The Binary String-to-String Correction Problem

by

Thomas D. Spreen
B.Sc., University of Victoria, 2010

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

MASTER OF SCIENCE

in the Department of Computer Science

The Binary String-to-String Correction Problem

by

Thomas D. Spreen
B.Sc., University of Victoria, 2010

Supervisory Committee

Dr. F. Ruskey, Co-supervisor
(Department of Computer Science)

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

ABSTRACT

Supervisory Committee

Dr. F. Ruskey, Co-supervisor
(Department of Computer Science)

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

String-to-String Correction is the process of transforming some *mutable string* M into an exact copy of some other string (the *target string* T), using a shortest sequence of well-defined edit operations. The formal STRING-TO-STRING CORRECTION problem asks for the optimal solution using just two operations: symbol deletion, and swap of adjacent symbols. String correction problems using only swaps and deletions are computationally interesting; in his paper *On the Complexity of the Extended String-to-String Correction Problem* (1975), Robert Wagner proved that the String-to-String Correction problem under swap and deletion operations only is NP-complete for unbounded alphabets.

In this thesis, we present the first careful examination of the binary-alphabet case, which we call Binary String-to-String Correction (BSSC). We present several special cases of BSSC for which an optimal solution can be found in polynomial time; in particular, the case where T and M have an equal number of occurrences of a given symbol has a polynomial-time solution. As well, we demonstrate and prove several properties of BSSC, some of which do not necessarily hold in the case of String-to-String Correction. For instance: that the order of operations is irrelevant; that symbols in the mutable string, if swapped, will only ever swap in one direction; that the length of the Longest Common Subsequence (LCS) of the two strings is monotone nondecreasing during the execution of an optimal solution; and that there exists no correlation between the effect of a swap or delete operation on LCS, and the optimality of that operation. About a dozen other results that are applicable to Binary String-to-String Correction will also be presented.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Dr. Frank Ruskey and Dr. Ulrike Stege, for their encouragement and support, and especially for their patience; and my father, Dr. Otfried Spreen, for inspiration.

Nothing in the world can take the place of persistence. Talent will not; nothing is more common than unsuccessful men with talent. Genius will not; unrewarded genius is almost a proverb. Education will not; the world is full of educated derelicts. Persistence and determination alone are omnipotent.

Calvin Coolidge

DEDICATION

For my wife Jaime and my daughter Avery

Chapter 1

Introduction

String-to-string correction is the process of transforming one string M (the *mutable string*) into an exact copy of another string T (the *target string*) using a set of well-defined edit operations. For example, one might attempt to transform the string $M =$ '`EXPEALIDOCIOUS`' into the string $T =$ '`COOLEX`'. Importantly, the target string T is never altered by any operation; thus, the problem is not one of morphing both strings into an identical state, but of transforming the mutable string M into an exact copy of the target string.

Whether such a transformation is possible depends on input and the types of operations allowed. Four commonly defined operations in the literature are *deletion*, *insertion*, *substitution*, and *adjacent symbol interchange* (also referred to as *swap*). We will demonstrate these four operations by example:

Deletion: $M =$ '`EXPEALIDOCIOUS`' $\longrightarrow$ $M' =$ '`EXEALIDOCIOUS`'
(symbol '`P`' is deleted)

Insertion: $M =$ '`EXPEALIDOCIOUS`' $\longrightarrow$ $M' =$ '`EXPEALIDOYCIOUS`'
(new symbol '`Y`' is added)

Substitution: $M =$ '`EXPEALIDOCIOUS`' $\longrightarrow$ $M' =$ '`EXPEALITOCIOUS`'
(symbol '`D`' replaced with new symbol '`T`')

Swap: $M =$ ‘EXPEALIDOCIOUS’ $\longrightarrow$ $M' =$ ‘EPXEALIDOCIOUS’
(adjacent symbols ‘XP’ swap positions, becoming ‘PX’)

We note that all of these sample operations were performed on the mutable string M, never on the target string T.

Typically some proper subset of the four operations is used as the set of allowable operations for a particular instance (for example: swaps, insertions and deletions, or just swaps and deletions). Some instances require the use of certain operations in order to be solvable. For example, transformation of $M =$ ‘YANKEE’ into an exact copy of $T =$ ‘KEYS’ is impossible unless insertion or substitution is allowed, since no ‘S’ symbol exists in M.

The next important concept to discuss is that of *optimality*. An optimal solution is one that transforms M into an exact copy of T using a minimum possible number of operations. We will demonstrate optimality with a worked example.

1.1 An Example

We return to our familiar example instance with $T =$ ‘COOLEX’ and $M =$ ‘EXPEALIDOCIOUS’ under the standard 26-symbol English alphabet, and consider possible solutions using just two allowable operations: deletion and swap. Since the insertion and substitution operations are not allowed for this instance, we must be careful with our deletions. In particular we require that one ‘C’, two ‘O’, one ‘L’, one ‘E’ and one ‘X’ symbol are preserved in the string M, so that we may then rearrange them with swap operations and eventually arrive at the target string ‘COOLEX’.

Suppose we selected the underlined symbols, and decided to delete the remaining (gray-coloured) symbols, as follows:

$M =$ ‘EXPEALIDOCIOUS’

With these decisions made, the solution would then proceed as follows:

00	‘EXPEALIDOCIOUS’	the original mutable string M
01	‘XPEALIDOCIOUS’	deletion of first ‘E’ symbol
02	‘XEALIDOCIOUS’	deletion of ‘P’ symbol
03	‘XELIDOCIOUS’	deletion of ‘A’ symbol
04	‘XELDOCIOUS’	deletion of first ‘I’ symbol
05	‘XELOCIOUS’	deletion of ‘D’ symbol
06	‘XELOCOUS’	deletion of ‘I’ symbol
07	‘XELOCOS’	deletion of ‘U’ symbol
08	‘XELOCO’	deletion of ‘S’ symbol
09	‘EXLOCO’	swap of adjacent symbols ‘XE’
10	‘EXLCOO’	swap of adjacent symbols ‘OC’
11	‘EXCLOO’	swap of adjacent symbols ‘LC’
12	‘ECXLOO’	swap of adjacent symbols ‘XC’
13	‘CEXLOO’	swap of adjacent symbols ‘XC’
14	‘CEXOLO’	swap of adjacent symbols ‘LO’
15	‘CEOXLO’	swap of adjacent symbols ‘XO’
16	‘COEXLO’	swap of adjacent symbols ‘EO’
17	‘COEXOL’	swap of adjacent symbols ‘LO’
18	‘COEOXL’	swap of adjacent symbols ‘XO’
19	‘COOEXL’	swap of adjacent symbols ‘EO’
20	‘COOELX’	swap of adjacent symbols ‘XL’
21	‘COOLEX’	swap of adjacent symbols ‘EL’

Thus we have found a solution with cost (also sometimes called *distance*) of 21: 8 deletions and 13 swaps.

Is this the best we can do to mutate ‘EXPEALIDOCIOUS’ into ‘COOLEX’? Suppose instead we selected the following set of underlined symbols to save, and decided to delete the remaining symbols:

$M =$ ‘EXPEALIDOCIOUS’

This is the exact same set of deletions as our first attempt above, except that this time we have elected to save the first (leftmost) 'E' symbol and delete the second one, instead of the other way around. Under these conditions, the solution would then be executed as follows:

00	'EXPEALIDOCIOUS'	the original mutable string M
01	'EXPALIDOCIOUS'	deletion of second 'E' symbol
02	'EXALIDOCIOUS'	deletion of 'P' symbol
03	'EXLIDOCIOUS'	deletion of 'A' symbol
04	'EXLDOCIOUS'	deletion of first 'I' symbol
05	'EXLOCIOUS'	deletion of 'D' symbol
06	'EXLOCOUS'	deletion of 'I' symbol
07	'EXLOCOS'	deletion of 'U' symbol
08	'EXLOCO'	deletion of 'S' symbol

At this point using only deletions, we have already achieved an interim state for which we also required a swap on our first attempt.

09	'EXLCOO'	swap of adjacent symbols 'OC'
10	'EXCLOO'	swap of adjacent symbols 'LC'
11	'ECXLOO'	swap of adjacent symbols 'XC'
12	'CEXLOO'	swap of adjacent symbols 'XC'
13	'CEXOLO'	swap of adjacent symbols 'LO'
14	'CEOXLO'	swap of adjacent symbols 'XO'
15	'COEXLO'	swap of adjacent symbols 'EO'
16	'COEXOL'	swap of adjacent symbols 'LO'
17	'COEOXL'	swap of adjacent symbols 'XO'
18	'COOEXL'	swap of adjacent symbols 'EO'
19	'COOELX'	swap of adjacent symbols 'XL'
20	'COOLEX'	swap of adjacent symbols 'EL'

This time, the transformation of M into T occurred with a cost of 20, not 21 as before.

