SHORTEST COMMON SUPERSEQUENCE

WITH APPLICATIONS

by

Muzhou Chen

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2024

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Binhai Zhu, for all the great guidance, patience, and encouragement throughout my academic journey. It has been an exceptional experience working with Dr. Binhai Zhu.

I would also like to extend my gratitude to my committee members, Dr. Brendan Mumey and Dr. Sean Yaw, for all their great guidance and support.

Lastly, I would like to thank my parents for their unwavering love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF ALGORITHMS

Algorithm Page

ABSTRACT

The shortest common supersequence problem (SCS) is a classical problem that was first studied by Maier in 1978. Recently, some research on the tree-child network inference problem, developed from the inference of phylogenetic networks, which involves a variant of SCS, has been studied. The variant of the shortest common supersequence problem on permutation strings was introduced to solve the tree-child network problem. In this thesis, we first provide the sketch of the proof of the NP-completeness of the SCS problem on the permutation strings from the feedback vertex set problem. From the reduction, we continue analyzing the inapproximability of this new variation. Then, we study the FPT tractability of SCS using different parameters. With the new reduction developed, which corrected and simplified the former reduction provided by Maier from the Vertex Cover problem, we extended the proof to the non-existence of the FPT algorithm parameterized by the number of the inserting positions in the longest input sequence even if it is fixed to be 3. On the other hand, an FPT algorithm parameterized by the solution size has been explored. Furthermore, we extended our proof on the NP-completeness and W[1] hardness of the k-disjoint-SCS problem as another variant of the SCS. Lastly, a new heuristic algorithm has been constructed for practical use by using the concept of maximum increasing substrings, which can be applied to both permutation strings and non-permutation strings. The empirical results show that both algorithms perform similarly in producing the shortest common supersequence, while our algorithm takes longer runtime on average due to the more complex computation during the process.

1

INTRODUCTION

Background

The Shortest Common Supersequence problem is a well-known NP-complete problem arising from many different fields. In general, given a set of sequences $S = \{s_1, s_2, ..., s_n\}$, the purpose is to minimize the length of the supersequence T over S. We say T is the supersequence over S if we can convert T to any s_i in S by removing some letters from T. Its applications can be found in data compression, robot motion planning, workflow control, and many other areas. For example, in workflow control, each production line will have different jobs that need to be completed. In fact, each job could have different tasks that need to be completed in a certain order to minimize the overall number of tasks and to reduce the cost. The graph in Figure 1.1 represents the management's decision to merge two jobs to minimize the size of the solution. The instance can be directly converted to the SCS problem, in which each individual task can be represented by a letter, and a sequence is represented by a job.

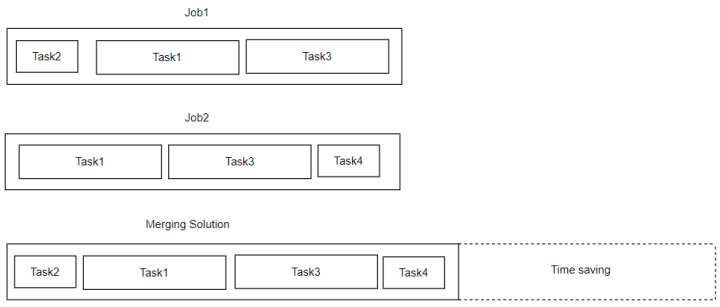

Figure 1.1: A Work Flow Control Example.

Similar to plan merging, another application of the SCS problem arises in robot task

plans, where it is also important to minimize the cost. An important type of helpful goal interaction occurs when certain operators in a plan can be grouped or merged together in such a way so as to make the resulting plan more efficient to execute [4]. However, classical planning theory does not offer much insight into ways of obtaining efficient plans [5]. A similar application can be found in computer-aided process planning for manufacturing and production environments which will require solutions to several fundamental problems in planning and geometric reasoning [7]. Another application involved is with database systems. In a database system enhanced with inference capabilities, a simple query involving a rule with multiple definitions may expand to more than one actual query that has to be run over the database [12]. Additionally, the SCS problem also arises in data compression [13]. It helps efficiently encode multiple sequences in dictionary-based compression algorithms, minimize redundancy, and achieve a more compact representation of a set of sequences. The difficulty of the foundational SCS problem brings the hardness from each field.

Most of the applications we have discussed so far are in the field of manufacturing, while the most popular field related to SCS problems is probably computational biology. Over the years, many complex problems related to the analysis of genomes have been explored in computational biology, which is a combination of biology, computer science, mathematics, and statistics. Most of these problems involve the analysis of the input of DNA and RNA sequences. A DNA string can be represented by a sequence of letters over $\{A, C, G, T\}$, which brings hardness on computing due to the large volume of genomic data. A great number of approaches have been developed for different aspects. For example, the restricted version of SCS problem with two input sequences can be solved by the standard method dynamic programming [6]. On the other hand, different data structures are used to represent genomic data for convenience, and a string is used as the most basic representation of a DNA/RNA sequence for comparison and manipulation. Many analyses and processes are produced on the input sequences in different problems, such as sequence alignment, overlap detection,

gap filling, and supersequence construction. One of the applications in this field is to use computers to aid protein sequence determination, where computer programs are used to determine the sequence of the amino acids in a protein from the knowledge of the structure of overlapping peptides obtained by hydrolysis of the protein [2]. The merging operation is widely used to determine the overlap between sequences for different biological purposes.

Some of the more complicated data structures, such as phylogenetic networks, are developed for reconstructing genomes and storing additional information. The phylogenetic networks, as one of the complex data structures, are commonly used to model the evolution of genomes [15]. The process of protein transportation involves the hardness of merging strings. It is shown that several transport protein families have evolved independently of each other, employing different routes at different times in evolutionary history to give topologically similar transmembrane protein complexes [11]. In some recent research, the tree-child network inference problem (TCN) has been introduced from the phylogenetic network problem with a constraint on the non-leaf node, and the shortest common supersequence problem for permutation strings (SCSP) has been introduced due to the equivalence to the TCN problem [1]. The example shown below indicates how phylogenetic networks store the sequences.

Based on the graph from Figure 1.2 shown below, we have three active letters in the alphabet $\Sigma = \{a, b, c\}$. The leaf nodes represent the letters from the alphabet. The purpose of having r_i's as internal nodes is to represent the arrangement of the leaf nodes. The example phylogenetic tree represents a string $T = abbacb$.

