An Almost Optimal Edit Distance Oracle

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel

Paweł Gawrychowski
University of Wrocław, Poland

Shay Mozes
The Interdisciplinary Center Herzliya, Israel

Oren Weimann
University of Haifa, Israel

Abstract

We consider the problem of preprocessing two strings S and T, of lengths m and n, respectively, in order to be able to efficiently answer the following queries: Given positions i, j in S and positions a, b in T, return the optimal alignment score of $S[i\,..\,j]$ and $T[a\,..\,b]$. Let $N = mn$. We present an oracle with preprocessing time $N^{1+o(1)}$ and space $N^{1+o(1)}$ that answers queries in $\log^{2+o(1)} N$ time. In other words, we show that we can efficiently query for the alignment score of every pair of substrings after preprocessing the input for almost the same time it takes to compute just the alignment of S and T. Our oracle uses ideas from our distance oracle for planar graphs [STOC 2019] and exploits the special structure of the alignment graph. Conditioned on popular hardness conjectures, this result is optimal up to subpolynomial factors. Our results apply to both edit distance and longest common subsequence (LCS).

The best previously known oracle with construction time and size $\mathcal{O}(N)$ has slow $\Omega(\sqrt{N})$ query time [Sakai, TCS 2019], and the one with size $N^{1+o(1)}$ and query time $\log^{2+o(1)} N$ (using a planar graph distance oracle) has slow $\Omega(N^{3/2})$ construction time [Long & Pettie, SODA 2021]. We improve both approaches by roughly a $\sqrt{N}$ factor.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Shortest paths

Keywords and phrases longest common subsequence, edit distance, planar graphs, Voronoi diagrams

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.39

Category Track A: Algorithms, Complexity and Games

Funding *Panagiotis Charalampopoulos*: Supported by Israel Science Foundation grant 592/17.
Shay Mozes: Partially supported by Israel Science Foundation grant 592/17.
Oren Weimann: Partially supported by Israel Science Foundation grant 592/17.

1 Introduction

String alignment is arguably the most popular problem in combinatorial pattern matching. Given two strings S and T of length m and n, the problem asks to compute the similarity between the strings according to some similarity measure. The two most popular similarity measures are edit distance and longest common subsequence (LCS). In both cases, the classical solution is essentially the same: Compute the shortest path from vertex $(0, 0)$ to vertex (m, n) in the so called *alignment graph* of the two strings. As taught in almost every elementary course on algorithms, computing this shortest path (and hence the optimal alignment of the two strings) can easily be done in $\mathcal{O}(N)$ time where $N = mn$, via dynamic programming. Interestingly, this time complexity cannot be significantly improved assuming popular conjectures such as the strong exponential time hypothesis (SETH) [1, 6, 8]. In fact, by now we seem to have a rather good understanding of the complexity of this problem for

different similarity measures and taking other parameters than the length of the strings into account, see [8].

Substring queries. A natural direction after having determined the complexity of a particular problem on strings is to consider the more general version in which we need to answer queries on substrings of the input string. This has been done for alignment [34, 36], pattern matching [24, 28], approximate pattern matching [17], dictionary matching [14, 15], compression [24], periodicity [27, 28], counting palindromes [33], longest common substring [3], computing minimal and maximal suffixes [5, 26], and computing the lexicographically k-th suffix [4].

Alignment oracles. Consider the shortest path from vertex (i, a) to vertex (j, b) in the alignment graph. It corresponds to the optimal alignment of two substrings: the substring of S between indices i and j and the substring of T between indices a and b. An *alignment oracle* is a data structure that, after preprocessing, can report the optimal alignment score of any two substrings of S and T. That is, given positions i, j in S and positions a, b in T, the oracle returns the optimal alignment score of $S[i\,..\,j]$ and $T[a\,..\,b]$ (or equivalently, the $(i-1, a-1)$-to-(j, b) distance in the alignment graph).

Tiskin [36, 37] considered a restricted variant of the problem, in which the queries are either the entire string S vs. a substring of T or a prefix of S vs. a suffix of T. For such queries, Tiskin gave an $\tilde{\mathcal{O}}(n+m)$-size oracle, that can be constructed in $\tilde{\mathcal{O}}(N)$ time, and answers queries in $\mathcal{O}(\log N/\log\log N)$ time [36]. For the general problem, Sakai [34] (building on Tiskin's work [36]) showed how to construct in $\mathcal{O}(N)$ time an alignment oracle with $\mathcal{O}(n+m)$ query time. In this work we show that, perhaps surprisingly, obtaining such an oracle can be done essentially for free! That is, at almost the same time it takes to compute just the alignment of S and T. More formally, our main result is:

▶ **Theorem 1.** *For two strings of lengths m and n, with $N = mn$, we can construct in $N^{1+o(1)}$ time an alignment oracle achieving either of the following tradeoffs:*

- *$N^{1+o(1)}$ space and $\log^{2+o(1)} N$ query time,*
- *$N \log^{2+o(1)} N$ space and $N^{o(1)}$ query time.*

Planar distance oracles and Voronoi diagrams. The starting point of our work is the recent developments in *distance oracles* for planar graphs. A distance oracle is a compact representation of a graph that allows to efficiently query the distance between any pair of vertices. Indeed, since the alignment graph is a planar graph, the state-of-the-art distance oracle for planar graphs of Long and Pettie [30] (which builds upon [13, 19, 21]) is an alignment oracle with space $N^{1+o(1)}$ and query time $\mathcal{O}(\log^{2+o(1)} N)$. However, the construction time of this oracle is $\Omega(N^{3/2})$. Our main contribution is an improved $N^{1+o(1)}$ construction time when the underlying graph is not just a planar graph but an alignment graph.

Our oracle has the same recursive structure as the planar graph oracles in [13, 21, 30] (in fact, the alignment graph, being a grid, greatly simplifies several technical, but standard, difficulties of the recursive structure). These oracles (inspired by Cabello's use of Voronoi diagrams for the diameter problem in planar graphs [10]) use the recursive structure in order to apply (at different levels of granularity) an efficient mechanism for *point location on Voronoi diagrams*. At a high level, a *Voronoi diagram* with respect to a subset S of vertices (called sites) is a partition of the vertices into $|S|$ parts (called Voronoi cells), where the cell of site $s \in S$ contains all vertices that are closer to s than to any other site in S. A *point location* query, given a vertex v, returns the site s such that v belongs to the Voronoi cell of s.

Our main technical contribution is a polynomially faster construction of the point location mechanism when the underlying graph is an alignment graph. We show that, in this case, the special structure of the Voronoi cells facilitates point location via a non-trivial divide and conquer. Unlike the planar oracles, which use planar duality to represent Voronoi diagrams, the representation and point location mechanisms we develop in this paper are novel and achieve the same query time, while being arguably simpler than those of Long and Pettie.[1]

It is common that techniques are originally developed for pattern matching problems (and in particular alignment problems) and later extended to planar graphs. A concrete example is the use of Monge matrices and unit-Monge matrices. However, it is much less common that techniques are first developed for planar graphs (in our case, the use of Voronoi diagrams) and only then translated to pattern matching problems.

Conditional lower bounds. Any lower bound on the time required to compute an optimal alignment of two strings directly implies an analogous lower bound for the sum of the preprocessing time and the query time of an alignment oracle. In particular, the existence of an oracle for which this sum is $\mathcal{O}(N^{1-\epsilon})$, for a constant $\epsilon > 0$, would refute SETH [6, 8].

In the *Set Disjointness* problem, we are given a collection of m sets $A_1, A_2, \ldots, A_m$ of total size M for preprocessing. We then need to report, given any query pair A_i, A_j, whether $A_i \cap A_j = \emptyset$. The Set Disjointness conjecture [18, 22, 32] states that any data structure with constant query time must use $M^{2-o(1)}$ space. Goldstein et al. [22] stated the following stronger conjecture.

▶ **Conjecture 2** (Strong Set Disjointness Conjecture [22]). *Any data structure for the Set Disjointness problem that answers queries in time t must use space $M^2/(t^2 \cdot \log^{\mathcal{O}(1)} M)$.*

The following theorem implies that, conditioned on the above conjecture, our alignment oracle is optimal up to subpolynomial factors; its proof is identical to that of [3, Theorem 1] as explained in [12].

▶ **Theorem 3** ([3, 12]). *An alignment oracle for two strings of length at most n with query time t must use $n^2/(t^2 \cdot \log^{\mathcal{O}(1)} n)$ space, assuming the Strong Set Disjointness Conjecture.*

Even though the main point of interest is in oracles that achieve fast (i.e. constant, polylogarithmic, or subpolynomial) query-time, the above lower bound suggests to study other tradeoffs of space vs. query-time. In Section 5 we show oracles with space sublinear in N. More formally, we prove the following theorem.

▶ **Theorem 4.** *Given two strings of lengths m and n with $N = mn$, integer alignment weights upper-bounded by w, and a parameter $r \in [\sqrt{N}, N]$ we can construct in $\tilde{\mathcal{O}}(N)$ time an $\tilde{\mathcal{O}}(Nw/\sqrt{r} + m + n)$-space alignment oracle that answers queries in time $\tilde{\mathcal{O}}(\sqrt{N} + r)$.*

For example, if the alignment weights are constant integers, by setting $r = \sqrt{N}$ we obtain an $\mathcal{O}(N^{3/4} + m + n)$-space oracle that answers queries in time $\tilde{\mathcal{O}}(\sqrt{N})$.

Other related works. When the edit distance is known to be bounded by some threshold k, an efficient edit distance oracle can be obtained via the Landau-Vishkin algorithm [29]. Namely, after $\mathcal{O}(n+m)$ time preprocessing of the input strings (oblivious to the threshold k),

[1] We believe that our efficient construction can also be made to work, for alignment graphs, with the dual representation of Voronoi diagrams used in [13, 21, 30], but we think the new representation makes the presentation more approachable as it exploits the structure of the alignment graph more directly.

given a substring of S, a substring of T, and a threshold k, in $\mathcal{O}(k^2)$ time one can decide whether the edit distance of these substrings is at most k, and if so, return it.