In fact, this 20-step solution is optimal.

Assuming that a given string pair $\langle T, M \rangle$ is such that the mutable string M has sufficient symbols to be transformed into the target string T by the operations of swap and deletion, transformation of the mutable string into the target string can be performed in quadratic time if no limit is placed on cost. However, finding an *optimal* (minimum cost) solution can be a more difficult computation; the complexity is dependent on the types of operations allowed, as the next section shows.

1.2 Complexity of the General Problem

Given the general string-to-string correction problem with an arbitrary alphabet and the four operations of deletion, insertion, substitution and swap, there are fifteen different combinations of these operations, such as substitution only, or deletion and substitution. Each combination has been previously studied and has a known complexity.

In 2011, Lee-Cultura [9] compiled a list of known complexity results for optimally solving String-to-String Correction given all nonempty, distinct-element subsets of the four common edit operations. A condensed version is reproduced here as Table 1.1 on the following page.

The formal STRING-TO-STRING CORRECTION problem ([SR20] in Garey/Johnson [5]), asks for an optimal solution using the two operations of deletion and swap. This corresponds to row 7 of Table 1.1, and the problem under these conditions was shown to be NP-complete by Wagner and Fischer [17] using a reduction from SET-COVERING. As such there likely exists no polynomial-time algorithm for determining minimum distance between T and M, as there is under the operations of substitution only (Hamming Distance [6]); insertion, deletion and substitution (Levenshstein Distance [10]); and insertion, deletion, substitution and swap (Damerau-Levenshstein Distance [3], or Extended String-to-String Correction [11]).

	Allowed Operations	Complexity	Source
01	Deletion only	$O(m)$	see Section 2.3.3
02	Insertion only	$O(n)$	see note 1
03	Substitution only	$O(n)$	see note 2
04	Swap only	$O(n^2)$	see Section 2.3.2
05	Deletion, Insertion	$O(nm)$	Bergroth et al. [2]
06	Deletion, Substitution	$O(nm)$	Wagner [16]
07	Deletion, Swap	NP-complete	Wagner and Fischer [17]
08	Insertion, Substitution	$O(nm)$	Wagner [16]
09	Insertion, Swap	NP-complete	Wagner and Fischer [17]
10	Substitution, Swap	$O(nm)$	Wagner [16]
11	Deletion, Insertion, Substitution	$O(nm)$	Bergroth et al. [2]
12	Deletion, Insertion, Swap	$O(nm)$	Wagner [16]
13	Deletion, Substitution, Swap	$O(nm)$	Wagner [16]
14	Insertion, Substitution, Swap	$O(nm)$	Wagner [16]
15	Deletion, Insertion, Substitution, Swap	$O(nm)$	Wagner [16]

Table 1.1: Running times for String-to-String Correction considering all 15 nonempty, distinct-element subsets of the set of four common edit operations. The variables n and m represent the string lengths of T and M respectively.

Note 1: This thesis does not explore the insertion operation with respect to BSSC, but we note that if the only operation allowed is insertion, then the strings T and M can be exchanged, and the deletion algorithm in Section 2.3 can be used to solve the instance with the exact same cost.

Note 2: This thesis does not explore the substitution operation with respect to BSSC, but we note that in the case that substitution is the only operation allowed, a simple linear-time algorithm can be constructed which scans both strings and replaces any symbol M_i in M which does not match its corresponding symbol T_i in T.

1.3 Motivation and Expected Outcomes

The binary-alphabet case of the String-to-String Correction problem under swaps and deletions (which we call Binary String-to-String Correction, or BSSC) has not been carefully scrutinized prior to this thesis. Our motivation is threefold. First, intuition suggests that the binary case may turn out to be simpler than the general problem, and therefore many cases (if not all) might be found that have a polynomial running time. Secondly, a careful study of BSSC may provide some insight on the general problem that will be of use in future research. Thirdly, binary-alphabet problems are inherently interesting, have many surprising properties, and have many applications (real and potential) in computer science.

We demonstrate several cases of BSSC for which a polynomial-time algorithm exists. For example, we will present an algorithm that optimally solves all cases for which T and M have an equal number of `'1'` symbols in low-order polynomial time, regardless of the number of `'0'` symbols in T or M or the length of the strings. A sample instance taking this form might be:

T : `'010010010001110'`
M : `'10010001000110000001'`

This is one of many forms of T and/or M which have polynomial-time optimal solvers.

Another important result we present, alluded to earlier, is independence of operations. In the general problem, there are many cases for which any required deletions must be performed first, before the swaps; this minimizes the distance required to swap, since a symbol that is to be deleted can never be involved in a swap operation (otherwise the cost becomes greater than optimal). Under BSSC, however, we will show that no such cases need ever occur; the deletions do not have to be performed first to guarantee the optimality of a solution.

This thesis will also consider Longest Common Subsequences (LCS) of T and M in detail (in Chapter 2), and we will present several results involving LCS. The reason for this is as follows. Consider the length of the LCS of T and M during the execution of an optimal solution, examining its length after each operation. For nontrivial instances, the LCS will begin as a subsequence that is shorter than T, and will ultimately become equivalent to

the entire string T after all swaps are completed. Thus intuition tells us to examine LCS closely as a bellwether of optimality. One result in this thesis actually shows that for a given instance involving strings T and M, the effect with respect to the LCS of T and M of a given single operation does not provide useful information about the optimality of that operation. However, we also show that an optimal solution can always be found for which the length of the LCS is monotone nondecreasing as the solution is implemented.

1.4 Organization

The organization of the thesis is as follows:

- Chapter 2 describes String-to-String Correction (S2SC) and Binary String-to-String Correction (BSSC) in detail; and previous literature regarding S2SC is surveyed.
- Chapter 3 presents several new theoretical results regarding BSSC.
- Chapter 4 utilizes many of the results from Chapter 3 to provably demonstrate that several instances of BSSC have polynomial-time solutions.
- Chapter 5 summarizes the results from this thesis, and discusses some avenues for future work on the problem.
- Appendix A describes selected implementations which were developed and utilized while researching the BSSC problem.
- Appendix B comprehensively lists definitions of terms and symbols used in this thesis, for easy reference.

Chapter 2

String-to-String Correction

The Binary String-to-String Correction Problem (BSSC) is a special case of the String-to-String Correction Problem (S2SC). We first provide a description of S2SC, and then describe BSSC in detail. The reader should note that Appendix B in this thesis provides a complete definition of terms and symbols, which can be used for easy reference.

2.1 The General Problem

A *string* is an ordered sequence of n symbols, selected (usually with repetitions allowed) from a specified set of symbols called an *alphabet*; for instance $X =$ `'COMPUTER SCIENCE'` is a string under the alphabet $\Sigma =$ {`'A', 'B', ..., 'Z', ' '`}. Some previous works on the String-to-String Correction problem deal with unbounded alphabets and/or infinite-length strings [16]; in this thesis we will deal only with finite-length alphabets and strings.

Each symbol in a string is distinguishable by an index subscript, counting from left to right and starting at index 1; for example, X_3 is the 3rd symbol from the left in string X. Thus if $X =$ `'COMPUTER SCIENCE'`, $X_3 =$ `'M'`.

An *instance* of String-to-String Correction is a pair of strings: T, the *target string*, and M, the *mutable string*. Such an instance is denoted $\langle T, M \rangle$.

Given an instance $\langle T, M \rangle$, the String-to-String Correction problem asks for the minimum number of operations required to transform M into T using only deletion and swap operations. Recall from Chapter 1 that this subset of the four standard edit operations is interesting due to the fact that under these operations, S2SC is an NP-complete problem. Thus deletion and swap will be the only operations considered in this thesis henceforth.

We denote deletion of symbol X_i from string X by $\delta(X, i)$, or just $\delta(i)$ if the string context is not ambiguous. Denote the swap of symbols X_i, X_{i+1} by $\sigma(X, i)$ or $\sigma(i)$. Swaps are performed on adjacent symbols only.

For example: given $X =$ '`EXAMPLE`', the operation $\delta(X, 4)$ yields $X' =$ '`EXAPLE`'; and $\sigma(X', 3)$ yields $X'' =$ '`EXPALE`'.

