An FPT Algorithm for STRING-TO-STRING CORRECTION

by

Serena Glyn Lee-Cultura
B.Sc., University of Victoria, 2007

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTERS OF SCIENCE

in the Department of Computer Science

An FPT Algorithm for STRING-TO-STRING CORRECTION

by

Serena Glyn Lee-Cultura
B.Sc., University of Victoria, 2007

Supervisory Committee

Dr. M. Serra, Supervisor
(Department of Computer Science)

Dr. U. Stege, Co-Supervisor
(Department of Computer Science)

Dr. J. Muzio, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. M. Serra, Supervisor
(Department of Computer Science)

Dr. U. Stege, Co-Supervisor
(Department of Computer Science)

Dr. J. Muzio, Departmental Member
(Department of Computer Science)

ABSTRACT

Parameterized string correction decision problems investigate the possibility of transforming a given string X into a target string Y using a fixed number of edit operations, k. There are four possible edit operations: swap, delete, insert and substitute. In this work we consider the $NP-$complete STRING-TO-STRING CORRECTION problem restricted to deletes and swaps and parameterized by the number of allowed operations. Specifically, the problem asks whether there exists a transformation from X into Y consisting of at most k deletes or swaps. We present a fixed parameter algorithm that runs in $O(2^k(k+m))$, where m is the length of the destination string. Further, we present an implementation of an extended version of the algorithm that constructs the transformation sequence ω of length ay most k, given its existence. This thesis concludes with a discussion comparing the practical run times obtained from our implementation with the proposed theoretical results. Efficient string correction algorithms have applications in several areas, for example computational linguistics, error detection and correction, and computational biology.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Micaela Serra,

Dr. Ulrike Stege,

Dr. Jon Muzio,

I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.

Edith Wharton

DEDICATION

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted. And so, it is a pleasure to thank the many people who made this thesis possible. To my mother and father, Barbara and Leo, who continue to learn, grow and develop and who have each been a great source of encouragement and inspiration to me throughout my life, a very special thank you for the myriad of ways in which you have actively supported me in my determination to find and realize my potential, and to make this contribution to our world. To Vanessa, for keeping me sane and relaxed during the whirlwind of writing. To Uncle Terry and Aunt Sandra, for helping me in every possible way imaginable, for continually going above and beyond the regular call of duty to ensure that I succeed in all aspects of being. And finally, to my beloved Grandma Doreen. I wish so much that you were here see this finally happen.

[**TO DO:** In Chapter 2 and 4 for all figures involving swaps and deletes, change the image so that single slash denotes tau, and a cross denotes a delete operation.]
[**TO DO:** in code printing $|X|$ to the csv file is incorrect. Both the upper and lower bound must be shifted down by one.]
[**TO DO:** change the running time in Chapter 4 search tree size is 2^k, and the running time of the algorithm is $(km2^k) \to (m2^{k+1}) \to (m2^k)$]
. [**TO DO:** As well, change it so that it says that the search tree size is 2^k. Use wording search tree in the table presenting running times.]

Chapter 1

Introduction

String correction explores the possibility of transforming a sequence of characters into a second sequence of characters under a predetermined set of string modifications, known as edit operations. There are four common edit operations considered in string correction, each of which is applied to individual or neighbouring characters throughout the sequence. They are *adjacent symbol interchange* (also called swap or transposition), *single symbol deletion*, *single symbol insertion* and *symbol substitution* (also called mutation). Given two strings, namely X and Y, the correction process involves determining a sequence of edit operations to apply to the source, X, such that after its application the resulting modified string will be equivalent to the target or destination, Y. It is important to note that during string correction, edits are applied to only the source string. That is, if the question is to determine a sequence of edit operations that will transform X into Y, then the operations will modify only X, leaving Y unchanged throughout the transformation.

A number of different string correction problems, each with their own application domain, are differentiated by the various edit operations permitted during the correction process. Some areas where efficient string correction algorithms are of importance are computational biology, computational linguistics, and error correction and detection [2]. For example, algorithms that detect the longest common subsequence (LCS) of two given strings are used in molecular biology to determine similarities between the input strings [11]. Another example is the use of Levenshtein distance algorithms in computational linguistics, specifically in speech recognition, to determine the similarities between a suggested hypothesis and the correct answer to a question. Levenshtein distance is defined as the number of edits required to transform

string X into string Y where the set of edit operations includes inserts, deletes, and substitutes [10], [12].

Table 1.1: String Correction Edit Summary Table. The permitted edits are shown in the leftmost column, while the best/first running time and reference are given in the remaining two columns. Note that all rows except 5 and 8 have polynomial running time. An algorithm for the $NP-$complete problems is investigated in this thesis and also published in [1] and [2].

edits involved	running times	papers
1) insert	$O(n+m)$	NA
2) delete	$O(k+m)$	NA
3) swap	$O(n^2)$	NA
4) substitute	$O(n)$	Hamming Distance [9]
5) insert swap	NP-complete	[6], [14], new work here and in [1], [2]
6) insert delete	$O(nm)$	Dynamic Programming [3]
7) insert substitute	$O(nm)$	Wagner Cellar algorithm [16]
8) swap delete	NP-complete	[6], [14], new work here and in [1], [2]
9) swap substitute	$O(nm)$	Wagner Cellar algorithm [16]
10) delete substitute	$O(nm)$	Wagner Cellar algorithm [16]
11) insert swap delete	$O(nm)$	Wagner Cellar algorithm [16]
12) insert swap substitute	$O(nm)$	Wagner Cellar algorithm [16]
13) insert delete substitute	$O(nm)$	Dynamic Programming [3]
14) swap delete substitute	$O(nm)$	Wagner Cellar algorithm [16]
15) insert swap delete substitute	$O(nm)$	Wagner Cellar algorithm [16]

Table 1.1 presents a complete collection of the 15 different non trivial string correction problems over the set of operations swap, delete, substitute, and insert. Only two of the problems listed are classified as having non polynomial running times, namely: (a) string correction permitting swaps and deletes, and (b) string correction permitting swaps and inserts. These cases are shown in rows 5 and 8 of Table 1.1.

A brief overview of the algorithms that solve different string correction problems is discussed below.

insertions only: (Row 1) If only insertions are permitted in transforming X to Y, iterate over both X and Y simultaneously, and for each time that $x_i \neq y_i$, insert

symbol y_i into the ith position of X, thereby shifting each x_j to the right, for $i \leq j \leq n$. Note that a string correction of this type is only possible if X is a subsequence of Y, and if so, X will be supplemented with exactly $|Y| - |X|$ symbols of Y.

deletions only: (Row 2) If only deletions are permitted in transforming X to Y, iterate over both X and Y simultaneously, and for each time that $x_i \neq y_i$, delete symbol y_i from the ith position of X, thereby shifting each x_j to the left, for $i + 1 \leq j \leq n$. Note that a string correction of this type is only possible if Y is a subsequence of X, and if so, exactly $|X| - |Y|$ will be removed from X. The implementation of this case is discussed in further details in Chapter 5.

Based on the above, it is clear that string correction problem permitting only insertions and the one permitting only deletions are inverse in nature. Thus, they can be considered equivalent problems because an algorithm which solves the former can be used to solve the latter by exchanging the source and destination string, and vice versa. For the remainder of this thesis, string correction allowing only insertions and string correction allowing only deletions are considered the same problem.

swaps only: (Row 3) The swaps only string correction problem requires that $|X| = |Y|$ and that each symbol of the character alphabet, Σ, must have the same number of occurrences in both strings. Given that these two conditions are satisfied, string correction involving swaps only can be achieved by repeating the following process until both X and Y have no remaining characters. Locate the first occurrence of y_1 in X, let this be symbol x_j. Swap x_j to the head position of X and then remove x_1 and y_1 from both strings. This case is discussed in further details in Chapter 4.

substitutions only: (Row 4) If only substitutions are permitted in transforming X to Y, then it is required that $|X| = |Y|$. The problem can be solved by substituting each x_i with y_i in the event that $x_i \neq y_i$. The number of substitutions equals the Hamming distance for X and Y [10].

insertions and deletions; insertions, deletions and substitutions: (Rows 6, 13) Dynamic programming is used to solve string correction when the permitted edits are insertions and deletions, or insertions, deletions and substitutions [3].

With the exception of string correction involving swaps and deletes (and therefore the case permitting swaps and inserts), the remaining string correction problems (Rows 7, $9-12$, 14, 15) are all solvable using Wagner's Cellar algorithm [16].

String correction allowing all four edit operations is called the Damerau-Levenshtein distance or Extended String to String Correction, and was first introduced in 1975 by Wagner and Lowrance [15]. A *parameterized* string correction decision problem asks whether such a transformation is possible in at most k edit operations, where the number of edits is the parameter. The parameterized subproblem of the Damerau-Levenshtein distance concerning only the delete and swap edit operations is referred to as the STRING-TO-STRING CORRECTION problem [2]. Formally, it can be defined as follows. Let the set of permitted edit operations be constrained to include deletions and swaps. Consider an alphabet Σ and two strings, X and $Y \in \Sigma^*$, where Σ^* denotes the set of all finite strings over Σ. Then the STRING-TO-STRING CORRECTION decision problem determines whether it is possible to transform X into Y using at most k edits, where $k \in \mathbb{N}$. In [16], it has been shown that the STRING-TO-STRING CORRECTION decision problem is NP-complete via reduction from the MINIMUM SET COVER problem.

As outlined in Table 1.1, previous work has been done concerning all edit operation combinations with the exception of the STRING-TO-STRING CORRECTION decision problem. This thesis investigates the parameterized complexity of the string correction problem involving only deletes and swaps. We present the first fixed parameter tractable (fpt) algorithm, *Algorithm S2S*, for solving the STRING-TO-STRING CORRECTION decision problem [2]. The development of *Algorithm S2S* completes Table 1.1 by determining a solution for string correction involving only swaps and deletes. *Algorithm S2S* includes a series of preprocessing steps (also referred to as reductions rules), that assist in identifying input instances that exhibit specific characteristics used to classify $[X,\ Y,\ k]$ as a yes or a no in polynomial time. Further, we modify the algorithm to include the steps necessary to construct the sequence of edit operations that transform X into Y. It is also shown that the theoretical running time of *Algorithm S2S* is in $O(2^k(k+m))$, proving that for parameter k, STRING-TO-STRING CORRECTION is a member of the parameterized complexity class FPT. Results obtained from the implementation and execution of *Algorithm S2S* indicate that in many cases, due to the reduction rules *Algorithm S2S* is able to determine

the outcome for $[X,\ Y,\ k]$ in $O(k+m)$.

This thesis is structured as follows. In Chapter 2, we introduce by example the STRING-TO-STRING CORRECTION decision problem. An overview of fixed parameter tractability is given in Chapter 3. The parameterized planar INDEPENDENT SET decision problem (ISDP) [6] is used as a case study to illustrate the behaviour of theoretical run times associated with fixed parameter tractable algorithms in comparison to run times that do not take advantage of parameter k. In Chapter 4 we present *Algorithm S2S*, a new fpt algorithm for solving the STRING-TO-STRING CORRECTION decision problem. Proof of correctness of *Algorithm S2S*, as well as several examples, are also provided. In the latter half of Chapter 4 the computational complexity of *Algorithm S2S* is presented. Chapter 5 discusses implementation of *Algorithm S2S*, including construction of the transformation sequence from X to Y as determined by *Algorithm S2S*. Practical running time results are also presented and analyzed. We conclude this thesis with a discussion of potential future work surrounding the STRING-TO-STRING CORRECTION decision problem.

Chapter 2

The STRING-TO-STRING CORRECTION Problem

The STRING-TO-STRING CORRECTION problem can be informally introduced with the following hypothetical illustrative example. Suppose a molecular genetics laboratory is studying the effects of a newly discovered radioactive element when applied to genome sequences of different species. Recent research shows that prolonged exposure, typically a month duration, can cause the occurrence of one of two possible genetic effects. Either the radioactive emissions can disrupt gene ordering by completely eliminating single gene instances, or by exciting the genes causing adjacent genes to switch position. Each of these effects occurs with equal likelihood. Given two genome sequences, genome X and genome Y, the question that researchers are trying to answer is: *If exposed to the radioactive material for at most some fixed number of months, is it possible for genome X to mutate into genome Y?*

The problem posed above in the context of an application to the pertubation of genes is analogous to the theoretical STRING-TO-STRING CORRECTION decision problem. More formally, the STRING-TO-STRING CORRECTION decision problem can be described as follows. Let Σ^* denote a set of all finite strings over the alphabet Σ. Consider a non-negative integer k, and two strings, X and $Y \in \Sigma^*$. The goal is to determine whether there exists a derivation from X to Y using a sequence of at most k edit operations, where an edit operation is defined as either an adjacent symbol interchange (swap or transposition) or a single symbol deletion (delete). A swap occurs when two consecutive symbols switch position, as shown in Figure 2.1.

The number of occurrences of each symbol is preserved during string permutation, thus the length of a string remains unchanged after the swap operation is applied. Deletion is the removal of an individual instance of an element from the given string, therefore shortening a string of length n to length $n-1$ [6]. An example of the delete edit operation is illustrated in Figure 2.2.

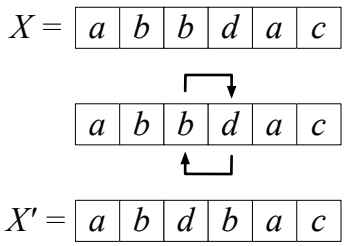

Figure 2.1: Swapping the third and fourth symbols, b and d, of $X = abbdac$. The length of the resulting string, $X' = abdbac$, remains unchanged.

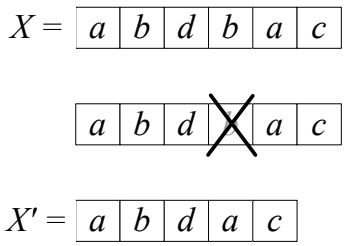

Figure 2.2: Deleting the fourth element from $X = abdbac$. The delete operation shortens the length of the given string by one, $X' = abdac$.

The following example illustrates the STRING-TO-STRING CORRECTION decision problem. Let $\Sigma = \{a, b, c\}$, then Σ^* is the set of all finite strings over the symbols a, b and c. Consider two strings $X = abcbcc$ and $Y = bacb$, for X and $Y \in \Sigma^*$ and the integer $k = 5$, denoting the maximum number of permitted edit operations. STRING-TO-STRING CORRECTION from X to Y can be achieved by swapping the first two symbols of X followed by deleting the last two symbols of X, as shown in Figure 2.3. This specific transformation requires 3 edit operations, namely two deletions and one swap. Because X can transform into Y using at most 5 edit operations, the STRING-TO-STRING CORRECTION from X to Y for value k is indeed possible. Now consider again X and Y as above, but let $k = 1$. STRING-TO-STRING CORRECTION from string X into string Y is not possible for the specified parameter, $k = 1$, as not enough edit operations are permitted for the transformation to occur.

In the previous example, the order in which the edit operations are executed is irrelevant. However, in general, the delete and swap edit operations cannot be regarded as commutative operations. For example, consider $X = abcbcc$ after the deletion of the first symbol followed by the transposition of the first two symbols. The

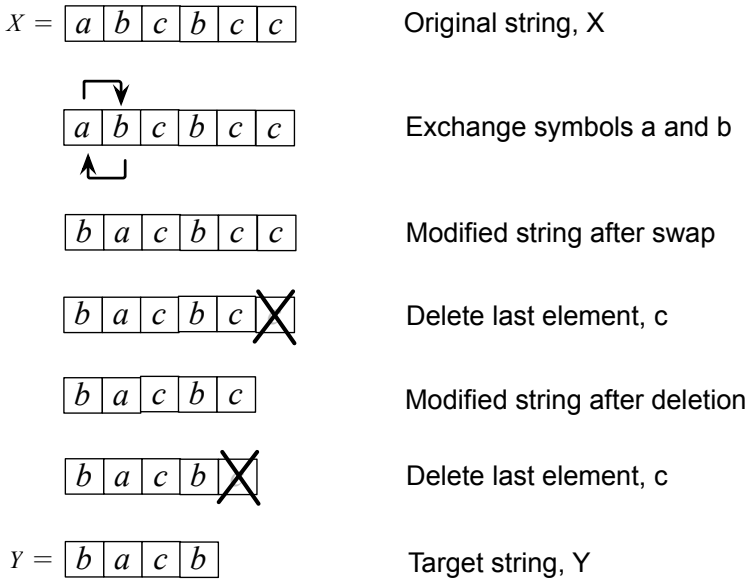

Figure 2.3: An example of the STRING-TO-STRING CORRECTION problem with inputs $X = abcbcc$, $Y = bacb$ and parameter $k = 5$. A transformation from X to Y can be achieved by swapping the first two symbols of X and then deleting the last two symbols of X. The resulting string, Y, is two elements shorter than the original string X due to the deletions performed during the string correction process.

initial deletion yields $X' = bcbcc$ by removing a to leave b as the new leading symbol in the string. Then swapping the first two symbols of X results in $X'' = cbbcc$. If this same sequence of edit operations is applied to X in reverse order, the initial swap gives $X' = bacbcc$ and the following deletion gives $X'' = acbcc$.

2.1 Notation and Terminology

the Let X and Y be two strings composed of characters from a common alphabet, Σ, that is, $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_m$, each x_i, $y_j \in \Sigma$ where $1 \leq i \leq n$ and $1 \leq j \leq m$. An instance of the STRING-TO-STRING CORRECTION problem is expressed by the ordered triple $[X, Y, k]$, where k denotes the number of permitted edits. Furthermore, any $[X, Y, k]$ for which there exists at least one transformation from X to Y using at most k edit operations, is referred to as a YES-instance. Otherwise, the ordered triple is a NO-instance. Notice that as insertions are not permitted during STRING-TO-STRING CORRECTION, any instance $[X, Y, k]$ with $|X| < |Y|$ is a NO-instance, and subsequently $n \geq m$ for every YES-instance.

We define the following functions for a given string $X \in \Sigma^*$.

- Let $\phi(X, a)$ denote the index of the first occurrence of symbol a in X. Then, if $\phi(X, a) = i$, X is of the form $x_1 x_2 \ldots x_{i-1} a x_{i+1} \ldots x_n$, with $x_j \neq a$ for $1 \leq j \leq i-1$.

- Let the tail function, $\tau(X)$, represent X after the symbol x_1 has been removed. Then $\tau(X) = x_2 x_3 \ldots x_n = x'_1 x'_2 \ldots x'_{n-1}$.

- Let the substring of X which results from deleting symbol x_i from X be denoted by $\delta(X, i)$. Thus, $\delta(X, i) = x_1 x_2 \ldots x_{i-1} x_{i+1} \ldots x_n = x'_1 x'_2 \ldots x'_{n-1}$.

- Let $\sigma(X, i)$ denote the string which results from X after swapping symbols x_{i-1} and x_i. Subsequent to application of σ, $\sigma(X, i) = x_1 x_2 \ldots x_{i-2} x_i x_{i-1} x_{i+1} \ldots x_n$.

Observe that $\tau(X) = \delta(X, 1)$ and $|\tau(X)| = |\delta(X, i)| = |X| - 1$. Note, however, that the length of X is preserved by function σ.

To illustrate the behaviour of these functions, consider the following examples with $X = aabccb$. Then $\phi(X, a) = 1$ and $\phi(X, c) = 4$. The tail function applied to X yields $\tau(X) = abccb$ with $|\tau(X)| = 5$ (see Figure 2.4 and Figure 2.5, respectively). $\delta(X, 4) = aabcb$ with $|\delta(X, 4)| = 5$ and $\sigma(X, 3) = abaccb$ with $|\sigma(X, 3)| = 6$ (see Figure 2.6 and Figure 2.7).

Let a *transformation sequence* ω be defined by $\omega = \omega_1 \cdot \omega_2 \cdot \ldots \cdot \omega_w$ where $\omega_i \in \{\tau, \sigma, \delta\}$ for $1 \leq i \leq w$. The length of ω, $|\omega|$, is strictly defined in terms of the number of occurrences of δ and σ, that is, the occurrences of τ are excluded from ω during length calculation. Let $\omega(X, Y)$ denote a transformation from X to Y. Observe that a given instance $[X, Y, k]$ is a YES-instance if there exists an $\omega(X, Y)$ with $|\omega| \leq k$. Extending the example above, if $X = aabccb$ and $Y = acbb$, then the application of the transformation sequence $\omega = \omega_1 \cdot \omega_2 \cdot \omega_3 = \tau(X) \cdot \sigma(X, 3) \cdot \delta(X, 4)$ yields $\delta(\sigma(\tau(X), 3), 4) = acbb = Y$. In this example, $|\omega| = 2$. Figure 2.8 illustrates the application of $\omega(X, Y)$ to $X = aabccb$.

Finally, X is a *supersequence* of Y provided that X can be constructed from Y just by inserting additional symbols, each from Σ, at different locations of Y. Whenever a symbol is to be inserted at index i, the index j of each x_j with $i \leq j \leq n$ is

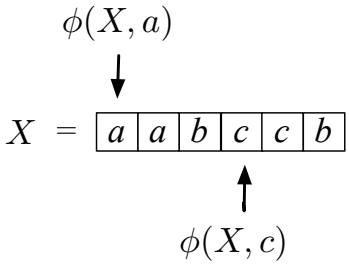

Figure 2.4: The original X. The first occurrence of a in X is $\phi(X, a) = 1$ and $\phi(X, c) = 4$.

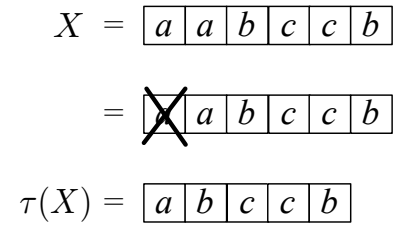

Figure 2.5: The result of the tail function when applied to X, i.e, $\tau(aabccb) = abccb$. The length of X is decreased by one during application of function τ.

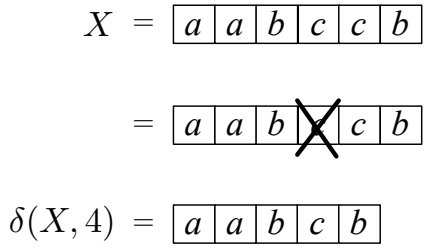

Figure 2.6: The original X, and the resulting string after symbol in the 4th position has been deleted. Initially, $|X| = 6$, however, the resulting string, $\delta(X, 4)$, is of length 5.

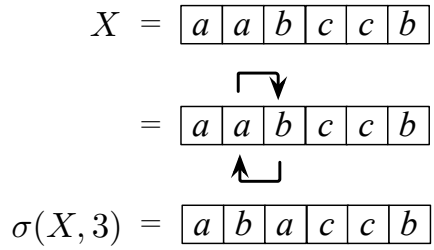

Figure 2.7: The original X, and the resulting string after the symbol in the 3rd position is swapped one position to the left, with the symbol occupying index 2. Note that $|X| = 6 = |\sigma(X, 3)|$.

increased by 1. This corresponds to shifting x_j to the right by one position to make space for the symbol to be inserted. The length of a string is increased by one for each newly inserted symbol. Furthermore, Y is called a *subsequence* of X if and only if X is a supersequence of Y.

The following example demonstrates the existence of subsequence-supersequence relationship between two strings, X and Y. Let $X = aabdcc$ and $Y = bcc$. X can be constructed from Y by prepending aa, and inserting the symbol d between the b and c, see Figure 2.9. Therefore, X is a supersequence of Y. In a similar manner, Y is constructed from X by deleting the two leading a symbols as well as the d, and thus, Y is a subsequence of X. Note that $X = aabdcc$ is not a supersequence of $Y = ccb$

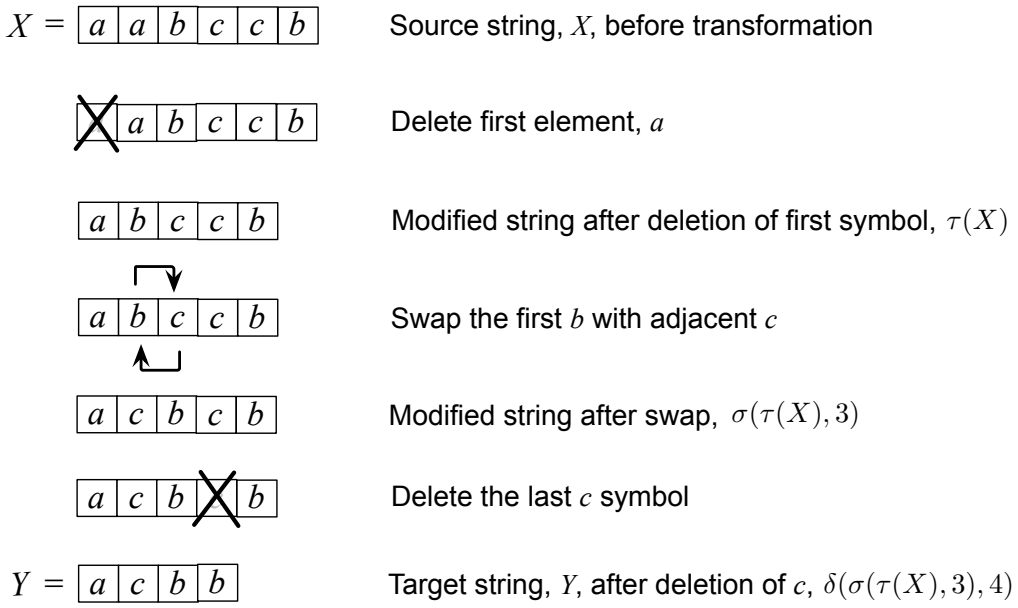

Figure 2.8: The application of $\omega = \delta(\sigma(\tau(X), 3), 4)$ to $X = aabccb$. ω transforms X into $Y = acbb$.

as X cannot be constructed from Y using only insertions. The concepts of super and subsequences are used in the new STRING-TO-STRING CORRECTION algorithm presented in Chapter 4.