Further, after an $\mathcal{O}(n+m)$-time preprocessing, given a substring X of S and a substring Y of T, the starting positions of substrings of Y that are at edit distance at most k from X can be returned in $\mathcal{O}(k^4 \cdot |Y|/|X|)$ time [17].

In a recent work [16] on dynamic string alignment, it was shown that, in the case where the alignment weights are small integers, two strings of total length at most n can be maintained under edit operations in $\tilde{\mathcal{O}}(n)$ time per operation so that an alignment of any pair of substrings can be queried in $\tilde{\mathcal{O}}(n)$ time.

2 Preliminaries

The alignment oracle presented in this paper applies to both edit distance and longest common subsequence (LCS). To simplify the presentation we focus on LCS but the extension to edit distance is immediate.

The LCS of two strings S and T is a longest string that is a subsequence of both S and T. We denote the length of an LCS of S and T by $\texttt{LCS}(S,T)$.

▶ **Example 5.** An LCS of $S = \texttt{acbcddaaeea}$ and $T = \texttt{abbbccdec}$ is $\texttt{abcde}$; $\texttt{LCS}(S,T) = 5$.

For strings S and T, of lengths m and n respectively (we will assume that $n \geq m$), the alignment graph G of S and T is a directed acyclic graph of size $N = \mathcal{O}(mn)$. For every $0 \leq x \leq m$ and $0 \leq y \leq n$, the alignment graph G has a vertex (x,y) and the following unit-length edges (defined only if both endpoints exist):

- $((x,y),(x+1,y))$ and $((x,y),(x,y+1))$,
- $((x,y),(x+1,y+1))$, present if and only if $S[x] = T[y]$.

Intuitively, G is an $(m+1) \times (n+1)$ grid graph augmented with diagonal edges corresponding to matching letters of S and T. See Figure 1. We think of the vertex $(0,0)$ as the top-left vertex of the grid and the vertex (m,n) as the bottom-right vertex of the grid. We shall refer to the rows and columns of G in a natural way. It is easy to see that $\texttt{LCS}(S,T)$ equals $n+m$ minus the length of the shortest path from $(0,0)$ to (m,n) in G.

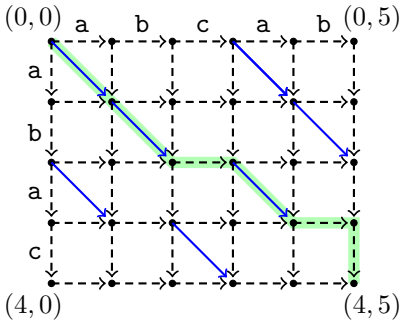

Figure 1 The alignment graph for $S = \texttt{abac}$ and $T = \texttt{abcab}$. We represent the horizontal and vertical edges by dashed black arrows, and the diagonal edges by blue arrows. A lowest scoring $(0,0)$-to-$(4,5)$ path is highlighted in green, it has weight 6 and corresponds to the LCS $\texttt{aba}$ of length $3 = 9 - 6 = |T| + |S| - 6$.

Multiple-source shortest paths. Given a planar graph with N vertices and a distinguished face h, the multiple-source shortest paths (MSSP) data structure represents all shortest

path trees rooted at the vertices of h. It can be constructed in $\mathcal{O}(N \log N)$ time, requires $\mathcal{O}(N \log N)$ space, and can report the distance between any vertex u of h and any other vertex v in the graph in $\mathcal{O}(\log N)$ time. These bounds were first obtained for alignment graphs [35] and then extended to arbitrary planar graphs [11, 25].

The MSSP data structure can be augmented at no asymptotic overhead (cf. [23, Section 5]), to allow for the following. First, to report a shortest u-to-v path ρ in time $\mathcal{O}(|\rho| \log \log \Delta)$, where Δ is the maximum degree of a vertex in G. Second, to support the following queries in $\mathcal{O}(\log N)$ time [21]: Given two vertices $u, v \in G$ and a vertex x of h report whether u is an ancestor of v in the shortest path tree rooted at x, and whether u occurs before v in a preorder traversal of this tree. (We consider shortest path trees as ordered trees with the order inherited from the planar embedding.)

Recursive decomposition. We assume without loss of generality that the length of each of the two strings is a power of 2, and hence the alignment graph is a $(2^a + 1) \times (2^b + 1)$ grid. We consider a recursive decomposition $\mathcal{A}$ of G such that in each level all pieces are of the same rectangular shape. At each level, each piece will be of size $(2^c + 1) \times (2^d + 1)$ for non-negative integers c and d. Consider a piece P of size $(2^c + 1) \times (2^d + 1)$, with $c + d \neq 0$. Assuming without loss of generality that $c \geq d$, in the next level we will partition P to two pieces, each of size $(2^{c-1} + 1) \times (2^d + 1)$, that share the middle row of P. See Figure 2. We view $\mathcal{A}$ as a binary tree and identify a piece P with the node corresponding to it in $\mathcal{A}$.

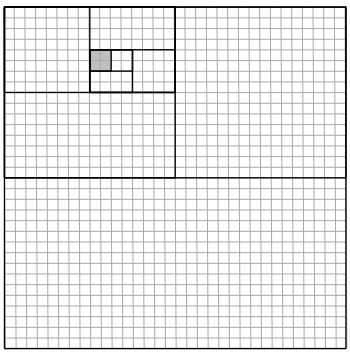

Figure 2 Illustration of pieces in a recursive decomposition of the alignment graph. Diagonal edges are not show to avoid clutter. All rectangular pieces that contain the gray square form a single root-to-leaf path in the tree $\mathcal{A}$.

Consider a piece $P \in \mathcal{A}$. The set ∂P of *boundary vertices* of a piece P consists of those vertices who have neighbours that are not in P. We call the vertices in $P \setminus \partial P$ the *internal vertices* of P. We denote by $\lrcorner(P)$ the set of boundary vertices of P that are either rightmost or bottommost in P, and by $\ulcorner(P)$ the set of boundary vertices of P that are either leftmost or topmost in P. We consider each of $\lrcorner(P)$ and $\ulcorner(P)$ to be ordered, such that adjacent vertices are consecutive and the earliest vertex is the bottom-left one. Therefore, whenever convenient, we refer to subsets of $\lrcorner(P)$ and $\ulcorner(P)$ as sequences. We define the *outside* of P, denoted by P^{out}, to be the set of vertices of $G \setminus (P \setminus \partial P)$ that are reachable from some vertex in P, i.e. the vertices of G that are not internal in P and are to the right or below some vertex of P. Note that any path from a vertex $u \in P$ to a vertex $v \in P^{out}$ must contain at least one vertex from $\lrcorner(P)$, and any path from a vertex $u \notin P$ to a vertex $u \in P$ must contain at least one vertex from $\ulcorner(P)$.

For any $r \in [1, nm]$, an *r-division* of G is a decomposition of G to pieces of size $\mathcal{O}(r)$, each with $\mathcal{O}(\sqrt{r})$ boundary vertices. Clearly, such a decomposition can be retrieved from $\mathcal{A}$, with all nodes being of the same depth. In particular, we will use recursive $(r_t, \ldots, r_1)$-divisions, where for every $i < t$, each piece of the r_i-division must be contained in some piece of the r_{i+1}-division. By convention, we will have r_t being a single piece consisting of the entire graph G. Such a recursive division can be materialized as follows: First, we select the appropriate depth of $\mathcal{A}$ for each r_i-division and mark all nodes of this depth. Then, we contract every edge of $\mathcal{A}$ of the form $(\mathsf{parent}(u), u)$ such that u is unmarked. Such a recursive $(r_t, \ldots, r_1)$-division can thus also be represented by a tree, which we will denote by $\mathcal{T}$.

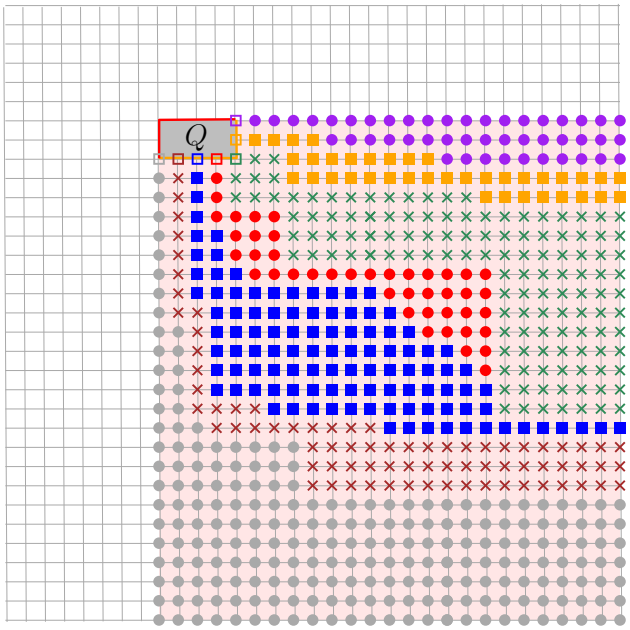

Figure 3 A piece Q (shaded gray). $\ulcorner(Q)$ is indicated by a red line, and $\lrcorner(Q)$ by an orange line. Q^{out} is shaded pink. The Voronoi diagram for Q^{out} with sites $\lrcorner(Q)$ (boxes) is also illustrated. Each site has a distinct color. Vertices in each Voronoi cell are indicated by a matching color.

Voronoi diagrams. Let H be a directed planar graph with real edge-lengths, and no negative-length cycles. Let h be a face of H, and let S be the set of vertices (called *sites*) of h. Each site $s \in S$ has a weight $\omega(s) \geq 0$ associated with it. The additively weighted distance $d^{\omega}(s, v)$ between a site $s \in S$ and a vertex $v \in H$ is defined as $\omega(s)$ plus the length of the shortest s-to-v path in H.

