm 1 `Quick Set Union`

```
Input: $\rho$ sorted sets $A_1, \dots, A_\rho$
Output: The merged set $S = A_1 \cup \dots \cup A_\rho$
1: Compute the median $\mu$ of the middles, note $A_j$ for $j \in [1..\rho]$ the sorted set containing $\mu$;
2: Partition the $\rho - 1$ sorted sets by the value $\mu$, skipping the sorted set $A_j$;
3: Find the maximum $l$ (minimum $r$) among the elements smaller (resp., greater) than $\mu$ in
   all sorted sets, except $A_j$;
4: Compute the insertion ranks of $l$ and $r$ in $A_j$, starting at the position of $\mu$;
5: Output the block in $A_j$ whose elements have values between $l$ and $r$;
6: Recurse on the elements smaller than or equal to $l$ and on the elements greater than or
   equal to $r$.
```

The number of comparisons performed by algorithm `Quick Set Union` is asymptotically the same as the number of comparisons performed by the algorithm `Sorted Set Union` described in the previous section. We divide the proof into two lemmas. We first describe an asymptotic upper bound on the overall number of comparisons performed by all the doubling searches of algorithm `Quick Set Union` (i.e., Steps 2 and 4 of Algorithm 1).

Lemma 1. *Let $g_1, \dots, g_k$ be the sizes of the k blocks into which the algorithm* ***`Quick Set Union`*** *partitions the sorted set A. The overall number of comparisons performed by the doubling searches of the algorithm* ***`Quick Set Union`*** *in A is within $O(\sum_{i=1}^{k} \log g_i)$.*

Proof. In A, every time the algorithm finds the insertion rank of one of the medians of the middles, or, in case that A contains the median of the middles, the insertion rank of l and r, it partitions A by a position separating two blocks, or yields that all the elements of A are less than μ or greater than μ. These doubling search steps can be represented as a tree (see Figure 3.2 for a tree representation of a particular instance). Each node of the tree corresponds to a step. Each internal node has two children, which correspond to the two subsets into which the step partitions A. The cost of each step is less than four times the logarithm of the size of the child subset with smaller size, because of the two doubling searches in parallel. The leaves of the tree correspond to the blocks themselves.

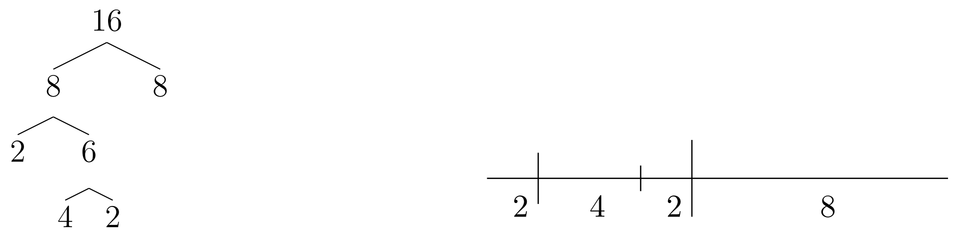


Figure 3.2: The tree that represents the doubling searches in a sorted set A formed by four blocks of respective sizes $2, 4, 2, 8$. The sizes of the subsets are noted in each node. Each internal node has two children, which correspond to the two subsets into which the recursive steps partition A: those blocks to the left of the median μ and those blocks to the right of the median μ.

At each step, the total cost is bounded by eight times the sum of the logarithms of the sizes of the leaf subsets: this is proven 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 step is carried out and a leaf subset is partitioned into two new subsets. At this step, a leaf of the tree is transformed into an 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 $4 \log a$, and hence the sum of the logarithms of the sizes of the leaves increases by $8(\log a + \log b) - 8 \log(a + b)$. But if $a \geq 4$, then $2 \log(a + b) \leq \log a + 2 \log b$. The result follows. □

The step that computes the median μ of the middles of the ρ sorted sets performs a number of comparisons linear in ρ. The step that finds the maximum l (resp., minimum r) among the elements smaller (resp., greater) than μ performs a number of comparisons linear in ρ. As shown in the following lemma, the overall number of comparisons performed during both steps (i.e., Steps 1 and 3 of 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 interval members of the partition certificate π (see Section 3.1.1 for the definition of π) and ρ is the number of sorted sets.

Consider the instance I depicted in Figure 3.3 for an example illustrating the origin of the term $\log \binom{\rho}{m}$. In I, there is a value v that has multiplicity $m > 1$ and the rest of the values have multiplicity 1. The elements with value v are present at the end of the last m sorted sets and the rest of the sorted sets are formed by only one block each. In I, the elements of the sorted set A_i are greater than the elements of the sorted set A_{i+1}, for $i \in [1..\rho - m]$. During the computation of the medians of the middles, the number of comparisons involving elements of value v is within $O(\log \binom{\rho}{m})$. The algorithm computes the median μ of the middles and partitions the sorted sets by the value μ. In each recursive call that involves sorted sets that contain elements of value v, the number of sorted sets is reduced by half. This is repeated until one occurrence of μ belongs to one of the last m sorted sets. The number of comparisons involving elements of value v up to this step is within $O(m \log \frac{\rho}{m}) = O(\log \binom{\rho}{m})$, where $\log \frac{\rho}{m}$ corresponds to the number of steps where μ does not belong to the last m sorted sets. The next recursive call will necessarily choose one element of value v as the median of the middles.

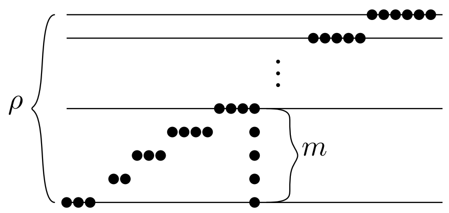


Figure 3.3: An instance I of the merging of sorted sets problem formed by ρ sorted sets. For each sorted set A the entry $A[i]$ is represented by a point of x-coordinate $A[i]$. In I, there is an element of value v with multiplicity m, which is present in the last m sorted set, and the rest of the sorted sets are formed by only one block each.

Lemma 2. *Given an instance I formed by ρ sorted sets, let $m_1, \ldots, m_\chi$ be the values of the members of the partition certificate π of size χ of I. Consider the steps that compute the medians of the middles and that find the elements l and r in the algorithm* ***`Quick Set Union`***. *The overall number of comparisons performed during those 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 sorted sets ρ. The number of comparisons performed by one of these steps is linear in the number of sorted sets in the subinstance (i.e., ignoring all the empty sorted sets of this subinstance). Let $\mathcal{T}(\pi, \rho)$ be the overall number of comparisons performed during the Steps 1 and 3 of algorithm `Quick Set Union`. 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 a and b be the number of sorted sets that are completely to the left and to the right of μ, respectively. Let c be the number of sorted sets that are partitioned by μ. Let π_a and π_b be the partition certificates yielded to the left and to the right of μ, respectively. Then, $\mathcal{T}(\pi, \rho) = \mathcal{T}(\pi_a, a+c) + \mathcal{T}(\pi_b, b+c) + \rho$ because of the two recursive calls and the number of comparisons performed by one execution of those steps. By induction, $\mathcal{T}(\pi_a, a+c) \leq \sum_{i=1}^{\chi_a} m_i \log \frac{a+c}{m_i} - a - c$ and $\mathcal{T}(\pi_b, b+c) \leq \sum_{j=1}^{\chi_b} m_j \log \frac{b+c}{m_j} - b - c$. Hence, we need to prove that $a + b \leq \sum_{i=1}^{\chi_a} m_i \log \left(1 + \frac{b}{a+c}\right) + \sum_{j=1}^{\chi_b} m_j \log \left(1 + \frac{a}{b+c}\right)$, but this is a consequence of $\sum_{i=1}^{\chi_a} m_i \geq a + c, \sum_{j=1}^{\chi_b} m_j \geq b + c$ (the number of blocks is greater than or equal to the number of sorted sets); $a \leq b + c, b \leq a + c$ (at least $\frac{\rho}{2}$ sorted sets are left to the left and to the right of μ); and $\log \left(1 + \frac{y}{x}\right)^x \geq y$ for $y \leq x$. $\square$

Similarly to what happens in the analysis of the number of comparisons performed by the algorithm `Sorted Set Union`, the number and sizes of the blocks into which the algorithm `Quick Set Union` partitions the sorted sets (and as a consequence, the partition certificate π) are independent of the algorithm, and are uniquely determined by the relations between the elements in the sorted sets. Combining Lemmas 1 and 2 yields an upper bound on the number of comparisons performed by the algorithm `Quick Set Union`:

Theorem 1. *Let $A_1, \ldots, A_\rho$ be ρ sorted sets formed by δ blocks of sizes $g_1, \ldots, g_\delta$ such that those blocks induce a partition certificate π of size χ of the merged set whose interval members have values $m_1, \ldots, m_\chi$. The algorithm* ***`Quick Set Union`*** *computes the merged sorted set $S = A_1 \cup \cdots \cup A_\rho$ performing a number of comparisons within*

$$\Theta\left(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i}\right).$$

The number of comparisons performed by the algorithms `Sorted Set Union` and `Quick Set Union` are asymptotically the same. So the algorithm `Quick Set Union` is instance optimal as well. In the following sections, we generalize the algorithm `Quick Set Union` to algorithms for merging maxima sets and for merging convex hulls.

3.2 Merging Maxima Sets

We formally define the merging maxima sets problem as follows:

Merging Maxima Sets: Given an instance I of ζ maxima sets $M_1, \ldots, M_\zeta$ in the plane, compute the maxima set M of their union $M_1 \cup \cdots \cup M_\zeta$. The points of each of the maxima sets are given in sorted order.[4] We refer to the output M as the *merged maxima set*.

In the following section, we describe the algorithm `Quick Maxima Union` for merging maxima sets in the plane. Similarly to the algorithm `Quick Set Union` for merging sorted sets (described in Section 3.1.2), the algorithm `Quick Maxima Union` identifies blocks of consecutive points in the maxima sets that form part of the merged maxima set (i.e., the output), in time logarithmic in the number of points in the blocks. One important difference with the algorithm `Quick Set Union` is that it also discards blocks of consecutive points in the maxima sets that are dominated by the same point, in time logarithmic in the number of points in the blocks. The algorithm `Quick Maxima Union` generalizes the algorithm `Quick Set Union`, and is a building block towards the synergistic algorithm that computes the maxima set of planar points, described in Section 4.2.2.

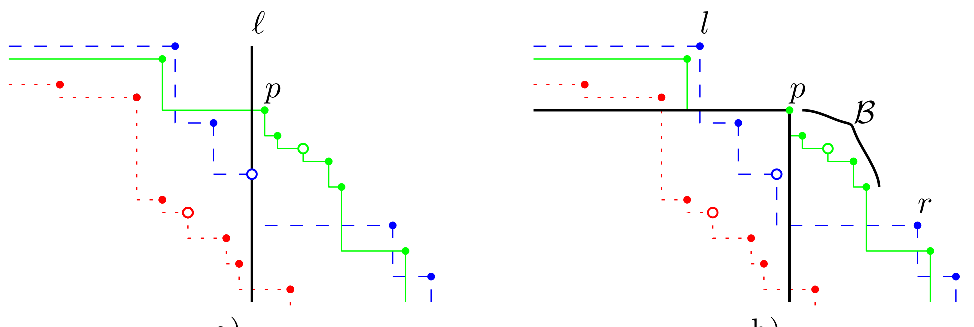


Figure 3.4: An instance of the merging maxima sets problem formed by three maxima sets. The three open disks represent the points at the middle positions in the maxima sets. a) The vertical bar represents the line ℓ of equation $x = \mu$, where μ is the median of the x-coordinates of the points at the middle positions in the maxima sets. The point p is the point of maximum y-coordinate to the right of ℓ. b) The region dominated by p is indicated by solid vertical and horizontal straight semi-line through p. Among the points that are not dominated by p and that do not belong to the maxima set N that contains p, let l be the point of maximum x-coordinate to the left of p, and let r be the point of maximum y-coordinate to the right of p. B is the block of consecutive points in N that are in the region determined by the vertical line of equation $x = l_x$ and the horizontal line of equation $y = r_y$.

3.2.1 Description of Algorithm Quick Maxima Union

The central idea of algorithm `Quick Maxima Union` is to choose a point p that forms part of the merged maxima set of the ζ maxima sets, discards all the points dominated by p, and partitions the ζ maxima sets in a balanced way. Such a balance is achieved by choosing p as the point of maximum y-coordinate to the right of the vertical line ℓ of equation $x = \mu$, where μ is the median of the x-coordinates of the points at the middle positions in the lists of points that represent the maxima sets. See Figure 3.4 for a graphical representation of some of the steps of the algorithm. The algorithm computes the median μ in time linear in

[4]The points of each of the maxima sets are given in sorted order by their x-coordinates or y-coordinates, which are equivalent in the context of maxima sets in the plane: if the points are sorted in ascending order by their x-coordinates, then their y-coordinates are sorted in decreasing order.

Algorithm 2 `Quick Maxima Union`

```
Input: A set of $\zeta$ maxima sets
Output: The merged maxima set of the union of the $\zeta$ maxima sets
1: Compute the median $\mu$ of the $x$-coordinates of the points at the middle position in the maxima sets;
2: Partition the maxima sets by the vertical line $\ell$ of equation $x = \mu$;
3: Find the point $p$ of maximum $y$-coordinate to the right of $\ell$, note $N$ the maxima set containing $p$;
4: Discard all points dominated by $p$;
5: Find the point $l$ of maximum $x$-coordinate such that $l_y > p_y$, in all the maxima sets except $N$;
6: Find the point $r$ of maximum $y$-coordinate such that $r_x > p_x$, in all the maxima sets except $N$;
7: Output the block $B$ of consecutive points in $N$ that are in the region determined by the vertical line of equation $x = l_x$ and the horizontal line of equation $y = r_y$;
8: Recurse separately on the non-discarded points to the left and to the right of $p$.
```

the number of maxima sets [15]. The point p is identified by performing ζ doubling searches: each searches for the value μ in the list of x-coordinates of the points in each of the ζ maxima sets. Those doubling searches guarantee that the algorithm `Quick Maxima Union` identifies blocks of consecutive points in the maxima sets that form part of the merged maxima set (i.e., the output), in time logarithmic in the number of points in the blocks, and that it discards blocks of consecutive points in the maxima sets that are dominated by the same point, in time logarithmic in the number of points in the blocks. The point p is the one of maximum y-coordinate over the points found by such doubling searches. The algorithm takes advantage of the fact that the points in the maxima sets are given in sorted order.

In each maxima set V, the algorithm then discards all the points dominated by p, through doubling searches for p_x and p_y in the x- and y-coordinates, respectively, of the points in V. The points dominated by p do not belong to the maxima set that contains p. The choice of p guarantees that at least half of the maxima sets have points dominated by p or to the left of p, and at least half of the maxima sets have points dominated by p or to the right of p.[5]

Let N denote the maxima set that contains p. In N, the algorithm identifies a block B of consecutive points that forms part of the merged maxima set M (p is contained in B), and outputs all the points in B. For that, among the points that are not dominated by p and that do not belong to N, the algorithm computes the points l and r of maximum x-coordinate to the left of p and of maximum y-coordinate to the right of p, respectively. In each maxima set, the candidates for l and r are the points immediately to the left of the leftmost dominated point and immediately to the right of the rightmost dominated point, respectively. The algorithm computes l and r in time linear in the number of maxima sets, once the points dominated by p are discarded.

Finally, the algorithm recurses separately on the non-discarded points to the left of p and on the non-discarded points to the right of p. See Algorithm 2 for a more formal description. Next, we analyze the time complexity of algorithm `Quick Maxima Union`.

[5]Those two sets are not necessarily disjoints.

3.2.2 Complexity Analysis of Algorithm Quick Maxima Union

Each algorithm for merging maxima sets must certify that blocks of consecutive points in the maxima sets are dominated, or are indeed in the merged maxima set (i.e., the output). We formalize here the notion of a *maxima partition certificate*, that permits to check the correctness of the output potentially in less time than that needed to recompute the output itself. A *maxima partition certificate* of an instance is a partition of the points of the maxima sets into regions so that, in each region, it is "easy" to certify whether the points form part of the output or not. We define a "language" of basic "arguments" for such certificates: *domination* (which justify the discarding of points from the input) and *maximality* (which justify the presence of points in the output) arguments, and their key positions in the instance. A maxima partition certificate will be verified by checking each of its arguments.

Given a maxima set M, let $M[a]$ denote the a-th point in M, and let $M[b..c]$ denote the block of consecutive $c-b+1$ points corresponding to positions from b to c in M.

Definition 7. *Given maxima sets $M, N_1, \ldots N_t$, $\langle M[a] \supset N_1[b_1..c_1], \ldots, N_t[b_t..c_t]\rangle$ is a* domination argument *if the point $M[a]$ dominates all the points in the blocks $N_1[b_1..c_1], \ldots, N_t[b_t..c_t]$.*

Lemma 3. *A* domination argument *$\langle M[a] \supset N_1[b_1..c_1], \ldots, N_t[b_t..c_t]\rangle$ can be checked in $O(t)$ coordinate comparisons.*

It is not enough to "eliminate" all the points that cannot be part of the output. Certifying that the remaining points cannot be "eliminated" still requires additional work: a correct algorithm must justify the exactness of its output. To this end, we define maximality arguments.

Definition 8. *Given maxima sets $M, N_1, \ldots, N_t$, $\langle M[a..b] \dashv N_1[c_1..d_1], \ldots, N_t[c_t..d_t]\rangle$ is a* maximality argument *if **either** the point $M[b]$ dominates the points $N_1[c_1], \ldots, N_t[c_t]$ and the x-coordinates of the points $N_1[c_1-1], \ldots, N_t[c_t-1]$ are less than the x-coordinate of the point $M[a]$ **or** the point $M[a]$ dominates the points $N_1[d_1], \ldots, N_t[d_t]$ and the y-coordinates of the points $N_1[d_1+1], \ldots, N_t[d_t+1]$ are less than the y-coordinate of the point $M[b]$.*

If $\langle M[a..b] \dashv N_1[c_1..d_1], \ldots, N_t[c_t..d_t]\rangle$ is a *maximality argument*, then the points in the block $M[a..b]$ are maximal among the maxima sets $M, N_1, \ldots, N_t$.

Lemma 4. *A* maximality argument *$\langle M[a..b] \dashv N_1[c_1..d_1], \ldots, N_t[c_t..d_t]\rangle$ can be checked in $O(t)$ comparisons.*

Those atomic arguments combine into a general definition of a maxima partition certificate that any correct algorithm for merging maxima sets in the decision tree model can be modified to output (without increasing the magnitude of its computational complexity):

Definition 9. *Given an instance I of the merging maxima sets problem, a* maxima partition certificate *C of I is a partition of the points into regions, so that in each region, the points of I that belong to the output can be decided using a constant number of domination and maximality arguments. The* size *of C is the number of regions which compose it.*

We divide the analysis of the complexity of algorithm `Quick Maxima Union` into two lemmas. We first bound the cumulated number of coordinate comparisons of the doubling searches for the medians of the x-coordinates of the points at the middle positions of the maxima sets (i.e., Step 2 of Algorithm 2) and the doubling searches that discard or output points (i.e., Steps 4 and 7 of Algorithm 2). The algorithm partitions the maxima sets into blocks of consecutive points, where each block is either discarded because it is dominated, or output because it forms part of the merged maxima set. Each block belongs to some argument of the maxima partition certificate computed by the algorithm.

Lemma 5. *Consider a maxima set N. Let $s_1, \dots, s_\beta$ be the sizes of the β blocks into which the algorithm* `Quick Maxima Union` *divides N. The cumulated number of coordinate comparisons performed by the doubling searches of the algorithm* `Quick Maxima Union` *in N is within $O(\sum_{j=1}^{\beta} \log s_j)$.*

Proof. In the maxima set N, every time the algorithm executes a doubling search for the insertion rank of one of the medians μ of the x-coordinates of the points at the middle positions in the maxima sets, it finds a position d inside a block B whose points will be discarded or output. The doubling searches for the insertion rank of p_x and p_y start from position d, and partition N into positions separating the blocks to the left of B, the block B itself, and the blocks to the right of B.

Similarly to the proof of Lemma 1 for Algorithm 1 on multisets, the steps that combine those doubling searches can be represented as a tree. Each internal node has two children, which correspond to the two subsets into which the recursive steps partition N: those blocks to the left of B and those to the right of B. At each node, the number of coordinate comparisons is bounded by $O(\log s_B + \log s)$, where s is the minimum between the sum of the sizes of the blocks to the left of B and the sum of the sizes of the blocks to the right of B, and s_B is the size of the block B. The size of each internal node is the size of the block discarded or output in this step. The size of each leaf is the sum of the sizes of the blocks in the child subset represented by this leaf.

At each step, the total number of coordinate comparisons is bounded by eight times the sum of the logarithms of the sizes of the nodes in the tree. This is proved 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, then a new step is done and a leaf subset is partitioned into two new subsets. At this step, a leaf of the tree is transformed into an internal node and two new leaves are created. Let w and z such that $w \leq z$ be the sizes of the new leaves created. Note that w and z are the sum of the sizes of the blocks to the left and to the right, respectively, of the block B. The cost of this step is less than $4\log w + 4\log s_B$. The cost of all the steps then increases by $4\log w + 4\log s_B$, and hence eight times the sum of the logarithms of the nodes in the tree increases by $8(\log w + \log z + \log s_B - \log(w + z + s_B))$. As $w \geq 3$ and $w \geq s_B$ the result follows. □

We bound the number of coordinate comparisons of the steps that compute the median μ of the x-coordinates of the points at the middle positions in the maxima sets (i.e., Step 1 of Algorithm 2) and the ones that find the points p, l and r (i.e., Steps 3, 5, and 6 of Algorithm 2) in the `Quick Maxima Union` algorithm. The number of coordinate comparisons performed by one execution of those steps is bounded by the number of maxima sets in the subinstance. The partition of the maxima sets by ℓ and the steps that discarded or output points decrease the number of maxima sets in the subinstances. We state the following lemma in function of the maxima partition certificate C computed by the algorithm `Quick Maxima Union`.