Both deletion and swap incur a cost of 1 per operation, and the minimum possible number of operations is considered optimal. We note that since both operations incur unit cost, the terms *cost*, *number of operations*, and occasionally *distance* may be used interchangeably.

2.1.1 Definition of a Solution

We next formally define a solution to an instance.

The sequence $(M^{(0)}, M^{(1)}, \ldots, M^{(C)})$ is a *solution* to the instance $\langle T, M \rangle$ if $M^{(0)} = M$, $M^{(C)} = T$, and for any given $M^{(i)}$, $0 \leq i < C$, $M^{(i+1)}$ differs from $M^{(i)}$ by exactly one deletion or swap operation; i.e. $M^{(i+1)}$ is the string that results after operation $\delta(M^{(i)}, j)$ for some j, $1 \leq j \leq m$, or after operation $\sigma(M^{(i)}, j)$ for some j, $1 \leq j \leq m-1$. We denote this sequence $(M^{(0)}, M^{(1)}, \ldots, M^{(C)})$ by $S(T, M) = (M^{(0)}, M^{(1)}, \ldots, M^{(C)})$.

For example: if we have an instance $\langle T, M \rangle$ with $T =$ '`AB`' and $M =$ '`BXA`', one solution is $S(T, M) =$ ('`BXA`', '`BA`', '`AB`'). The operations used in effecting this solution are $\delta(M^{(0)}, 2)$ and $\sigma(M^{(1)}, 1)$.

The *cost* of a solution S to an instance $\langle T, M \rangle$, which is equal to the number of opera-

tions required to transform T into M, is denoted by $|S(T, M)|$.

Note that for any two solutions $S_1(T, M)$ and $S_2(T, M)$, if $|S_1(T, M)| = |S_2(T, M)|$ then the two solutions must also share the exact same number of swaps and deletions. This is because for any solvable instance $\langle T, M\rangle$, the number of deletions is invariant: it is always $|M| - |T|$, the difference in the string lengths of M and T.

As an alternate representation of a solution to an instance, we will also define an intuitive mapping of each symbol in T to a selected symbol in M, as follows:

The *rearrangement map* ξ for solution $S(T, M)$ is the one-to-one function $\xi : \{1, \ldots, n\} \to \{1, \ldots, m\}$ that, for instance $\langle T, M\rangle$ and given index i, $1 \leq i \leq n$, yields the index in M to which symbol T_i is mapped in a solution $S(T, M)$ to the instance. Thus,

$$M_{\xi(1)} M_{\xi(2)} \ldots M_{\xi(n)} \;=\; T_1 T_2 \ldots T_n.$$

We also make use of the partial function ξ^{-1} denoting the partial inverse of the rearrangement map. If $\xi(a) = b$, then $\xi^{-1}(b) = a$ and $\xi(\xi^{-1}(a)) = \xi^{-1}(\xi(a)) = a$.

The rearrangement map allows us to easily visualize solutions by drawing lines between symbol T_i in T and $M_{\xi(i)}$ in M. Such a visualization is helpful for understanding the solution from a more intuitive standpoint, akin to a real-world problem such as rearranging tiles or dominos.

For example, consider the following instance:

Let T = '`RICE`' and M = '`CORRECTION`'. Here, one ξ function yields $\xi(1) = 4$, $\xi(2) = 8$, $\xi(3) = 6$, and $\xi(4) = 5$. Similarly $\xi^{-1}(4) = 1$, $\xi^{-1}(5) = 4$, $\xi^{-1}(6) = 3$, $\xi^{-1}(8) = 2$. The inverse function ξ^{-1} is undefined for other values. See Figure 2.1.

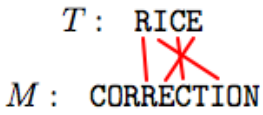

Figure 2.1: Visualization of a rearrangement map.

2.1.2 Equality of String-Reversed Instance

We observe here that for some instance $\langle T, M \rangle$, if *both* strings T and M are reversed, the new instance will have the same qualities as the original instance; that is, the exact same number of swaps and deletions will be required to transform M into T. This property certainly does not hold if only one of T and M is reversed. See Figure 2.2 for an example of a string-reversed instance.

Observation 1 (Equality of String-Reversed Instance of String-to-String Correction). *Let $\langle T, M \rangle$ and $\langle T^R, M^R \rangle$ be instances of S2SC, where T^R is the reverse of T and M^R is the reverse of M. Then $|S(T^R, M^R)| = |S(T, M)|$ and the number of different optimal solutions for $\langle T, M \rangle$ and $\langle T^R, M^R \rangle$ is also identical.*

T: `'COOLEX'`	T^R: `'XELOOC'`
M: `'EXPEALIDOCIOUS'`	M^R: `'SUOICODILAEPXE'`

Figure 2.2: The two instances $\langle T, M \rangle$ and $\langle T^R, M^R \rangle$ have the same number of operations in their respective optimal solutions.

2.1.3 Multiple Operation Types on the Same Symbol

With the following lemma we show that it is never optimal to swap a symbol that will eventually be deleted.

Lemma S (Non-optimality of Swapping and Deleting the Same Symbol). *Let $\langle T, M\rangle$ be an instance of S2SC, and let $S = S(T, M)$ be some solution to $\langle T, M\rangle$ in which some symbol M_k is involved in a swap (by either of $\sigma(k)$ or $\sigma(k-1)$) and then later deleted. Then S is a non-optimal solution to $\langle T, M\rangle$.*

Proof. By contradiction. Let M_k in M be a symbol that is to be swapped, and then deleted by some optimal solution $S(T, M)$. Suppose without loss of generality that we are to swap symbol $M_k =$ `'a'` with symbol $M_{k+1} =$ `'b'` by $\sigma(k)$. Our initial condition is $M =$ `'...ab...'`. After swap $\sigma(k)$, symbol M_k is now at position $k+1$; and after deleting it by $\delta(k+1)$, we have $M =$ `'...b...'`. The swap and deletion incur cost of $1 + 1 = 2$; but we could achieve the same state $M =$ `'...b...'` by simply deleting M_k and not performing the swap, at a cost of 1. Therefore S is not a minimum-cost solution, contradicting our assumption. □

A very simple example illustrates: suppose we want to transform $M =$`'bxa'` into $T =$`'ab'` under the operations of deletion and swap. The only way to achieve an optimal solution of one deletion and one swap (and hence with a cost of 2) is to delete the `'x'` symbol first, *before* any swaps. Otherwise our solution involves one deletion and two swaps (at a cost of 3, and clearly not optimal). We show later the interesting result that for binary alphabets, the order of operations is irrelevant (Theorem 4 in Section 3.15).

2.1.4 Swapping Identical Symbols

We establish a prohibition on the swap of identical symbols, since a cost will be incurred for the swap operation, but such a swap will not change M in any way.

Observation 2 (Non-optimality of Swapping Identical Symbols). *Let $\langle T, M\rangle$ be an instance of S2SC, and let $S = S(T, M)$ be a solution to $\langle T, M\rangle$. If S contains a swap $\sigma(k)$ for which two identical consecutive symbols $M_k = M_{k+1}$ are exchanged, then S is a non-optimal solution.*

2.2 Binary String-to-String Correction

Binary String-to-String Correction, or BSSC, is exactly the String-to-String Correction problem under the operations of deletion and swap as in Section 2.1, but utilizing strictly a *binary alphabet*; that is, an alphabet of size exactly two. For example: $\Sigma_1 = \{$'α', 'β'$\}$, $\Sigma_2 = \{$`'e', '+'`$\}$ and $\Sigma_3 = \{$`'Y', 'N'`$\}$ are all binary alphabets.

For this thesis, the only binary alphabet we will use is $\Sigma = \{$`'0', '1'`$\}$; note that no numerical values should be inferred from these symbols.

2.2.1 Terms

Some terms useful for working with the BSSC problem will be introduced here. Note that a full glossary of terms is presented for reference as Appendix B of this thesis.

An *instance* of BSSC is two binary strings T and M, and denoted $\langle T, M \rangle$ as with the general problem. For example: $\langle$ `'01110', '0011010111'` $\rangle$ is an instance of BSSC.

Denote the number of `'0'` symbols in a binary string X by $\#_0(X)$, and the number of `'1'` symbols by $\#_1(X)$.

A *solvable* instance of BSSC is one in which $\#_0(T) \leq \#_0(M)$ and $\#_1(T) \leq \#_1(M)$; that is, a sufficient number of each symbol exists in M such that M may be transformed into an exact copy of T using only the operations of deletion and swap.

