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 \(O(N)\) 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

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 \(O(N)\) 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 \cite{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 \cite{34,36}, pattern matching \cite{24,28}, approximate pattern matching \cite{17}, dictionary matching \cite{14,15}, compression \cite{24}, periodicity \cite{27,28}, counting palindromes \cite{33}, longest common substring \cite{3}, computing minimal and maximal suffixes \cite{5,26}, and computing the lexicographically $k$-th suffix \cite{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 \cite{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 $O(n + m)$-size oracle, that can be constructed in $O(N)$ time, and answers queries in $O(\log N / \log \log N)$ time \cite{36}. For the general problem, Sakai \cite{34} (building on Tiskin’s work \cite{36}) showed how to construct in $O(N)$ time an alignment oracle with $O(n + m)$ 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 $N^{1+\omega(1)}$ time an alignment oracle achieving either of the following tradeoffs:
> 1. $N^{1+\omega(1)}$ space and $\log^{2+\omega(1)} N$ query time,
> 2. $N \log^{2+\omega(1)} N$ space and $N^{\omega(1)}$ 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 \cite{30} (which builds upon \cite{13,19,21}) is an alignment oracle with space $N^{1+\omega(1)}$ and query time $O(\log^{2+\omega(1)} N)$. However, the construction time of this oracle is $\Omega(N^{3/2})$. Our main contribution is an improved $N^{1+\omega(1)}$ 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 \cite{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 \cite{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 $s$. 

---

Voronoi diagrams

... Voronoi diagrams is a partition of the vertices into ...

... state-of-the-art distance oracle for planar graphs. A distance oracle is a compact ...

... The starting point of our work is the Planar distance oracles and Voronoi diagrams. ...

... Theorem 1. For two strings of lengths $m$ and $n$, with $N = mn$, we can construct ...

... Our oracle has the same recursive structure as the planar graph oracles in ...

---

... 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.\footnote{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.}

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 $O(N^{1-\epsilon})$, 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.

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

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].

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

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.

\begin{theorem}[[3,12]]
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 $\tilde{O}(N)$ time an $\tilde{O}(Nw/\sqrt{r} + m + n)$-space alignment oracle that answers queries in time $\tilde{O}(\sqrt{N} + r)$.

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

**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 $O(n + m)$ time preprocessing of the input strings (oblivious to the threshold $k$),
given a substring of $S$, a substring of $T$, and a threshold $k$, in $O(k^2)$ time one can decide whether the edit distance of these substrings is at most $k$, and if so, return it.

Further, after an $O(n + m)$-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 $O(k^4 \cdot |Y|/|X|)$ 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 $\tilde{O}(n)$ time per operation so that an alignment of any pair of substrings can be queried in $\tilde{O}(n)$ 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 $\text{LCS}(S,T)$.

▶ **Example 5.** An LCS of $S = \text{abcddaee}$ and $T = \text{abbccdec}$ is $\text{abcde}$; $\text{LCS}(S,T) = 5$.

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 $N = O(mn)$. 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 $\text{LCS}(S,T)$ equals $n + m$ minus the length of the shortest path from $(0, 0)$ to $(m, n)$ in $G$.

![Figure 1](image-url) The alignment graph for $S = \text{abac}$ and $T = \text{abcab}$. 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 $\text{aba}$ 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 $O(N \log N)$ time, requires $O(N \log N)$ space, and can report the distance between any vertex $u$ of $h$ and any other vertex $v$ in the graph in $O(\log N)$ 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 $\rho$ in time $O(|\rho| \log \log \Delta)$, where $\Delta$ is the maximum degree of a vertex in $G$. Second, to support the following queries in $O(\log N)$ 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](image) 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 $\partial P$ 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 $\mathcal{A}(P)$ the set of boundary vertices of $P$ that are either rightmost or bottommost in $P$, and by $\mathcal{P}(P)$ the set of boundary vertices of $P$ that are either leftmost or topmost in $P$. We consider each of $\mathcal{A}(P)$ and $\mathcal{P}(P)$ 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 $\mathcal{A}(P)$ and $\mathcal{P}(P)$ as sequences. We define the outside of $P$, denoted by $P^\text{out}$, 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 $v \in P^\text{out}$ must contain at least one vertex from $\mathcal{A}(P)$, and any path from a vertex $u \notin P$ to a vertex $u \in P$ must contain at least one vertex from $\mathcal{P}(P)$. 