Lemma 6. *Given an instance formed by ζ maxima sets, the cumulated number of coordinate comparisons performed during the steps that compute the median μ and that find the points p,*

l and r of algorithm ***Quick Maxima Union*** *is within $O(\sum_{i=1}^{\delta} \log \binom{\zeta}{m_i})$, where δ is the size of the maxima partition certificate C computed by the algorithm, and $m_1, \ldots, m_\delta$ is a sequence where m_i is the number of blocks that form the i-th argument of C.*

Proof. We prove this lemma by induction over δ and ζ. Let $T(\delta, \zeta)$ be the cumulated number of coordinate comparisons performed during the execution of the steps that compute the medians μ of the x-coordinates of the middles points (i.e., Step 1) and during the steps that find the points p, l and r (i.e., Steps 3, 5 and 6) in the algorithm `Quick Maxima Union`. We prove that $T(\delta, \zeta) \leq \sum_{i=1}^{\delta} m_i \log \frac{\zeta}{m_i} - \zeta$, where m_i is the number of maxima sets whose blocks form the i-th argument of C. Let μ be the first median of the x-coordinates of the middles points of the maxima sets computed by the algorithm. Let c and d be the number of maxima sets that have non-discarded points only above of p_y and only to the right of p_x, respectively. Let b be the number of maxima sets that have non-discarded points above p_y and to the right of p_x. Let δ_c and δ_d be the number of arguments computed by the algorithm above p_y and to the right of p_x, respectively. Then, $T(\delta, \zeta) = T(\delta_c, c+b) + T(\delta_d, d+b) + \zeta$ because of the two recursive calls and one execution of the Steps 1, 3, 5 and 6. By the Induction Hypothesis, $T(\delta_c, c+b) \leq \sum_{i=1}^{\delta_c} m_i \log \frac{c+b}{m_i} - c - b$ and $T(\delta_d, d+b) \leq \sum_{j=1}^{\delta_d} m_j \log \frac{d+b}{m_j} - d - b$. We need to prove that $c + d \leq \sum_{i=1}^{\delta_c} m_i \log \left(1 + \frac{d}{c+b}\right) + \sum_{j=1}^{\delta_d} m_j \left(1 + \frac{c}{d+b}\right)$, but this is a consequence of $\sum_{i=1}^{\delta_c} m_i \geq c + b, \sum_{j=1}^{\delta_d} m_j \geq d + b$ (the number of blocks is greater than or equal to the number of maxima sets); $c \leq d + b, d \leq c + b$ (at least $\frac{\zeta}{2}$ maxima sets are left to the left and to the right of μ); and $\log \left(1 + \frac{y}{x}\right)^x \geq y$ for $y \leq x$. □

Given an instance I of ζ maxima sets, the size of the maxima partition certificate computed by the algorithm `Quick Maxima Union` on I is proportional to the minimum size among all possible partition certificates for I. We formalize this result as follows:

We describe the algorithm `Left-to-Right` for merging maxima sets. Such an algorithm is directly inspired by the algorithm `Sorted Set Union` for merging sorted sets [24] (described in Section 3.1.1). Though the number of comparisons performed by the algorithm `Left-to-Right` could be greater than the number of comparisons performed by the algorithm `Quick Maxima Union` on an instance I, the size (i.e., the number of regions) of the maxima partition certificate computed by the algorithm `Left-to-Right` is minimum among valid maxima partition certificates for I.

The algorithm `Left-to-Right` chooses the first point from the left of each maxima set, and computes the points u and v of maximum and second maximum y-coordinate among those points, respectively. Let N be the maxima set that contains u. Let a be the position of u in N (i.e., $u = N[a]$). The algorithm searches for the insertion rank of v_y in N. Let b be the position of the rightmost point g in N such that $g_y > v_y$. The block $N[a..b]$ forms part of the merged maxima set (i.e., the output). The algorithm outputs the block $N[a..b]$ and discards it for future iterations. If g dominates v, the algorithm discards all points dominated by g. The algorithm restarts the computation on the non-discarded points.

Lemma 7. *Given an instance I formed by ζ maxima sets, the algorithm* ***Left-to-Right*** *computes a maxima partition certificate of minimal size for I.*

Proof. Let u and v be the points with maximum and second maximum y-coordinate among the first points from left to right of the non-discarded points of the maxima sets. Let N be the maxima set that contains u. Then, all points in N with y-coordinate greater than v_y form part of the merged maxima set, and v is the point that allows the algorithm to output the greatest number of consecutive points including u in N. Let g and h be consecutive points in N such that $g_y > v_y > h_y$. If g dominates v, then g is the rightmost point in N that dominates v. Hence, g is the point in N that dominates the maximum number of consecutive points including v in the maxima set that contains v. Those two arguments are enough to prove that the algorithm computes a maxima partition certificate with the minimal number of arguments. □

The maxima partition certificate computed by the algorithm `Quick Maxima Union` in any instance I has size proportional to the size of the partition certificate of minimum size among all valid maxima partition certificates for I:

Lemma 8. *The algorithm* ***`Quick Maxima Union`*** *computes a maxima partition certificate whose size is a constant factor of the size of the maxima partition certificate of minimal size.*

Proof. Consider an instance I. In I, suppose that there is a block B of consecutive points in a maxima set N that the algorithm `Left-to-Right` outputs. Suppose that the algorithm `Quick Maxima Union` running on I computes a point p that is contained in B, such that p is the point of maximum y-coordinate among the points of x-coordinates greater than some of the medians (i.e., p is the point computed in Step 3 of Algorithm 2). Let r be the point of maximum y-coordinate such that $r \notin N$ and $r_x > p_x$ (i.e., r is the point computed in Step 6 of Algorithm 2). Let h be the point used by the algorithm `Left-to-Right` to identify the rightmost point in B. By definition, $r_y < h_y$. Let l be the point of maximum x-coordinate such that $l \notin N$ and $l_y > p_y$ (i.e., l is the point computed in Step 5 of Algorithm 2). Let u be the point used by the `Left-to-Right` algorithm to discard dominated points before it identifies the points of B. By definition, l is the same point as u. The algorithm `Quick Maxima Union` therefore outputs (at least) the block B using a constant number of arguments. The result follows. □

Combining Lemmas 5 and 6 yields an upper bound on the number of coordinate comparisons performed by the algorithm `Quick Maxima Union` in function of the size of the partition certificate computed by it.

Theorem 2. *Given ζ maxima sets, the algorithm* ***`Quick Maxima Union`*** *computes the merged maxima set of the ζ maxima sets performing a number of coordinate comparisons within*

$$O\left(\sum_{j=1}^{\beta} \log s_j + \sum_{i=1}^{\delta} \log \binom{\zeta}{m_i}\right),$$

where β is the number of blocks of sizes $s_1, \ldots, s_\beta$, respectively, in the maxima partition certificate C of minimum size δ computed by the algorithm, and $m_1, \ldots, m_\delta$ is a sequence where m_i is the number of blocks that form the i-th argument of C. Such a number of comparisons is worst-case optimal over instances formed by ζ maxima sets that have maxima partition certificates C of size δ formed by β blocks of sizes $s_1, \ldots, s_\beta$ such that $m_1, \ldots, m_\delta$ is a sequence where m_i is the number of blocks that form the i-th argument of C, in the decision tree model.

Proof. We prove that the number of coordinate comparisons performed by the algorithm `Quick Maxima Union` is asymptotically optimal in the worst case over instances formed by ζ maxima sets that have maxima partition certificates C of size δ formed by β blocks of sizes

$s_1, \dots, s_\beta$ such that $m_1, \dots, m_\delta$ is a sequence where m_i is the number of blocks that form the i-th argument of C.

The upper bound is a consequence of Lemma 5, 6, and 8. We describe the lower bound below: it is a simple adversary argument, based on the definition of a family of "hard" instances for each possible value taken by the parameters of the analysis, building over each other.

First, we verify the lower bound for "easy" instances, of finite difficulty: general instances of ζ maxima sets of sizes $r_1, \dots, r_\zeta$ that admit a maxima partition certificate of constant size (i.e., the merged maxima set M is formed by a constant number of blocks of consecutive points in the maxima sets). Such instances require $\Omega(\sum_{i=1}^{\zeta} \log r_i)$ operations: no correct algorithm can afford to ignore a single maxima set, which could contribute to the merged maxima set M, and for each discarded maxima set it needs to certify that all points lie underneath the staircase that represents M.

Such distributions of "elementary" instances can be duplicated to produce δ distributions of elementary subinstances and concatenated to define a distribution of harder instances. Suppose that such elementary subinstances are pairwise independent, in the sense that none of the points in one subinstance dominate or are dominated by the points in another subinstance. If the maxima sets of those elementary subinstances are combined to produce a total of ζ maxima sets in the hard instance, then any algorithm that merges such ζ maxima sets must distinguish in each of the δ elementary subinstances, which of the ζ maxima sets compose such a subinstance. Hence, following similar arguments to the one described by Demaine et al. [24], a lower bound of $\Omega(\sum_{i=1}^{\delta} \log \binom{\zeta}{m_i})$ is obtained.

Combining the lower bound for computing the maxima sets of the elementary instances with the lower bound for distinguishing which of the ζ maxima sets compose each elementary subinstance, we obtain a lower bound of $\Omega(\sum_{j=1}^{\beta} \log s_j + \sum_{i=1}^{\delta} \log \binom{\zeta}{m_i})$. □

The histories of the computation of the maxima set and of that of the convex hull are strongly correlated: most of the results on one problem also generalize to the other one. Our results on the merging of maxima sets similarly generalize to the merging of convex hulls, yet they require more sophisticated techniques to identify dominant and eliminator blocks, which we describe in the next section.

3.3 Merging Convex Hulls

We formally define the merging convex hulls problem as follows:

Merging Convex Hulls: Given an instance of κ convex hulls $C_1, \dots, C_\kappa$ in the plane, compute the convex hull of their union $C_1 \cup \dots \cup C_\kappa$. The points of each of the convex hulls are given in clockwise order.

The computation of the convex hull in the plane reduces to the computation of the upper hull (the computation of the lower hull is symmetric, and completes it into the computation of the convex hull) [48]. We focus here on the merging upper hulls problem:

Merging Upper Hulls: Given an instance of κ upper hulls $U_1, \dots, U_\kappa$ in the plane, compute the upper hull U of their union $U_1 \cup \dots \cup U_\kappa$. The points of each of the upper hulls are given in sorted order by their x-coordinates. We refer to the output U as the *merged upper hull.*

Algorithm 3 `Quick Hull Union`

```
Input: A set of $\kappa$ upper hulls
Output: The merged upper hull $U$
1: Compute the median $\mu$ of the slopes of the edges at the middle positions of the $\kappa$ upper
   hulls;
2: Identify the "pivot" point $p$ that has a supporting line of slope $\mu$;
3: Partition the $\kappa$ upper hulls by the vertical line through $p$;
4: For each upper hull $V$, compute the (at most) two tangents of $V$ through $p$: the ones
   to the left and right of $p$, and discard the blocks of consecutive points below the line
   segments determined by the points of tangency;
5: Output a block of points in the upper hull $W$ containing $p$ that forms part of the merged
   upper hull, by computing common tangents between $W$ and the other upper hulls;
6: Discard all points that lie below the line segments determined by the points in the
   common tangents between $W$ and the other upper hulls;
7: Recurse on the resulting upper hulls to the left and to the right of $p$.
```

We describe the algorithm `Quick Hull Union` for merging upper hulls and analyze its time complexity. This algorithm is inspired by the algorithm `Simplified Ultimate Planar Convex Hull`, described by Chan et al. [17], and by the algorithms `Quick Set Union` and `Quick Maxima Union` described in previous sections. The `Quick Hull Union` algorithm is an essential building block towards the synergistic algorithm for computing the convex hull of planar points, described and analyzed in Section 4.3.2.

3.3.1 Description of Algorithm Quick Hull Union

In the context of merging upper hulls, in each of the upper hulls the points are given sorted by their x-coordinates, and the slopes of the edges monotonically decrease from left to right. The algorithm `Quick Hull Union` takes advantage of those facts: its pseudocode is described in Algorithm 3. Similarly to the `Quick Maxima Union`, in each of the upper hulls, the algorithm identifies blocks of consecutive points that form part of the merged upper hull U and blocks of consecutive points that lie underneath U. The algorithm uses a divide-and-conquer approach to take advantage of the positions of the points.

In each upper hull V, the edge at the middle position is the one whose slope is the median among the slopes of the edges of V. Let μ be the median of the slopes of the edges at the middle positions of the upper hulls. Similarly to Kirkpatrick and Seidel's algorithm [48] (described and analyzed in Section 2.4.1), we show that the point p that has a supporting line (see Definition 3 page 15) of slope μ is a good candidate for discarding points that cannot be part of U (i.e., the output). Note that the time complexity of computing μ is linear in the number κ of upper hulls, but that the time complexity of pairing the points and computing the median of the slopes of the lines through the pairs is linear in n, the number of points.

The algorithm `Quick Hull Union` identifies a "pivot" vertex p of the merged upper hull U, and uses p to discard blocks of consecutive points that cannot be part of U. It computes the median μ of the slopes of the edges at the middle positions of the upper hulls, and identifies p as the point that has a supporting line ℓ of slope μ. Note that p is the extreme point in the direction orthogonal to ℓ. Taking advantage that in each upper hull V, the slopes of the edges are sorted, the algorithm identifies the extreme point in the direction orthogonal to ℓ by performing a doubling search for the value μ in the list of slopes of the edges of V. (See Figure 3.5 for a graphical representation of these steps.)

To know which points are to the left and which ones are to the right of p, the algorithm partitions the points in the upper hulls by the vertical line through p. Such partitioning is carried out by performing a doubling search for the value p_x in the x-coordinates of the points in each upper hull.

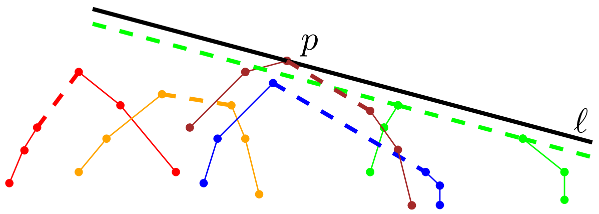


Figure 3.5: An instance of the merging upper hulls problem. The edges at the middle positions of the upper hulls are marked by thick dashed segments, and the one whose slope is the median μ has been extended into a line. The straight line ℓ is the supporting line of slope μ. Line ℓ passes through the "pivot" vertex p.

For each upper hull V, the algorithm then computes the (at most) two tangents of V through p: the one passing through a point to the left of p in V, and the one passing through a point to the right of p in V. If all the points of V are to the left or to the right of p, then the algorithm only computes one tangent. In V, the algorithm discards the blocks of consecutive points below the line segments determined by the points of tangency. It computes all the tangents via doubling searches [5] (see Figure 3.6 for an example).

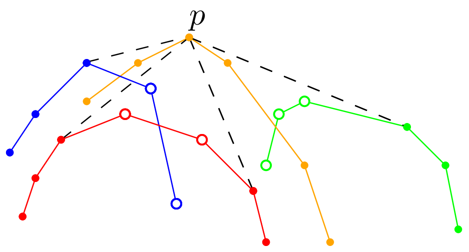


Figure 3.6: An instance of the merging upper hulls problem. The tangents of the upper hulls through p are marked by dashed lines. The open disks mark the points that are discarded because they are below the line segments defined by the points of tangency.

Before the recursive step, in the upper hull W containing p, the algorithm identifies a block B of consecutive points that forms part of the merged upper hull U (p is included in B). The algorithm certifies that B forms part of U by computing common tangents between a portion of W and the rest of the upper hulls. Computing a common tangent between two upper hulls could be costly, but if there is a line separating them, then the time complexity is logarithmic [58]. The algorithm takes advantage of this fact by using as separating lines between such a portion of W and the rest of the upper hulls two tangents through p computed in the previous step: the one of minimum slope to the left of p and the one of maximum slope to the right of p (i.e., ignoring the portion of W in the same half plane as the other upper hulls). Block B is determined by the common tangents passing through the points nearest to p in W (one point to the left of p and the other one to the right). To avoid the computation of all common tangents, the algorithm interweaves the different tangent computations (similarly to how Demaine et al.'s algorithm [24] interweaves doubling searches to compute the intersection of sorted sets). We devote the rest of the section to describe this step in more details.

We describe how to identify the part of B to the right of p (the left counterpart is symmetric). Let λ be the tangent of maximum slope between p and the upper hulls to the right of p (i.e., the tangent of maximum slope among those computed in the previous step of the algorithm). Let W' be the portion of the upper hull W containing p above λ. The tangent λ separates W' from the rest of the upper hulls. Among the common tangents between W' and the upper hulls below λ, let τ be the one passing through the nearest point to p in W'. Let q and r be the points that lie in τ, such that q belongs to W' and r belongs to one of the upper hulls below λ. The point q determines the end of the right portion of B (see Figure 3.7 for a graphical representation of those definitions).

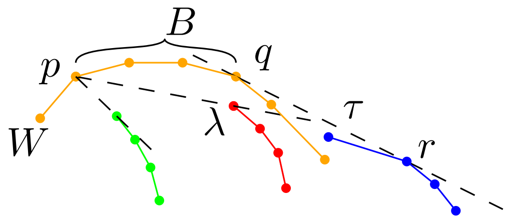


Figure 3.7: The state of algorithm `Quick Hull Union` during an execution of the step that computes the block B that forms part of the merged upper hull U. The upper hull W contains the point p. λ marks the tangent of maximum slope between p and the upper hulls to the right of p. τ marks the common tangent between the portion of W above λ and one of the upper hulls below λ passing through the point nearest to p in W. The points q and r lie in τ.

Given two upper hulls X and Y separated by a vertical line, Barbay and Chen [5] described an algorithm that computes the common tangent τ between them, in time within $O(\log a + \log b)$, where a and b are the positions of the points that lie in τ in the sequences of points representing X and Y, respectively. At each step this algorithm considers two points: one from X and the other from Y, and in at least one upper hull, it can certify, in constant time, if the point that lies in τ is to the right or to the left of the point considered. A minor variant manages the case where the separating line is not vertical: as the first step, in each upper hull, the algorithm computes the supporting line of slope equal to the slope of the separating line, by performing doubling searches.

To compute the point q that determines the right portion of B, the algorithm `Quick Hull Union` executes several instances of the algorithm described by Barbay and Chen [5] to compute the common tangents between W' and the upper hulls below λ, always considering the same point u in W' (λ is a separating line). Once all decisions about the point u in W' are reached, the upper hulls below λ can be divided into two sets: (i) those whose common tangents pass through a point to the left of u in W' and (ii) those whose common tangents pass through a point to the right of u in W'. If the set (i) is not empty, then the algorithm stops the computation in the set (ii). For each upper hull V in the set (ii), the algorithm discards the block of points to the left of the penultimate point considered in V (i.e., the algorithm can certify that this block lies underneath the common tangent between W' and V). This step continues until there is just one instance running, and computes the tangent τ in this instance. The algorithm discards all points to the left of r (i.e., all points that lie below the arc of the output that leaves W clockwise and follows τ).

After identifying the block B of the output, the algorithm recurses on the resulting upper hulls to the left and right of p. In the following section, we analyze the time complexity of algorithm `Quick Hull Union`.

3.3.2 Complexity Analysis of Algorithm Quick Hull Union

Similarly to the situation described in the analysis of the merging of maxima sets described in Section 3.2.2, each algorithm that solves the merging upper hulls problem must certify that some blocks of points in the upper hulls cannot participate in the merged upper hull U, and that some other blocks are indeed in U. We formalize the notion of a hull partition certificate, which can be used to check the correctness of the output in potentially less time than what would be required to recompute the output itself. A *hull partition certificate* of an instance of κ upper hulls is a partition of the points of the upper hulls into regions so that in each region, it is "easy" to certify whether the points form part of the output or not. This notion of hull partition certificate yields a measure of the difficulty of an instance ("short" partition certificates characterize "easy" instances, while "long" partition certificates suggest "difficult" instances). We define a language of basic arguments for such hull partition certificates: *eliminator* arguments justify the discarding of points from the input and *convex* arguments justify the presence of points in the output. A hull partition certificate is formed by eliminator and convex arguments and will be verified by checking each of its arguments. (See the two examples of arguments on a simple instance depicted in Figure 3.8.)

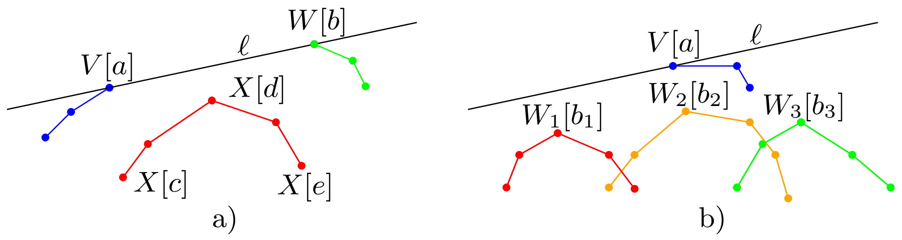


Figure 3.8: Example of arguments: a) an eliminator argument formed of 3 blocks and b) a convex argument formed of 4 blocks. In both arguments, note that the blocks of V end or start in $V[a]$, because the algorithm only needs to specify the position a inside V.

Given an upper hull V, let $V[a]$ denote the a-th point in V, and let $V[b..c]$ denote the block of $c-b+1$ consecutive points corresponding to the positions from b to c in V, respectively.

Definition 10. *Consider the upper hulls V, W, and X. Let ℓ be the straight line through the points $V[a]$ and $W[b]$. $\langle V[a], W[b] \supset X[c..d..e]\rangle$ is an* eliminator argument *if the points of the block $X[c..e]$ are between the vertical lines through $V[a]$ and $W[b]$, the slope of ℓ is between the slopes of the two edges in X that precede and follow the point $X[d]$, and the point $X[d]$ lies below ℓ.*

If $\langle V[a], W[b] \supset X[c..d..e]\rangle$ is an eliminator argument, then the points of the block $X[c..e]$ cannot contribute to the merged upper hull.

Several blocks that are "eliminated" by the same pair of points can be combined into a single argument. Those eliminator arguments are the ones used in the Steps 4 and 6 of Algorithm 3 to discard points that cannot be part of the merged upper hull.

It is not enough to discard some points that do not contribute to the output. Certifying that the remaining points cannot be discarded still requires additional work: a correct algorithm must justify the exactness of its output. To this end we define convex arguments.