Results from this thesis (and other works [1]) show that we may safely set aside some solvable instances from serious scrutiny, since they have been found to have straightforward polynomial-time solutions. We therefore define instances for which analysis is merited due to the presence of certain characteristics, as follows.

A *reduced instance* $\langle T, M \rangle$ of BSSC is an instance satisfying the following properties:

1. $\#_0(T) < \#_0(M)$;
2. $\#_1(T) < \#_1(M)$;
3. $T \not\subseteq M$; (T is not a subsequence of M)
4. $T_1 \neq M_1$; and
5. $T_{last} \neq M_{last}$, where T_{last} and M_{last} denote the trailing (rightmost) symbols in T and M respectively.

Later sections demonstrate why the presence of each of the listed properties in a reduced instance is desirable for analysis.

A *block* is a substring $X_r \dots X_s$ of a binary string X such that $X_i = X_r$ for $r \leq i \leq s$, and such that adjacent symbols satisfy $X_{r-1} \neq X_r$ or $X_{s+1} \neq X_r$ (note that X_{r-1} does not exist if $r = 1$, and X_{s+1} does not exist if $s = |X|$). That is, a block is a maximal run of contiguous identical symbols (see Figure 2.3).

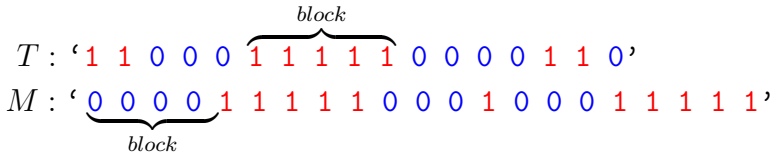

Figure 2.3: Example string pair $\langle T, M \rangle$ with blocks coloured.

We use B_i^X to denote the i^{th} block in the string X, counting from left to right. The last (rightmost) block is sometimes denoted B_{last}^X. Thus, using our previous example, we can relabel the diagram as in Figure 2.4:

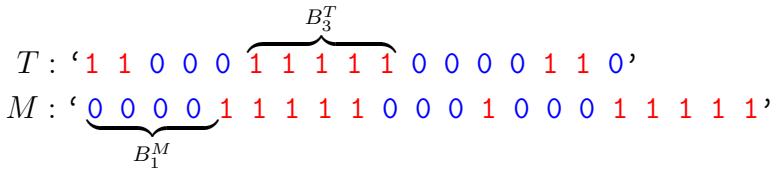

Figure 2.4: Example string pair $\langle T, M \rangle$ with some labeled blocks.

We use $\beta(X)$ to denote the number of blocks in binary string X. For example: $\beta(000110100) = 5$.

Some instances of S2SC are *single-type operation cases*. For BSSC, such cases are those in which either:

1. $\#_0(T) = \#_0(M)$ and $\#_1(T) = \#_1(M)$, in which case the problem can (and must) be solved with swap operations only; OR
2. $T \subseteq M$; i.e. T is a subsequence of M, in which case the problem can (and must) be solved with delete operations only.

Wagner and Fischer (1974) showed that single-type operation cases are solvable in polynomial time; in Section 2.3 we illustrate these findings with polynomial-time algorithms for each of the two single-type operation cases.

2.2.2 Equality of Symbol-Exchanged Instance

Suppose that for some instance $\langle T, M \rangle$, all symbols in both T and M are replaced by their counterpart in the binary alphabet Σ (that is, each symbol `‘0’` is replaced by symbol `‘1’` and vice versa). Then, the following observation tells us that the new instance has the same qualities as the original instance; that is, the exact same number of swaps and deletions are required to transform M into T.

Observation 3 (Equality of Symbol-Exchanged Instance of Binary String-to-String Correction). *Let $\langle T, M\rangle$ and $\langle T^X, M^X\rangle$ be instances of BSSC, where T^X and M^X are exact duplicates of T and M, but with symbol* `'0'` *replacing each* `'1'` *and symbol* `'1'` *replacing each* `'0'`*. Then $|S(T^X, M^X)| = |S(T, M)|$ and the number of optimal solutions is also identical.*

See Figure 2.5 for an example. This property does not hold in general if any fewer than *all* symbols in both T and M are exchanged with their counterparts in Σ.

T: `'001001'`	T^X: `'110110'`
M: `'1001010100'`	M^X: `'0110101011'`

Figure 2.5: The two instances $\langle T, M\rangle$ and $\langle T^X, M^X\rangle$ have the same number of operations in their respective optimal solutions.

The symbol-exchange property is valuable for implementation testing. Suppose we wish to test an experimental algorithm on every possible instance of BSSC in which $|T| = 4$ and $|M| = 6$. Then, we need only consider as possibilities for M the list of strings beginning with symbol $M_1 =$ `'0'`; i.e. `'000000'`, `'000001'`, `'000010'`, ..., `'011110'`, `'011111'`. This is because by Observation 3, `'100000'` is equivalent to `'011111'`, `'100001'` is equivalent to `'011110'`, and so on– right up to `'111111'` being equivalent to `'000000'`. For example, the instance $\langle$`'0011','101110'`$\rangle$ is equivalent by Observation 3 to $\langle$`'1100','010001'`$\rangle$. Thus, the last half of the list of 2^6 possibilities for M (in our example) can be safely ignored, reducing computation time significantly. Note that the entire list of 2^4 possibilities for T must still be considered. We could alternatively (and equivalently) examine the first half of all possibilities for T while considering all possibilities for M, but we choose to halve the M possibilities because M (for interesting instances of BSSC) will have greater length than T and thus the computational savings will be greater.

Further reductions to the number of test cases required for experimental implementations are possible: Observation 1 (Equality of String-Reversed Instance of S2SC) allows us to halve the number of cases yet again. This observation tells us that every instance of S2SC (for which BSSC is a member) has a reversed-sense mirror instance, which can be safely ignored during testing. Thus, using Observations 1 and 3 in concert we may reduce our original list

of all possible instances for given string sizes down to one quarter of its length.

2.3 History and Previous Work

This section provides information on previous work in the area, and presents several algorithms based on previous work which will be of use in this thesis.

2.3.1 History

The first serious treatment of String-to-String Correction (S2SC) with regard to finding an optimal solution was by Wagner and Fischer in 1974 [17]. One year later, Wagner [16] presented his CELLAR algorithm, a dynamic programming method that efficiently finds optimal solutions to S2SC under most sets of operations, with some notable exceptions (see Table 1.1). Bergroth *et al.* [2] also worked on the problem and found efficient algorithms for solving S2SC under certain operation sets.

More recently, Abu-Khzam *et al.* [1] proved that the decision version of String-to-String Correction is fixed-parameter tractable and provide a fixed-parameter algorithm that solves the problem in $O(1.6181^k n)$ time, where n is the length of the shorter of the two strings and k, the parameter, is the number of operations permitted.

2.3.2 Swaps-only Case

In the case where $\#_0(T) = \#_0(M)$ and $\#_1(T) = \#_1(M)$, we present the following new algorithm that transforms M into T in time $O(m^2)$ in the worst case, where $m = |T| = |M|$. Algorithm `BinaryStringSwapCorrect` takes as input two equal-length binary strings T and M and returns the least number of swaps required to transform M into T.

The swap counter *numSwaps* and the match counter ρ are both initialized to 0 at the commencement of the algorithm. The symbol ρ indicates the number of leading elements in

T: ‘0110111101110000011’
M: ‘0111100111010001110’

Figure 2.6: Example instance $\langle T, M \rangle$.

T and M which match. The strings are scanned from left to right and each leading match increments the ρ counter; when the first mismatch occurs, a swap of the first two dissimilar elements in M occurs, beginning at index ρ. For instance, if M_j is the first symbol that does not match its corresponding symbol in T (i.e. $T_j \neq M_j$), then M is scanned from M_j to M_m, and the first occurrence of dissimilar contiguous symbols M_x, M_{x+1} in M with $j \leq x < m$ is where the swap will occur.

For example: $\langle T, M \rangle$ in Figure 2.6 above we obtain $\rho = 3$ leading matches; that is, the first mismatch occurs at index 4. The first contiguous dissimilar symbols in M after index ρ are $M_5 =$‘1’ and $M_6 =$‘0’; thus this is where a swap will occur using Algorithm `BinaryStringSwapCorrect`.

After the swap has occurred, the swap counter is incremented and a recount of the leading matches occurs, beginning at index ρ and continuing until the next mismatch occurs. For every new match found, ρ is incremented. A swap is initiated as before, and the process continues until $\rho = m$ at which point T and M are identical.