As the tree grows larger by the increasing size of the input string, the process will be more complicated due to the increasing number of internal nodes and the additional edges needed. The construction of the phylogenetic network varies in the process of combining those phylogenetic trees, depending on different rules. For example, the parsimonious tree-child networks for multiple trees can be constructed from the lineage taxon strings (LTSs)

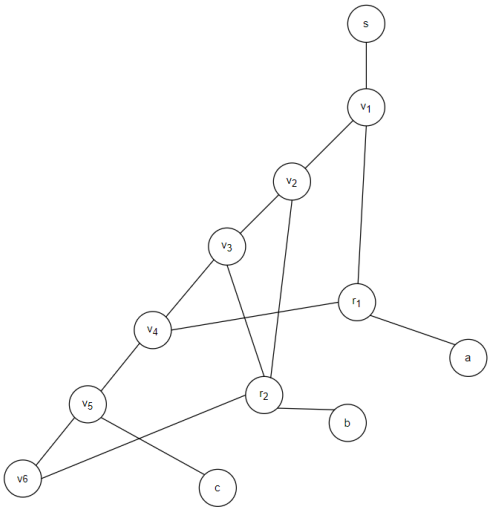

Figure 1.2: Example of Phylogenetic Tree.

of taxa. The Tree-Child Network (TCN) problem has been discussed, and a parameter of hybridization number (HN) is introduced as $HN(N) = \sum_{i=1}^{R(r)}(indegree(v_i) - 1)$, where $R(r)$ represents the set of internal nodes and we want to minimize $HN(v)$. The recent research by Bultaeu and Zhang has proved the equivalence of the TCN and SCS problem, which is simplified as shown in Table 1.1 below. Basically, the TCN problem can be solved by reducing it to multiple SCS sub-problems. The instance is represented by lineage taxon strings of the taxa, which can be effectively transformed from the SCS instance [1] .

The SCS problem here also needs to minimize the length of a supersequence by a given set of subsequences. By adding a constraint on the input set of strings, Bulteau and Zhang introduced the Shortest Common Supersequence on permutation strings (SCSP) to prove the NP-hardness for line trees, which is a variant of the TCN problem with permutation leaf nodes. Over the years, many negative results have been developed on SCS and its related problems; however, it is still challenging to get a better solution close to the optimal due to

Table 1.1: Equivalence of SCS and TCN problems.

$Tree_1$	$Tree_2$	$Tree_3$	SCS
ab	bd	ad	abd
c	ac	bc	abc
d	ϵ	ϵ	d

its hardness.

Existing Results

Existing Theoretical Results

The Shortest Common Supersequence (SCS) problem was first studied by Maier in 1978 [9]. The study on the SCS problem is an extension of the study on the complexity of the longest common subsequence problem (LCS) as its complement. Maier proved the NP-completeness of the SCS problem for the alphabet with arbitrary size by constructing a reduction from the Vertex Cover problem. A threading scheme was introduced to cover the LCS problem, and the idea of dividing the solution into different blocks to fix the length of the result remains in the reduction for the SCS problem. Furthermore, the same result was obtained for the SCS problem on a fixed alphabet size of 5. A similar reduction from the Vertex Cover problem holds by using a size-5 alphabet to encode the graph.

Some of the variants of the SCS problem have been explored by Timkovskii [14]. Specifically, the problem was shown to be NP-hard even when the length of the input sequences is 2. The same result holds if the maximum number of occurrences of the letter over the input sequences is fixed to be 3. More precisely, in the case of the maximum number of occurrences of a letter over the input sequences being 3, the problem was shown to be NP-hard, and the problem becomes polynomial-time solvable if the maximum letter

occurrences is 2. The proof is based on the Bounded Regular Refinement (BRR) problem as it is a polynomial-time equivalent problem to the SCS problem with a fixed length of input sequences of 2. The reduction is constructed from a variant of the Cycle Cutting Vertex Set problem (CCVS), 3-CCVS, in which each vertex in the given directed graph (digraph) connects to exactly 3 edges. For any v in the vertex set V, v must have either two incoming edges and one outgoing edge or one incoming edge and two outgoing edges to satisfy the total number of degrees. The 3-CCVS problem is shown to be NP-complete by a reduction from the CCVS problem. The construction is then done by transforming any digraph G to a digraph G' in which any vertex has exactly 3 edges by adding additional vertices to spread the additional edges.

Further discussion on the SCS problem changed the focus to its inapproximability. The paper by Jiang and Li [6] has shown the non-existence of a polynomial-time approximation algorithm scheme (PTAS) for the SCS problem by using the result from the proof of NP-hardness by Timkovskii. Additionally, the further proof indicates that SCS with an input number of sequences n does not have a polynomial-time approximation algorithm with a ratio $log^{\delta}n$ unless NP is in $DTIME(2^{polylogn})$. A new approach is introduced for average-case analysis of the performance of an algorithm using Kolmogorov complexity. The result indicates that a near-optimal solution for SCS can be found in the average case.

The former results indicate the inapproximability of the SCS problem. In 2003, Pietrzak was the first to study the $W[1]$-hardness of the fixed alphabet shortest common supersequence (FSCS) problem, with the parameter being the number of input sequences [10]. The proof was obtained by a reduction from the Partitioned Clique (pClique), and pClique was proven to be in $W[1]$ by a reduction from the original Clique problem. A later research by Dondi includes an FPT algorithm on a new variant called Constrained Shortest Common Supersequence (C-SCS) problem, with an input of two sequences, and a constraint T_X of the lower bound of the number of occurrences of the letter X in the solution [3].

Existing Heuristic Methods

A heuristic algorithm, called Majority-Merge, has been developed by Jiang and Li to solve the Shortest Common Supersequences problem [6]. On the input set of subsequences S, the algorithm initializes the result to be an empty string. By iterating each sequence from S, the occurrences of the first letter from the input set of sequences are calculated. The algorithm takes the letter with the maximum occurrence, appends it to the end of the result, and removes it from the original sequences. The result is returned once all input sequences are empty.

Organization

1. In the chapter two, the problem statements and notations from the thesis are defined. Some additional definitions that are used to construct the thesis are introduced.

2. In the third chapter, we discuss the theoretical and parameterized results. For the SCSP problem, we prove its inapproximability. For the general SCS problem, we investigate the existence and non-existence of FPT algorithms parameterized by solution size and by the number of inserting positions. Additionally, we introduce a new variant of the k-disjoint-SCS problem and prove its NP-completeness and W[1]-hardness.

3. In the fourth chapter, a new heuristic algorithm is designed. The analysis is performed based on the comparison to the existing heuristic algorithm Majority-Merge. Three input sets of sequences with different properties are used to test the overall performance of the algorithm. Both algorithms perform similarly in producing the shortest common supersequence over all three datasets, but our algorithm takes slightly more runtime due to its computational complexity.

4. In the conclusion chapter, a summary of all the observations from the thesis is discussed. Additionally, some potential future works are discussed as open problems in the

end.

PRELIMINARIES

In this chapter, we introduce the relevant definitions and necessary notations from the thesis.

FPT Algorithms

FPT (Fixed-Parameter tractable) algorithms are commonly used to solve NP-hard problems. We consider the parameterized problems to be decision problems. The FPT algorithms take a fixed parameter k, in which the expression of the runtime of the algorithm is represented as a function $T(n, k)$ on the input size of n and a parameter k. We define the parameterized problem to be fixed-parameter tractable (FPT) if there exists an algorithm which runs in $f(k)n^c$ for some constant c, where f is any computable function.

