Quantum Information and Computation, Vol. 24, No. 3&4 (2024) 0267–0294

QUANTUM ALGORITHMS FOR THE SHORTEST COMMON SUPERSTRING AND TEXT ASSEMBLING PROBLEMS

KAMIL KHADIEV
Institute of Computational Mathematics and Information Technologies, Kazan Federal University Kremlyovskaya, 35, Kazan, Russia

CARLOS MANUEL BOSCH MACHADO
Institute of Computational Mathematics and Information Technologies, Kazan Federal University, Kremlyovskaya, 35, Kazan, Russia

ZEYU CHEN
School of Mathematics Sciences, Zhejiang University, Hangzhou 310058, P. R. China

JUNDE WU
School of Mathematics Sciences, Zhejiang University, Hangzhou 310058, P. R. China

Received October 15, 2023
Revised February 24, 2024

In this paper, we consider two versions of the Text Assembling problem. We are given a sequence of strings $s^1, \ldots, s^n$ of total length L that is a dictionary, and a string t of length m that is a text. The first version of the problem is assembling t from the dictionary. The second version is the "Shortest Superstring Problem"(SSP) or the "Shortest Common Superstring Problem"(SCS). In this case, t is not given, and we should construct the shortest string (we call it superstring) that contains each string from the given sequence as a substring. These problems are connected with the sequence assembly method for reconstructing a long DNA sequence from small fragments. For both problems, we suggest new quantum algorithms that work better than their classical counterparts. In the first case, we present a quantum algorithm with $O(m+\log m\sqrt{nL})$ query complexity. In the case of SSP, we present a quantum algorithm with $\tilde{O}(n^3 1.728^n + L + n^{1.5}\sqrt{L})$ query complexity. Here $\tilde{O}$ hides not only constants but logarithms of L and n also.

Keywords: quantum algorithms, shortest superstring, strings, DNA assembly

1 Introduction

In this paper, we are interested in running-time-efficient solutions for two problems that are The Shortest Common Superstring Problem and The Text Assembling Problem. In the general case, the problem is as follows. For a positive integer n, a sequence of n strings $S = (s^1, \ldots, s^n)$ is given. We call it a dictionary. We assume that the total length of the dictionary strings is $L = |s^1| + \cdots + |s^n|$. Additionally, a string t of length $m = |t|$ is given. We call it text. Our goal is to assemble t from the dictionary strings S. Here we have two types of the problem:

- **The Shortest Common Superstring Problem (SCS).** It is also known as the "Shortest Superstring Problem"(SSP). In this case, the text t is not given. We should construct the shortest string t (we also call it superstring) that contains each string from the dictionary S as a substring.

- **The Text Assembling Problem.** The string t is given, and we should assemble t only using strings from S. Here, we can use a string from dictionary S several times or not use it at all. We allow overlapping of dictionary strings during the assembly process.

These problems are connected with the sequence assembly method for reconstructing a long DNA sequence from small fragments [65] which is a well-known problem in bioinformatics. The sequence assemble problem has two types. The first one is the Reference-guided genome assembly method that constructs an existing long DNA string from the sequence S. For the problem, we should know the string t apriori and check whether we can construct it from S. This case is close to The Text Assembling Problem. The second type of the sequence assemble problem is de-novo assembly; in this problem, we do not have the string t at all, and we should construct it using all strings from S. The Shortest Superstring Problem is used as one of the main tools for solving de-novo assembly problems [21]. The problem has applications in other areas such as virology and immunology (the SCS models the compression of viral genome); the SCS can be used to achieve data compression; in scheduling (solutions can be used to schedule operations in machines with coordinated starting times), and others. According to [60, 62, 63, 70], The Shortest Common Superstring Problem is NP-hard. So, approximation algorithms are also explored, the best-known algorithm is [37]. At the same time, researchers are interested in exact solutions also. The algorithm based on [20, 32] has $O^*(2^n + nL)$ query complexity. Here O^* notation hides polynomial factors. If we have a restriction on the length of the strings s^i, then there are better algorithms. If a length $|s^i| \leq 3$, then there is an algorithm [28] with $O^*(1.443^n)$ query complexity. For a constant c, if a length $|s^i| \leq c$, then there is a randomized algorithm [29] with $O^*(2^{(1-f(c))n})$ query complexity where $f(c) = 1/(1 + 2c^2)$. We can see that $2^{(1-f(c))n} \geq 1.851^n$ for any $c \geq 2$.

The Text Assembling Problem is much easier. It was considered in [49]. The authors of that paper presented a deterministic algorithm with $\tilde{O}(m+L)$ and a lower bound is $\Omega(m+L)$. Here $\tilde{O}$ hides not only constants but logarithms of L and n also. The other version of the problem that does not allow overlapping of string on assembling process [48] has a randomized algorithm with $\tilde{O}(m\sqrt{L} + L)$ query complexity and a lower bound is also $\Omega(m + L)$.

We refer to [66, 11, 41, 1] for a good introduction to quantum algorithms. There are many problems where quantum algorithms outperform the best-known classical algorithms. Some of them can be found here [24, 34]. Problems for strings are examples of such problems [38, 46, 68, 54, 55, 10, 64, 5, 42, 50]. One of the most popular performance metrics for quantum algorithms is *query complexity*. So, we explore problems from this point of view.

The best-known quantum algorithm for The Text Assembling Problem was presented in [49] and has $O\left(m + \log m \cdot (\log n + \log\log m) \cdot \sqrt{n \cdot L}\right)$ query complexity. Note, that in the non-overlapping case [48], it is $\tilde{O}(mL^{1/4} + \sqrt{nL})$. The quantum lower bounds for both cases [49, 48] are $\Omega(\sqrt{m} + \sqrt{L})$. A quantum algorithm for SCS is not known at the moment.

In this paper, we present new quantum algorithms for these two problems:

- **Shortest Common Superstring Problem(SCS).** We present a quantum algorithm for the SCS problem with $\tilde{O}\left(n^3 1.728^n + L + n^{1.5}\sqrt{L}\right)$ query complexity. The algorithm is based on Grovers search algorithm [30, 22], Maximum search algorithm [25, 26] and the Dynamic programming approach for a Boolean cube [16, 20, 32]. Additionally, we used a quantum algorithm based on quantum string matching algorithm [68] for searching duplicates in S, and checking whether a string is a substring of one another string from S. We have this subproblem because strings in S can have different lengths. As far as we know, our algorithm is the first quantum algorithm for the SCS problem.

- **The Text Assembling Problem** We present a quantum algorithm with $O(m+\log m \cdot \sqrt{nL})$ query complexity. The algorithm is a modification of the algorithm from [49] and uses a technique from [52, 56]. The new idea is $(\log n + \log\log m)$ times faster than the known algorithm in the case the dictionary S has a big size. It achieves a quantum speed-up if $O(n)$ strings from S have length at least $|s^i| = \omega(\log^2 m)$, and $m = O(\log m \cdot \sqrt{nL})$. The condition is better than it was in [49] and allows us to obtain a quantum speed-up in much more real-world cases.

The structure of this paper is the following. Section 2 contains preliminaries. We discuss the SCS problem in Section 4, and the Text Assembling problem in Section 5. Section 6 concludes the paper.

The paper is the extended version of the paper [47] that was presented at the International Conference on Micro- and Nano-Electronics 2021.

2 Preliminaries

Let us consider a string $u = (u_1, \ldots, u_m)$. Let $u[i, j]$ denote a substring $(u_i, \ldots, u_j)$ for $1 \leq i \leq j \leq m$. Let $|u| = m$ be the length of a string u. We assume that $u_i \in \Sigma$ where Σ is some finite size alphabet. For simplicity, we assume that $\Sigma = \{0, 1\}$, but all results are valid for any finite-size alphabet.

Let us present some notes on big-O notations. We assume that $O(f(n, L))$ hides constants factors; $\tilde{O}(f(n, L))$ hides logarithms of n and L; $O^*(f(n, L))$ hides polynomials of n and L.

2.1 Formal Definitions for Problems

Let us discuss formal definitions of the problems.

2.1.1 The Shortest Common Superstring Problem

For a positive integer n, a sequence of n strings $S = (s^1, \ldots, s^n)$ is given. We should construct the shortest string t (we call it superstring), i.e. $|t|$ is the minimal possible such that each s^i is a substring of t for $i \in \{1, \ldots, n\}$. In other words, for each $i \in \{1, \ldots, n\}$ there is $1 \leq q_i \leq |t|$ such that $t[q_i, q_i + |s^i| - 1] = s^i$.

Informally, we want to construct the shortest t that contains all strings from S as substrings. Let us denote the problem as $\texttt{SCS}(S)$.

2.1.2 The Text Assembling Problem.

For some positive integers n and m, a sequence of n strings $S = (s^1, \ldots, s^n)$ is given. We call S a dictionary. Additionally, we have a string t of length m. We call t a text. We should

present a sequence $s^{i_1}, \ldots, s^{i_r}$ and positions $q_1, \ldots, q_r$ such that $q_1 = 1$, $q_r = n - |s^{i_r}| + 1$, $q_j \leq q_{j-1} + |s^{i_{j-1}}|$ for $j \in \{2, \ldots, r\}$. Additionally, $t[q_j, q_j + |s^{i_j}| - 1] = s^{i_j}$ for $j \in \{1, \ldots, r\}$. Note that a sequence $(i_1, \ldots, i_r)$ can have duplicates.

Informally, we want to construct t from S with possible overlapping. Let us denote the problem as $\texttt{TAO}(t, S)$.

2.2 Connection with Real-World Problems

The considered problems have a strong relationship with the sequence assembly method for reconstructing a long DNA sequence from small fragments [65]. Two types of the sequence assemble problem exist. The first one is de novo assembly. In this problem, we do not have the string t and should construct it using all strings from the dictionary. This problem is NP-complete and typically is solved by heuristic algorithms. The SCS problem is one of the possible interpretations of this problem.