The algorithm is correct because for a mismatch found at index $\rho + 1$ with all symbols $M_1 \dots M_\rho$ in M matching their corresponding symbols $T_1 \dots T_\rho$ in T, the matching symbol for $T_{\rho+1} = \alpha$ is the first occurrence M_k of α in M to the right of $M_{\rho+1}$; and the only way to align this symbol into position $M_{\rho+1}$ is to swap it with the non matching symbols $M_{\rho+1} \dots M_{k-1}$ in M. Thus, due to the unique nature of BSSC (i.e. the fact that it uses a binary alphabet) it is sufficient to find the first occurrence of non matching symbols in M to the right of M_ρ, since this swap will necessarily involve a symbol α and will bring that symbol closer to index $\rho + 1$ (i.e. leftward) where it will match with $T_{\rho+1}$.

We note that a symmetrical algorithm beginning at the end (rightmost symbol) of the equal-length strings T, M and working right to left would work equally well.

The running time of the algorithm is $O(m^2)$ in the worst case since there could be up to m mismatches, and for each mismatch $T_i \neq M_i$ it is possible that the entire string M (with length $|M| = m$) may need to be scanned to find a matching symbol. This running time corresponds with the upper bound on the number of swap operations, which we establish in Section 3.7.

```
01)   global binary string T, M
02)   global integer ρ          // number of leading elements in T and M
03)                             // that match in position and value
04)
05)   Algorithm UpdateMatchCount
06)     while T_ρ = M_ρ and ρ < |T|
07)       ρ ← ρ + 1
08)
09)   Algorithm SwapNext
10)     local integer x ← ρ
11)     while M_x = M_{x+1}
12)       x ← x + 1
13)     σ(M, x)                 // swaps positions of elements M_x and M_{x+1}
14)                             // in the binary string M
15)
16)   Algorithm BinaryStringSwapCorrect(targetString, mutableString)
17)     T ← targetString
18)     M ← mutableString
19)     ρ ← 1
20)     local integer numSwaps ← 0
21)     UpdateMatchCount
22)     while ρ < |T|
23)       SwapNext
24)       UpdateMatchCount
25)       numSwaps ← numSwaps + 1
26)     return numSwaps
```

Example of usage:
minSwapsRequired = `BinaryStringSwapCorrect(1011010, 0011101)`

2.3.3 Deletes-Only Case

In the case where $T \subseteq M$ (i.e. T is a subsequence of M), a simple linear-time algorithm for deleting extraneous symbols from M is provided here. For each symbol $T_1 \dots T_n$ in T, a matching symbol in M is located and marked as "saved". Once all n symbols in T have been located in sequence in M, every symbol in M not marked "saved" is deleted.

This algorithm has running time $O(m)$ where $m = |M| \geq n$, since it scans both strings T and M once to determine the symbols to save in M, and then scans M once more, deleting each unsaved symbol.

```
01)   Algorithm SolveWithSubsequence(T, M)
02)     local integer i ← 1
03)     local integer j ← 1
04)     do
05)       if  i ≤ |T|  and  T_i = M_j
06)         mark symbol M_j saved
07)         i ← i + 1
08)       j ← j + 1
09)     while  j ≤ |M|
10)     delete all symbols in M not marked saved
```

2.3.4 Leading Matches

The following lemma is given by Abu-Khzam *et al.* in their 2011 paper *Charge and reduce: A fixed-parameter algorithm for String-to-String Correction* [1] as Corollary 1, a consequence of Proposition 1. We provide a full proof here.

The lemma confirms the intuitive result that for an instance of S2SC for which the first n symbols in T and M are identical, these n symbols may be removed from the strings without changing the problem. For example, if $T =$`'BBBXYX'` and $M =$`'BBBYXYZXY'`, $T' =$`'XYX'` and $M' =$`'YXYZXY'`, instances $\langle T, M\rangle$ and $\langle T', M'\rangle$ are identical in the sense that the exact same swaps and deletions performed on $\langle T, M\rangle$ to transform T into M will also be performed on $\langle T', M'\rangle$ to transform T' into M'.

Lemma L (Leading Matches). *Let $\langle T, M\rangle$ be some instance with $n = |T|$, $m = |M|$, and such that the first k symbols in both T and M are identical (that is, $T_1 = M_1, T_2 = M_2, \ldots T_k = M_k$). Let $T^\lambda = T_{k+1}T_{k+2}\ldots T_n$ and $M^\lambda = M_{k+1}M_{k+2}\ldots M_m$. Then, $\Psi(T^\lambda, M^\lambda) = \Psi(T, M)$; that is, the number of operations in an optimal solution to $\langle T^\lambda, M^\lambda\rangle$ is identical to the number required in an optimal solution to $\langle T, M\rangle$.*

Proof. Begin with instance $\langle T, M\rangle$, with optimal solution cost $\Psi(T, M)$. Suppose we prepend a symbol $\alpha \in \Sigma$ to the beginning of both T and M, creating new strings $T' = \alpha T$ and $M' = \alpha M$. Then it is clear that for instance $\langle T', M'\rangle, \xi(1) = 1$ has cost 0, since $T'_1 = M'_1 = \alpha$ and each of these symbols is the leading symbol in their respective string. Since $|\xi(1) - 1| = 0$, $\Psi(T', M') \leq \Psi(T, M)$. As well, since the number of non-matching symbols in $\langle T', M'\rangle$ is unchanged from that of $\langle T, M\rangle$ (we have added only matching symbols), $\Psi(T, M) \leq \Psi(T', M')$. Therefore $\Psi(T, M) = \Psi(T', M')$.

For cases with multiple leading matched symbols: let $k > 1$ be the number of leading symbols in T and M which exactly match. Let $T^\lambda = T_{k+1}T_{k+2}\ldots T_n$ and $M^\lambda = M_{k+1}M_{k+2}\ldots M_m$. Start with instance $\langle T^\lambda, M^\lambda\rangle$ and accomplish this prepend operation k times, beginning with $\alpha_1 = T_k = M_k$ and ending with $\alpha_k = T_1 = M_1$, each time with cost 0. After the final addition of α_k, we have constructed the strings T and M from T^λ and M^λ respectively, at cost 0. Therefore, since we have shown that $\Psi(\alpha_i T^\lambda, \alpha_i M^\lambda) = \Psi(T^\lambda, M^\lambda)$ holds, after k iterations $\Psi(T, M) = \Psi(T^\lambda, M^\lambda)$. □

2.3.5 T_1 Maps to First Occurrence in M

In [1], Abu-Khzam *et al.* demonstrated that for any instance $\langle T, M \rangle$, regardless of alphabet size, T_1 can always be mapped to the first (leftmost) occurrence of its identical symbol in M. We provide a proof here within the context of BSSC.

Lemma A (Abu-Khzam *et al.*, 2011). *Let $\langle T, M \rangle$ be an instance of BSSC, with $T_1 = \alpha$. Let $M_a = \alpha$ be the first occurrence of symbol α in M. Then there exists an optimal solution $S(T, M)$ to $\langle T, M \rangle$ in which the associated rearrangement map ξ satisfies $\xi(1) = a$.*

Proof. By contradiction. Assume that mapping $T_1 = \alpha$ to M_a, the first (leftmost) occurrence of symbol α in M, is non-optimal. This means that in an optimal solution, T_1 is mapped to some symbol $M_k = \alpha$ which is to the right of M_a. Then symbol M_a is either deleted, or it is swapped to the right of M_k. If it is to be deleted, we can instead map T_1 to M_a and delete M_k, resulting in a solution with the same number of operations; and if M_a is to be swapped to the right of M_k, we can instead map T_1 to M_a and map M_k to some T_i with $i > 1$, avoiding the swap operation entirely. In either case we have obtained a solution which is at least as efficient as an optimal mapping, contradicting our assumption. Therefore mapping T_1 to M_a results in a solution that is at least as efficient as an optimal solution with some other mapping. □

2.4 Longest Common Subsequence

Longest Common Subsequence, or LCS, is a well-known concept in Computer Science that is referred to frequently in this thesis. Therefore, due to its utility in this work we provide a definition here.

A *subsequence* of a string X is a string that is a subset of the symbols in X, appearing in the same order as they appear in X. For instance, a subsequence of $X =$ `'EXAMPLE'` is $X' =$ `'APE'`.

A subsequence of a string X may contain noncontiguous symbols from X; this differentiates it from a *substring* of X, which is always a contiguous subset of X. Thus for $X =$

`'EXAMPLE'`, `'AMP'` is both a substring and a subsequence, while `'APE'` is a subsequence but not a substring.