FPT also denotes the class that contains the fixed parameter tractable problems. The W hierarchy is a collection of computational complexity classes, where FPT = $W[0]$. In the W hierarchy, FPT $\subseteq W[1] \subseteq ...W[\text{p}]\subseteq$ XP, where XP denotes the class that contains all parameterized problems which admit $O(n^k)$ algorithms and FPT $\neq$ XP. The list of computational complexity classes $W[i]$ follows an order by increasing hardness. Many classic computational problems belong to the $W[1]$ class, such as the Independent Set problem and the k-clique problem. They are proven not to have an FPT algorithm unless FPT = $W[1]$.

The O^* Notation

For a parameterized algorithm which takes an input of k and a parameter x, the notation $O^*(f(k))$ indicates the algorithm runs in $O(f(k) * poly(|x|))$.

Polynomial-time approximation scheme (PTAS)

Polynomial-time approximation scheme (PTAS) is a type of approximation algorithm. For

the minimization problem, the algorithm takes a variable ϵ and use $1+\epsilon$ as a factor to produce a solution of size at most $(1+\epsilon)*L$, where L is the optimal solution. The factor $1-\epsilon$ is used for maximization problem and produce the solution with a lower bound $(1-\epsilon)*L$. The running time is bounded to be in the polynomial time on the problem size n and can be exponential in terms of ϵ.

Sequences and Alphabet

A sequence in computational biology refers to a collection of genetic letters from an alphabet Σ in a certain order, where an alphabet Σ refers to the domain of the genetic letters that can be used to construct the sequences. The length of the sequence s is represented by $|s|$. An input set of sequences is represented by $S = \{s_1, s_2, ..., s_n\}$.

Shortest Common Supersequence

Given two sequences $A_1 = a_1a_2a_3...a_l$ and $A_2 = b_1b_2b_3...b_i$. A sequence T is the common supersequence of A_1 and A_2 if we can convert T to both A_1 and A_2 by removing some letters from T. The shortest common supersequence is a common supersequence T with the minimum length. The shortest common supersequence can be generalized to n sequences $A_1, A_2, ..., A_n$.

Graphs and Cycle

Both directed graph and undirected graph have been used in this thesis. A graph can be represented by $G = (V, E)$ with the set of edges $E = \{e_1, e_2, ..., e_m\}$ and a set of vertices $V = \{v_1, v_2, ..., v_m\}$. In a directed graph (digraph) $D = (V, E)$, each edge $e_i = (x_i, y_i)$ represents an edge with direction from vertex x_i to y_i, where $(x_i, y_i) \neq (y_i, x_i)$ as long as $x_i \neq y_i$. A cycle in a graph is an non-empty path where only the starting vertex and the last vertex are the same.

Increasing Subsequences

Given two sequences $A = a_1a_2a_3...a_m$ and $B = b_1b_2b_3...b_m$. The increasing subsequences of B on A are represented by a set T. For each sequence $t_i \in T$, t_i is a continuous portion of B, such that $B = t_1 \cup t_2... \cup t_l$, and t_i is a subsequence of A for any t_i in T.

Problem (1): Shortest Common Supersequence problem (SCS)

Input: A set of sequences $S = \{s_1, s_2, ..., s_n\}$, an alphabet Σ

Question: Find the supersequence T of s_i's with the minimum length.

Problem (2): Shortest Common Supersequence problem on Permutation Strings (SCSP)

Input: A set of permutation sequences $S = \{s_1, s_2, ..., s_n\}$ over an alphabet Σ

Question: Find the supersequence T with the minimum length.

In this variant, a constraint on the input set of sequences s is applied, where for every s_i in S, s_i is a permutation over Σ.

Problem (3): Feedback Vertex Set problem (FVS)

Input: A digraph $D = (V, E)$

Question: Find the minimum number of vertices T that need to be removed from G so that there is no cycle remaining in the graph.

Problem (4): Vertex Cover problem (VC)

Input: An undirected graph $G = (V, E)$

Question: Find a set of of vertices V' such that all the edges from E can be covered by the vertices in V'.

The graph is covered by V' if and only if every edge $e_i \in E$ is covered by V'. For any edge $e_i = (x_i, y_i)$, e_i is covered by V' if at least one of x_i and y_i is in V'.

Problem (5): k-disjoint-SCS problem

Input: A supersequence T over the set of sequences $S = \{s_1, s_2, ..., s_n\}$.

Question: Are there k disjoint segments of T, $T_1, T_2, ..., T_m$ with total length l, such that for any S_i in S, s_i is a subsequence of some T_m in T?

Problem (6): Exact Set-Cover problem

Input: A set $B = \{b_1, b_2, ..., b_m\}$. A set Y of n subsets of B, i.e., $Y = \{Y_i | Y_i \subset B, i \in [n]\}$.

Question: Are there k elements (disjoint subsets of B) subsets in Y such that the union of Y_i is exactly equal to B?

Topological Sort Algorithm

```
1:  Function TopologicalSort(G)
2:    visited[v] = False
3:    For  v in G
4:      if visited[v] == False:
5:        TopologicalSortRec(v, visited, stack)
6:        end
7:    end
8:    stack.push(v)
9:  End Function
10: Function TopologicalSortRec(v, visited, stack)
11:   visited[v] = True
12:   stack = [];
13:   For  v in V
14:     if visited[v] == False:
15:       TopologicalSortRec(v, visited, stack)
16:       end
17:   end
18:   return stack
19: End Function
```

Algorithm 2.1:
The topological sort algorithm.

THEORETICAL AND PARAMETERIZED RESULTS

In this chapter, we investigate the Shortest Common Supersequence problem and its variants on the input of $S = \{s_1, s_2, ..., s_n\}$ with the maximum length of the input sequences being m, i.e., $m = max(|s_i|), 1 \leq i \leq n$.

Inapproximability of SCSP

The reduction constructed based on the FVS problem can be applied to the general SCS problem for the improvement of efficiency and correctness, where the original NP-completeness was proven by the reduction from the VC problem by Maier [9], and the reduction from FVS was proven by Timkovskii [14]. We will first provide a sketch of the proof by Timkovskii on the NP-completeness of SCS with each input sequence being of length 2 (SCS-2).

Theorem 1: SCS-2 is NP-complete.

Proof. The proof of the NP-completeness on the SCS-2 problem is based on the reduction from the FVS problem. Given the instance associated with the FVS problem as a digraph $D = (V, E)$, for any edge $e_k = (v_i, v_j)$, we convert it to a corresponding sequence $v_i v_j$ to form the set of sequences for the SCS-2 problem. We claim that the FVS has a solution size of T iff SCS-2 has a solution size of $N + |T|$, where $N = |V|$.

$\rightarrow$: In the input graph D, each edge is converted to a sequence. Each cycle in graph D will lead to an additional duplicated letter when constructing the solution for the SCS to cover all edges. Since the rest of the graph contains no cycle, that portion of the solution of SCS-2 problem has a fixed length of N, which is the number of vertices in G. Therefore, if FVS has a solution of T, SCS-2 must have a solution of size $N + |T|$.