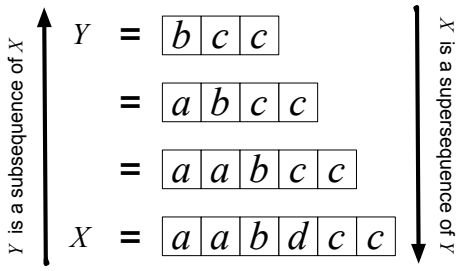

Figure 2.9: The left arrow indicates that Y is a subsequence of X because it can be constructed from X by deleting the two a's at the beginning of X as well as the d between the b and c. Similarly, X is a supersequence of Y because it can be constructed from Y by prepending two a's and inserting a d in between the b and c. This is illustrated by the right arrow originating at Y and finishing at X.

2.2 Chapter Summary

This chapter provided an introduction to the STRING-TO-STRING CORRECTION problem. Specifically, it discussed the string correction problem with the edit operation set restricted to swaps and deletes. Chapter 4 presents a new algorithm which solves such a problem. The new STRING-TO-STRING CORRECTION algorithm exploits the parameterized nature of the given problem by integrating the parameter as a limiting or terminating factor. Thus, the STRING-TO-STRING CORRECTION problem belongs to the FPT complexity class. Chapter 3 provides a basic introduction to the FPT complexity class.

Chapter 3

Fixed Parameter Tractable Algorithms

The study of classical complexity theory deals with the analysis and classification of computational problems according to the amount of resources needed in order for a solution to be attained. The standard metrics used to measure the resources required are the time it takes to run an algorithm as well as the memory space consumed by an algorithm. The measurements are expressed as a function of the problem input size, denoted as n [7]. A problem which executes in polynomial time is called *tractable*, whereas a problem requiring a likely non-polynomial amount of time is called *intractable* [7]. Much work has been done in the area of classical complexity theory, dividing the field into rigid classification categories. However, beyond the classical domain there are several different aspects of a problem which can be used as alternative or supplementary metrics. Information provided by the additional metrics can be used throughout the algorithm development process to tailor a solution with a faster running time than solutions which do not consider these metrics. Our work is situated on the framework of parameterized complexity [4].

This chapter introduces a complexity class called FPT, the class of parameterized problems that are Fixed-Parameter Tractable, in which computational complexity measurements incorporate both the problem input size as well as the additional input parameter, denoted by k [13].

In order to best explain FPT, it is useful to start with an example, making the

process toward a solution clearer. The example considered is taken from graph theory, which has a rich set of problems.

In the following example, the Independent Set Decision Problem (ISDP) on planar graphs[1] is solved using two different algorithms. The first algorithm provides a solution which has an exponential running time based on the size of the input graph. The design of the second algorithm yields a running time which is exponential only in the parameter k, resulting in a faster running time. This difference in efficiency is highlighted by comparing the worst case running times of each algorithm when provided with identical input.

Several graph theory problems are concerned with determining the existence of a certain type of vertex subset of a given graph. Consider ISDP on planar graphs with input $G = (V, E)$ and a non-negative integer k. The problem input size, n, is the size of the graph as measured by the number of vertices, that is, $|V| = n$. The task is to determine the existence of a set of vertices $C \subseteq V$ of size at least k such that each edge in the graph has at most one endpoint in C [6]. A set satisfying these requirements is called an *independent set (IS)*.

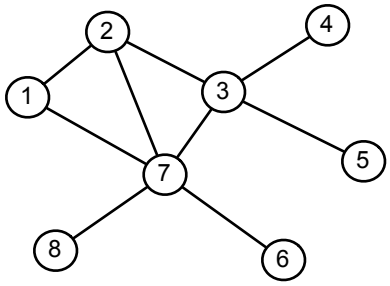

Figure 3.1: The graph G composed of 8 vertices and 9 edges.

As an example, consider the graph G composed of eight vertices, labeled 1 through 8, with the edge set $E = \{(1,2),(2,3),(3,4),(3,5),(1,7),(2,7),(3,7),(6,7),(7,8)\}$, as shown in Figure 3.1. The vertex set $C_1 = \{1,4,6\}$ is an IS of size 3. The vertex set $C_2 = \{1,4,7\}$ is not an IS because the edge $(1,7)$ has both of its endpoints in C_2. See Figures 3.2 and 3.3.

[1]Recall that a planar graph is defined as a graph which can be drawn in R^2 such that no two edges intersect [8].

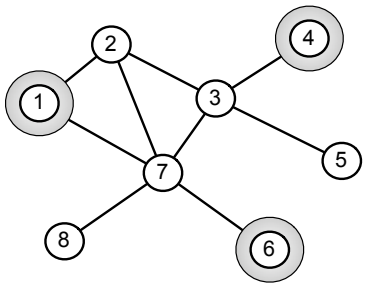

Figure 3.2: For the graph G, as shown in Figure 3.1, the vertex set $C_1 = \{1, 4, 6\}$ is an IS of size 3.

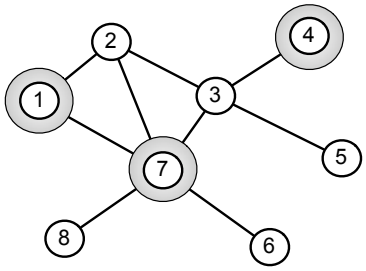

Figure 3.3: For the graph G, as shown in Figure 3.1, the vertex set $C_2 = \{1, 4, 7\}$ does not form an IS because the edge $(1, 7)$ has both of its endpoints contained in C_2.

The first solution uses a brute force approach based on constructing a complete binary search tree. It can be summarized as follows. Select an arbitrary vertex x from V. A binary decision regarding the inclusion or exclusion of x to the candidate IS directs the construction of a binary search tree, where a node represents the selected vertex and an edge rooted at that node indicates that either x is or is not a member of the candidate IS. For each vertex in V, our search tree has two branches. One that corresponds to its inclusion into the candidate IS and one that corresponds to its exclusion from the candidate IS. This results in a complete binary search tree of size $2^{n+1} - 1$, height n, and consisting of 2^n potential solution paths.[2,3] For this algorithm, only internal nodes[4] of the binary search tree represent vertices selected from G. Each leaf node represents the set of vertices that corresponds to the branch that is terminated by that particular leaf node. The leaves are included in the count towards the tree's size. The tree represents all candidate IS solutions for the given input. Traversing a direct path from the root to a leaf represents a single candidate solution, each of which is tested to see whether it provides an IS of size k or greater for the given G. The algorithm terminates successfully upon encountering the first IS of size k, that is a set C with at least k vertices such that no two vertices are adjacent. If no path in the binary search tree provides a solution, the algorithm terminates

[2]Recall that the height of a rooted tree $T = \{V, E\}$ is defined as the largest level number corresponding to a leaf of T. The tree's root is at level 0 [8].

[3]We define the size of a tree $T = \{V, E\}$ to be the number of nodes from which its composed, that is, $|V|$. This definition includes leaf nodes.

[4]The set of internal nodes, or *branch node*, belonging to a tree is defined as the set of all non terminal vertices belong to that tree [8].

unsuccessfully.

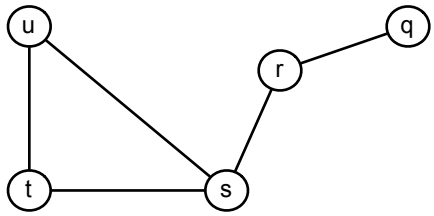

Figure 3.4: The graph G, used as input to the brute force algorithm that solves the planar ISDP. Figure 3.5 illustrates a complete binary search tree providing all potential solutions to planar ISDP for $k = 2$.

The complete process is shown in Figure 3.5, as applied to the graph depicted in Figure 3.4. The dotted path represents a single candidate IS. For $k = 2$, the following vertex sets satisfy planar ISDP: $C_1 = \{u, r\}$, $C_2 = \{u, q\}$, $C_3 = \{t, r\}$, $C_4 = \{t, q\}$, and $C_5 = \{s, q\}$. Note that the complete binary search tree constructed by this algorithm is the same regardless of the $k-$value or edge set of the input graph.

Analyzing the computational complexity of this algorithm measured as a function of the input instance size, i.e. $|V|$, results in a non polynomial time complexity of $O(2^{n+1})$. Note that the exponential behaviour is a function of the graph size.

The purpose of introducing a parameter is to redirect the explosive combinatorial behaviour such that it depends solely on the parameter and is uninfluenced by n [4]. If this behavioural shift is achieved, the input size only affects the algorithmic complexity on a polynomial scale. Consequently, the parameterized problem can then be solved efficiently, provided that the value of k remains small. Thus, by using fixed parameterization, many problems which are classically categorized as intractable can be reassessed as tractable. The transfer of exponential growth behaviour between input size and parameter is the central concept behind an alternate complexity theory known as *parameterized complexity* [5]. Formally, a parameterized problem is classified as *fixed parameter tractable* if its running time can be expressed in the form $f(k)n^{O(1)}$ or $f(k) + n^{O(1)}$, where $f(k)$ is a computable function concerning only the parameter [13]. The set of such parameterized problems forms the FPT complexity class [13]. We denote an algorithm solving a parameterized problem with a fixed

parameterized running time as an *FPT algorithm*.

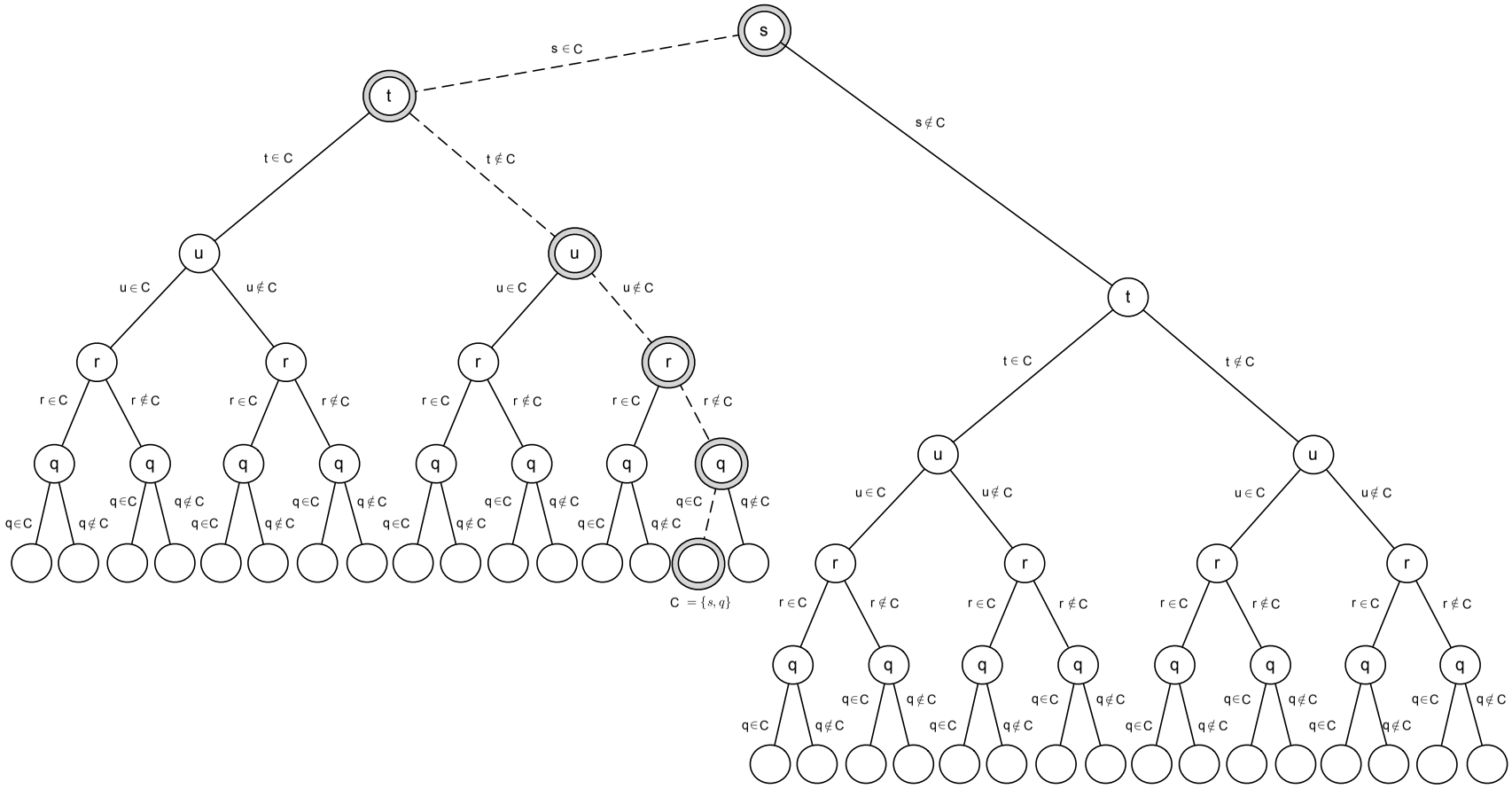

Figure 3.5: The complete binary search tree resulting from the brute force algorithm for solving the planar ISDP for graph G of Figure 3.4 and $k = 2$. The dotted path highlights a candidate IS solution containing vertices s and q. The binary search tree has height $n = 5$ and contains $2^{n+1} - 1 = 63$ nodes. It consists of $2^n = 32$ candidate solutions.

Expanding the algorithmic design and computational analysis of planar ISDP to include a fixed parameter allows for complexity analysis and measurements to occur on a more refined scale than performing measurements involving only the problem input size, as attention can be directed to specific characteristics of the problem. The following FPT algorithm exemplifies this notion. The parameter is chosen to represent the cardinality of the IS, that is k. We start by introducing a known property of loop free connected planar graphs which is used in our FPT algorithm for solving planar ISDP.

Property 1. *Every loop free connected planar graph contains at least one vertex with degree of at most 6 [8].*

Consider now a different algorithmic solution for planar ISDP, designed to accommodate the parameter k. Select a vertex x with degree at most 6 from the given graph. The existence of such a vertex is guaranteed by Property 1. The chosen vertex and its neighbours form a set of size at most 7. Label these vertices x_i for $1 \leq i \leq 7$. We create a non-binary search tree where each search tree node has at most 7 children. A node in the tree contains the graph structure G, the parameter k, and the selected vertex x. The root node, for example, contains the original graph, the original parameter, and the first vertex selected from the graph. We denote the closed neighbourhood of $x \in G$ with $N(x)$ which contains all vertices adjacent to x and x itself. Each edge in the bounded search tree represents the inclusion of a vertex w in $N(x)$ to the candidate IS. G is modified to G' by removing w and all adjacent vertices from G. The parameter is decremented to accommodate the inclusion of w into the candidate IS, that is $k' = k - 1$. The result is the graph-parameter pair (G', k') on which the algorithm recurses. A branch in the bounded search tree terminates if the graph to be recursed on (G'), equals the empty graph and $k' = 0$. In this case, the corresponding candidate IS is not of the required size, and is therefore discarded. If each candidate IS in the bounded search tree terminates under these circumstances, the algorithm aborts classifying (G, k) as a NO-instance for planar ISDP. The algorithm terminates classifying (G, k) as a YES-instance upon encountering the first candidate IS with height of k. In this way a search tree with size at most $7^{k+1} - 1$ and height at most k is constructed. Similar to the brute force algorithm for planar ISDP, each path in the bounded search tree from root to leaf represents a candidate IS. Each candidate IS is tested to see if it represents a YES-instance to planar ISDP. Testing occurs until either a YES-instance is found or all candidate solutions have

been tested. The running time of this algorithm is $O(7^k n)$ and thus, planar ISDP is in FPT for parameter k.

To clarify the Planar ISDP FPT algorithm presented above, we consider the following example in which planar ISDP for $k = 2$ is solved on the graph G of Figure 3.4. Steps towards the construction of the bounded search tree are as follows. The complete bounded search tree is presented in Figure 3.6. Given the graph G, let s be the first vertex selected, thus, the root node of the bounded search tree contains $(G, 2)$, and s. Vertex s and its neighbours form the set $N_0 = \{s, r, t, u\}$. Let $x_1 = s$, $x_2 = r$, $x_3 = t$, and $x_4 = u$. For each $x_i \in N_0$, remove x_i and its neighbours from G and connect an edge to the root node indicating that x_i is a member of the candidate IS associated with that edge. In this example, 4 branches extend from the root node, one for each of s, t, u, and r. The left most edge of the bounded search tree in Figure 3.6 indicates that s has been added to the candidate IS associated with that edge and that k has been decremented by one, resulting in $k_1 = 1$. Vertex s, its neighbour set, and all incident edges are removed from G, forming G_1 (See G_1 bottom center on Figure 3.6). The algorithm recurses on (G_1, k_1).

This process is also carried out for each of the remaining three edges incident to the root node of the bounded search tree. These edges, labeled $t \in C$, $u \in C$, and $r \in C$, appear between the 0th and first level of the bounded search tree in Figure 3.6.[5] Removal of each vertex and its neighbour set from G results in the graph-parameter pairs: (G_2, k_2), (G_3, k_3), and (G_4, k_4), where $k_i = 1$ for $1 \leq i \leq 4$. Each of G_i, for $1 \leq i \leq 4$, is non empty and $k_i = 1$ for $1 \leq i \leq 4$, so none of the candidate IS's are terminated and the algorithm recurses on each pair. Graphs G and $G_1 - G_4$ are shown below the bounded search tree in Figure 3.6. In the next recursive call, G_1 is composed of the single vertex, q, so q is selected from G_1. Vertex q and its (empty) neighbour set, are removed from G_1, creating the empty graph G_5. k_1 is decremented resulting in $k_5 = 0$. G_5 is the empty graph and k_1 is decremented resulting in $k_5 = 0$. The candidate IS $C = \{q, s\}$ is an IS of size k. Thus algorithm terminates classifying (G, k) as a YES-instance for planar ISDP.

[5]Recall that the *depth* of a node is defined as the length of the path from the that node to the root. A *level* of a tree refers to the set of all nodes at a given depth. The root node is at depth zero. [8]

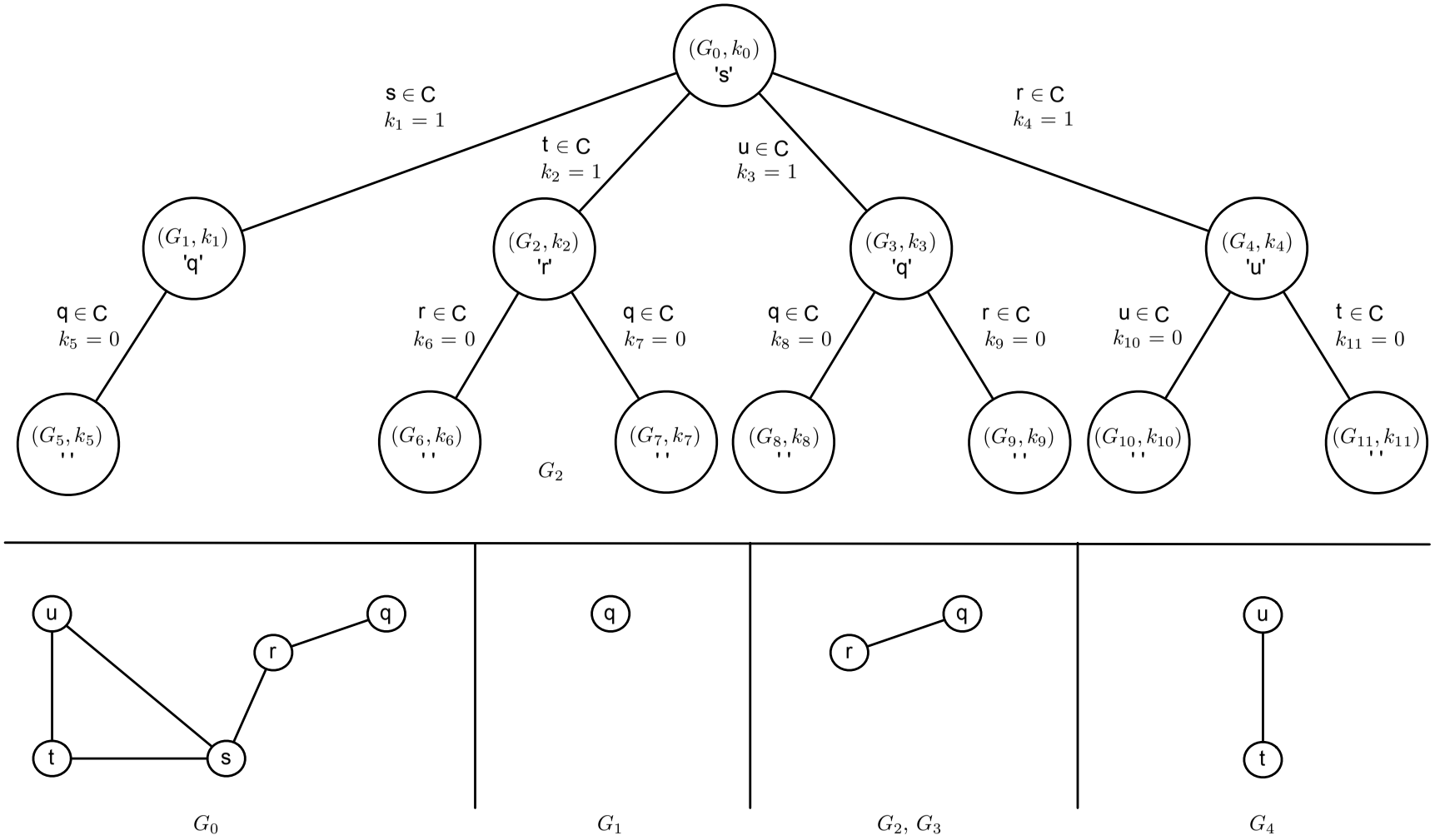

Figure 3.6: bounded search tree resulting from application of the FPT algorithmic solution on the graph G in Figure 3.4. The bounded search tree has height $k = 2$, 12 nodes and 7 candidate solution paths. The graphs resulting from the removal of vertex x_i, $N(x_i)$ and all incident edges, from the parent graph are shown below the bounded search tree. Graphs G_5 through G_{11} are equal to the empty graph.

To illustrate the significant decrease in running time achieved by the FPT algorithm for planar ISDP, we consider a graph G with $|V| = n$ and we compare the worst case running times attained by the brute force algorithm against the FPT algorithm for values $k = 2$ and $k = 3$, as summarized in Table 3.1. For the brute force algorithm, the worst case running time is attained when every path in the binary search tree is tested before determining if (G, k) is a YES-instance or a NO-instance. This situation can occur when either there does not exists an IS of size k, or the entire set of candidate IS's is tested before one of size k is found. In these cases, the brute force algorithm constructs a binary search tree of height n composed of $2^{n+1} - 1$ nodes. The algorithm's running time is in $O(2^n)$. For the FPT algorithm, the worst case running time occurs when (G, k) is a YES-instance. In this case, a bounded search tree of height k consisting of $7^{k+1} - 1$ nodes is built. The FPT algorithm has a running time in $O(7^k n)$.

Table 3.1: Running Time Comparisons of Algorithm BF (columns 2 and 4) and Algorithm FPT (columns 3 and 5) for solving the planar ISDP for $k = 2$ and $k = 3$ on graph G. All times are given in seconds.

n	$k = 2$, 2^{n+1}	$k = 2$, $7^{k+1}n$	$k = 3$, 2^{n+1}	$k = 3$, $7^{k+1}n$
1	4	343	4	2401
2	8	686	8	4802
3	16	1029	16	7203
4	32	1372	32	9604
5	64	1715	64	12005
6	128	2058	128	14406
7	256	2401	256	16807
8	512	2744	512	19208
9	1024	3087	1024	21609
10	2048	3430	2048	24010
11	4096	3773	4096	26411
12	8192	4116	8192	28812
13	16384	4459	16384	31213
14	32768	4802	32768	33614
15	65536	5145	65536	36015
20	2097152	6860	2097152	48020
50	2251799813685248	17150	2251799813685248	120050

We examine first the the worst case running times attained by both planar ISDP algorithms for $k = 2$. Table 3.1 indicates that for a graph with $|V| \leq 10$, planar ISDP with $k = 2$ is solved more efficiently by the brute force algorithm than the FPT algorithm. However, it can also be observed that for $k = 2$ the FPT algorithm results in a faster running time on all graphs containing greater than 10 nodes. As the size of the input graph increases, the running time of the brute force algorithm increases exponentially, yet the running time of the FPT algorithm increases linearly. This is because the exponential component of the FPT algorithm's running time depends solely on the parameter value and is unaffected by the size of the input graph. The benefits of shifting the exponential behaviour such that it is affected only by the parameter can be seen most clearly for problem instances with a large input graph and a small parameter value. For example, planar ISDP with $k = 2$ on a graph composed of 50 nodes has a worst case running time of $2251799813685248s$ when solved using the brute force algorithm, whereas the worst case running time attained when the FPT algorithm is applied to the same instance is $17150s$.

Similar growth trends in running time of the two algorithms can be observed for planar ISDP with $k = 3$ (see the two rightmost columns of Table 3.1). For an arbitrary graph composed of 14 nodes or less, the brute force algorithm performs faster than the FPT algorithm. However, input graphs containing a minimum of 15 nodes are solved faster by the FPT algorithm.