For any \( r \in [1, nm] \), an \( r \)-division of \( G \) is a decomposition of \( G \) to pieces of size \( O(r) \), each with \( O(\sqrt{r}) \) 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 thus also be represented by a tree, which we will denote by \( T \).

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 \( \text{VD}(S, \omega) \) of \( H \) is a partition of the vertices of \( H \) into pairwise disjoint sets, one set \( \text{Vor}(s) \) for each site \( s \in S \). The set \( \text{Vor}(s) \), called the Voronoi cell of \( s \), contains all vertices of \( H \) that are closer (w.r.t. \( d^\omega(\cdot, \cdot) \)) to \( s \) than to any other site in \( S \). If \( v \in \text{Vor}(s) \) 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 \( P^{\text{out}} \) of a piece \( P \in \mathcal{A} \) with sites \( S \subseteq \mathcal{A}(P) \). We next discuss the structure of such Voronoi diagrams.

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

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

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

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

\textbf{Lemma 7.} For any row \( a \) and any site \( s \), the vertices of row \( a \) that belong to \( \text{Vor}(s) \) 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 \( j_a \leq j_{a+1} \).

\textbf{Lemma 8.} There is a rightmost vertex \( s_\triangledown \) in \( \text{Vor}(s) \) that is also a bottommost one.

\textbf{Lemma 9.} For every \( v \in \text{Vor}(s) \), \( \text{Vor}(s) \) contains a path from \( v \) to \( s_\triangledown \).

Our representation of a Voronoi diagram for \( P\text{out} \) with sites \( \mathcal{J}(P) \) consists of the following. For each \( s \in \mathcal{J}(P) \), we store:
1. the rightmost bottommost vertex \( s_\triangledown \) of \( \text{Vor}(s) \), defined in Corollary 8,
2. a vertex \( \text{last}(s, s_\triangledown) \) on a shortest \( s\text{-to}-s_\triangledown \) path, whose definition will be given later.

\section{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 \((r_2, \ldots, r_1)\)-division of \( G \) for some \( N = r_1 > \cdots > r_1 = O(1) \) to be specified later. Recall that our convention is that the \( r_1\)-division consists of \( G \) itself. Further, we set the \( r_1\)-division to consist of pieces of size \( 2 \times 2 \). 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 \( r_1\)-division by \( \mathcal{R}_1 \). Let \( \mathcal{T} \) denote the tree representing this recursive \((r_1, \ldots, r_0)\)-division, where each singleton piece \( \{v\} \) at level 0 is attached as a child of a piece \( P \) at level 1 such that \( v \in \mathcal{J}(P) \)—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 \( \mathcal{J}(P) \) 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 \( \mathcal{J}(P) \) for \( Q \setminus (P \setminus \partial P) \).
3. If \( i < t - 1 \), for each vertex \( u \in \mathcal{J}(P) \) we store \( \text{VD}(u, Q) \): the Voronoi diagram for \( Q\text{out} \) with sites \( \mathcal{J}(Q) \) 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 \( \text{last}(s, s_\triangledown) \). 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 \( u \in \mathcal{J}(R_i) \) for a non-empty prefix of
The piece $R_0$ consists of just $u$, and the piece $R_4$ is the entire alignment graph. Note that $u \in \text{last}(R_1)$. In this example, $\text{lev}(u) = 1$, $\text{anc}(u,v) = 3$, and $\text{anc}(u,v) = 4$ (because $v \in \partial R_3$). A $u$-to-$w$ shortest path $\rho$ is shown in dashed red. $\text{last}(u,w)$ is the last vertex of $\rho$ that belongs to $\text{last}(R_2)$, which is the vertex $z$. The distance from $u$ to $w$ is $\text{dist}(u,z) + \text{dist}(z,w)$. $\text{dist}(u,z)$ is stored in the reverse MSSP of $R_2$. $\text{dist}(z,w)$ is stored in the MSSP of $R_3 \setminus (R_2 \setminus \partial R_2)$. Similarly, a $u$-to-$v$ shortest path $\rho$ is shown in dashed blue. Because $v \in \partial R_3$, $\text{last}(u,v)$ is $v$ itself.