$\leftarrow$: Given a solution of SCS-2, we can simply construct the result by counting the duplicated letters. Assuming that there is an additional duplicated letter that is not in FVS solution T,

the solution will not be optimal since the corresponding sequences of the remaining edges in graph D after removing the vertices in the FVS solution will only need N letters to cover. This length N portion of the solution can be produced by applying the topological sort on the remaining graph after removing the FVS solution. It can be proved that after the duplicated letters in T are deleted from D, the resulting graph does not contain any cycle. We refer to [14] for more details. Therefore, it is a contradiction to the assumed optimality. □

We will then follow the example in Figure 3.1 for a detailed explanation. With the example of digraph D_1, the construction of the instance S of SCS is shown below.

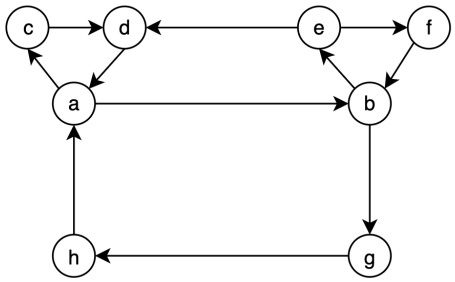

Figure 3.1: Input Graph D_1 for FVS.

$$S = \{ac, ab, be, bg, cd, da, ed, ef, fb, gh, ha\}$$

$\rightarrow$: Given the solution of FVS as $\{a, b\}$, we solve the corresponding SCS-2 problem on S with the following steps.

Since the topological sorting algorithm can be applied on the directed acyclic graph, the first step is removing $\{a, b\}$ from D_1 to form D_1'. Then, apply topological sorting algorithm on the input D_1'.

The algorithm starts by initializing a stack for storing the result. Mark the current

node v_i as visited, and recursively call the TopoligicalSortRec() function for all the vertices adjacent to v_i. Then push the current node to the stack.

Applying the algorithm on D_1' will produce a topological order for D_1'. In our example, the topological order for D_1' could be $\{cedfgh\}$. By adding the result from FVS to the head and tail of the string, we have our result for SCS-2 to be $\{abcedfghab\}$, where the order of $\{a, b\}$ does not matter as the duplication on both head and tail will cover all the possible arrangements for constructing the supersequence.

Based on the property of topological order, each vertex appears before all the nodes it points to. The fact that each vertex only appears once in the result means that the supersequence for D_1' must be optimal. By adding the solution of the FVS as discussed above, every edge with the vertices in the FVS solution will be covered. Therefore, the SCS-2 of S is $\{abcedfghab\}$, with a length of $|V| + T = 10$.

$\leftarrow$: Given a solution for SCS-2, the order is not necessarily $\{abcedfghab\}$, for example, $\{abcedfbgha\}$ can be another solution for SCS-2. Since the duplicated letters are still the same, we can convert our FVS solution, which is 2, by just counting the duplicated letters.

Based on the reduction provided by Timkovskii [14], we extend the reduction from the FVS problem for the analysis on its inapproximability. In fact, we obtained this proof on the NP-completeness of SCSP independently in 2023, while Bulteau and Zhang provided the same reduction [1].

Theorem 2: SCSP is NP-complete.

Proof. To show the NP-completeness of the SCSP problem, we establish the reduction of FVS $\leq_p$ SCSP. Let $D = (V, E)$ be a general digraph as the instance of Feedback Vertex Set, where $V = (v_1, v_2, v_3, ..., v_n)$.

We convert each vertex v_i in V to a letter in the alphabet Σ, then define the corresponding $*_i$ and $*_i'$ as blocks. For each edge $e_j = (v_a, v_b)$ in D, we construct a sequence starting with $v_a v_b$, followed by $*_a *_a' *_b *_b', ..., *_n *_n'$. Lastly, we add the remaining letters to the

end, ordered by their indices in V from the alphabet excluding $v_a v_b$ and the blocks, which is $\Sigma \backslash \{v_a, v_b\}$. We claim that the FVS has a solution of size $|S_{FVS}| = X$ for D iff the constructed set of the sequences has the shortest common supersequence of length $|S_{SCSP}| = X + T$, where $T = 4n$.

$\rightarrow$: Based on the construction, the solution for SCSP is fixed and formed by three sections. The first section correlates with the solution of FVS, and the last section is formed by putting all remaining vertices in order. The two sections are forced to separate by the $2n$ stars as blocks. The total length of the last two sections is fixed to be $3n$. In the first section of the SCS solution, each cycle that has been removed to form the FVS solution will lead to a duplication of a letter so the SCS can cover the cycle. Therefore, if FVS has a solution of X, SCSP will have a solution of $X + T$, where $T = 4n$ since the first section has a length of $n + X$.

$\leftarrow$: On the other hand, we select such a string S_{SCSP} as a solution of SCSP with a length of $X + T$. Assuming that the string is separated into two parts, $S_{SCSP} = AB$, such that the second portion B is the smallest suffix that contains all blocks $*_a *'_a *_b *'_b, ..., *_n *'_n$ as subsequence. Let w be the subsequence of the remaining part A, it must satisfy $|w| \leq S_{SCSP} - 2n = S_{FVS} + 2n < 2n$. It shows that A may not contain all the blocks. Due to its order, for any sequence for example $s_i = v_a v_b *_a *'_a *_b *'_b, ..., *_n *'_n v_a v_b ... v_n$, the first section of $v_a v_b$ must be a subsequence of A, and the rest portion is covered by part B in the solution. Since the position of the first section is fixed, $S_{FVS} = S_{SCSP} - 4n$. Therefore, if the solution for SCSP is $S_{SCSP} = X + T$, the corresponding solution for FVS is $S_{FVS} = X$.

□

The following example is provided for further explanation. Following the construction, a set of sequences S is converted as the instance of SCSP. Given the input graph D_2 as shown in Figure 3.2.

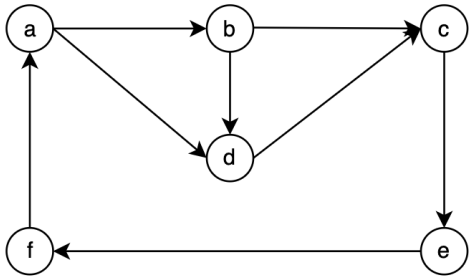

Figure 3.2: Input Graph D_2 for FVS.

The construction of instance D_2 to S is shown as follow:

$$\Sigma = \{a, b, c, d, e, f, *_a, *_b, *_c, *_d, *_e, *_f, *'_a, *'_b, *'_c, *'_d, *'_e, *'_f\},$$

$$S =$$

$$\{ab *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f cdef,$$

$$ad *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f bcef,$$

$$bc *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f adef,$$

$$bd *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f acef,$$

$$ce *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f abdf,$$

$$dc *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f abef,$$

$$ef *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f abcd,$$

$$fa *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f bcde\}.$$

$\rightarrow$: Based on the construction, each edge is corresponding to a sequence in S. In this case, the graph D_2 has a solution for Feedback Vertex Set of $\{c\}$ with a length of 1.