This chapter introduced the reader to the complexity class called FPT and, using planar ISDP, demonstrated the decrease in theoretical running time achievable by use of FPT algorithms. Chapter 4 presents our new FPT algorithm that solves the STRING-TO-STRING CORRECTION decision problem.

Chapter 4

A New FPT Algorithm for Solving the STRING-TO-STRING CORRECTION Problem

In Chapter 3, the reader is introduced to the definition as well as some basic characteristics of the complexity class FPT using Planar ISDP as a case study. In this chapter, a new FPT algorithm that solves the STRING-TO-STRING CORRECTION decision problem using the bounded search tree approach is presented. The STRING-TO-STRING CORRECTION decision problem asks whether a transformation from X to Y requiring at most k edit operations is possible, but is unconcerned with the construction of such a transformation. Here, we provide an algorithmic solution that solves the STRING-TO-STRING CORRECTION decision problem by classifying the input instance to be either a YES-instance or a NO-instance. An algorithm that outputs the sequence of edit steps as a transformation sequence is also presented in Chapter 5.

Section 4.1 presents a collection of theorems and lemmas that act as a foundational framework for our new recursive algorithm for STRING-TO-STRING CORRECTION. The algorithm presented, *Algorithm S2S*, is divided into two phases, a preprocessing phase in which a series of reduction rules are applied to the problem instance, and a branching phase which explores the application of both deletes and swaps to the instance. A reduction rule is defined as a rule that either classifies [X, Y, k] as a YES-instance or NO-instance in polynomial time, or reduces the problem

input size by removing the leading character from both X and Y via function τ. A simple proof for each reduction rule follows in Section 4.2.2. Section 4.3 analyzes the running time of *Algorithm S2S* and shows that it runs in $\mathcal{O}(2^k(k+m))$. We have published a preliminary version of the work presented in this chapter in [1] and in [2].

4.1 Supporting Lemmas and Theorems

In this section we present a collection of lemmas as well as our main theorem (see Theorem 1) needed in support of our new solution to the STRING-TO-STRING CORRECTION decision problem (Section 4.2).

Lemma 1. *Let [X, Y, k] be a YES-instance of the STRING-TO-STRING CORRECTION problem and let $\phi(X, y_1) = i$. Then each $x_{i'}$, with $1 \leq i' \leq i-1$, must be deleted or swapped to the right of x_i.*

Proof. Since $\phi(X, y_1) = i$, there does not exist an $x_{i'}$ $(1 \leq i' \leq i-1)$ that is identical to y_1. For x_i to be the first symbol of X, no symbols can precede it, and thus, each $x_{i'}$ must be removed from the prefix $x_1 \ldots x_i$. Removal of $x_{i'}$ can be achieved by deleting $x_{i'}$ from X, or by swapping $x_{i'}$ to the right of x_i. If $x_{i'}$ is deleted from X, it no longer appears before x_i. $x_{i'}$ cannot remain in the same position and it would be beneficial to swap it to the left, as it is not equal to y_1. Thus, in order for x_i to become the first symbol of X, each $x_{i'}$ must be either deleted from X or swapped to the right of x_i. □

Lemma 2. *Let [X, Y, k] be a YES-instance for the STRING-TO-STRING CORRECTION decision problem. Then there exists a transformation sequence $\omega(X, Y)$ with $|\omega(X, Y)| \leq k$ in which all deletions appear before any swaps.*

Proof. (By Contradiction) Assume that no such transformation sequence, $\omega(X, Y)$, exists. Then in each $\omega(X, Y)$, there exists at least one instance of σ before a δ. Consider a shortest transformation sequence, $\omega'(X, Y)$. Let ω'_i be the first swap of $\omega'(X, Y)$ and let ω'_j be the first delete preceded by ω'_i. There are two cases to consider, either the symbol to be deleted by ω'_j is involved in ω'_i or not.

Case 1: Suppose that the symbol to be deleted by ω'_j is not one of the symbols that is involved in ω'_i. Then ω'_j is independent of all preceding swaps, including ω'_i.

Moving ω'_j before all preceding swaps produces $\omega''(X, Y)$, a new sequence of edit operations that transforms X into Y. The order of the elements in $\omega'(X, Y)$ is permuted, i.e., no element is added or removed, so $|\omega''(X, Y)| = |\omega'(X, Y)|$. Thus, $\omega''(X, Y)$ is a solution with $|\omega''(X, Y)| \leq k$, in which all deletions appear before any swaps, contradicting the assumption that no such transformation sequence exists.

Case 2: Suppose, on the other hand, that the symbol to be deleted by ω'_j is involved in the swap ω'_i. Then, in $\omega'(X, Y)$, a target symbol is both swapped and deleted. Removing the swap from $\omega'(X, Y)$ to create $\omega''(X, Y)$ yields a shorter transformation with the same behavior. This new $\omega''(X, Y)$ is shorter than $\omega'(X, Y)$, contradicting the assumption that $\omega'(X, Y)$ is the shortest transformation sequence. □

Corollary 1. *Let [X, Y, k] be a YES-instance of the STRING-TO-STRING CORRECTION decision problem. If $\omega(X, Y)$ is a shortest solution, then no element of X is both swapped and deleted.*

Lemma 3. *If ω is a shortest transformation sequence solving the STRING-TO-STRING CORRECTION decision problem for [X, Y, k], then no consecutive identical symbols are ever swapped.*

Proof. (By Contradiction) Suppose that for some instance $[X, Y, k]$ of the STRING-TO-STRING CORRECTION decision problem there exists a shortest transformation sequence $\omega(X, Y)$, in which consecutive identical symbols are swapped. Let $\omega(X, Y) = \omega_1 \cdot \omega_2 \cdot \ldots \cdot \omega_t$, where $|\omega(X, Y)| \leq k$ and $\omega_i \in \{\delta, \sigma, \tau\}$ (*recall that by definition, occurrences of τ do not count toward the length of $\omega(X, Y)$*). Then there exists an ω_i of the form $\omega_i = \sigma(X, j)$ for which $x_j = x_{j-1}$, i.e., an ω that swaps consecutive identical symbols in X. Swapping identical characters has no affect on X. experiments, then $\omega'(X, Y) = \omega(X, Y)$. Since σ contributes to the length of a transformation sequence, $|\omega(X, Y)| - 1 = |\omega(X, Y)'|$. Therefore $\omega'(X, Y)$ is a shorter transformation sequence for the STRING-TO-STRING CORRECTION decision problem of $[X, Y, k]$ than $\omega(X, Y)$, contradicting the assumption that $\omega(X, Y)$ is a shortest solution. □

We say that y_1 is *mapped* to an $x_i \in X$, for $1 \leq i \leq n$ if each $x_{i'}$ for $1 \leq i' < i$, is either deleted or swapped to the right of x_i.

Theorem 1. *If [X, Y, k] is a YES-instance for STRING-TO-STRING CORRECTION, then there exists an optimal solution, $\omega(X, Y)$, where each y_1 is mapped to its first occurrence in X.*

Proof. (By Contradiction) Suppose that there is no shortest transformation sequence ω where y_1 is mapped to its first occurrence in X. Then, if $\phi(X, y_1) = i$, there exists an i' with $i < i'$ such that $x_i = x_{i'}$ to which y_1 is mapped. By Lemma 1, each symbol to the left of $x_{i'}$ must either be deleted or swapped to the right of $x_{i'}$. In particular this applies to x_i. Deleting x_i and keeping $x_{i'}$, or keeping x_i and deleting $x_{i'}$ (since x_i and $x_{i'}$ both match y_1) will yield transformation sequences of equal length if the two symbols are neighbours. In this case, both solutions involving the deletion of either x_i or $x_{i'}$, are shortest solutions. If they are not neighbours, then $x_{i'}$ must be swapped to the left, incurring an extra cost. Thus, deleting $x_{i'}$ and keeping x_i never results in a shorter transformation sequence. If x_i is swapped to the right of $x_{i'}$, identical symbols need to be swapped, which contradicts Lemma 3. Thus, there exists a shortest solution $\omega(X, Y)$ for the STRING-TO-STRING CORRECTION decision problem of $[X, Y, k]$ in which each y_1 is mapped to it's first occurrence in Y. □

Corollary 2. *If $x_1 = y_1$ for some X and Y, then [X, Y, k] is a YES-instance for STRING-TO-STRING CORRECTION if and only if $[\tau(X), \tau(Y), k]$ is a YES-instance for STRING-TO-STRING CORRECTION.*

Proof. We show equivalence in two steps.

Claim 1: *Suppose $x_1 = y_1$. If $[\tau(X), \tau(Y), k]$ is a YES-instance, then [X, Y, k] is a YES-instance.*

Suppose that $x_1 = y_1$ and that $[\tau(X), \tau(Y), k]$ is a YES-instance. Then there exists an $\omega = \omega_1 \cdot \omega_2 \cdot \ldots \cdot \omega_t = \omega_t(\omega_{t-1}(\ldots \omega_1(X) \ldots))$, with $|\omega| \leq k$ and $\omega_i \in \{\tau, \delta, \sigma\}$ that solves the STRING-TO-STRING CORRECTION decision problem from $\tau(X)$ to $\tau(Y)$ for some X and Y. Since $x_1 = y_1$, X and Y are of the form $a\tau(X)$ and $Y = a\tau(Y)$, respectively, for some $a \in \Sigma$. Prepending ω with $\omega_0 = \tau(X)$ results in the transformation sequence $\omega' = \omega_0 \cdot \omega = \omega_0 \cdot \omega_1 \cdot \ldots \cdot \omega_t = \omega_t(\omega_{t-1}(\ldots \omega_1(\omega_0(X)) \ldots))$. Invoking the assignment $\omega'_{i+1} = \omega_i$ results in the transformation $\omega' = \omega'_1 \cdot \omega'_2 \cdot \ldots \cdot \omega'_{t+1}$, which is a transformation sequence from X to Y of length at most k. Recall that τ is not included in the length calculation of a transformation sequence, and thus

$|\omega'| = |\omega| \leq k$. Thus, ω' solves the STRING-TO-STRING CORRECTION problem for $[X,\ Y,\ k]$.

Claim 2: *Suppose* $x_1 = y_1$. *If [X, Y, k] is a YES-instance, it is also true that* $[\tau(X),\ \tau(Y),\ k]$ *is a YES-instance.*

Suppose that $x_1 = y_1$ and that $[X,\ Y,\ k]$ is a YES-instance. By Theorem 1, there exsists an optimal solution in which x_1 is mapped to y_1 as it is the first occurrence of y_1 in X. Therefore, x_1 is not deleted. Thus, y_1 and x_1 can be removed via function τ from X and Y respectively without incurring any extra cost. Thus $[\tau(X), \tau(Y), k]$ is a YES-instance. □

4.2 A New FPT Algorithm for STRING TO STRING CORRECTION

This section outlines our recursive FPT algorithm that solves the STRING-TO-STRING CORRECTION decision problem using the bounded binary search tree approach. The algorithm, *Algorithm S2S* (see pages 31 - 32), accepts as input an ordered triple, $I = [X,\ Y,\ k]$, and outputs its classification as either a YES-instance or a NO-instance. *Algorithm S2S* is composed of two parts, a preprocessing phase and a branching phase. Initially, $[X,\ Y,\ k]$ enters the preprocessing phase of the algorithm. The purpose of preprocessing is to determine whether it is possible, in polynomial time, to classify $[X,\ Y,\ k]$ as a YES-instance or a NO-instance. The second part of *Algorithm S2S* uses branching to determine whether an $\omega(X,\ Y)$ exists[1], by constructing a bounded binary search tree based on the decision to either swap or delete symbols of X that must be rearranged or removed in order to transform X into Y. The height of the binary search tree is bounded by the parameter value, k. The branching phase is only entered if $[X,\ Y,\ k]$ is not determined to be YES-instance or a NO-instance by the preprocessing phase. During *Algorithm S2S*, X is modified via σ and δ functions, and τ is applied to both X and Y. The remainder of this section provides a thorough explanation of *Algorithm S2S*.

[1]From now on in this section, we assume that $|\omega(X,\ Y)| \leq k$.

4.2.1 Preprocessing Phase

A series of reduction rules is applied to [X, Y, k] at the beginning of each recursive call in order to determine whether it can be identified as a YES-instance or a NO-instance after only a polynomial number of steps. The preprocessing phase of *Algorithm S2S* includes eight scenarios in which [X, Y, k] can be classified as a NO-instance and three scenarios in which [X, Y, k] can be identified as a YES-instance in polynomial running time. If preprocessing is completed and the instance is not yet classified as a YES-instance or a NO-instance, further algorithmic steps which require exponential running time are needed to determine whether the construction of a transformation $\omega(X, Y)$ is possible. The additional investigative steps are handled in the branching phase of *Algorithm S2S*. A complete solution including both preprocessing and branching is given in the pseudocoded *Algorithm S2S* below.

The first four reduction rules, in lines 1-9, test the validity of [X, Y, k] by ensuring that the instance satisfies some basic, yet necessary, structural characteristics to be determined a YES-instance.

1: **if** k < 0 **then**
2: **return** FALSE

The first reduction rule detects NO-instances by testing whether the parameter value, k, is negative (*Algorithm S2S*, page 31, line 1). There are two scenarios where $k < 0$. The first is when the original [X, Y, k] is invalid because it contains a negative parameter, k. To understand the second scenario, note that *Algorithm S2S* is recursive and that k is decremented during each recursive S2S call in which δ or σ is applied to X. A negative k value results when an excess of k edit operations is applied (i.e., when $|\omega| > k$). Both occasions lead to the determination of [X, Y, k] as a NO-instance, and the algorithm terminates.

3: **if** $|X| - |Y| > k$ **then**
4: **return** FALSE

The second opportunity to detect a NO-instance occurs when the number of permitted edits is smaller than the difference in lengths of the two strings. In this case, the two strings can never be equated since they cannot be edited to the same length (*Algorithm S2S*, page 31, line 3 - 4).

Algorithm S2S: A bounded binary search tree algorithm for solving the STRING-TO-STRING CORRECTION decision problem.

Require: The ordered triple $[X,\ Y,\ k]$, where X is the source string, Y is the target string and k is an upper bound on the number of edits.
Ensure: FALSE if X cannot be converted into Y using at most k edits; TRUE otherwise.

{Preprocessing phase of *Algorithm S2S*. $[X,\ Y,\ k]$ is assessed to see whether it can be classified as a YES or a NO-instance in polynomial time.}

{Number of edits is required to be nonnegative.}
1: **if** $k < 0$ **then**
2: **return** FALSE

{The string correction is not possible since there are not enough edits to equate the length of X and Y.}
3: **if** $|X| - |Y| > k$ **then**
4: **return** FALSE

{The string correction is not possible since X is shorter than Y and symbol insertion is not a permitted edit operation.}
5: **if** $|X| < |Y|$ **then**
6: **return** FALSE

{Verify that X contains the minimum number of each symbol needed to complete the string correction; numOccurrences(X, a) returns the number of times the symbol a occurs in the sequence X.}
7: **for all** symbol in Σ **do**
8: **if** numOccurences(X, symbol) < numOccurences(Y, symbol) **then**
9: **return** FALSE

{The first symbol in X matches the first symbol in Y. The matching symbols are removed via function τ. $\phi(X,\ a)$ returns the index of the first occurrence of symbol a in sequence X.}
10: **if** $\phi(X, y_1) = 1$ **then**
11: **return** S2S($[\tau(X),\ \tau(Y),\ k]$)

5: **if** $|X| < |Y|$ **then**
6: **return** FALSE

The next opportunity to detect a NO-instance is when X is shorter than Y (*Algorithm S2S*, page 31, line 5). This situation only occurs if the original input is invalid.

Algorithm S2S: *continued from previous page, page 31.*

```
1: {If each symbol in Y appears in X in the same order and the
   difference in length between the two strings is bounded above by k,
   then only edit operation δ is necessary; superSequence(X, Y)
   returns TRUE if X is a super sequence of Y.}
2: if superSequence(X, Y) then
3:   if |X| − |Y| ≤ k then
4:     return TRUE
5:   else
6:     return FALSE

   {Only edit operation σ is necessary. As long as both X and Y
   are of equal length and have the same number of each symbol
   in Y then the swapsOnly function is called.}
7: if |X| = |Y| then
8:   for all char in Y do
9:     if numOccurences(X, char) ! = numOccurences(Y, char) then
10:      return FALSE
11:  return swapsOnly([X, Y, k])

   {Branching phase of Algorithm S2S. Both σ and δ are permitted.
   The algorithm branches on a choice to either delete or swap the
   symbol directly before φ(X, y1).}
12: if S2S([δ(X, φ(X, y1) − 1), Y, k − 1]) then
13:  return TRUE
14: return S2S([σ(X, φ(X, y1) − 1), Y, k − 1])
```

```
7:  for all symbol in Σ do
8:      if numOccurences(X, symbol) < numOccurences(Y, symbol) then
9:          return FALSE
```

The fourth NO-instance classification occurs when there exists a symbol in Σ that appears more frequently in Y than in X (*Algorithm S2S*, page 31, line 8). The exclusion of symbol insertion from the set of possible edit operations deems any $[X,\ Y,\ k]$ of this form a NO-instance, regardless of the k value. Satisfying these four reduction rules ensures that for the given $[X,\ Y,\ k]$, $k \geq 0$, $|X| \geq |Y|$, and there does not exist a symbol that occurs more times in Y than in X.

10: **if** $\phi(X, y_1) = 1$ **then**
11: **return** $[\tau(X), \tau(Y), k]$

Observe that if X and Y are of the forms $ax_2 \ldots x_n$ and $ay_2 \ldots y_m$ respectively, for some $a \in \Sigma$, then a can be removed from the first position of each string without affecting the answer to the STRING-TO-STRING CORRECTION decision problem on $[X, Y, k]$, refer to Corollory 2. Furthermore, $[X, Y, k]$ is a YES-instance if and only if $[\tau(X), \tau(Y), k]$ is a YES-instance. Thus, *Algorithm S2S* recursively reduces $[X, Y, k]$ if X and Y share a common symbol as their first element via function τ. This reassignment is achieved through the recursive call S2S($\tau(X)$, $\tau(Y)$, k) (*Algorithm S2S*, page 31, lines 10 - 11).

The preprocessing techniques discussed thus far consist of sets of constraints based strictly on details pertaining to the relationship between the structural characteristics of X, Y and k. No edit operations are involved during the application of these reduction rules. At this stage, an $[X, Y, k]$ that has not been determined as a YES-instance or a NO-instance satisfies the following five conditions.

Condition 1. The parameter value, k, is non-negative.

Condition 2. The parameter value, k, is at least the difference in lengths between X and Y.

Condition 3. The length of Y is at most the length of Y.

Condition 4. x_1 does not equal y_1, that is, $\phi(X, y_1) \neq 1$.

Condition 5. For each $a \in \Sigma$, if a appears s times in Y, then a occurs at least t times in X, where $s \leq t$.

An example of an instance satisfying all five conditions is [*caabccba*, *accbc*, 4]. However, [*abbcda*, *abcb*, 4] does not satisfy Condition 4 because $x_1 = y_1$. As previously stated, function τ removes the leading character from a given string. Recall also that τ is invoked on X and Y simultaneously, when $x_1 = y_1$, (*Algorithm S2S*, page 31, lines 10 - 11). Equating the first symbol of X and Y requires δ, σ, or a combination of the two. In addition, unless both X and Y are initially of length 0, each successful transformation sequence contains function τ.

The remaining two reduction rules, shown in *Algorithm S2S* on page 32 lines 2 - 11, exhibit a deeper look into the structure of $[X,\ Y,\ k]$ with the main objective to identify whether the problem is solvable for instance $[X,\ Y,\ k]$ using exclusively δ operations or exclusively σ operations.[2]

```
2:  if superSequence(X, Y) then
3:      if |X| − |Y| ≤ k then
4:          return TRUE
5:      else
6:          return FALSE
```

The case involving only deletions is explored first. In addition to Conditions 1 - 5 discussed above, for the deletes only case to apply, Y must be a subsequence of X (and thus X is a supersequence of Y) with X longer than Y by at most k symbols (*Algorithm S2S*, page 32, lines 2 - 3). Then there exists an $\omega(X, Y)$ consisting only of operation δ and *Algorithm S2S* terminates, classifying $[X,\ Y,\ k]$ as a YES-instance. However, if X is a supersequence of Y but $|X| - |Y| > k$, then $[X,\ Y,\ k]$ is a NO-instance because it requires an excess of k deletions to transform X into Y (*Algorithm S2S*, page 32, line 6). For an in-depth justification supporting lines 2 - 6 above, see Correctness of Reduction Rules 4.2.2 in Section 4.2.2. In Chapter 5 we present *Algorithm deletesOnly*, a polynomial running time algorithm that determines the transformation sequence for the case involving only deletions.

If the deletions only case does not apply and X and Y are of equal length, then a transformation sequence consisting of only σ may exist. Details of *Algorithm swapsOnly*, as well as the conditions under which *Algorithm swapsOnly* is called, are described in Section 4.2.1.1.

4.2.1.1 Swaps Only Algorithm

If X and Y are of equal length, then δ cannot be included in $\omega(X, Y)$ as its application to X will result in $|X| < |Y|$. Furthermore, if $\omega(X, Y)$ exists, it is composed completely of operation σ. In addition to Conditions 1 - 5, for the swaps only case to apply, each symbol a in Y must have the same number of occurrences in both X

[2] A transformation sequence involving exclusively σs or exclusively δs also includes occurrences of function τ

and Y. If there exists an a for which this does not hold, then [X, Y, k] is classified as a NO-instance and *Algorithm S2S* terminates (*Algorithm S2S*, page 32, line 9 - 10). Otherwise, *Algorithm S2S* calls *Algorithm swapsOnly* to determine whether a transformation sequence consisting exclusively of operation σ exists. This is demonstrated on page 32 by lines 7 - 11 of *Algorithm S2S*.

```
7:  if |X| = |Y| then
8:      for all char in Y do
9:          if numOccurrences(X, char) ! = numOccurrences(Y, char) then
10:             return FALSE
11:     return swapsOnly([X, Y, k])
```

A notable difference between the deletes only and swaps only case is that the former is able to classify [X, Y, k] as a YES-instance or a NO-instance based strictly on the structural characteristics of the instance. That is, for the deletes only case the application of neither δ nor τ is required in solving the STRING-TO-STRING CORRECTION decision problem, where as in the swaps only case, modification of [X, Y, k] by σ and τ is necessary.

Algorithm swapsOnly first verifies that X is not the empty string. Note that $|X| = 0$ if and only if $|Y| = 0$, since $|X| = |Y|$ is a precondition to the function call to *Algorithm swapsOnly*. If both X and Y are empty, then a transformation sequence $\omega(X, Y)$ exists and the algorithms terminates, classifying [X, Y, k] as a YES-instance. If X and Y are both nonempty, *Algorithm swapsOnly* determines the index of the first occurrence of symbol y_1 in string X, namely $\phi(X, y_1)$, and copies that value to a new variable, *icpy*. This is illustrated by *Algorithm swapsOnly*, page 36, lines 1 - 4.

```
1:  if|X| = 0 then
2:      return TRUE
3:  i ← φ(X, y1)
4:  icpy ← i
```

The existence of $\phi(X, y_1)$ is guaranteed given that both X and Y are nonempty and that each a in Y has an equal number of occurrences in Y and X(*Algorithm S2S* page 32 line 9). The behaviour of *Algorithm swapsOnly* is governed by the value of $\phi(X, y_1)$.

Algorithm swapsOnly: An algorithm for the STRING-TO-STRING CORRECTION decision problem that only uses edit operation σ: swapsOnly([X, Y, k]); swapsOnly([X, Y, k]) determines whether there exists a transformation sequence, $\omega(X, Y)$ of length of most k.

Require: The ordered triple [X, Y, k], where X is the source string, Y is the target string and k is an upper bound on the number of edits. As well, $|X| = |Y|$ and each symbol has an equal number of occurrences in both X and Y.

Ensure: FALSE if X cannot be converted into Y using at most k swaps; TRUE otherwise.