By 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 $A$ require $\mathcal{O}(N \log^2 N)$ space, since $\sum_{P \in A} |P| = \mathcal{O}(N \log N)$ 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 $R_i$, 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 $R_i$, we store $\mathcal{O}(\sqrt{r_i})$ Voronoi diagrams each of size $\mathcal{O}(\sqrt{r_i})$. The stated bound follows. ▷
We initially set \( Q \). To reduce clutter, let us denote the set of vertices of \( Q \) that intersect row \( x \) by \( \mathcal{R}_x \). As before, let \( R_1, R_2, \ldots, R_h \) 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 \( \ell = \text{lev}(u) \) and \( h = \text{anc}(u,v) \). We then have \( u \in \mathcal{J}(R_{\ell}) \) and \( v \in R_{h}^{\text{out}} \). We will answer \( \text{dist}(u,v) \) by identifying \( \text{last}(u,v) \), which lies on \( \mathcal{J}(R_{h-1}) \). As explained above, we can then obtain \( \text{dist}(u,v) \) from the MSSP data structures stored for \( R_{h-1} \). See Algorithm 1.

**Algorithm 1** \( \text{DIST}(u,v) \)

1: if \( u \) and \( v \) belong to the same piece in \( \mathcal{R}_1 \) then
2: return the answer by brute force
3: \( w \leftarrow \text{GetLast}(u,v) \)
4: return \( \text{dist}(u,w) + \text{dist}(w,v) \)

We now show how to implement the procedure \( \text{GetLast} \) for finding \( \text{last}(u,v) \); see Algorithm 2 for a pseudocode. First, note that if \( h = \ell + 1 \) we can simply return \( u \) as \( \text{last}(u,v) \). Hence, in what follows, we assume that \( h > \ell + 1 \). The procedure \( \text{GetLast} \) 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 \( \mathcal{J}(R_{i-1}) \) 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 \( \text{GetNextCandidates} \) that adds at most two vertices of \( \mathcal{J}(R_{i}) \) to \( W_{i} \). The guarantee is that, if \( w \) is a vertex of \( \mathcal{J}(R_{i-1}) \) 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 \( 2^\ell \) vertices of \( \mathcal{J}(R_{h-1}) \), one of which can be returned as \( \text{last}(u,v) \). To figure out which one, for each such vertex \( w \), we use the MSSP data structures to compute \( \text{dist}(u,w) + \text{dist}(w,v) \), and return a vertex for which the minimum is attained.

**Algorithm 2** \( \text{GetLast}(u,v) \)

1: \( \ell \leftarrow \text{lev}(u) \)
2: \( h \leftarrow \text{anc}(u,v) \)
3: \( W_{\ell} \leftarrow \{ u \} \)
4: for \( i = \ell + 1 \) to \( h - 1 \) do
5: \( W_{i} \leftarrow \emptyset \)
6: for each \( w \in W_{i-1} \) do
7: \( W_{i} \leftarrow W_{i} \cup \text{GetNextCandidates}(w,i,v) \)
8: return argmin_{w \in W_{h-1}} \text{dist}(u,w) + \text{dist}(w,v) \)

It remains to describe the procedure \( \text{GetNextCandidates} \). Consider any \( w \in W_{i-1} \). To reduce clutter, let us denote \( R_i \) by \( Q \). Since \( w \in \mathcal{J}(R_{i-1}) \), the Voronoi diagram \( \text{VD}(w,Q) \) for \( Q^{\text{out}} \) is stored by the oracle. The procedure \( \text{GetNextCandidates} \) finds two sites of \( \text{VD}(w,Q) \), 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 \( \text{VD}(w,Q) \) is a vertex of \( \mathcal{J}(Q) \) on a shortest \( u \)-to-\( v \) path.

Let \((x, y)\) be the bottom-right vertex of \( Q \). Every vertex \( v = (x_v, y_v) \) of \( Q^{\text{out}} \setminus \partial Q \) either has \( x_v > x \) or \( y_v > y \). We describe the case when \( x_v > x \); the case when \( y_v > y \) is analogous. Let \( \Gamma \) denote the set of \( s \rightarrow s\downarrow \) paths \( \rho_s \) stored in \( \text{VD}(w,Q) \) according to the order of the sites \( s \) along \( \mathcal{J}(Q) \). For every row \( x > x \), let \( \Gamma_x \) denote the subset of paths in \( \Gamma \) that intersect row \( x \), ordered according to the order of the intersection vertices along row \( x \).
Lemma 11. For every \( x, x' \) with \( x > x' \), \( \Gamma_x \) is a subsequence of \( \Gamma_{x'} \).