The *additively weighted Voronoi diagram* $\mathsf{VD}(S, \omega)$ of H is a partition of the vertices of H into pairwise disjoint sets, one set $\mathsf{Vor}(s)$ for each site $s \in S$. The set $\mathsf{Vor}(s)$, called the *Voronoi cell* of s, contains all vertices of H that are closer (w.r.t. $d^{\omega}(.,.)$) to s than to any other site in S. If $v \in \mathsf{Vor}(s)$ then we call s the site of v, and say that v belongs to the site s. Throughout the paper, we will only consider additively weighted Voronoi diagrams for the outside P^{out} of a piece $P \in \mathcal{A}$ with sites $S \subseteq \lrcorner(P)$. We next discuss the structure of such Voronoi diagrams.

We resolve ties between sites in favor of the site $s = (x, y)$ for which $(\omega(s), x, y)$ is lexicographically largest. Since the alignment graph is planar, this guarantees that the vertices in $\mathsf{Vor}(s)$ are spanned by a subtree of a shortest paths tree rooted at s: for every vertex $v \in \mathsf{Vor}(s)$, for any vertex u on a shortest s-to-v path, we must have $u \in \mathsf{Vor}(s)$.

Hence, each Voronoi cell is a simply connected region of the plane. The structure of the alignment graph dictates that any shortest path is monotone in the sense that it only goes right and/or down. This property immediately implies the following lemma.

▶ **Lemma 6.** *For any $a \leq c$ and $d \leq f$, if $u = (a, f)$ and $v = (c, d)$ both belong to $\mathsf{Vor}(s)$ then every vertex $w = (b, e)$ with $a \leq b \leq c$ and $d \leq e \leq f$ also belongs to $\mathsf{Vor}(s)$.*

Proof. Suppose $w = (b, e)$ belongs to $\mathsf{Vor}(s')$ for some $s' \neq s$. Since shortest paths only go right and down, the shortest s'-to-w path must cross either the s-to-u path or the s-to-v path, which is a contradiction. ◀

Lemma 6 together with the fact that $\mathsf{Vor}(s)$ is connected implies the following characterization of the structure of $\mathsf{Vor}(s)$, which roughly says that $\mathsf{Vor}(s)$ has the form of a double staircase, as illustrated in Figure 3.

▶ **Corollary 7.** *For any row a and any site s, the vertices of row a that belong to $\mathsf{Vor}(s)$ form a contiguous interval of columns $[i_a, j_a]$. Furthermore, the sequences i_a and j_a are monotone non-decreasing, $i_a \leq i_{a+1} \leq j_a + 1$, and $j_a \leq j_{a+1}$.*

▶ **Corollary 8.** *There is a rightmost vertex $s_{\searrow}$ in $\mathsf{Vor}(s)$ that is also a bottommost one.*

▶ **Corollary 9.** *For every $v \in \mathsf{Vor}(s)$, $\mathsf{Vor}(s)$ contains a path from v to $s_{\searrow}$.*

Our representation of a Voronoi diagram for P^{out} with sites $\lrcorner(P)$ consists of the following. For each $s \in \lrcorner(P)$, we store:

1. the rightmost bottommost vertex $s_{\searrow}$ of $\mathsf{Vor}(s)$, defined in Corollary 8,
2. a vertex $\mathsf{last}(s, s_{\searrow})$ on a shortest s-to-$s_{\searrow}$ path, whose definition will be given later.

3 The Alignment Oracle

In this section, we describe our oracle and prove that its space and query time are as in Theorem 1. In the next section we will show how to construct the oracle in $N^{1+o(1)}$ time.

Consider a recursive $(r_t, \ldots, r_1)$-division of G for some $N = r_t > \cdots > r_1 = \mathcal{O}(1)$ to be specified later. Recall that our convention is that the r_t-division consists of G itself. Further, we set the r_1-division to consist of pieces of size 2×2. We also consider an r_0-division, in which each vertex v of G is a singleton piece. Let us denote the set of pieces of the r_i-division by $\mathcal{R}_i$. Let $\mathcal{T}$ denote the tree representing this recursive $(r_t, \ldots, r_0)$-division, where each singleton piece $\{v\}$ at level 0 is attached as a child of a piece P at level 1 such that $v \in \lrcorner(P)$—this is well-defined for all singletons apart from $\{(0, 0)\}$ and $\{(m, n)\}$, each of which is attached to the single level-1 piece containing it.

The oracle consists of the following. For each $0 \leq i \leq t - 1$, for each piece $P \in \mathcal{R}_i$ whose parent in $\mathcal{T}$ is $Q \in \mathcal{R}_{i+1}$:

1. If $i > 0$, we store an MSSP with sources $\lrcorner(P)$ for the graph obtained from P by flipping the orientation of all edges; we call this the *reverse MSSP of P*.
2. If $i > 0$, we store an MSSP with sources $\lrcorner(P)$ for $Q \setminus (P \setminus \partial P)$.
3. If $i < t - 1$, for each vertex $u \in \lrcorner(P)$ we store $\mathsf{VD}(u, Q)$: the Voronoi diagram for Q^{out} with sites $\lrcorner(Q)$ and additive weights the distances in G from u to these sites.

To complete the description of the oracle it remains to specify the definition of $\mathsf{last}(s, s_{\searrow})$. Before doing so, let us distinguish, for a (source) vertex u and a (target) vertex v, two levels of the recursive division that are of interest. Let R_0 be the singleton piece $\{u\}$. Let $R_1, R_2, \ldots, R_t$ be the ancestors of R_0 in $\mathcal{T}$. Note that $u \in \lrcorner(R_i)$ for a non-empty prefix of

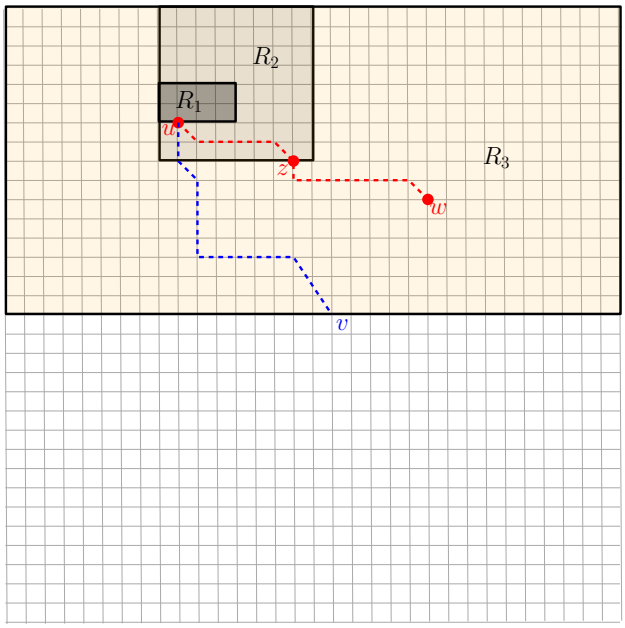

Figure 4 A vertex u and all the pieces of a recursive $(r_4, r_3, r_2, r_1, r_0)$-division that contain u. The piece R_0 consists of just u, and the piece R_4 is the entire alignment graph. Note that $u \in \sqsupset(R_1)$. In this example, $\mathsf{lev}(u) = 1$, $\mathsf{anc}(u, w) = 3$, and $\mathsf{anc}(u, v) = 4$ (because $v \in \partial R_3$). A u-to-w shortest path ρ is shown in dashed red. $\mathsf{last}(u, w)$ is the last vertex of ρ that belongs to $\sqsupset(R_2)$, which is the vertex z. The distance from u to w is $\mathsf{dist}(u, z) + \mathsf{dist}(z, w)$. $\mathsf{dist}(u, z)$ is stored in the reverse MSSP of R_2. $\mathsf{dist}(z, w)$ is stored in the MSSP of $R_3 \setminus (R_2 \setminus \partial R_2)$. Similarly, a u-to-v shortest path ρ is shown in dashed blue. Because $v \in \partial R_3$, $\mathsf{last}(u, v)$ is v itself.

the sequence of ancestors $R_0, R_1, \ldots, R_t$. Let $\mathsf{lev}(u) = \mathrm{argmax}_i\{u \in \sqsupset(R_i)\}$. Further, let $\mathsf{anc}(u, v) = \mathrm{argmin}_i\{v \in R_i \setminus \partial R_i\}$. Note that $\mathsf{anc}(u, v)$ is well defined since $v \in R_t = G$ and $\partial G = \emptyset$. Also note that if v is reachable from u then $\mathsf{lev}(u) < \mathsf{anc}(u, v)$. This is because all vertices in $R_{\mathsf{lev}(u)} \setminus \partial R_{\mathsf{lev}(u)}$ are unreachable from u.

Denote $H = R_{\mathsf{anc}(u,v)-1}$. We define $\mathsf{last}(u, v)$ as any boundary vertex of $\sqsupset(H)$ that lies on a shortest u-to-v path ρ. The idea behind this definition is that $\mathsf{last}(u, v)$ partitions this u-to-v path into a prefix and a suffix, each of which is represented in one of the MSSP data structures stored for H; The prefix of ρ ending at $\mathsf{last}(u, v)$ is represented in the shortest path tree rooted at $\mathsf{last}(u, v)$ in the reverse MSSP of H. The suffix of ρ starting at $\mathsf{last}(u, v)$ is represented in the shortest path tree rooted at $\mathsf{last}(u, v)$ in the MSSP for $H' \setminus (H \setminus \partial H)$ with sources $\sqsupset(H)$, where $H' = R_{\mathsf{anc}(u,v)}$ is the parent of H in $\mathcal{T}$. This allows us to efficiently compute $\mathsf{dist}(u, v)$ (the u-to-v distance) given $\mathsf{last}(u, v)$. See Figure 4. This concludes the description of the oracle.