{If X is the empty string, then the string correction from X to Y is complete.}
1: **if** $|X| = 0$ **then**
2: **return** TRUE
3: $i \leftarrow \phi(X, y_1)$ {First occurrence of y_1 in X is at index i}
4: $icpy \leftarrow i$

{As long as the number of permitted edits has not been exceeded, $x_{\phi(X,y_1)}$ is at most k swaps from the head position of X, then swap the target symbol to the first position of X.}
5: **while** $i - 1 \leq k$ **do**
6: **if** $i > 1$ **then**
7: **for** $j \leftarrow 0$, $j < i - 1$, $j++$ **do** {Swap $x_{\phi(X,y_1)}$ to first position in X}
8: $X \leftarrow \sigma(X, icpy - 1)$
9: $icpy - -$
10: $k \leftarrow k - (i - 1)$
11: $X \leftarrow \tau(X)$ {Remove head of X}
12: $Y \leftarrow \tau(Y)$ {Remove head of Y}
13: **if** $|X| > 0$ **then**
14: $i \leftarrow \phi(X, y_1)$ {Update ϕ value}
15: $icpy \leftarrow i$
16: **else**
17: **return** TRUE

{Return false if there is not enough permitted edits to swap $x_{\phi(X,y_1)}$ to first position in X.}
18: **return** FALSE

```
5:   while i − 1 ≤ k do
6:       if i > 1 then
7 :          for j = 0, j < i − 1, j + + do
8:              X ← σ(X, icpy − 1)
9:              icpy − −
10:         k ← k − (i − 1)
```

Algorithm swapsOnly then determines whether there are still enough edits remaining to swap $x_{\phi(X,y_1)}$ to the first position of X. Exactly $i-1$ swaps are required to equate x_1 and y_1, thus $i-1$ must be at most k (*Algorithm swapsOnly*, page 36, line 5). If $i > 1$, at least one swap is needed to relocate $x_{\phi(X,y_1)}$ to the head position of X. *Algorithm swapsOnly* proceeds to swap $x_{\phi(X,y_1)}$ to the leftmost position of X, by exchanging it with its left neighbour exactly $i-1$ times. During each iteration of the for loop, *icpy* is decremented to account for the advancement of $x_{\phi(X,y_1)}$ one position to the left (*Algorithm swapsOnly*, page 36, lines 7 - 9). As well, the parameter k is decremented to accommodate for the spent edits (*Algorithm swapsOnly*, page 36, line 10). The result is a pair, X and Y, such that $x_1 = y_1$. This is also the case when $i = 1$ (from *Algorithm swapsOnly*, page 36, line 6) and no swapping is required.

```
11:      X ← τ(X)
12:      Y ← τ(Y)
```

Since the lead symbols of X and Y are identical, and x_1 and y_1 are removed from X and Yrespectively (*Algorithm swapsOnly*, page 36, lines 11 - 12)

```
13:      if |X| > 0 then
14:          i ← φ(X, y1)
15:          icpy ← i
16:      else
17:          return TRUE
```

If at this point, X (and Y) is non empty, *Algorithm swapsOnly* determines $\phi(X, y_1)$ based on the new leading symbols of Y. *icpy* is reset to equal the new $\phi(X, y_1)$ and the process repeats. However, if X (and Y) is equal to the empty string, then *Algorithm swapsOnly* classifies $[X,\ Y,\ k]$ as a YES-instance since a string correction has been found using only swaps and *Algorithm swapsOnly* returns true. This is illustrated on

lines 13 - 17.

18: **return FALSE**

If $i-1$ ever exceeds k, then there are not enough edits remaining to swap $x_{\phi(X,y_1)}$ to the first position of X. *Algorithm swapsOnly* does not enter the while loop in this case and instead classifies $[X,\ Y,\ k]$ as a NO-instance and returns false.

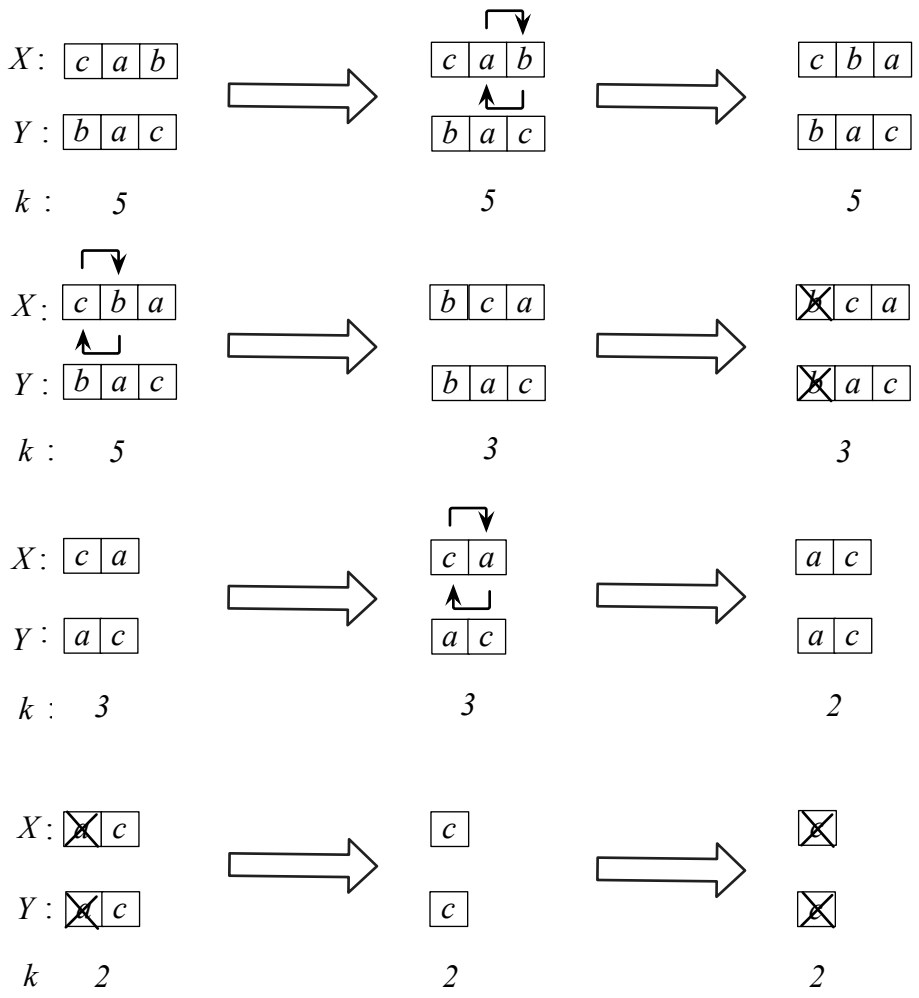

Figure 4.1: Application of the *Algorithm swapsOnly* to the YES-instance, [*cab*, *bac*, 5]. Progression of algorithmic steps begins in the top left corner through to the lower right corner.

To illustrate the above algorithm, we consider the example shown in Figure 4.1 where the question is to determine whether there exists an $\omega(cab, bac)$, with $k \leq 5$. After verifying that both X and Y are nonempty, *Algorithm swapsOnly* determines

that $\phi(cab, b) = 3$ (*Algorithm swapsOnly*, page 36, lines 1 - 3). Since $\phi(cab, b) - 1 = 2 \leq k = 5$, enough edits remain to swap the first occurrence of y_1 in X, namely b located in position 3, to the head position of X. b is swapped with its left adjacent symbol twice, first exchanging positions with the a in position 2, and next with the c in position 1. The result is $X = bca$, and thus $x_1 = y_1$. k is decremented to accommodate the two spent edits, resulting in the assignment $k = 3$ (*Algorithm swapsOnly*, page 36, lines 5 - 10). The first positions of X and Y contain a common symbol, namely b, thus by Corollory 2, the b is removed from X and Y via function τ. After modification via τ, $X = ca$, $Y = ac$ and $k = 3$. Removal of Y's head symbol via function τ results in a new y_1, and consequently, ϕ searches for the first occurrence of symbol a in X before returning to the top of the while loop (*Algorithm swapsOnly*, page 36, lines 11 - 14).

During the second pass through the while loop, $X = ca$, $Y = ac$ and $k = 3$. $\phi(ca, a) = 2$, indicating that a single swap is required to move the first a in X into the head position. Since $k = 3$, the required swap is permitted and $\sigma(ca, \phi(ca, a) - 1)$ swaps the first occurrence of a in X one position to the left, resulting in $X = ac$. *Algorithm swapsOnly* decrements the parameter, and thus $k = 2$ (*Algorithm swapsOnly*, page 36, lines 5 - 10). $x_1 = y_1$, thus function τ removes the head symbol of both X and Y resulting in $X = Y = c$ (*Algorithm swapsOnly*, page 36, lines 11 - 12). X is non-empty, and thus ϕ searches for the first occurrence of c in X ($\phi(c, c) = 1$) before entering the third pass of the while loop. Since $i = 1$, no edits are required to reposition c, and k remains unchanged. τ removes the common symbol in the head position, equating both X and Y to the empty string. Since $|X| = 0$, *Algorithm swapsOnly* terminates with [*cab*, *bac*, 5] determined to be a YES-instance.

The preceding example of *Algorithm swapsOnly* classifies [*cab*, *bac*, 5] as a YES-instance. Consider *Algorithm swapsOnly* applied to the NO-instance [*cab*, *bac*, 2], as shown in Figure 4.2. The algorithm first determines that $\phi(cab, b) = 3$ and as a result, the b in X is swapped to the left exactly twice, first with the a in position 2, and then with the c in position 1. The result of the two swaps is $X = bca$. The parameter is reduced from 2 to 0, and τ removes the lead symbol, b, from both X and Y. The instance is updated to [*ca*, *ac*, 0]. Since X is non-empty, ϕ searches for the first occurrence of a in X, and $i = 2$ is reassigned before *Algorithm swapsOnly* returns to the top of the while loop.

It is now the case that $i - 1 = 1$ is greater than the number of permitted edits, specifically, not enough edits remain to swap the a in X to the head position. *Algorithm swapsOnly* does not enter the while loop, but instead terminates with [*cab*, *bac*, 2] as a NO-instance.

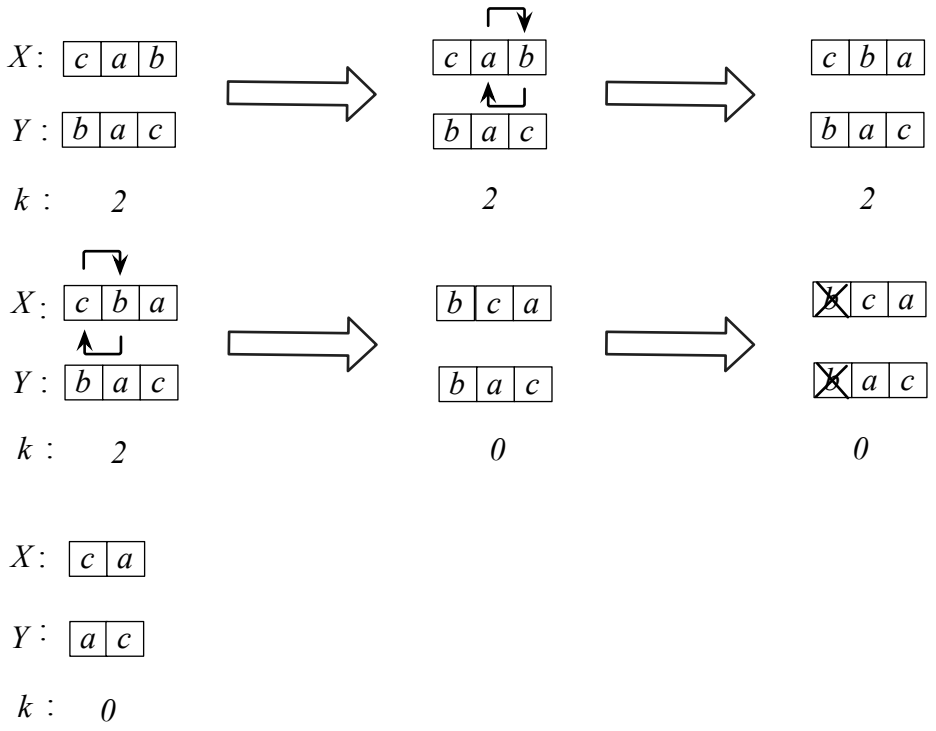

Figure 4.2: Application of the *Algorithm swapsOnly* to the NO-instance, [*cab*, *bac*, 2]. Progression of algorithmic steps begins in the top left corner through to the lower right corner.

The above section describes the preprocessing steps that are applied to an instance in order to classify it as a YES-instance or a NO-instance. An instance [X, Y, k] is *reduced* if no reduction rule can be applied. A proof justifying each reduction rule is presented in Section 4.2.2 of this chapter.

4.2.2 Correctness of Reduction Rules

This section presents a proof for each reduction rule provided in *Algorithm S2S*. Each rule accepts as input an ordered triple [X, Y, k]. The reductions rules below appear

in order of application, that is, Reduction Rules i applies only if Reduction Rules through $i-1$ do not apply.

Reduction Rule 1 *If $k < 0$, then [X, Y, k] is a NO-instance.*

Proof: By problem definition, the parameter k must be nonnegative.

Reduction Rule 2 *If $|X| - |Y| > k$, then [X, Y, k] is a NO-instance.*

Proof: X and Y must be the same length if they are to be determined equal strings. Thus, the parameter must be at least as large as the difference in length between X and Y.

Reduction Rule 3 *If $|X| < |Y|$, then [X, Y, k] is a NO-instance.*

Proof: If X is shorter then Y, the STRING-TO-STRING CORRECTION is not possible. Insertion of characters into X is not allowed as the only permitted edit operations are swaps and deletions. Performing a deletion on X shortens the length of the string. Similarly, swapping characters in X leaves the length of X unchanged.

Reduction Rule 4 *Given a symbol $a \in \Sigma$, if a occurs more in Y than in X, then [X, Y, k] is a NO-instance.*

Proof: Similar argument as 2. If there exists a symbol which occurs more frequently in Y than in X, the STRING-TO-STRING CORRECTION is not possible as insertions are not permitted.

Reduction Rule 5 *If x_1 and y_1 are identical, then [X, Y, k] is reduced by replacing X with $\tau(X)$ and Y with $\tau(Y)$.*

Proof: This follows directly from Corollory 2.

Reduction Rule 6 *If X is a supersequence of Y, then STRING-TO-STRING CORRECTION of [X, Y, k] is a YES-instance if $|X| - |Y| \leq k$ and a NO-instance otherwise.*

Proof: By definition, X is a supersequence of Y provided that X can be constructed from Y just by inserting additional symbols, each from Σ, at difference locations of Y(Chapter 2, Section 2.1). Let the newly inserted symbols form the set Γ. Then, Y can be constructed from X just by deleting each member of Γ from X. $|\Gamma| = |X| - |Y|$, so given that $|\Gamma| \leq k$, there are enough permitted edit operations to be spent as deletions and the STRING-TO-STRING CORRECTION is a YES-instance. On the other hand, if $|\Gamma| > k$, not enough

deletions can take place to equate X and Y, and the STRING-TO-STRING CORRECTION is a NO-instance.

Reduction Rule 7 *If X and Y are of equal length, and there exists a symbol in Y, say y_i, for which the number of occurrences in X and Y is not equal, then the STRING-TO-STRING CORRECTION for [X, Y, k] is a NO-instance. However, if $|X| = |Y|$ and each y_i, $1 \leq i \leq m$, has an equal number of occurrences in both X and Y, then the STRING-TO-STRING CORRECTION of [X, Y, k] may be possible. In this case, the classification of [X, Y, k] is attained via Algorithm swapsOnly.*

Proof: Let X and Y contain the same number of characters. Then deletions cannot take place since X and Y are the same length, and insertions are not permitted. Given that $|X| = |Y|$, if $[X, Y, k]$ is a YES-instance for STRING-TO-STRING CORRECTION, then the solution, $\omega(X, Y)$, must be composed entirely of swaps. Since symbol mutation is not permitted, each y_i, $1 \leq i \leq m$, of Y appears the same number of times in X. If there exists a y_i for which this is not true, then $[X, Y, k]$ is a NO-instance. If each y_i has equal occurrences in X and Y, then the STRING-TO-STRING CORRECTION decision problem for $[X, Y, k]$ is can only be determined via *Algorithm swapsOnly.*

4.2.3 Branching of *Algorithm S2S*

The previous section describes how *Algorithm S2S* determines whether $[X, Y, k]$ is a YES-instance or a NO-instance based on specific structural characteristics of X and Y, as well as the value of parameter k. It also explains how *Algorithm S2S* determines the existence of $\omega(X, Y)$, given that $[X, Y, k]$ is solvable using exclusively operation δ or operation σ. In this section we present an algorithmic approach for solving the STRING-TO-STRING CORRECTION decision problem for the remaining case, when *both* deletes and swaps are required.

Branching occurs when $[X, Y, k]$ cannot be classified as a YES-instance or NO-instance using the reduction rules. In this case, if an $\omega(X, Y)$ exists, then it must contain both deletes and swaps. In the branching section of *Algorithm S2S*, the existence of a transformation sequence is determined through construction, that is, edit operations are applied to X in search of ω. Recall that *Algorithm swapsOnly* also behaves in this manner, whereas the case involving only deletes is answered without

modifying $[X,\ Y,\ k]$. Based on the reduction rules presented in Section 4.2.1, an instance that is in reduced form satisfies the following six conditions.

Condition 1. The parameter value, k, is non-negative.

Condition 2. The parameter value, k, is at least equal to the difference in lengths between X and Y.

Condition 3. Y is shorter than X.

Condition 4. x_1 does not equal y_1, that is, $\phi(X, y_1) \neq 1$.

Condition 5. For each $a \in \Sigma$, if a appears s times in Y, then a occurs t times in X, where $s < t$.

Condition 6. $[X,\ Y,\ k]$ is not solvable using strictly deletes or strictly swaps.

Lines 12 - 14 below, taken from *Algorithm S2S* on page 32, determine the existence of a transformation sequence $\omega(X,\ Y)$ for instance $I = [X,\ Y,\ k]$ by construction of a bounded binary search tree. Let the reduced form of instance I be denoted by $I_R = [X',\ Y', k]$. Note that any reduction rule applied thus far will not have modified k. Figure 4.3 illustrates a node, Node A, containing the instance I and its reduced form. Observe that if the instance is already in reduced form, then $I = I_R$. Branches incident to Node A, as shown in Figure 4.4, represent the application of an edit operation to the instance I_R.

```
12:  if S2S([δ(X, φ(X, y1) − 1),  Y, k − 1]) then
13:      return TRUE
14:  return S2S([σ(X, φ(X, y1) − 1),  Y, k − 1])
```

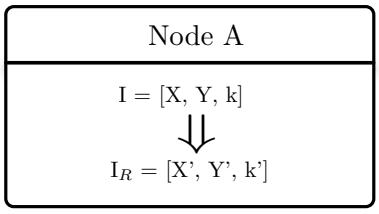

Figure 4.3: An example node used in the construction of the bounded binary search tree solution for the branching portion of *Algorithm S2S*. Node A represents the instance $[X,\ Y,\ k]$ and its reduced form, $I_R = [X', Y', k]$. The downward arrow between I and I_R represents the application of reduction rules to I through recursive calls to *Algorithm S2S*. If no reduction rules can be applied then $I = I_R$.

The search tree construction is as follows. Reduction rules are applied to I, resulting in I_R. This work occurs within Node A. *Algorithm S2S* determines $\phi(X, y_1)$ prior to branching. By Conditions 4 - 5, it can be concluded that $1 < \phi(X, y_1) \leq n$. It follows directly that there exists at least one symbol in X which precedes $x_{\phi(X,y_1)}$, namely $x_{\phi(X,y_1)-1}$. By Lemma 1, $x_{\phi(X,y_1)-1}$ must either be deleted or swapped to the right of $x_{\phi(X,y_1)}$. This results into two branches originating from Node A. The left branch represents the application of δ to $x_{\phi(X,y_1)-1}$, whereas the right branch represents the application of σ to $x_{\phi(X,y_1)-1}$. The resulting instances are $I_\delta = [\delta(X, \phi(X, y_1) - 1),\ Y, k - 1]$ and $I_\sigma = [\sigma(X, \phi(X, y_1) - 1),\ Y, k - 1]$, respectively, each of which becomes an I for a child node of Node A, see Nodes B and C of Figure 4.4. In each case, k is decremented to accommodate the edit operation performed.

A branch in the bounded binary search tree terminates if *Algorithm S2S* classifies the corresponding I as a YES-instance or a NO-instance. Branch termination occurs during either the preprocessing phase or branching portion of *Algorithm S2S*. If I is classified as a NO-instance, the algorithm continues to search for an ω by constructing different branches of the binary search tree as described above. This is achieved by recursing on the instances I_δ and I_σ. In the YES-instance case, the algorithm terminates by returning TRUE, indicating that an ω does exist.

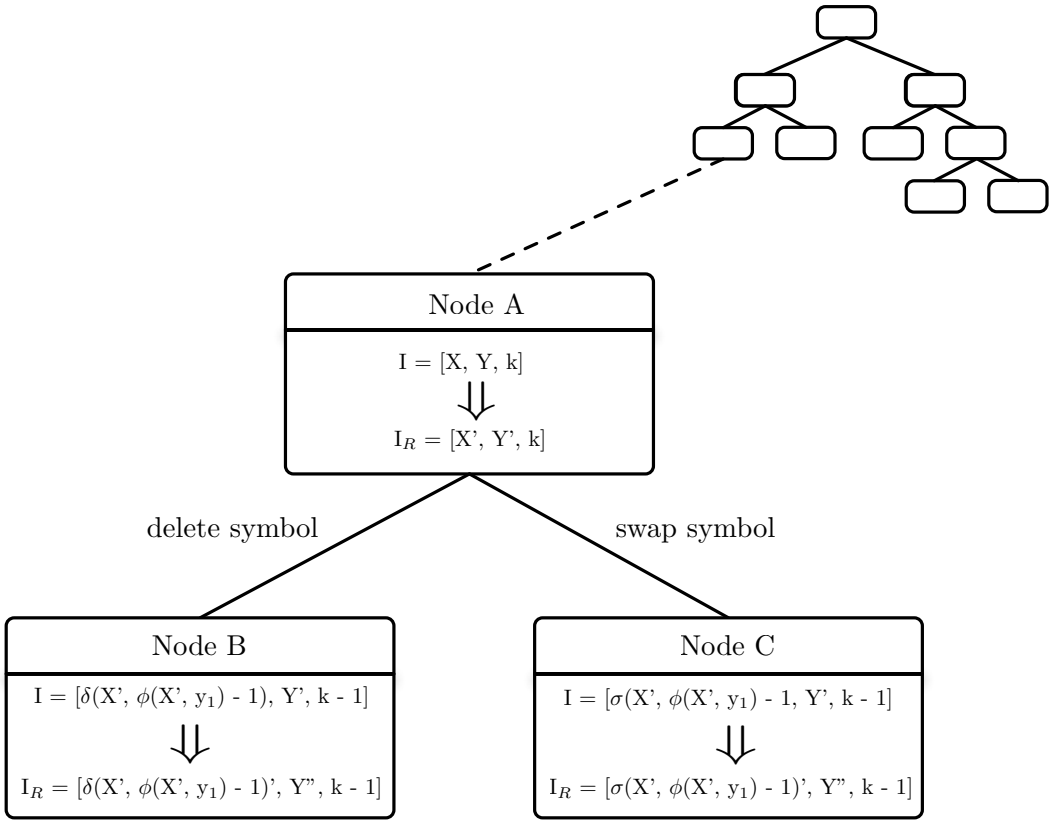

Figure 4.4: Within Node A, reduction rules are applied to I resulting in I_R. Node A's left child, Node B, contains I_R after it has undergone a deletion and the right child, Node C, contains I_R after it has undergone a swap operation. The modified I_R is renamed I for each child of Node A.

To illustrate the branching behaviour *Algorithm S2S*, consider the STRING-TO-STRING CORRECTION decision problem with instance $I = [abbdce, bcbd, 5]$, as shown in Figure 4.5. None of the reduction rules can be applied to I, demonstrating that the instance is in reduced form, i.e., $I = I_R$. Thus, the root of the binary search tree, Node A, contains $I = I_R = [abbdce, bcbd, 5]$. $y_1 = b$, thus *Algorithm S2S* searches for the first occurrence of b in X. *Algorithm S2S* determines that $\phi(abbdce, b) = 2$. $x_{\phi(X,b)}$ is directly preceded by the symbol a. Recall that by Lemma 1, $x_{\phi(X,b)-1}$ must either be deleted or swapped to the right of $x_{\phi(X,y_1)}$. A solution involving the delete edit operation is explored first (*Algorithm S2S*, page 32, line 12). The branch connecting Node A and Node B represents the deletion of symbol a from X, which results in the instance $I_\delta = [bbdce, bcbd, 4]$. I_δ is renamed I for Node B, and *Algorithm S2S* recurses on $I = [bbdce, bcbd, 4]$. Note that k is decremented to accommodate the spent edit operation.

During the next pass through *Algorithm S2S*, it is determined that X and Y share a common first element, namely the symbol b, thus the instance is not in reduced form. Since $\phi(X, y_1) = \phi(bbdce, b) = 1$, function τ is applied to both X and Y during the preprocessing phase. The algorithm recurses on [*bdce*, *cbd*, 4], (*Algorithm S2S*, page 31, lines 10 - 11). At this point, no reduction rules can be applied to [*bdce*, *cbd*, 4] and the instance is in reduced form. This is illustrated by the contents of Node B in Figure 4.5.

Algorithm S2S enters the branching portion and determines $\phi(X, y_1) = \phi(bdce, c) = 3$. Since $\phi(X, y_1) \neq 1$, there exists a symbol, namely d, that must be either deleted or swapped to the right of $x_{\phi(X,y_1)} = c$. The branch connecting Node B and Node C represents the deletion of d from X. This edit operation occurs during the recursive call S2S([$\delta(bdce, \phi(bdce, c) - 1)$, *cbd*, 3]). The resulting instance, I_δ =[*bce*, *cbd*, 3] becomes the I for Node C. Note that at this time, symbol d occurs more frequently in Y than in X, leading *Algorithm S2S* to classify [*bce*, *cbd*, 3] as a NO-instance during the preprocessing phase, (*Algorithm S2S*, page 31, lines 8 - 9). Node C shows that the application of reduction rules to the instance [*bce*, *cbd*, 3] does not lead to discovery of a transformation sequence, $\omega(X, Y)$. The recursive call responsible for this NO-instance classification originates at *Algorithm S2S*, page 32, line 12. Since the call returns FALSE, the previous deletion of symbol d from X is incorrect and so *Algorithm S2S* next considers the solution involving a swap.