Definition 11. *Consider the upper hulls $V, W_1, \dots, W_t$. $\langle V[a] \dashv W_1[b_1], \dots, W_t[b_t]\rangle$ is a* convex argument *if there is a straight line ℓ through $V[a]$ such that the slope of ℓ is between the slopes of the edges that precede and follow the points $W_1[b_1], \dots, W_t[b_t]$, respectively, and the points $W_1[b_1], \dots, W_t[b_t]$ lie below ℓ.*

If $\langle V[a] \dashv W_1[b_1], \dots, W_t[b_t]\rangle$ is a convex argument, then the point $V[a]$ is a vertex of the merged upper hull of $V, W_1, \dots, W_t$. Blocks of consecutive points can also be "easily" certified as part of the output using similar arguments: when the first and last points p and q, respectively, in such blocks are vertices of the merged upper hull U, and all the other points lie below the line through p and q. Those convex arguments are the ones used in Step 5 of Algorithm 3 to certify that blocks of consecutive points in the upper hulls form part of U.

Those arguments are a two-dimensional generalization of the arguments described by Demaine et al. [24] for merging sorted sets, and are inspired by the ones introduced by Barbay and Chen [5] for the binary merging upper hulls. Those arguments combine into a general definition of hull partition certificate that any correct algorithm for merging upper hulls in the algebraic decision tree model can be modified to output without increasing the magnitude of its computational complexity. In particular, the algorithms described by Kirkpatrick and Seidel [48] and Levcopoulos et al. [50] implicitly compute such partition certificates. See Figure 3.9 for an example of such a partition certificate.

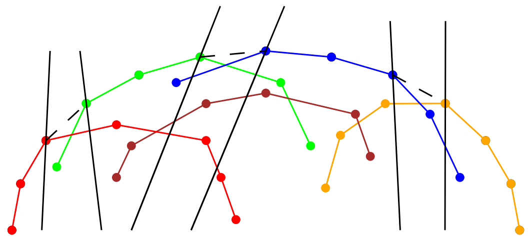


Figure 3.9: A hull partition certificate of size 7 of an instance of the merging upper hulls problem. The thick black lines mark the division between the 7 regions.

Definition 12. *Given an instance I of the merging upper hulls problem, a* hull partition certificate *of I is a partition of the points into regions, so that in each region, the points of I that belong to the output can be decided using a constant number of eliminator and convex arguments. The* size *of I is the number of regions which compose it.*

The algorithm `Quick Hull Union` partitions the upper hulls into blocks of consecutive points, where each block is either discarded or output. A block is discarded if it is underneath the merged upper hull U, or is output if it forms part of U. Each of such blocks forms part of an argument of the hull partition certificate computed by the algorithm. Similarly to the algorithm `Quick Maxima Union`, we separate the analysis into three lemmas. In Lemma 9, we analyze the steps that identify the "pivot" point p, that partition the upper hull by the vertical line through p, and that discard blocks of points (i.e., Steps 2, 3, 4, and 6 of Algorithm 3). In Lemma 10, we analyze the step that outputs blocks of points (i.e., Step 5 of Algorithm 3). In Lemma 11, we analyze the step that computes the medians of the slopes of the edges at the middle positions (i.e., Step 1 of Algorithm 3).

Lemma 9. *Given an upper hull V, the cumulated time complexity of the steps identify the "pivot" points p, that partition the upper hull by the vertical line through p, and that discard blocks of points of algorithm* ***`Quick Hull Union`*** *considering only points of V is within $O(\sum_{j=1}^{\beta}\log s_j)$, where $s_1,\dots,s_\beta$ are the sizes of the β blocks into which the algorithm partitions V.*

Proof. The analysis is as in the proof of Lemma 5. □

The blocks that are discarded in each execution of the Steps 4 and 6 of Algorithm 3 are certified using a single eliminator argument. In the same way, the block that is output in Step 5 of Algorithm 3 is certified using a single convex argument. We state the following lemmas in function of the hull partition certificate computed by the algorithm.

Lemma 10. *Given a block B that forms part of the merged upper hull U, the time complexity of the step that outputs B of algorithm* ***`Quick Hull Union`*** *is within $O(w\log s_B)$, where s_B is the size of B and w is the number of arguments in the convex argument used by the algorithm to certify that B forms part of U.*

This analysis is a consequence of the w searches for the common tangent in Step 5 of Algorithm 3. The amount of arguments in the hull partition certificate and the number of blocks in each of the arguments both are related to the time complexity of Step 1 of Algorithm 3.

Lemma 11. *Given κ upper hulls, the cumulated time complexity of the steps that compute the medians of the slopes of the edges at the middle positions of the upper hulls of algorithm* ***`Quick Hull Union`*** *is within $O(\sum_{i=1}^{\delta}\log\binom{\kappa}{m_i})$, where δ is the size of the hull partition certificate C computed by the algorithm, and $m_1,\dots,m_\delta$ is a sequence where m_i is the number of blocks in the i-th argument of C.*

Proof. We prove this lemma by induction over δ and κ. The time complexity of each of these steps is linear in the number of upper hulls in the subinstance (i.e., ignoring all the empty upper hulls of this subinstance). Let $\mathcal{T}(\delta,\kappa)$ be the cumulated time complexity of these steps. We prove that $\mathcal{T}(\delta,\kappa)\le\sum_{i=1}^{\delta}m_i\log\frac{\kappa}{m_i}-\kappa$, where m_i is the number of blocks that form the i-th argument of C. Let p be the first "pivot" point computed by the algorithm. Once the tangents between p and the upper hulls have been computed, let c and d be the number of upper hulls that have non-discarded point only to the left of p and only to the right of p, respectively. Let b be the number of upper hulls that have non-discarded points to the left and to the right of p. Let δ_c and δ_d be the number of arguments computed by the algorithm to the left and to the right of p, respectively. Then, $\mathcal{T}(\delta,\kappa)=\mathcal{T}(\delta_c,c+b)+\mathcal{T}(\delta_d,d+b)+\kappa$ because of the two recursive calls and one execution of the Step 1 of Algorithm 3. By Induction Hypothesis, $\mathcal{T}(\delta_c,c+b)\le\sum_{i=1}^{\delta_c}m_i\log\frac{c+b}{m_i}-c-b$ and $\mathcal{T}(\delta_d,d+b)\le\sum_{j=1}^{\delta_d}m_j\log\frac{d+b}{m_j}-d-b$. We need to prove that $c+d\le\sum_{i=1}^{\delta_c}m_i\log\left(1+\frac{d}{c+b}\right)+\sum_{j=1}^{\delta_d}m_j\log\left(1+\frac{c}{d+b}\right)$, but this is a consequence of $\sum_{i=1}^{\delta_c}m_i\ge c+b,\sum_{j=1}^{\delta_d}m_j\ge d+b$ (the number of blocks is greater than or equal to the number of upper hulls); in the worst case, $c\le d+b, d\le c+b$ (at least $\frac{\kappa}{2}$ sequences are left to the left and to the right of p); and $\log\left(1+\frac{y}{x}\right)^x\ge y$ for $y\le x$. □

Combining Lemmas 9, 10, and 11 yields the following theorem:

Theorem 3. *Given an instance I formed by κ upper hulls, the algorithm* `Quick Hull Union` *computes the merged upper hull U of I in time within*

$$O\left(\sum_{j=1}^{\beta} \log s_j + \sum_{k=1}^{\delta} w_k \log n_k + \sum_{i=1}^{\delta} \log \binom{\kappa}{m_i}\right);$$

where β is the number of blocks in the hull partition certificate C computed by the algorithm; $s_1, \dots, s_\beta$ are the sizes of such blocks; δ is the size of C; and $m_1, \dots, m_\delta$ is a sequence where m_i is the number of blocks that form the i-th argument of C; w_k is the number of arguments in the k-th convex argument; and n_k is the size of the k-th block of U.

In the context of computing the convex hull, we prove in Section 4.3.2 that a synergistic algorithm based on that result asymptotically outperforms both the algorithm originally proposed by Kirkpatrick and Seidel [48] (even when taking into account the refined analysis described by Afshani et al. [1]), and the algorithm proposed by Levcopoulos et al. [50]. We depict a family of instances where the synergistic algorithm is faster than those previous algorithms by a factor logarithmic in the size of the input (Ex. 8 page 55). We describe an analysis of algorithm `Quick Hull Union` in function of the smallest possible size δ of a partition certificate for a particular instance.

Theorem 4. *Given an instance I formed by κ upper hulls of sizes $r_1, \dots, r_\kappa$ such that the merged upper hull U admits a hull partition certificate of size δ, the algorithm* `Quick Hull Union` *computes U in time within*

$$O\left(\kappa\delta \log \frac{h}{\delta} + \delta \sum_{i=1}^{\kappa} \log \frac{r_i}{\delta}\right),$$

where h is the number of points in U.

Proof. The size of the hull partition certificate C computed by the algorithm `Quick Hull Union` in the instance I is a constant factor of the size δ of a hull partition certificate P of minimal size for I, such that in each region there is just one block that forms part of the output, and such a block can be certified using a single convex argument. Indeed, if a region R of P contains a block B that forms part of the merged upper hull U, then the algorithm `Quick Hull Union` can certify that B forms part of U using a constant number of arguments. This is a consequence of the step that computes the blocks that form part of the output (i.e., Step 5) of algorithm `Quick Hull Union`. This step computes the block of maximum size (p included) that can be certified that forms part of the output using a single convex argument. In addition, the algorithm partitions each upper hull in at most a constant factor of δ blocks. Combining the results from Lemmas 9, 10, and 11 with the concavity of the logarithm function (i.e., $\sum_{j=1}^{\beta} \log s_j \leq \beta \log \frac{\sum_{j=1}^{\beta} s_j}{\beta}$), we obtain that the time complexity of the algorithm is within $O(\sum_{i=1}^{\kappa}\sum_{j=1}^{\delta} s_{ij} + \sum_{k=1}^{\delta} w_k \log n_k) \subseteq O(\delta \sum_{i=1}^{\kappa} \log \frac{r_i}{\delta} + \kappa\delta \log \frac{h}{\delta})$, where s_{ij} is the size of the j-th block of the i-th upper hull, w_k is the number of arguments in the k-th convex argument, and n_k is the size of the k-th block of U. □

We conclude this chapter by discussing potential improvement to our results.

3.4 Discussion

We described two instance optimal algorithms for merging sorted sets: `Sorted Set Union` (based on `MergeSort`) and `Quick Set Union` (based on `QuickSort`). The computational cost of instance optimal algorithms is within a constant factor of the optimal in every instance, as opposed to just the worst case.

Concerning the merging of maxima sets, we described and analyzed the algorithm `Quick Maxima Union`. We proved an upper bound for the number of coordinate comparisons performed by the algorithm `Quick Maxima Union` in terms of a minimal size decomposition of the maxima sets into blocks of consecutive points, which is determined by a maxima partition certificate of the instance. Such decomposition provides a measure of difficulty to evaluate the efficiency of any algorithm. We proved that the algorithm `Quick Maxima Union` is optimal in the worst case over instances with the same value for this particular measure of difficulty, in the decision tree model (Theorem 2). A way to improve on this is to design an instance optimal algorithm for merging maxima sets or to prove that such instance optimal algorithms do not exist for this problem. The main issue is to find, for each instance, the best of such decompositions (i.e, the one that can be encoded in the minimum size) in time proportional to the shortest binary encoding of it.

Following similar ideas to the ones stated for the merging of maxima sets, it is possible to prove an almost identical lower bound for the time complexity of any algorithm that merges convex hulls in the algebraic decision tree model. But our analyses of the time complexity of algorithm `Quick Hull Union` do not match such a lower bound (Theorems 3 and 4). The difference between the upper and lower bounds is related to the analysis of the step that finds the blocks of the convex hulls that form part of the output (Lemmas 9 and 10). We conjecture that there is no way to improve the computation of such blocks, which suggests that a better lower bound is needed.

This concludes our results about adaptive merging. In the following section, we combine those merging algorithms with algorithms that partition the input sequence into "easy" instances, to obtain synergistic algorithms for sorting multisets, and for computing maxima sets and convex hulls of planar points.

Chapter 4

Synergistic Computation Analysis

The final piece of the puzzle in our approach to obtain synergistic algorithms for sorting multisets, and for computing maxima sets and convex hulls of planar points, is to design algorithms that partition the input sequence of elements into "easy" subsequences. Our definitions of "easy" subsequences are determined by measures of difficulty that depend on the order into which the elements are listed in the sequence, and by the fact that the computation in such "easy" subsequences can be done in time linear in the sizes of the subsequences (Sections 4.1.1, 4.2.1, and 4.3.1). We devote this chapter to the study of such partitioning algorithms, and their combination with the merging algorithms described in Chapter 3 into synergistic algorithms for those three problems. Those combinations provide us with a better understanding of the problems and more efficient solutions (Sections 4.1.2, 4.2.2, and 4.3.2).

Concerning the sorting of multisets, in addition to the runs (a notion of local input order which we described in Section 2.3.2), we develop a new technique that takes advantage of the pivot positions (i.e., positions p in the multiset such that all the elements to the left of p are smaller than or equal to all the elements to the right of p, a notion of global input order) when partitioning the sequence of numbers that represents a multiset (Section 4.1.1).

Concerning the computation of the maxima set of planar points, we define an algorithm that partitions a sequence of points into smooth subsequences, such that the maxima sets of the smooth subsequences can be computed in time linear in the sizes of the subsequences (Section 4.2.1). The definition of smooth sequence is related to the lexicographic order of the points in the sequence (see Definition 14 page 50).

Concerning the computation of the convex hull of planar points, we improve Levcopoulos et al.'s partitioning algorithm [50] (described in Section 2.4.2) by presenting a linear time algorithm that partitions a polygonal chain P of n points into simple polygonal chains. The entropies of the distributions of the sizes of the simple polygonal chains of the partitions obtained by both algorithms are asymptotically the same. We refine Levcopoulos et al.'s measure of difficulty and analysis of the computation of the convex hull in terms of the entropy of the distribution of the sizes of such simple polygonal chains (Section 4.3.1).

We built upon previous results taking advantage of either some notions of order, or the input structure, to describe solutions which take advantage of both in a synergistic way. The ways to take advantage of the input structure are limited [56, 1], but in Section 4.4, we discuss other possible measures of the input order.

4.1 Sorting Multisets

In our search to find better partitioning techniques to speed up the sorting of multisets, we define a simple yet new notion of global input order, formed by the number of pre-existing pivot positions in the input (i.e., positions p in the multiset such that all the elements to the left of p are smaller than or equal to all the elements to the right of p), not mentioned in previous surveys [28, 54] nor extensions [7] (Section 4.1.1).

We show examples of the difficulty of combining both some measures of the input order and the input structure of a multiset into a single synergistic algorithmic technique for sorting it. But we describe synergistic sorting algorithms, which never perform worse than `MergeSort with Counters` and `Minimal MergeSort` (previously defined in Section 2.3), and outperform both of them on some large classes of instances, by taking advantage both of some measures of the input order (local and global) and the input structure, in a synergistic way (Section 4.1.2).

4.1.1 A Measure of Global Input Order

Given a multiset M, Iverson [41] defined pivot position as a position p in M such that all the elements in previous positions are smaller than or equal to all the elements at p or in the following positions (e.g., $(3, 2, 1, 6, 5, 4)$ has one pivot position in the middle between 1 and 6). In 1962, Iverson [41] described an improved version of `BubbleSort` [49] that identifies such pivot positions (as pairs of consecutive elements that the algorithm have placed at their final positions and on which it does not make further comparisons). In the following section, 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 13. *Given a multiset $M = \langle x_1, \dots, x_n \rangle$ of size n, a* pivot position *in M is a position $p \in [2..n]$ 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 M divide it into subsequences of consecutive elements such that the range of positions of the elements at each subsequence coincides with the range of positions of the same elements in the sorted sequence of M: the more there are of such positions, the more "global" order there is in M (as opposed to "local" measures as the runs). Detecting such positions takes only a linear number of comparisons by applying the first phase of algorithm `BubbleSort` [49], which sequentially compares the elements from left to right, in a first step, and then a modification of such phase from right to left, in a second step. The positions of the elements that do not interchange their values during both executions are the pivot positions in M.

Lemma 12. *Given a multiset M of size n with ϕ pivot positions $p_1, \dots, p_\phi$, the ϕ pivot positions can be detected using $2n - 2$ comparisons.*

Proof. Consider the first phase of algorithm `BubbleSort` [49], which sequentially compares the elements in positions $i - 1$ and i of M, for $i \in [2..n]$: we name it the `bubble-up` step. If $M[i-1] > M[i]$, then the elements interchange their values. As consequence of this step the elements with large values tend to move to the right. In an execution of the `bubble-up` step in M, the elements that do not interchange their values are those elements whose values are greater than or equal to all the elements on their left.

We define the `bubble-down` step similarly to the `bubble-up` step, with the difference that it scans the sequence from right to left, interchanging the elements in positions $i - 1$ and

i if $M[i-1] > M[i]$, for $i \in [2..n]$. In an execution of the bubble-down step in M, the elements that do not interchange their values are those elements whose values are smaller than or equal to all the elements on their right. Hence, the positions of the elements that do not interchange their values during the executions of both the bubble-up and bubble-down steps are the pivot positions in M. □

The ϕ pivot positions simply divide a multiset of size n into $\phi + 1$ subinstances of sizes $n_0, \ldots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Each of such subinstances I_i for $i \in [0..\phi]$ has its own number of runs ρ_i and alphabet size σ_i, on which the synergistic solutions described in this work can be applied, from merely sorting multiset (Section 4.1.2) to supporting multiselection (Section 5.1.1) and to the more sophisticated deferred data structures (Section 5.1.2).

4.1.2 Synergistic Sorting

We start this section by describing examples that prove the difficulty of measuring the interaction of some forms of input order (i.e., runs and pivot positions) and the input structure (i.e., the distribution of the multiplicities of the distinct elements). We describe then an algorithm that takes advantage both of some measures of the input order and the input structure, at once, in a synergistic way.

Comparison Between Sorting Algorithms

In Section 2.3.3, we showed that the worst-case performances of the algorithms MergeSort with Counters (described in Section 2.3.1) and Minimal MergeSort (described in Section 2.3.2) are incomparable, in the sense that neither one performs always better than the other in the worst case over instances of size n. We describe here two more sophisticated algorithms, which are modifications and combinations of the algorithms Minimal MergeSort and MergeSort with Counters, that still do not take full advantage both of some measures of the input order (local and global) and the input structure. So, a new approach will be necessary to take better advantage both of some measures of the input order and the input structure.

Algorithm Dovetailing MergeSort is a dovetailing combination of the algorithms Minimal MergeSort and MergeSort with Counters. This algorithm runs both Minimal MergeSort and MergeSort with Counters in parallel, and when one of them manages to sort the multiset, it returns the sorted sequence and then finishes. The number of comparisons performed by algorithm Dovetailing MergeSort in any instance I is twice the minimum of the number of comparisons performed by the algorithms Minimal MergeSort and MergeSort with Counters on I. Algorithm Dovetailing MergeSort also needs to duplicate the input in order to run both algorithms in parallel.

Combining the ideas of identifying and merging runs from Takaoka [63] with the use of counters by Munro and Spira [56], we describe the Small vs Small algorithm to sort a multiset. The algorithm Small vs Small identifies the runs using the same linear scanning as Minimal MergeSort, and associates counters to the elements in the same way that MergeSort with Counters does. Once the runs are identified, this algorithm initializes a heap with the runs ordered by sizes. At each merging step the two shortest runs are selected for merging and both are removed from the heap. The pair is merged, and the resulting run is inserted into the heap. The process is repeated until only one run is left and the sorted sequence is known. The complexity of the `Small vs Small` algorithm is adaptive to the sizes of the resulting runs in the merging process.

Given an instance I, the number of comparisons performed by the algorithm `Small vs Small` on I is never worst than a constant factor of the number of comparisons performed by the algorithm `Dovetailing MergeSort` on I. But the next example shows that there are families of instances where the number of comparisons performed by the algorithm `Small vs Small` is asymptotically less than the number of comparisons performed by the algorithm `Dovetailing MergeSort` by a factor logarithmic in the size of the input.

> **Example 5.** *Consider the family of instances* $\langle 1, 2, \ldots, \sigma, 1, 2, \ldots, \sigma \ldots, 1, 2, \ldots, \sigma \rangle$
>
> Consider a family of instances formed by ρ runs, each of size σ. The number of comparisons performed by algorithm `MergeSort with Counters` for sorting each of such instances is within $\Theta(\rho\sigma \log \sigma)$, while the number of comparisons performed by algorithm `Minimal MergeSort` for sorting each of such instances is within $\Theta(\rho\sigma \log \rho)$. On the other hand, the number of comparisons performed by algorithm `Small vs Small` for sorting each of such instances is better, within $\Theta(\rho\sigma)$: at each level of the binary tree representing the merging order, the sum of the sizes of the runs is halved.

Even though algorithm `Small vs Small` is adaptive to the sizes of the resulting runs, it does not take advantage of the fact that there may exist a pair of runs that can be merged very quickly, but it rather pairs one of such runs with another run of the same size, taking time linear in the sum of the sizes to merge them. We show this disadvantage in Example 6, where we compare algorithm `Small vs Small` with the new synergistic algorithm `Quick Synergy Sort`, which we describe in the following section.

Quick Synergy Sort Algorithm

Given a multiset M, the algorithm `Quick Synergy Sort` identifies the runs of M in linear time through a scanning process, and merges the runs using the algorithm `Quick Set Union` (described in Section 3.1.2). The following theorem bounds the number of comparisons performed by the algorithm `Quick Synergy Sort` in terms of the block decomposition of the runs of M (as seen in Section 3.1.1):

Theorem 5. *Let M be a multiset of size n formed by ρ runs, σ distinct elements, and δ blocks of sizes $g_1, \ldots, g_\delta$ such that those blocks induce a partition certificate π of size χ of the output whose members have values $m_1, \ldots, m_\chi$. The algorithm* ***`Quick Synergy Sort`*** *sorts M performing a number of data comparisons*[1] *within*

$$n + O\left(\sum_{i=1}^{\delta} \log g_i + \sum_{i=1}^{\chi} \log \binom{\rho}{m_i}\right) \subseteq O(n \log(\min\{\rho, \sigma\})) \subseteq O(n \log n).$$

There are families of instances where algorithm `Quick Synergy Sort` performs significantly better than algorithm `Small vs Small` (described above). Consider for instance the following example:

[1]Note that the quantity $\sum_{i=1}^{\delta} \log g_i$ is within $O(n)$ but is much smaller than n for "easy" instances.