The second type is Reference-guided genome assembly. This problem is similar to $\texttt{TAO}(t, S)$ problems but we should use a string from the dictionary only once. This differs from our problem where we can use strings from the dictionary S several times. There are polynomial algorithms for the Reference-guided genome assembly problem that are presented in [19, 67]. The current methods for DNA assembling belong to the "next-generation" [71] or "second-generation" sequencing (NGS). They allow us to read many short substrings of DNA in a parallel way. Typically, the length of a DNA sequence (that is t in our case) is about $10^8 - 10^9$ and the length of each piece is about $10^2 - 10^4$.

The difference between our $\texttt{TAO}(t, S)$ and NGS problems is significant. At the same time, the NGS allows us to have several duplicates of a string in S. It is like relaxing the "single usage of a string from S" condition. It allows us to use a string from S the fixed number of times. That is why our ideas can be used as a possible solution for the sequence assembly method in real-world examples.

An additional difference between the problems (in both cases de novo assembly and Reference-guided genome assembly) is the possibility of errors in the string t for the case of DNA sequence assembling [69]. That is not allowed in our problems. That is why our algorithm should have improvement if it is used for DNA sequence assembling.

At the same time, our problems and algorithms are interesting as is because they solve fundamental problems and have applications in other areas like belonging a text to some natural language and others.

2.3 Quantum Query Model

We use the standard form of the quantum query model. Let $f : D \to \{0, 1\}, D \subseteq \{0, 1\}^N$ be an N variable function. An input for the function is $x = (x_1, \ldots, x_N) \in D$ where $x_i \in \{0, 1\}$ for $i \in \{1, \ldots, N\}$.

We are given oracle access to the input x, i.e. it is implemented by a specific unitary transformation usually defined as $|i\rangle|z\rangle|w\rangle \to |i\rangle|z + x_i \pmod 2\rangle|w\rangle$ where the $|i\rangle$ register indicates the index of the variable we are querying, $|z\rangle$ is the output register, and $|w\rangle$ is some auxiliary work-space. It can be interpreted as a sequence of control-not transformations such that we apply inversion operation (X-gate) to the second register that contains $|z\rangle$ in a case of the first register equals i and the variable $x_i = 1$. We interpret the oracle access transformation as N such controlled transformations for each $i \in \{1, \ldots, N\}$.

An algorithm in the query model consists of alternating applications of arbitrary unitaries independent of the input and the query unitary, and a measurement in the end. The smallest number of queries for an algorithm that outputs $f(x)$ with a probability that is at least $\frac{2}{3}$ on all x is called the quantum query complexity of the function f.We refer the readers to [66, 11, 1, 41] for more details on quantum computing.

In this paper, we are interested in the query complexity of the quantum algorithms. We use modifications of Grover's search algorithm [30, 22] as quantum subroutines. For these subroutines, time complexity is more than query complexity for additional log factor [17, 31].

3 Tools

Our algorithms use several data structures and algorithmic ideas like segment tree [53], suffix array [61], rolling hash [36], and prefix sum [23]. Let us describe them in this section.

3.1 Rolling Hash for Strings Equality Checking

The rolling hash was presented in [36]. For a string $u = (u_1, \dots, u_{|u|})$, we define a rolling hash function $h_p(u) = \left(\sum_{i=1}^{|u|} u_i \cdot 2^{i-1} \right) \bmod p$, where p is a prime. The presented implementation is for the binary alphabet but it can be easily extended for an arbitrary alphabet.

We can use the rolling hash and the fingerprinting method [27] for comparing two strings u and v. The technique has many applications including quantum ones [8, 39, 45, 7, 6, 14, 13, 4, 2, 3, 44, 40]. Let us randomly choose p from the set of the first r primes, such that $r \leq \frac{\max(|u|,|v|)}{\varepsilon}$ for some $\varepsilon > 0$. Due to Chinese Remainder Theorem and [27], if we have $h_p(u) = h_p(v)$ and $|u| = |v|$, then $u = v$ with error probability at most ε. If we compare δ different pairs of numbers, then we should choose an integer p from the first $\frac{\delta \cdot \max(|u|,|v|)}{\varepsilon}$ primes for getting the error probability ε for the whole algorithm. Due to Chebishev's theorem, the r-th prime number $p_r \approx r \ln r$. So, if our data type for integers is enough for storing $\frac{\delta \cdot \max(|u|,|v|)}{\varepsilon} \cdot (\ln(\delta) + \ln(\max(|u|, |v|)) - \ln(\varepsilon))$, then it is enough for computing the rolling hash.

For a string u, we can compute a prefix rolling hash, that is $h_p(u[1, i])$ for $i \in \{1, \dots, |u|\}$. It can be computed with $O(|u|)$ query and time complexity using the formula

$$h_p(u[1, i]) = \left(h_p(u[1, i-1]) + (2^{i-1} \bmod p) \cdot u_i\right) \bmod p \text{ and } h_p(u[1, 0]) = 0 \text{ by definition.}$$

Assume that we store $\mathcal{K}_i = 2^{i-1} \bmod p$. We can compute all $\mathcal{K}_i$ with no queries and $O(|u|)$ time complexity using formula $\mathcal{K}_i = (\mathcal{K}_{i-1} \cdot 2) \bmod p$.

Similarly to prefix hashes we can define and compute suffix hashes $h_p(u[j, |u|])$ for each suffix $u[j, |u|]$.

3.2 Segment Tree with Range Updates

We consider a standard segment tree data structure [53] for an array $b = (b_1, \dots, b_l)$ for some integer l. Assume that each element b_i is a pair (g_i, d_i), where g_i is a target value, that is used in the segment tree, and d_i is some additional value, that is used for another part of the algorithm. The segment tree is a full binary tree such that each node corresponds to a segment of the array b. If a node v corresponds to a segment $(b_{left}, \dots, b_{right})$, then we

store $\max(g_{left}, \ldots, g_{right})$ in the node. A segment of a node is the union of segments that correspond to their two children. Leaves correspond to single elements of the array b.

A segment tree for an array b can be constructed with $O(l)$ query and time complexity. The data structure allows us to invoke the following requests with $O(\log l)$ query and time complexity.

- **Range update.** It has four integer parameters i, j, x, y ($1 \le i \le j \le l$). The procedure should assign $g_q \leftarrow x$ and $d_q \leftarrow y$ if $g_q < x$ for $i \le q \le j$. For this purpose, it goes down from the root and searches for nodes covered by the segment $[i, j]$. Then, the procedure updates these nodes by the new maximum. Let $\textsc{Update}(st, i, j, x, y)$ be a procedure that performs this operation for the segment tree st with $O(\log l)$ query and time complexity. The detailed implementation is in Appendix A.

- **Push.** The procedure pushes all updates that were done by Update procedure before from nodes down to leaves and updates leaves with actual values. Note that updating leaves implies updating corresponding elements of the b array. Let $\textsc{Push}(st)$ be a procedure that implements the operation for the segment tree st with $O(l)$ query and time complexity. The detailed implementation is in Appendix A.

- **Request.** For an integer i ($1 \le i \le l$), we check the leaf that corresponds to b_i and return its value. Let $\textsc{Request}(st, i)$ be a function that returns b_i from the segment tree st with constant query and time complexity.

Let $\textsc{ConstructSegmentTree}(b)$ be a function that constructs and returns a segment tree for an array b with $O(l)$ query and time complexity. We refer the readers to [53] for more details on the segment tree with range updates.

3.3 Suffix Array

A suffix array [61] is an array $suf = (suf_1, \ldots, suf_l)$ for a string u where $l = |u|$ is the length of the string. The suffix array is the lexicographical order for all suffixes of u. Formally, $u[suf_i, l] < u[suf_{i+1}, l]$ for any $i \in \{1, \ldots, l-1\}$. Let $\textsc{ConstructSuffixArray}(u)$ be a procedure that constructs the suffix array for the string u. The query and time complexity of the procedure is as follows:

Lemma 1 ([57]) *A suffix array for a string u can be constructed with $O(|u|)$ query and time complexity.*

4 Shortest Common Superstring Problem

We discuss our algorithm for the SCS problem in this section. Assume that we have a pair i and j such that s^i is a substring of s^j. In that case, if a superstring t contains the string s^j as a substring, then t contains the string s^i too. Therefore, we can exclude s^i from the sequence S, and it does not affect the solution. Excluding such strings is the first step of the algorithm. In the rest part of the section, we assume that no string s^i is a substring of any string s^j for $i, j \in \{1, \ldots, n\}$.

Secondly, let us reformulate the problem in a graph form. Let us construct a complete directed weighted graph $G = (V, E)$ by the sequence S. A node v^i corresponds to the string

s^i for $i \in \{1, \dots, n\}$. The set of nodes (vertexes) is $V = (v^1, \dots, v^n)$. The weight of an edge between two nodes v^i and v^j is the length of the maximal overlap for s^i and s^j. Formally,

$$w(i,j) = \max_{1 \le r \le \min\{|s^i|,|s^j|\}} \{r : s^i[|s^i| - r + 1, |s^i|] = s^j[1, r]\}.$$

We can see that any path that visits all nodes exactly once represents a superstring. Note that no string is a substring of another one. That is why, we cannot exclude any node from the path that corresponds to a superstring. Let $P = (v^{i_1}, \dots, v^{i_\ell})$ be a path. Let the weight of the path P be $w(P) = w(v^{i_1}, v^{i_2}) + \dots + w(v^{i_{\ell-1}}, v^{i_\ell})$ that is the sum of weights of all edges from P; let $|P| = \ell$. The path that visits all nodes exactly once and has maximal weight represents the shortest superstring. We formulate the above discussion as the following lemma and present its proof in Appendix D for completeness:

Lemma 2 *The path P that visits all nodes of G exactly once and has the maximal possible weight corresponds to the shortest common superstring t for the sequence S. (See Appendix D for the proof).*

In fact, the mentioned problem on the graph G is the Travelling Salesman Problem on a complete graph. The quantum algorithm for the TSP was developed in [16]. At the same time, the algorithm in [16] skips some details. Here we present the algorithm with all details to have detailed complexity (in this paper we are interested even in log factors) and for completeness of presentation. At the same time, before TSP we have several subproblems that should be solved for SCS and have not quantum algorithms yet. These subproblems are removing substrings and constructing the graph. They are discussed in the remaining part of the section.