The branch connecting Node B and Node D, shown of Figure 4.5, represents the exchange of positions of symbols d and c in X of instance $I = [bdce, cbd, 4]$. The algorithm then recurses on [*bcde*, *cbd*, 3], via the call to S2S([$\sigma(bdce, \phi(bdce, c)-1), cbd, 3$]), see *Algorithm S2S*, page 32, line 14. [*bcde*, *cbd*, 3] is in reduced form, so the branching portion of *Algorithm S2S* is entered. $\phi(X, y_1) = \phi(bcde, c) = 2$, indicating that the b must be either deleted or swapped to the right of $x_{\phi(X,c)} = c$. The deletion of b from *bcde* is represented by the branch connecting Node D and Node E, Figure 4.5. *Algorithm S2S* recurses on the new instance [*cde*, *cbd*, 2] during the call S2S([$\delta(bcde, \phi(bcde, c) - 1)$, *cbd*, 2]). During application of reduction rules to [*cde*, *cbd*, 2], symbol b appears more frequently in Y than in X, thus the [*cde*, *cbd*, 2] is determined as a NO-instance and the originating recursive call returns FALSE, (*Algorithm S2S*, page 31, lines 8 - 9). Node E shows that the application of reduction

rules to [*cde*, *cbd*, 2] leads to classification of the instance as a NO-instance. As in the case with Node C, *Algorithm S2S* next considers the solution involving a swap.

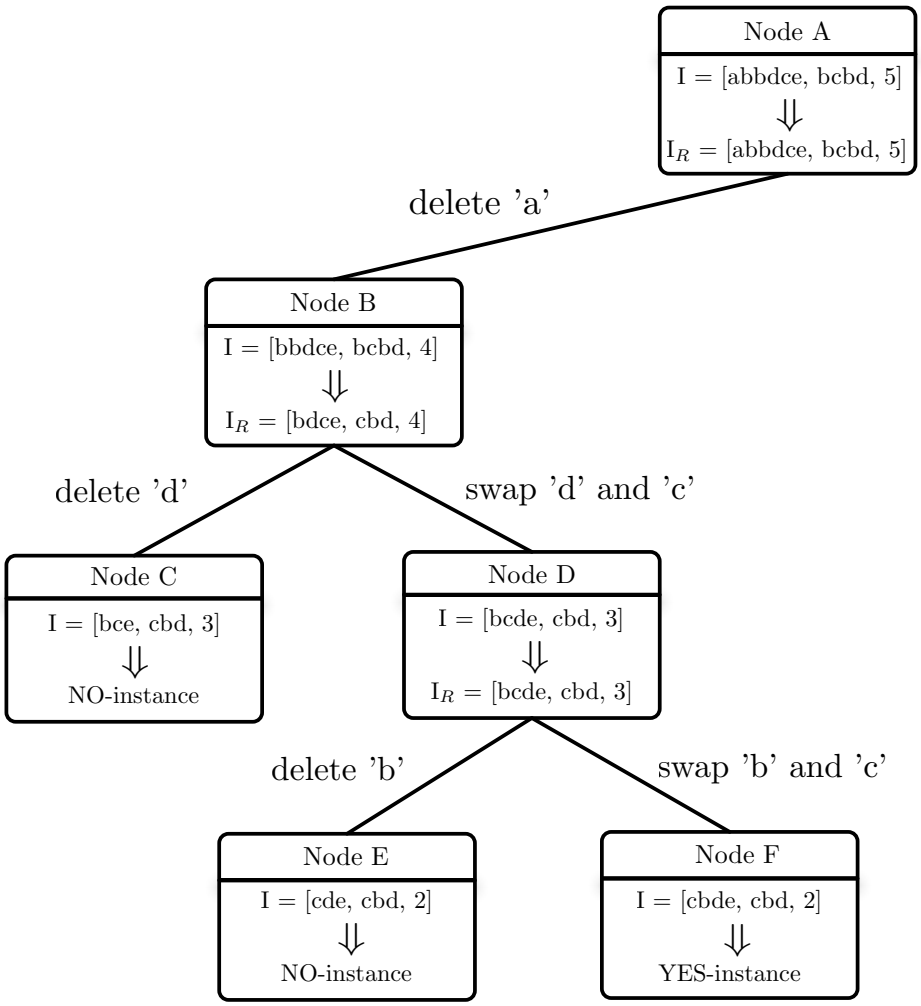

Figure 4.5: The binary search tree constructed by *Algorithm S2S* for instance [*abbdce*, *bcbd*, 5]. *Algorithm S2S* determines a solution for the STRING-TO-STRING CORRECTION decision problem in Node F, so Node A, the root of the binary search tree, only consists of a left side. Each branch connecting a parent and child node is labelled with the edit operation that is applied to the parent nodes I_R, resulting in the child nodes I.

Algorithm S2S proceeds by swapping symbols $x_{\phi(X,y_1)} = x_{\phi(bcde,c)} = c$ and $x_{\phi(X,y_1)-1} = x_{\phi(bcde,c)-1} = b$. The swap is conducted through the call S2S([$\sigma(bcde, \phi(bcde, c)-1)$, *cbd*, 2]) and represented by the connecting branch between Node D and Node F. The resulting instance, [*cbde*, *cbd*, 2], has c as a common symbol in the head position of

X and Y, thus the instance is not reduced as $\phi(X, y_1) = \phi(cbde, c) = 1$. Function τ is applied to both X and Y, and *Algorithm S2S* recurses on [$\tau(cbde)$, $\tau(cbd)$, 2]. The resulting instance, [bde, bd, 2], also has a shared element, namely b, in the head position of X and Y. Function τ is applied to X and Y again, and *Algorithm S2S* recurses on [de, d, 2]. $\phi(X, y_1) = \phi(de, d) = 1$, so the algorithm recurses on [$\tau(de)$, $\tau(e)$, 2] = [d, $\emptyset$, 2]. Recall that the definition of a supersequence states that given two strings, $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_m$, X is a supersequence of Y provided that X can be constructed from Y only by inserting additional symbols, each from Σ, at different locations of Y. Under this definition, each $\{X|$ with $X \neq \emptyset$, $x_i \in \Sigma$ for $1 \leq i \leq n\}$ is a supersequence of $Y = \emptyset$. Since X is a supersequence of Y, and $|X| - |Y| = 1 - 0 \leq k = 2$, *Algorithm S2S* determines that the decision problem for the current instance, [d, $\emptyset$, 2], is solvable using exclusively the delete operation. Figure 4.6 shows the individual reduction rules as applied to $I = [cbde, cbd, 2]$. *Algorithm S2S* returns TRUE.

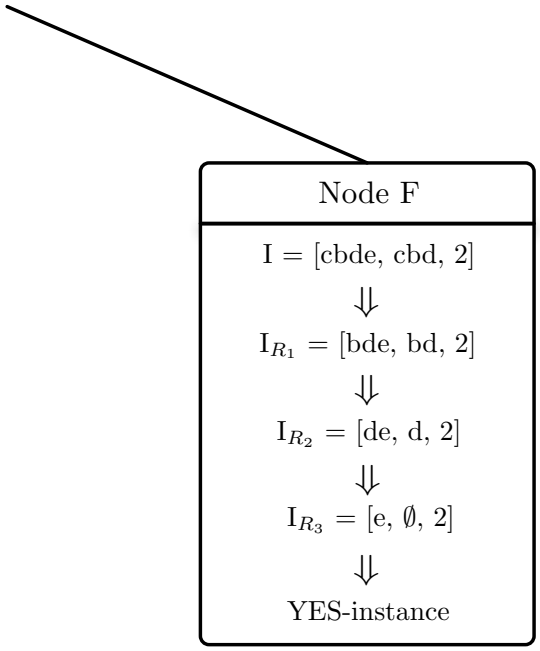

Figure 4.6: In Node F of Figure 4.5, a series of reductions rules is applied to the instance [cbde, cbd, 2] before its classification as a YES-instance. Each reduction rule is applied during a separate recursive call to *Algorithm S2S*. The final YES-instance classification results from the deletes only reduction rule on page 32, line 4.

4.3 Run Time of *Algorithm S2S*

This section presents a complexity analysis for *Algorithm S2S*. The algorithm is decomposed into its corresponding reduction rules and the branching part, each is individually analyzed and assigned a complexity function, $f(n, m, k)$, where n and m are the lengths of X and Y respectively, and k is an upper bound on the number of permitted edits. A brief complexity argument is given for each run time provided. Figure 4.8 summarizes these findings. This section assumes the reader has a basic understanding of complexity analysis for the worst case scenario of a given segment of pseudocode.

4.3.1 Run Time Analysis of the Basic Preprocessing Steps

This sections provides a detailed run time analysis of Reduction Rules $1-5$ as stated on page 40.

Reduction Rule 1 Determining whether the parameter is less than zero occurs in constant time. Thus, Reduction Rule 1 is $O(1)$.

Reduction Rule 2 Comparing the difference in lengths between X and Y with the parameter can be done in constant time. Thus, Reduction Rule 2 is $O(1)$.

Recall that Reduction Rule i only applies if Reduction Rule $i-1$ does not apply, indicating that for the following reduction rules, $|X| - |Y| \leq k$. For simplification purposes, the remaining complexity functions will be expressed in terms of only m and k, based on the fact that $n \leq k + m$. That is, n is replaced with $k + m$.

Reduction Rule 3 Determining whether $|X| < |Y|$, occurs in constant time. Thus, Reduction Rule 3 is $O(1)$.

Reduction Rule 4 The fourth reduction rule compares the number of occurrences of each symbol in Y to its number of occurrences in X. Y can be composed of at most $|Y| = m$ different symbols, leading to a theoretical run time that is linear in the length of Y. Reduction Rule 4 is computed in $O(m)$.

Reduction Rule 5 Determining $\phi(X, y_1)$ requires scanning through at most $|X| = n$ symbols in X in search of the first occurrence of y_1. This search is linear in n. Removing the leading symbol from both X and Y adds only a constant amount of time to the complexity function, thus Reduction Rule 5 is $O(k + m)$.

4.3.2 Run Time Analysis of superSequence(X, Y)

In order to identify whether X is a supersequence of Y, each element of Y must be located in X such that for all $1 \leq j \leq m$, y_j precedes y_{j+1} in X. The superSequence algorithm first determines $\phi(X, y_1) = i$, then removes the first i characters from X and the head character from Y, and finally recurses on the reduced (X, Y) pair. If during a recursive call, $|X| = 0$ but $|Y| > 0$ ever occurs, then X is not a supersequence of Y. Otherwise, X is indeed a supersequence of Y. By shortening X to length $|n| - i$ at each iteration, we are ensured that X is iterated over at most once. Thus by Reduction Rule 6, the superSequence algorithm runs in $O(k + m)$.

4.3.3 Run Time Analysis of *Algorithm swapsOnly*

The initial step of *Algorithm swapsOnly* asks if X is equal to the empty string. This is done by verifying the condition $|X| = 0$ (*Algorithm swapsOnly*, page 36, line 1). Comparison of two numbers is performed in constant time. For each $x_i \neq y_i$, the following occurs. *Algorithm swapsOnly* locates the first occurrence of y_1 in X, that is $\phi(X, y_1) = i$. This is performed in $O(i)$, where $i \leq \min\{k + 1, m\}$. Repositioning x_i to the first index in X requires exactly $i - 1$ swaps. Once x_i has been swapped to the head position of X and the parameter has been decremented, the constant time operation τ is applied to both X and Y (*Algorithm swapsOnly*, page 36, lines 10 - 12). Given that X and Y are not equal, *Algorithm swapsOnly* determines the next $\phi(X, y_1) = i'$, and the process repeats provided that $i' - 1 \leq k - (i - 1)$.

The cumulative total time to swap each $x_i \neq y_i$ is $\sum_{j=1}^{k}(i - 1)$. It occurs in $O(k)$ since only k swaps are permitted. The cumulative total time to search for each $x_i \neq y_i$ is $\sum_{j=1}^{k} i$, where each $i = \min\{k+1, m\}$, taking into account that after the swaps have been performed k is decremented by $i-1$. It may be possible that $\min\{k+1, m\} = m$ for some iteration of the algorithm, and $\min\{k + 1, m\} = k + 1$ for other iterations. Thus $\sum_{j=1}^{k} i$ is the greater of $\sum_{j=1}^{k}(k + 1)$ and $\sum_{j=1}^{k} m$. These run in $O(k + 1)$ and $O(m)$, respectively. Thus, *Algorithm swapsOnly* has a total running time equal to the greater of:

1. $\sum_{j=1}^{k}(i - 1) + \sum_{j=1}^{k}(k + 1)$ which is $O(k) + O(2k) = O(3k) = O(k)$, or
2. $\sum_{j=1}^{k}(i - 1) + \sum_{j=1}^{k} m$ which is $O(k) + O(m) = O(k + m)$.

Therefore *Algorithm swapsOnly* runs in $O(k + m)$.

4.3.4 Maximum Size of the Search Tree

Section 4.2.3 describes the recursive construction of a binary search tree with a node representing an instance and its reduced form, $I = [X,\ Y,\ k]$ and $I_R = [X', Y', k']$ respectively, and the application of either a delete or a swap to $\phi(X, y_1) - 1$ represented by the node's left and right incident edges. Recall that k is decremented each time an edit operation is preformed on $\phi(X, y_1) - 1$. If for some branch in the binary search tree, the corresponding k value is decremented below zero, the branch is determined to be a NO-instance. This is a result of Condition 1. Thus, the maximum height of the tree is bounded above by k, and the resulting tree is classified as a bounded binary search tree. Figure 4.7 shows the complete bounded binary search tree corresponding to an instance $I = [X,\ Y, 4]$, with each edge labeled with the corresponding edit operation preformed on I_R. The complete tree is composed of exactly $2^{k+1} - 1 = 31$ nodes. The maximum size of a binary search tree of height k is in $O(2^k)$.

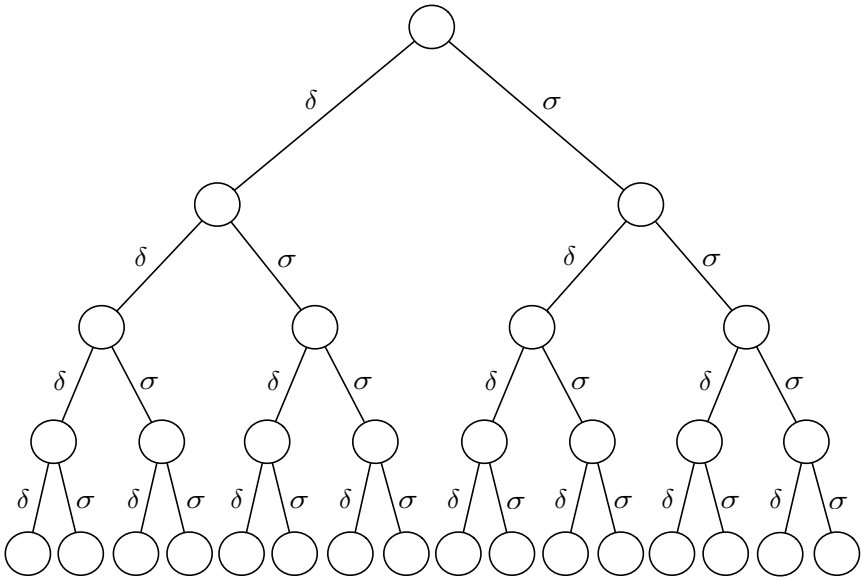

Figure 4.7: The bounded search tree with each branch labeled with its corresponding edit operation.

4.3.5 Run Time Analysis of *Algorithm S2S*

Figure 4.8 summarizes the theoretical running time of each reduction rule, as well as the size of the search tree constructed by the algorithm. The theoretical running time for *Algorithm S2S* corresponds to the amount of time required to build the associated

bounded binary search tree. The preprocessing steps are applied to $[X,\ Y,\ k]$ at each node, reducing I into I_R when possible. The maximum amount of time spent at any node is bound above by the most time consuming preprocessing step. The most computationally expensive reduction rule occurs percisely when determining the number of appearances of each character in both X and Y(as in Condition 5), during either the case where only delete edit operations or during *Algorithm swapsOnly*. Each of these reduction rules runs in $O(k + m)$.

Figure 4.8: Decomposition of *Algorithm S2S* into it corresponding reduction rules, each paired with corresponding run time analysis.

Rule	Theoretical Running Time	Page, Lines
Reduction Rule 1	$O(1)$	pg 31, 1 - 2
Reduction Rule 2	$O(1)$	pg 31, 3 - 4
Reduction Rule 3	$O(1)$	pg 31, 5 - 6
Reduction Rule 4	$O(m)$	pg 31, 7 - 9
Reduction Rule 5	$O(k + m)$	pg 31, 10 - 11
superSequence(X, Y)	$O(k + m)$	pg 32, 2 - 6
Algorithm swapsOnly	$O(k + m)$	pg 32, 7 - 11
Search Tree Size	$O(2^k)$	pg 32, 12 - 14

Branching occurs after instance reduction. The delete and swap edit operations require a constant amount of time. Thus, the running time for determining the classification of $[X,\ Y,\ k]$ is equal to the size of tree multiplied by the work performed at each node. This is, *Algorithm S2S* runs in $O(2^k(k + m))$ and therefore the STRING-TO-STRING CORRECTION decision problem is a member of the class FPT.

This chapter introduces *Algorithm S2S*, a new fpt algorithm for solving the STRING-TO-STRING CORRECTION decision problem. Lemmas, theorems and corollaries in support of *Algorithm S2S* are provided in Section 4.1. Each preprocessing steps and the branching portion of *Algorithm S2S* was analyzed to determine the overall theoretical running time of $2^k k + m$. We also show that the STRING-TO-STRING CORRECTION decision problem is a member of the class FPT. The next chapter discusses the implementation of *Algorithm S2S*, including the details to extend the solution so that the corresponding transformation sequence is provided in the event that $[X,\ Y,\ k]$ is a YES-instance. Details pertaining to the testing methodology

are also discussed at length. Chapter 5 concludes with an in depth analysis of the practical running times attained during the testing phase of implementations.

Chapter 5

Implementation and Experimental Results

In this chapter we present the reader with details pertaining to the implementation of *Algorithm S2S*. First we introduce the construction and maintenance of the transformation sequence ω, as well as a complete algorithm for solving the case involving exclusively edit operation δ. Section 5.2 gives a brief description of the implementation environment, basic design choices and method by which the code was tested. The chapter concludes with an analysis of the results from the running of the algorithm.

5.1 Implementation

Algorithm S2S, introduced in Chapter 4, classifies a given input instance as either a YES-instance or a NO-instance, thus solving only the STRING-TO-STRING CORRECTION decision problem. It is extremely useful in the implementation to extend *Algorithm S2S* by supplementing the previously provided pseudocode with a tracking of all the edit operations, such that a complete construction from X to Y can also be delivered as part of the answer in a YES-instance. Three main additions are necessary as described in the following.

1. We integrate steps to determine a transformation sequence $\omega(X, Y)$ in the event that $[X, Y, k]$ is a YES-instance. Upon encountering a YES-instance, the modified *Algorithm S2S* returns a sequence of edit operations of length at most k that results in Y when applied in succession to X. If a transformation from X to Y does not exist, the implementation returns that $[X, Y, k]$ is a NO-instance.

2. We incorporate a backtracking system responsible for maintaining the correctness of ω and $[X,\ Y,\ k]$ during construction of the bounded binary search tree.

3. We introduce a new algorithm, namely *Algorithm deletesOnly*, which modifies $[X,\ Y,\ k]$, given that it is known to be a YES-instance and the remaining operations are solely δs.[1]

Note that our new implementation is constructive while the algorithm discussed in Chapter 4 focused on determining only a yes/no answer to the STRING-TO-STRING CORRECTION decision problem.

5.1.1 Constructing the Transformation Sequence ω

Recall that a transformation sequence ω is defined as $\omega = \omega_1\omega_2 \ldots \omega_w$, where $\omega_j \in \{\tau, \delta, \sigma\}$, for $1 \leq j \leq w$. For implementation purposes, we define each ω_j as an ordered triple $(\gamma,\ i,\ x_i)$, where $\gamma \in \{\tau, \sigma, \delta\}$, i is a location in X, and x_i is the symbol at position i in X. Let $T([X,\ Y, k])$ denote the bounded binary search tree which is constructed during the application of *Algorithm S2S* to the instance $[X,\ Y,\ k]$ (see Chapter 4, Section 4.2.3). $T([X,\ Y, k])$ and ω are generated simultaneously. Initially, when $T([X,\ Y, k])$ equals the empty tree, ω is set to the empty sequence. ω is updated as the bounded binary search tree is constructed. Each time that X is modified by τ, σ or δ, the corresponding ordered triple is appended to ω.

If a branch in $T([X,\ Y, k])$ is determined to be a YES-instance, then ω is returned. If a branch in $T([X,\ Y, k])$ is determined to be a NO-instance, then the current state of ω does not represent a transformation from X to Y. In this case, *Algorithm S2S* continues to search for an ω by constructing different branches of $T([X,\ Y, k])$, as described in Chapter 4. However, before recursive construction of $T([X,\ Y, k])$ proceeds, ω and $[X,\ Y,\ k]$ must be restored to their states prior to the incorrectly applied functions. The reversal of edit operations is accomplished via the backtracking method.

[1] Recall that the case exclusively involving δs is solvable without modification of X as described in Chapter 4.

5.1.2 Maintaining the Correctness of ω and $[X, Y, k]$: Backtracking Method

Let $\omega_{|\omega|} = (\gamma, i, x_i)$ denote the last triple that is appended to ω at the time that $[X,\ Y,\ k]$ is classified as a NO-instance. There are two scenarios which necessitate backtracking and they are distinguished by the type of edit operation belonging to $\omega_{|\omega|}$, specifically whether γ is a δ or a σ. Figures 5.1 and 5.2 illustrate the state of the constructed bounded binary search tree $T([X, Y, k])$ for the two scenarios, prior to the invocation of backtracking. The figures assume an instance $[X,\ Y,\ k]$ with $k = 2$. The location in $T([X, Y, 2])$ where $[X, Y, 2]$ is classified to be a NO-instance is represented by the black leaf node. The dotted lines show the branch and node that are to be added to $T([X, Y, 2])$ once backtracking is complete.

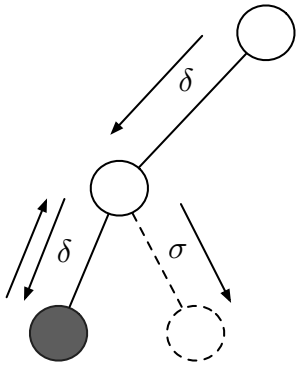

Figure 5.1: $\omega_{|\omega|}$ has $\gamma = \delta$ when $[X,\ Y,\ k]$ is classified as a NO-instance, shown by the black terminal node. Backtracking reverses the deletion on X, by inserting the previously deleted symbol back into its original position. The parameter k is incremented, and *Algorithm S2S* proceeds by applying σ to $x_{\phi(X,y_1)-1}$. Application of σ is represented by the dotted portion of the figure.

We consider first the scenario where $[X,\ Y,\ k]$ is determined to be a NO-instance, and $\gamma = \delta$ for $\omega_{|\omega|}$, see Figure 5.1. Recalling from Chapter 4, the case exclusively involving δ is identifiable based on the characteristics of $[X,\ Y,\ k]$. Furthermore, *Algorithm S2S* can classify $[X,\ Y,\ k]$ as a YES-instance or a NO-instance without modification of X. This implies that δ is never incorrectly applied to X during the deletions only case, and thus, any such incorrectly applied δ must result from the branching portion of *Algorithm S2S*. Specifically, δ belonging to $\omega_{|\omega|}$ originates from *Algorithm S2S*, page 32, line 12.

The second backtracking scenario is where $[X,\ Y,\ k]$ is determined a NO-instance, and $\gamma = \sigma$ for $\omega_{|\omega|}$, see Figure 5.2. An incorrectly applied σ can originate either during *Algorithm swapsOnly*, or during the branching portion of *Algorithm S2S* (*Algorithm S2S*, pages 31 - 32, lines 11 and 14).

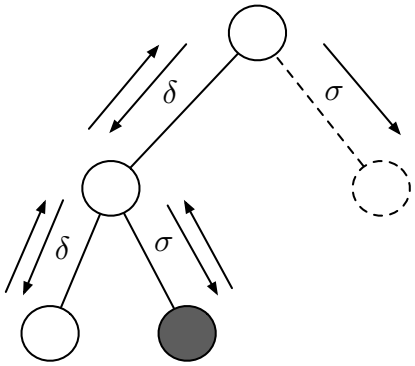

Figure 5.2: $\omega_{|\omega|}$ has $\gamma = \sigma$ when $[X,\ Y,\ k]$ is classified as a NO-instance, shown by the black terminal node. Backtracking reverses the swap on X, by exchanging the previously swapped symbols back to their original positions. The parameter k is incremented, and *Algorithm S2S* proceeds by constructing a swap branch for the next node possessing only a deletion branch i.e., a left child. If each node in $T([X,\ Y, 2])$ has both a left and a right branch, then $T([X,\ Y, 2])$ is complete.

In both scenarios, $[X,\ Y,\ k]$ must be restored to the I_R belonging to the parent of the node associated with the NO-instance classification of $[X,\ Y,\ k]$. If $\gamma = \delta$, this includes reversing the applied δ by inserting x_i back into X in position i and incrementing k. If $\gamma = \sigma$, backtracking reverses the incorrectly applied σ by exchanging the positions of x_i and x_{i-1}. In both cases, k is incremented by one and $\omega_{|\omega|} = (\gamma, i, x_i)$ is also removed from the end of ω. The dotted portion of $T([X,\ Y, 2])$, shown in Figures 5.1 - 5.2, represents the application of σ to $[X,\ Y,\ k]$ after backtracking has transpired. Note that both cases involving backtracking thus far result in *Algorithm S2S* proceeding with the application of σ to X.