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

We define the set of critical rows to be all rows \( x \) such that \( (x,y) = s_\gamma \) for some \( y \in [0,n] \) and \( s \in \mathcal{J}(Q) \). Lemma 11 implies that if we consider the evolution of the sequences \( \Gamma_x \) as \( x \) increases from \( x_{\text{to}} \) to \( m \) as a dynamic process, changes occur only at critical rows. More precisely, if the row of \( s_\gamma \) 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 \( \Gamma_x \) 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 \( \Gamma_{x_{\text{to}}+1} \). Then, we go over the critical rows in increasing order, and remove the path \( \rho_s \) from the BST when we reach the row of \( s_\gamma \).

For any \( x_{\text{to}} < x \leq m \), we can access the BST representation of \( \Gamma_x \) 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 \( \rho_s \) 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 \( \rho_s \). 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 \( \Gamma_x \).

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

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

Lemma 12. The oracle answers distance queries in time \( O(2^t \log^2 N) \).

Proof. First, \( \text{lev}(u) \) and \( \text{anc}(u,v) \) can be (naively) computed in \( O(\log N) \) time by going over the ancestors of \( \{u\} \) in \( T \): for each (rectangular) ancestor piece \( R \) of \( \{u\} \), in \( O(1) \) 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 \( \text{dist}(u,v) \) query, we make \( O(2^t) \) calls to \( \text{GetNextCandidates} \), each requiring \( O(\log^2 N) \) time, for a total of \( O(2^t \log^2 N) \) time. Finally, we make \( O(2^t) \) queries to the MSSP data structures, requiring \( O(2^t \log N) \) time in total.

Remark 13. Since our query procedure computes \( \text{last}(u,v) \), and we have MSSP data structures that capture the \( u \)-to-\( \text{last}(u,v) \) and the \( \text{last}(u,v) \)-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].

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

\textbf{Proof.} The space of the oracle is \( O \left( N \log^2 N + N \log N \cdot \sum_{i=0}^{t-1} r_{i+1}/r_i \right) \) 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, \( t = O(\log_p N) \) and the space becomes \( O(N \log^2 N + N \log N \cdot p \log_p N) \).

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 \( O(N \log N \cdot 2^{\log N / g(N)} \log N) = N \log^{2+o(1)} N \) 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. \hfill \blacksquare

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

\begin{itemize}
\item \textbf{Observation 15.} The query algorithm for \( \text{dist}(u, v) \) takes \( O(2^{\text{lev}(u)} \log^2 N) \) time and uses only Voronoi diagrams \( \text{VD}(u, Q) \) for \( Q \in R_i \) with \( i > \text{lev}(u) \).
\end{itemize}

\section{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 \((r_1, \ldots, r_0)\)-division and all of the MSSP structures stored for all pieces in \( \mathcal{A} \) can be done in \( O(N \log^2 N) \) 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 \( Q^{\text{out}} \) with sites a subsequence \( U \) of \( \mathcal{J}(Q) \)—we will only build Voronoi diagrams with \( U = \mathcal{J}(Q) \), but during the analysis, we will also consider Voronoi diagrams with sites \( U \subseteq \mathcal{J}(Q) \). In what follows, when we talk about a piece \( H \neq Q \), we will really mean its intersection with \( Q^{\text{out}} \), assuming that it is non-empty. Similarly, when we talk about \( \partial H, \tau(H) \), and \( \mathcal{J}(H) \) we will really mean the intersection of \( \partial H, \tau(H) \), and \( \mathcal{J}(H) \) with \( Q^{\text{out}} \), respectively. See Figure 5 for an illustration.

\begin{itemize}
\item \textbf{Lemma 16.} Let \( X \in \{ \tau(H), \mathcal{J}(H) \} \). 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 \( v \).
\end{itemize}

\textbf{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-\( u \) path belong to \( \text{Vor}(s_a) \). Towards a contradiction, suppose that \( s_a \) 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 \( s_a \)-to-\( u \) path must cross the shortest \( s_v \)-to-\( v \) path in some vertex \( b \). Then \( b \in \text{Vor}(s_a) \cap \text{Vor}(s_v) \), which is a contradiction as Voronoi cells are disjoint. \hfill \blacksquare

The above lemma means that the vertices of each of \( \tau(H) \) and \( \mathcal{J}(H) \) 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 \( \tau(H) \) or \( \mathcal{J}(H) \) 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 $A$ (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 $s \downarrow$ if and only if $s$ is the site of some vertex in $\tau(H)$, and $s$ is not the site of any vertex in $\cup(H)$.