Let us present three procedures:

- RemovingDublicatesAndSubstrings(S) is the first step of the algorithm that removes any duplicates from S and strings that are substrings of any other strings from S. The implementation of the procedure is presented in Algorithm 1. The algorithm is following. Firstly, we sort all strings of S by the length in ascending order. Secondly, for each string s^i we check whether it is a substring of $\tilde{s}^i$. Here

  $$\tilde{s}^i = s^{i+1}\$ \dots \$ s^n,$$

  that is the concatenation of all strings from S with indexes bigger than i using \$ symbol as a separator, and \$ is a symbol that cannot be in any string. If s^i is a substring of $\tilde{s}^i$, then there is $j > i$ such that s^i is a substring of s^j or $s^i = s^j$ because all strings are separated by "non-alphabetical" symbol.

  The implementation uses $IsSubstring(s^i, \tilde{s}^i)$ subroutine that returns $True$ if s^i is a substring of $\tilde{s}^i$ and $False$ otherwise. We use quantum algorithm for the strings matching problem [68] with $O^*\left(\sqrt{|s^i|} + \sqrt{|\tilde{s}^i|}\right)$ query complexity. So, if the procedure returns true, we can exclude the string from the final sequence S.

  For access to $\tilde{s}^i$, we do not need to concatenate these strings. It is enough to implement GetSymbol(i, j) function that returns j-th symbol of $\tilde{s}^i$. The index of the string $i \in \{1, \dots, n\}$, and the index of the symbol $j \in \{1, \dots, |s^{i+1}| + \dots + |s^n| + i - 2\}$. The function has $O(1)$ query complexity. A detailed description of the implementation is presented in

Appendix E. Complexity of the RemovingDublicatesAndSubstrings(S) procedure is discussed in Lemma 3.

- ConstructTheGraph(S) constructs the graph $G = (V, E)$ by S. The main idea is the following one. We randomly choose a prime p among the first $20nL$ primes. Then, we compute the rolling hash function with respect to p for each prefix and suffix of s^i, where $i \in \{1, \dots, n\}$. Assume that we have ComputePrefixAndSuffixHashes(s^i, p) subroutine for computing the prefix and suffix hashes and storing them in an array. After that, we can take the result of the rolling hash for any prefix or suffix of s^i with constant query complexity.

  For each pair of strings s^i and s^j we define a search function $sp_{i,j} : \{0, \dots, min(|s^i|, |s^j|)\} \to \{0, 1\}$ such that $sp_{i,j}(r) = 1$ iff $h_p(s^i[r+1, |s^i|]) = h_p(s^j[1, r])$ that means $s^i[r+1, |s^i|] = s^j[1, r]$ with high probability. We define $sp_{i,j}(0) = 1$. In fact, $w(v^i, v^j)$ is the maximal 1-result argument of $sp_{i,j}$. We can find it using First One Search algorithm [26, 52, 58, 59, 35] with $O(\sqrt{min(|s^i|, |s^j|)})$ query complexity.

  The implementation of the subroutine is presented in Algorithm 2. Here we assume that we have FirstOneSearch($sp_{i,j}$) subroutine. Complexity of the procedure is discussed in Lemma 4

- ConstructSuperstringByPath(P) constructs the target superstring by a path P in the graph $G = (V, E)$. Implementation of the procedure is presented in Algorithm 3.

Algorithm 1 Implementation of RemovingDublicatesAndSubstrings(S) for $S = (s^1, \dots, s^n)$.

```
setOfDeletingIndexes ← {}
SortByLength(S)
for i ∈ {1, ..., n} do
    if IsSubstring(s^i, s̃^i) = True then                ▷ s^i is a substring of s̃^i
        setOfDeletingIndexes ← setOfDeletingIndexes ∪ {i}
    end if
end for
for i ∈ setOfDeletingIndexes do          ▷ The query complexity of the for-loop is O(n)
    Remove s^i from S
end for
n ← n − |setOfDeletingIndexes|  ▷ We update n by the actual value that is the size of S.
```

Lemma 3 *The* RemovingDublicatesAndSubstrings(S) *procedure removes duplicates with* $O(n\sqrt{L}\log^2 L \log n) = \tilde{O}(n\sqrt{L})$ *query complexity ans the error probability at most* 0.1.

Proof: Firstly, we sort all strings by the length that has $O(n)$ query complexity. Then, we invoke IsSubstring($s^i, \tilde{s}^i$) procedure n times. Due to [68] , each invocation has the following query complexity

$$O\left(\sqrt{|\tilde{s}^i|}\log\sqrt{\frac{|\tilde{s}^i|}{|s^i|}}\log|s^i| + \sqrt{|s^i|}\log^2|s^i|\right).$$

Algorithm 2 Implementation of ConstructTheGraph(S) for $S = (s^1, \ldots, s^n)$.

$V = (v^1, \ldots, v^n)$
$p \in_R \{p_1, \ldots, p_{20nL}\}$ ▷ We randomly choose a prime p among the first $20nL$ primes
for $i \in \{1, \ldots, n\}$ **do**
    ComputePrefixAndSuffixHashes(s^i, p)
end for
for $i \in \{1, \ldots, n\}$ **do**
    for $j \in \{1, \ldots, n\}$ **do**
        if $i \neq j$ **then**
            $maxOverlap \leftarrow$ FirstOneSearch($sp_{i,j}$)
            $E \leftarrow E \cup \{(v^i, v^j)\}$
            $w(v^i, v^j) \leftarrow maxOverlap$
        end if
    end for
end for
return (V, E)

Algorithm 3 Implementation of ConstructSuperstringByPath(P) for $P = (v^{i_1}, \ldots, v^{i_\ell})$.

$t = s^{i_1}$
for $j \in \{2, \ldots, \ell\}$ **do**
    $t \leftarrow t \circ s^{i_j}[w(v^{i_{j-1}}, v^{i_{j-1}}) + 1, |s^{i_j}|]$ ▷ Here $\circ$ is the concatenation operation.
end for
return t

Note that $|\tilde{s}^i| = O(|s^{i+1}|+\cdots+|s^n|) \le O(|s^1|+\cdots+|s^{i-1}|+|s^{i+1}|+\cdots+|s^n|) = O(L-|s^i|)$. Therefore, according to the Cauchy–Bunyakovsky–Schwarz inequality we have the following complexity:

$$= O\left(\sqrt{L-|s^i|}\log L\log|s^i| + \sqrt{|s^i|}\log^2|s^i|\right) = O\left((\sqrt{L-|s^i|}+\sqrt{|s^i|})\log L\log|s^i|\right)$$

$$= O\left(\sqrt{2\cdot(L-|s^i|+|s^i|)}\log L\log|s^i|\right) = O\left(\sqrt{L}\log L\log|s^i|\right)$$

Each invocation of IsSubstring$(s^i, \tilde{s}^i)$ has constant error probability and it can be accumulated during n invocations. That is why we repeat each invocation $O(\log n)$ times to obtain constant total error probability, for example at most 0.1 total error probability can be achieved.

So, the total query complexity of the procedure is

$$O\left(\sum_{i=1}^{n}\sqrt{L}\log L\log|s^i|\log n\right) = O\left(n\sqrt{L}\log L\log\left(\max_{i=\{1,\ldots,n\}}|s^i|\right)\log n\right) =$$

$$O(n\sqrt{L}\log^2 L\log n) = \tilde{O}(n\sqrt{L}).$$

Here $\tilde{O}$ hides not only constants but logarithms of L and n also. □

Lemma 4 *The* ConstructTheGraph(S) *procedure constructs the graph with* $O(L+n^{1.5}\sqrt{L})$ *query complexity and the error probability at most* 0.1.

Proof: Computing prefix and suffix hashes for a string s^i have $O(|s^i|)$ query complexity. So, computing them for all strings has $O(|s^1|+\cdots+|s^n|) = O(L)$ query complexity.

Complexity of computing overlaps for a fixed s^i is at most $O(n\sqrt{|s^i|})$. Therefore, the total complexity of computing all overlaps for all strings is

$$O(n\sqrt{|s^1|}+\cdots+n\sqrt{|s^n|}) = O(n(\sqrt{|s^1|}+\cdots+\sqrt{|s^n|})) = O(n\sqrt{n(|s^1|+\cdots+|s^n|)}) = O(n^{1.5}\sqrt{L}).$$

So the total complexity is $O(L+n^{1.5}\sqrt{L})$.

Each FirstOneSearch invocation has an error probability. Therefore, the total error probability can be close to 1. At the same time, the algorithm is a sequence of First One Search algorithms that can be converted to an algorithm with constant error probability at most 0.05 without affecting query complexity using [52, 56] technique.

For a fixed s^i, the number of hash comparisons is at most $O(n|s^i|)$ that is at most n different suffixes for each prefix of s^i. Therefore, the total number of different pairs of numbers that are compared is $O(n|s^1|+\cdots+n|s^n|) = O(nL)$. Hence, if we randomly choose a prime p among the first $20nL$, then we can archive the error probability at most 0.05 of all computations. So, the total error probability is at most 0.1. □

Let us discuss the main part of the algorithm. We consider a function $L : 2^V \times V \times V \to \mathbb{R}$ where 2^V is the set of all subsets of V. The function L is such that $L(Y, v, u)$ is the maximum of all weights of paths that visit all nodes from Y exactly once, start from the node v, and finish in the node u. If there is no such path, then we assume that $L(Y, v, u) = -\infty$.

Let the function $F : 2^V \times V \times V \to V^*$ be such that $F(Y, v, u)$ is the path that visits all nodes of Y exactly once, starts from the node v, finishes in the node u and has the maximal weight. In other words, for $P = F(Y, u, v)$ we have $w(P) = L(Y, u, v)$. We assume, that $L(\{v\}, v, v) = 0$ and $F(\{v\}, v, v) = (v)$ for any $v \in V$ by definition.

Let us discuss properties of the function.