Given a node location in $T([X,\ Y, k])$ containing $[X,\ Y,\ k]$ and ω, the application of backtracking results in traversing to the parent node of the given node. Backtracking recurses on the modified $[X,\ Y,\ k]$ and ω until either of the following two situations is encountered: (a) a node without a right child is located, or (b) no such node exists because $T([X,\ Y, k])$ is a complete binary tree. In case (a), *Algorithm S2S* proceeds by applying σ to $[X,\ Y,\ k]$, and the search for $\omega(X,\ Y)$ continues. In case (b), the complete bounded binary search tree has been constructed, no such ω has been found,

and thus *Algorithm S2S* terminates by classifying $[X,\ Y,\ k]$ as a NO-instance. This case can be identified by the search reaching the root node of $T([X,\ Y, k])$ in which case $|\omega| = 0$.

In summary, backtracking is recursively applied to $[X,\ Y,\ k]$ and ω until it finds a node in $T([X,\ Y, k])$ that does not contain a right child, reverses each operation applied to X between the newly found node and terminal node representing the NO-instance classification, and removes from ω each $\omega_{|\omega|}$ associated with a reversed operation, thus maintaining the correctness of ω. This is illustrated in detail in Figure 5.3.

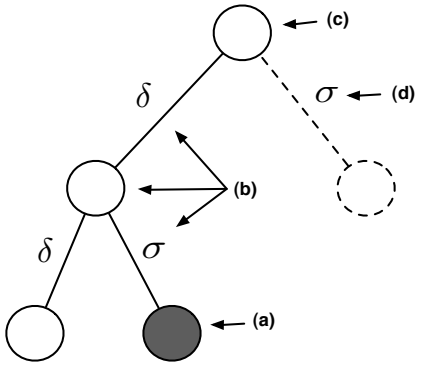

Figure 5.3: Summary of the backtracking steps. **a)** the shaded node is where the $[X,\ Y,\ k]$ is classified as a NO-instance, **b)** arrows point to the edits that must be reversed to correct the state of ω and $[X,\ Y,\ k]$ before *Algorithm S2S* can proceed with a swap, **c)** node found with no right child. This is the return point of backtracking for the given example tree, **d)** *Algorithm S2S* proceeds by applying σ to $[X,\ Y,\ k]$.

5.1.3 Deletes Only Algorithm

Reduction Rule 6 presented in Chapter 4 classifies $[X,\ Y,\ k]$ as a YES-instance, given that X is a supersequence of Y and that $|X| - |Y| \leq k$. This classification is achievable without applying edit operations to X, and thus the transformation sequence is not constructed. In this section, we present *Algorithm deletesOnly*, which modifies X into Ythrough application of δ and τ to X. *Algorithm deletesOnly* is only called if STRING-TO-STRING CORRECTION for $[X,\ Y,\ k]$ is solvable exclusively using edit operation δ. That is, line 4 on page 32 is changed from a **return** TRUE, to a call to the *deletesOnly algorithm* as shown below.

```
2:  if superSequence(X, Y) then
3:      if |X| − |Y| ≤ k then
4:          return deletesOnly([Y, X, k])
5:      else
6:          return FALSE
```

Algorithm deletesOnly: An algorithm for STRING-TO-STRING CORRECTION that only uses edit operation δ: deletesOnly(X, Y, k). deletesOnly(X, Y, k) is only called if there exists a transformation sequence, $\omega(X, Y)$ of length of most k.

Require: The ordered triple $[X, Y, k]$, where X is the source string, Y is the target string and k is an upper bound on the number of edits.

Ensure: TRUE, X is transformed into Y using at most k δs;
A transformation sequence, ω, of size at most k is also constructed.

```
1:  for all char in Y do
2:      i ← φ(X, y1)        {First occurrence of y1 in X is at index i}
        {Delete all symbols in front of y1 in X.}
3:      for j = 0; j < i − 1; j++ do
4:          X ← δ(X, 1)
5:      k ← k − (i − 1)
6:      X ← τ(X)                          {Remove head of X.}
7:      Y ← τ(Y)                          {Remove head of Y.}
    {Delete all remaining symbols in X.}
8:  if |X| > 0 then
9:      for i = 0; i < |X|; i++ do
10:         X ← δ(X, 1)
11:     k ← k − |X|
```

Algorithm deletesOnly is presented on page 60. It is useful to explain it line by line. Starting with lines 1 - 4, the first occurrence of y_1 is located in X and assigned to i, via function ϕ. Each x_j, for $1 \leq j < i$, is deleted, leaving $x_1 = x_{\phi(X,y_1)}$.

1: **for each** char in Y **do**
2: $\quad i \leftarrow \phi(X, y_1)$
3: $\quad$ **for** $j = 0; j < i - 1; j++$ **do**
4: $\quad\quad X \leftarrow \delta(X, 1)$

The parameter k is decremented to account for the deletions performed (*Algorithm deletesOnly*, page 60, line 5). The result is an X-Y pair with $x_1 = y_1$. Then, function τ removes the head symbol of each string, (*Algorithm deletesOnly*, page 60, lines 6 - 7). This process s repeated fir each symbol in Y.

5: $\quad k \leftarrow k - (i - 1)$
6: $\quad X \leftarrow \tau(X)$
7: $\quad Y \leftarrow \tau(Y)$

At this point, each y_i, with $1 \leq i \leq m$, has been located in X, and Y is equal to the empty string. The only step remaining is to delete all characters still in X, and decrement k accordingly, (*Algorithm deletesOnly*, page 60, lines 8 - 11).

8: **if** $|X| > 0$ **then**
9: $\quad$ **for** $i = 0; i < |X|; i++$ **do**
10: $\quad\quad X \leftarrow \delta(X, 1)$
11: $\quad k \leftarrow k - |X|$

To illustrate the above algorithm, consider the example where we are trying to determine the transformation sequence ω to the YES-instance, $[abcbcddc, cbd, 5]$, presented in Figure 5.4 on page 62. *Algorithm deletesOnly* begins by determining that $\phi(X, c) = 3$, and assigning $i = 3$. Each symbol in front of $x_{\phi(X,c)}$, specifically the a and b, is then deleted via the for loop, resulting in $X = cbcddc$. k is reassigned the value 3. $x_1 = y_1$, so by Corollory 2 c is removed from X and Y via function τ. The modified instance thus far is $[bcddc, bd, 3]$.

During the second iteration through the outer for loop, $\phi(X, b) = 1$. The inner for loop is not entered, k is unchanged, and τ removes both x_1 and y_1, resulting in $[cddc, d, 3]$. The third and last, iteration of the outer for loop determines $\phi(X, d) = 2$. The head symbol of X, c, is deleted by the inner for loop and k is reassigned the value 2. Since $x_1 = y_1 = d$, the lead symbol of X and Y is removed via function τ, resulting

in $[dc, \emptyset, 2]$. The remaining two symbols, d and c, of X are removed iteratively by the last for loop, and k is updated to reflect the number of deletions performed to equate X to $\emptyset$.

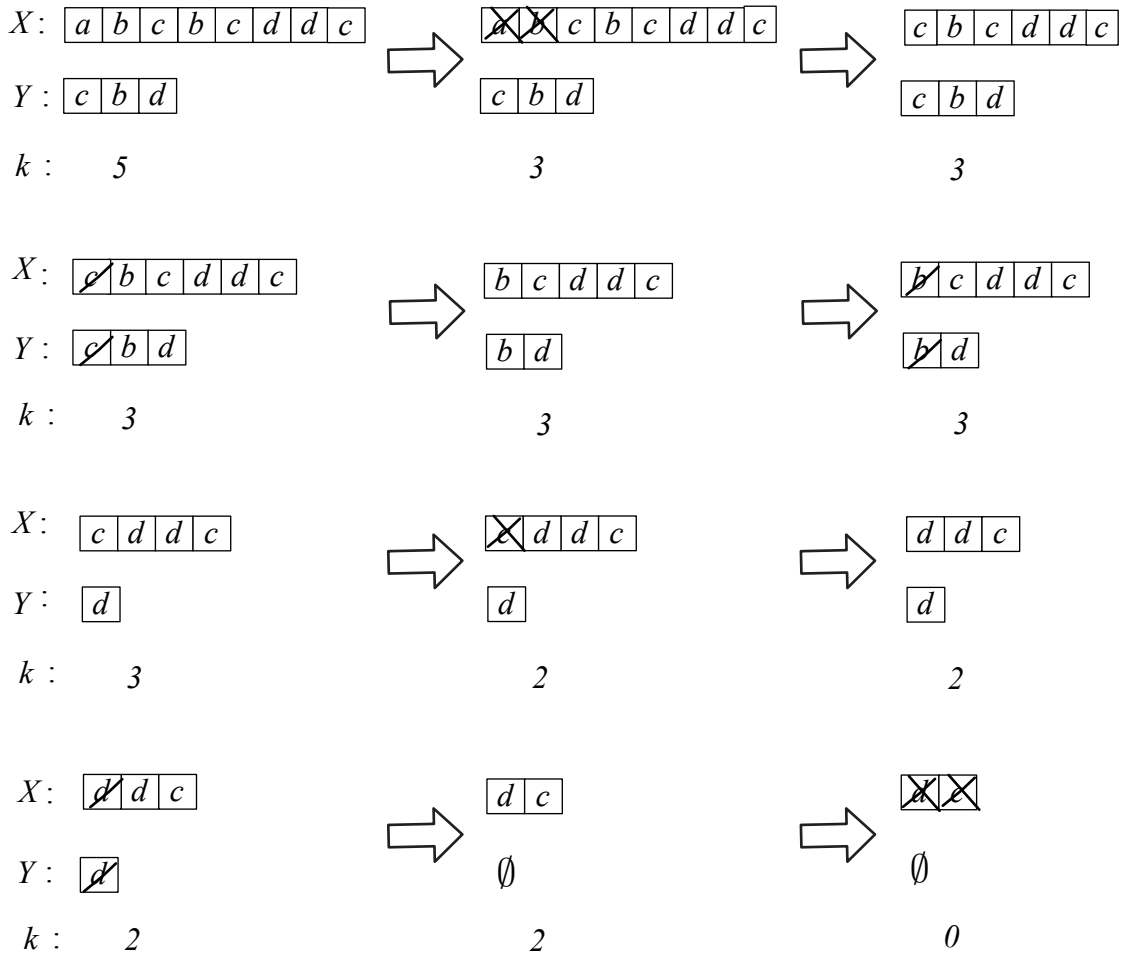

Figure 5.4: Application of the *Algorithm deletesOnly* to the YES-instance, $[abcbcddc, cbd, 5]$. Progression of algorithmic steps begins in the top left corner through to the lower right corner.

5.2 The Experimental Platform

The implementation and testing of the modified Algorithm S2S was initially developed on a MacBook Pro, running Mac OSXv10.6.6. The machine is equipped with a 2.53 GHz Intel Core 2 Duo Processor with 4 GB of RAM. Later, experiments were run on two different machines, namely the MacBook Pro described above and a more

powerful Linux-based machine running Scientific Linux 5.2 (Baron). The Linux machine contains a 2.66 GHz Intel Xeon Quad Core Processor E5430 with 5.37 GB of RAM.

The complete code for the modified algorithm, *stscStress.py*, is available in the Appendix. Python 2.5 was selected as the implementation language due to the ease of working with the Python list structure. X and Y are represented by lists of single characters, whereas ω is a list of ordered triples.

5.2.1 Testing

Verification of proper translation from pseudocode and correctness of *Algorithm S2S* to Python code was accomplished using the following method. We consider a complete bounded binary search tree of height 4, $T(X,\ Y, 4)$, as previously shown in Figure 4.7. Each branch in $T(X,\ Y, 4)$ represents a sequence of edit operations applied to $[X,\ Y,\ k]$. For each branch of $T([X,\ Y, k])$ we construct a YES-instance and a NO-instance, for a total of 32 tests. Each time ω is updated during the construction of $T([X,\ Y, k])$, ω is also recorded into a results text file. Upon completion of running each test case, the results file was examined and the different stages of ω were compared to $T([X,\ Y, k])$. The results files corresponding to each of the 16 input files that were constructed as YES-instances contained all stages of ω up to the classification of $[X,\ Y,\ k]$, including the resulting ω. The 16 input files that were constructed as NO-instances contained every possible ω represented by the branches of $T([X,\ Y, k])$, showing that the implementation of *Algorithm S2S* had indeed constructed the complete $T([X,\ Y, k])$ before classifying $[X,\ Y,\ k]$ as a NO-instance. Tables 5.5 and 5.6 list the problem instance, expected output (manually determined), and resulting ω in YES-instance scenarios.

Figure 5.5: The collection of YES-instance test cases which was used to verify proper translation from pseudocode to Python code. The instance, expected classification and corresponding ω are presented below. For each instance, the classification was determined manually.

X	Y	k	Result	ω
ababcdda	bbdd	4	Yes	$\delta\tau\delta\tau\delta\tau\tau\delta$
abadabcd	bdbdc	4	Yes	$\delta\tau\delta\tau\delta\tau\sigma\tau\tau$
ddbace	bca	4	Yes	$\delta\delta\tau\sigma\tau\tau\delta$
abaddca	bdcad	4	Yes	$\delta\tau\delta\tau\sigma\tau\sigma\tau\tau$
dabaee	aab	4	Yes	$\delta\tau\sigma\tau\tau\delta\delta$
adbabda	dabab	4	Yes	$\delta\tau\sigma\tau\tau\delta\sigma\tau\tau$
bacde	cdb	4	Yes	$\delta\sigma\tau\sigma\tau\tau\delta$
abbdc	cabb	4	Yes	$\delta\sigma\sigma\sigma\tau\tau\tau\tau$
dabcc	ad	4	Yes	$\sigma\tau\tau\delta\delta\delta$
abacdb	baba	4	Yes	$\sigma\tau\tau\delta\delta\sigma\tau\tau$
bacdacd	abdca	4	Yes	$\sigma\tau\tau\delta\tau\sigma\tau\tau\delta$
abcdacb	badbac	4	Yes	$\sigma\tau\tau\delta\tau\sigma\sigma\tau\tau\tau$
abcdaab	badca	4	Yes	$\sigma\tau\tau\sigma\tau\tau\tau\delta\delta$
abdbbacd	babdadc	4	Yes	$\sigma\tau\tau\sigma\tau\tau\delta\tau\sigma\tau\tau$
abcdab	dabcb	4	Yes	$\sigma\sigma\sigma\tau\tau\tau\tau\delta\tau$
aabbab	bababa	4	Yes	$\sigma\sigma\tau\tau\sigma\tau\tau\sigma\tau\tau$

Figure 5.6: The collection of NO-instance test cases which was used to verify proper translation from pseudocode to Python code. The instance, expected classification and corresponding ω are presented below. For each instance, the classification was determined manually.

X	Y	k	Result	ω
bdbdaa	aa	3	No	no ω
abadabcd	bdbdc	3	No	no ω
ddbace	bca	3	No	no ω
abaddca	bdcad	3	No	no ω
dabaee	aab	3	No	no ω
adbabda	dabab	3	No	no ω
bacde	cdb	3	No	no ω
abbdc	cabb	3	No	no ω
dabcc	ad	3	No	no ω
abacdb	baba	3	No	no ω
bacdacd	abdca	3	No	no ω
abcdacb	badbac	3	No	no ω
abcdaab	badca	3	No	no ω
abdbbacd	babdadc	3	No	no ω
abcdab	dabcb	3	No	no ω
aaaab	baaaa	3	No	no ω

5.2.2 The Experiments

A set of experiments was run to ensure that the actual execution times attained were within the expected theoretical running time of $2^k(k+m)$, as determined in Chapter 4. This section outlines how the experiments were executed. The analysis from the results of these experiments is discussed in the following section.

Execution of *stscStress.py* used two additional files, *randomInstanceSet.py* and *TestMasterStress.py*. When executed, *randomInstanceSet.py* generates a series of experiment files each containing Σ, pseudo randomly generated X, pseudo randomly generated Y, and k, each of which occupies its own line of the input file. The user specifies 3 parameters during the execution of *randomInstanceSet.py*, namely: the

number of input files to generate, the size of the parameter k, and the length of Y. The result is a batch of pseudo randomly generated input files. As well, *randomInstanceSet.py* builds *TestMasterStress.py* by appending a new line for each new input file. When *TestMasterStress.py* is executed, each line previously added is used as a command in the terminal to run a separate execution of *stscStress.py* on the corresponding input file. The results are recorded in a csv file, *results.csv*. Figures 5.7 - 5.8 illustrate the methodology.

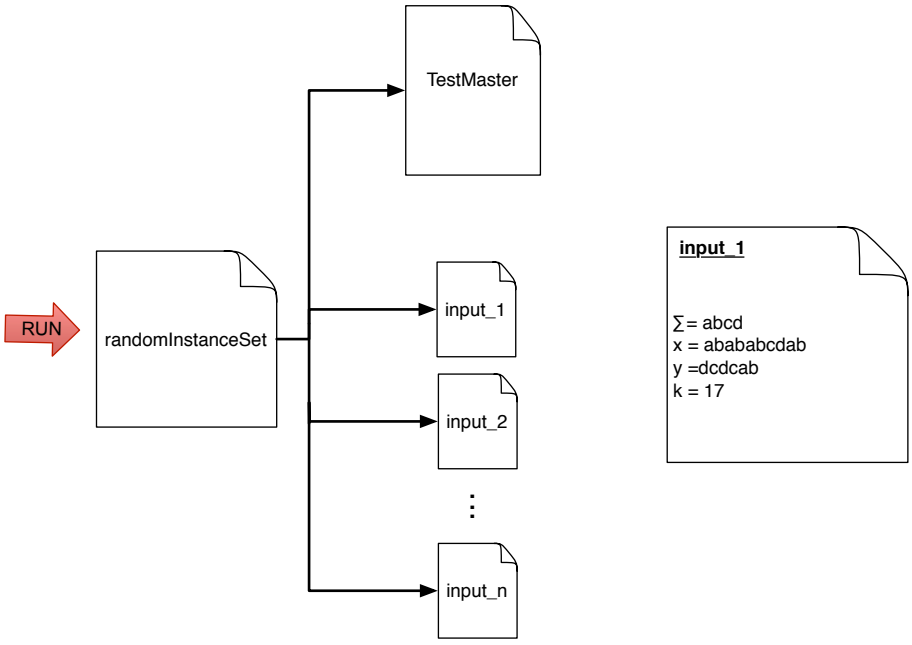

Figure 5.7: When *randomInstanceSet.py* is executed the user supplies the number of input files to generate, the length of Y, and the parameter k. The result is a batch of input files and *TestMasterStress.py*. For each created input file a command to execute *s2scStress.py* using a designated input file is appended to *TestMasterStress.py*.

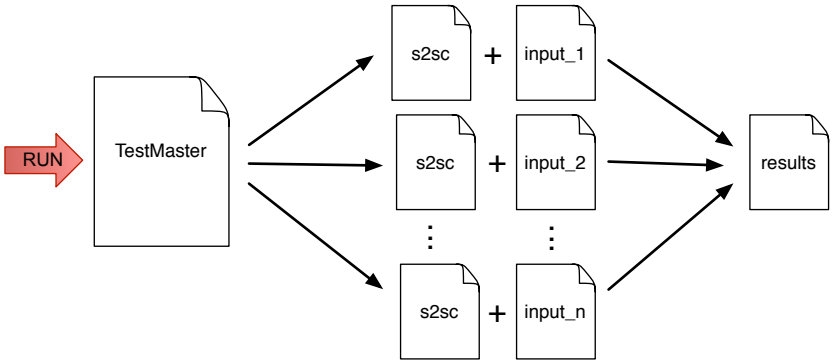

Figure 5.8: Execution of *TestMasterStress.py*. Each line corresponds to a different input file. The results from each execution are appended to *results.csv*.

5.3 Practical Running Time

This section presents a discussion of the results by executing *Algorithm S2S* on 3 batches of experiments. We examine the practical running time of *Algorithm S2S* by randomly generating a collection of instances, given a fixed parameter, and measuring the amount of time required to determine if $[X,\ Y,\ k]$ is a YES-instance or a NO-instance. *Algorithm S2S* was executed using three batches of files, each containing 360 pseudo randomly generated experiments. The batches are distinguishable by the alphabet size. Specifically, they correspond to $|\Sigma| = 2, 13$, and 20. A batch consists of 36 different sets of 10 input files, each representing instances with like characteristics. Within each set, $|Y|$ and k are constant, while $|X|$ varies between files, but never exceeds $|Y| + k$ (from Reduction Rule 2) and is always at least $|Y|$ characters long. For example, a set may have $k = 25$ and $|Y| = 15$, with the values for $|X|$ being $21, 22, 25, 27, 40, 31, 36, 32, 19, 15$. Tables 5.1 through 5.3 summarize the results obtained from running *Algorithm S2S* on the three batches of files. A row in the table represents the results from a set of input files for fixed $|Y|$ and k. $|X|$, $|\omega|$, and execution time are each presented as a range showing the minimum and maximum values. For example, the first row in Table 5.1 represents the execution of 10 different experiments, each with $|Y| = 5$ and $k = 5$. The 10 randomly generated X strings vary from five to ten characters in length. The longest transformation sequence for this set of experiments consists of 3 edit operations. Each of these experiments was completed between $37\mu s$-$166\mu s$. As well, none of the experiments in this set exceeded the maximum amount of memory allocated to the program, i.e., encountered a max-

imum recursion depth Runtime Error (MDR).

Of the 1080 total input files, 15 resulted in an MRD. The number of times an MRD was encountered per set of experiments is shown in the column labeled MRD occurrences in Table 5.1. Notably, MRD only occurred for input files generated where $|\Sigma| = 2$. Instances which require the most recursion depth are those which construct a bounded binary search tree of height k. MRD can result regardless of the classification of $[X,\ Y,\ k]$, that is, building a $T([X,\ Y, k])$ with height k can occur when $[X,\ Y,\ k]$ is a YES-instance or a NO-instance. MRD occurred for the following $(|\Sigma|, |k, |\ Y|)$ triples: $(2, 5, 10)$, $(2, 5, 25)$, $(2, 10, 25)$, $(2, 15, 20)$, $(2, 15, 25)$, $(2, 15, 50)$, $(2, 20, 25)$, $(2, 20, 50)$, $(2, 25, 50)$, and $(2, 35, 50)$. Since k is the bounding factor for the height of $T([X,\ Y, k])$, *Algorithm S2S* has the potential to construct a tree of great height when k is large. Furthermore, if $[X,\ Y,\ k]$ is a NO-instance and is not classified during the first pass through the preprocessing steps, i.e., before the first recursive call, then *Algorithm S2S* attempts to construct the entire tree due to preprocessing. However, it is rare that a complete tree is built. We speculate that MRD results from the attempted construction of a $T([X,\ Y, k])$ with height k.

In Tables 5.2 and 5.3, there are very few input sets that resulted in a non zero value for $|\omega|$. If the entry in the $|\omega|$ column is 0, each of the 10 input files belonging to the set were classified as a NO-instance by *Algorithm S2S*. For $\Sigma = 13$, of the 360 test cases run, only eight were classified as YES-instance's. $|\Sigma| = 20$ had only seven of 360 test cases resulting in YES-instance's. In contrast, for $|\Sigma| = 2$, *Algorithm S2S* encountered 186 YES-instance's. For $|\Sigma| = 13$ and $|\Sigma| = 20$, the YES-instance's occur when k is much greater than Y, specifically for the $(|\Sigma|, k, |\ Y|)$ triples: $(13, 20, 5)$, $(13, 25, 5)$, $(13, 35, 5)$, $(20, 20, 5)$, $(20, 25, 5)$, $(20, 35, 5)$. This can be explained by the fact that although $|\Sigma|$ is not a parameter of the STRING-TO-STRING CORRECTION decision problem, it does greatly affect the likelihood of a YES-instance. If $|\Sigma|$ is small, then the probability that the symbols of X form a superset of Y is much greater than with a larger $|\Sigma|$. Thus, an $[X,\ Y,\ k]$ generated from a restricted Σ has a higher probability of satisfying Condition 5 and of being a YES-instance. On the other hand, a larger $|\Sigma|$ will typically produce $[X,\ Y,\ k]$'s requiring more edits than $[X,\ Y,\ k]$'s generated using a smaller $|\Sigma|$.

Of the 720 test cases with $|\Sigma| = 13$ and $|\Sigma| = 20$, 705 were classified as NO-

instances, 15 were classified as YES-instances. For these two alphabet sizes, the shortest amount of time required to determine YES-instance classification was $162\mu s$ for $|\Sigma| = 13$, and $135\mu s$ for $|\Sigma| = 20$. The longest amount for classification of a yes was $439\mu s$ for $|\Sigma| = 13$, and $714\mu s$ for $|\Sigma| = 20$. The average times to classify a NO-instance when $|\Sigma| = 13$ and $|\Sigma| = 20$ are $893ms921\mu s$ and $494ms710\mu s$.[2] For $|\omega|$ columns in which a 0 appears, the lower bounds on the range represent the duration of time required to classify a NO-instance. For $|\Sigma| = 2$ the average time required to classify a NO-instance is $48s27ms240\mu s$. These values suggest that in these cases, NO-instance classification typically occurs during the preprocessing steps which are performed at root of $T([X, Y, k])$, and that neither a δ or a σ branch need be constructed. Thus, although *Algorithm S2S* is theoretically classified as an exponential running time algorithm, it can be concluded that *Algorithm S2S* has extremely good preprocessing steps, leading to efficient practical running time classifications for many NO-instances.