Lemma 17. For any $s \in S$ and any level $\ell$, there is a unique level-$\ell$ piece $H \in A$ for which $Vor(s) \cap \tau(H) \neq \emptyset$ and $Vor(s) \cap \cup(H) = \emptyset$, and this piece contains $s \downarrow$.

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

Remark 18. The condition in the statement of Lemma 17 is equivalent to: $s \downarrow \in H \setminus \cup(H)$.

Lemma 17 provides a criterion on the partitions of $\tau(H)$ and $\cup(H)$ for determining whether a piece $H$ contains any vertex $s \downarrow$. The following lemma describes a binary search procedure, PARTITION, which gets as input a sequence $U$ of candidate sites, and returns the partition of $\tau(H)$ or $\cup(H)$; i.e., the subsequence of sites in $U$ whose Voronoi cells contain the vertices of $\tau(H)$ or $\cup(H)$. The procedure PARTITION will be a key element in the overall construction algorithm. The following lemma describes an implementation of PARTITION using distance queries $\text{dist}(u, v)$ with $u \in \cup(H)$ 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 $U \subseteq \cup(Q)$ of sites and their additive weights, we can perform the procedure PARTITION (that computes a partition of $\tau(H)$ or $\cup(H)$ w.r.t. $U$) in the time required by $O(|U| \cdot \log n)$ distance queries $\text{dist}(u, v)$ with $u \in \cup(Q)$ and $v \in \partial H$. We will ensure that such queries can be answered efficiently whenever PARTITION is called by the main algorithm.

Proof. We will only prove the statement for $\tau(H)$ as the case of $\cup(H)$ is analogous. We start with a single interval, which is all of $\tau(H)$. 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 $\tau(H)$ 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 inactive. 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 \( O(\log n) \) distance queries \( \text{dist}(u, v) \), with \( u \in \mathcal{J}(Q) \) 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 \( (r_1, \ldots, r_0) \)-division.

Consider some level \( i \), and assume that we have already computed all Voronoi diagrams \( \text{VD}(u, R) \) for pieces \( R \in \bigcup_{j>i} \mathcal{R}_j \). Let \( Q \in \mathcal{R}_{i} \) be the parent of \( P \) in \( \mathcal{T} \). Our goal is to compute, for every \( u \in \mathcal{J}(P) \), the representation of \( \text{VD}(u, Q) \), the Voronoi diagram of \( Q^\text{out} \) with sites \( \mathcal{J}(Q) \) and additive weights \( \text{dist}(u, \mathcal{J}(Q)) \). Recall that this representation consists of the vertices \( s \) and \( \text{last}(s, s) \) for every site \( s \in \mathcal{J}(Q) \). We would like to compute this representation in time roughly proportional to its size \( |\mathcal{J}(Q)| \).

By Observation 15, using the already computed parts of the oracle for levels \( j > i \), we can already answer any distance query \( \text{dist}(s, v) \) for any \( s \in \mathcal{J}(Q) \) and any \( v \in Q^\text{out} \) in \( O(2^i \log^2 n) \) time. These are precisely the distance queries required for computing partitions of pieces \( H \) in \( Q^\text{out} \) w.r.t. sites in \( \mathcal{J}(Q) \) (Lemma 19).