Based on the solution for FVS, which is FVSSol, since the order does not affect the solution on its length and correctness, there is no specific requirement on the order of FVSSol. We can then form our solution for SCS to be SCSSol by using the following steps:

1. Copy FVSSol to the begining and the ending of an empty sequence as to initialize SCSSol.

2. Initial $SCSSol = \{FVSSol, FVSSol\}$ and iterate through the edges $e = \{v_i, v_j\}$ in D_2.

3. If v_i is in FVSSol and v_j is not in SCSCol, add v_j before FVSSol at the tail: $SCSSol = \{FVSSol, ..., v_j, FVSSol\}$.

4. If v_i is not in FVSSol, v_j is in FVSSol, and v_i is not in SCSSol, we add v_i before FVSSol in the end: $SCSSol = \{FVSSol, ..., v_i, FVSSol\}$.

5. If v_i is not in FVSSol, v_j is not in FVSSol and v_j is not in SCSSol, we add v_iv_j before FVSSol in the end: $SCSSol = \{FVSSol, ..., v_iv_j, FVSSol\}$.

6. If v_i is not in FVSSol, v_j is not in FVSSol, v_j is in SCSSol, and v_j goes before v_i in SCSSol, we swap the order as follow: $SCSSol = \{FVSSol, ...v_j, ..., v_i, FVSSol\}$.

7. Add $\{*_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f abcdef\}$ to SCSSol and exit.

Since we removed a cycle $\{abcefa\}$ from the graph and the converted graph does not contain any cycle, in our set S, we have the first section of the solution for the SCS to be $cefabdc$. As the solution in the FVS shows that only one cycle has been removed, the solution of our supersequence for the edge representations will need one letter for both the beginning and the end of our solution. This is due to the fact that any path will need one additional edge from the tail to the head to form a cycle, whereas a path just needs a permutation of all vertices to represent. As there is no cycle remaining in graph D_2 after removing a, we form our final solution to be $cefabdc *_a *'_a *_b *'_b *_c *'_c *_d *'_d *_e *'_e *_f *'_f abcdef$, with a length of $X + T = 24 + 1 = 25$, where $T = 4n = 24$.

As we have $2n$ blocks in the middle section and they are in the same order of $*_a *'_a *_b *'_b$

$*_c *'_c *_d *'_d *_e *'_e *_f *'_f$, we guarantee the solution for this middle section must be following the same order. The solution for our last part is simply $abcdef$. Based on the way we constructed our sequence, since every time we remove two letters from V, the optimal solution must be fixed.

$\leftarrow$: Assume that the solution for SCSP doesn't follow the fixed order, e.g., (the first part of the answer)$*_a *'_a *_b *'_b *_c *'_c k *_d *'_d *_e *'_e *_f *'_f$ (the last part of the answer). Since in the first part of our SCS solution, the sequence is formed and closed by the solution of the FVS from D_2, the solution to SCS with k in the middle of blocks will cause additional duplications of the solution from the FVS after k in our solution to cover all cycles in D_2. On the other hand, due to the way we construct our S, the solution for the last part is fixed to be the letters of all vertices' representations in order. Therefore, the solution of SCS of $X + T$ indicates that the corresponding solution for the FVS is optimal.

Based on the above reduction, we now prove that the SCSP problem has no PTAS, unless $P = NP$.

Theorem 3: The SCSP cannot have a PTAS unless P=NP.

Proof. By considering the instance from the reduction above, a length $X + T$ solution is produced for SCSP, and $X + T$ is the optimal solution size. As SCSP is a minimization problem, by the definition, the associate factor for the potential approximation algorithm is $(1 + \epsilon)$, and the algorithm runs in polynomial time. Since ϵ can be assigned by any number, we simply assign $\epsilon = 1/(4T^2)$ and to form the solution to be $(1 + 1/(4T^2))(X + T)$. The equation is equivalent to $X + T + (X + T)/(4T^2)$. As the instance from the reduction above has the property of $1 \leq X \leq T$, the second portion of $(X + T)/(4T^2)$ results $(X + T)/(4T^2) \leq 2T/4T^2 = 1/2T$. Since the first half of equation $X + T$ gives us the optimal solution, and the second half must produce less than one add-on to the solution, such an approximation algorithm will produce the optimal solution with size $X + T$. This implies that the approximation algorithm with a factor $1 + \epsilon$ would solve SCSP optimally, and this

is a contradiction. Therefore, SCSP has no PTAS, unless $P = NP$.

□

By proving the hardness of SCSP on the permutations, the result motivates us to study the FPT-tractability of the general SCS problem.

FPT algorithms on the SCS problem

The previous results indicate the inapproximability of the SCS problem. Therefore, we turn our focus to the potential FPT algorithms for the SCS problem. As Pietrzak mentioned [10], the exact classification of SCS and its variants seem to be harder than W[P]; over the years, only Dondi found an FPT algorithm on one of its variants in 2013, the C-SCS problem [3]. Since the C-SCS version only takes two input sequences, there can be some potential improvements that we could make by taking this constraint off. In our research, two new parameters have been introduced: the solution size and the number of inserting positions. We provide the following analysis of the existence of FPT algorithms parameterized by these two new variables. We first consider whether an FPT algorithm exists and is parameterized by the solution size.

Existence of FPT Algorithm Parameterized by Solution Size

As just discussed, for the SCS problem, we introduce a new parameter k as the optimal solution size. In the input set of sequences $S = \{s_1, s_2, ..., s_n\}$, let m be the maximum length over s_i's. An FPT algorithm is constructed as shown in Algorithm 3.2.

Theorem 4: The SCS problem can be solved by an FPT algorithm parameterized by solution size k in $O^*(k^k)$.

Proof. Based on the algorithm, it takes $O^*(k^k)$ time to enumerate all possible strings T. For each string T_i, we use brute force to check if T_i is a supersequence of S, and it takes $O(m * n)$

time to complete. Therefore, the algorithm runs in $O(k^k * m * n) = O^*(k^k)$.

□

1: ***Function FPT(k)***$(S = \{s_1, s_2, ..., s_n\})$

2: Check if $k \leq m$ or $k \leq n$, if so, then return false.

3: Enumerate all possible strings of length k, represented by $T = \{T_1, T_2, ...\}$.

4: **For** each string T_i:

5: Using brute-force to check if T_i is a supersequence of the set of sequences S.

6: **End**

7: ***End Function***

Algorithm 3.2:
The FPT algorithm parameterized by solution size k.