Example 6. *Consider the family of intances* $\left\langle \frac{\rho-1}{\rho}n+1,\ldots,n,\frac{\rho-2}{\rho}n+1,\ldots,\frac{\rho-1}{\rho}n,\ldots,1,\ldots,\frac{1}{\rho}n \right\rangle$
Consider a family of instances formed by ρ runs of size $\frac{n}{\rho}$ each. The runs are pairwise disjoint and the elements of each run are consecutive in the output. The number of comparisons performed by the algorithm `Small vs Small` for sorting each of such instances is within $\Theta(n \log \rho)$, while the number of comparisons performed by the algorithm `Quick Synergy Sort` is within $\Theta\left(n + \rho\log\rho + \rho\log\frac{n}{\rho}\right) = \Theta(n + \rho\log n)$ (which is better than $\Theta(n\log\rho)$ for $\rho \in o(n)$).

We extend the result described in Theorem 5 to take advantage of the number of existing pivot positions in the multiset M (i.e., a form of global input order) in a way that can be combined with the notion of runs (i.e., a form of local input order), and with the distribution of the multiplicities of the distinct elements in M (i.e, the input structure).

Corollary 1. *Let M be a multiset of size n with ϕ pivot positions. The ϕ pivot positions divide M into $\phi+1$ subinstances of sizes $n_0,\ldots,n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Each subinstance I_i for $i \in [0..\phi]$ of size n_i is formed by ρ_i runs, σ_i distinct elements, and δ_i blocks of sizes $g_{i1},\ldots,g_{i\delta_i}$. Such blocks induce a partition certificate π_i of size χ_i of the output whose members have values $m_{i1},\ldots,m_{i\chi_i}$. There is an algorithm that sorts M performing a number of comparisons within*

$$3n + O\left(\sum_{i=0}^{\phi}\left\{\sum_{j=1}^{\delta_i}\log g_{ij} + \sum_{j=1}^{\chi_i}\log\binom{\rho_i}{m_{ij}}\right\}\right) \subseteq O\left(\sum_{i=0}^{\phi} n_i \log(\min\{\rho_i,\sigma_i\})\right) \subseteq O(n\log n).$$

In the following section, we generalize the concept of runs (i.e., a form of local input order) in multisets to maxima sets by describing sequences of points where the computation of the maxima sets of such sequences can be done in linear time, and we describe a synergistic algorithm that takes advantage both of such new measure of the input order and of the input structure as defined by Kirkpatrick and Seidel [47], in a synergistic way.

4.2 Maxima Sets

Similarly to the sorting of multisets, we describe first a linear time algorithm that partitions a sequence of points into "easy" subsequences, where the maxima set of such subsequences can be computed in linear time. The number and sizes of the "easy" subsequences depend on the order in which the points are listed (Section 4.2.1). We combine then such partitioning algorithm with the merging maxima sets algorithm, described in Section 3.2, into an algorithm that computes maxima sets in the plane, taking advantage both of some measure of the input order and the input structure, in a synergistic way (Section 4.2.2).

4.2.1 Input Order Adaptivity

In many cases, the maxima set can be computed in time linear in the size of the input, independently from its size. For instance, consider an order of the points where (1) the maximal points are given in order sorted by one coordinate, and (2) for each maximal point

p, all the points dominated by p are given immediately after p (in any relative order). This order has similar properties to the lexicographical order regarding the computation of the maxima set, but is less restricted. The maxima set of a sequence of points given in this order can be extracted and validated in linear time by a simple greedy algorithm, which throws an exception if the input is not in such an order. Each of the various ways to deal with such exceptions directly yields an algorithm adaptive to some measure of the input order [2]. For example, consider an instance such that the points found to be out of order are inserted in the partial maxima set M computed up to each of such points using a constant number of coordinate comparisons or that the points found to be out of order dominate one (or both) of the end points of M. Let us label such a sequence *smooth*, and by extension any input subsequence of consecutive positions which have the same property:

Definition 14. *Let $S = p_1, \ldots, p_n$ be a sequence of n planar points. For $i \in [1..n]$, let M_i be the maxima set of the points $p_1, \ldots, p_i$. If, for all $i \in [1..n-1]$, the point p_{i+1} dominates one (or both) of the end points of M_i or p_{i+1} is identified that it is dominated by some of the points of M_i using a constant number of coordinate comparisons, then we say S is* smooth.

Given a sequence S of points, let ζ denote the minimal number of smooth subsequences into which S can be decomposed. Most interestingly for our synergistic purpose, such a decomposition can be computed in time linear in the size of S. We name the algorithm that detects such ζ smooth subsequences, and computes their individual maxima sets `Smooth Partitioning`.

The algorithm `Smooth Partitioning` considers the points of S one by one, and fixes a constant c. Let p be the current point considered. At each step, the algorithm maintains the maxima set M up to this point of the smooth sequence to which p belongs. If p dominates one (or both) of the end points of M, then M is updated by inserting p and deleting all the points of M dominated by p. If p is identified as dominated by some of the points of M using at most c coordinate comparisons, then the maxima set up to p of the smooth sequence to which p belongs is M. Otherwise, a new smooth sequence starting at p is detected.

Using the algorithm `Smooth Partitioning` to detect the minimum number ζ of smooth subsequences into which the sequence of points can be partitioned and to compute their maxima sets, then merging such maxima sets two by two yields an algorithm that performs within $O(n(1+\log\zeta)) \subseteq O(n\log n)$ coordinate comparisons. Such a result is orthogonal to previous input structure adaptive results [1, 47] described in Section 2.4.1: it can be worse than $O(n(1+\log h))$ when the output size h is small and the input is in a "bad" order, and it can be much better than $O(n(1+\log h))$ when h is large and the input is in a "good" order.

In the following, we design an algorithm which never performs asymptotically worse than those algorithms (whose number of coordinate comparisons are within $O(n(1+\log\zeta))$ and $O(n(1+\log h))$), and, in large classes of instances, asymptotically outperforms both by a factor logarithmic in the size of the instance (i.e., by more than a constant factor).

4.2.2 Synergistic Computation of Maxima Sets

The algorithm `Quick Synergy Maxima` decomposes the planar points into the minimal number ζ of smooth subsequences and computes their maxima sets using the algorithm `Smooth Partitioning` (described above), and then merges such maxima sets using the algorithm `Quick Maxima Union` (described in Section 3.2). The algorithm `Quick Synergy Maxima` takes advantage of the number of smooth subsequences and the distribution of their sizes, and of the block decomposition of the maxima sets of the smooth subsequences as described in Section 3.2, and hence of the output size. We prove this more formally in the following theorem:

Theorem 6. *Let S be a sequence of planar points which can be partitioned into ζ smooth sequences. Let h be the number of points in the maxima set M of S. The algorithm* `Quick Synergy Maxima` *computes M performing a number of coordinate comparisons*[2] *within*

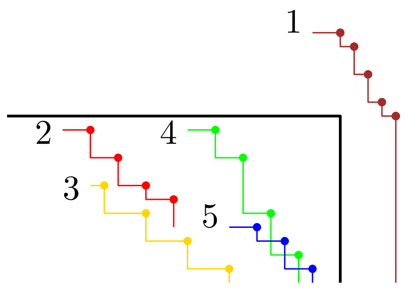
$$2n + O\left(\sum_{j=1}^{\beta} \log s_j + \sum_{i=1}^{\delta} \log \binom{\zeta}{m_i}\right) \subseteq O(n(1+\log(\min\{\zeta, h\}))) \subseteq O(n \log n);$$

where β is the number of blocks of sizes $s_1, \ldots, s_\beta$, respectively, in the maxima partition certificate C of the maxima sets of such ζ smooth sequences computed by the merging algorithm; δ is the size of C (such that $\delta \leq h$); and $m_1, \ldots, m_\delta$ is a sequence where m_i is the number of blocks in the i-th argument of C.

Proof. This is a consequence of Theorem 2 and of the linear execution time of the partitioning algorithm `Smooth Partitioning`. □

On large classes of instances, the `Quick Synergy Maxima` algorithm outperforms both the Kirkpatrick and Seidel's algorithm [47] as well as the combination of the algorithm `Smooth Partitioning` with a binary merging algorithm (similar to the `Small vs Small` algorithm described in Section 4.1.2 for sorting multisets). Also it never performs asymptotically worse than those algorithms.

Example 7. *A sequence S of planar points that can be partitioned into $\zeta = 5$ smooth subsequences. The numbers indicate the order into which the sequence of points are given: each from left to right internally. Every point in the maxima set 1 dominates all the points in the other maxima sets. The other maxima sets are more intricately woven together.*



The number of coordinate comparisons performed by the Kirkpatrick and Seidel's algorithm [47] on such instance is within $\Theta(n(1+\log h))$, where h is the number of points in the maxima set labeled 1. The refined analysis described by Afshani et al. [1] of the Kirkpatrick and Seidel's algorithm yields a number of coordinate comparisons within $\Theta(h \log n)$. The number of coordinate comparisons performed by the combination of the `Smooth Partitioning` algorithm with a binary merging algorithm is within

[2]Note that the quantity $\sum_{j=1}^{\beta} \log s_j$ is within $O(n)$ but is much smaller for "easy" instances.

$\Theta(n(1+\log\zeta))$, where ζ is the number of smooth subsequences. On the other hand, the number of coordinate comparisons performed by the algorithm `Quick Synergy Maxima` is within $\Theta(n)$: once it computes the first point of the maxima set of S, it discards all the points except the points in the maxima set labeled 1. If $h \in \Theta(n)$ and $\zeta \in \Theta(n)$, then the algorithm `Quick Synergy Maxima` is faster than those algorithms by a factor logarithmic in the size of the input.

The histories of the computation of the maxima set and of the convex hull are strongly correlated: most of the results on one problem also generalize to the other. Our results on the computation of the maxima set similarly generalize to the computation of the convex hull, albeit with some quirks.

4.3 Convex Hulls

Similarly to sorting multiset and computing maxima sets, we describe how to partition a sequence of points into simple polygonal chains in linear time (Section 4.3.1). This notion of input order (i.e., the decomposition into simple polygonal chains) for the computation of the convex hull is less restrictive than the one seen for the computation of the maxima set (i.e., the decomposition into smooth subsequences), in the sense that it can consider more sophisticated sequences as "easy" sequences.

We combine then such linear time partitioning algorithm with the merging convex hulls algorithm described in Section 3.3, into a synergistic algorithm for computing the convex hull of planar points (Section 4.3.2). Such synergistic algorithm takes advantage both of the decomposition of the points into simple polygonal chains (i.e., a form of input order), and of the relative positions of the points (i.e., the input structure), in a synergistic way.

4.3.1 Input Order Adaptivity

Levcopoulos et al. [50] described an algorithm that computes the convex hull of a polygonal chain S of n planar points, taking advantage of the minimal number κ of simple polygonal chains into which S can be partitioned (see Section 2.4.2 for details). The time complexity of the partitioning and merging steps of such algorithm are both within $\Theta(n(1+\log\kappa)) \subseteq \Theta(n\log n)$. We describe an improved partitioning algorithm running in time linear in the number of points, which is key to the synergistic result of Theorem 8. The entropy of the distribution of the sizes of the partition obtained by such linear time partitioning algorithm is asymptotically the minimum among all the partitions of S into simple polygonal chains. As a consequence, we describe a new algorithm that computes convex hulls whose analysis takes into account the relative imbalance between the sizes of the simple polygonal chains (Corollary 2). This new convex hull algorithm makes it possible, among other improvements, to compute the convex hulls of some instances in linear time, even though the minimum number κ of simple polygonal chains into which such instances can be partitioned is logarithmic in the number of points (something that the algorithm described by Levcopoulos et al. [50] does not permit).

Linear Time Partitioning Algorithm

Algorithm 4 `Galloping Partition`

```
Input: A sequence of $n$ planar points $p_1, \dots, p_n$
Output: A sequence of simple polygonal chains
1: Initialize $i$ to 1
2: for $t = 1, 2, \dots$ do
3:     if $i + 2^t - 1 > n$ or the chain $p_i, \dots, p_{i+2^t-1}$ is not simple then
4:         Output the chain $p_i, \dots, p_{i+2^{t-1}-1}$
5:         Update $i \leftarrow i + 2^{t-1}$ and $t \leftarrow 1$
6:     end if
7: end for
```

The algorithm `Galloping Partition` searches one by one for the largest integer t such that the polygonal chain formed by the first 2^t points is simple. It identifies this polygonal chain as simple and restarts the computation in the rest of the points. Its pseudocode is described in Algorithm 4. It identifies a simple polygonal chain of size k in time within $O(k)$, because the sizes of the tested chains form a geometric progression of ratio 2. The time complexity of such algorithm is linear in the number n of points, but we prove that the entropy of the distribution of the sizes of the resulting k simple polygonal chains is within a constant factor of the entropy of the distribution of the sizes of any[3] partition of the sequence of n points into the minimum possible number κ of simple polygonal chains:

Theorem 7. *Given a polygonal chain P of n planar points, the algorithm* ***`Galloping Partition`*** *computes in linear time a partition of P into k simple polygonal chains of sizes $r_1, \dots, r_k$, such that $n(1 + \mathcal{H}(r_1, \dots, r_k)) \in O(n(1 + \alpha))$, where α is the minimum value for the entropy function $\mathcal{H}(s_1, \dots, s_\kappa)$ of any partition of P into κ simple polygonal chains, of respective sizes $s_1, \dots, s_\kappa$.*

Proof. Consider a partition π of P into κ simple polygonal chains of sizes $s_1, \dots, s_\kappa$. Fix the polygonal chain c_i of size s_i. The polygonal chain c_i contributes $\frac{s_i}{n} \log \frac{n}{s_i}$ to the value of $\mathcal{H}(s_1, \dots, s_\kappa) = \sum_{i=1}^{\kappa} \frac{s_i}{n} \log \frac{n}{s_i}$. The algorithm `Galloping Partition` partitions c_i into simple polygonal chains (if the ends of c_i are blended with other chains of π consider them as separated chains, this does not decrease the entropy of the partition obtained by the algorithm). One of such chains is at least of size $\frac{s_i}{2}$, and in the worst case, the sizes of the rest of them form a decreasing geometric progression of ratio $\frac{1}{2}$. Hence, the chains into which the algorithm partitions c_i contribute $O(\sum_{i=1}^{\infty} \frac{s_i}{2^i} \log \frac{2^i n}{s_i}) = O(s_i + \frac{s_i}{n} \log \frac{n}{s_i})$ to the entropy of the partition obtained by the algorithm. Summing over all chains, the result follows. □

Given the convex hulls of the simple polygonal chains obtained by the algorithm `Galloping Partition`, an algorithm that merges two by two the shortest ones takes advantage of the potential disequilibrium in the distribution of their sizes, a result that improves upon the algorithm described by Levcopoulos et al. [50]:

Corollary 2. *Given a polygonal chain P of n planar points that can be partitioned into κ simple polygonal chains of respective sizes $r_1, \dots, r_\kappa$, there is an algorithm that computes the convex hull of P in time within*

$$O(n(1 + \mathcal{H}(r_1, \dots, r_\kappa))) \subseteq O(n(1+\log \kappa)) \subseteq O(n \log n).$$

[3]For a given polygonal chain, there can be several partitions into simple polygonal chains of minimum size for it.

This time complexity is worse-case optimal over instances of n points that can be partitioned into κ simple polygonal chains of sizes $r_1, \dots, r_\kappa$.

In the following section, we synthesize the algorithm `Galloping Partition` and the algorithm `Quick Hull Union` in a synergistic algorithm to compute the convex hull of a set of planar points.

4.3.2 Synergistic Computation of Convex Hulls

We describe a synergistic algorithm that computes the convex hull of planar points. It is synergistic in the sense that it takes advantage both of a decomposition of the points into simple polygonal chains (i.e., a form of input order) and of the relative positions of the points (i.e., the input structure) at once, such that on some large classes of instances, asymptotically outperform previous algorithms described by Kirkpatrick and Seidel [48] and Levcopoulos et al. [50] (both reviewed in Section 2.4) by more than a constant factor. Those previous solutions take advantage only of one of those aspects.

Similarly to the merging of convex hulls, we focus on the computation of the upper hull (the computation of the lower hull is symmetric and complete it into the computation of the convex hull [48]). Given a set S of planar points, the algorithm `Quick Synergy Hull` computes the upper hull of S. It proceeds in two phases. It first partitions S into simple polygonal chains using the algorithm `Galloping Partition` (described in Section 4.3.1), and computes the upper hulls of such simple polygonal chains [50], both steps in time linear in the number of points in S. Then it merges those upper hulls using the algorithm `Quick Hull Union` (described in Section 3.3).

We state the time complexity of the `Quick Synergy Hull` in function of the number and sizes of a decomposition of the points into simple polygonal chains, and in function of the hull partition certificate of the upper hulls of such simple polygonal chains obtained by the algorithm `Quick Hull Union`. We state this more formally in the following theorem:

Theorem 8. *Consider a set S of n planar points that can be partitioned into κ simple polygonal chains of sizes $r_1, \dots, r_\kappa$ (such that $\sum_{i=1}^{\kappa} r_i = n$); and also can be partitioned into h subsets of sizes $n_1, \dots, n_h$ (such that $\sum_{i=1}^{h} n_i = n$), where each subset can be enclosed by a triangle completely below the upper hull of S. The algorithm **`Quick Synergy Hull`** computes the upper hull U of S in time within*

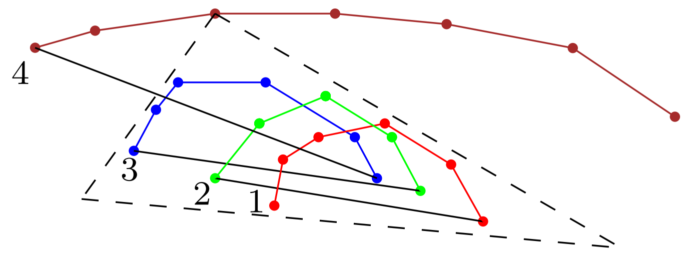
$$\begin{aligned} O\left(n + \sum_{j=1}^{\delta} w_j \log s_j + \sum_{i=1}^{\delta} \log \binom{\kappa}{m_i}\right) &\subseteq O(n(1 + \min\{\mathcal{H}(r_1, \dots, r_\kappa), \mathcal{H}(n_1, \dots, n_h)\})) \\ &\subseteq O(n(1 + \log(\min\{\kappa, h\}))) \\ &\subseteq O(n \log n), \end{aligned}$$

where the upper hulls of such simple polygonal chains admit a hull partition certificate C of minimum size δ (such that $\delta \leq h$), $m_1, \dots, m_\delta$ is a sequence where m_i is the number of blocks in the i-th argument of C (such that $m_i \leq \kappa$ for $i \in [1..\delta]$), w_j is the number of blocks in the j-th convex argument of C, and s_j is the size of the j-th block of U in C.

Proof. This result is a consequence of Theorem 4 and 7. For example, if the simple polygonal chains obtained by the algorithm `Galloping Partition` are all of constant size (i.e., the algorithm cannot take advantage of such measure of the input order), then the time complexity of the algorithm `Quick Synergy Hull` and the one described by Kirkpatrick and Seidel [48] (as analyzed by Afshani et al. [1]) are asymptotically the same. This algorithm also takes advantage of the positions of the points when using the algorithm `Quick Hull Union` described in Section 3.3 to improve upon the algorithm described in Corollary 2. □

On large classes of instances, the algorithm `Quick Synergy Hull` outperforms both the algorithm described by Levcopoulos et al. [50] and the `Marriage-before-conquest Hull` algorithm described by Kirkpatrick and Seidel [48] (even when analyzed by Afshani et al. [1]), as well as any dovetailing combination of them; and never performs asymptotically worse than them.

Example 8. *Consider the sequence S of n points and its decomposition into $\kappa = 4$ simple polygonal chains. The numbers indicate the order in which the sequences of points are given: each from left to right internally, and mark the simple polygonal chains drawn below. The line segment that joins the leftmost and rightmost points of the chain 4 is an eliminator argument for all other simple polygonal chains. The simple polygonal chains 1, 2 and 3 are more intricately woven together.*



Consider the computation of the upper hull on such instance, the time complexity of the algorithm `Marriage-before-conquest Hull` is within $\Theta(n \log h)$, where h is the number of points in the upper hull labeled 4. The refined analysis described by Afshani et al. [1] of the algorithm `Marriage-before-conquest Hull` yields a time complexity within $\Theta(n + h \log n)$ (all the points in the sequences 1, 2 and 3 can be enclosed by a triangle completely below the upper hull of the points, noted in dashes in the illustration above, hence $n_1 = \cdots = n_{h-1} = 1$ and $n_h = n - h + 1$ in the formula $\Theta(n(1 + \mathcal{H}(n_1, \ldots, n_h))))$. The time complexity on such instance of the algorithm described by Levcopoulos et al. [50] is within $\Theta(n(1 + \log \kappa))$, where κ is the minimal number of simple polygonal chains into which the sequence of points can be partitioned. The time complexity of the algorithm described in Corollary 2 is within $\Theta(n + \kappa \log n)$ (suppose that the sizes of the simple polygonal chains labeled 1 to $\kappa - 1$ are a constant c and that the size of the simple polygonal chain labeled κ is $n - (\kappa - 1)c$, then $r_1 = \cdots = r_{\kappa-1} = c$ and $r_\kappa = n - (\kappa - 1)c$ in the formula $\Theta(n(1 + \mathcal{H}(r_1, \ldots, r_\kappa))))$. On the other hand, the time complexity on such instances of the algorithm `Quick Synergy Hull` is within $\Theta(n)$: once it computes the first vertex of the upper hull of S, it discards all the points except the points in the upper hull labeled 4. If $h \in \Theta(n)$ and $\kappa \in \Theta(n)$, then the algorithm `Quick Synergy Hull` is faster than those algorithms by a factor logarithmic in the size of the input.

In the following, we discuss some extensions to the results described in this chapter.

4.4 Discussion

We have built upon previous results taking advantage of either some notions of input order, or the input structure, to describe solutions that take advantage of both in a synergistic way. Munro and Spira [56] described an algorithm for sorting multisets that takes advantage of the distribution of the multiplicities of the distinct elements in the multiset. Afshani et al. [1] described instance optimal algorithms in the order-oblivious setting for computing the maxima set and the convex hull of points in 2D and 3D. As seen in Chapter 2, the algorithm described by Munro and Spira [56] and Afshani et al. [1] take the best advantage of the input structure. Nevertheless, the matter is not so clear when we consider algorithms that take advantage of the (many possible notions of) input order.