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

We shall find the vertices \( s \_ \_ \) of \( \text{VD}(u, Q) \) in each such piece \( H \) separately. We invoke \textsc{Partition} on \( \tau(H) \) and on \( \mathcal{J}(H) \), and use Lemma 17 to determine whether \( H \) contains any vertices \( s \_ \_ \). If so, we zoom in on each of the two child pieces of \( H \) in \( \mathcal{A} \) until, after \( O(\log N) \) steps, we get to a constant-size piece, in which we can find \( s \_ \_ \) by brute force. Note, however, that we are aiming for a running time that is roughly proportional to \( |\mathcal{J}(Q)| \), but that the running time of \textsc{Partition} depends on the number of sites \( U \) w.r.t. which we partition. This is problematic since, e.g., when \( H \) contains \( s \_ \_ \) just for a single site \( s \), we can only afford to invest \( O(1) \) time in locating \( s \_ \_ \) in \( H \). In this case, computing the partition w.r.t. \( |\mathcal{J}(Q)| \) 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 \( |\tau(H)| \), 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 \( s \_ \_ \) in \( H \) (actually in \( H \setminus \mathcal{J}(H) \)), rather than to the size of \( H \) or of \( \tau(H) \). We call such a sequence a \textit{safe} sequence of sites for \( H \), which we now define formally. Recall that the Voronoi diagram \( \text{VD}(u, Q) \) of \( Q^\text{out} \) has sites \( \mathcal{J}(Q) \). Let \( U \) be a subsequence of \( \mathcal{J}(Q) \). Consider the Voronoi diagram \( \text{VD} \) of \( Q^\text{out} \) whose sites are the elements of \( U \) (with the same additive distances as in \( \text{VD}(u, Q) \)). We
say that $U$ is safe for $H$ if and only if the sets $\{(s, s_\prec) : s$ is a site and $s_\prec \in H \setminus j(H)\}$ are identical for $VD'$ and $VD(u, Q)$.

\begin{observation}
A sequence that is safe for $H$ is also safe for any child $H'$ of $H$ in $A$.
\end{observation}

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

\begin{algorithm}
\begin{algorithmic}
\STATE $\text{Input:}$ The additive weight $\omega(s)$ for each $s \in j(Q)$, a piece $H$ in $A$ that is internally disjoint from $Q$ and has a non-empty intersection with $Q^{out}$, and a sequence $U \subseteq j(Q)$ that is safe for $H$.
\STATE $\text{Output:}$ All vertices $s_\prec$ for $s \in j(Q)$ that belong to $H$.
\STATE 1: \textbf{if} $|H| = 4$ \textbf{then}
\STATE 2: \quad Find all vertices $s_\prec$ in $H$ for all $s \in U$ by computing $\omega(s) + \text{dist}(s, v)$ for all $s \in U$ and all $v$ adjacent to some vertex of $H$.
\STATE 3: \textbf{for} each child $H'$ of $H$ in $A$ \textbf{do}
\STATE 4: \quad $Z \leftarrow \text{Partition}(U, \omega, j(H'))$
\STATE 5: \quad $Z' \leftarrow \text{Partition}(U, \omega, j(H'))$
\STATE 6: \quad $L := Z \setminus Z'$
\STATE 7: \textbf{if} $L \neq \emptyset$ \textbf{then}
\STATE 8: \quad $V := Z \setminus \{z \in Z \setminus L : \text{both the predecessor and successor of } z \text{ in } Z \text{ are not in } L\}$
\STATE 9: \quad $\text{Zoom}(V, \omega, H')$
\end{algorithmic}
\end{algorithm}

The procedure $\text{Zoom}$ takes as input a piece $H$ and a sequence $U \subseteq j(Q)$ that is safe for $H$. In order to compute $VD(u, Q)$, we call the procedure $\text{Zoom}(j(Q), \omega, H)$ 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 $j(Q)$ is a safe sequence of the respective piece. For each child $H'$ of $H$ in $A$, 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 $L = \{s \in j(Q) : s_\prec \in H' \setminus j(H')\}$ 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 $\text{Zoom}(V, \omega, H')$. In order to prove the correctness of procedure $\text{Zoom}$, it remains to show that $V$ is indeed safe. The following two lemmas show this (see also Figure 6).

\begin{lemma}
Let $U$ be safe for a piece $H$, such that $\text{Partition}(U, \omega, j(H)) = U$. Suppose that there are three elements $u_1, u_2, u_3$ of $U$ that appear consecutively (in this order) in both $\text{Partition}(U, \omega, j(H))$ and $\text{Partition}(U, \omega, j(H'))$. Then, $U \setminus \{u_2\}$ is also safe for $H$.
\end{lemma}

\begin{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 $\text{Vor}(\cdot)$, and those of $VD'$ by $\text{Vor}'(\cdot)$. Note that for every $u \in U \setminus \{u_2\}$, $\text{Vor}(u) \subseteq \text{Vor}'(u)$. By Lemma 17, in $VD$, $u_1 \setminus u_2 \setminus u_3 \setminus H \setminus j(H)$. Hence, in $VD'$, $u_1 \setminus u_2 \setminus u_3 \setminus H \setminus j(H)$.