▶ **Lemma 10.** *The oracle occupies space* $\mathcal{O}\left(N \log^2 N + N \log N \cdot \sum_{i=0}^{t-1} r_{i+1}/r_i\right)$.

Proof. The reverse MSSPs over all pieces of $\mathcal{A}$ require $\mathcal{O}(N \log^2 N)$ space, since $\sum_{P \in \mathcal{A}} |P| = \mathcal{O}(N \log N)$ and since the reverse MSSP of a piece P requires space $\mathcal{O}(|P| \log |P|)$.

For each $i \in (0, t-1]$, for each of the $\mathcal{O}(N/r_i)$ pieces in $\mathcal{R}_i$, we store an MSSP of size $\mathcal{O}(r_{i+1} \log r_{i+1})$. For each $i \in [0, t-1)$, for each of the $\mathcal{O}(N/r_i)$ pieces in $\mathcal{R}_i$, we store $\mathcal{O}(\sqrt{r_i})$ Voronoi diagrams each of size $\mathcal{O}(\sqrt{r_{i+1}})$. The stated bound follows. ◀

Query. We now describe how to answer a distance query $\mathsf{dist}(u, v)$. First, note that if u and v are in the same piece P in $\mathcal{R}_1$ we can report $\mathsf{dist}(u, v)$ in $\mathcal{O}(1)$ time using brute force.

Let R_0 be the singleton piece $\{u\}$. As before, let $R_1, R_2, \ldots, R_t$ be the ancestors of R_0 in $\mathcal{T}$. Since the distance query originates from an LCS query, v must be reachable from u. Let $\ell = \mathsf{lev}(u)$ and $h = \mathsf{anc}(u, v)$. We then have $u \in \lrcorner(R_\ell)$ and $v \in R_\ell^{out}$. We will answer $\mathsf{dist}(u, v)$ by identifying $\mathsf{last}(u, v)$, which lies on $\lrcorner(R_{h-1})$. As explained above, we can then obtain $\mathsf{dist}(u, v)$ from the MSSP data structures stored for R_{h-1}. See Algorithm 1.

Algorithm 1 DIST(u, v)

```
if u and v belong to the same piece in R_1 then
    return the answer by brute force
w ← GetLast(u, v)
return dist(u, w) + dist(w, v)
```

We now show how to implement the procedure GETLAST for finding $\mathsf{last}(u, v)$; see Algorithm 2 for a pseudocode. First, note that if $h = \ell + 1$ we can simply return u as $\mathsf{last}(u, v)$. Hence, in what follows, we assume that $h > \ell + 1$. The procedure GETLAST proceeds in iterations for $i = \ell+1, \ldots, h-1$. At the beginning of the iteration with value i, the procedure has a subset W_{i-1} of $\lrcorner(R_{i-1})$ such that some $w \in W_{i-1}$ belongs to a shortest u-to-v path. We initially set $W_\ell := \{u\}$, which trivially satisfies the requirement for $i = \ell + 1$. For each $w \in W_{i-1}$ the iteration uses a procedure GETNEXTCANDIDATES that adds at most two vertices of $\lrcorner(R_i)$ to W_i. The guarantee is that, if w is a vertex of $\lrcorner(R_{i-1})$ that belongs to a shortest u-to-v path, then at least one of the two added vertices also belongs to a shortest u-to-v path. Since $|W_i| \leq 2|W_{i-1}|$, at the end of the last iteration (the one for $h - 1$), we have a subset W_{h-1} of at most 2^t vertices of $\lrcorner(R_{h-1})$, one of which can be returned as $\mathsf{last}(u, v)$. To figure out which one, for each such vertex w, we use the MSSP data structures to compute $\mathsf{dist}(u, w) + \mathsf{dist}(w, v)$, and return a vertex for which the minimum is attained.

Algorithm 2 GETLAST(u, v)

```
ℓ ← lev(u)
h ← anc(u, v)
W_ℓ ← {u}
for i = ℓ + 1 to h − 1 do
    W_i ← ∅
    for each w ∈ W_{i−1} do
        W_i ← W_i ∪ GetNextCandidates(w, i, v)
return argmin_{w∈W_{h−1}} dist(u, w) + dist(w, v)
```

It remains to describe the procedure GETNEXTCANDIDATES. Consider any $w \in W_{i-1}$. To reduce clutter, let us denote R_i by Q. Since $w \in \lrcorner(R_{i-1})$, the Voronoi diagram $\mathsf{VD}(w, Q)$ for Q^{out} is stored by the oracle. The procedure GETNEXTCANDIDATES finds two sites of $\mathsf{VD}(w, Q)$, one of which is the site of v. Indeed, if w is a vertex on a u-to-v shortest path then the site of v in $\mathsf{VD}(w, Q)$ is a vertex of $\lrcorner(Q)$ on a shortest u-to-v path.

Let $(x_\lrcorner, y_\lrcorner)$ be the bottom-right vertex of Q. Every vertex $v = (x_v, y_v)$ of $Q^{out} \setminus \partial Q$ either has $x_v > x_\lrcorner$ or $y_v > y_\lrcorner$. We describe the case when $x_v > x_\lrcorner$; the case when $y_v > y_\lrcorner$ is analogous. Let Γ denote the set of $s \to s_\searrow$ paths ρ_s stored in $\mathsf{VD}(w, Q)$ according to the order of the sites s along $\lrcorner(Q)$. For every row $x > x_\lrcorner$, let Γ_x denote the subset of paths in Γ that intersect row x, ordered according to the order of the intersection vertices along row x.

▶ **Lemma 11.** *For every x, x' with $x > x'$, Γ_x is a subsequence of $\Gamma_{x'}$.*

Proof. This is a direct consequence of the fact that each path ρ_s goes monotonically down and right, and from the fact that ρ_s and $\rho_{s'}$ are disjoint for $s \neq s'$. ◀

We define the set of *critical rows* to be all rows x such that $(x, y) = s_{\searrow}$ for some $y \in [0, n]$ and $s \in \lrcorner(Q)$. Lemma 11 implies that if we consider the evolution of the sequences Γ_x as x increases from $x_{\lrcorner}$ to m as a dynamic process, changes occur only at critical rows. More precisely, if the row of $s_{\searrow}$ is x for some site s, then $\rho_s \in \Gamma_x$ but $\rho_s \notin \Gamma_{x+1}$. We can therefore maintain the sequences Γ_x in a persistent binary search tree. (A binary search tree can be made partially persistent at no extra asymptotic cost in the update and search times using a general technique for pointer-machine data structures of bounded degree [9].) Initially, the BST stores the sequence $\Gamma_{x_{\lrcorner}+1}$. Then, we go over the critical rows in increasing order, and remove the path ρ_s from the BST when we reach the row of $s_{\searrow}$.

For any $x_{\lrcorner} < x \leq m$, we can access the BST representation of Γ_x by finding the predecessor x' of x among the critical rows, and accessing the persistent BST at time x'.

Let $v = (x, y)$. We say that v is right (left) of a path $\rho_s \in \Gamma_x$ if y is greater (smaller) than any vertex of ρ_s at row x. We will either find a path $\rho_s \in \Gamma_x$ to which v belongs, or identify the last path $\rho_s \in \Gamma_x$ such that v is right of ρ_s. In the former case the site of v is s, and in the latter case the site of v is either s or the successor of s in Γ_x.

Recall that (1) the MSSP data structure, given a root vertex r and two vertices w, z, can determine in $\mathcal{O}(\log N)$ time whether w is left/right/ancestor/descendant of z in the shortest path tree rooted at r, (2) for each shortest path ρ_s represented in $\mathsf{VD}(w, Q)$, the representation contains $\mathsf{last}(s, s_{\searrow})$, and (3) the prefix of ρ_s ending at $\mathsf{last}(s, s_{\searrow})$ is represented in the shortest path tree rooted at $\mathsf{last}(s, s_{\searrow})$ in the reverse MSSP of H (recall that $H = R_{\mathsf{anc}(u,v)-1}$), while the suffix of ρ_s starting at $\mathsf{last}(s, s_{\searrow})$ is represented in the shortest path tree rooted at $\mathsf{last}(s, s_{\searrow})$ in the MSSP for $H' \setminus (H \setminus \partial H)$, where H' is the parent of H.

We perform binary search on Γ_x to identify the path ρ_s such that either $v \in \rho_s$ or ρ_s is the last path of Γ_x that is left of v. Focus on a step of the binary search that considers a path ρ_s. Denote $\mathsf{last}(s, s_{\searrow}) = (x_b, y_b)$. If $y < y_b$, we query the MSSP structure that contains the prefix of ρ_s, and otherwise we query the MSSP data structure that contains the suffix of ρ_s. In either case, the query either returns that v is on ρ_s or tells us whether v is left or right of ρ_s. In the former case we conclude that the site of v is s. In the latter case we continue the binary search accordingly. Each step of the binary search takes $\mathcal{O}(\log n)$ time. Note that $\log n = \mathcal{O}(\log N)$. Thus, the binary search takes $\mathcal{O}(\log^2 N)$ time, and when it terminates we have a site s that is either the site of v or the site such that ρ_s is the last path of Γ_x that is left of v. This implies that the site of v is either s or the successor of s in Γ_x, and concludes the description of GetNextCandidates.

▶ **Lemma 12.** *The oracle answers distance queries in time $\mathcal{O}(2^t \log^2 N)$.*

Proof. First, $\mathsf{lev}(u)$ and $\mathsf{anc}(u, v)$ can be (naively) computed in $\mathcal{O}(\log N)$ time by going over the ancestors of $\{u\}$ in $\mathcal{T}$: for each (rectangular) ancestor piece R of $\{u\}$, in $\mathcal{O}(1)$ time, we can retrieve the coordinates of R's corners and check whether $v \in R \setminus \partial R$ using v's coordinates. Overall, for a $\mathsf{dist}(u, v)$ query, we make $\mathcal{O}(2^t)$ calls to GetNextCandidates, each requiring $\mathcal{O}(\log^2 N)$ time, for a total of $\mathcal{O}(2^t \log^2 N)$ time. Finally, we make $\mathcal{O}(2^t)$ queries to the MSSP data structures, requiring $\mathcal{O}(2^t \log N)$ time in total. ◀

▶ Remark 13. Since our query procedure computes $\mathsf{last}(u, v)$, and we have MSSP data structures that capture the u-to-$\mathsf{last}(u, v)$ and the $\mathsf{last}(u, v)$-to-v shortest paths, an optimal alignment can be returned in time proportional to the total length of the two substrings.