The literature about disorder measures for sorting, and algorithms that exploit such measures to speed up the sorting of multisets is replete with distinct measures (e.g., the number of inversions, the number of runs, and the number of shuffles up-sequences) as described in the surveys by Estivill-Castro and Wood [28], by Moffat and Petersson [54], and by Mannila [52]. Some of such measures induce a partition of the data, which can potentially yield another synergistic solution in combination with the algorithm `Quick Set Union` (described in Section 3.1.2). The main point on this approach is that the time necessary to obtain the partitioning can not exceed the merging time.

Along the same line of thought, concerning the computation of the convex hull of a set of planar points, Ahn and Okamoto [2] described a notion of input order different to the one considered here (i.e., taking advantage of other features in the order in which the points are listed), which can potentially yield another synergistic solution in combination with the algorithm `Quick Hull Union` (described in Section 3.3). This is true for any of the many notions of order which could be adapted from sorting [54, 52].

We explored several generalizations of those partitioning techniques to the more challenging problem of computing the Delaunay triangulation of a set of planar points, and showed the difficulties of obtaining an optimal partition in terms of several measures that we proposed [9]. Any algorithm computing the convex hull in 3D can be used to compute the Delaunay triangulation, or its dual, the Voronoi Diagram, in dimension 2: the first step to generalize convex hull results in the plane to higher dimensions will be to consider their generalization to the computation of Delaunay triangulations.

Our runs-adaptive result on sorting multisets is based on a constant-term optimal partition in terms of the entropy of the distribution of the sizes of the runs. Similarly, our smooth-adaptive results on computing the maxima set of planar points is based on a constant-term optimal partition of the sequence of points into smooth sequences in terms of the entropy of the distribution of the sizes of the smooth sequences. It would be tempting to generalize such partitioning results for the computation of planar convex hulls in terms of the distribution of the sizes of the simple polygonal chains into which the sequence of points can be partitioned, but finding an optimal partition for this problem seems much harder.

This concludes our results about synergistic sorting, and the synergistic computation of the maxima set and convex hull of planar points. In the following chapter, we consider the extensions of those results to the context of supporting offline and online queries.

Chapter 5

Synergistic Data Structures

In this chapter, we consider the extension of the results described in Chapter 4 to supporting offline and online queries. We describe algorithms and deferred data structures that support queries related to the problems of sorting multisets, and computing the maxima set and the convex hull of planar points. Such algorithms and deferred data structures take advantage of some measures of the input order and the input structure on one hand, and of some measures of the query order and the query structure on the other hand, in a synergistic way.

As seen in Section 2.5.1, the support of rank and select queries is tightly related to the task of sorting multisets. We extend the algorithm `Quick Synergy Sort`, described in Section 4.1.2, with the analysis of a multiselection algorithm which takes advantage of not only the number and sizes of the runs (i.e., a notion of input order) and the multiplicities of the elements (i.e., the input structure) in the multiset, but also of the positions hit by the queries (i.e., the query structure), in the offline setting (Section 5.1.1). Hence, such multiselection algorithm extends the result from Kaligosi et al. [44] which takes advantage only of the query structure. In the online setting, where the queries arrive one by one, we describe and analyze deferred data structures that support rank and select queries taking advantage of some measures of the input order and the input structure on one hand, and of some measures of the query order and the query structure on the other hand, in a synergistic way (Section 5.1.2). We define the notion of query order by a measure that captures how close the positions of two consecutive queries are in terms of the number of pivots computed by the deferred data structures. Hence, such deferred data structures extend the result from Barbay et al. [6] which takes advantage only of the query structure. As an additional result, we describe two compressed data structures that represent a multiset taking advantage both of some measures of the input order and the input structure, while supporting rank and select queries on the multiset (Section 5.1.3).

Related to the computation of the maxima set of planar points, we define the problem of supporting *domination* queries, i.e., given a set of data points and a set of query points, we need to determine which of the query points are below the staircase represented by the maxima set of the data points. In the offline setting, we describe an algorithm that answers several domination queries at once. It takes advantage of a decomposition of the data points into smooth sequences (i.e., a notion of input order), the relative positions of the data points (i.e., the input structure), and the relative positions between the query points and the data points (i.e., the query structure) (Section 5.2.1). We extend such a result to the online setting by describing a deferred data structure supporting domination queries. Such deferred data structure takes advantage of the relative order in which the queries are given (i.e., the query order) in addition to the other measures considered (Section 5.2.2).

Related to the computation of the convex hull of planar points, we study the problem of supporting *point membership* queries as defined by Karp et al. [45], i.e. given a set of data points and a set of query points, determine which of the query points are included in the convex hull of the data points. In the offline setting, similarly to the support of domination queries, we describe an algorithm that answers several point membership queries at once. It takes advantage of a decomposition of the data points into simple polygonal chains (i.e., a form of input order), the relative positions of the data points (i.e., the input structure), and the relative positions between the query points and the data points (i.e., the query structure) (Section 5.3.1). We extend such result to the online setting by describing a deferred data structure that supports point membership queries taking advantage of some measure of the query order in addition to the other measures considered (Section 5.3.2).

In Section 5.4, we discuss how we could improve our results by defining finer measures of difficulties and analyzing our data structures in terms of such measures.

5.1 Supporting Rank and Select Queries on a Multiset

In Section 2.5.1, we defined rank and select queries, which are among the most well-studied operations on sequences, along with some solutions supporting them [39, 15, 34, 27].

We describe a multiselection algorithm (Section 5.1.1), based on the sorting algorithm `Quick Synergy Sort` introduced in Section 4.1. Such multiselection algorithm is an intermediate result leading to two deferred data structures supporting rank and select queries on a multiset in the online setting (Section 5.1.2). On some large classes of instances, the multiselection algorithm and the two deferred data structures asymptotically outperform all previous solutions [45, 6] described in Section 2.5.1, by taking advantage of the number and sizes of the runs in the multiset (i.e., a notion of input order), of the multiplicities of the distinct elements in the multiset (i.e., the input structure), and of the gaps between the positions hit by the queries (i.e., the query structure), in a synergistic way. One of such deferred data structures also takes advantage of a measure of the query order. Additionally, we also describe two compressed data structures that represent a multiset taking advantage both of a measure of the input order and the input structure, while supporting rank and select queries on it (Section 5.1.3).

5.1.1 Offline Setting

Given a multiset M and q select queries, the algorithm `Synergy MultiSelection` follows the same first steps as the algorithm `Quick Synergy Sort` while answering the select queries. But once it has computed the ranks of all the elements in the block B that contains the pivot of value μ, it answers the queries corresponding to B. The algorithm determines which select queries correspond to elements smaller than or equal to l, where l is the maximum element to the left of B. It also determines which elements correspond to elements greater than or equal to r, where r is the minimum element to the right of B. It then recurses on both sides. See Algorithm 5 for a formal description of the algorithm `Synergy MultiSelection`.

Algorithm 5 `Synergy MultiSelection`

Input: A multiset M and a set Q of q offline select queries
Output: The q selected elements

```
1: Compute the $\rho$ runs into which $M$ can be partitioned;
2: Compute the median $\mu$ of the middles of the $\rho$ runs, note $j \in [1..\rho]$ the run containing $\mu$;
3: Perform doubling searches for the value $\mu$ in all runs except the $j$-th, starting at both
   ends of the runs in parallel;
4: Find the maximum $l$ (minimum $r$) among the elements smaller (resp., greater) than $\mu$ in
   all runs except the $j$-th;
5: Perform doubling searches for $l$ and $r$ in the $j$-th run, starting at the position of $\mu$;
6: Answer the queries corresponding to the block $B$ whose values are between $l$ and $r$, in
   the $j$-th run;
7: Compute the set $Q_l$ of queries that go to the left of $l$ and the set $Q_r$ of queries that go
   to the right of $r$;
8: Recurse on the elements smaller than or equal to $l$ with $Q_l$, and on the elements greater
   than or equal to $r$ with $Q_r$.
```

Complexity Analysis of the Algorithm Synergy MultiSelection

We extend the notion of the decomposition of sorted sets into blocks seen in the analyses of the merging sorted sets algorithms `Sort Set Union` and `Quick Set Union` (described in Section 3.1), to the context of partial sorting. Next, we introduce the definitions of *pivot blocks* and *selection blocks* (see Figure 5.1 for a graphical illustration of those definitions). Those new definitions allow us to measure the difficulty of an instance in terms of a measure of the input order, the input structure, and the query structure.

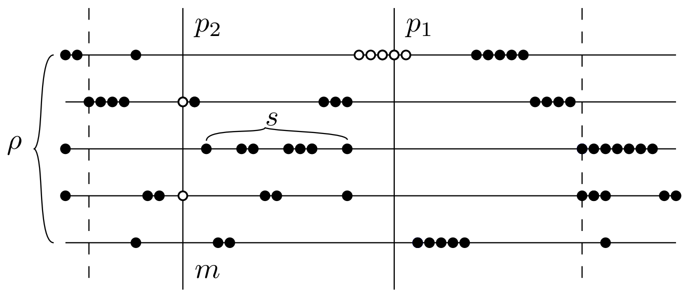


Figure 5.1: An instance of the multiselection problem where the multiset M is formed by $\rho = 5$ runs. In each run, the entry $M[i]$ is represented by a point of x-coordinate $M[i]$. The dash lines represent the answers of the two select queries. The solid vertical lines represent the positions p_1 and p_2 of the first two pivots computed by the algorithm `Synergy MultiSelection`. 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, each corresponding to the pivot p_2.

Definition 15. *Consider a multiset M formed by ρ runs and δ blocks. The* pivot blocks *are the blocks of M that contain the pivots and the elements of value equal to the pivots during the steps of the algorithm* ***Synergy MultiSelection****.*

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

Definition 16. *Consider the run R formed by blocks, and q select queries. The algorithm* `Synergy MultiSelection` *computes ξ pivots in the process of answering the q queries, and finds the insertion ranks of the ξ pivots inside R. Those positions determine a partition of size (at most) $\xi+1$ of R where each element of the partition is formed by consecutive blocks or is empty. We call the elements of such partition* selection blocks. *The set of selection blocks includes the set of pivot blocks.*

Using Definitions 15 and 16, we generalize the results proved in Section 4.1.2 on sorting multisets to the more general problem of multiselection.

Theorem 9. *Consider a multiset M of size n formed by ρ runs and δ blocks; and q offline select queries over M corresponding to elements of ranks $r_1, \dots, r_q$. Let ξ be the number of pivots computed by the algorithm* `Synergy MultiSelection` *in the process of answering the q queries. Let $s_1, \dots, s_\beta$ be the sizes of the β selection blocks determined by the ξ pivots in all runs. Let $m_1, \dots, m_\lambda$ be the numbers of pivot blocks among those selection blocks corresponding to the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \dots, \rho_\xi$ be the sequence where ρ_i is the number of runs that have elements with values between the pivots p_i and p_{i+1} sorted by ranks, for $i \in [1..\xi]$. Let $\Delta_j = r_{j+1} - r_j$ for $j \in [0..q]$, $r_0 = 0$ and $r_{q+1} = n$. The number of comparisons performed by the algorithm* `Synergy MultiSelection` *while answering the queries is within*

$$n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\xi} \rho_i \log \rho_i\right) \subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i\right)$$
$$\subseteq O(n \log q).$$

Proof. The doubling searches that find the insertion ranks of the pivots during the overall execution of the algorithm generate within $O(\sum_{i=1}^{\beta} \log s_i)$ comparisons. Indeed, at each run, a constant factor of the sum of the logarithm of the sizes of the selection blocks bounds the number of comparisons performed by those doubling searches (see the proof of Lemma 1 analyzing the Steps 2 and 4 of Algorithm 1 for details).

The pivots computed by the algorithm `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 sorted by ranks form multisets M_j such that for every pair of selection blocks in M_j, the elements of one are smaller than the elements of the other. Consider the steps that compute the medians of the middles in the algorithm `Quick Synergy Sort`, the number of comparisons performed by those steps would be within $O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i\right)$ on this supposed instance (see the proof of Lemma 2 analyzing the Steps 1 and 3 of Algorithm 1 for details). The number of comparisons needed to sort the multisets 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 4.1.1, can be applied as the first step of the multiselection algorithm. The ϕ pivot positions divide the input of size n into $\phi+1$ subinstances of sizes $n_0, \dots, n_\phi$. For each subinstance I_i for $i \in [0..\phi]$, the multiselection algorithm determines which select queries correspond to I_i and executes then Algorithm 5 inside I_i in order to answer those queries.

Corollary 3. *Let M be a multiset of size n with ϕ pivot positions. The ϕ pivot positions divide M into $\phi + 1$ subinstances of sizes $n_0, \dots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Let q be the number of offline select queries over M, such that q_i queries correspond to the subinstance I_i, for $i \in [0..\phi]$. In each subinstance I_i of size n_i formed by ρ_i runs, the algorithm* **`Synergy MultiSelection`** *selects ξ_i pivots when it answers the q_i queries corresponding to elements of ranks $r_{i1}, \dots, r_{iq_i}$. The ξ_i pivots determine β_i selection blocks of sizes $s_{i1}, \dots, s_{i\beta_i}$ inside I_i. Let $m_{i1}, \dots, m_{i\lambda_i}$ be the numbers of pivot blocks among those selection blocks corresponding to the λ_i pivots with multiplicity greater than 1, respectively. Let $\rho_{i0}, \dots, \rho_{i\xi_i}$ be the sequence where ρ_{ij} is the number of runs that have elements with values between the pivots p_{ij} and $p_{i(j+1)}$ sorted by ranks, for $j \in [1..\xi_i]$. Let $\Delta_{ij} = r_{ij+1} - r_{ij}$ for $j \in [0..q_i]$, $r_{i0} = \sum_{k=0}^{i} n_k$ and $r_{iq_i+1} = \sum_{k=0}^{i+1} n_k$. There is an algorithm that answers the q select queries performing a number of data comparisons within*

$$O\left(n + \sum_{i=0}^{\phi}\left\{\sum_{j=1}^{\beta_i}\log s_{ij} + \beta_i \log \rho_i - \sum_{j=1}^{\lambda_i} m_{ij}\log m_{ij} - \sum_{j=0}^{\xi_i}\rho_{ij}\log \rho_{ij}\right\}\right)$$
$$\subseteq O\left(n + \sum_{i=0}^{\phi}\left\{n_i \log n_i - \sum_{j=0}^{q_i}\Delta_{ij}\log \Delta_{ij}\right\}\right) \subseteq O(n \log q).$$

Similarly to the results described by Dobkin and Munro [27], in this section, we focus only on the multiselection problem. But, those results could be easily extended to support `rank`(x) queries: at each step of the algorithm, once the median μ has been computed, the element x is compared with μ to decide if the answer to `rank`(x) is less than, equal to, or greater than the rank of μ. We exploit that idea in the following section.

In the results 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 two deferred data structures for answering online rank and select queries, both inspired by the algorithm `Synergy MultiSelection`.

5.1.2 Online Setting

We describe two deferred data structures that answer q rank and select queries arriving one at a time over a multiset M, progressively sorting M. Both deferred data structures take advantage of some measures of the input order (local and global), the input structure, and of the query structure, in a synergistic way. The first data structure is in the RAM model of computation, at the cost of not taking advantage of the order in which the queries are given. The second one is in the pointer-machine model (a more constrained model) but does take advantage of some measures of the query order.

Taking Advantage of Some Measures of the Input Order and the Input Structure, but only of the Query Structure

Given a multiset M of size n, the `RAM Aware` deferred data structure is composed of a bitvector A of size n, in which it marks the elements in M that have been identified as pivots when it answers the online queries; a dynamic predecessor and successor structure B over the bitvector A, which returns the two successive pivots between which the query fits; and for each pivot p found, the data structure stores several pointers: in each run, to the insertion ranks of p, and in the run that contains p, to the beginning and end of the block g to which p belongs, and to the position of p inside g. The dynamic predecessor and successor structure B requires the RAM model of computation in order to answer *predecessor* and *successor* queries in time within $o(\log n)$ [11].

Theorem 10. *Consider a multiset M of size n formed by ρ runs and δ blocks. Let ξ be the number of pivots computed by the* `RAM Aware` *deferred data structure in the process of answering q online rank and select queries over M. Let $r_1, \dots, r_q$ be the ranks of the elements corresponding to the q queries. Let $s_1, \dots, s_\beta$ be the sizes of the β selection blocks determined by the ξ pivots in all runs. Let $m_1, \dots, m_\lambda$ be the numbers of pivot blocks among those selection blocks corresponding to the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \dots, \rho_\xi$ be the sequence where ρ_i is the number of runs that have elements with values between the pivots p_i and p_{i+1} sorted by ranks, for $i \in [1..\xi]$. Let u and $g_1, \dots, g_u$ be the number of rank queries and the sizes of the identified and searched blocks in the process of answering the u rank queries, respectively. Let $\Delta_i = r_{i+1} - r_i$, $r_0 = 0$ and $r_{q+1} = n$. The* `RAM Aware` *deferred data structure answers the q online rank and select queries in time within*

$$O(n + \sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\xi} \rho_i \log \rho_i$$
$$+\xi \log\log n + u \log n \log\log n + \sum_{i=1}^{u} \log g_i) \subseteq$$
$$O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i + q \log n\right) \subseteq O(n \log q + q \log n).$$

Proof. The algorithm answers a new `select`(i) query by accessing in A the query position i. If $A[i]$ is 1, then the element e that answers the query has been computed as pivot, and hence the algorithm answers the query in constant time by following the position of e inside the block at which e belongs. If $A[i]$ is 0, then the algorithm finds the nearest pivots to its left and right using the predecessor and successor structure, B. If the position i is inside a block to which one of the two nearest pivots belongs, then the algorithm answers the query and in turn finishes. If not, it then applies the same steps as the algorithm `Synergy MultiSelection` in order to answer the query; it updates the bitvector A and the dynamic predecessor and successor structure B whenever a new pivot is computed; and for each pivot p computed, the structure stores several pointers: in each run, to the insertion ranks of p, and in the run that contains p, to the beginning and end of the block g to which p belongs, and to the position of p inside g.

The algorithm answers a new `rank`(x) query by finding for all $j \in [1..\rho]$, the selection block s_j in the j-th run such that x is between the smallest and the greatest values of s_j. For that, the algorithm performs a sort of parallel binary searches for the value x taking advantage of the pivots that have been computed by the algorithm. The algorithm accesses the position $\frac{n}{2}$ in A. If $A[\frac{n}{2}]$ is 1, then the element e of rank $\frac{n}{2}$ has been computed as pivot. Following the pointer to the block g to which e belongs, the algorithm decides if x is to the right, to the left or inside g by performing a constant number of data comparisons. In the last case, a binary search for the value x inside g yields the answer to the query. If $A[\frac{n}{2}]$ is 0, then the algorithm finds the nearest pivots to the left and to the right of the position $\frac{n}{2}$ using the predecessor and successor structure, B. Following the pointers to the blocks that contain those pivots, the algorithm decides if x is inside one of such blocks, to the right of the rightmost block, to the left of the leftmost block, or between the two blocks. In the last case, 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. Those steps identify several new pivots, and in consequence several new blocks in the structure. □

The `RAM Aware` deferred data structure includes the pivot positions, seen in Section 4.1.1, as a natural extension of the algorithm. The ϕ pivot positions are marked in the bitvector A. For each pivot position p, the structure stores pointers to the end of the runs detected to the left of p; to the beginning of the runs detected to the right of p; and to the position of p in the multiset.

Corollary 4. *Let M be a multiset of size n with ϕ pivot positions. The ϕ pivot positions divide M into $\phi + 1$ subinstances of sizes $n_0, \dots, n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Let q be the number of online rank and select queries over M, such that q_i queries correspond to the subinstance I_i, for $i \in [0..\phi]$. In each subinstance I_i of size n_i formed by ρ_i runs, the `RAM Aware` deferred data structure computes ξ_i pivots in the process of answering the q_i online rank and select queries over I_i. Let $r_{i1}, \dots, r_{iq_i}$ be the ranks of the elements corresponding to the q_i queries. Let u_i and $g_{i1}, \dots, g_{iu_i}$ be the number of rank queries and the sizes of the identified and searched blocks in the process of answering the u_i rank queries over I_i, respectively. Let $s_{i1}, s_{i2}, \dots, s_{i\beta_i}$ be the sizes of the β_i selection blocks determined by the ξ_i pivots in all runs of I_i. Let $\Delta_{ij} = r_{ij+1} - r_{ij}$ for $j \in [0..q_i]$, $r_{i0} = \sum_{k=0}^{i} n_k$ and $r_{iq_i+1} = \sum_{k=0}^{i+1} n_k$. There is a structure that answers those online queries in time within*

$$O\left(n + \sum_{i=0}^{\phi}\left\{\beta_i \log \rho_i + \sum_{j=1}^{\beta_i} \log s_{ij} + \xi_i \log\log n + u_i \log n_i \log\log n_i + \sum_{j=1}^{u_i} \log g_{ij})\right\}\right)$$
$$\subseteq O\left(n + \sum_{i=0}^{\phi}\left\{n_i \log n_i - \sum_{j=0}^{q_i} \Delta_{ij} \log \Delta_{ij} + q_i \log n_i\right\}\right) \subseteq O(n \log q + q \log n).$$

The `RAM Aware` deferred data structure takes advantage of the query structure, some measures of the input order (local and global) and the input structure. Changing the order of the rank and select queries does not affect the total time complexity of the `RAM Aware` deferred data structure. Once the structure identifies the nearest pivots to the left and to the right of the query positions, the steps of the algorithms are the same as in the offline case (Section 5.1.1). We describe next a deferred data structure taking advantage of some measure of the query order and the query structure and of some measures of the input order (local and global) and the input structure.

Taking Advantage of the Order and Structure in both the Input and the Queries

To take advantage of the order in the queries, 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. This distance is measured in the number of computed pivots between the two positions. For that we use a *finger search tree* [37] which is a search tree maintaining *fingers* (i.e., pointers) to elements in the tree. Such finger search trees support efficient updates and searches in the vicinity of the fingers. Brodal [16] described an implementation of finger search trees that supports a search 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.