Let $X', X'', \ldots X^{(n)}$ be n strings, with $n > 1$. A *common subsequence* of $X', X'', \ldots X^{(n)}$ is a string which is a subsequence of each $X^{(i)}$ in $X', X'', \ldots X^{(n)}$. Finally, a *longest common subsequence* of $X', X'', \ldots X^{(n)}$ is a common subsequence of maximum length. It is important to note that an LCS of a set of strings is not necessarily unique.

The problem of finding an LCS under n inputs is NP-hard in general [12]; however, in the case where just two strings X and Y are being examined, an LCS can be found in time $O(mn)$, where m, n are the lengths of X and Y [2].

Chapter 3

Results

In this chapter we present several results regarding the Binary String-to-String Correction Problem. The order in which these findings are presented is approximately based on the complexity of the result, ranging from the relatively simple (the effect of deletion and swap operations on the number of blocks in a string) to the more complicated (showing that the order of operations in a solution to an instance of BSSC is irrelevant). There is also occasionally a forced presentation order, since some results depend on other results that must be presented first.

3.1 Effect of Deletion on the Number of Blocks

We show that deletion of a symbol X_i from string X can leave the number of blocks in X unchanged, or it can reduce the number of blocks in X by 1 or 2. That is, deletion never increases the number of blocks in a string.

Lemma 1 (Effect of Deletion on Block Size). *Let X be a binary string. Then any deletion $\delta(X, k)$ decreases $\beta(X)$ by at most 2.*

Proof. By examining all possible cases. Consider a deletion of symbol X_d from string X. Let B_k^X be the block in X to which symbol X_d belongs. If $|B_k^X| > 1$ prior to the deletion, then the number of blocks in X is unchanged after the deletion; otherwise, X_d comprises the

entire block, in which case the number of blocks in X is reduced by 1 if B_k^X is the first or last block in X, and by 2 otherwise. □

Some examples: no single deletion from string $X = \texttt{110011}$ changes the number of blocks in X. Alternatively, if $X = \texttt{101}$, $\delta(X, 1)$ and $\delta(X, 3)$ both decrease the number of blocks in X by 1, while $\delta(X, 2)$ decreases the number of blocks by 2.

3.2 Effect of Swap on the Number of Blocks

The following lemma shows that a swap can both increase or decrease the number of blocks in a string, or leave the number of blocks unchanged.

Lemma 2 (Effect of Swap on Block Size). *Let X be a binary string. Then any swap $\sigma(X, k)$ increases or decreases $\beta(X)$ by at most 2.*

Proof. By examining all possible cases. Table 3.1 below includes every possibility with regard to symbols neighbouring a swap, and demonstrates that after a swap $\sigma(X, i)$ in string X (creating new string X'), the difference in block size between strings X and X' is in the range $\{-2 \ldots 2\}$.

X	X'	Δ # Blocks
$\ldots 001$	$\ldots 010$	+1
$\ldots 101$	$\ldots 110$	-1
$\ldots 0010 \ldots$	$\ldots 0100 \ldots$	0
$\ldots 0011 \ldots$	$\ldots 0101 \ldots$	+2
$\ldots 1010 \ldots$	$\ldots 1100 \ldots$	-2
$\ldots 1011 \ldots$	$\ldots 1101 \ldots$	0

Table 3.1: Effect of SWAP on the number of blocks.

Swaps involving leading blocks have the same effect as swaps involving trailing blocks and thus are not shown in the table. □

3.3 Counting the Number of Swaps in a Solution

Our method for determining the number of swaps in a solution is as follows: perform all deletions first, obtaining an equal-length instance; next, measure the distance between each of the mapped symbols in some mapping ξ; and finally sum these distances. Since two mappings are involved in every swap, and both symbols from Σ are necessarily a member of every swap, we restrict our distance measurements to all those mappings involving a single symbol (arbitrarily `'1'`) to avoid double counting.

Notationally: if P is a proposition, then the value of $[\![P]\!]$ is 1 if P is true and is 0 if P is false.

Observation 4 (Counting Swaps in a Solution). *Let $\langle T, M\rangle$ be an instance of BSSC, let $S(T, M)$ be a solution, and let $\langle T, M'\rangle$ be the instance that results after all deletions in S have been completed. Then the number s of swaps in S required to transform M' into T is obtained as follows:*

$$s \;=\; \sum_{i=1}^{n} (|i - \xi(i)| \cdot [\![T_i = \text{`1'}]\!])$$

Proof. The number of swaps is the sum of the distances between i and $\xi(i)$ because after each swap $\sigma(i)$ from solution $S(T, M)$, symbol M_i is moved exactly one index closer to its desired position $\xi^{-1}(i)$; that is, if $|i - \xi(k)| = d$, then exactly d swaps will be required to reposition symbol $M_{\xi(k)}$ to position k in M.[1] □

[1] This counting method is used by program `OptimalSolver` to determine the minimum number of swaps required given all possible deletion sets for an instance, and thus find an optimal solution. More information is in Appendix A.

3.4 The Non-crossing Lemma

We show that any optimal solution never maps a symbol $T_i = \alpha$ in T to $M_j = \alpha$ in M if there exists an M_k in M that would require fewer swaps to move into position M_i. That is, we must always choose minimum-cost mappings.

Lemma 3 (Non-crossing Lemma). *Let ξ be the rearrangement map associated with an optimal solution $S(T, M)$. If $i < j$ and $T_i = T_j$, then $\xi(i) < \xi(j)$.*

Proof. For $i < j$ and $k < l$ (without loss of generality), suppose an optimal solution S satisfies $\xi(i) = l$ and also $\xi(j) = k$. Then $M_k = M_l$ with $k < l$. But because of our assumption, S contains $M^{(s)}, M^{(t)}, s < t$ such that $M^{(s)} = M_1 \dots M_k M_l \dots M_m$, $M^{(t)} = M_1 \dots M_l M_k \dots M_m$. Thus a swap of identical symbols occurs, contradicting the assumed optimality of S by Observation 2. □

Figure 3.1 illustrates the proof with an example non-optimal mapping.

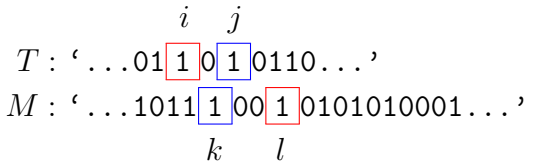

Figure 3.1: Crossed rearrangement map. The red-to-red and blue-to-blue assignments cannot occur in an optimal solution to $\langle T, M \rangle$.

3.5 Swap Adjacency

Lemma 3, the Non-Crossing Lemma, implies that the rearrangement map for an optimal solution must always contain minimum-cost mappings; we next show the related result that two symbols M_j, M_l in M that are to be swapped in an optimal solution need never have any symbol M_k between them (that is, $j < k < l$) that is to be deleted. In doing this, we will show that for any mapping in which two symbols in M which are to be swapped with each

other but have symbols between them, an alternate solution exists for which the symbols in M which are to be swapped are adjacent.

This interesting characteristic is due to the fact that BSSC utilizes a binary alphabet; thus, symbol M_k can always be mapped to be involved in the required swap, and the deletion can instead be performed on M_j or M_l, since either $M_k = M_j$ or $M_k = M_l$.

Theorem 1 (Swap Adjacency). *Let ξ be the rearrangement map for an optimal solution $S(T, M)$ to instance $\langle T, M\rangle$. If $T_i \neq T_{i+1}$, $\xi(i+1) < \xi(i)$ and each of the intermediate symbols $M_{\xi(i+1)+1} \ldots M_{\xi(i)-1}$ are to be deleted, then there exists an optimal solution with rearrangement map ξ' such that $\xi'(i+1) = \xi'(i) - 1$.*

Proof. Without loss of generality let $T_i =$`'0'` and $T_{i+1} =$`'1'`, which implies $M_{\xi(i)} =$`'0'` and $M_{\xi(i+1)} =$`'1'`. Let $k > 0$ be the number of intermediate symbols between $M_{\xi(i+1)}$ and $M_{\xi(i)}$ that are to be deleted. First observe that among these k intermediate symbols, the sequence `'01'` cannot occur, since this would imply the existence of a solution with fewer operations (a swap would not be required) and S would not be optimal.