By setting the r_i's appropriately, we obtain the following tradeoffs, which are identical to those of Pettie and Long for arbitrary planar graphs [30].

▶ **Proposition 14.** *For two strings of lengths m and n, with $N = mn$, there is an alignment oracle achieving either of the following tradeoffs:*
- $N \log^{2+o(1)} N$ *space and* $N^{o(1)}$ *query time,*
- $N^{1+o(1)}$ *space and* $\log^{2+o(1)} N$ *query time.*

Proof. The space of the oracle is $\mathcal{O}\left(N \log^2 N + N \log N \cdot \sum_{i=0}^{t-1} r_{i+1}/r_i\right)$ by Lemma 10. We will choose r_i's for $i \geq 1$ to be a geometric progression with common ratio p to be specified below. In that case, $t = \mathcal{O}(\log_p N)$ and the space becomes $\mathcal{O}(N \log^2 N + N \log N \cdot p \log_p N)$. First, let us set $p = N^{1/g(N)}$ for some $g(N)$ which is $\omega(\log N / \log\log N)$ and $o(\log N)$. Then, $p = 2^{\log N/g(N)} = 2^{o(\log\log N)} = \log^{o(1)} N$. We get $\mathcal{O}(N \log N \cdot 2^{\log N/g(N)} \log N) = N \log^{2+o(1)} N$ space and $N^{o(1)}$ query time. Second, let us set $p = N^{1/f(N)}$, for some $f(N)$ which is $\omega(1)$ and $o(\log\log N)$. We get $N^{1+o(1)}$ space and $\log^{2+o(1)} N$ query time. ◀

The following observation will prove useful in the efficient construction algorithm of the oracle that will be presented in the next section.

▶ **Observation 15.** *The query algorithm for $\mathsf{dist}(u, v)$ takes $\mathcal{O}(2^{t-\mathsf{lev}(u)} \log^2 N)$ time and uses only Voronoi diagrams $\mathsf{VD}(u, Q)$ for $Q \in \mathcal{R}_i$ with $i > \mathsf{lev}(u)$.*

4 An Efficient Construction Algorithm

In this section, we present an algorithm for constructing the alignment oracle in $N^{1+o(1)}$ time (thus completing the proof of Theorem 1). The computation of the recursive decomposition, the recursive $(r_t, \ldots, r_0)$-division and all of the MSSP structures stored for all pieces in $\mathcal{A}$ can be done in $\mathcal{O}(N \log^2 N)$ time. It therefore only remains to analyze the time it takes to construct all the representations of Voronoi diagrams stored by the oracle.

Consider some additively weighted Voronoi diagram for Q^{out} with sites a subsequence U of $\lrcorner(Q)$—we will only build Voronoi diagrams with $U = \lrcorner(Q)$, but during the analysis, we will also consider Voronoi diagrams with sites $U \subseteq \lrcorner(Q)$. In what follows, when we talk about a piece $H \neq Q$, we will really mean its intersection with Q^{out}, assuming that it is non-empty. Similarly, when we talk about ∂H, $\ulcorner(H)$, and $\lrcorner(H)$ we will really mean the intersection of ∂H, $\ulcorner(H)$, and $\lrcorner(H)$ with Q^{out}, respectively. See Figure 5 for an illustration.

▶ **Lemma 16.** *Let $X \in \{\ulcorner(H), \lrcorner(H)\}$. Let $u, v \in X$ belong to distinct Voronoi cells. If u precedes v (in X) then the site $s_u \in U$ of u precedes (in U) the site $s_v \in U$ of v.*

Proof. For any vertex $a \in X$, that belongs to the cell of a site s_a, all vertices in the shortest s_a-to-a path belong to $\mathsf{Vor}(s_a)$. Towards a contradiction, suppose that s_u succeeds s_v in U. By the planarity of the graph and the fact that paths can go only down and right it follows that the shortest s_u-to-u path must cross the shortest s_v-to-v path in some vertex b. Then $b \in \mathsf{Vor}(s_u) \cap \mathsf{Vor}(s_v)$, which is a contradiction as Voronoi cells are disjoint. ◀

The above lemma means that the vertices of each of $\ulcorner(H)$ and $\lrcorner(H)$ can be partitioned into maximal contiguous intervals of vertices belonging to the Voronoi cell of the same site in U. When we say that we compute the partition of $\ulcorner(H)$ or $\lrcorner(H)$ with respect to U, we mean that we compute these intervals, specified by their endpoints, and, for each interval, the site from U to which the vertices of the interval belong.

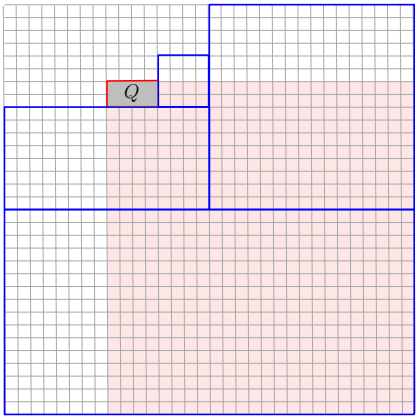

Figure 5 Covering Q^{out} (shaded pink) with siblings of ancestors of Q in $\mathcal{A}$ (blue boxes). When we refer to a piece H (e.g. a blue box), we only refer to the portion of H that belongs to Q^{out}.

The following simple observation allows us to compute partitions using binary search. It says that a piece H contains $s_{\searrow}$ if and only if s is the site of some vertex in $\sqcap(H)$, and s is not the site of any vertex in $\sqcup(H)$.

▶ **Lemma 17.** *For any $s \in S$ and any level ℓ, there is a unique level-ℓ piece $H \in \mathcal{A}$ for which $\mathsf{Vor}(s) \bigcap \sqcap(H) \neq \emptyset$ and $\mathsf{Vor}(s) \bigcap \sqcup(H) = \emptyset$, and this piece contains $s_{\searrow}$.*

Proof. By Corollary 9, for all $v \in \mathsf{Vor}(s)$, there exists a v-to-$s_{\searrow}$ path all of whose vertices are in $\mathsf{Vor}(s)$. Hence, for every level-ℓ piece H for which $\mathsf{Vor}(s) \bigcap \sqcap(H) \neq \emptyset$ and $s_{\searrow} \notin H$, we must also have that $\mathsf{Vor}(s) \bigcap \sqcup(H) \neq \emptyset$. We can thus focus on the at most four level-ℓ pieces that contain $s_{\searrow}$. It is readily verified that the bottom-right of those pieces is the only one for which $\mathsf{Vor}(s) \bigcap \sqcup(H) = \emptyset$. ◀

▶ Remark 18. The condition in the statement of Lemma 17 is equivalent to: $s_{\searrow} \in H \setminus \sqcup(H)$.

Lemma 17 provides a criterion on the partitions of $\sqcap(H)$ and $\sqcup(H)$ for determining whether a piece H contains any vertex $s_{\searrow}$. The following lemma describes a binary search procedure, Partition, which gets as input a sequence U of candidate sites, and returns the partition of $\sqcap(H)$ or $\sqcup(H)$; i.e., the subsequence of sites in U whose Voronoi cells contain the vertices of $\sqcap(H)$ or $\sqcup(H)$. The procedure Partition will be a key element in the overall construction algorithm. The following lemma describes an implementation of Partition using distance queries $\mathsf{dist}(u, v)$ with $u \in \sqcup(Q)$ and $v \in \partial H$. We will ensure that such queries can be answered efficiently whenever Partition is called by the main algorithm.

▶ **Lemma 19.** *Given a sequence $U \subseteq \sqcup(Q)$ of sites and their additive weights, we can perform the procedure Partition (that computes a partition of $\sqcap(H)$ or $\sqcup(H)$ w.r.t. U) in the time required by $\mathcal{O}(|U| \cdot \log n)$ distance queries $\mathsf{dist}(u, v)$ with $u \in \sqcup(Q)$ and $v \in \partial H$.*

Proof. We will only prove the statement for $\sqcap(H)$ as the case of $\sqcup(H)$ is analogous. We start with a single interval, which is all of $\sqcap(H)$. We will call an interval *active* if we have not concluded that all of its vertices belong to the same Voronoi cell. For each active interval L, we have a set C_L of *candidate sites*. Thus, initially, the single interval $\sqcap(H)$ is active, and U is the set of its candidate sites.

The algorithm proceeds by divide and conquer. As long as we have an active interval L, we perform the following: we compute the site $u \in C_L$ with the minimum additively weighted

distance to the midpoint of L. This is done in the time required by C_L distance queries of the form specified in the statement of the lemma. Then, we split L at this midpoint: for the left part of L the set of candidate sites is now $\{v \in C_L : v \leq u\}$, while for the right part of L the set of candidate sites is now $\{v \in C_L : v \geq u\}$. If either of these two sets is of size 1, the corresponding interval becomes inactivate. That is, we recurse on at most two active intervals of roughly half the length. In the end, in a left-to-right pass, we merge consecutive intervals all of whose vertices belong to the same Voronoi cell.

Let us now analyze the time complexity of the above algorithm. First, observe that the sequences of candidates of any two intervals at the same level of the recursion are internally disjoint. Thus, each site is a candidate for at most two active intervals at the same recursive level. Second, at level j of the recursion, the length of every active interval is $\mathcal{O}(|\sqsubset(H)|/2^j)$. Hence, the total time required to process all intervals is proportional to the time required by $\mathcal{O}(|U| \cdot \log n)$ distance queries $\mathsf{dist}(u, v)$, with $u \in \sqcup(Q)$ and $v \in \partial H$. ◀

We now present the algorithm for computing the representations of the Voronoi diagrams stored by the oracle. The algorithm performs the computation in order of decreasing levels of the recursive $(r_t, \ldots, r_0)$-division.