Property 1 *Suppose $Y \subseteq V, v, u \in Y$, an integer $k < |Y|$. The function L is such that*

$$L(Y, v, u) = \max_{Y' \in \{Y' : Y' \subset Y, |Y'| = k, v \in Y', u \notin Y'\}} \left\{ \max_{y \in Y'} \{L(Y', v, y) + L((Y \backslash Y') \cup \{y\}, y, u)\} \right\}$$

and $F(Y, u, v)$ is the path that is concatenation of corresponding paths.

Proof: Let us fix a set Y' such that $Y' \subset Y$, $|Y'| = k$. Let $P^1(Y') = F(Y', v, y_{max}(Y'))$ and $P^2(Y') = F((Y \backslash Y') \cup \{y\}, y_{max}(Y'), u)$, where $y_{max}(Y')$ is the target argument for the inner maximum.

The path $P(Y') = P^1(Y') \circ P^2(Y')$ belongs to Y, starts from v and finishes in u, where $\circ$ means concatenation of paths excluding the duplication of common node $y_{max}(Y')$.

Let us consider all paths such that they visit all elements from Y', then other elements of Y. In other words, paths $T = (v^{i_1}, \ldots, v^{i_\ell})$ such that $\{v^{i_1}, \ldots, v^{i_k}\} = Y'$, and $\{v^{i_{k+1}}, \ldots, v^{i_\ell}\} = Y \backslash Y'$. So, $v^{i_k} \in Y'$, and $\{v^{i_k}, \ldots, v^{i_\ell}\} = Y \backslash Y' \cup \{v^{i_k}\}$. Therefore, due to selecting $y_{max}(Y')$ as a target element for maximum, we can be sure that $w(P(Y')) \geq w(T)$.

Let $P = P(Y'_{max})$ such that we reach the outer maximum on Y'_{max}. It belongs to Y, starts from v, and finishes in u. Therefore, $w(P) \leq L(Y, v, u)$.

Assume that there is a path $T = (u^{i_1}, \ldots, u^{i_\ell})$ such that $w(T) = L(Y, v, u)$ and $w(T) > w(P)$. Let us select $Y'' = \{u^{i_1}, \ldots, u^{i_k}\}$. So, it is such that $Y'' \subset Y$ and $|Y''| = k$. Due to the above discussion $T' = P(Y'')$. Therefore, $w(P(Y'')) > w(P(Y'_{max}))$ contradicts the definition of $P(Y'_{max})$ as a path where we reach the outer maximum. □

As a corollary, we obtain the following result. Note that each pair of edges is connected.

Corollary 1 *Suppose $Y \subset V$, $v, u \in Y$. The function L is such that*

$$L(Y, v, u) = \max_{y \in Y \backslash \{u\}} \left(L(Y \backslash \{u\}, v, y) + w(y, u) \right).$$

and $F(Y, u, v)$ is the corresponding path.

Using this idea, we construct the following algorithm.

Step 1. Let $\alpha = 0.055$. We classically compute $L(S, v, u)$ and $F(S, v, u)$ for all $Y \subset V$ such that $|Y| \leq (1 - \alpha)\frac{n}{4}$ and $v, u \in Y$

Step 2. Let $V_4 \subset V$ be such that $|V_4| = \frac{n}{4}$. Then, we have

$$L(V_4, u, v) = \max_{V_\alpha \in \{V_\alpha : V_\alpha \subset V_4, |V_\alpha| = (1-\alpha)\frac{n}{4}\}, y \in V_\alpha} \left(L(V_\alpha, v, y) + L((V_4 \backslash V_\alpha) \cup \{y\}, y, u) \right).$$

Let $V_2 \subset V$ be such that $|V_2| = \frac{n}{2}$. Then, we have

$$L(V_2, u, v) = \max_{V_4 \in \{V_4 : V_4 \subset V_2, |V_4| = \frac{n}{4}\}, y \in V_4} \left(L(V_4, v, y) + L((V_2 \backslash V_4) \cup \{y\}, y, u) \right).$$

Finally,

$$L(V,u,v) = \max_{V_2 \in \{V_2 : V_2 \subset V, |V_2| = \frac{n}{2}\}, y \in V_2} \left(L(V_2, v, y) + L((V \backslash V_2) \cup \{y\}, y, u) \right).$$

We can compute $L(V,u,v)$ and corresponding $F(V,u,v)$ using three nested procedures for maximum finding. As such procedure, we use Dürr-Høyer [25, 26] quantum minimum finding algorithm. The maximal weight of paths $MaxWeight$ and the corresponding path can be computed as a maximum of $L(V,u,v)$ among all $u, v \in V$ as presented in the next statement.

$$MaxWeight = \max_{u,v \in V} L(V,v,u).$$

Let us discuss the implementation of Step 1. It is presented as a recursive function GetL(Y,v,u) for $Y \subset V, u, v \in V$ with cashing that is Dynamic Programming approach in fact. The function is based on Corollary 1.

Algorithm 4 GetL(Y,v,u).

```
if v = u and Y = {v} then                                   ▷ Initialization
    L({v}, v, v) ← 0
    F({v}, v, v) ← (v)
end if
if L(Y, v, u) is not computed then
    weight ← −∞
    path ← ()
    for y ∈ Y\{u, v} do
        if GetL(Y\{u}, v, y) + w(y, u) > weight then
            weight ← L(Y\{u}, v, y) + w(y, u)
            path ← F(Y\{u}, v, y) ∪ u
        end if
    end for
    L(Y, v, u) ← weight
    F(Y, v, u) ← path
end if
return L(Y, v, u)
```

Algorithm 5 Step1.

```
for Y ∈ 2^V such that |Y| ≤ (1 − α)n/4 do
    for v ∈ Y do
        for u ∈ Y do
            GetL(Y, v, u)        ▷ We are computing L(Y, v, u) and F(Y, v, u) for Step 2.
        end for
    end for
end for
```

Let QMax$((x_1, \ldots, x_N))$ be the implementation of the quantum maximum finding algorithm [25, 26] for a sequence $(x_1, \ldots, x_N)$. The most nested quantum maximum finding

algorithm for some $V_4 \subset V, |V_4| = \frac{n}{4}$ and $u, v \in V_4$ is

$$\text{QMax}((L(V_\alpha, v, y) + L((V_4 \backslash V_\alpha) \cup \{y\}, y, u) : V_\alpha \subset V_4, |V_\alpha| = (1-\alpha)\frac{n}{4}, y \in V_\alpha)).$$

The second quantum maximum finding algorithm for some $V_2 \subset V, |V_2| = \frac{n}{2}$ and $u, v \in V_2$ is

$$\text{QMax}((L(V_4, v, y) + L((V_2 \backslash V_4) \cup \{y\}, y, u) : V_4 \subset V_2, |V_4| = n/4, y \in V_4)).$$

Note that $|V_4| = n/4$ and $|V_2 \backslash V_4| = n/4$. We use the invocation of QMax (the most nested quantum maximum finding algorithm) instead of $L(V_4, v, y)$ and $L(V_2 \backslash V_4, y, u)$.

The third quantum maximum finding algorithm for some $u, v \in V$ is

$$\text{QMax}((L(V_2, v, y) + L((V \backslash V_2) \cup \{y\}, y, u) : V_2 \subset V, |V_2| = n/2, y \in V_2))$$

Note that $|V_2| = n/2$ and $|V \backslash V_2| = n/2$. We use the invocation of QMax (the second quantum maximum finding algorithm) instead of $L(V_2, v, y)$ and $L((V \backslash V_2) \cup \{y\}, y, u)$.

The fourth quantum maximum finding algorithm among all $u, v \in V$ is

$$\text{QMax}(L(V, v, u) : v, u \in V)$$

The procedure QMax returns not only the maximal value but the index of the target element. Therefore, by the "index" we can obtain the target paths using the F function. So, the resulting path is $P = P^1 \circ P^2$, where P^1 is the result path for $L(V_2, v, y)$ and P^2 is the result path for $L((V \backslash V_2) \cup \{y\}, y, u)$.

$P^1 = P^{1,1} \circ P^{1,2}$, where $P^{1,1}$ is the result path for $L(V_4, v, y)$ and $P^{1,2}$ is the result path for $L((V_2 \backslash V_4) \cup \{y\}, y, u)$. In the same way, we can construct $P^2 = P^{2,1} \circ P^{2,2}$.

$P^{1,1} = P^{1,1,1} \circ P^{1,1,2}$, where $P^{1,1,1}$ is the result path for $L(V_\alpha, v, y)$ and $P^{1,1,2}$ is the result path for $L((V_4 \backslash V_\alpha) \cup \{y\}, y, u)$. Note, that these values were precomputed classically in Step 1, and were stored in $F(V_\alpha, v, y)$ and $F((V_4 \backslash V_\alpha) \cup \{y\}, y, u)$ respectively.

In the same way, we can construct

$$P^{1,2} = P^{1,2,1} \circ P^{1,2,2}, \qquad P^{2,1} = P^{2,1,1} \circ P^{2,1,2}, \qquad P^{2,2} = P^{2,2,1} \circ P^{2,2,2}.$$

The final path is

$$P = P^1 \circ P^2 = (P^{1,1} \circ P^{1,2}) \circ (P^{2,1} \circ P^{2,2}) =$$

$$\Big((P^{1,1,1} \circ P^{1,1,2}) \circ (P^{1,2,1} \circ P^{1,2,2})\Big) \circ \Big((P^{2,1,1} \circ P^{2,1,2}) \circ (P^{2,2,1} \circ P^{2,2,2})\Big)$$

Note the Durr-Hoyer algorithm QMax has an error probability at most 0.1 and is based on the Grover search algorithm. So, because of several nested QMax procedures, we should use the bounded-error input version of the Grover search algorithm that was discussed in [33, 12, 15].

Let us present the final algorithm as Algorithm 6. The complexity of the algorithm is presented in Theorem 4.

Theorem 1 *Algorithm 6 solves* $\texttt{SCS}(S)$ *with query complexity*

$$O\left(n^3 1.728^n + L + n^{1.5}\sqrt{L} + n\sqrt{L}\log^2 L \log n\right) = \tilde{O}\left(n^3 1.728^n + L + n^{1.5}\sqrt{L}\right)$$

and the error probability at most $1/3$.