Non-existence of FPT Algorithm Parameterized by Number of Inserting Positions in the Longest Input Sequence

The FPT algorithm parameterized by solution size is not viable in practice since the parameter k needs to be small enough to avoid an exponential increase in the running time. Therefore, another approach to solve SCS has been explored: On the input set of sequences $S = \{s_0, s_1, s_2, ..., s_{n-1}\}$, pick the longest sequence s_0 from S to form the initial solution T. Inserting letters into s_0 to obtain the final solution T, we introduce another parameter which is the number of inserting positions p in s_0. We show that the SCS problem does not have an FPT algorithm parameterized by the number of the inserting positions p. The simplified proof is based on the reduction from the Vertex Cover problem to the SCS problem developed by Maier [9].

To establish the reduction of $VC \leq_p SCS$, we convert the instance of the VC problem as follows. For an input graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ and $E = \{e_1, e_2, ..., e_m\}$,

we set $k = max(m, n)$, and convert each vertex in V to be a letter in alphabet respectively, and add $*$'s to form blocks in the following steps. Each edge e_i is converted to a corresponding sequence s_i in S. Suppose that we have an edge $e_i \in E$, where $e_i = \{v_x, v_y\}$. The corresponding sequence s_i will be in the form of

$$s_i = e_i e_i v_x \overbrace{*...*}^{4k} v_y e_i e_i.$$

Given input $G = \{V, E\}, V = \{v_1, v_2, ..., v_n\}, E = \{e_1, e_2, ..., e_m\}$, we construct s_0 as

$$s_0 = v_1 v_2 ... v_n \overbrace{*...*}^{4k} e_1 e_1 e_2 e_2 ... e_m e_m e_1 e_1 e_2 e_2 ... e_m e_m \overbrace{*...*}^{4k} v_1 v_2 ... v_n.$$

By this construction, the solution of SCS is formed by inserting letter representations of VC solution into three fixed portions in the sequence s_0 as A, B, and C. Given a solution T' for VC on graph G, for any edge $e_i = (v_x, v_y)$, if v_y letter representation is in subsequence C, then we insert $e_i e_i$ letter representation into subsequence A. If not, then insert $e_i e_i$ letter representation into subsequence B to satisfy the content of supersequence. The solution is constructed from s_0 in the following form:

$$A v_1 v_2 v_3 ... v_n \overbrace{*...*}^{4k} e_1 e_1 e_2 e_2 ... e_m e_m C e_1 e_1 e_2 e_2 ... e_m e_m \overbrace{*...*}^{4k} v_1 v_2 v_3 ... v_n B.$$

We claim that if the VC problem has a solution of size $|T'|$, then the corresponding SCS will have a solution of $8k + 6m + 2n + |T'|$. Since VC has a solution T' of size $|T'|$, for every edge $e_i = (v_x, v_y)$, either v_x or v_y must be in the solution of vertex cover. For the case of v_x be in subsequence C, $e_i e_i$ is added in the subsequence B, such that $e_i e_i$ is satisfied by the first $e_1 e_1 e_2 e_2 ... e_m e_m$ section, v_x is satisfied by the subsequence C, v_y is satisfied by the second $v_1 v_2 v_3, ..., v_n$ section. And the last $e_i e_i$ is covered in the subsequence B. Since the number of stars is chosen from $max(m, n)$, the blocks fix the position of our insertion. As a

result, the solution for the corresponding SCS problem is $8k + 4m + 2n + |T'|$.

On the other hand, if the SCS problem has a solution of $8k + 4m + 2n + |T'|$, then the graph G has a vertex cover of size $|T'|$. Based on the construction, Since the number of blocks is $max(m, n)$, the solution is divided by the stars. The length of the rest of the sections is fixed to be $8k + 4m + 2n$ except the subsequence C since all the new edges will be either inserted in front of the solution or at the end of the sequence. As the middle section is fixed to be C, it produces an adding of length $|T'|$. Therefore, a solution of $8k + 4m + 2n + |T'|$ indicates the graph G has a vertex cover of size $|T'|$. Additionally, since the inserting positions are fixed to be in the subsequences of A, B, and C, $p = 3$.

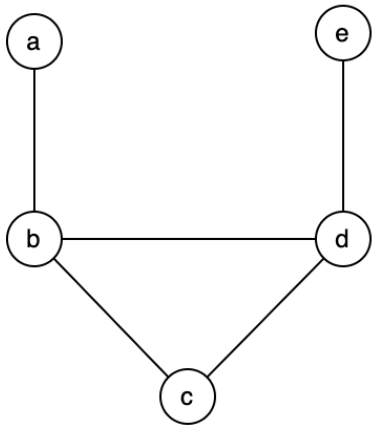

Figure 3.3: Input Graph G for VC.

An example is provided to explain the details of the reduction $VC \leq SCS$. The graph G shown in Figure 3.3 is defined by $G = \{V, E\}$, which $V = \{a, b, c, d, e\}$, $E = \{ab, bd, bc, dc, de\}$, or $\{e_1, e_2, e_3, e_4, e_5\}$ respectively. Graph G has a solution for vertex cover being $\{b, d\}$.

We apply the transformation from G to the instance of SCS problem S. By iterating

through the edges, we form the set of sequences S as follow:

$$S = \{s_0 = abcde\overbrace{*...*}^{20}e_1e_1e_2e_2e_3e_3e_4e_4e_5e_5e_1e_1e_2e_2e_3e_3e_4e_4e_5e_5\overbrace{*...*}^{20}abcde,$$

$$s_1 : e_1e_1a\overbrace{*...*}^{20}be_1e_1,$$

$$s_2 : e_2e_2b\overbrace{*...*}^{20}de_2e_2,$$

$$s_3 : e_3e_3b\overbrace{*...*}^{20}ce_3e_3,$$

$$s_4 : e_4e_4d\overbrace{*...*}^{20}ce_4e_4,$$

$$s_5 : e_5e_5d\overbrace{*...*}^{20}ee_5e_5\}.$$

The solution T is obtained by inserting letters into s_0 as shown. The size of our solution will be based on the insertions in the subsequences X, C, and Y with fixed positions as shown:

$$T = Xabcde\overbrace{*...*}^{20}e_1e_1e_2e_2e_3e_3e_4e_4e_5e_5Ce_1e_1e_2e_2e_3e_3e_4e_4e_5e_5\overbrace{*...*}^{20}abcdeY.$$

By adding bd to C, we then iterate through s_i's from s_1.By iterating through s_1, e_1e_1 is added to the front since b is contained in the subsequence C. By applying the same rule to the rest of the sequences, we have our final result as shown below:

$$T = e_1e_1abcde\overbrace{*...*}^{20}e_1e_1e_2e_2e_3e_3e_4e_4e_5e_5bde_1e_1e_2e_2e_3e_3e_4e_4e_5e_5\overbrace{*...*}^{20}abcdee_2e_2e_3e_3e_4e_4e_5e_5.$$

As a result, the solution has a total length of $8 * 5 + 6 * 5 + 2 * 5 + 2 = 82$.

Theorem 5: SCS does not have an FPT algorithm parameterized by the number of the inserting positions p in the longest input sequence s_0.