Consider some level i, and assume that we have already computed all Voronoi diagrams $\mathsf{VD}(u, R)$ for pieces $R \in \bigcup_{j>i} \mathcal{R}_j$. Consider any piece $P \in \mathcal{R}_{i-1}$. Let $Q \in \mathcal{R}_i$ be the parent of P in $\mathcal{T}$. Our goal is to compute, for every $u \in \sqcup(P)$, the representation of $\mathsf{VD}(u, Q)$, the Voronoi diagram of Q^{out} with sites $\sqcup(Q)$ and additive weights $\mathsf{dist}(u, \sqcup(Q))$. Recall that this representation consists of the vertices $s_{\searrow}$ and $\mathsf{last}(s, s_{\searrow})$ for every site $s \in \sqcup(Q)$. We would like to compute this representation in time roughly proportional to its size $|\sqcup(Q)|$. By Observation 15, using the already computed parts of the oracle for levels $j > i$, we can already answer any distance query $\mathsf{dist}(s, v)$ for any $s \in \sqcup(Q)$ and any $v \in Q^{out}$ in $\mathcal{O}(2^t \log^2 n)$ time. These are precisely the distance queries required for computing partitions of pieces H in Q^{out} w.r.t. sites in $\sqcup(Q)$ (Lemma 19).

The computation is done separately for each $u \in \sqcup(P)$. First, we compute the additive weights $\mathsf{dist}(u, \sqcup(Q))$ in $\mathcal{O}(|\sqcup(Q)| \cdot \log n)$ time using the MSSP data structure stored for $Q \setminus (P \setminus \partial P)$ with sites $\sqcup(P)$. Next, we cover Q^{out} using $\mathcal{O}(\log N)$ pieces from $\mathcal{A}$ that are internally disjoint from Q (i.e. they may only share boundary vertices). These pieces are the $\mathcal{O}(\log N)$ siblings of the (weak) ancestors of Q in $\mathcal{A}$ that have a non-empty intersection with Q^{out} (see Figure 5). Notice that these pieces are in $\mathcal{A}$ but not necessarily in $\mathcal{T}$.

We shall find the vertices $s_{\searrow}$ of $\mathsf{VD}(u, Q)$ in each such piece H separately. We invoke Partition on $\sqsubset(H)$ and on $\sqcup(H)$, and use Lemma 17 to determine whether H contains any vertices $s_{\searrow}$. If so, we zoom in on each of the two child pieces of H in $\mathcal{A}$ until, after $\mathcal{O}(\log N)$ steps, we get to a constant-size piece, in which we can find $s_{\searrow}$ by brute force. Note, however, that we are aiming for a running time that is roughly proportional to $|\sqcup(Q)|$, but that the running time of Partition depends on the number of sites U w.r.t. which we partition. This is problematic since, e.g., when H contains $s_{\searrow}$ just for a single site s, we can only afford to invest $\tilde{\mathcal{O}}(1)$ time in locating $s_{\searrow}$ in H. In this case, computing the partition w.r.t. $|\sqcup(Q)|$ is too expensive. Even computing the partition just w.r.t. the sites whose Voronoi cell has non-empty intersection with H, which is bounded by $|\sqsubset(H)|$, is too expensive. To overcome this problem we will show that it suffices to compute the partition w.r.t. a smaller sequence of sites, whose size is proportional to the number of sites s with $s_{\searrow}$ in H (actually in $H \setminus \sqcup(H)$), rather than to the size of H or of $\sqsubset(H)$. We call such a sequence a *safe* sequence of sites for H, which we now define formally. Recall that the Voronoi diagram $\mathsf{VD}(u, Q)$ of Q^{out} has sites $\sqcup(Q)$. Let U be a subsequence of $\sqcup(Q)$. Consider the Voronoi diagram VD' of Q^{out} whose sites are the elements of U (with the same additive distances as in $\mathsf{VD}(u, Q)$). We

say that U is *safe for* H if and only if the sets $\{(s, s_{\searrow}) : s \text{ is a site and } s_{\searrow} \in H \setminus \lrcorner(H)\}$ are identical for VD' and $\mathsf{VD}(u, Q)$.

▶ **Observation 20.** *A sequence that is safe for H is also safe for any child H' of H in $\mathcal{A}$.*

We will discuss the details of safe sequences after first providing the pseudocode of the procedure Zoom for finding the vertices $s_{\searrow}$ in a piece H.

Algorithm 3 Zoom(U, ω, H)

Input: The additive weight $\omega(s)$ for each $s \in \lrcorner(Q)$, a piece H in $\mathcal{A}$ that is internally disjoint from Q and has a non-empty intersection with Q^{out}, and a sequence $U \subseteq \lrcorner(Q)$ that is safe for H.
Output: All vertices $s_{\searrow}$ for $s \in \lrcorner(Q)$ that belong to H.

1: **if** $|H| = 4$ **then**
2: Find all vertices $s_{\searrow}$ in H for all $s \in U$ by computing $\omega(s) + \mathsf{dist}(s, v)$ for all $s \in U$ and all v adjacent to some vertex of H.
3: **for** each child H' of H in $\mathcal{A}$ **do**
4: $Z \leftarrow$ Partition$(U, \omega, \ulcorner(H'))$
5: $Z' \leftarrow$ Partition$(U, \omega, \lrcorner(H'))$
6: $L \leftarrow Z \setminus Z'$
7: **if** $L \neq \emptyset$ **then**
8: $V \leftarrow Z \setminus \{z \in Z \setminus L :$ both the predecessor and successor of z in Z are not in $L\}$
9: Zoom(V, ω, H')

The procedure Zoom takes as input a piece H and a sequence $U \subseteq \lrcorner(Q)$ that is safe for H. In order to compute $\mathsf{VD}(u, Q)$, we call the procedure Zoom$(\lrcorner(Q), \omega, H)$ for each of the $\mathcal{O}(\log N)$ pieces H that we use to cover Q^{out}. Clearly, in each of those initial $\mathcal{O}(\log N)$ calls $\lrcorner(Q)$ is a safe sequence of the respective piece. For each child H' of H in $\mathcal{A}$, we check whether the condition of Lemma 17 is satisfied in lines 4-7. Note that since U is a safe sequence for H, U is also a safe sequence for H' and hence our computation of the set $L = \{s \in \lrcorner(Q) : s_{\searrow} \in H' \setminus \lrcorner(H')\}$ is correct. If L is non-empty, we recurse on H' (cf. line 7). In line 8, we construct a safe sequence V for H', of size proportional to $|L|$ and then, in line 9, we call Zoom(V, ω, H'). In order to prove the correctness of procedure Zoom, it remains to show that V is indeed safe. The following two lemmas show this (see also Figure 6).

▶ **Lemma 21.** *Let U be safe for a piece H, such that* Partition$(U, \omega, \ulcorner(H)) = U$. *Suppose that there are three elements u_1, u_2, u_3 of U that appear consecutively (in this order) in both* Partition$(U, \omega, \ulcorner(H))$ *and* Partition$(U, \omega, \lrcorner(H))$. *Then, $U \setminus \{u_2\}$ is also safe for H.*

Proof. To avoid confusion we denote the Voronoi diagram of Q^{out} with sites U by VD and the one with sites $U \setminus \{u_2\}$ by VD'. We denote the Voronoi cells of VD by $\mathsf{Vor}(\cdot)$, and those of VD' by $\mathsf{Vor}'(\cdot)$. Note that for every $u \in U \setminus \{u_2\}$, $\mathsf{Vor}(u) \subseteq \mathsf{Vor}'(u)$. By Lemma 17, in VD, ${u_1}_{\searrow}, {u_2}_{\searrow}, {u_3}_{\searrow} \notin H \setminus \lrcorner(H)$. Hence, in VD', ${u_1}_{\searrow}, {u_3}_{\searrow} \notin H \setminus \lrcorner(H)$.

▷ Claim. Every vertex y of $\mathsf{Vor}(u_2) \cap H$ belongs either to $\mathsf{Vor}'(u_1)$ or to $\mathsf{Vor}'(u_3)$.

Proof. Consider the last vertex z_1 of $\lrcorner(H)$ that is in $\mathsf{Vor}(u_1)$, and the first vertex z_3 of $\lrcorner(H)$ that is in $\mathsf{Vor}(u_3)$. Let ρ_1 be a shortest u_1-to-z_1 path, and ρ_3 be a shortest u_3-to-z_3 path. Note that all vertices of ρ_1 belong to $\mathsf{Vor}(u_1)$ and all vertices of ρ_3 belong to $\mathsf{Vor}(u_3)$. Consider any vertex y of $\mathsf{Vor}(u_2) \cap H$. The vertex y lies to the right of ρ_1 and to the left of ρ_3. In VD', the vertices of ρ_1 belong to $\mathsf{Vor}'(u_1)$ and the vertices of ρ_3 belong to $\mathsf{Vor}'(u_3)$.