Algorithm 6 Algorithm for $\texttt{SCS}(S)$.

RemovingDublicatesAndSubstrings(S)
$(V, E) \leftarrow$ ConstructTheGraph(S)
Step1()
$weight, path \leftarrow$ QMax($L(V, v, u) : v, u \in V$)
$t \leftarrow$ ConstructSuperstringByPath($path$)
return t

Proof: The correctness of the algorithm follows from the above discussion. Let us present an analysis of query complexity. The complexity of removing all duplicates and substrings in S by the procedure RemovingDublicatesAndSubstrings(S) is $O(n\sqrt{L}\log^2 L \log n)$ and the error probability is at most 0.1 due to Lemma 3. The complexity of constructing the graph is $O(L + n^{1.5}\sqrt{L})$ and the error probability is at most 0.1 due to Lemma 4.

We continue with complexity of Step 1 (Classical preprocessing). Here we check all subsets of size at most $(1-\alpha)\frac{n}{4}$, starting and ending nodes, and neighbor nodes. The query complexity is

$$\sum_{i=1}^{(1-\alpha)\frac{n}{4}} O\left(\binom{n}{i} n^3\right) = O(n^3 1.728^n).$$

Complexity of Step 2 (Quantum part) is the complexity of four nested Durr-Hoyer maximum finding algorithms. Due to [25, 30, 26], the complexity for the most nested QMax is

$$Q_1 = O\left(\sqrt{\binom{n/4}{\alpha n/4}} \cdot \sqrt{n}\right)$$

because the searching space size is $\binom{n/4}{\alpha n/4} \cdot n$ and the query complexity of extracting the subset form its number is $O(n)$. These two operations are invoked sequentially. The complexity for the second QMax is

$$Q_2 = \left(\sqrt{\binom{n/2}{n/4}} \cdot \sqrt{n} \cdot Q_1\right) = O\left(\sqrt{\binom{n/2}{n/4}\binom{n/4}{\alpha n/4}} \cdot n\right)$$

because of the similar reasons. The complexity for the third QMax is

$$Q_3 = O\left(\sqrt{\binom{n}{n/2}} \cdot \sqrt{n} \cdot Q_2\right) = O\left(\sqrt{\binom{n}{n/2}\binom{n/2}{n/4}\binom{n/4}{\alpha n/4}} \cdot n^{1.5}\right).$$

The complexity for the fourth (that is the final) QMax is

$$O\left(\sqrt{n^2} \cdot Q_3\right) = O\left(\sqrt{\binom{n}{n/2}\binom{n/2}{n/4}\binom{n/4}{\alpha n/4}} \cdot n^{2.5}\right)$$

because the searching space size is n^2.

So, the total complexity of Step 2 is

$$O\left(\sqrt{\binom{n}{n/2}\binom{n/2}{n/4}\binom{n/4}{\alpha n/4}} \cdot n^{2.5}\right) = O(n^{2.5} 1.728^n).$$

The complexity of ConstructSuperstringByPath is $O(L)$.

We invoke RemovingDublicatesAndSubstrings, ConstructTheGraph, Step 1, Step 2 and ConstructSuperstringByPath sequentially. Therefore, the total complexity is the sum of complexities for these steps. So, the total complexity is

$$O\left(n\sqrt{L}\log^2 L\log n + L + n^{1.5}\sqrt{L} + n^3 1.728^n + n^{2.5}1.728^n + L\right)$$

$$= O\left(n^3 1.728^n + L + n^{1.5}\sqrt{L} + n\sqrt{L}\log^2 L\log n\right) = \tilde{O}\left(n^3 1.728^n + L + n^{1.5}\sqrt{L}\right).$$

Step 2, RemovingDublicatesAndSubstrings and ConstructTheGraph have error probability. Each of them has at most constant error probability. Using repetition, we can achieve at most 0.1 error probability for each of the procedures and at most 0.3 for the whole algorithm. □

Constructing Graph in the Case of Random Strings

In this Section, we discuss an alternative implementation of ConstructTheGraph procedure that is ConstructTheGraph2(S) that constructs the graph $G = (V, E)$ by S. For the algorithm we need a quantum procedure AllOnesSearch$(i, \mathcal{I}, r)$ that

- accepts an index of a string $i \in \{1, \ldots, n\}$, a set of indexes of strings $\mathcal{I} \subset \{1, \ldots, n\}$, and an index of a symbol from a string $r \in \{1, \ldots, |s^i|\}$.
- returns a set of indexes $\mathcal{I}' \subset \mathcal{I}$ such that for any $j \in \mathcal{I}'$ we have $s^i[n - r + 1] = s^j[r]$.

The function is based on Grover's search algorithm [30, 22] and has $O\left(\sqrt{|\mathcal{I}| \cdot |\mathcal{I}'|}\right)$ complexity. The procedure and analysis are presented in [52].

The main idea of the algorithm is the following.

- **Step 1**. Initially, we assign $w(v^i, v^j) \leftarrow 0$ for each $i, j \in \{1, \ldots, n\}$.
- **Step 2**. We consider all strings s^i, for $i \in \{1, \ldots, n\}$. For each string s^i we start from $\mathcal{I} \leftarrow \{1, \ldots, n\} \backslash \{i\}$, and $r \leftarrow 1$. We do Step 3. until $\mathcal{I} = \emptyset$.
- **Step 3**. For $i \in \{1, \ldots, n\}$, $\mathcal{I} \subset \{1, \ldots, n\}$, and $r \in \{1, \ldots, |s^i|\}$ we invoke $\mathcal{I}' \leftarrow$ AllOnesSearch$(i, \mathcal{I}, r)$. Then, we update $w(v^i, v^j) \leftarrow w(v^i, v^j) + 1$ for each $j \in \mathcal{I}'$. After that, if $\mathcal{I}' \neq \emptyset$ we repeat Step 3 for $\mathcal{I} \leftarrow \mathcal{I}'$, and $r \leftarrow r + 1$. If $\mathcal{I}' = \emptyset$ then we stop the process for the current s^i.

The implementation of the procedure is presented in Algorithm 7. The complexity of the procedure is discussed in Lemma 5.

The procedure has good complexity in the case of s^i are random.

Lemma 5 *Assume that the alphabet Σ has order $|\Sigma| = C$ and all strings s^i are random. The* ConstructTheGraph2(S) *procedure constructs the graph with $O(n^2 \cdot \frac{C^{1/2}}{C-1})$ query complexity on average and the error probability at most* 0.1.

Algorithm 7 Implementation of ConstructTheGraph2(S) for $S = (s^1, \ldots, s^n)$.

```
V = (v^1, ..., v^n)
for i ∈ {1, ..., n} do
    for j ∈ {1, ..., n} do
        if i ≠ j then
            w(v^i, v^j) ← 0
        end if
    end for
end for
for i ∈ {1, ..., n} do
    I ← {1, ..., n}\{i}
    r ← 1
    I ← AllOnesSearch(i, I, r)
    while I' ≠ ∅ do
        for j ∈ I do
            w(v^i, v^j) ← w(v^i, v^j) + 1
        end for
        I ← AllOnesSearch(i, I, r)
        r ← r + 1
    end while
end for
return (V, E)
```

Proof: We use the All Ones Search Problem to search all $k \in \mathcal{I}$ such that $s^i[n-r+1] = s^k[r]$ and the complexity of it is $O\left(\sqrt{|\mathcal{I}| \cdot |\mathcal{I}'|}\right)$, where $|\mathcal{I}|, |\mathcal{I}'| \leq n-1$. In this case, we have $m = |\mathcal{I}|$ with. Since the alphabet Σ has order C and all strings s^i are random, the output of AllOnesSearch$(i, \mathcal{I}, r)$ has order $\frac{1}{C} \cdot |\mathcal{I}|$ on average. Therefore, the total complexity of constructing the graph is

$$n \cdot \sum_{k=1}^{L} O\left(\sqrt{\frac{1}{C} \cdot \frac{n-1}{C^{k-1}} \cdot \frac{n-1}{C^{k-1}}}\right) = O\left(n^2 \cdot \frac{C^{1/2}}{C-1}\right)$$

□

The complexity of the whole algorithm in that case is presented in the next Corollary

Corollary 2 *Assume that the alphabet Σ has order $|\Sigma| = C$ and all strings s^i are random. Algorithm 6 with this assumption solves $\mathtt{SCS}(S)$ with*

$$O\left(n^3 1.728^n + n^2 \cdot \frac{C^{1/2}}{C-1} + L + n\sqrt{L}\log^2 L \log n\right) = \tilde{O}\left(n^3 1.728^n + n^2 \cdot \frac{C^{1/2}}{C-1} + L + n\sqrt{L}\right)$$

query complexity in avarage and error probability at most $1/3$, where C. In the case of $C = const$, the query complexity is

$$O\left(n^3 1.728^n + n\sqrt{L}\log^2 L \log n + L\right) = \tilde{O}\left(n^3 1.728^n + L + n\sqrt{L}\right).$$

Proof: Due to Lemma 5, the complexity of graph constructing is $O(n^2 \cdot \frac{C^{1/2}}{C-1})$. The rest part is the same as in Theorem 4. So, we obtain the required complexity. The second claim follows

from $O(n^2 \cdot \frac{C^{1/2}}{C-1}) = O(n^2) = O(n^3 1.728^n)$ if $C = const$. □

5 The Text Assembling Problem

The algorithm is a modification of the algorithm from [49]. Our algorithm has better complexity compared to the existing one. Here we present an almost full description of the algorithm for completeness of the presentation.

In this section, we use a quantum subroutine for comparing two strings u and v in the lexicographical order. Let us denote it QCompare(u, v). It is based on the First One search algorithm [26, 52, 58, 59, 35] that is a modification of Grover's search algorithm [30, 22]. The procedure in different forms was discussed in several papers [18, 38, 9, 43, 35, 51, 54]. The main property of the subroutine is presented in the following lemma.

Lemma 6 ([43]) *The quantum algorithm* QCompare(u, v) *compares strings u and v in the lexicographical order with $O(\sqrt{\min(|u|, |v|)})$ query complexity and the error probability is at most* 0.1.