As $|\Sigma| = 13$ and $|\Sigma| = 20$ yield almost exclusively NO-instances, the remainder of this section provides an analysis of the results obtained from executing the batch of experiments with $|\Sigma| = 2$. Table 5.4 provides a more detailed look at the results for the experiments run with $|\Sigma| = 2$. Each cell in the table represents a set, as previously described, and is populated with the number of YES-instances and MRDs. For example, of the 10 executed input files with $k = 35$ and $|Y| = 50$, 2 input files were classified a YES-instance and 4 input files reached maximum recursion depth. The remaining 4 files are NO-instances. The lower triangular region of Table 5.1 represents the sets of experiments with $|Y| \leq k$, where as the upper triangular region represents the sets of experiments with $k < |Y|$. When the k is greater than $|Y|$ almost all of the input files for the give set are classified as YES-instances. Conversely, when $|Y|$ is greater than k, the number of YES-instances approaches 0. When k and $|Y|$ are similar in value, as represented by the central cells of Table 5.1, the number of YES-instances decreases and NO-instances increases. It can be concluded that when k is greater than $|Y|$, encountering a YES-instance is more probable than encountering a NO-instance, as there are several edits available to transform X into Y. Conversely, if k is less than $|Y|$, the likelihood of a YES-instance decreases as there are fewer edits permitted to transform X into Y.

[2]These values represent the averages attained from all NO-instance's for $|\Sigma| = 13$ and $|\Sigma| = 20$, with the exception of 2 cases per each, with abnormally long running times.

It should be noted that of the 1080 experiments that were run, eight are characterized as having much longer running times than the remaining 1072. Due to the lengthy duration of these experiments, they were executed on the more computationally powerful Linux machine discussed in Section 5.2. Details outlining the exact input files concerned, as well as actual running time attained, are outlined in Appendix A. All other experiments were executed on the MacBook Pro computer.

In summary, this chapter introduced a means for constructing a transformation sequence ω in the event that $[X,\ Y,\ k]$ is a YES-instance, as well as a method for maintaining the correctness of ω as NO-instances classifications are encountered through construction of $T([X,\ Y, k])$. *Algorithm deletesOnly* was also presented. Details pertaining to the testing platform were discussed, including both testing process and the types of input files executed. The chapter concluded with an in depth analysis of the practical running times attained.

Table 5.1: Results corresponding to the batch of experiments with $|\Sigma| = 2$. A set of 10 files is executed for a fixed $k, |Y|$ pair. $|X|$, $|\omega|$ and execution time are expressed as range values. $|Y|$, k and ω remain constant values for the set. The upper bound as calculated by the the theoretical running time of $2^k(k+m)\mu s$ is given in the last column. * indicates that at least one experiment was run on the Linux machine. ** indicates that at least one experiment in the set encountered an MDR.

$\|X\|$	$\|Y\|$	k	$\|\omega\|$	actual execution time	theoretical time $2^k(k+m)\mu s$
5-10	5	5	0-3	$37\mu s$-$166\mu s$	$320\mu s$
11-15**	10	5	0-5	$37\mu s$-$2ms162\mu s$	$480\mu s$
16-20	15	5	0-6	$36\mu s$-$3ms370\mu s$	$640\mu s$
21-25	20	5	0	$36\mu s$-$1ms385\mu s$	$800\mu s$
26-30**	25	5	0	$36\mu s$-$3ms392\mu s$	$960\mu s$
51-55	50	5	0	$36\mu s$-$4ms843\mu s$	$1ms760\mu s$
5-14	5	10	0-10	$37\mu s$-$127\mu s$	$15ms360\mu s$
17-24	10	10	0-9	$37\mu s$-$14ms537\mu s$	$20ms480\mu s$
18-25	15	10	0- 0	$36\mu s$-$448\mu s$	$25ms600\mu s$
21-29	20	10	0-69	$36\mu s$-$94ms550\mu s$	$30ms720\mu s$
25-35**	25	10	0	$36\mu s$-$105ms410\mu s$	$35ms840\mu s$
50-60	50	10	0	$34\mu s$-$145ms853\mu s$	$61ms440\mu s$
6-16	5	15	0-12	$36\mu s$-$134\mu s$	$655ms360\mu s$
10-24	10	15	0-15	$35\mu s$-$277\mu s$	$819ms200\mu s$
15-29	15	15	0-15	$34\mu s$-$1s957ms253\mu s$	$983ms40\mu s$
23-35**	20	15	0-15	$224\mu s$-$3s63ms23\mu s$	$1s146ms880\mu s$
27-40**	25	15	0-22	$36\mu s$-$2s182ms498\mu s$	$1s310ms720\mu s$
50-62**	50	15	0	$35\mu s$-$4s677ms633\mu s$	$2s129ms920\mu s$
12-25	5	20	0-20	$48\mu s$-$139\mu s$	$26s214ms400\mu s$
10-29	10	20	0-20	$36\mu s$-$275\mu s$	$31s457ms280\mu s$
15-32	15	20	0-18	$37\mu s$-$909ms270\mu s$	$36s700ms160\mu s$
21-40	15	20	0-20	$36\mu s$-$1m1s596ms640\mu s$	$36s700ms160\mu s$
25-45	20	20	0-20	$34\mu s$-$145ms853\mu s$	$41s943ms40\mu s$
24-44**	25	20	0-47	$34\mu s$-$1m41s372ms756\mu s$	$47s185ms920\mu s$
53-68**	50	20	0-364	$37\mu s$-$2m19s732ms883\mu s$	$1m13s400ms320\mu s$
6-30	5	25	0-23	$52\mu s$-$163\mu s$	$16m46s632ms960\mu s$
12-34	10	25	0-25	$143\mu s$-$413\mu s$	$19m34s405ms120\mu s$
15-36	15	25	7-22	$200\mu s$-$1ms871\mu s$	$22m22s177ms280\mu s$
24-45	20	25	0-24	$74\mu s$-$690\mu s$	$25m9s949ms440\mu s$
27-47	25	25	0-27	$37\mu s$-$17ms486\mu s$	$27m57s721ms600\mu s$
57-70**	50	25	0	$46\mu s$-$1h23m47s303ms82\mu s$	$41m56s582ms400\mu s$
9-40	5	35	0-33	$35\mu s$-$186\mu s$	$15d21h46m29s534ms720\mu s$
10-43	10	35	0-34	$41\mu s$-$288\mu s$	$17d21h29m48s226ms560\mu s$
16-46	15	35	0-32	$37\mu s$-$516\mu s$	$19d21h13m6s918ms400\mu s$
27-54	20	35	14-35	$210\mu s$-$540\mu s$	$21d20h56m25s610ms240\mu s$
32-52	25	35	19-32	$288\mu s$-$10ms478\mu s$	$23d20h39m44s302ms80\mu s$
53-86*,**	50	35	0-35	$37\mu s$-$3d21h9m40s800ms674\mu s$	$33d9h16m17s761ms280\mu s$

Table 5.2: Results corresponding to the batch of experiments with $|\Sigma| = 13$. A set of 10 files is executed for a fixed $k, |Y|$ pair. $|X|$, $|\omega|$ and execution time are expressed as range values. $|Y|$, k and ω remain constant values for the set. The upper bound as calculated by the the theoretical running time of $2^k(k+m)\mu s$ is given in the last column. Experiments for $|\Sigma| = 13$ did not result in any MRDs so the column has not been included.

$\|X\|$	$\|Y\|$	k	$\|\omega\|$	actual execution time	theoretical time $2^k(k+m)\mu s$
5-10	5	5	0	$35\mu s$-$40\mu s$	$320\mu s$
10-15	10	5	0	$36\mu s$-$80\mu s$	$480\mu s$
15-20	15	5	0	$37\mu s$-$46\mu s$	$640\mu s$
20-25	20	5	0	$35\mu s$-$49\mu s$	$800\mu s$
25-29	25	5	0	$34\mu s$-$37\mu s$	$960\mu s$
50-55	50	5	0	$35\mu s$-$45\mu s$	$1ms760\mu s$
5-14	5	10	0	$34\mu s$-$46\mu s$	$15ms360\mu s$
10-19	10	10	0	$36\mu s$-$44\mu s$	$20ms480\mu s$
16-25	15	10	0	$34\mu s$-$51\mu s$	$25ms600\mu s$
22-30	20	10	0	$34\mu s$-$42\mu s$	$30ms720\mu s$
25-34	25	10	0	$36\mu s$-$41\mu s$	$35ms840\mu s$
51-60	50	10	0	$35\mu s$-$51\mu s$	$61ms440\mu s$
7-20	5	15	0	$35\mu s$-$39\mu s$	$655ms360\mu s$
10-23	10	15	0	$34\mu s$-$153\mu s$	$819ms200\mu s$
16-28	15	15	0	$34\mu s$-$40\mu s$	$983ms40\mu s$
20-35	20	15	0	$35\mu s$-$40\mu s$	$1s146ms880\mu s$
27-39	25	15	0	$35\mu s$-$56\mu s$	$1s310ms720\mu s$
52-63	50	15	0	$34\mu s$-$36\mu s$	$2s129ms920\mu s$
5-24	5	20	0-7	$35\mu s$-$4s895ms558\mu s$	$26s214ms400\mu s$
13-30	10	20	0	$35\mu s$-$5s197ms94\mu s$	$31s457ms280\mu s$
15-34	15	20	0	$36\mu s$-$43\mu s$	$36s700ms160\mu s$
20-40	20	20	0	$35\mu s$-$3s199ms934\mu s$	$41s943ms40\mu s$
27-44	25	20	0	$35\mu s$-$256\mu s$	$47s185ms920\mu s$
53-68	50	20	0	$35\mu s$-$38\mu s$	$1m13s400ms320\mu s$
8-25	5	25	0-25	$34\mu s$-$439\mu s$	$16m46s632ms960\mu s$
10-35	10	25	0	$34\mu s$-$1m36s225ms367\mu s$	$19m34s405ms120\mu s$
20-34	15	25	0	$35\mu s$-$40\mu s$	$22m22s177ms280\mu s$
20-41	20	25	0	$34\mu s$-$40\mu s$	$25m9s949ms440\mu s$
25-47	25	25	0	$34\mu s$-$43\mu s$	$27m57s721ms600\mu s$
53-73	50	25	0	$33\mu s$-$56\mu s$	$41m56s582ms400\mu s$
13-40	5	35	0-23	$36\mu s$-$3m22s442ms896\mu s$	$15d21h46m29s534ms720\mu s$
12-32	10	35	0	$35\mu s$-$49\mu s$	$17d21h29m48s226ms560\mu s$
15-51*	15	35	0	$35\mu s$-$211h51m29s162ms302\mu s$	$19d21h13m6s918ms400\mu s$
28-54	20	35	0	$40\mu s$-$67\mu s$	$21d20h56m25s610ms240\mu s$
32-60	25	35	0	$40\mu s$-$64\mu s$	$23d20h39m44s302ms80\mu s$
53-87	50	35	0	$35\mu s$-$50\mu s$	$33d9h16m17s761ms280\mu s$

Table 5.3: Results corresponding to the batch of experiments with $|\Sigma| = 20$. A set of 10 files is executed for a fixed $k, |Y|$ pair. $|X|$, $|\omega|$ and execution time are expressed as range values. $|Y|$, k and ω remain constant values for the set. The upper bound as calculated by the the theoretical running time of $2^k(k+m)\mu s$ is given in the last column. Experiments for $|\Sigma| = 20$ did not result in any MRDs and so the column has not been included.

$\|X\|$	$\|Y\|$	k	$\|\omega\|$	actual execution time	theoretical time $2^k(k+m)\mu s$
5-10	5	5	0	$36\mu s$-$50\mu s$	$320\mu s$
11-15	10	5	0	$35\mu s$-$38\mu s$	$480\mu s$
15-20	15	5	0	$36\mu s$-$40\mu s$	$640\mu s$
20-24	20	5	0	$36\mu s$-$47\mu s$	$800\mu s$
26-30	25	5	0	$35\mu s$-$40\mu s$	$960\mu s$
50-55	50	5	0	$35\mu s$-$40\mu s$	$1ms760\mu s$
5-15	5	10	0	$35\mu s$-$41\mu s$	$15ms360\mu s$
10-19	10	10	0	$36\mu s$-$70\mu s$	$20ms480\mu s$
15-25	15	10	0	$34\mu s$-$39\mu s$	$25ms600\mu s$
21-29	20	10	0	$35\mu s$-$40\mu s$	$30ms720\mu s$
26-35	25	10	0	$35\mu s$-$50\mu s$	$35ms840\mu s$
53-60	50	10	0	$34\mu s$-$46\mu s$	$61ms440\mu s$
7-20	5	15	0	$34\mu s$-$40\mu s$	$655ms360\mu s$
11-23	10	15	0	$35\mu s$-$38\mu s$	$819ms200\mu s$
16-27	15	15	0	$34\mu s$-$46\mu s$	$983ms40\mu s$
20-32	20	15	0	$36\mu s$-$48\mu s$	$1s146ms880\mu s$
25-40	25	15	0	$35\mu s$-$40\mu s$	$1s310ms720\mu s$
51-65	50	15	0	$34\mu s$-$37\mu s$	$2s129ms920\mu s$
5-25	5	20	0-15	$36\mu s$-$231\mu s$	$26s214ms400\mu s$
13-32	10	20	0	$36\mu s$-$39\mu s$	$31s457ms280\mu s$
15-33	15	20	0	$35\mu s$-$45\mu s$	$36s700ms160\mu s$
22-39	20	20	0	$36\mu s$-$52\mu s$	$41s943ms40\mu s$
25-44	25	20	0	$34\mu s$-$47\mu s$	$47s185ms920\mu s$
53-70	50	20	0	$34\mu s$-$50\mu s$	$1m13s400ms320\mu s$
12-30	5	25	0-20	$36\mu s$-$2m53s629ms627\mu s$	$16m46s632ms960\mu s$
12-34	10	25	0	$35\mu s$-$61\mu s$	$19m34s405ms120\mu s$
16-37	15	25	0	$39\mu s$-$55\mu s$	$22m22s177ms280\mu s$
22-42*	20	25	0	$36\mu s$-$50\mu s$	$25m9s949ms440\mu s$
26-46	25	25	0	$35\mu s$-$54\mu s$	$27m57s721ms600\mu s$
52-72	50	25	0	$34\mu s$-$60\mu s$	$41m56s582ms400\mu s$
8- 35	5	35	0-33	$35\mu s$-$714\mu s$	$15d21h46m29s534ms720\mu s$
10-41	10	35	0	$36\mu s$-$57\mu s$	$17d21h29m48s226ms560\mu s$
16-44	15	35	0	$34\mu s$-$49\mu s$	$19d21h13m6s918ms400\mu s$
23-54	20	35	0	$36\mu s$-$20h26m46s366ms40\mu s$	$21d20h56m25s610ms240\mu s$
29-58	25	35	0	$35\mu s$-$54\mu s$	$23d20h39m44s302ms80\mu s$
53-85	50	35	0	$35\mu s$-$45\mu s$	$33d9h16m17s761ms280\mu s$

Table 5.4: A detailed summary of the results obtained from the experiments with $|\Sigma| = 2$. Each column represents experiments with the fixed k value listed to the far left column, and the set $|Y|$ value in the first row of the table.

	$Y = 5$	$Y = 10$	$Y = 15$	$Y = 20$	$Y = 25$	$Y = 50$
$k = 5$	Yes x 3	Yes x 3 MRD x 1	Yes x 2	Yes x 0	Yes x 0 MRD x 1	Yes x 0
$k = 10$	Yes x 8	Yes x 6	Yes x 6	Yes x 1	Yes x 0 MRD x 1	Yes x 0
$k = 15$	Yes x 6	Yes x 6	Yes x 7	Yes x 6 MRD x 1	Yes x 3 MRD x 1	Yes x 0 MRD x 1
$k = 20$	Yes x 9	Yes x 9	Yes x 7	Yes x 7	Yes x 5 MRD x 1	Yes x 1 MRD x 1
$k = 25$	Yes x 9	Yes x 10	Yes x 10	Yes x 10	Yes x 7	Yes x 0 MRD x 3
$k = 35$	Yes x 8	Yes x 9	Yes x 9	Yes x 10	Yes x 10	No x 4 MRD x 4

Chapter 6

Conclusions

The work in this thesis concentrates on the NP-complete STRING-TO-STRING CORRECTION decision problem restricted to only the swap and delete edit operations. We investigate the parameterized complexity of the problem where the parameter k is the number of permitted edits. We provide the first fixed-parameter tractable result for STRING-TO-STRING CORRECTION by introducing a new algorithm, *Algorithm S2S*, which solves the problem through construction of a bounded binary search tree of height at most k. A series of preprocessing steps is applied to instance $I = [X, Y, k]$ at each node. By recursively preprocessing at each node of the search tree, we were able to efficiently classify several $[X,\ Y,\ k]$'s to be NO-instances without needing to construct the complete bounded binary search tree. The maximum time spent at each node of the search tree results when $[X,\ Y,\ k]$ is reduced via Reduction Rule 7, *Algorithm swapsOnly*. This algorithm runs in $O(k + m)$. The search tree size is $O(2^k)$, therefore STRING-TO-STRING CORRECTION can be solved in $O(2^k k + m)$, showing that the problem is in the class FPT for parameter k. As well, we implemented the search version of the STRING-TO-STRING CORRECTION decision problem by modifying *Algorithm S2S* to include a method of determining the transformation sequence ω in the case that $[X,\ Y,\ k]$ is a YES-instance. To do this we modified the algorithm for solving the case where the only permitted edits are deletes. We remark that the construction of ω did not increase the theoretical run time of *Algorithm S2S*. We verified that most NO-instances that resulted from our experiments could be classified in polynomial time, with respect to the input size of X and Y. This behaviour was verified for $[X,\ Y,\ k]$ with $X \leq 85,\ Y \leq 50$, and $k \leq 35$, for $|\Sigma| \in \{2, 13, 20\}$.

We have contributed to the field of String Correction by presenting the first tractable algorithm for STRING-TO-STRING CORRECTION involving swaps and deletes only. *Algorithm S2S* completes Table 1.1 by addressing string correction restricted to the case involving edit operations swap and deletion, and equivalently swap and insertion.

We presented an improved theoretical running time of $O(1.6181^k m)$ in [1], [2]. The algorithm presented in [2] uses the new *charge and reduce* technique. Future work includes implementing the improved algorithm and comparing the results with those of *Algorithm S2S* to determine whether the theoretical run time improves our current implementation. Other interesting future work investigates improving the theoretical run time of *Algorithm S2S* in further reducing the size of the bounded search tree.

Perhaps the most interesting remaining question is that of solving the open problem of determining a polynomial running time kernelization for the STRING-TO-STRING CORRECTION decision problem, that is, an improved preprocessing of $[X,\ Y,\ k]$ that reduces the instance size such that it can be expressed as a function of k. Efficient preprocessing should not just improve the theoretical running time of *Algorithm S2S*, but also allow the implementation to complete larger and harder instances, since the computationally exhaustive part of the algorithm (the branching portion) would encounter smaller instances.

Bibliography

[1] F.N. Abu-Khzam, H. Fernau, M.A. Langston, S. Lee-Cultura, and U. Stege. A fixed-parameter algorithm for string-to-string correction. In *Proceedings of the 16th Computing: the Australasian Theory Symposium (CATS)*, volume 109 of *Conferences in Research and Practices in Information Technology CRPIT*, pages 31–37, 2010.

[2] F.N. Abu-Khzam, H. Fernau, M.A. Langston, S. Lee-Cultura, and U. Stege. Charge and reduce: A fixed-parameter algorithm for string-to-string correction. *Discrete Optimization*, 8(1):41–49, 2011.

[3] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 39–48, 2000.

[4] R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Springer, 1999.

[5] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1990.

[7] M. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley and Sons Inc., 2002.

[8] R.P. Grimaldi. *Discrete and Combinatorial Mathematics: and applied introduction*. Addison Wesley Longman Inc., 1999.

[9] R.W. Hamming. Error detecting and error correcting codes. *The Bell Systems Technical Journal*, 26(2):147–160, 1950.

[10] J.B. Kruskal. An overview of sequence comparison: Timewarps, string edits, and marcomolecules. *SIAM Journal on Computing*, 25(1):201–237, April 1983.

[11] G.M. Landau, E.W. Myers, and J.P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27:557–582, April 1998.

[12] J. Nerbonne, W. Heeringa, and P. Kleiweg. Edit distance and dialect proximity: Introduction to reissue edition. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, 1999.

[13] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[14] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, January 1974.

[15] R. Wagner and R. Lowrance. An extension of the string-to-string correction problem. *Journal of ACM*, 22(2):177–183, April 1975.

[16] R.A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proceedings of seventh annual ACM symposium on Theory of computing*, STOC'75, pages 218–223, 1975.

Appendix A

Experiments Executed on Linux Machine

Table A.1 gives the details of the eight experiments that were executed on the Linux machine.

Table A.1: Results from the instances that required a considerable amount of time to complete. The two leftmost columns shows the input file name and alphabet size. The problem parameters, $|X|$ and $|Y|$ are given in the following two columns. The table does not have a column indicating the k value, as each experiment had a corresponding k value of 35. The actual execution time and corresponding theoretical running time are provided in the remaining two columns.

input file	$\|\Sigma\|$	$\|X\|$	$\|Y\|$	**actual execution time**	**theoretical time** $2^k(k+m)$
instance_35_50_0	2	85	50	$3d21h9m40s800ms674\mu s$	$33d19h16m17s761ms266\mu s$
instance_35_50_6	2	81	50	$10h24m40s65m739\mu s$	$33d19h16m17s761ms266\mu s$
instance_35_50_8	2	66	50	$3d1h1m17s277ms932\mu s$	$33d19h16m17s761ms266\mu s$
instance_35_15_5	13	50	15	$8d19h51m29s162ms302\mu s$	$19d21h13m6s918ms499\mu s$
instance_35_15_6	13	37	15	$4h59m50s661ms450\mu s$	$19d21h13m6s918ms499\mu s$
instance_35_15_7	13	33	15	$6h3m44s427ms922\mu s$	$19d21h13m6s918ms499\mu s$
instance_35_20_6	20	50	20	$20h26m46s366ms40\mu s$	$21d20h56m25s610ms236\mu s$
instance_35_20_9	20	48	20	$18h46m9s224ms353\mu s$	$21d20h56m25s610ms236\mu s$

Appendix B

Execution

B.1 Instructions to Run Experiments

The 3 python programs; *randomInstanceSet.py*, *TestMasterStress.py* and *stscStress.py*, as well as all input files needed to execute the experiments that were completed as part of this thesis can be found at: http://www.web.uvic.ca/∼sleecult/. A detailed README file explaining the steps needed to execute the provided input files or create and execute a new batch of input files, is also included.