\begin{claim}
Every vertex $y$ of $\text{Vor}'(u_2) \cap H$ belongs either to $\text{Vor}'(u_1)$ or to $\text{Vor}'(u_3)$.
\end{claim}

\begin{proof}
Consider the last vertex $z_1$ of $j(H)$ that is in $\text{Vor}(u_1)$, and the first vertex $z_3$ of $j(H)$ that is in $\text{Vor}(u_3)$. Let $p_1$ be a shortest $u_1$-to-$z_1$ path, and $p_3$ be a shortest $u_3$-to-$z_3$ path. Note that all vertices of $p_1$ belong to $\text{Vor}(u_1)$ and all vertices of $p_3$ belong to $\text{Vor}(u_3)$. Consider any vertex $y$ of $\text{Vor}(u_2) \cap H$. The vertex $y$ lies to the right of $p_1$ and to the left of $p_3$. In $VD'$, the vertices of $p_1$ belong to $\text{Vor}'(u_1)$ and the vertices of $p_3$ belong to $\text{Vor}'(u_3)$.
\end{proof}
Figure 6 Illustration for Lemma 21. Part of $Q^{\text{out}}$ (pink) for some piece $Q$ (gray) is shown. A piece $H$ is indicated by a black rectangle. The sites of $\partial(Q)$ are numbered 1 through 7. The partition of $\partial(H)$ w.r.t. $\partial(Q)$ is $U = (1, 2, 4, 5, 6, 7)$. Hence, $U$ is safe for $H$. The partition of $\partial(H)$ w.r.t. $\partial(Q)$ is $(1, 4, 5, 6, 7)$. Since sites 1, 2, 4 are consecutive in both partitions, $(1, 4, 6, 7)$ is also safe for $H$. Further, $\partial(Q)$ is also clearly safe for $H$. However, $(1, 3, 4, 5, 6, 7)$ may not be safe for $H$, as $3\downarrow$ 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 $\text{PARTITION}(U, \omega, \partial(H)) = U$, the only such sites are $u_1$ and $u_3$, and the claim follows.

By the above claim, the sets $\{(s, s_{\downarrow}) : s \text{ is a site and } s_{\downarrow} \in H \setminus \partial(H)\}$ 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 $\text{ZOOM}(V, \omega, H')$ made by procedure $\text{ZOOM}(U, \omega, H)$ for a child $H'$ of $H$ in $A$, $V$ 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 $s_{\downarrow} \in H' \setminus \partial(H')$.

$Z$ 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 $O(\log N)$ calls to $\text{ZOOM}(U, \omega, H)$, each with $U = \partial(Q)$. 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 \( \{ s \in \mathcal{J}(Q) : s \succ w \in H^t \setminus \mathcal{J}(H^t) \} \). Note that in each level of the tree \( \mathcal{A} \) each \( s \in \mathcal{J}(Q) \) is an element of exactly one such set. Hence, each \( s \in \mathcal{J}(Q) \) 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 \( s \succ w \) for all \( s \in \mathcal{J}(Q) \) reduces to \( \mathcal{O}(|\mathcal{J}(Q)| \cdot \log^2 N) \) distance queries \( \text{dist}(u, v) \), with \( u \in \mathcal{J}(Q) \) and \( v \in Q^{\text{out}} \). 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 \( \text{dist}(s, s \succ w) \) query also computes \( \text{last}(s, s \succ w) \), and hence these values can also be retrieved in \( \mathcal{O}(2^t \log^2 N) \) time. Thus, \( \text{VD}(u, Q) \), which is of size \( \mathcal{O}(|\mathcal{J}(Q)|) \) can be computed in time \( \mathcal{O}(|\mathcal{J}(Q)| \cdot 2^t \log^4 N) \), which is \( |\mathcal{J}(Q)| \cdot N^{o(1)} \) for both choices of \( t \) in Proposition 14. Therefore, the time to compute \( \text{VD}(u, Q) \) for all pieces is \( N^{o(1)} \cdot \sum_{t} N \cdot |\mathcal{J}(Q)| = N^{o(1)} \cdot \sum_{i=0}^{t-1} \frac{N}{t^i} \sqrt{t^i} \), which is \( N^{1+\Theta(1)} \) 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 \( w_{\text{match}} \) for aligning a pair of matching letters, \( w_{\text{mis}} \) for aligning a pair of mismatching letters, and \( w_{\text{del}} \) for letters that are not aligned. One may assume without loss of generality that \( 2w_{\text{match}} \geq 2w_{\text{mis}} \geq w_{\text{del}} \) \[36\]. Given \( w_{\text{match}}, w_{\text{mis}} \) and \( w_{\text{del}} \), we define \( w'_\text{match} = 0, w'_\text{mis} = w_{\text{match}} - w_{\text{mis}} \) and \( w'_\text{del} = \frac{1}{2}w_{\text{match}} - w_{\text{del}} \). 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 \( \frac{1}{2}(m + n)w_{\text{match}} - W \)).