Let us present a quantum algorithm for the $\texttt{TAO}(t, S)$ problem. Let $long_i$ be an index of the longest string from S that starts in the position i of the string t, where $1 \leq i \leq m$. Formally, $long_i = j$ if s^j is the longest string from S such that $t[i, i + |s^j| - 1] = s^j$. Let $long_i = -1$ if there is no such string s^j. If we construct the array $long = (long_1, \ldots, long_m)$, then we can construct $Q = (q_1, \ldots, q_r)$ and $I = (i_1, \ldots, i_r)$ with $O(m)$ query complexity. Note that Q and I arrays are a solution to the problem $\texttt{TAO}(t, S)$. A procedure ConstructQI$(long)$ that constructs Q and I by $long$ is presented in Appendix B. If there is no a (Q, I) decomposition of t, then the procedure returns $NULL$. Let us discuss how to construct the array $long$.

Step 1. We start from constructing a suffix array suf by the string t. After that, we present an array $a = (a_1, \ldots, a_m)$ such that $a_i = (len_i, ind_i)$, and a_i corresponds to suf_i. We compute values len_i and ind_i on the next steps. Here len_i is the length of the longest string s^j that is a prefix of the suffix $t[suf_i, n]$ and ind_i is its index. Before processing all strings we initialize the values a_i by $(0, -1)$ that are neutral values for our future operations.

Step 2. We construct a segment tree st for the array a such that a node of the tree stores the maximum of len_i for i belonging to the node's segment.

Step 3. We process s^j, for each $j \in \{1, \ldots, n\}$. We compute the minimal index low and the maximal index $high$ such that each suffix $t[suf_i, n]$ has s^j as a prefix for $low \leq i \leq high$. Suffixes in the suffix array are sorted, therefore, all target suffixes are situated sequentially. Hence, we can use the Binary search algorithm for computing low and $high$. The QCompare subroutine from Lemma 6 is used as a string comparator. We present the implementation of the step in Appendix C as SearchSegment(s^j) subroutine. It returns the pair $(low, high)$, or $(NULL, NULL)$ if no suffix of t contains s^j as a prefix.

Step 4. We update nodes of the segment tree that correspond to elements of a in range $[low, high]$ by a pair $(|s^j|, j)$.

We repeat Steps 3 and 4 for each string from S. After that, we finish the construction of a by application of push operation for the segment tree.

Step 5. We can construct $long$ by a and suf. If $a_i = (len_i, ind_i)$, then we assign $long_{suf_i} \leftarrow ind_i$. We can do it because of the definitions of $long$, suf_i, len_i, and ind_i.

The whole algorithm is presented as Algorithm 8, and its complexity is discussed in Theorem 2.

Algorithm 8 The quantum algorithm for the text t constructing from a dictionary S problem

```
suf ← ConstructSuffixArray(t)
a ← [(0, −1), . . . , (0, −1)]                    ▷ Initialization by 0-array
st ← ConstructSegmentTree(a)
for j ∈ {1, . . . , m} do
    (low, high) ← SearchSegment(s^j)
    Update(st, low, high, (|s^j|, j))
end for
Push(st)
for i ∈ {1, . . . , n} do
    (len, ind) ← Request(st, i)
    long_{suf_i} ← ind
end for
(Q, I) ← ConstructQI(long)
return (Q, I)
```

Theorem 2 *Algorithm 8 solves* $\texttt{TAO}(t, S)$ *problem with* $O\left(m + \log m \cdot \sqrt{nL}\right)$ *query complexity and the error probability at most* $1/3$.

Proof: The correctness of the algorithm follows from the construction. Due to results from Section 3, the query complexity for each of procedures ConstructSuffixArray, ConstructSegmentTree, and Push are $O(m)$. Construction of the array *long*, and construction of ConstructQI have $O(m)$ query complexity because each of them contains just one linear loop.

Due to Lemma 6, the query complexity of QCompare for s^j is $O(\sqrt{|s^j|})$. The procedure QSearchSegment invokes QCompare procedure $O(\log m)$ times for each string $s^1, \ldots, s^n$. So, the complexity of processing all strings from S is $O\left(\log m \cdot \sum_{j=1}^{n} \sqrt{|s^j|}\right)$.

Let us use the Cauchy-Bunyakovsky-Schwarz inequality and $L = \sum_{j=1}^{n} |s^j|$ equality for simplifying the statement.

$$\leq O\left(\log m \cdot \sqrt{n \sum_{j=1}^{n} |s^j|}\right) = O\left(\log m \cdot \sqrt{n \cdot L}\right).$$

The total query complexity is

$$O\left(m + m + m + m + m + \log m \cdot \sqrt{nL}\right) = O\left(m + \log m \cdot \sqrt{nL}\right)$$

Let us discuss the error probability. Only the QCompare subroutine can have an error. We invoke it several times. The subroutine's error probability is at most 0.1. The error can accumulate and reach a probability close to 1. At the same time, QCompare is the First One Search algorithm for a specific search function. Due to [52, 56], we can modify the sequential invocations of the First One Search algorithm to an algorithm that has the error probability at most $1/3$ without complexity changing. □

6 Conclusion

We present a quantum algorithm for the SSP or SCS problem. It works faster than existing classical algorithms. At the same time, there are faster classical algorithms in the case of restricted length of strings [28, 29]. It is interesting to explore quantum algorithms for such a restriction. Can quantum algorithms be better than classical counterparts in this case? Another open question is approximating algorithms for the problem. As we mentioned before, such algorithms are more useful in practice. So, it is interesting to investigate quantum algorithms that can be applied to practical cases.

In the case of the Text Assemble problem, upper and lower bounds are far apart. It is interesting to find a better quantum lower bound or improve the upper bound.

For both problems, an open question is developing a quantum algorithm for the case with possible typos.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 61877054, 12031004,12271474, and the Foreign Experts in Culture and Education Foundation under Grant No. DL2022147005L. Kamil Khadiev thanks these projects for supporting his visit. Kamil Khadiev has been has been supported by the Kazan Federal University Strategic Academic Leadership Program (PRIORITY-2030).

References

1. F. Ablayev, M. Ablayev, J. Z. Huang, K. Khadiev, N. Salikhova, and D. Wu. On quantum methods for machine learning problems part i: Quantum tools. *Big Data Mining and Analytics*, 3(1):41–55, 2019.
2. F. Ablayev, M. Ablayev, K. Khadiev, and A. Vasiliev. Classical and quantum computations with restricted memory. *LNCS*, 11011:129–155, 2018.
3. F. Ablayev and A. Gainutdinova. Complexity of quantum uniform and nonuniform automata. In *Developments in Language Theory*, volume 3572 of *LNCS*, pages 78–87. Springer, 2005.
4. F. Ablayev, A. Gainutdinova, K. Khadiev, and A. Yakaryılmaz. Very narrow quantum OBDDs and width hierarchies for classical OBDDs. *Lobachevskii Journal of Mathematics*, 37(6):670–682, 2016.
5. Farid Ablayev, Marat Ablayev, Kamil Khadiev, Nailya Salihova, and Alexander Vasiliev. Quantum algorithms for string processing. In *Mesh Methods for Boundary-Value Problems and Applications*, volume 141 of *Lecture Notes in Computational Science and Engineering*, 2022.
6. Farid Ablayev, Marat Ablayev, Alexander Vasiliev, and Mansur Ziatdinov. Quantum fingerprinting and quantum hashing. computational and cryptographical aspects. *Baltic Journal of Modern Computing*, 4(4):860, 2016.
7. Farid Ablayev and Alexander Vasiliev. On the computation of boolean functions by quantum branching programs via fingerprinting. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 15, 2008.
8. Farid M Ablayev and Alexander Vasiliev. Algorithms for quantum branching programs based on fingerprinting. *Int. J. Software and Informatics*, 7(4):485–500, 2013.
9. Jeffrey A Aborot. An oracle design for grovers quantum search algorithm for solving the exact string matching problem. In *Theory and Practice of Computation: Proceedings of Workshop on Computation: Theory and Practice WCTP2017*, pages 36–48. World Scientific, 2019.
10. Shyan Akmal and Ce Jin. Near-optimal quantum algorithms for string problems. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2791–2832.

SIAM, 2022.
11. A. Ambainis. Understanding quantum algorithms via query complexity. In *Proc. Int. Conf. of Math. 2018*, volume 4, pages 3283–3304, 2018.
12. A. Ambainis, Kaspars Balodis, Jnis Iraids, Kamil Khadiev, Vladislavs Kevickis, Krišjnis Prsis, Yixin Shen, Juris Smotrovs, and Jevgnijs Vihrovs. Quantum Lower and Upper Bounds for 2D-Grid and Dyck Language. In *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, volume 170 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:14, 2020.
13. A. Ambainis and R. Freivalds. 1-way quantum finite automata: strengths, weaknesses and generalizations. In *FOCS'98*, pages 332–341. IEEE, 1998.
14. A. Ambainis and N. Nahimovs. Improved constructions of quantum automata. *Theoretical Computer Science*, 410(20):1916–1922, 2009.
15. Andris Ambainis, Kaspars Balodis, Jānis Iraids, Kamil Khadiev, Vladislavs Kļevickis, Krišjānis Prūsis, Yixin Shen, Juris Smotrovs, and Jevgēnijs Vihrovs. Quantum bounds for 2d-grid and dyck language. *Quantum Information Processing*, 22(5):194, 2023.
16. Andris Ambainis, Kaspars Balodis, Jānis Iraids, Martins Kokainis, Krišjānis Prūsis, and Jevgēnijs Vihrovs. Quantum speedups for exponential-time dynamic programming algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1783–1793. SIAM, 2019.
17. S Arunachalam and R de Wolf. Optimizing the number of gates in quantum search. *Quantum Information and Computation*, 17(3&4):251–261, 2017.
18. Hafiz Md Hasan Babu, Lafifa Jamal, Sayanton Vhaduri Dibbo, and Ashis Kumer Biswas. Area and delay efficient design of a quantum bit string comparator. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 51–56. IEEE, 2017.
19. Shakuntala Baichoo and Christos A Ouzounis. Computational complexity of algorithms for sequence comparison, short-read assembly and genome alignment. *Biosystems*, 156:72–85, 2017.
20. Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.
21. Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, Michael T. Hallett, and Harold T. Wareham. Parameterized complexity analysis in computational biology. *Bioinformatics*, 11(1):49–57, 1995.
22. Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(4-5):493–505, 1998.
23. T. H Cormen, C. E Leiserson, R. L Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.
24. Ronald de Wolf. *Quantum computing and communication complexity*. University of Amsterdam, 2001.
25. C. Dürr and P. Høyer. A quantum algorithm for finding the minimum. *arXiv:quant-ph/9607014*, 1996.
26. Christoph Dürr, Mark Heiligman, Peter Høyer, and Mehdi Mhalla. Quantum query complexity of some graph problems. *SIAM Journal on Computing*, 35(6):1310–1328, 2006.
27. Rūsiņš Freivalds. Fast probabilistic algorithms. In *Mathematical Foundations of Computer Science 1979*, volume 74 of *LNCS*, pages 57–69, 1979.
28. Alexander Golovnev, Alexander S Kulikov, and Ivan Mihajlin. Solving 3-superstring in 3 n/3 time. In *International Symposium on Mathematical Foundations of Computer Science*, pages 480–491. Springer, 2013.
29. Alexander Golovnev, Alexander S Kulikov, and Ivan Mihajlin. Solving scs for bounded length strings in fewer than 2n steps. *Information Processing Letters*, 114(8):421–425, 2014.
30. Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.
31. Lov K Grover. Trade-offs in the quantum search algorithm. *Physical Review A*, 66(5):052314, 2002.

32. Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962.
33. Peter Høyer, Michele Mosca, and Ronald de Wolf. Quantum search on bounded-error inputs. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming*, pages 291–299, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
34. Stephen Jordan. Quantum algorithms zoo, 2021. http://quantumalgorithmzoo.org/.
35. Ruslan Kapralov, Kamil Khadiev, Joshua Mokut, Yixin Shen, and Maxim Yagafarov. Fast classical and quantum algorithms for online k-server problem on trees. *CEUR Workshop Proceedings*, 3072:287–301, 2022.
36. Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.
37. Marek Karpinski and Richard Schmied. Improved inapproximability results for the shortest superstring and related problems. In *Proceedings of the Nineteenth Computing: The Australasian Theory Symposium-Volume 141*, pages 27–36, 2013.
38. K. Khadiev and A. Ilikaev. Quantum algorithms for the most frequently string search, intersection of two string sequences and sorting of strings problems. In *International Conference on Theory and Practice of Natural Computing*, pages 234–245, 2019.
39. K. Khadiev and A. Khadieva. Reordering method and hierarchies for quantum and classical ordered binary decision diagrams. In *CSR 2017*, volume 10304 of *LNCS*, pages 162–175. Springer, 2017.
40. K. Khadiev and A. Khadieva. Quantum online streaming algorithms with logarithmic memory. *International Journal of Theoretical Physics*, 60:608–616, 2021.
41. Kamil Khadiev. Lecture notes on quantum algorithms. *arXiv preprint arXiv:2212.14205*, 2022.
42. Kamil Khadiev and Syumbel Enikeeva. Quantum version of self-balanced binary search tree with strings as keys and applications. In *International Conference on Micro- and Nano-Electronics 2021*, volume 12157, pages 587 – 594. International Society for Optics and Photonics, SPIE, 2022.
43. Kamil Khadiev, Artem Ilikaev, and Jevgenijs Vihrovs. Quantum algorithms for some strings problems based on quantum string comparator. *Mathematics*, 10(3):377, 2022.
44. Kamil Khadiev and Aliya Khadieva. Quantum and classical log-bounded automata for the online disjointness problem. *Mathematics*, 10(1), 2022.
45. Kamil Khadiev, Aliya Khadieva, and Alexander Knop. Exponential separation between quantum and classical ordered binary decision diagrams, reordering method and hierarchies. *Natural Computing*, pages 1–14, 2022.
46. Kamil Khadiev and Dmitry Kravchenko. Quantum algorithm for dyck language with multiple types of brackets. In *Unconventional Computation and Natural Computation*, pages 68–83, 2021.
47. Kamil Khadiev and Carlos Manuel Bosch Machado. Quantum algorithm for the shortest superstring problem. In *International Conference on Micro- and Nano-Electronics 2021*, volume 12157, pages 579 – 586. International Society for Optics and Photonics, SPIE, 2022.
48. Kamil Khadiev and Vladislav Remidovskii. Classical and quantum algorithms for assembling a text from a dictionary. *NONLINEAR PHENOMENA IN COMPLEX SYSTEMS*, 24(3):207–221, 2021.
49. Kamil Khadiev and Vladislav Remidovskii. Classical and quantum algorithms for constructing text from dictionary problem. *Natural Computing*, 20(4):713–724, 2021.
50. Kamil Khadiev, Nikita Savelyev, Mansur Ziatdinov, and Denis Melnikov. Noisy tree data structures and quantum applications. *Mathematics*, 11(22):4707, 2023.
51. K.R. Khadiev and D.I. Lin. Quantum online algorithms for a model of the request-answer game with a buffer. *Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, 162(3):367–382, 2020.
52. Robin Kothari. An optimal quantum algorithm for the oracle identification problem. In *31st International Symposium on Theoretical Aspects of Computer Science*, page 482, 2014.
53. Antti Laaksonen. *Guide to Competitive Programming*. Springer, 2017.

54. François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *Algorithmica*, pages 1–36, 2022.
55. François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
56. Troy Lee, Rajat Mittal, Ben W Reichardt, Robert Špalek, and Mario Szegedy. Quantum query complexity of state conversion. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 344–353. IEEE, 2011.
57. Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *String Processing and Information Retrieval*, pages 268–284, Cham, 2018. Springer International Publishing.
58. C. Y.-Y. Lin and H.-H. Lin. Upper bounds on quantum query complexity inspired by the elitzur-vaidman bomb tester. In *30th Conference on Computational Complexity (CCC 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
59. C. Y.-Y. Lin and H.-H. Lin. Upper bounds on quantum query complexity inspired by the elitzur–vaidman bomb tester. *Theory of Computing*, 12(18):1–35, 2016.
60. David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978.
61. Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 90, page 319327. Society for Industrial and Applied Mathematics, 1990.
62. Martin Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125(2):205–228, 1994.
63. Martin Middendorf. Shortest common superstrings and scheduling with coordinated starting times. *Theoretical Computer Science*, 191(1-2):205–214, 1998.
64. Ashley Montanaro. Quantum pattern matching fast on average. *Algorithmica*, 77(1):16–39, 2017.
65. Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.
66. M. A Nielsen and I. L Chuang. *Quantum computation and quantum information*. Cambridge univ. press, 2010.
67. Mihai Pop, Adam Phillippy, Arthur L Delcher, and Steven L Salzberg. Comparative genome assembly. *Briefings in bioinformatics*, 5(3):237–248, 2004.
68. Hariharan Ramesh and V Vinay. String matching in $O(\sqrt{n} + \sqrt{m})$ quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003.
69. Armin Shmilovici and Irad Ben-Gal. Using a vom model for reconstructing potential coding regions in est sequences. *Computational Statistics*, 22(1):49–69, 2007.
70. Virginia Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In *International Symposium on Mathematical Foundations of Computer Science*, pages 793–800. Springer, 2005.
71. Karl V Voelkerding, Shale A Dames, and Jacob D Durtschi. Next-generation sequencing: from basic research to diagnostics. *Clinical chemistry*, 55(4):641–658, 2009.

Appendix A An Implementation of Segment Tree's Operations

Assume that the following elements are associated with each node v of the segment tree:

- $g(v)$ is the target value for a segment tree. If it is not assigned, then $g(v) = -\infty$
- $d(v)$ is an additional value.
- $l(v)$ is the left border of the segment that is associated with the node v.
- $r(v)$ is the right border of the segment that is associated with the node v.
- $LeftC(v)$ is the left child of the node.
- $RightC(v)$ is the right child of the node

If a node v is a leaf, then $LeftC(v) = RightC(v) = NULL$. Let st be associated with the root of the tree.

Let us present Algorithm 9 for the Update operation. It is a recursive procedure.

Algorithm 9 UPDATE(v, i, j, x, y). Update $[i, j]$ segment by (x, y) for a segment tree with a root v

```
if l(v) = i and r(v) = j and g(v) < x then
    g(v) ← x, d(v) ← y
else
    m ← r(LeftC(v))
    if m ≥ j then
        UPDATE(LeftC(v), i, j, x, y)
    else
        if m < i then
            UPDATE(RightC(v), i, j, x, y)
        else
            UPDATE(LeftC(v), i, m, x, y)
            UPDATE(RightC(v), m + 1, j, x, y)
        end if
    end if
end if
```

Let us present the implementation for the Push operation in Algorithm 10 and 11. It is a recursive procedure.

Algorithm 10 PUSH(v). Push operation a segment tree with a root v

```
PUSH_BASE(v, −1, −1)
```

Appendix B The Implementation of ConstructQI($long$) Procedure for TAO(S, t) Problem

Algorithm 12 contains the implementation.

Algorithm 11 PUSH_BASE(v, g, d). Push operation for a segment tree with a root v and assigning a values g and d

```
if v ≠ NULL then
    if g(v) > g then
        g ← g(v), d ← d(v)
    end if
    PUSH_BASE(LeftC(v), g, d)
    PUSH_BASE(RightC(v), g, d)
end if
```

Appendix C The Implementation of SearchSegment(u) Procedure for TAO(S) Problem

Algorithm 13 contains the implementation.

Appendix D The Proof of Lemma 2

Lemma 2 *The path P that visits all the nodes of G exactly once and has the maximal possible weight corresponds to the shortest common superstring t for the sequence S.*

Proof: Firstly, let us show that a path that visits all nodes of G corresponds to a superstring. Assume that we have a path P. We collect a string t by P according to CONSTRUCTSUPERSTRINGBYPATH procedure. So, we add a string corresponding to each node at least once. Therefore, t contains all strings at least once.