In the description of the `RAM Aware` deferred data structure from Theorem 10, we substitute the dynamic predecessor and successor structure B by a finger search tree $F_{\texttt{select}}$, as described by Brodal [16]. Once a block g is identified, every element in g is a valid pivot for the rest of the elements in the multiset M. In order to capture such idea, we modify the structure $F_{\texttt{select}}$ so that it contains blocks instead of singleton pivots. Each element in $F_{\texttt{select}}$ points in M to the beginning and end of the block g that it represents, and in each run, to the position where the elements of g partition the run. This modification allows the structure to answer select queries, taking advantage of some measure of the query order and the query structure and of some measures of the input order and the input structure. But in order to answer rank queries, the structure needs another finger search tree $F_{\texttt{rank}}$. In $F_{\texttt{rank}}$, the structure stores for each block g identified, the value of one of the elements in g, and pointers in M to the beginning and end of g, and in each run, to the position where the elements of g partition the run. We name such a structure the `Full Synergy` deferred data structure.

Theorem 11. *Consider a multiset M of size n formed by ρ runs and δ blocks. Let γ be the number of blocks identified by the `Full Synergy` deferred data structure in the process of answering q online rank and select queries over M. Let $r_1, \dots, r_q$ be the ranks of the elements corresponding to the q queries. Let $s_1, \dots, s_\beta$ be the sizes of the β selection blocks determined by the pivots in the γ blocks in all runs. Let $m_1, \dots, m_\lambda$ be the numbers of pivot blocks corresponding to the λ pivots with multiplicity greater than 1, respectively. Let $\rho_0, \dots, \rho_\gamma$ be the sequence where ρ_i is the number of runs that have elements with values between the values in the blocks i and $i+1$ sorted by ranks, for $i \in [1..\gamma]$. Let $d_1, \dots, d_{q-1}$ be the sequence where d_j is the number of identified blocks between the block that answers the j-th query and the one that answers the $j+1$-th query before starting the steps to answer the $j+1$-th query, for $j \in [1..q-1]$. Let u and $g_1, \dots, g_u$ be the number of rank queries and the sizes of the identified and searched blocks in the process of answering the u rank queries, respectively. Let $\Delta_i = r_{i+1} - r_i$, $r_0 = 0$ and $r_{q+1} = n$. The `Full Synergy` deferred data structure answers the q online queries performing a number of data comparisons within*

$$n + O\left(\sum_{i=1}^{\beta} \log s_i + \beta \log \rho - \sum_{i=1}^{\lambda} m_i \log m_i - \sum_{i=0}^{\gamma} \rho_i \log \rho_i + \sum_{i=1}^{q-1} \log d_i + \sum_{i=1}^{u} \log g_i\right)$$
$$\subseteq O\left(n \log n - \sum_{i=0}^{q} \Delta_i \log \Delta_i + q \log n\right) \subseteq O(n \log q + q \log n),$$

Proof. The algorithm answers a new $\texttt{select}(i)$ query by searching in $F_{\texttt{select}}$ for the nearest pivots to the left and to the right of the query position i. If i is contained in an element of $F_{\texttt{select}}$, then the block g that contains the element in the position i has already been identified. If i is not contained in an element of $F_{\texttt{select}}$, then the returned finger f points to the nearest block b to the left of i. In $F_{\texttt{select}}$, the block that follows f is the nearest block to the right of i. It then applies the same steps as the algorithm `Synergy MultiSelection` in order to answer the query. In $F_{\texttt{select}}$, given f, the algorithm inserts each block identified in the process of answering the query in constant time, and stores the respective pointers to positions in M. In $F_{\texttt{rank}}$, the algorithm searches for the value of one of the elements in the block b pointed by f. Once the algorithm obtains the finger returned by this search, in $F_{\texttt{rank}}$, the algorithm inserts the value of one of the elements of each block identified in constant time and stores the respective pointers to positions in M (see Figure 5.2 for a graphical representation of some of the parameters used in the analysis).

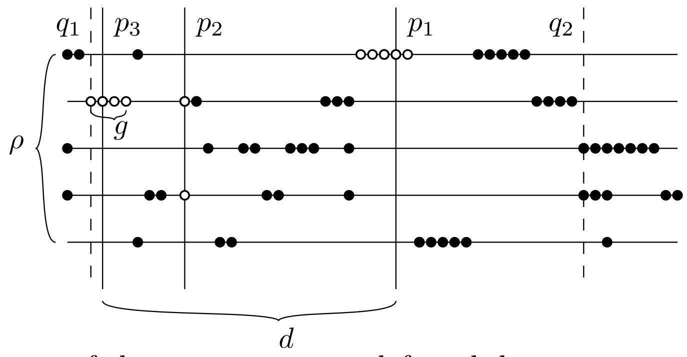


Figure 5.2: The state of the `Full Synergy` deferred data structure on an instance where the multiset M is formed by $\rho = 5$ runs. In each run, the entry $M[i]$ is represented by a point of x-coordinate $M[i]$. The dash lines represent the positions q_1 and q_2 of the first two queries. The solid vertical lines represent the positions p_1, p_2 and p_3 of the first three pivots computed by the `Full Synergy` 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 identified 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 to the query q_1.

The algorithm answers a new $\texttt{rank}(x)$ query by finding for all $j \in [1..\rho]$, the selection block s_j in the j-th run such that x is between the smallest and the greatest values of s_j. For that the algorithm searches for the value x in $F_{\texttt{rank}}$. The number of data comparisons performed by such searching process is within $O(\log d)$, where d is the number of blocks in $F_{\texttt{rank}}$ 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 of the block pointed by f, (ii) r is between the ranks of the elements 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 to the query. In case (iii), the algorithm applies the same steps as the algorithm `Synergy MultiSelection` in order to compute the median μ of the middles and partitions the selection blocks by μ. In $F_{\texttt{select}}$ and $F_{\texttt{rank}}$, the algorithm inserts the identified blocks and their corresponding pointers. The algorithm then decides to which side x belongs. $\square$

The process of detecting the ϕ pivot positions, seen in Section 4.1.1, allows the `Full Synergy` deferred data structure to insert those pivots in $F_{\texttt{select}}$ and $F_{\texttt{rank}}$. For each pivot position p, in $F_{\texttt{select}}$ and $F_{\texttt{rank}}$, the structure stores pointers to the end of the runs detected to the left of p; to the beginning of the runs detected to the right of p; and to the position of p in the multiset.

Corollary 5. *Let M be a multiset of size n with ϕ pivot positions. The ϕ pivot positions divide M into $\phi+1$ subinstances of size $n_0,\dots,n_\phi$ (such that $\sum_{i=0}^{\phi} n_i = n$). Let q be the number of online rank and select queries over M, such that q_i queries correspond to the subinstance I_i, for $i \in [0..\phi]$. In each subinstance I_i of size n_i formed by ρ_i runs, the **`Full Synergy`** deferred data structure identifies γ_i blocks in the process of answering the q_i online rank and select queries over I_i. Let $r_{i1},\dots,r_{iq_i}$ be the ranks of the elements corresponding to the q_i queries. Let $s_{i1}, s_{i2},\dots,s_{i\beta_i}$ be the sizes of the β_i selection blocks determined by the γ_i blocks in all runs of I_i. Let $m_{1i},\dots,m_{i\lambda_i}$ be the numbers of pivot blocks among these selection blocks corresponding to the λ_i pivots with multiplicity greater than 1, respectively. Let $\rho_{i0},\dots,\rho_{i\xi_i}$ be the sequence where ρ_{ij} is the number of runs that have elements with values between the pivots p_{ij} and $p_{i(j+1)}$ sorted by ranks, for $j \in [1..\xi_i]$. Let $d_{i1}, d_{i2},\dots,d_{iq_{i-1}}$ be the sequence where d_{ij} is the number of identified blocks between the block that answers the ij-th query and the one that answers the $i(j+1)$-th query before starting the steps for answering the $i(j+1)$-th query, for $j \in [1..q_i-1]$. Let u_i and $g_{i1},\dots,g_{iu_i}$ be the number of rank queries and the sizes of the identified and searched blocks in the process of answering the u_i rank queries over I_i, respectively. Let $\Delta_{ij} = r_{ij+1} - r_{ij}$ for $j \in [0..q_i]$, $r_{i0} = \sum_{k=0}^{i} n_k$ and $r_{iq_i+1} = \sum_{k=0}^{i+1} n_k$. There exists a data structure that answers the q online rank and select queries performing a number of data comparisons within*

$$O\left(n + \sum_{i=0}^{\phi}\left\{\sum_{j=1}^{\beta_i}\log s_{ij} + \beta_i\log\rho_i - \sum_{j=1}^{\lambda_i} m_{ij}\log m_{ij} - \sum_{j=0}^{\xi_i}\rho_{ij}\log\rho_{ij} + \sum_{j=1}^{q_i-1}\log d_{ij} + \sum_{j=1}^{u_i}\log g_{ij}\right\}\right)$$

$$\subseteq O\left(n + \sum_{i=0}^{\phi}\left\{n_i\log n_i - \sum_{j=0}^{q_i}\Delta_{ij}\log\Delta_{ij} + q_i\log n_i\right\}\right) \subseteq O(n\log q + q\log n).$$

The `Full Synergy` deferred data structure has two advantages over the `RAM Aware` deferred data structure: (i) it is in the pointer-machine model of computation, which is less powerful than the RAM model; and (ii) it takes advantage of some measure of the query order and the query structure and of some measures of the input order (local and global) and the input structure, when the `RAM Aware` deferred data structure does not take advantage of any measure of the query order. Next, we present two compressed data structures, taking advantage of the block representation of a multiset M while supporting rank and select queries over M.

5.1.3 Compressed Data Structures

We describe two compressed representations of a multiset M of size n formed by ρ runs and δ blocks while supporting rank and select queries over it. The first compressed data structure represents M in $\delta\log\rho + 3n + o(\delta\log\rho + n) \subseteq O(n\log n)$ bits and supports each rank query in constant time and each select query in time within $O(\log\log\rho) \subseteq O(\log\log n)$. The second compressed data structure represents M in $\delta\log\delta + 2n + O(\delta\log\log\delta) + o(n) \subseteq O(n\log n)$ bits and supports each select query in constant time and each rank query in time within $O\left(\frac{\log\delta}{\log\log\delta}\right) \subseteq O\left(\frac{\log n}{\log\log n}\right)$.

Given a bitvector V, $\texttt{rank}_1(V,\ j)$ finds the number of occurrences of bit 1 in $V[0..j]$, and $\texttt{select}_1(V,\ i)$ finds the position of the i-th occurrence of bit 1 in V. Given a sequence S over an alphabet of size ρ, $\texttt{rank}(S,\ c,\ j)$ finds the number of occurrences of character c in $S[0..j]$; $\texttt{select}(S,\ c,\ i)$ finds the position of the i-th occurrence of character c in S; and $\texttt{access}(S,\ j)$ returns the character at position j in S.

Rank Aware Compressed Data Structure

The `Rank Aware` compressed data structure supports each rank query in constant time, and each select query in time within $O(\log\log\rho) \subseteq O(\log\log n)$, using $\delta\log\rho + 3n + o(\delta\log\rho + n) \subseteq O(n\log n)$ bits. It contains three bitvectors A, B and C of size n supporting for $V \in \{A, B, C\}$, $\texttt{rank}_1(V,\ j)$ and $\texttt{select}_1(V,\ i)$ in constant time using $n + o(n)$ bits each [21]. It contains a data structure S representing a sequence of length δ from an alphabet of size ρ supporting $\texttt{rank}(S,\ c,\ j)$ in time within $O(\log\log\rho)$, $\texttt{access}(S,\ j)$ in time within $O(\log\log\rho)$, and $\texttt{select}(S,\ c,\ i)$ in constant time, using $\delta\log\rho + o(\delta\log\rho)$ bits [36]. Given the blocks $g_1, \dots, g_\delta$ in sorted order, A contains the information of the lengths of such blocks in this order: the set bits mark the positions where each block starts. B contains the information of the lengths of the blocks similar to A but with the blocks maintaining the original order, such that all blocks belonging to the same run are consecutive. C contains the information of the length of the runs: the set bits mark the positions where each run starts. For each block g in sorted order, S contains the run to which g belongs.

Theorem 12. *Let M be a multiset of size n formed by ρ runs and δ blocks. The `Rank Aware` compressed data structure represents M in $\delta\log\rho + 3n + o(\delta\log\rho + n) \subseteq O(n\log n)$ bits, supporting each rank query in constant time, and each select query in time within $O(\log\log\rho) \subseteq O(\log\log n)$.*

Proof. To answer $\texttt{rank}(M,\ x)$, the following operations are executed: $\texttt{rank}_1(C,\ i)$ returns the run r that contains x in constant time, where i is the position of x in the original order of M; $\texttt{select}_1(C,\ r)$ returns the position q where r starts in the original order of M in constant time; $\texttt{rank}_1(B,\ i) - \texttt{rank}_1(B,\ q-1)$ returns the position p inside of r of the block g that contains x in constant time; $\texttt{select}(S,\ r,\ p)$ returns the position j of g in sorted order in constant time; and $\texttt{select}_1(A,\ j)$ returns the rank of the first element in g in constant time.

To answer $\texttt{select}(M,\ i)$, the following operations are executed: $\texttt{rank}_1(A,\ i)$ returns the position j of the block g in sorted order that contains the selected element e in constant time; $\texttt{access}(S,\ j)$ returns the run r that contains e in time within $O(\log\log\rho)$; $\texttt{rank}(S,\ r,\ j)$ returns the position p of g inside r in time within $O(\log\log\rho)$; and $\texttt{select}_1(B,\ p + \texttt{rank}_1(B,\ \texttt{select}_1(C,\ r)))$ returns the position where g starts in the original order of M in constant time. □

We describe next a compressed data structure that represents a multiset, taking advantage of its block representation, but unlike the `Rank Aware` compressed data structure, the structure supports select queries in constant time.

Select Aware Compressed Data Structure

The `Select Aware` compressed data structure supports each select query in constant time, and each rank query in time within $O\left(\frac{\log\delta}{\log\log\delta}\right) \subseteq O\left(\frac{\log n}{\log\log n}\right)$, using $\delta\log\delta + 2n + O(\delta\log\log\delta) +$

$o(n) \subseteq O(n \log n)$ bits. It contains the same two structures A and B described above, and a structure representing a permutation π of the numbers $[1..\delta]$ supporting the direct operator $\pi()$ in constant time and the inverse operator $\pi^{-1}()$ in time within $O\left(\frac{\log \delta}{\log\log\delta}\right)$ using $\delta\log\delta + O(\delta\log\log\delta)$ bits [55]. Given the blocks $g_1, \ldots, g_\delta$ in sorted order, $\pi(i)$ returns the position j of the block g_i in the original order of M and $\pi^{-1}(j) = i$ if the position of the block g_i is j in the original order of M.

Theorem 13. *Let M be a multiset of size n formed by δ blocks. The* ***`Select Aware`*** *compressed data structure represents M in $\delta\log\delta + 2n + O(\delta\log\log\delta) + o(n) \subseteq O(n\log n)$ bits, supporting each select query in constant time, and each rank query in time within $O\left(\frac{\log \delta}{\log\log\delta}\right) \subseteq O\left(\frac{\log n}{\log\log n}\right)$.*

Proof. To answer `select`(M, i), the following operations are executed: `rank`$_1(A, i)$ returns the position j of the block g_j in sorted order that contains the selected element in constant time; $\pi(j)$ returns the position p of g_j in the original order of M in constant time; and `select`$_1(B, p)$ returns the position where g_j starts in M in constant time.

To answer `rank`(M, x), the following operations are executed: `rank`$_1(B, i)$ returns the position j of the block g that contains x in constant time, where i is the position of x in the original order of M; $\pi^{-1}(j)$ returns the position p of g in sorted order in time within $O\left(\frac{\log \delta}{\log\log\delta}\right)$; and `select`$_1(A, p)$ returns the rank of the first element of g in constant time. □

In the following section, we extend some of these results to the context of supporting domination queries. The domination queries are related to the computation of the maxima set of planar points.

5.2 Supporting Domination Queries on a Set of Planar Points

In this section, we extend the results about the synergistic computation of the maxima set (described in Section 4.2.2), and the multiselection algorithm and the deferred data structures supporting rank and select queries on a multiset (described in Section 5.1) to supporting offline and online domination queries on a set of planar points.

We define the problem of supporting domination queries as follows (see Figure 5.3 for a graphical representation of such definition):

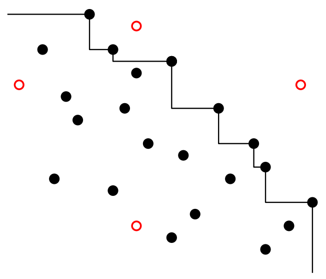


Figure 5.3: An instance of the problem of supporting domination queries. The black solid disks mark the data points, and the red open disks mark the query points. The staircase that represents the maxima set of the data points is marked.

SUPPORTING DOMINATION QUERIES: Given a set of data points and a set of query points in the plane, determine which of the query points are above the staircase defined by the maxima set of the data points (i.e., which of the query points are not dominated by any of the data points).

The support of domination queries is related to the computation of the maxima set of planar points. In the offline setting, we describe the algorithm `Synergy Domination` that answers several domination queries at the same time (Section 5.2.1), inspired by the algorithm `Quick Synergy Maxima` (previously described in Section 4.2.2). In the online setting, where the queries arrive one by one, we describe the `Full Synergy Domination` deferred data structure that progressively computes the maxima set of the data points in a "lazy" way while answering the domination queries (Section 5.2.2). Similarly to the support of rank and select queries over a multiset, both the `Synergy Domination` algorithm and the `Full Synergy Domination` deferred data structure take advantage of some measures of the input order and the input structure of the data points, and of the query structure. In the case of the deferred data structure, it also takes advantage of some measure of the query order (see Section 5.2.2 for the details of the analysis).

5.2.1 Offline Setting

Given a set S of data points and a set Q of query points, the algorithm `Synergy Domination` first computes the maxima sets of the ζ smooth sequences into which S can be partitioned. It then follows the same first steps of the algorithm `Quick Synergy Maxima`. But once it has computed, in one of such maxima sets, a block B of consecutive points that are in the maxima set of S, it partitions the points in Q into four sets. The first set is formed by the query points that dominate some of the points in B (such points are output). The second set is formed by the query points that are dominated by the points in B (such points are discarded). The third and fourth sets Q_l and Q_r are formed by the query points that remain to the left and to the right of the points in B, respectively. The algorithm `Synergy Domination` then recurses on the remaining data points to the left and to the right of the points in B with the sets Q_l and Q_r, respectively. A formal description of it is given in Algorithm 6.

As a pedagogical intermediate step before analyzing the complexity of the algorithm `Synergy Domination`, we first analyze the complexity of a simpler algorithm, which we name `Simpler Domination`, for supporting domination queries. The algorithm `Simpler Domination` is directly based on the algorithm described by Kirkpatrick and Seidel [47] for computing the maxima set of planar points. The algorithm `Simpler Domination` therefore does not compute the maxima sets of the smooth subsequences.

Given a set S of data points and a set Q of query points, the algorithm `Simpler Domination` computes the median μ of the x-coordinates of the points in S, in time linear in the number of points in S, using the median finding algorithm described by Blum et al. [15]. It computes the point p of maximum y-coordinates to the right of the vertical line ℓ of equation $x = \mu$, and discards all the points in S dominated by p, both steps in time linear in the number of points in S. It discards also the points in Q dominated by p, and outputs the points in Q that dominate p. Finally, it recurses on the non-discarded data and query points to the left and to the right of ℓ, respectively.

Similarly to the analysis of the Kirkpatrick and Seidel's algorithm [47] described by Afshani et al. [1], we analyze the algorithm `Simpler Domination` in function of the entropy

Algorithm 6 `Synergy Domination`

```
Input: A set $S$ of data points and a set $Q$ of query points
Output: The subset of points in $Q$ that are not dominated by any of the points in $S$
1: Compute the maxima sets of the $\zeta$ smooth sequences into which $S$ can be partitioned;
2: Compute the median $\mu$ of the $x$-coordinates of the points at the middle position in the maxima sets;
3: Partition the maxima sets by the vertical line $\ell$ of equation $x = \mu$;
4: Find the data point $p$ of maximum $y$-coordinate to the right of $\ell$, note $N$ the maxima set containing $p$;
5: Discard all data points dominated by $p$;
6: Find the points $l$ and $r$ of maximum $x$-coordinate such that $l_y > p_y$ and maximum $y$-coordinate such that $r_x > p_x$, respectively, in all the maxima sets except $N$;
7: Find the block $B$ of consecutive points in $N$ that are in the region determined by the vertical line of equation $x = l_x$ and the horizontal line of equation $y = r_y$;
8: Output the query points in $Q$ that dominate points in $B$;
9: Discard the query point in $Q$ that are dominated by points in $B$;
10: Compute the set $Q_l$ of remaining query points to the left of the points in $B$, and the set $Q_r$ of remaining query points to the right of the points in $B$;
11: Recurse separately on the non-discarded data points to the left of the points in $B$ with the set $Q_l$ of query points, and on the non-discarded data points to the right of the points in $B$ with the query points $Q_r$.
```

of a partition of the data points determined by the positions of the query points, a concept captured by the definition of *dominance points* (see Figure 5.4 for a graphical illustration of such definition):

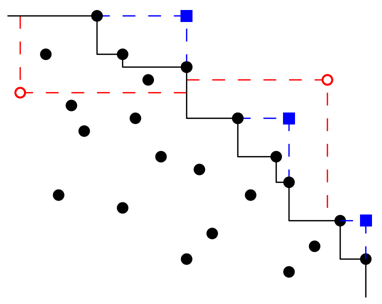


Figure 5.4: An instance of the problem of supporting domination queries. The black solid disks mark the data points. The red open disks mark the query points. The blue solid squares mark the dominance points.