FR-Dijkstra. We define the dense distance graph (DDG) of a piece \( P \) as a directed bipartite graph with vertices \( \partial P \) and an edge from every vertex \( u \in \tau(P) \) to every vertex \( v \in \mathcal{J}(P) \) with weight equal to the length of the shortest \( u \)-to-\( v \) path in \( P \). We denote this graph as \( \text{DDG}_P \). \footnote{For general planar graphs, the DDG of a piece is usually defined as a complete directed graph on \( \partial P \).} 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.

\textbf{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.

\textbf{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 \[38,39\].
A warmup. Let us first show how to construct in $\tilde{O}(N)$ time an $\tilde{O}(N)$-size oracle that answers queries in $\tilde{O}(\sqrt{N})$ 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 $A'$, obtained from the recursive decomposition tree $A$ by deleting all descendants of pieces in the $r$-division. For each piece $P \in A'$, we compute and store $\text{DDG}_P$. In each of the $\mathcal{O}(\log N)$ levels of $A$, 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 $\tilde{O}(N)$.

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 $\rho$ 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 $A'$. 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 $\text{sep}(Q)$—which stands for separator. Observe that $\rho$ must contain at least one vertex from $\text{sep}(Q)$. Consider the set that consists of $P_z$ and the siblings of weak ancestors of $P_z$ in $A'$ that are descendants of $Q_z$ 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 $\rho$ between a vertex of $\partial P_z$ and a vertex of $\text{sep}(Q)$ can be partitioned into subpaths $\rho_1, \ldots, \rho_k$ such that each $\rho_i$ lies entirely within some piece $R_i$ in the cone of $P_z$ and both $\rho_i$’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 $\tilde{O}(\sqrt{N} + r)$. By setting $r = \sqrt{N}$ we get the promised complexities.

![Figure 7](image_url) The piece $Q$ is shown. $\text{sep}(Q)$ 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 $P_v$ are shaded by pink.
The tradeoff. We can now describe the entire tradeoff of Theorem 4. We assume that $r > w^2$, since otherwise $N w / \sqrt{r} = \Omega(N)$ and Theorem 4 is satisfied by the warmup solution. For a piece $P$, we will show how to store $\text{DDG}_P$ 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 $\text{DDG}_P$ in $\tilde{\mathcal{O}}(1)$ time. Our approach closely follows ideas from [2].

For the remainder, we deviate from our ordering convention of $\tau(P)$; the first vertex is now the top-right vertex of $P$, and the last is the bottom-left one. $\tau(P)$ 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 $\tau(P)$ by $v_i$ and the $j$-th vertex of $\tau(P)$ 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 $\text{DDG}_P$ to be finite. To achieve this, for each edge between two vertices of $\tau(P)$, 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 $\text{DDG}_P$ with entry $M[i,j]$ storing the distance from $v_i$ to $u_j$, and let $k = |\tau(P)| = |\tau(P)|$. Matrix $M$ satisfies the Monge property, namely:

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

(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

$$|M[i+1,j] - M[i,j]| \leq w.$$  

(2)

This is because edges $v_i v_{i+1}$ and $v_{i+1} u_i$ 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 $v_i v_{i+1}$ 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


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

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

(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 $\text{DDG}_P$ 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


An Almost Optimal Edit Distance Oracle


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