Pick an intermediate symbol M_d such that $\xi(i+1) < d < \xi(i)$. Since M_d is to be deleted, we observe that if $M_d =$`'0'`, we can instead map T_i to M_d and delete symbol $M_{\xi(i)}$, without changing the number of operations (there will still be k deletions and one swap). Similarly if $M_d =$`'1'`, we can instead map T_{i+1} to M_d and delete symbol $M_{\xi(i+1)}$ without changing the number of operations.

Repeat this process for all of the k intermediate symbols between $M_{\xi(i+1)}$ and $M_{\xi(i)}$ until the assignments for T_i and T_{i+1} are adjacent in M; call this revised rearrangement map ξ' with the property that $\xi'(i+1) = \xi'(i) - 1$. □

Figure 3.2 below illustrates the concept of swap adjacency:

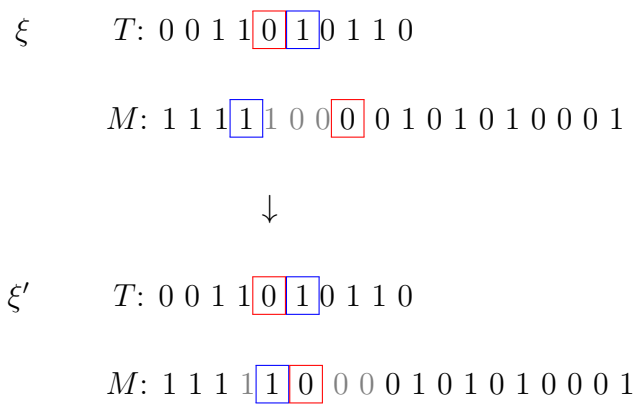

Figure 3.2: Swap adjacency. Symbols in M that are to be deleted (gray) are never required to lie between two symbols in M that are to be swapped.

3.6 Unidirectional Movement of Symbols

This section shows another unique characteristic of BSSC: any symbol that is repositioned by a swap will only ever move in one direction. It will never return to its original index position (unless shifted there by a deletion) nor will it ever be swapped in the opposite direction (assuming that all swap operations are completed as part of an optimal solution).

The theorem is stated for the swaps-only case of BSSC; this allows us to ignore the incidental movement of a symbol caused by a wholesale index shift within the string M after a deletion operation.

Theorem 2 (Unidirectional Movement of Symbols). *Let $\langle T, M\rangle$ be an instance of BSSC for which $|T| = |M|$, and let M_k be the symbol at position k in M. If M_k is swapped with an adjacent symbol at least once in an optimal solution, then symbol M_k will never return to position k in some successor M' of M.*

Proof. We first demonstrate that any required swap in an optimal solution is always completely productive in the sense that *both* symbols involved in the swap are moved closer to

their intended position in the rearrangement map ξ. Let $\langle T, M \rangle$ be an instance of BSSC with $m = n$, let M_i and M_{i+1} be an adjacent pair of symbols in the mutable string M with $1 \leq i < m$, and finally let $S(T, M)$ be an optimal solution with rearrangement map ξ. See Figure 3.3 for an illustrative example.

i	1	2	3	4	5	6	7	8	9
$\xi(i)$	2	3	1	4	6	5	7	9	8
T :	‘0	0	1	1	1	0	1	0	1’
M :	‘1	0	0	1	0	1	1	1	0’
$\xi^{-1}(i)$	3	1	2	4	6	5	7	9	8

Figure 3.3: Optimal solution $S(T, M)$ with $n = m = 9$.

First, if $\xi^{-1}(i) < \xi^{-1}(i+1)$, the symbols must not be swapped since M_i precedes M_{i+1} in ξ and thus their order relative to each other is already correct, by the alternate definition of a solution using the partial function ξ: consider that the symbol $T_{\xi^{-1}(i)}$ lies to the left of symbol $T_{\xi^{-1}(i+1)}$ and ξ is optimal by assumption, so if symbols M_i and M_{i+1} were to be swapped, they would eventually have to be swapped with each other again to match the order of their mapped symbols in T, resulting in a repeated swap of the same two symbols and therefore a non-optimal solution.

If $\xi^{-1}(i) > \xi^{-1}(i+1)$, then $M_i \neq M_{i+1}$ (otherwise a swap of identical symbols occurs and S is not optimal). Furthermore, $\xi^{-1}(i) > \xi^{-1}(i+1)$ implies that M_i and M_{i+1} must be swapped since M_{i+1} precedes M_i in ξ. Suppose $\xi^{-1}(i) > \xi^{-1}(i+1)$ and thus $\sigma(i)$ is performed. This exchanges the positions of M_i and M_{i+1}, and their position relative to each other in ξ is achieved: M_{i+1} now precedes M_i, which corresponds to their mapped symbols in T in which symbol $T_{\xi^{-1}(i+1)}$ precedes symbol $T_{\xi^{-1}(i)}$. Thus, a required swap will always improve the position of *both* symbols involved in the swap.

During the execution of an optimal solution, consider symbol M_k that is swapped with symbol M_{k+1}, so that it now lies at position $k+1$ in string M. To return symbol M_k to position k, it must therefore be swapped in the opposite direction. But since we have shown that any swap required in an optimal solution is *productive* in the sense that both symbols

involved are moved closer to their corresponding mapped positions in T, the second (opposite direction) swap of M_k will reverse the productive effect of the first swap. That is, it will take M_k further away from its intended position, adding two unnecessary swaps to the solution and rendering the solution non-optimal.

Therefore, if M_k is swapped with an adjacent symbol at least once in an optimal solution, then symbol M_k will never return to position k in M. □

3.7 Upper Bound on Operations

We next establish an upper bound on the number of swaps and/or deletions required for a given instance of BSSC.

Theorem 3 (Upper Bound on the Number of Operations). *Let $\langle T, M\rangle$ be a solvable instance of BSSC with $|T| = n$ and $|M| = m$. Then an upper bound on $|S(T, M)|$, the number of operations required in any solution to the instance, is $(\#_0(T))(\#_1(T)) + m - n$.*

Proof. For deletions, it is clear by inspection that the number of deletions required to reduce the string length of M to the length of T is invariant; it is exactly $m - n$. Thus any solution $S(T, M)$ will always involve exactly $m - n$ deletion operations.

To establish the upper bound on swaps, we will construct a worst-case example. Assume $T \not\subseteq M$ (T is not a subsequence of M), otherwise no swaps are required. Also assume for simplicity that all required deletions (if any) have been performed, leaving $n = m$. This is permissible since we have shown that no deletion need ever be performed between two symbols which are to be swapped (by Theorem 1, the Swap Adjacency Theorem).

Given a solution S with rearrangement map ξ, consider assignment $\xi(i) = j$ with $i \neq j$. Thus symbol $M_j \in M$ is matched to symbol $T_i \in T$. Since by definition the swap operation is performed on adjacent symbols only, we must perform $|i - j|$ swaps in order to match T_i with M_j. Then the worst-case scenario for this single mismatched symbol is $\xi(1) = n$

(without loss of generality), requiring $n-1$ swaps to match the single symbol T_1 with M_n.

Now consider not one, but two assignments in the rearrangement map ξ. Here, the worst-case scenario (placing the symbols to be matched as far away from each other as possible) is $\xi(1) = n-1$ and $\xi(2) = n$. Note that $\xi(1) = n$ and $\xi(2) = n-1$ is impossible by Lemma 3, the Non-crossing Lemma; and any different rearrangement map ξ' containing $\xi'(i) = j$ with $i > 2$ or $j < n-1$ will have a smaller number of swaps required than those in ξ, since their symbols will be closer to each other and thus the value $|i-j|$ will not be maximized. Thus for $\xi(1) = n-1$ and $\xi(2) = n$, the number of swaps required for performing the matches $\xi(i)$ and $\xi(j)$ will be $((n-1)-1)+(n-2) = 2(n-2)$.

Continuing this process, place $\lfloor \frac{n}{2} \rfloor$ `'0'` symbols as far apart as possible. This gives $\xi(1) = n - \lfloor \frac{n}{2} \rfloor$, $\xi(2) = n - \lfloor \frac{n}{2} \rfloor + 1$, $\ldots$ $\xi(\lfloor \frac{n}{2} \rfloor) = n$, at a cost of $\frac{n}{2}$ swaps each.

The worst-case scenario occurs exactly when $\#_0(T) = \#_0(M) = \lfloor \frac{n}{2} \rfloor$, and $\#_1(T) = \#_1(M) = \lceil \frac{n}{2} \rceil$ (without loss of generality). Consider that in a string of length n, $\#_0(M)$ of which are `'0'`s and $\#_1(M)$ of which are `'1'`s, the $\#_0(M)$ zeroes must be moved a distance of $\#_1(M)$ each, for a cost of $C = (\#_0(M))(\#_1(M))$. This cost function is analogous to maximizing the area of a rectangle given a perimeter of length n, and is maximal when the rectangle is a square; that is, $\#_0(M) = \#_1(M)$ (this implies $\#_0(T) = \#_1(T)$).