B.2 randomInstanceSet.py

randomInstanceSet.py is responsible for generating a collection of input files based on parameters entered by the user. The user specifies three parameters during the execution of *randomInstanceSet.py*, namely: the number of input files to generate, the size of the parameter k, and the length of Y. The result is a batch of input files each containing Σ, pseudo randomly generated X, pseudo randomly generated Y, and k. In addition, *randomInstanceSet.py* builds the test file *TestMasterStress.py*, by appending a new line for each new input file.

```
#  randomInstances.py
#  Created by serena on 11-05-21.
#  Copyright 2011 __Serena Lee-Cultura. All rights reserved.
#  Purpose: Prompts the user for a parameter and a number of test cases
#           to generate. Creates desired number of test case files

import random

y_string = ""
x_string = ""
#change alphabet below to contain desired characters for string composition
alphabet = 'abcdefghijklmnopqrst'

# read in parameter from command line
k = int(raw_input('enter parameter value: '))
ylength = int(raw_input('enter length of y: '))

# update TestMasterStress on the fly
MASTERFILE = open("TestMasterStress.py", "a")

for count in range(0, 10):
    # generate a new name for the test file
    instanceRG = "testCasesStress/instance_" + str(k) + "_" + str(ylength) + "_" + str
        (count)
    FILE = open(instanceRG, "w")
    # write parameter and alphabet
    FILE.write(str(k) + '\n')
    FILE.write(str(alphabet) + '\n')

    for yi in range(0, ylength):
        # randomly select a char from the alphabet
        y_string += random.choice(alphabet)

    for xi in range(0, random.randint(ylength, k + ylength)):
        # randomly select a char from the alphabet
        x_string += random.choice(alphabet)

    FILE.write(x_string + '\n')
    FILE.write(y_string)
    y_string = ""
    x_string = ""
    FILE.close()
    MASTERFILE.write("subprocess.call([\"python\", \"stscStress.py\", \"")
    MASTERFILE.write(instanceRG + "\"])" + '\n')
    MASTERFILE.write("print \"\\n\"" + '\n')

MASTERFILE.write('\n\n' + "getTimingInfo()" + '\n\n')
MASTERFILE.close()
```

B.3 TestMasterStress.py

TestMasterStress.py responsible for executing a batch of input files in series. Each time that a new input file is created by *randomInstanceSet.py*, *TestMasterStress.py* is appended with a command to execute the corresponding experiment. Experimental results are recorded in a csv file called results.csv.

```
# TestMasterStress.py
# Created by serena on 11-05-21.
# Copyright 2011 __Serena Lee-Cultura. All rights reserved.
# Purpose: to manage testing of multiple test files.

import subprocess
import sys
import os

def getNoAverage():
    resultsFile = open("resultsStressTesting20.csv", "a")
    noTimesFile = open("noTimes", "r")
    noTimesList = []

    while(True):
        timesString = noTimesFile.readline()
        if len(timesString) == 0:
            break
        else:
            timesString = timesString[:-1]
            noTimesList.append(int(timesString))
    total = 0
    for value in noTimesList:
        total += value
    average = total / len(noTimesList)
    noTimesFile.close()
    formattedAverage = elapsedTime(average)
    resultsFile.write( "\n" + "Average Time for a No: " + formattedAverage + "\n" +
        "Number of No:" + str(len(noTimesList)))
    resultsFile.close()

def getYesAverage():
    resultsFile = open("resultsStressTesting20.csv", "a")
    yesTimesFile = open("yesTimes", "r")
    yesTimesList = []

    while(True):
        timesString = yesTimesFile.readline()
        if len(timesString) == 0:
            break
        else:
            timesString = timesString[:-1]
            yesTimesList.append(int(timesString))
    total = 0
    for value in yesTimesList:
        total += value
    average = total / len(yesTimesList)
    yesTimesFile.close()
    formattedAverage = elapsedTime(average)
    resultsFile.write( "\n" + "Yes Average Time: " + formattedAverage + "\n Number of
        Yes: " + str(len(yesTimesList)))
    resultsFile.close()

def getTimingInfo():
    timingInfoFile = open("timingfile", "r+")
    resultsFile = open("resultsStressTesting20.csv", "a")
    runTimesList = []

    xFile = open("x", "r+")
```

```
    xList = []

    omegaFile = open("omega", "r+")
    omegaList = []

    while(True):
        xString = xFile.readline()
        if len(xString) == 0:
            break
        else:
            xString = xString[:-1]
            xList.append(int(xString))
    minxLength = min(xList)
    maxxLength = max(xList)

    while(True):
        omegaSequence = omegaFile.readline()
        omegaString = omegaFile.readline()
        if len(omegaString) == 0:
            break
        else:
            omegaString = omegaString[:-1]
            omegaList.append(int(omegaString))
    minOmega = min(omegaList)
    maxOmega = max(omegaList)

    while(True):
        runTimeString = timingInfoFile.readline()
        if len(runTimeString) == 0:
            break
        else:
            runTimeString = runTimeString[:-1]
            runTime = int(runTimeString)
            runTimesList.append(runTime)

    minRunTime = min(runTimesList)
    minRunTime2 = elapsedTime(minRunTime)
    maxRunTime = max(runTimesList)
    maxRunTime2 = elapsedTime(maxRunTime)

    resultsFile.write( "\n" + " , " + "MIN |X|: " + str(minxLength) + " ,, " + "MAX |X|:
        " + str(maxxLength))
    resultsFile.write( "\n" + " , " + "MIN |w|: " + str(minOmega) + " ,, " + "MAX |w|: "
        + str(maxOmega))
    resultsFile.write( "\n" + " , " + "MIN TIME: " + minRunTime2 + " ,  " + " , " + "MAX
        TIME: " +  maxRunTime2 + "\n")

    os.remove("timingfile")
    os.remove("x")
    os.remove("omega")
    resultsFile.close()

def elapsedTime(microSecCount):
    executionTime = ""

    hrs = microSecCount/3600000000
    if hrs >= 1:
        totalHrs = hrs
        microSecCount = microSecCount - (hrs*3600000000)
```

```
        executionTime = executionTime + str(hrs) + "h "

    mins = microSecCount/60000000
    if mins >= 1:
        totalMins = mins
        microSecCount = microSecCount - (mins*60000000)
        executionTime = executionTime + str(mins) + "m "

    secs = microSecCount/1000000
    if secs >= 1:
        totalSecs = secs
        microSecCount = microSecCount - (secs*1000000)
        executionTime = executionTime + str(secs) + "s "

    milli = microSecCount/1000
    if milli >= 1:
        totalMilli = milli
        microSecCount = microSecCount - (milli*1000)
        executionTime = executionTime + str(milli) + "ms "
    executionTime = executionTime + str(microSecCount) + "us"
    return executionTime

resultsFile = open("resultsStressTesting20.csv", "a")
resultsFile.write("DATE" + ", " + "FILE" + ", " + "|X|" + ", " + "|Y|" + ", " + "k" + ",
    " + "|w|" + ", " + "DURATION" + ", " + "RESULT")
resultsFile.close()

# Randomly generated test cases
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_4"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_0"])
print "\n"
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_25_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_5_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_8"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_4"])
print "\n"
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_25_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_10_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_1"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_8"])
print "\n"
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_25_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_5"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_15_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_1"])
print "\n"
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_9"])
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_25_9"])
```

```
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_5"])
print "\n"
```

```
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_20_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_5_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_5"])
print "\n"
```

```
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_2"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_25_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_25_50_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_5_9"])
print "\n"
```

```
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_10_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_15_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_6"])
```

```
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_20_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_25_9"])
print "\n"
# determine ranges for test case running times
getTimingInfo()
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_0"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_1"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_2"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_3"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_4"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_5"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_6"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_7"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_8"])
print "\n"
subprocess.call(["python", "stscStress.py", "testCasesStress/instance_35_50_9"])
print "\n"
getTimingInfo()
getNoAverage()
getYesAverage()
```

B.4 stsc.py

stsc.py is contains the algorithms for solving the STRING-TO-STRING CORRECTION decision problem restricted to only swaps and deletes. It is executed when the user runs *TestMasterStress.py*.

```
# Author: Serena Lee-Cultura
# Date: Feb 27th 2011

# Implementation of Algorithm S2S, first presented in "A Fixed-Parameter
# Algorithm for String to String Correction", by Abu-Khzam, Fernau, Langston,
# Lee-Cultura, and Stege.

# This implementation treats X as the source and Y as the destination string
# Thus edit operations are only applied to X, and Y is ony modified via
# function tau
# ********************************************************************************


# imported libraries
import datetime
import sys
import time

# global vars
transformationSequence = []     # tracks edits preformed on the string
del_dict = {}                   # tracks number of deletions for each symbol
y_counts_dict = dict()
x_counts_dict = dict()

xCpy = []
yCpy = []
kCpy = []


# functions ********************************************************************

def doubleWrite(string):
    """
        Purpose: Writes the given string to BOTH the console and the output file
        Require: string - the string to print
    """
    print string
    outFile.write(string + "\n")


def transformationSequenceBuilder(editBundle):
    """
        Purpose: Updated the tarnsformationSequence list so that it has a record of
                 the last edit performed: delete, swap, branch, tau
        Require: editBundle - an ordered triple (edit operation, index of X, char)
    """
    global transformationSequence
    transformationSequence.append(editBundle)
    return


def lenthOfTransformationSequence():
    """
        Purpose: returns the length of the transformation sequence, i.e, the number of
                 edits used. This does not include instances of tau
    """
    transformationSequenceCpy = []
    transformationSequenceCpy.extend(transformationSequence)
    i = 0
```

```
    for count in range(0, len(transformationSequenceCpy)):
        editBundle = transformationSequenceCpy.pop()
        if editBundle[0] == 'tau':
            continue
        else:
            i = i + 1
    return i


def phi(string, char):
    """
        Purpose: Finds the first occurence of char in string
        Require: string - the string to search, char - the character to index
        Ensure:  The position of symbol in string, -1 if not found
    """
    i = 0
    for symbol in string:
        if (string[i] == char):
            return i
        i = i + 1
    return -1

# tau related ********************************************************************
def tau(string, updateTransformation):
    """
        Purpose: Removes the head symbol from string
        Require: string - the string to remove the lead element from,
                 updateEditHistory - True if tau called on X, False otherwise.
        Ensure:  string minus the lead element
        Note:    tau needs to only update tarnsformationSequenceBuilder when doing a
                 tau(x), but not for when tau(y) is called
    """
    global x_counts_dict, del_dict, y_counts_dict
    if len(string) == 0:
        pass
    else:
        if updateTransformation == True:
            transformationSequenceBuilder(("tau", 0, string[0]))
            x_counts_dict[string[0]] = x_counts_dict[string[0]] - 1
        else:
            y_counts_dict[string[0]] = y_counts_dict[string[0]] - 1
        string = string[1:len(string)]
    return string


def reverseTau(instance, symbol):
    """
        Purpose: reverses a previous tau operation both strings x and y
        Require: instance - containing x y k, symbol - symbol to insert at index 0
        Ensure: instance with x and y both prepended with symbol at head position
    """
    # unwrap instance
    x = instance[0]
    y = instance[1]
    k = instance[2]
    # prepend both strings with symbol
    x = reverseDelta(x, 0, symbol, True, False)
    y = reverseDelta(y, 0, symbol, False, False)
    return instance
```

```
    for count in range(0, len(transformationSequenceCpy)):
        editBundle = transformationSequenceCpy.pop()
        if editBundle[0] == 'tau':
            continue
        else:
            i = i + 1
    return i


def phi(string, char):
    """
        Purpose: Finds the first occurence of char in string
        Require: string - the string to search, char - the character to index
        Ensure:  The position of symbol in string, -1 if not found
    """
    i = 0
    for symbol in string:
        if (string[i] == char):
            return i
        i = i + 1
    return -1

# tau related ********************************************************************
def tau(string, updateTransformation):
    """
        Purpose: Removes the head symbol from string
        Require: string - the string to remove the lead element from,
                 updateEditHistory - True if tau called on X, False otherwise.
        Ensure:  string minus the lead element
        Note:    tau needs to only update tarnsformationSequenceBuilder when doing a
                 tau(x), but not for when tau(y) is called
    """
    global x_counts_dict, del_dict, y_counts_dict
    if len(string) == 0:
        pass
    else:
        if updateTransformation == True:
            transformationSequenceBuilder(("tau", 0, string[0]))
            x_counts_dict[string[0]] = x_counts_dict[string[0]] - 1
        else:
            y_counts_dict[string[0]] = y_counts_dict[string[0]] - 1
        string = string[1:len(string)]
    return string


def reverseTau(instance, symbol):
    """
        Purpose: reverses a previous tau operation both strings x and y
        Require: instance - containing x y k, symbol - symbol to insert at index 0
        Ensure: instance with x and y both prepended with symbol at head position
    """
    # unwrap instance
    x = instance[0]
    y = instance[1]
    k = instance[2]
    # prepend both strings with symbol
    x = reverseDelta(x, 0, symbol, True, False)
    y = reverseDelta(y, 0, symbol, False, False)
    return instance
```

```
# delta related ***********************************************************************
def superSequence(str1, str2):
    """
        Purpose: Determines if str1 is a supersequence of str2
        Require: x - suspected supersequence, y - the suspected subsequence
        Ensure:  True if x is a supersequence of y, False otherwise.
    """
    str1cpy = str1          # copy x, since you plan to chop it up
    str2cpy = str2          # copy y, same reason
    for char in str2cpy:
        i = phi(str1cpy, str2cpy[0])
        if i < 0:
            return False
        str1cpy = str1cpy[i+1:len(str1cpy)]
        str2cpy = str2cpy[1:len(str2cpy)]
    return True


def delta(string, ind):
    """
        Purpose: Deletes the element at position index from string, updates
                 transformationSequence
        Require: string - to delete from, ind - position of element to delete,
                 bool - True is X, False otherwise
        Ensure:  updated string
        Caller:  called from branching and deltaOnly
        Note:    must determine what string it is manipulating so that it can
                 update the appropriate tracking dictionary
    """
    global x_counts_dict, del_dict
    transformationSequenceBuilder(("delta", ind, string[ind]))
    x_counts_dict[string[ind]] = x_counts_dict[string[ind]] - 1
    del_dict[string[ind]] = del_dict[string[ind]] - 1
    substring1 = string[0:ind]
    substring2 = string[ind + 1: len(string)]
    string = substring1 + substring2
    return string


def reverseDelta(string, ind, symbol, whichString, updateDelDict):
    """
        Purpose: reverses a previous delete operation in given string
                 this function is used to reverse delete and tau
        Require: string - to reverse edit on, ind - index to insert symbol,
                 symbol - symbol to insert
        Ensure:  string with symbol inserted at index
        Note:    This function does not handle incrementing k, since that is not
                 done in both reverse of delete and tau
    """
    global x_counts_dict, del_dict, y_counts_dict

    if ((0 <= ind) & (ind < len(string))):
        string.insert(ind, symbol)
    # boundry case when passed in the empty string
    elif ((0 <= ind) & (ind == len(string))):
        string.insert(ind, symbol)
    else:
```

```
        pass
    if whichString == True:
        x_counts_dict[string[ind]] = x_counts_dict[string[ind]] + 1
    else:
        y_counts_dict[string[ind]] = y_counts_dict[string[ind]] + 1

    if updateDelDict == True:
        del_dict[string[ind]] = del_dict[string[ind]] + 1
    return string


def deltaOnly(instance):
    """
        Purpose: Determines the transformation sequence given an instance that can be
                 solved using only edit operation delete
        Require: [x, y, k]
        Ensure:  a transformation sequence that corrects X to Y
    """
    # unwrap instance
    x = instance[0]
    y = instance[1]
    k = instance[2]
    if len(x) == 0:
        return True
    if len(y) == 0:
        return deltaOnly([delta(x, 0), y, k - 1])
    if y[0] == x[0]:
        return deltaOnly([tau(x, True), tau(y, False), k])
    else:
        return deltaOnly([delta(x, 0), y, k - 1])


# sigma related ***********************************************************************
def sigma(string, ind, regSwap):
    """
        Purpose: Swaps symbols in position ind and ind - 1
        Require: string - to swap characters in, ind - position that is swapped to
                 the left, regSwap - True if swap False if reverseSwap
        Ensure:  string with symbols swapped, transformationSequence is updated
        Caller:  called from branching and sigmaOnly
    """
    if regSwap == True:
        transformationSequenceBuilder(("sigma", ind, string[ind]))
    temp = string[ind-1]
    substring1 = string[0:ind-1]
    substring2 = string[ind: len(string)]
    string = substring1 + substring2
    string.insert(ind, temp)
    return string

def sigmaOnly(instance):
    """
        Purpose: Determines if stsc is possible from instance[0] to instance[1] using
                 only sigma
        Require: instance
        Ensure:  True if stsc is possible, False otherwise
    """
    # unwrap instance
    x = instance[0]
```

```
    y = instance[1]
    k = instance[2]
    if len(x)== 0 and len(y) == 0:
        return True
    i = phi(x, y[0])
    # not enough edits to swap the first occurrence of y_1 in X to first position
    if i > k:
        return False
    # first symbol of X and Y match. Remove head symbol from each string and recurse
    elif i == 0:
        return sigmaOnly([tau(x, True), tau(y, False), k])
    # first occurrence of y_1 in X is located at index i, 0 < i <= k.  sigma is
    # used to move symbol y_1 closer to the first position of X
    else:
        return sigmaOnly([sigma(x, i, True), y, k-1])

def reverseSigma(string, ind):
    """
        Purpose: reverses the previous sigma operation
        Require: string - to reverse characters in, ind - index of char to swap to the
                 right
        Ensure:  string with characters in positions ind and ind - 1 exchanged
    """
    # False argument indicates not to update edit_history
    return sigma(string, ind, False)


def editUndo(instance):
    global transformationSequence, x_counts_dict, y_counts_dict, del_dict
    #unwrap instance
    x = instance[0]
    y = instance[1]
    k = instance[2]
    # if at root of edit tree
    if len(transformationSequence) == 0:
        doubleWrite("\tno edit to reverse")
        return [x, y, k]
    else:
        # update edit_history by popping off last edit
        editBundle = transformationSequence.pop()
        if editBundle[0] == 'delta' or editBundle[0] == 'sigma':
            # update number of edits
            k = k + 1

    # must undo all tail operations, do not increase edits
    if editBundle[0] == 'tau':
        [x, y, k] = reverseTau([x, y, k], editBundle[2])
        [x, y, k] = editUndo([x, y, k])

    # reverse delete and always proceed with swap
    elif editBundle[0] == 'delta':
        x = reverseDelta(x, editBundle[1], editBundle[2], True, True)

    # reverse swap and parent minimum once
    elif editBundle[0] == 'sigma':
        x = reverseSigma(x, editBundle[1])
        # if at root of edit tree
        if len(transformationSequence) != 0:
            [x, y, k] = editUndo([x, y, k])
```

```
    return [x, y, k]


# STSC Algorithm ******************************************************************
def STSC(instance):
    """
        Purpose: A bounded binary search tree algorithm for solving stsc on given
                 instance.
        Require: instance - an ordered triple [X, Y, k], where X is the source, Y
                 is the target, and k an upper bound for the number of edits
        Ensure:  False if X cannot be converted to Y using at most k edits;
                 True otherwise
    """
    global xCpy, yCpy, kCpy

    # unwrap instance
    x = instance[0]
    y = instance[1]
    k = instance[2]

    [xCpy, yCpy, kCpy] = [x, y, k]

    # Preprocessing phase of the STSC algorithm. [X, Y, k] is assessed to see if
    # it can be classified as a YES-instance or a NO-instance in polynomial time.

    # rr: number of edits is required to be nonnegative
    if k < 0:
        return False

    # rr: stsc is not possible since X is shorter than Y and symbol insertion
    #     is not permitted.
    if len(x) < len(y):
        return False

    # rr: verify that X contains the minimum number of each symbol needed to
    #     complete stsc.
    for char in y:
        if x_counts_dict[char] < y_counts_dict[char]:
            [x, y, k] = [xCpy, yCpy, kCpy]
            return False

    if len(y) > 0:
        # rr: the first symbol in X matches the first symbol in Y. The head symbol
        #     of each string is removed via the tau function. phi(X, a) returns the
        #     index os the first occurence of symbol a in X.
        if phi(x, y[0]) == 0:
            return STSC([tau(x, True), tau(y, False), k])

    # rr: if each symbol in Y appears in X in the same order as in Y, and the
    #     difference in length between the two strings is bound above by k,
    #     then the only edit operation delta is necessary.  superSequence(X, Y)
    #     returns True if X is a superSequence of Y, False otherwise.
    if superSequence(x, y):
        if len(x) - len(y) <= k:
            return deltaOnly([x, y, k])
        else:
            return False

    # rr: as long as both X and Y are of equal length and have the same number
```

```
    #     of each symbol, then only edit operation sigma is necessary. the number
    #     of occurrences of each symbol is tracked by del_dict
    if len(x) == len(y):
        for key in del_dict:
            if del_dict[key] != 0:
                return False
        if sigmaOnly([x, y, k]) == True:
            return True
        else:
            [x, y, k] = [xCpy, yCpy, kCpy]
            return False

    # Branching portion of the algorithm.  Both delta and sigma are permitted.
    # The algorithm branches on a choice to either delete or swap the symbol
    # directly before phi(X, y_1)
    if STSC([delta(x, phi(x, y[0]) - 1), y, k - 1]):
        return True
    [x, y, k] = editUndo([xCpy, yCpy, kCpy])
    return STSC([sigma(x, phi(x, y[0]), True), y, k - 1])


def writeToTimingFile(timingfile, timeSpent, transformationSequencefile, possible, xfile,
    x):
    days = timeSpent.days
    seconds = timeSpent.seconds
    microseconds = timeSpent.microseconds

    daysInMicroseconds = days * 86400000000
    secondsInMicroseconds = seconds * 1000000
    totalMicroseconds = daysInMicroseconds + secondsInMicroseconds + microseconds

    timingfile.write(str(totalMicroseconds) + "\n")

    if (possible == True):
        transformationSequencefile.write(str(transformationSequence) + "\n")
        transformationSequencefile.write(str(lenthOfTransformationSequence()) + "\n")

    else:
        transformationSequencefile.write("[]" + "\n")
        transformationSequencefile.write("0" + "\n")
    xfile.write(str(len(x)) + "\n")



def writeInterestingResults(resultfile, inputfile, x_str, y_str, num_edits_str,
    lenthTransSeq, runTime, act_res):
    """
        Purpose: writes the results of the algorithm into a csv file
    """
    now = datetime.datetime.now()
    days = runTime.days
    seconds = runTime.seconds
    microseconds = runTime.microseconds

    daysInMicroseconds = days * 86400000000
    secondsInMicroseconds = seconds * 1000000
    totalMicroseconds = daysInMicroseconds + secondsInMicroseconds + microseconds
    print "writeInterestingResults ~ totalMicroseconds : " + str(totalMicroseconds)
    timeSpent = elapsedTime(totalMicroseconds)
```

```
    if act_res == True:
        act_res = "Possible"
    else:
        act_res = "Not Possible"
    x_str = x_str[:-1]
    num_edits_str = num_edits_str[:-1]
    resultfile.write("\n" + now.strftime("%d-%m-%d") + ", " + inputfile + ", " + str(len
        (x_str)) + ", " + str(len(y_str)) + ", " + num_edits_str + ", " + str
        (lenthTransSeq) + ", " + timeSpent + ", " + act_res + "\n")
    return


def elapsedTime(microSecCount):
    executionTime = ""
    print "Elapsed Time ~ microSecCount : " + str(microSecCount)
    hrs = microSecCount/3600000000
    if hrs >= 1:
        totalHrs = hrs
        microSecCount = microSecCount - (hrs*3600000000)
        executionTime = executionTime + str(hrs) + "h "

    mins = microSecCount/60000000
    if mins >= 1:
        totalMins = mins
        microSecCount = microSecCount - (mins*60000000)
        executionTime = executionTime + str(mins) + "m "

    secs = microSecCount/1000000
    if secs >= 1:
        totalSecs = secs
        microSecCount = microSecCount - (secs*1000000)
        executionTime = executionTime + str(secs) + "s "

    milli = microSecCount/1000
    if milli >= 1:
        totalMilli = milli
        microSecCount = microSecCount - (milli*1000)
        executionTime = executionTime + str(milli) + "ms "
    executionTime = executionTime + str(microSecCount) + "us"
    return executionTime


# main ********************************************************************
# set up problem: file io, string validation, dictionary set up

# opens the input/output file for reading and writing
input = sys.argv[1]
print sys.argv[1]
inFile = open(input, "r")
outFile = open("outputStress", "w")
resultsFile = open("resultsStressTesting20.csv", "a")
timingFile = open("timingfile", "a")
omegaFile =open("omega", "a")
xFile = open("x", "a")

noTimesFile = open("noTimes", "a")
yesTimesFile = open("yesTimes", "a")

# read in each line of the text file to be processed
```

```
numEditsString = inFile.readline()                          # number of edit operation
alphaString = inFile.readline()                             # symbols in the alphabet
xx_string = inFile.readline()                               # destination string
yy_string = inFile.readline()                               # source string
resultString = inFile.readline()                            # expected result

resultString = resultString[:-1]
k = int(numEditsString)                                     # convert string to int

# create a list out of the alphabet, src, and dst string
alpha = []                                                  # create empty list
alpha.extend(alphaString)                                   # convert string into list
alpha = alpha[:-1]                                          # removes newline

# x is the string we want to convert into y
x = []                                                      # create empty list
x.extend(xx_string)                                         # convert string to list
x = x[:-1]                                                  # removes newline

# for each string create dictionary with (key-char, val-# of instances of char)
# assign the value 0 for each symbol
for chars in alpha:
    x_counts_dict[chars] = 0

# update value to be chars symbol count in string x
for chars in x:
    x_counts_dict[chars] = x_counts_dict[chars] + 1
    if not chars in alpha:
        exit(0)

# y is the destination string
y = []                                                      # create empty list
y.extend(yy_string)                                         # convert string to list
y = y[:-1]                                                  # remove newline

# assign the value 0 for each symbol
for chars in alpha:
    y_counts_dict[chars] = 0

# update value to be chars symbol count in string y
for chars in y:
    y_counts_dict[chars] = y_counts_dict[chars] + 1
    if not chars in alpha:
        exit(0)

# initialize deletion Counts for each char in the alphabet
# del_dict = {}
for char in alpha:
    del_dict[char] = 0

for char in alpha:
    del_dict[char] = x_counts_dict[char] - y_counts_dict[char]


# bundle up strings and number of edits together for easy passing
instance = [x, y, k]
startTime = datetime.datetime.now()
#startus = startTime.microsecond
startTime.strftime("%H:%M:%S")
```

```
numEditsPerformed = 0

try:
    if STSC(instance) == True:
        endTime = datetime.datetime.now()
        runTimeUS = endTime - startTime
        writeToTimingFile(timingFile, runTimeUS, omegaFile, True, xFile, xx_string)
        numEditsPerformed = lenthOfTransformationSequence()
        writeInterestingResults(resultsFile, input, xx_string, yy_string, numEditsString,
            numEditsPerformed, runTimeUS, True)

        days = runTimeUS.days
        seconds = runTimeUS.seconds
        microseconds = runTimeUS.microseconds
        daysInMicroseconds = days * 86400000000
        secondsInMicroseconds = seconds * 1000000
        totalMicroseconds = daysInMicroseconds + secondsInMicroseconds + microseconds

        yesTimesFile.write(str(totalMicroseconds) + "\n")

        doubleWrite("Correction Possible")
    else:
        endTime = datetime.datetime.now()
        runTimeUS = endTime - startTime
        writeToTimingFile(timingFile, runTimeUS, omegaFile, False, xFile, xx_string)
        writeInterestingResults(resultsFile, input, xx_string, yy_string, numEditsString,
            numEditsPerformed, runTimeUS, False)
        days = runTimeUS.days
        seconds = runTimeUS.seconds
        microseconds = runTimeUS.microseconds
        daysInMicroseconds = days * 86400000000
        secondsInMicroseconds = seconds * 1000000
        totalMicroseconds = daysInMicroseconds + secondsInMicroseconds + microseconds

        noTimesFile.write(str(totalMicroseconds) + "\n")

        doubleWrite("Correction Not Possible")
except RuntimeError:
    resultsFile.write("\n" + "," + "," + "RuntimeError: " + input + " maximum recursion
        depth" + "\n")

# close all files ************************************************************
inFile.close()
outFile.close()
resultsFile.close()
timingFile.close()
omegaFile.close()
xFile.close()

noTimesFile.close()
yesTimesFile.close()
exit(0)
```