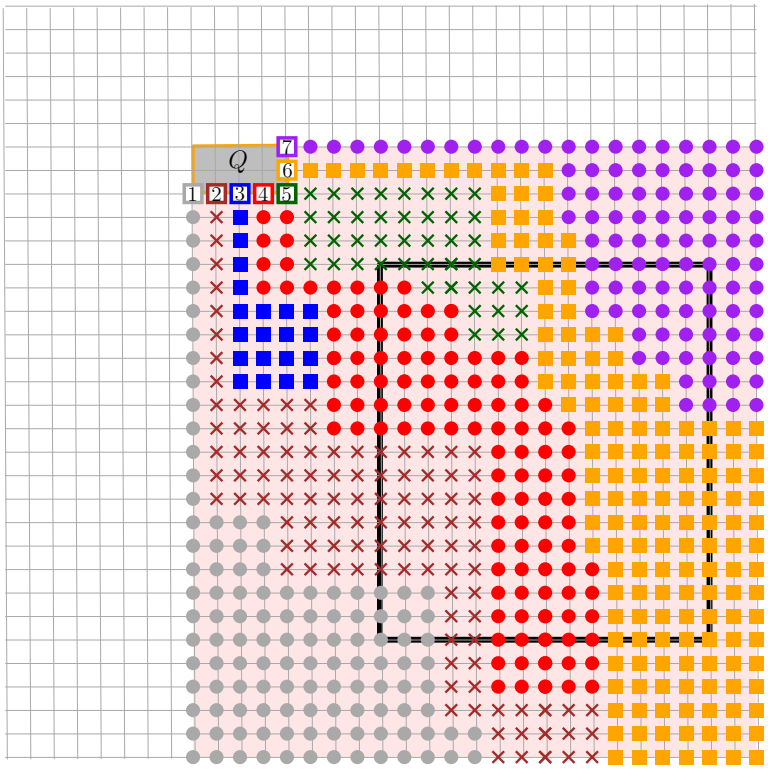

Figure 6 Illustration for Lemma 21. Part of Q^{out} (pink) for some piece Q (gray) is shown. A piece H is indicated by a black rectangle. The sites of $\lrcorner(Q)$ are numbered 1 through 7. The partition of $\ulcorner(H)$ w.r.t. $\lrcorner(Q)$ is $U = (1, 2, 4, 5, 6, 7)$. Hence, U is safe for H. The partition of $\lrcorner(H)$ w.r.t. $\lrcorner(Q)$ is $(1, 2, 4, 6, 7)$. Since sites $1, 2, 4$ are consecutive in both partitions, $(1, 4, 5, 6, 7)$ is also safe for H. Further, $\lrcorner(Q)$ is also clearly safe for H. However, $(1, 3, 4, 5, 6, 7)$ may not be safe for H, as $3_{\searrow}$ could be in $H \setminus \partial H$ in the Voronoi diagram with sites $1, 3, 4, 5, 6, 7$ and the same additive weights.

Hence, by Lemma 16, in VD', y can only belong to a site $s \neq u_2$ that is weakly between u_1 and u_3. Since u_1, u_2, u_3 appear consecutively in $\textsc{Partition}(U, \omega, \ulcorner(H)) = U$, the only such sites are u_1 and u_3, and the claim follows. ◁

By the above claim, the sets $\{(s, s_{\searrow}) : s \text{ is a site and } s_{\searrow} \in H \setminus \lrcorner(H)\}$ are identical for VD and VD'. Since U is safe for H, so is $U \setminus \{u_2\}$. ◀

▶ **Lemma 22.** *Suppose that U is safe for H. Then, in each recursive call $\textsc{Zoom}(V, \omega, H')$ made by procedure $\textsc{Zoom}(U, \omega, H)$ for a child H' of H in $\mathcal{A}$, V is a safe sequence for H'.*

Proof. Since U is a safe sequence for H, U is also a safe sequence for H' (cf. Observation 20). Hence, L is the set of sites s such that $s_{\searrow} \in H' \setminus \lrcorner(H')$.

Z is the set of sites in U whose Voronoi cells have non-empty intersection with H'. In line 8, we remove from Z all vertices of $Z \setminus L$ that are not preceded or succeeded by a vertex in L. Therefore, by considering the removal of these sites one at a time, and directly applying Lemma 21 to each such removal, the resulting sequence V is safe for H'. ◀

This establishes the correctness of our construction algorithm. Let us now analyze its time complexity. Initially, we make $\mathcal{O}(\log N)$ calls to $\textsc{Zoom}(U, \omega, H)$, each with $U = \lrcorner(Q)$. In each recursive call, for a child H' of a piece H, the set U of sites is of size proportional to

the size of the set $\{s \in \lrcorner(Q) : s_{\searrow} \in H' \setminus \lrcorner(H')\}$. Note that in each level of the tree $\mathcal{A}$ each $s \in \lrcorner(Q)$ is an element of exactly one such set. Hence, each $s \in \lrcorner(Q)$ contributes to $\mathcal{O}(\log N)$ calls to Zoom: the $\mathcal{O}(\log N)$ initial ones and at most one more per level of $\mathcal{A}$ (which is of depth $\mathcal{O}(\log N)$).

Thus, by Lemma 19, computing $s_{\searrow}$ for all $s \in \lrcorner(Q)$ reduces to $\mathcal{O}(|\lrcorner(Q)| \cdot \log^2 N)$ distance queries $\mathsf{dist}(u, v)$, with $u \in \lrcorner(Q)$ and $v \in Q^{out}$. We can answer each such query with the portion of the oracle that has already been computed in $\mathcal{O}(2^t \log^2 N)$ time. Now, recall that a $\mathsf{dist}(s, s_{\searrow})$ query also computes $\mathsf{last}(s, s_{\searrow})$, and hence these values can also be retrieved in $\mathcal{O}(2^t \log^2 N)$ time. Thus, $\mathsf{VD}(u, Q)$, which is of size $\mathcal{O}(|\lrcorner(Q)|)$ can be computed in time $\mathcal{O}(|\lrcorner(Q)| \cdot 2^t \log^4 N)$, which is $|\lrcorner(Q)| \cdot N^{o(1)}$ for both choices of t in Proposition 14. Therefore, the time to compute $\mathsf{VD}(u, Q)$ for all pieces is $N^{o(1)} \cdot \sum_Q |\lrcorner(Q)| = N^{o(1)} \cdot \sum_{i=0}^{t-1} \frac{N}{r_i}\sqrt{r_i}$, which is $N^{1+o(1)}$ for both choices of t. This concludes the proof of Theorem 1.

5 Tradeoffs with $o(N)$ space

In this section we prove Theorem 4. Recall that for this result we consider integer alignment weights upper-bounded by w: A weight w_{match} for aligning a pair of matching letters, w_{mis} for aligning a pair of mismatching letters, and w_{del} for letters that are not aligned. One may assume without loss of generality that $2w_{match} > 2w_{mis} \geq w_{del}$ [36]. Given w_{match}, w_{mis} and w_{del}, we define $w'_{match} = 0$, $w'_{mis} = w_{match} - w_{mis}$ and $w'_{del} = \frac{1}{2}w_{match} - w_{del}$. These weights are also upper-bounded by w. Then, a shortest path (of length W) in the alignment grid with respect to the new weights, corresponds to a highest scoring path with respect to the original weights (of score $\frac{1}{2}(m+n)w_{match} - W$).

FR-Dijkstra. We define the *dense distance graph* (DDG) of a piece P as a directed bipartite graph with vertices ∂P and an edge from every vertex $u \in \ulcorner(P)$ to every vertex $v \in \lrcorner(P)$ with weight equal to the length of the shortest u-to-v path in P. We denote this graph as DDG_P.[2] DDG_P can be computed in time $\mathcal{O}((|\partial P|^2 + |P|) \log |P|) = \mathcal{O}(|P| \log |P|)$ using the MSSP data structure. In their seminal paper, Fakcharoenphol and Rao [20] designed an efficient implementation of Dijkstra's algorithm on any union of DDGs—this algorithm is nicknamed FR-Dijkstra. FR-Dijkstra exploits the fact that, due to planarity, the adjacency matrix of each DDG can be decomposed into Monge matrices (defined formally in equation (1) below). In our case, since each DDG is a bipartite graph, the entire adjacency matrix is itself Monge (this will be shown below). Let us now give an interface for FR-Dijkstra that is convenient for our purposes.

▶ **Theorem 23** ([20, 23, 31]). *Dijkstra's algorithm can be run on the union of a set of DDGs with $\mathcal{O}(M)$ vertices in total (with multiplicities) and an arbitrary set of $\mathcal{O}(M)$ extra edges in the time required by $\mathcal{O}(M \log^2 M)$ accesses to edges of this union.*

▶ Remark 24. In our case, the runtime of the algorithm encapsulated in the above theorem can be improved to $\mathcal{O}(M \log\log(nw))$. One of the two $\mathcal{O}(\log M)$ factors stems from the decomposition of the adjacency matrix into Monge submatrices, which is not necessary in our case. The second $\mathcal{O}(\log M)$ comes from the use of binary heaps. In our case, these heaps store integers in $\mathcal{O}(nw)$ and can be thus implemented with $\mathcal{O}(\log\log(nw))$ update and query times using an efficient predecessor structure [38, 39].

[2] For general planar graphs, the DDG of a piece is usually defined as a complete directed graph on ∂P.

A warmup. Let us first show how to construct in $\tilde{\mathcal{O}}(N)$ time an $\tilde{\mathcal{O}}(N)$-size oracle that answers queries in $\tilde{\mathcal{O}}(\sqrt{N})$ time using well-known ideas [20]. We will then improve the size of the data structure by efficiently storing the computed DDGs.

Let us consider an r-division of G, for an r to be specified later. Further, consider the tree $\mathcal{A}'$, obtained from the recursive decomposition tree $\mathcal{A}$ by deleting all descendants of pieces in the r-division. For each piece $P \in \mathcal{A}'$, we compute and store DDG_P. In each of the $\mathcal{O}(\log N)$ levels of $\mathcal{A}$, for some value y, we have $\mathcal{O}(N/y)$ pieces, each with $\mathcal{O}(y)$ vertices and $\mathcal{O}(\sqrt{y})$ boundary vertices. Hence, both the construction time and the space occupied by these DDGs are $\tilde{\mathcal{O}}(N)$.