Secondly, let us compute the length of the string t collected by $P = (v^{i_1}, \dots, v^{i_\ell})$. The first node from the string adds $|s^{i_1}|$ symbols to t. Each other node v^{i_j} adds $|s^{i_j}| - w(i_{j-1}, i_j)$ symbols. Therefore, the total length is

$$|s^{i_1}| + |s^{i_2}| - w(i_1, i_2) + \dots + |s^{i_\ell}| - w(i_{\ell-1}, i_\ell) = \sum_{j=1}^{\ell} |s^{i_j}| - \sum_{j=2}^{\ell} w(i_{j-1}, i_j) = \sum_{j=1}^{\ell} |s^{i_j}| - w(P).$$

Note that if P visits each node exactly once, then $\sum_{j=1}^{\ell} |s^{i_j}| = \sum_{j=1}^{n} |s^j| = L$, and the length of the string t is $L - w(P)$.

Remind that according to the CONSTRUCTTHEGRAPH procedure, the graph G is a full graph. Let $P' = (v^{i_1}, \dots v^{i_\ell})$ be a path that visits a node v^{i_r} at least twice. In that case, we remove the node v^{i_r} from the path. The new path $P'' = (v^{i_1}, \dots, v^{i_{r-1}}, v^{i_{r+1}}, \dots, v^{i_\ell})$ still valid because the graph is full and any pair of nodes are connected. Let us compare lengths m' and m'' of two superstrings that are collected by P' and P'', respectively.

The connection between $v^{i_{r-1}}$ and $v^{i_{r+1}}$ in P'' gives us a string u'' which a prefix is $s^{i_{r-1}}$ and a suffix is $s^{i_{r+1}}$. The length of u'' is minimal possible because $w(i_{r-1}, i_{r+1})$ is the maximal overlap. The connection between $v^{i_{r-1}}$ and $v^{i_{r+1}}$ via v^{i_r} in P' gives us a string u' which a prefix is $s^{i_{r-1}}$ and a suffix is $s^{i_{r+1}}$. Definitely, $|u'| \geq |u''|$. Therefore, $m' \geq m''$. So, superstrings that are collected by paths visiting each node exactly once are shorter than paths that visit some node at least twice.

Thirdly, let us show that any path P'' that does not visit all nodes and corresponds to a superstring can be extended to a path that visits all nodes such that the new collected superstring is not longer than the original one. Assume that we have a path $P'' = (v^{i_1}, \dots, v^{i_\ell})$

Algorithm 12 ConstructQI($long$). Constructing Q and I from $long$

```
d ← 1
i_d ← long_1
q_d ← 1
left ← 2
right ← |s^{i_1}| + 1
while q_d < n do
    max_i ← left
    max_q ← −1
    if long_{left} > 0 then
        max_q ← left + |s^{long_{left}}| − 1
    end if
    for j ∈ {left + 1, ..., right} do
        if long_j > 0 and j + |s^{long_j}| − 1 > max_q then
            max_i ← j
            max_q ← j + |s^{long_j}| − 1
        end if
    end for
    if max_q = −1 or max_q < right then
        Break the While loop and return NULL    ▷ We cannot construct another part of
the string t
    end if
    d ← d + 1
    i_d ← long_{max_i}
    q_d ← max_i
    left ← right + 1
    right ← max_q + 1
end while
return (Q, I)
```

that does not visit v^r, end the corresponding string t is a superstring. Therefore, t contains s^r as a substring too. So, there is v^{i_j} and v^{i_j+1} such that the string u that is collected from s^{i_j} and s^{i_j+1} by removing their overlapping has s^r as a substring. Note that it cannot be three sequential nodes from the path P'' because otherwise, the middle string should be a substring of s^r. At the same time, it is an impossible situation because we remove all duplicates and substrings in the first step of our algorithm. Assume that s^r starts in u in position j_s and finishes in position j_f. Therefore, $u[1, j_f]$ has s^{i_j} as a prefix and s^r as a suffix. At the same time, $u[j_s, |u|]$ has s^r as a prefix and s^{i_j+1} as a suffix.

Let us update P'' by inserting v^r between v^{i_j} and v_{i_j+1}. $P''' = (v^{i_1}, \ldots, v^{i_j}, v^r, v_{i_j+1}, \ldots, v^{i_\ell})$. Let us look to the string u' that is collected from s^{i_j}, s^r, and s^{i_j+1} according to connection of v^{i_j}, v^r, and v_{i_j+1} in P'''. Assume that s^r starts in u' in position j'_s and finishes in position j'_f. Therefore, $u'[1, j'_f]$ has $s^{i'_j}$ as a prefix and s^r as a suffix; and it is the shortest possible such string. At the same time, $u'[j'_s, |u'|]$ has s^r as a prefix and s^{i_j+1} as a suffix; and it is the shortest possible such string.

Hence, $|u'[1, j'_f]| \leq |u[1, j_f]|$, and $|u'[j'_s, |u'|]| \leq |u[j_s, |u|]|$. At the same time,

$$|u| = |u[1, j_f]| + |u[j_s, |u|]| - |s^r| \geq |u'[1, j'_f]| + |u'[j'_s, |u|]| - |s^r| = |u'|$$

We can see that other parts of P''' and P'' are the same. Therefore, strings t''' and t'' collected by these paths are such that $|t'''| \leq |t''|$. Hence, any path that does not visit all nodes and corresponds to a superstring can be completed by the rest nodes and the new corresponding superstring does not become longer. Therefore, we can search the required path only among paths that visit all nodes exactly once.

Finally, let us show that if P has the maximal weight and visits each node exactly once, then it is the shortest superstring. Assume that we have another path P' that visits each node exactly once but the weight $w(P') < w(P)$. Let m and m' be the lengths of the superstrings collected by P and P', respectively. Then, $m' = L - w(P') > L - w(P) = m'$. Therefore, the superstring collected by P is the shortest superstring. □

Appendix E The implementation of the function GetSymbol(i,j)

For access to $\tilde{s}^i$, we do not need to concatenate these strings. It is enough to implement GetSymbol(i, j) function that returns j-th symbol of $\tilde{s}^i$. The index of the string $i \in \{1, \ldots, n\}$, and the index of the symbol $j \in \{1, \ldots, |s^{i+1}| + \cdots + |s^n| + i - 2\}$. The function has $O(1)$ query complexity. The implementation of the function is based on Binary search and has $O(\log n)$ time complexity and $O(1)$ query complexity, it is presented in Algorithm 14. The subroutine requires precalculated array *start* such that $start[i] = |s^1| + \cdots + |s^{i-1}| + 1$ for $i \in \{1, \ldots, n\}$, and $start[1] = 1$. It can be computed in $O(n)$ query and time complexity using formula $start[i] = start[i-1] + |s^{i-1}|$ for $i \in \{2, \ldots, n\}$. It can be done once before the whole algorithm.

Algorithm 13 SEARCHSEGMENT(u). Searching for an indexes segment of suffixes for t that have u as a prefix

$low \leftarrow NULL$, $high \leftarrow NULL$
$l \leftarrow 1$
$r \leftarrow n$
$Found \leftarrow False$
while $Found = False$ and $l \leq r$ **do**
  $mid \leftarrow (l + r)/2$
  $pref \leftarrow t[suf_{mid}, \min(n, suf_{mid} + |u| - 1)]$
  $pref1 \leftarrow t[suf_{mid-1}, \min(n, suf_{mid-1} + |u| - 1)]$
  $compareRes \leftarrow$ QCOMPARE$(pref, u), compareRes1 \leftarrow$ QCOMPARE$(pref1, u)$
  if $compareRes = 0$ and $compareRes1 = -1$ **then**
    $Found \leftarrow true$
    $low \leftarrow mid$
  end if
  if $compareRes < 0$ **then**
    $l \leftarrow mid + 1$
  end if
  if $compareRes \geq 0$ **then**
    $r \leftarrow mid - 1$
  end if
end while
if $Found = True$ **then**
  $l \leftarrow 1$
  $r \leftarrow n$
  $Found \leftarrow False$
  while $Found = False$ and $l \leq r$ **do**
    $mid \leftarrow (l + r)/2$
    $pref \leftarrow t[suf_{mid}, \min(n, suf_{mid} + |u| - 1)]$
    $pref1 \leftarrow t[suf_{mid+1}, \min(n, suf_{mid+1} + |u| - 1)]$
    $compareRes \leftarrow$ QCOMPARE$(pref, u), compareRes1 \leftarrow$ QCOMPARE$(pref1, u)$
    if $compareRes = 0$ and $compareRes1 = +1$ **then**
      $Found \leftarrow true$
      $high \leftarrow mid$
    end if
    if $compareRes \leq 0$ **then**
      $l \leftarrow mid + 1$
    end if
    if $compareRes > 0$ **then**
      $r \leftarrow mid - 1$
    end if
  end while
end if
return $(low, high)$

Algorithm 14 Implementation of GetSymbol(i, j) for $i \in \{1, \ldots, n\}, j \in \{1, \ldots, |s^{i+1}| + \cdots + |s^n| + i - 2\}$.

$j \leftarrow j - 1 + start[i + 1] + i$▷ We update all symbol indexes by indexes in the concatenation of all strings from S via \$-separator.
$left \leftarrow i + 1$, $right \leftarrow n$
$symbol \leftarrow NULL$ ▷ The symbol is unknown at the moment
while $left \leq right$ **do**
  $mid \leftarrow \lfloor (left + right)/2 \rfloor$
  $jm \leftarrow start[mid] + (mid - 1)$ ▷ The index of the first symbol of s^{mid}
  if $j \geq jm$ and $j < jm + |s^{mid}|$ **then** ▷ j is inside s^{mid}
    $j' \leftarrow j - jm + 1$
    $symbol \leftarrow s^{mid}_{j'}$
    stop the while loop.
  end if
  if $j = jm + |s^{mid}|$ **then** ▷ the j-th symbol is separator
    $symbol \leftarrow \$$
    stop the while loop.
  end if
  if $j < jm$ **then**
    $right \leftarrow mid - 1$
  else
    $left \leftarrow mid + 1$
  end if
end while
return *symbol*