Proof. Assume that there exists an FPT algorithm for the SCS problem on a parameter

of the total number of inserting position p in the longest input sequence s_0. So the time complexity is defined as $O(f(p)N^c)$. Based on the reduction shown above, the process fixes the number of inserting positions $p = 3$ as of the specific sections X, C, and Y. Furthermore, the reduction indicates that the SCS instance remains NP-complete even with a fixed number of inserting positions of 3. However, this indicates a contradiction to the result of $O(f(p)N^c)$ from the FPT algorithm since $O(f(3)N^c)$ is polynomial. Therefore, there does not exist an FPT algorithm with a runtime $O(f(p)N^c)$ and parameterized by p even for a fixed $p = 3$, unless P = NP.

□

NP-completeness and W[1]-hardness of k-disjoint-SCS problem

The k-disjoint-SCS problem arises from the consideration of proving the W[1]-hardness on the SCS problem with a solution size of $l + k$, where l is the longest sequence in the input S and k is the number of letters inserted into the longest input sequence to form the final solution T. We call this case the $(l+k)$-instance of SCS, where k is the parameter. We failed to prove the W[1]-hardness on this $(l+k)$-instance; however, the k-disjoint-SCS problem can overcome some difficulties occurred from our failed attempt. Alternatively, we show that the k-disjoint-SCS problem is NP-complete and W[1]-hard by providing the reduction from Exact Set-Cover. The Exact Set-Cover problem was proven to be W[1]-hard by a reduction from the MULTICOLOERED-CLIQUE problem [8].

Given the instance of Exact Set-Cover problem $Y = \{Y_1, Y_2, ..., Y_n\}$ and $B = \{b_1, b_2, ..., b_m\}$, the subsets in Y are sorted based on the order in B. For each subset Y_i we use $E(Y_i)$ as the sequence of the elements in Y_i. We use two notations # and $*$ to form the blocks in T as follow:

$$\#E(Y_1) * E(Y_1)\#\#E(Y_2) * E(Y_2)\#\#...\#\#E(Y_n) * E(Y_n)\#.$$

and we convert b_i to S_i as follow:

$$\#b_i * b_i\#.$$

Theorem 6: k-disjoint-SCS problem is NP-complete and W[1]-hard.

Proof. Based on the construction, we claim that the Exact Set-Cover problem has a solution k iff there are k disjoint segments of $T = \{T_1, T_2, ..., T_k\}$ and the total length of $q = 2m + 3k$, such that for all $S_i \in M$, S_i is a subsequence for some $T_j \in T$.

$\rightarrow$: If the Exact Set-Cover problem has a solution of k, the k-disjoint-SCS problem will have k disjoint segments of $T = \{T_1, ..., T_k\}$ with a total length of $2m + 3k$, where the $2m$ comes from the m distinct pairs of b_i's, and the $3k$ is formed by the $2k$ of #'s and k of $*$'s. The format of T is fixed as #'s can't be in the middle of two b_i's. We prove by contradiction, assuming that we have one Y_a which contains $2t$ of #'s. The total number of #'s and $*$'s is $3t + 3(k-1) > 3k$, which means $2m + 3t + 3(k-1) > q$, and that provides a contradiction. Therefore, the original claim is true.

$\leftarrow$: On the other hand, if the k-disjoint-SCS problem has a solution of k, and the total length $q = 2m + 3k$, then the Exact Set-Cover problem has a solution k. As of the construction, the k disjoint segments indicates the result of Exact Set-Cover will have a solution of k.

Since the Exact Set-Cover problem is known to be NP-complete and is W[1]-hard, the k-disjoint-SCS problem is also NP-complete and W[1]-hard.

□

We explain the details by an example. Given a set $B = \{1, 2, 3, 4, 5, 6\}$, $Y_1 = \{1, 2, 3\}$, $Y_2 = \{2, 3, 5\}$, $Y_3 = \{4, 5\}$, $Y_4 = \{5, 6\}$, $Y_5 = \{4, 5, 6\}$, we construct T and S as follow:

$$S_1 = \#1 * 1\#,$$

$$S_2 = \#2 * 2\#,$$

$$S_3 = \#3 * 3\#,$$

$$S_4 = \#4 * 4\#,$$

$$S_5 = \#5 * 5\#,$$

$$S_6 = \#6 * 6\#.$$

For $k = 2$, $q = 2m + 3k = 18$, the solution for Exact Set-Cover is $\{Y_1, Y_5\}$. The solution for k-disjoint-SCS have the only solution of $T_1 = \#123 * 123\#$ and $T_2 = \#456 * 456\#$, with a total length of $9 + 9 = 18$.

Closing Note

The results in this chapter have been accepted to be presented at AAIM'24, Dallas, Texas, September 21 - 23, 2024.

EMPIRICAL RESULTS

Following the negative result of non-existence of any potential FPT algorithm parameterized by the number of inserting positions in the longest input sequence, we changed our focus to the practical algorithms. By the first consideration of developing an algorithm based on the conflicting pairs, the algorithm can not avoid the fact that using conflicting pairs results in losing the important information of the sequential order. However, another heuristic algorithm is introduced for practice application in this chapter by using the maximum number of increasing substrings. Some empirical results are produced by the analysis in comparing the new algorithm with the existing Majority-Merge algorithm on different data sets.

Algorithm Description

The algorithm is constructed by using the concept of the increasing subsequence. Given two sequences X and Y, the set of increasing subsequences are generated. Suppose that the number of increasing subsequences of Y to X is k, the supersequence T of X and Y can be generated by duplicating X for k times so that Y becomes a subsequence of T. By using this approach, on the input of the SCS problem, the minimum number of times that we need to duplicate on the target string can be found by iterating through all $s_i \in S$. Another iteration is needed on the input set of strings for removing the unnecessary letters from the duplication in order to shorten the result. The algorithm is shown below (Algorithm 4.3).

The algorithm takes $O(m*n)$ to calculate the maximum number of increasing substrings and to determine what the minimum number of duplications is needed to construct the result. Then the next step of determining which letters are not needed from the result takes $O(n * m * n)$ for iterating through all $s_i \in S$ and the original result. The algorithm then runs in $O(m*n)+O(n*m*n) = O(n^3)$ time. The algorithm does not track the property of

Function SCSByMIS(S)

for each $s_i \in S$:

  Calculate the maximum number of increasing substrings k by the target string s_1

  End for

Duplicate target string s_1 k times to form the initial solution T

for each letter x in T:

  for each sequence $s_i \in S$:

    if $x = y$:

      mark x in T

End for

  End for

    End if

Remove letters that are not marked in T and return the result

check if $k \leq m$ or $k \leq n$ then return false:

Enumerate all possible strings of length k, represented by $T = \{T_1, T_2...\}$

for each string T_i:

  using brute-force to check if T_i is a supersequence of the set of sequences S

End

End Function

Algorithm 4.3:
The SCSByMIS algorithm.

the parameters we introduced before such as inserting position. As a result, the algorithm is not proved to be parameterized by any new factor.