Definition 17. *Given a set S of data points and a set Q of q query points, let M be the maxima set of S. Let $B_0, B_1, \ldots, B_\gamma$ be the γ blocks of consecutive points in M (such that $\gamma \leq 2q+1$), sorted by the x-coordinates of the points in them, such that for each $i \in [1..\gamma]$, either there is a point in Q whose x-coordinate is between the x-coordinates of the rightmost point in B_{i-1} and the leftmost point in B_i or there is a point in Q whose y-coordinate is between the y-coordinates of the rightmost points in B_{i-1} and the leftmost point in B_i. For each block B_i, there is a virtual* dominance point *p_i associated with it. The x-coordinate of p_i is the x-coordinate of the rightmost point in B_i, and the y-coordinate of p_i is the y-coordinates* *of the leftmost point in B_i. Note that p_i is not necessarily part of S nor of Q.*

It follows from Definition 17 that the dominance points are maximal points among the set S of data points. We define the dominance entropy $\mathcal{H}(I)$ of an instance I to represent its difficulty in the context of the supporting domination query problem.

Definition 18. *Consider an instance I of the supporting domination problem formed by a set S of data points and a set Q of query points. Consider a partition Π of the points in S into γ subsets $S_1, \dots, S_\gamma$. We say that Π is* respectful *if the points of each member of Π can be enclosed by a rectangle completely below the staircase determined by the γ dominance points of I. Let $\mathcal{H}(\Pi) = \mathcal{H}(n_1, \dots, n_\gamma)$ be the entropy of a partition Π, where $n_1, \dots, n_\gamma$ are the sizes of the members of Π. Define the* dominance entropy *$\mathcal{H}(I)$ of I to be the minimum of $\mathcal{H}(\Pi)$ over all respectful partitions Π of S.*

Given an instance I formed by a set S of data points and a set Q of query points, the number of coordinate comparisons performed by the algorithm `Simpler Domination` can be bounded in terms of the number of coordinate comparisons needed to compute the maxima set of S, plus the dominance points, which is related to the dominance entropy $\mathcal{H}(I)$ of I:

Lemma 13. *Given an instance I formed by a set S of n data points and q query points, let γ be the number of dominance points of I. Let $\mathcal{H}(I)$ be the dominance entropy of I. The algorithm* `Simpler Domination` *answers the q domination queries performing a number of coordinate comparisons within*

$$O\left(n\mathcal{H}(I) + \sqrt{q}\log n\right) \subseteq O(n \log \gamma + \sqrt{q}\log n).$$

Proof. In a preprocessing phase, the algorithm builds a k-d tree [12] in which at each step, it tests which of the query points are dominated by the points of S, which ones form part of the output, and which ones need more computation to make a decision.

Consider any respectful partition Π of S. Fix any member N_i of Π of size n_i. The algorithm `Simpler Domination` in the worst case considers the n_i points of N_i in all the levels of the recursion tree from the first one to the $\lceil \log \frac{n}{n_i} \rceil + 1$. At each level of the recursion tree, the number of points in the subinstance is reduced by half.

In the following levels, if the rectangle R enclosing the points in N_i is completely below the maxima set M of S, then every time the algorithm computes a point in M, it discards at least half of the remaining points of N_i. So, the numbers of remaining points of N_i by levels form a geometry progression of ratio 1/2. As a result, the number of coordinate comparisons involving points from N_i is within $O(n_i \log \frac{n}{n_i} + n_i)$.

Otherwise, if the rectangle R enclosing the points in N_i is completely below the staircase determined by the dominance points, but not completely below M, then let p_i be one of the dominance points such that p_i dominates all the points in N_i. If the point p of M computed by the algorithm dominates at least half of the points in N_i, then the analysis is similar to the case where R is completely below M. Otherwise, if there are points of N_i to the left and to the right of p that are not dominated by p, then in the following recursive calls that involve the rightmost points of N_i to the left of p, and the leftmost points of N_i to the right of p, the algorithm stops the computation: all the query points that dominate (resp., are dominated) by p_i are output (resp., discarded) (those are the same points that dominate (resp., are dominated) by p). Only two branches of the recursion tree require more computation. Similarly to the other case, the number of coordinate comparisons involving points from N_i is within $O(n_i \log \frac{n}{n_i} + n_i)$.

Combining the above facts that bound the number of coordinate comparisons involving points of N_i by $O(n_i \log \frac{n}{n_i} + n_i)$, the result follows. □

We adapt the concept of dominance points to the context of a decomposition of ζ maxima sets into blocks of consecutive points as obtained by the merging algorithm `Quick Maxima Union` (described in Section 3.2):

Definition 19. *Consider an instance I formed by ζ maxima sets and a set Q of q query points. Let M be the maxima set of the union of the ζ maxima sets. Let C be a maxima partition certificate of minimum size δ of the union of the ζ maxima sets. The points of Q partition the δ blocks in C that form M into subsets such that two blocks A and B belong to the same subset if there is not point p in Q such that the lines of equations $x = p_x$ or $y = p_y$ leave the points of A and B in different half-planes. For each of such subsets S_i, there is a* dominance block point *p_i associated with S_i. The x-coordinate of p_i is the x-coordinate of the rightmost point in S_i, and the y-coordinate of p_i is the y-coordinate of the leftmost point in S_i.*

We adapt the dominance entropy $\mathcal{H}(I)$, that represents the difficulty of a general instance of the supporting domination queries problem to instances of the same problem formed by ζ maxima sets and q query points.

Definition 20. *Consider an instance I formed by ζ maxima sets and q query points. Let C be a maxima partition certificate of minimum size of the union of the ζ maxima sets. Let β be the number of blocks in C. Let γ be the number of dominance block points in I. Consider a partition Π of the β blocks in C into γ subsets $B_1, \ldots, B_\gamma$. We say that Π is* block respectful *if the points of each member of Π can be enclosed by a rectangle completely below the staircase determined by the γ dominance block points of I. Let $\mathcal{H}(\Pi) = \mathcal{H}(n_1, \ldots, n_\gamma)$ be the block entropy of a partition Π, where $n_1, \ldots, n_\gamma$ are the sizes (i.e., the number of blocks) of the members of Π. Define the* block dominance entropy *$\mathcal{H}_B(I)$ of I to be the minimum of $\mathcal{H}(\Pi)$ over all block respectful partitions Π of the blocks in C.*

We reduce the analysis of the performance of the algorithm `Synergy Domination` to the analysis of the performance of the algorithm `Simpler Domination`, described in Lemma 13, using Definition 19 of dominance block points instead of Definition 17 of dominance points.

Theorem 14. *Consider an instance I of n data points that can be partitioned into ζ smooth sequences, and q query points. Let C be a maxima partition certificate of minimum size of the union of the ζ maxima sets. Let β be the number of blocks of consecutive points in C. Let γ be the number of dominance block points in I. Let $\mathcal{H}_B(I)$ be the block dominance entropy of I. The number of coordinate comparisons performed by the algorithm `Synergy Domination` while answering the q domination queries is within*

$$O(n + \beta\mathcal{H}_B(I) + q^{1/2} \log \beta) \subseteq O(n + \beta \log \gamma + q^{1/2} \log \beta).$$

Proof. There are two key points to prove the theorem. First, we consider the blocks of consecutive points of the maxima sets that form the maxima partition certificate C as singleton points, and then Lemma 13 is applied. Second, even though the partition of the maxima sets by the median μ of the x-coordinates of the points at the middle positions could be unbalance, the algorithm finds a point p in the maxima set of S with at least $\zeta/2$ points at each side of the vertical line through p, using only $O(\zeta)$ coordinate comparisons. The latest key point is used in the proof of Theorem 2 to prove the optimality of the algorithm `Quick Maxima Union`. □

Though the techniques used in the analyses of the algorithms `Synergy MultiSelection` (described in Section 5.1.1) and `Synergy Domination` (described in this section) seem very different, they both achieve the same goal of counting only the comparisons associated to the nodes that are expanded in the recursion tree of the algorithms `Quick Set Union` (described in Section 3.1.2) and `Quick Maxima Union` (described in Section 3.2.1), respectively, when answering the queries.

In the following section, we extend the results described above to the context where the queries arrive one by one (i.e., the online setting).

5.2.2 Online Setting

We describe a deferred data structure that answers a set of domination queries arriving one by one on a set S of points, progressively computing the maxima set of S. Such deferred data structures takes advantage of some measure of the input order and the input structure, as well as of some measure of the query order and the query structure. It is in the pointer-machine model.

Given a set S of n data points and the maxima sets of the ζ smooth sequences into which S can be partitioned, the `Full Synergy Domination` deferred data structure is composed of a finger search tree F, as described by Brodal [16], in which the structure marks the blocks of consecutive points of the maxima set of S that have been computed in the process of answering the queries; and for each identified block B, the data structure stores pointers to the portions of the maxima sets that have non-dominated points to the left and to the right of B, and to the beginning and end of the block B in the maxima set that contains it.

Theorem 15. *Consider an instance I formed by n data points that can be partitioned into ζ smooth sequences, and q query points. Let C be a maxima partition certificate of minimum size of the union of the ζ maxima sets. Let β be the number of blocks of consecutive points in C. Let $\mathcal{H}_B(I)$ be the block dominance entropy of I. Let $d_1, \dots, d_{q-1}$ be the sequence where d_j is the number of identified blocks between the block that answers the j-th query and the one that answers the $j+1$-th query before starting the steps to answer the $j+1$-th query, for $j \in [1..q-1]$. Let $g_1, \dots, g_q$ be the sizes of the blocks in which the algorithm obtains the final answer to the queries. The number of coordinate comparisons performed by the `Full Synergy Domination` deferred data structure while answering the q online domination queries is within*

$$O\left(n + \beta\mathcal{H}_B(I) + \sum_{i=1}^{q-1} \log d_i + \sum_{j=1}^{q} \log g_j\right) \subseteq O(n + \beta \log \gamma + q \log h),$$

where γ is the number of dominance block points of I, and h is the number of points in the maxima set of S.

Proof. The algorithm answers a new domination query by searching in F for the nearest blocks A and B of M to the left and to the right of the query point p, respectively. If p_x (resp., p_y) are in the range of x-coordinates (resp., y-coordinates) of the points of A or B, then the algorithm answer the domination query in A or B, through a binary search in them. Otherwise, the structure applies the same steps as the algorithm `Synergy Domination` in the points between the blocks A and B to answer the query. The algorithm inserts in F each block of M identified in the process of answering the query, in constant time, because all these blocks are in the vicinity of the finger returned in the first step. The structure stores the respective pointers to positions to the left and right of these identified blocks in the ζ maxima sets. □

In the following section, we consider the extension of some of the results described above to the support of point membership queries on the convex hull of a set of data points.

5.3 Supporting Point Membership Queries on the Convex Hull of a Set of Planar Points

In this section, we extend the results about the synergistic computation of the convex hull (described in Section 4.3.2), and the results about supporting domination queries (described in Section 5.2) to support *point membership* queries on the convex hull of a set of data points (see Section 2.5.2 page 21 for the definition of the problem).

We describe solutions for supporting point membership queries in the offline setting (i.e., when all the queries are known from the beginning) in Section 5.3.1, and in the online setting (i.e., when the queries arrive one by one) in Section 5.3.2.

In the offline setting, we describe the algorithm `Synergy Membership` that supports several point membership queries at once (Section 5.3.1), inspired by the algorithm `Quick Synergy Hull` (described in Section 4.3.2). The algorithm `Synergy Membership` takes advantage of some measure of the input order and the input structure in the data points, as defined in Section 4.3.2, and of the query structure (captured by the concepts of *upper and lower block entropies* in Definition 21).

In the online setting, when the queries arrive one by one, we describe the `Full Synergy Membership` deferred data structure that computes the convex hull of the data points in a "lazy" way while answering the queries. It takes advantage of some measure of the input order and the input structure of the data points, of the query structure, as well as of some measure of the query order (see Section 5.3.2 for the details of the analysis).

5.3.1 Offline Setting

Given a set S of data points and a set Q of query points, the algorithm `Synergy Membership` follows the same first steps of the algorithm `Quick Synergy Hull` (described in Section 4.3.2). A minor difference in relation to the support of domination queries is that the algorithm `Synergy Membership` needs to partially compute both the upper and lower hulls of S in order to answer the point membership queries. The algorithm `Synergy Membership` first partitions S into κ simple polygonal chains, and computes the upper and lower hulls of such simple polygonal chains, both steps in time linear in the number of points in S. It then interweaves the "lazy" computation of the upper hull U and lower hull L of S: in one of the

Algorithm 7 `Synergy Membership`

```
Input: A set $S$ of data points and a set $Q$ of query points
Output: The subset of points in $Q$ that are inside the convex hull of $S$
1: Compute the $\kappa$ upper and lower hulls of the simple polygonal chains into which $S$ can
   be partitioned;
2: Compute the median $\mu_U$ (resp., $\mu_L$) of the slopes of the middle edges of the $\kappa$ upper
   (resp., lower) hulls;
3: Identify the "pivot" point $p_U$ (resp., $p_L$) that has a supporting line of slope $\mu_U$ (resp.,
   $\mu_L$) in the upper (resp., lower) hulls;
4: Partition the $\kappa$ upper (resp., lower) hulls by the vertical line through $p_U$ (resp., $p_L$);
5: For each upper (resp., lower) hull $V$, compute the two tangents of $V$ through $p_U$ (resp.,
   $p_L$), and discard the blocks of consecutive points below (resp., above) the line segments
   determined by the points of tangency;
6: Identify a block $A$ (resp., $B$) in the upper hull (resp., lower hull) containing $p_U$ (resp.,
   $p_L$) that forms part of the upper (resp., lower) hull of $S$;
7: Partition $S$ and $Q$ into vertical strips by the vertical lines through the leftmost and the
   rightmost points of $A$ and $B$;
8: Answer the queries on the strips in which the entire portions of the upper and lower hulls
   of $S$ are known;
9: Recurse on the upper hulls to the left and to the right of the block $A$, and on the lower
   hulls to the left and to the right of the block $B$, using the corresponding query points.
```

upper hulls, it computes a block A of consecutive points that forms part of U, and in one of the lower hulls, it computes a block B of consecutive points that forms part of L. The vertical lines through the leftmost and rightmost points of the blocks A and B partition the data points and the query points into (at most 5) vertical strips. If in one of such strips, the entire portions of both U and L have been already computed, then the algorithm answers the queries and stops the computation in the strip. If only the entire portion of U or L has been computed, then the algorithm can answer if the query points inside the strip are below U or above L. It then recurses on the upper (resp., lower) hulls of the simple polygonal chains and its corresponding query points to the left and right of A (resp., B). It maintains the query points sorted by x-coordinates. See Figure 5.5 for a graphical illustration of those steps. A formal description of the algorithm `Synergy Membership` is given in Algorithm 7.

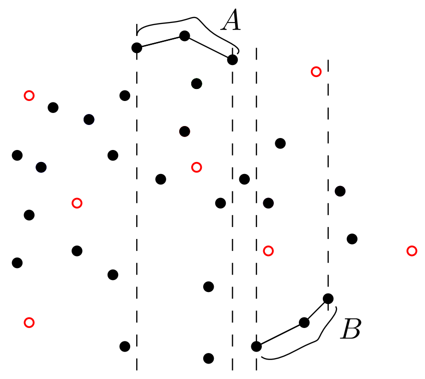


Figure 5.5: An instance of the problem of supporting point membership queries on the convex hull of a set of data points. The black solid disks mark the data points. The red open disks mark the query points. The state of the algorithm `Synergy Membership` after the computation of the block A of the upper hull of the data points and the block B of the lower hull of the data points. The vertical lines through the leftmost and rightmost points of the blocks A and B are marked with dashed lines.

Similarly to the analysis of the algorithm `Synergy Domination` (described in Section 5.2.1), we analyze the time complexity of the algorithm `Synergy Membership` in function of a partition of the data points into vertical strips determined by the positions of the query points. Our analysis is similar to the one described by Sen and Gupta [61] for computing the convex hull of a set S of planar points (described in Section 2.4.1), in which the points of S are partitioned into vertical strip by the vertical lines through the points in the convex hull of S. First, we define a measure of difficulty for membership queries.

Definition 21. *Consider an instance I formed by a set S of n data points and q query points. Let κ be the minimum number of simple polygonal chains into which S can be partitioned. Let C and D be hull partition certificates of the union of the upper and lower hulls of such κ simple polygonal chains, respectively. Let U and L be the upper hull and lower hull of S, respectively. Let $A_1, \dots, A_\gamma$ be the γ blocks of consecutive points of U (such that $\gamma \leq q$) in C such that all the q query points are inside the vertical strips determined by the leftmost and rightmost points of each of such blocks. Let $B_1, \dots, B_\lambda$ be the λ blocks of consecutive points of L (such that $\lambda \leq q$) in D such that all the q query points are inside the vertical strips determined by the leftmost and rightmost points of each of such blocks. Let $n_1, \dots, n_{2\gamma+1}$ be the numbers of blocks in C in a partition of the upper hulls into vertical strips determined by the leftmost and rightmost points of the blocks $A_1, \dots, A_\gamma$. Let $m_1, \dots, m_{2\lambda+1}$ be the numbers of blocks in D in a partition of the lower hulls into vertical strips determined by the leftmost and rightmost points of the blocks $B_1, \dots, B_\lambda$. (Such that $\sum_{i=1}^{2\gamma+1} n_i + \sum_{j=1}^{2\lambda+1} m_j \leq n$.) Define the* upper block entropy *$\mathcal{H}_B(U)$ of I to be $\mathcal{H}(n_1, \dots, n_{2\gamma+1})$. Define the* lower block entropy *$\mathcal{H}_B(L)$ of I to be $\mathcal{H}(m_1, \dots, m_{2\lambda+1})$. Define the* solution block length *b of I to be the sum of the number of points in the blocks $A_1, \dots, A_\gamma$ and $B_1, \dots, B_\lambda$.*

We analyze the time complexity of the algorithm `Synergy Membership` on an instance I in function of the lower and upper block entropies of I:

Theorem 16. *Given an instance I formed by a set S of n data points and q query points. Let h be the number of points in the convex hull of S. Let $\mathcal{H}_B(U)$ be the upper block entropy of I. Let $\mathcal{H}_B(L)$ be the lower block entropy of I. Let b be the solution block length of I. The time complexity of the algorithm* ***`Synergy Membership`*** *to answer the q offline point membership queries is within*

$$O\left(n(\mathcal{H}_B(L) + \mathcal{H}_B(U)) + q \log \frac{b}{q}\right) \subseteq O\left(n \log(\min\{h, q\}) + q \log \frac{h}{q}\right).$$

Proof. The analysis is similar to the ones in the proofs of Lemma 13 and Theorem 14. The difference is that instead of using a k-d tree to supporting range searching, the algorithm `Synergy Membership` answers the queries inside a vertical strip by performing doubling searches for the x-coordinates of the points in Q in the list of x-coordinates of points in the block that determined the strip. The term $q \log \frac{b}{q}$ corresponds to the worst-case complexity of such searches. □

In the following section, we extend the algorithm `Synergy Domination` to a deferred data structure supporting online point membership queries on the convex hull of a set of points.

5.3.2 Online Setting

We describe a deferred data structure that supports online point membership queries (i.e., arriving one by one) on the convex hull of a set of points S, progressively computing the convex hull of S. It takes advantage of some measure of the input order and the input structure, as well as of some measure of the query order and the query structure.

Given a set S of n data points and q query points that arrive one by one, the `Full Synergy Membership` deferred data structure partitions S into κ simple polygonal chains, and computes the convex hulls of such chains. Let U and L be the upper and lower hull of S, respectively. In the process of answering the queries, the structure identified some blocks of consecutive points in the upper and lower hulls of the simple polygonal chains that form part of U and L, respectively. It uses two finger search trees F_U and F_L, as described by Brodal [16], to mark them. In F_U, the structure marks the blocks that form part of U. In F_L, the structure marks the blocks that form part of L. For each block B found that form part of U (resp., L), the data structure stores pointers to the portions of the upper (resp., lower) hulls of the simple polygonal chains that have non-discarded points to the left and to the right of B, and to the beginning and end of the block B in the upper (resp., lower) hull that contains it.

Theorem 17. *Consider an instance I of n data points that can be partitioned into κ simple polygonal chains, and q point membership queries. Let U and L be the upper and lower hull of the data points, respectively. Let $\mathcal{H}_B(U)$ be the upper block entropy of I. Let $\mathcal{H}_B(L)$ be the lower block entropy of I. Let $d_1, \dots, d_{q-1}$ be the sequence where d_i is the number of identified blocks that form part of U between the block that answers the i-th query and the one that answers the $i+1$-th query before starting the steps to answer the $i+1$-th query, for $i \in [1..q-1]$. Let $e_1, \dots, e_{q-1}$ be the sequence where e_j is the number of identified blocks that form part of L between the block that answers the j-th query and the one that answers the $j+1$-th query before starting the steps to answer the $j+1$-th query, for $j \in [1..q-1]$. Let $f_1, \dots, f_q$ be the sizes of the blocks in U in which the algorithm obtains the answers to the queries. Let $g_1, \dots, g_q$ be the sizes of the blocks in L in which the algorithm obtains the answers to the queries. The time complexity of the `Full Synergy Membership` deferred data structure to answer the q online queries is within*

$$O\left(n(\mathcal{H}_B(U) + \mathcal{H}_B(L)) + \sum_{i=1}^{q-1}(\log d_i + \log e_i) + \sum_{j=1}^{q}(\log f_j + \log g_j)\right)$$
$$\subseteq O\left(n \log(\min\{h, q\}) + q \log \frac{n}{q}\right).$$

Proof. The algorithm answers a new point membership query p by searching in F_U for the already computed blocks A_U and B_U closest to p: the one to the left of p, and the one to the right of p, respectively. In F_L, the algorithm also searches for the already computed blocks

A_L and B_L closest to p: the one to the left of p, and the one to the right of p, respectively. If p_x is in the range of the x-coordinates of the points of A_U or B_U, then the structure can answer whether p is below or above U, through a binary search in the block. A symmetric situation arise if the p_x is in the range of the x-coordinates of the points of A_L or B_L, then the structure can answer whether p is above or below L, through a binary search in the block. Otherwise, if p_x is not in the range of the x-coordinates of the points in A_U or B_U, then the structure applies the same steps as the algorithm `Synergy Membership` in order to answer the query in the portions of the upper hulls to the right of A_U and to the left of B_U. Symmetrically, if p_x is not in the range of the x-coordinates of the points in A_L or B_L, then the structure applies the same steps as the algorithm `Synergy Membership` in order to answer the query in the portions of the lower hulls to the right of A_L and to the left of B_L. In F_U and F_L, the algorithm inserts each block of U and L, respectively, identified in the process of answering the query, in constant time, because all such blocks are in the vicinity of the finger return in the first steps. The structure stores the respective pointers to positions to the left and to the right of each of such identified blocks in the upper and lower hulls of the simple polygonal chains. □

In the following, we discuss some ideas to improve the results described in this chapter.