We next show how to compute the weight of an optimal alignment of $S[i\,..\,j]$ and $T[a\,..\,b]$, i.e. compute the shortest path ρ from $u = (i-1, a-1)$ to $v = (j, b)$, where $i < j$ and $a < b$. If u and v belong to $P \setminus \partial P$ for a piece P of the r-division, then both $S[i\,..\,j]$ and $T[a\,..\,b]$ are of length $\mathcal{O}(\sqrt{r})$, and we can hence run the textbook dynamic programming algorithm which requires $\mathcal{O}(r)$ time. Henceforth, we consider the complementary case.

Let P_u and P_v be the distinct r-division pieces that contain u and v, respectively. Further, let Q be the lowest common ancestor of P_u and P_v in $\mathcal{A}'$. For $z \in \{u, v\}$, let Q_z be the child of Q that contains z. The set of vertices $Q_u \cap Q_v$ are denoted by $\mathsf{sep}(Q)$—which stands for separator. Observe that ρ must contain at least one vertex from $\mathsf{sep}(Q)$. Consider the set that consists of P_z and the siblings of weak ancestors of P_z in $\mathcal{A}'$ that are descendants of Q, and call it the *cone of* P_z. The cone of P_z covers Q_z and its elements are pairwise internally disjoint. See Figure 7 for an illustration. Now, observe, that any shortest path ρ between a vertex of ∂P_z and a vertex of $\mathsf{sep}(Q)$ can be partitioned into subpaths $\rho_1, \ldots, \rho_k$ such that each ρ_i lies entirely within some piece R_i in the cone of P_z and both ρ_i's endpoints are boundary vertices of R_i. Using these two observations, we can compute a shortest u-to-v path by running FR-Dijkstra on the cones of P_u and P_v, and, possibly, the following extra edges. In the case where the source u (resp. target v) is not a boundary vertex, we include $\mathcal{O}(\sqrt{r})$ additional edges: for each boundary vertex x of P_u (resp., P_v), an edge from u to x (resp., from x to v) with length equal to that of the shortest path from u to x (resp. from x to v). The weights of such edges can be computed in $\mathcal{O}(r)$ time using dynamic programming. Thus, a query can be answered in time $\tilde{\mathcal{O}}(\sqrt{N} + r)$. By setting $r = \sqrt{N}$ we get the promised complexities.

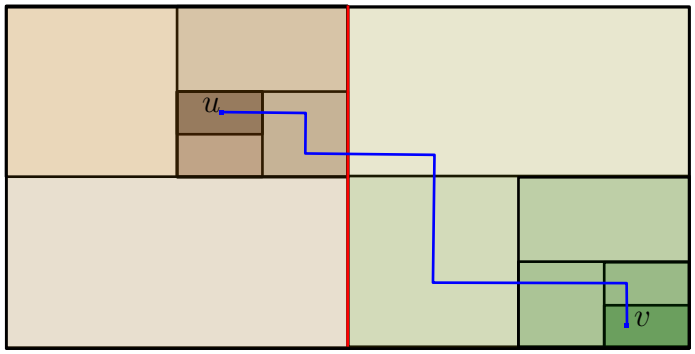

Figure 7 The piece Q is shown. $\mathsf{sep}(Q)$ is denoted by red, while a shortest u-to-v path is shown in blue. The pieces in the cone of P_u are shaded by brown, while the pieces in the cone of P_z are shaded by pink.

The tradeoff. We can now describe the entire tradeoff of Theorem 4. We assume that $r > w^2$, since otherwise $Nw/\sqrt{r} = \Omega(N)$ and Theorem 4 is satisfied by the warmup solution. For a piece P, we will show how to store DDG_P in $\mathcal{O}(|\partial P| \cdot w) = \mathcal{O}(w\sqrt{|P|})$ instead of $\mathcal{O}(|P|)$ space. Our representation will allow retrieving the length of any edge of DDG_P in $\tilde{\mathcal{O}}(1)$ time. Our approach closely follows ideas from [2].

For the remainder, we deviate from our ordering convention of $\ulcorner(P)$; the first vertex is now the top-right vertex of P, and the last is the bottom-left one. $\lrcorner(P)$ is ordered as before where the first vertex is the bottom-left one and the last is the top-right one. We denote the i-th vertex of $\ulcorner(P)$ by v_i and the j-th vertex of $\lrcorner(P)$ by u_j. Note that we can infer whether any vertex u_j is reachable from a vertex v_i in $\mathcal{O}(1)$ time. For ease of presentation we would like the weights of all edges of DDG_P to be finite. To achieve this, for each edge between two vertices of $\ulcorner(P)$, we introduce (in P) an artificial edge with weight w in the opposite direction. It is readily verified that all v_i-to-u_j distances that were finite before the introduction of such edges remain unchanged. This is because the shortest path between two vertices of this modified graph that lie on the same column (resp. row) consists solely of vertical (resp. horizontal) edges.

Let M be the adjacency matrix of DDG_P with entry $M[i,j]$ storing the distance from v_i to u_j, and let $k = |\ulcorner(P)| = |\lrcorner(P)|$. Matrix M satisfies the Monge property, namely:

$$M[i+1,j] - M[i,j] \leq M[i+1,j+1] - M[i,j+1] \tag{1}$$

for any $i \in [1, k-1]$ and $j \in [1, k-1]$. This is because the shortest v_i-to-u_j and v_{i+1}-to-u_{j+1} paths must necessarily cross.

In addition, for any fixed $j \in [1,k]$, for all $i \in [1, k-1]$, we have

$$|M[i+1,j] - M[i,j]| \leq w. \tag{2}$$

This is because edges v_iv_{i+1} and $v_{i+1}v_i$ both have weight at most w. This implies that $M[i,j] \leq M[i+1,j] + w$, as a shortest v_i-to-u_j path cannot be longer than the concatenation of the edge v_iv_{i+1} with a shortest v_{i+1}-to-u_j path. Similarly, we have $M[i+1,j] \leq M[i,j] + w$.

Our representation of M is as follows, and fairly standard [2, 16, 36]. We define a $(k-1) \times (k-1)$ matrix P, satisfying

$$P[i,j] = M[i,j] + M[i+1,j+1] - M[i,j+1] - M[i+1,j].$$

Equations (1) and (2) imply that, for any $i \in [1, k-1]$, the sequence $M[i+1,j] - M[i,j]$ is nondecreasing and contains only values in $[-w, w]$. Hence, P has $\mathcal{O}(kw)$ non-zero entries. Now, observe that

$$\sum_{r \geq i, c \geq j} P[r,c] = M[i,j] + M[k,k] - M[i,k] - M[k,j]. \tag{3}$$

We store the last row and column of M. By (3), this means that retrieving $M[i,j]$ boils down to computing $\sum_{r \geq i, c \geq j} P[r,c]$. We view the non-zero entries of P as points in the plane and build in $\tilde{\mathcal{O}}(kw)$ time an $\tilde{\mathcal{O}}(kw)$-size 2D-range tree over them [7], which can return $\sum_{r \geq i, c \geq j} P[r,c]$ for any i, j in $\tilde{\mathcal{O}}(1)$ time. The overall space required by our representation of DDG_P is thus $\tilde{\mathcal{O}}(kw) = \tilde{\mathcal{O}}(|\partial P| \cdot w)$, and any entry of M can be retrieved in $\tilde{\mathcal{O}}(1)$ time.

In total, over all $\mathcal{O}(N/r)$ pieces of the r-division, the space required is $\tilde{\mathcal{O}}((N/r) \cdot \sqrt{r} \cdot w) = \tilde{\mathcal{O}}(Nw/\sqrt{r})$. This level dominates the other levels of the decomposition, as the sizes of pieces, as well as their boundaries, decrease geometrically in each root-to-leaf path. Note that, for the dynamic programming part of the query algorithm, we can simply store the strings, which take $\mathcal{O}(m+n)$ space. This concludes the proof of Theorem 4.

References

1. Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS*, pages 59–78, 2015.
2. Amir Abboud, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Near-optimal compression for the planar graph metric. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 530–549. SIAM, 2018.
3. Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020.
4. Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591, 2015.
5. Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theoretical Computer Science*, 638:112–121, 2016.
6. Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018.
7. Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244 – 251, 1979.
8. Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97, 2015.
9. Gerth Stølting Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.
10. Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. *ACM Transactions on Algorithms*, 15(2):21:1–21:38, 2019.
11. Sergio Cabello, Erin W. Chambers, and Jeff Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.
12. Panagiotis Charalampopoulos. *Data Structures for Strings in the Internal and Dynamic Settings*. PhD thesis, King's College London, 2020.
13. Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 138–151, 2019.
14. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, pages 8:1–8:15, 2020.
15. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, pages 22:1–22:17, 2019.
16. Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic string alignment. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, pages 9:1–9:13, 2020.
17. Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *61st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989, 2020.
18. Hagai Cohen and Ely Porat. On the hardness of distance oracle for sparse graph. *CoRR*, 2010. arXiv:1006.1117.
19. Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017*, pages 962–973, 2017.

20 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.

21 Paweł Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 515–529, 2018.

22 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *15th International Symposium Algorithms and Data Structures, WADS 2017*, pages 421–436, 2017.

23 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and partial Monge matrices, and their applications. *ACM Transactions on Algorithms*, 13(2):26:1–26:42, 2017.

24 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014.

25 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 146–155, 2005.

26 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, pages 28:1–28:12, 2016.

27 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient data structures for the factor periodicity problem. In *19th International Symposium on String Processing and Information Retrieval, SPIRE 2012*, pages 284–294, 2012.

28 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551, 2015.

29 Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

30 Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *32nd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 2517–2536, 2021.

31 Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *18th Annual European Symposium on Algorithms, ESA 2010*, pages 206–217, 2010.

32 Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the Thorup-Zwick bound. *SIAM Journal on Computing*, 43(1):300–311, 2014.

33 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, pages 290–303, 2017.

34 Yoshifumi Sakai. A substring-substring LCS data structure. *Theoretical Computer Science*, 753:16–34, 2019.

35 Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

36 Alexander Tiskin. Semi-local string comparison: algorithmic techniques and applications, 2007. arXiv:0707.3619.

37 Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008.

38 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

39 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.