Data Description

The algorithm is tested on three different data sets. The first data set is generated by random numbers, where the property of each sequence in S is a permutation of the alphabet. The second data set is generated in the same way without the constraint on the input strings. Lastly, the third data set is constructed by the Escherichia coli U00096.3 genomic DNA sequence from the website https://www.ebi.ac.uk/ena/browser/view/U00096; partial DNA sequence from the original data set has been split to form the input sequences as for simulating the performance in the real world scenario.

Empirical Results

We implemented our new SCSByMIS algorithm based on *Algorithm*4.3. The algorithms were run on a desktop with AMD Ryzen 7 3800X 8-Core Processor, 3.89 GHz, 32GB RAM.

We first tested both Majority-Merge and SCSByMIS on the simulated permutation strings, the solutions are represented by T_M and T_S respectively shown below in Table 4.1. By increasing the input size $|S|$ and the size of alphabet, the two algorithms performed similarly in the length of the solution; however, the runtime of our algorithm increases significantly.

On the other hand, we tested our algorithm on non-permutation strings, which is simulated based on real world alphabet over $\{A, C, G, T\}$. The length of each string is randomly generated between 100 and 200, and the results are shown below in Table 4.2. The Majority-Merge algorithm performs better in this case. The smaller alphabet causes increasing number of letter duplications in the input strings as the string growth large, which

$\|S\|$	$\|\Sigma\|$	$\|T_M\|$	$\|T_S\|$	Runtime of Majority-Merge(s)	Runtime of SCSByMIS(s)
100	10	67	67	0.003	0.068
200	11	84	85	0.006	0.311
300	12	97	97	0.011	0.817
400	13	122	115	0.018	1.650
500	14	133	134	0.029	2.964
600	15	147	149	0.042	4.787
700	16	172	170	0.058	7.393
800	17	198	195	0.080	10.766
900	18	219	213	0.106	14.899
1000	19	243	243	0.137	20.694

Table 4.1: Test on simulated permutation strings.

Majority-Merge will take advantage of.

Lastly, we tested our algorithm on the real-world data constructed by Escherichia coli U00096.3, in which each input sequence has a length of 20 over an alphabet $\{A, C, G, T\}$. Based on the result from Table 4.3, Majority-Merge has an overall better performance on both runtime and the solution size. Due to the extremely high repetitions of each letter in the input sequence, a similar result to test two is observed. In test 2, since the length of each input sequence is generated between 100 and 200, the overall solution size for our algorithm is between 711 and 759. There is approximately a 40 increase in the solution size over the tests from input size 100 to 1000. Compared to our last test, the solution size for our algorithm falls between 228 and 279 over the tests, with about 50 increase in the solution size. As a result, our algorithm obtained a similar performance in the last test.

$\|S\|$	$\|T_M\|$	$\|T_S\|$	Runtime of Majority-Merge(s)	Runtime of SCSByMIS(s)
100	503	711	0.011	0.678
200	509	745	0.020	2.736
300	517	749	0.030	5.990
400	516	857	0.041	10.906
500	524	854	0.052	16.807
600	515	755	0.060	24.129
700	523	731	0.071	33.305
800	521	793	0.082	43.526
900	524	725	0.091	53.586
1000	524	759	0.101	67.992

Table 4.2: Test on simulated non-permutation strings.

$\|S\|$	$\|T_M\|$	$\|T_S\|$	Runtime of Majority-Merge(s)	Runtime of SCSByMIS(s)
100	174	228	0.004	0.278
200	173	264	0.008	1.107
300	171	266	0.012	2.463
400	176	266	0.016	4.328
500	175	268	0.019	6.828
600	176	274	0.023	9.788
700	174	276	0.027	13.431
800	174	278	0.032	17.902
900	174	278	0.036	22.412
1000	174	279	0.040	27.559

Table 4.3: Test on non-permutation strings from real-world data.

CONCLUSION

In this thesis, we studied the SCS problem and its variants. By considering the hardness of the SCS problem on the permutation strings, we provided a sketch of the reduction from the FVS problem, which was originally used to prove the non-existence of PTAS result mentioned by Jiang and Li [6]. Using the reduction, we proved that even the SCSP problem does not have PTAS, unless P = NP. The further development of the FPT algorithm produces a non-practical FPT algorithm that is parameterized by the solution size. Additionally, the non-existence of the FPT algorithm, which is parameterized by the number of the inserting position p in the longest input sequence, has been proved by using a previous reduction from VC to SCS introduced by Maier [9]. In the last part of Chapter 3, we introduced a new variant of the k-disjoint-SCS problem and provided a reduction from the Exact Set-Cover problem to show the NP-completeness and W[1]-hardness. In our fourth chapter, a new heuristic algorithm has been developed and tested over three data sets with different properties.

For some potential future work, we are still seeking any approximation algorithm for the SCS problem, possibly for some special cases. Some new parameters can be developed to use in constructing a new FPT algorithm for practice. The last question is whether we can design an FPT algorithm to solve the $(l + k)$-instance, where l is the length of the longest input sequence, and k is the number of letters inserted into the longest input sequence to form the final solution T.

REFERENCES CITED

[1] Laurent Bulteau and Louxin Zhang. The tree-child network problem and the shortest common supersequences for permutations are np-hard. *arXiv preprint arXiv:2307.04335*, 2023.

[2] Margaret O Dayhoff. Computer aids to protein sequence determination. *Journal of theoretical biology*, 8(1):97–112, 1965.

[3] Riccardo Dondi. The constrained shortest common supersequence problem. *Journal of Discrete Algorithms*, 21:11–17, 2013.

[4] David E Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2-3):143–181, 1992.

[5] Caroline C Hayes. A model of planning for plan efficiency: Taking advantage of operator overlap. In *IJCAI*, pages 949–953, 1989.

[6] Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[7] Raghu Karinthi, Dana Nau, and Qiang Yang. Handling feature interactions in process-planning. *Applied Artificial Intelligence an International Journal*, 6(4):389–415, 1992.

[8] Manuel Lafond, Binhai Zhu, and Peng Zou. Genomic Problems Involving Copy Number Profiles: Complexity and Algorithms. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[9] David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978.

[10] Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003.

[11] Milton H Saier Jr. Computer-aided analyses of transport protein sequences: gleaning evidence concerning function, structure, biogenesis, and evolution. *Microbiological reviews*, 58(1):71–93, 1994.

[12] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[13] James A Storer. *Data compression: methods and theory*. Computer Science Press, Inc., 1987.

[14] VG Timkovskii. Complexity of common subsequence and supersequence problems and related problems. *Cybernetics*, 25:565–580, 1989.

[15] Louxin Zhang, Niloufar Abhari, Caroline Colijn, and Yufeng Wu. A fast and scalable method for inferring phylogenetic networks from trees by aligning lineage taxon strings. *Genome Research*, 33(7):1053–1060, 2023.