5.4 Discussion

A perspective for future work is to refine the synergistic results about supporting point membership queries on the convex hull of a set of planar points described in this section. Afshani et al. [1] described an upper hull algorithm that takes advantage of a partition of the points into enclosing triangles completely below the upper hulls of the points. We only managed to refine the analysis of the algorithm and the deferred data structure supporting point membership queries in terms of a partition of the blocks that form the upper hulls of the simple polygonal chains into vertical strips (inspired by similar results described by Sen and Gupta [61]). Such partition is more restrained than the one considered by Afshani et al. [1]. It is not clear how we can measure the interactions between the query points and the data points if we consider a partition of the upper hulls of the simple polygonal chains determined by the arguments that form a hull partition certificate.

Even though our synergistic data structures proved to be more efficient when supporting queries than previous data structures that take advantage only of a subset of the features considered in this work, we do not know if our data structures are the best we can hope for solving these problems. One step in this direction is to provide lower bounds for these problems in terms of some of the measures that we have defined here. This is a very interesting and challenging task because of the (many) parameters involved in our analyses.

We describe algorithms and data structures supporting rank and select queries on multisets. However, we only support "membership" queries in Computational Geometry: domination queries and point membership queries. But supporting rank and select queries on the maxima set (resp., the convex hull) of a set of planar points implies computing all the maximal points (resp., all the vertices of the convex hull) to the left of the query, which result in a very different situation to the support of rank and select queries on multisets.

This concludes the description of our synergistic results. In the next chapter, we discuss more general issues left open for improvement.

Chapter 6

Discussion

We have shown that it is possible to take advantage, at the same time, of features that depend on the order in which the input data is given and of features that depend on the structure of the input data for three fundamental problems in Computer Science. We further extended these results to the context of supporting queries, and described solutions that, in addition to taking advantage of some features of the input data, also take advantage of some features of the order in which the queries are given and of the structure of the queries. On large classes of instances, such solutions improve the best previous solutions by more than a constant factor while never performing worse by more than a constant factor.

In the context of supporting online rank and select queries on multisets, the deferred data structures described in this work take advantage of the runs in the multiset, a measure that depend on the order in which the elements are given. In this context, the concept of runs was introduced previously by Kaligosi et al. [44] and Barbay et al. [6], but for a different purpose than the refined analysis of the complexity presented in this work. In the context of supporting queries in Computational Geometry, other queries have been considered [1]. We clarify the differences and the research perspectives that these facts suggest in Section 6.1, and other perspectives for future research in Sections 6.2, 6.3, and 6.4, where we discuss the support of other queries to the ones considered in this work. In Section 6.5, we state a general paradigm for synergistic techniques, and discuss its application to the problems of computing the Delaunay triangulation and the Voronoi diagram of a set of planar points. At a metalevel, we discuss the importance of categorizing techniques of multivariate analysis of algorithms in Section 6.6.

6.1 Comparison with Previous Work

Kaligosi et al.'s multiselection algorithm [44] and Barbay et al.'s deferred data structure [6] use the very same concept of runs as the one described in this work. The difference is, the algorithms and data structures described in Chapters 4 and 5 *detect* the existing runs in the input 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. At best, a combination of both approaches could shave a constant factor off the number of comparisons performed by the sorting and the multiselection algorithms, and by the deferred data structures supporting rank and select queries on multisets.

Johnson and Frederickson [34] described an algorithm answering a single select query in a set of sorted arrays of sizes $r_1, r_2, \ldots, r_\rho$, in time within $O(\sum_{i=1}^{\rho} \log r_i)$. When there is a single query, using their algorithm on pre-existing runs outperforms the deferred data structures described in Section 5.1.2. Yet it is not clear how to generalize their algorithm into a deferred data structure in order to support more than one query, so that the deferred data structure takes advantage of the information obtained by answering previous queries. The difference is somehow negligible as the cost of such a query is anyway dominated by the cost ($n-1$ comparisons) of partitioning the input into runs. To this date, the generalization of Johnson and Frederickson's algorithm into a deferred data structure which optimally supports any quantity of queries is still an open problem.

In the following sections, we discuss the support of queries in Computational Geometry different from those considered in this work.

6.2 Other Queries in Computational Geometry

Afshani et al. [1], among other results, described a general framework to study online versions of point location and related problems. They considered the following online problem: given a set S of n points in $\mathbb{R}^d$ and a mapping function M from points in $\mathbb{R}^d$ to "answers" in some space for some constant d, build a data structure to compute $M(q)$ for any query point $q \in \mathbb{R}^d$, while trying to minimize the average query cost over all the points in S. Through their framework they described a solution to such a problem assuming the existence of an oracle data structure that answers the following queries: given a point $q \in \mathbb{R}^d$, answer $M(q)$ in $O(\log m + \kappa)$ worst-case time for output of size κ, where m is a parameter describing the size of M, and given a convex polyhedral cell γ of size a, test if every point q in γ have the same answer $M(q)$ in time within $O(am^{1-\alpha})$, where $\alpha > 0$. In a preprocessing algorithm, the framework builds a tree data structure T that represents a partition of $\mathbb{R}^d$ by recursively subdividing it into b polyhedral regions. The preprocessing algorithm subdivides each region R until every point $q \in S \cap R$ has the same answer. In a query algorithm, the framework searches in T the answer to the query. If the preprocessing time is not considered, they proved matching upper and lower bounds for the average query cost over all the points in S, in the order-oblivious setting. They applied such framework to online versions of several problems: point location queries in 2D, half-space range reporting queries in 2D and 3D, dominance reporting queries in 2D and 3D, orthogonal range reporting/counting queries in 2D. These queries are the most studied and the ones that appear the most frequently in practice.

In a more general setting, S can be replaced by a probability distribution, and the goal is to bound the expected query time for a point randomly chosen from S. In this case, the preprocessing time depends on the probability distribution, and could be overkill if we only consider a few queries. Therefore, a deferred data structure that "lazily" builds the tree data structure while answering the queries would take advantage of the number of queries and of the probability distribution S to minimize the expected query time, including any preprocessing. For each probability distribution, the issue lies in analyzing how much time the data structure spends partitioning the space in a "lazy" way while answering the queries. We leave the generalization of our deferred data structure results to these problems as a topic of future work.

6.3 Compressed Data Structures in Computational Geometry

In Section 5.1.3, we described compressed data structures supporting rank and select on multisets, which are directly inspired by our synergistic (analysis of) sorting algorithms. A natural question arises about whether our synergistic (analysis of) algorithms for computing maxima sets and convex hulls also yield similar compressed data structures supporting domination queries on a set of planar points and point membership queries on the convex hull of a set of planar points, respectively. The answer to such question is "yes", but such compressed data structures would encode the maxima set/convex hull partition certificate as opposed to the maxima set/convex hull itself, which might have some applications when the data points are added, removed or moved, but which has much less applications than the compressed data structures on multisets.

6.4 Range Queries on Multisets

In another line of though, consider an array $A[1..n]$ of n comparable objects. A *range minimum query* [14] consists of a pair of integers i and j such that $1 \leq i \leq j \leq n$, and is answered by $\texttt{RminQ}_A(i,j)$, the leftmost position of a minimum in $A[i..j]$. Such queries have a wide range of applications in various data structures and algorithms, including text indexing [32], pattern matching [22], and more elaborate kinds of range queries [20]. If the array A is formed by repeated elements, then it is possible to define more specific queries related to the $\texttt{RminQ}_A(i,j)$ operation. Given m elements of minimum values of ranks $r_1,\dots,r_m$ in the range $A[i..j]$, the *range leftmost minimum queries* $\texttt{RLminQ}_A(i,j)$ returns r_1, the *range rightmost minimum queries* $\texttt{RRminQ}_A(i,j)$ returns r_m, and *range k-th minimum query* $\texttt{RKminQ}_A(i,j)$ returns r_k. Related queries are *previous smaller value* $\texttt{PSV}_A(i)$ and *previous larger value* $\texttt{PLV}_A(i)$, which are answered by the positions of the nearest smaller and larger values among $A[1..i]$, respectively. By analogy the operations *range maximum query* $\texttt{RmaxQ}_A(i,j)$, *next smaller value* $\texttt{NSV}_A(i)$, and *next larger value* $\texttt{NLV}_A(i)$ can be defined.

Fischer [30] described a non-systematic succinct index (which does not access the original data when answering queries) using $2n + o(n)$ bits and supporting $\texttt{RminQ}_A(i,j)$ in zero accesses to A and constant accesses to the index, which can be built in time within $O(n)$. Gawrychowski and Nicholson [35], among other results, described an index that supports $\texttt{RminQ}_A(i,j)$ and $\texttt{RmaxQ}_A(i,j)$ on A in constant time using $3n + o(n)$ bits. Fischer [31] described a non-systematic succinct index using $2.54n + o(n)$ to support $\texttt{PSV}_A(i), \texttt{NSV}_A(i)$ and $\texttt{RminQ}_A(i,j)$. By combining both approaches, Jo and Rao [43] described encodings that support a wide range of queries: an encoding using $3.322n + o(n)$ bit that supports $\texttt{RminQ}_A(i,j)$, $\texttt{RmaxQ}_A(i,j)$, $\texttt{RRminQ}_A(i,j)$, $\texttt{RRmaxQ}_A(i,j)$, $\texttt{PSV}_A(i)$ and $\texttt{PLV}_A(i)$ in constant time, and an encoding using $4.585n + o(n)$ bits that supports $\texttt{RminQ}_A(i,j)$, $\texttt{RRminQ}_A(i,j)$, $\texttt{RLminQ}_A(i,j)$, $\texttt{RKminQ}_A(i,j)$, $\texttt{PSV}_A(i)$, $\texttt{NSV}_A(i)$, $\texttt{RmaxQ}_A(i,j)$, $\texttt{RRmaxQ}_A(i,j)$, $\texttt{RLmaxQ}_A(i,j)$, $\texttt{RKmaxQ}_A(i,j)$, $\texttt{PLV}_A(i)$, $\texttt{NLV}_A(i)$ in constant time.

In the worst case over all possible arrays of n comparable objects, any fully separated index requires at least $2n$ bits in order to support $\texttt{RminQ}_A(i,j)$ queries. Taking advantage of the repetitions in multisets is unlikely to yield much better results for indexes that only support $\texttt{RminQ}_A(i,j)$, from the point of view of these indexes, instances varying in repetitions such as $(2,1,3)$ and $(2,1,2)$ are indistinguishable.

Those results suggest the following questions:

1. Is there a compressed index for an ordered input A that supports the operations $\texttt{PSV}_A(i), \texttt{NSV}_A(i)$ and $\texttt{RminQ}_A(i,j)$, while taking advantage of the local and global order and of the repetitions in A?

2. Is there a compressed index for an ordered input A that supports the operations $\texttt{RminQ}_A(i,j)$ and $\texttt{RmaxQ}_A(i,j)$, while taking advantage of the local and global order and of the repetitions in A?

3. Is there a compressed index for A that supports a wide range of operations, while taking advantage of the local and global order and of the repetitions in A in a synergistic way, so that it takes less space on instances where it can take advantage of both at the same time?

The block decomposition used in the analyses of the algorithms `Sorted Set Union` and `Quick Set Union` (seen in Section 3.1) is the basis of the compressed data structures for supporting rank and select queries on multisets (described in Section 5.1.3). Such decomposition contains all the information to sort the entire multiset, which can be much more than what is needed to answer range minimum queries and related queries. A new approach therefore is needed to answer the previous questions.

6.5 A General Paradigm for Synergistic Techniques

We partially answered a question suggested by Afshani et al. [1] related to a unified theory about order-dependent and order-oblivious algorithms, by describing solutions taking advantage both of some measures of the input order and the input structure.

We conjecture that synergistic techniques taking advantage of more than one "easiness" aspect will be of practical importance if the gap between theoretical analysis and practice is to ever be reduced.

We proved that it is possible to take advantage at the same time of characteristics that depend on the order and on the structure of the data, such that on some large classes of instances, these synergistic solutions outperform any solution that only takes advantage of one of those features. We described features essential for a problem to be amenable to this approach. If a problem can be partitioned efficiently into "simpler" instances taking advantage of the input order, and if the structure resulting from such simpler instances can be merged efficiently taking advantage of the input structure, then our synergistic paradigm can be applied to the problem. Using this technique, on large classes of instances, our solutions for sorting multisets, and computing the maxima set and the convex hull of planar points, perform order of magnitude better than previous solutions.

Problems that could fit into this paradigm are the computation of the Delaunay triangulation and its dual, the Voronoi diagram, of a set of planar points. These problems are natural extensions to the problems we considered in this work.

Concerning the partitioning phase of our synergistic approach, it is possible to take advantage of some measure of the input order when computing the Delaunay triangulation and the Voronoi diagram of a set of planar points. Djidjev and Lingas [26] defined a *monotone histogram* as a sequence of points sorted with respect to two orthogonal directions. They described an algorithm which, given a monotone histogram, computes the Voronoi diagram (and hence the Delaunay triangulation) of a sequence of points in linear time. This suggests a way of partitioning the input into subsequences such that the Voronoi diagram and the Delaunay triangulation of each subsequence can be computed in linear time in its length.

Concerning the merging phase of our synergistic approach, Kirkpatrick [46] described an algorithm for the merging of two arbitrary Voronoi diagrams. Given the Voronoi diagrams of two disjoint point sets P and Q, the algorithm finds the Voronoi diagram of $P \cup Q$ in time within $O(|P|+|Q|)$. The plane is partitioned into points closer to P, points closer to Q, and points equidistant from P and Q. The points equidistant from P and Q are defined as the *contour* separating P and Q. The contour is composed of straight line segments: it is formed from the edges of the Voronoi diagram of $P \cup Q$ that separates the points in P from the points in Q. Inside the region of points closer to P (resp., Q) the Voronoi diagram of $P \cup Q$ and the Voronoi diagram of P (resp., Q) are identical. Thus, the merging of two Voronoi diagrams can be seen as the process of cutting the Voronoi diagrams of P and Q along the contour. This leads to a divide-and-conquer algorithm for constructing the Voronoi diagram of an n-point set and hence for computing the Delaunay triangulation in time within $O(n \log n)$. This suggests that there are easy instances in which the merging of these structures can be done efficiently.

6.6 Importance of the Parameterization of Structure and Order

The computational complexity of most problems is studied in the worst case over instances of fixed size n, for n asymptotically tending to infinity. This approach was refined for NP-hard problems under the term "parameterized complexity" [33], for polynomial problems under the term "Adaptive (Analysis of) Algorithms" [28, 54], and more generally for data encodings under the term of "Data Compression" [7], for a wide range of problems and data types.

Such a variety of results has motivated various classifications, in the context of NP-hard problems with a theory of Fixed Parameter Tractability [33], and in the context of sorting in the decision tree model with a theory of reduction between parameters [59]. We introduced other perspectives from which to classify algorithms and data structures. Through the study of the sorting of multisets, and the computation of the maxima set and the convex hull, the two latter in the plane, according to the potential "easiness" in both the order and the values in the input, we aimed to introduce a way to classify refined techniques of complexity analysis between the ones considering the input order and the ones considering the input structure; and to show examples of the difficulty of combining both into a single hybrid algorithmic technique. Through the study of the online support of queries according to the potential "easiness" in both the order and the structure in the queries themselves (in addition to the potential easiness in the data being queried), we aimed to introduce such new categorizations which depend on the queries. We predict that such analysis techniques will take on more importance in the future, along with the growth of the gap between practical cases and the worst case over instances of fixed sizes.

Bibliography

[1] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. *Journal of the ACM (JACM)*, 64(1):3:1–3:38, 2017.

[2] H.-K. Ahn and Y. Okamoto. Adaptive algorithms for planar convex hull problems. *IEICE Transactions on Information and Systems (TIS)*, 94-D(2):182–189, 2011.

[3] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1974.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 20–29, New York, NY, USA, 1996. ACM.

[5] J. Barbay and E. Y. Chen. Convex hull of the union of convex objects in the plane: an adaptive analysis. In *Proceedings of the Annual Canadian Conference on Computational Geometry (CCCG)*, 2008.

[6] J. Barbay, A. Gupta, S. R. Satti, and J. Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms (JDA)*, 36:3–17, 2016.

[7] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science (TCS)*, 513:109–123, 2013.

[8] J. Barbay and C. Ochoa. Synergistic solutions for merging and computing planar convex hulls. In *Computing and Combinatorics - 24rd International Conference (COCOON), Qingdao, China, July 2-4, 2018, Proceedings*, 2018.

[9] J. Barbay, C. Ochoa, and P. Perez-Lantero. Refining the analysis of divide and conquer: How and when. 2015. arXiv:1505.02820v3.

[10] J. Barbay, C. Ochoa, and S. R. Satti. Synergistic solutions on multisets. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM), July 4-6, 2017, Warsaw, Poland*, pages 31:1–31:14, 2017.

[11] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences (JCSS)*, 65(1):38 – 72, 2002.

[12] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, Sept. 1975.

[13] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters (IPL)*, 5(3):82–87, 1976.

[14] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing (SICOMP)*, 22(2):221–242, 1993.

[15] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computational System Science (JCSS)*, 7(4):448–461, 1973.

[16] G. S. Brodal. Finger search trees with constant insertion time. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 540–549. Society for Industrial and Applied Mathematics, 1998.

[17] T. M. Chan, J. Snoeyink, and C.-K. Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional voronoi diagrams. *Discrete & Computational Geometry (DCG)*, 18(4):433–454, 1997.

[18] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the ACM (JACM)*, 17(1):78–86, 1970.

[19] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry (DCG)*, 6(5):485–524, Aug. 1991.

[20] K.-Y. Chen and K.-M. Chao. On the range maximum-sum segment query problem. In *Proceedings of the International Symposium Algorithms and Computation (ISAAC)*, LNCS 3341, pages 294–305. Springer, 2004.

[21] D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[22] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, and T. Walen. Improved algorithms for the range next value problem and applications. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 205–216. IBFI Schloss Dagstuhl, 2008.

[23] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[24] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the* 11^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[25] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments (ALENEX), Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.

[26] H. N. Djidjev and A. Lingas. On computing voronoi diagrams for sorted point sets. *International Journal of Computational Geometry & Applications (IJCGA)*, 05(03):327–337, 1995.

[27] D. P. Dobkin and J. I. Munro. Optimal time minimal space selection algorithms. *Journal of the ACM (JACM)*, 28(3):454–461, 1981.

[28] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[29] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computational System Science (JCSS)*, 66(4):614–656, 2003.

[30] J. Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the Latin American Theoretical INformatics Symposium (LATIN)*, LNCS 6034, pages 158–169. Springer, 2010.

[31] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science (TCS)*, 412(22):2451 – 2456, 2011.

[32] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science (TCS)*, 410(51):5354–5364, 2009.

[33] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[34] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC), April 28-30, 1980, Los Angeles, California, USA*, pages 420–428, 1980.

[35] P. Gawrychowski and P. K. Nicholson. *Automata, Languages, and Programming: 42nd International Colloquium (ICALP) Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, chapter Optimal Encodings for Range Top-k, Selection, and Min-Max, pages 593–604. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[36] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 368–373, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics.

[37] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing (STOC)*, pages 49–60, New York, NY, USA, 1977. ACM Press.

[38] C. A. R. Hoare. Algorithm 64: Quicksort. *Communication of the ACM (CACM)*, 4(7):321, 1961.

[39] C. A. R. Hoare. Algorithm 65: Find. *Communication of the ACM (CACM)*, 4(7):321–322, 1961.

[40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE)*, 40(9):1098–1101, September 1952.

[41] K. E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

[42] R. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters (IPL)*, 2(1):18 – 21, 1973.

[43] S. Jo and S. R. Satti. *Computing and Combinatorics: 21st International Conference, COCOON 2015, Beijing, China, August 4-6, 2015, Proceedings*, chapter Simultaneous Encodings for Range and Next/Previous Larger/Smaller Value Queries, pages 648–660. Springer International Publishing, Cham, 2015.

[44] K. Kaligosi, K. Mehlhorn, J. I. Munro, and P. Sanders. Towards optimal multiple selection. In *Proceedings of the International Conference on Automata, Languages, and Programming (ICALP)*, pages 103–114, 2005.

[45] R. M. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM Journal on Computing (SICOMP)*, 17(5):883–902, 1988.

[46] D. G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 18–27, Washington, DC, USA, 1979. IEEE Computer Society.

[47] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of the Annual Symposium on Computational Geometry (SoCG)*, pages 89–96, New York, NY, USA, 1985. ACM.

[48] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing (SICOMP)*, 15(1):287–299, 1986.

[49] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[50] C. Levcopoulos, A. Lingas, and J. S. B. Mitchell. Adaptive algorithms for constructing convex hulls and triangulations of polygonal chains. In *Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 80–89, London, UK, 2002. Springer-Verlag.

[51] C. Levcopoulos and O. Petersson. Sorting shuffled monotone sequences. *Information and Computation (IC)*, 112(1):37–50, 1994.

[52] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers (TC)*, 34(4):318–325, 1985.

[53] A. A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters (IPL)*, 25(1):11–12, Apr. 1987.

[54] A. Moffat and O. Petersson. An overview of adaptive sorting. *Australian Computer Journal (ACJ)*, 24(2):70–77, 1992.

[55] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science (TCS)*, 438:74–88, 2012.

[56] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing (SICOMP)*, 5(1):1–8, 1976.

[57] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms (JDA)*, 25:2–20, 2014.

[58] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane (revised edition). Technical Report RUU-CS-81-03, Department of Information and Computing Sciences, Utrecht University, 1981.

[59] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics (DAM)*, 59:153–179, 1995.

[60] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[61] S. Sen and N. Gupta. Distribution-sensitive algorithms. *Nordic Journal on Computing (NJC)*, 6:194–211, 1999.

[62] J. M. Steele and A. C. Yao. Lower bounds for algebraic decision trees. *Journal of Algorithms (JoA)*, 3(1):1 – 8, 1982.

[63] T. Takaoka. Partial solution and entropy. In R. Královič and D. Niwiński, editors, *Mathematical Foundations of Computer Science (MFCS) 2009: 34th International Symposium, Novy Smokovec, High Tatras, Slovakia, August 24-28, 2009. Proceedings*, pages 700–711, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.