Therefore, the maximum number of operations required in the solution to some instance $\langle T, M \rangle$ is the sum of the maximum number of deletions and the maximum number of swaps, that is, $(m-n)+(\#_0(T))(\#_1(T))$; and this number is maximized when $\#_0(T) = \#_1(T)$. $\square$

3.8 Trailing Matches

We note that the following lemma was also developed independently and concurrently by Nathaniel Watt in his M.Sc. thesis [18] as Reduction Rule 3.2.6.

Lemma 4 (Trailing Matches). *Let $\langle T, M \rangle$ be some instance with $n = |T|$, $m = |M|$, and*

such that the last k symbols in both T and M are identical (that is, $T_{n-k+1} = M_{m-k+1}$, $T_{n-k+2} = M_{m-k+2}, \ldots T_n = M_m$). Let $T' = T_1 T_2 \ldots T_{n-k}$ and $M' = M_1 M_2 \ldots M_{m-k}$. Then $\Psi(T', M') = \Psi(T, M)$; that is, the number of operations in an optimal solution to $\langle T', M' \rangle$ is identical to the number required in an optimal solution to $\langle T, M \rangle$.

Proof. By Observation 1, the problem $\langle T, M \rangle$ has exactly the same number of operations required as $\langle T^R, M^R \rangle$, the same problem with each string reversed. Therefore we can reverse the strings T, M and apply Lemma L, yielding $\Psi(T, M) = \Psi(T', M')$. □

3.9 T_n Maps to Last Occurrence in M

Extending Lemma A from Abu-Khzam *et al.*, we show a similar result for the trailing symbol in T.

Lemma 5 (Mapping of T_n). *Let $\langle T, M \rangle$ be an instance of BSSC, with $T_n = \alpha$. Let $M_a = \alpha$ be the last occurrence of symbol α in M. Then there exists an optimal solution $S(T, M)$ to $\langle T, M \rangle$ in which the associated rearrangement map ξ contains $\xi(n) = a$.*

Proof. By Observation 1, the problems $\langle T, M \rangle$ and $\langle T^R, M^R \rangle$ are equivalent. Therefore we can reverse the strings T, M and apply Lemma A, yielding an optimal match for T_1 in T^R; this is also an optimal match for T_n in T. □

3.10 B_1^T Maps to First Occurrences in M

We show that the entire first (leftmost) block of k contiguous symbols in T can always be mapped to the first k occurrences of that symbol in M in an optimal solution.

Lemma 6 (Mapping of B_1^T). *Let $\langle T, M \rangle$ be an instance of BSSC, with B_1^T of size k such that $T_1 = T_2 = \ldots T_k = \alpha$. Let $M_{a_1} = M_{a_2} = \ldots M_{a_k} = \alpha$ be the first k occurrences of symbol α in M. Then there exists an optimal solution $S(T, M)$ to $\langle T, M \rangle$ in which the associated rearrangement map ξ satisfies $\xi(1) = a_1, \xi(2) = a_2, \ldots \xi(k) = a_k$.*

Proof. Assume $|B_1^T| = k > 1$ (otherwise revert to Lemma A), and without loss of generality let $\alpha =$ `'1'`. We have already shown that T_1 can be mapped to the first occurrence of `'1'` in M. Consider T_2: it is now the leftmost unmapped `'1'` in T and by the same argument in Lemma A, it can be mapped to the second `'1'` in M (call it M_s). If, for example, we mapped it to some $M_w =$ `'1'` which was to the right of M_s, then we would have to either delete M_s or swap it across M_w, drawing a contradiction.

We can continue to apply this argument for any particular T_i which is one of the k `'1'` symbols in B_1^T, since all of symbols $T_1 \dots T_{i-1}$ have been previously mapped, and no benefit is gained by mapping the $i + 1^{\text{th}}$ or later occurrence of `'1'` in M. □

3.11 B_{last}^T Maps to Last Occurrences in M

We show that the entire last (rightmost) block of k contiguous symbols in T can always be mapped to the last k occurrences of that symbol in M in an optimal solution.

Lemma 7 (Mapping of B_{last}^T). *Let $\langle T, M\rangle$ be an instance of BSSC, with B_{last}^T of size k such that $T_{n-k} = T_{n-k+1} = \dots T_n = \alpha$. Let $M_{a_1} = M_{a_2} = \dots M_{a_k} = \alpha$ be the last k occurrences of symbol α in M. Then there exists an optimal solution $S(T, M)$ to $\langle T, M\rangle$ in which the associated rearrangement map ξ contains $\xi(n-k) = a_1, \xi(n-k+1) = a_2, \dots \xi(n) = a_k$.*

Proof. By Observation 1, the problems $\langle T, M\rangle$ and $\langle T^R, M^R\rangle$ are equivalent. Therefore we can reverse the strings T, M and apply Lemma 6, yielding an optimal match for the k leading symbols $T_1 \dots T_k$ in T^R; this is also an optimal match for the k trailing symbols $T_{n-k} \dots T_n$ in T. □

3.12 Matching Contiguous Symbols after B_1^T

Consider the following instance of BSSC:

T : '11101100010...'
M : '00100110111111101000011...'

The red symbols in T denote B_1^T, and the red symbols in M denote their mapping in some optimal solution, by Lemma 6. Note that the three blue symbols immediately after B_1^T match with the three symbols immediately after the last red symbol in M. In the following lemma we show that these matching contiguous symbols in M can always be mapped in an optimal solution.

Lemma 8 (Contiguous Matches after B_1^T). *Let $\langle T, M\rangle$ be an instance of BSSC, with B_1^T of size k such that $T_1 = T_2 = \ldots T_k = \alpha$. Let $M_{a_1} = M_{a_2} = \ldots M_{a_k} = \alpha$ be the first k occurrences of symbol α in M. If there exists substring $M^\lambda = M_{a_k+1} \ldots M_{a_k+r}$ in M with $r \geq 1$ and a substring $T^\lambda = T_{k+1} \ldots T_{k+r}$ in T such that $M^\lambda = T^\lambda$, then there exists an optimal solution to $\langle T, M\rangle$ in which each symbol in the substring M^λ has a mapping; that is, none of the symbols in M^λ are to be deleted.*

Proof. Without loss of generality, assume that M_d is the leftmost symbol in M^λ, and that $M_d = T_{k+1} =$ '0'. Thus $M_{d-1} = T_k$ and $\xi(k) = d - 1$ by Lemma 6. Let $S(T, M)$ be an optimal solution and let $\xi(k + 1) = e$, with $e \neq k + 1$ (otherwise symbols $M_1 \ldots M_e$ are leading matches and may be removed from the problem by Lemma L.

Note that T_{k+1} is the first occurrence of symbol '0' in T; and since $T_{k+1} =$'0', we require a '0' symbol immediately to the right of symbol M_{d-1} in M. If $d = e$ we are done. If $d \neq e$ and symbol M_d is unmapped, create a revised mapping ξ' containing $\xi'(k+1) = d$ and which deletes symbol M_e without increasing cost, since M_d is already the correct symbol (a '0') in the correct place (immediately to the right of symbol M_{d-1}). Therefore a mapping can be found for M_d in an optimal solution (since ξ' incurs the same cost as ξ) and a different symbol deleted.

After M_d is mapped in this way, remaining symbols in M^λ that are unmapped may be mapped in exactly the same way in the revised mapping ξ'. □

3.13 Matching Contiguous Symbols prior to B^T_{last}

Consider the following instance of BSSC:

T : '...10110100010000'
M : '...00101000110010100110111111011'

The red symbols in T denote B^T_{last}, and the red symbols in M denote their mapping in some optimal solution, by Lemma 7. Note that the two blue symbols immediately prior to (left of) B^T_{last} match with the two symbols immediately prior to the first red symbol in M. In the following lemma we show that these matching contiguous symbols in M can always be mapped in an optimal solution.

Lemma 9 (Contiguous Matches prior to B^T_{last}). *Let $\langle T, M\rangle$ be an instance of BSSC, with B^T_{last} of size k such that $T_{n-k} = T_{n-k+1} = \ldots T_n = \alpha$. Let $M_{a_1} = M_{a_2} = \ldots M_{a_k} = \alpha$ be the last k occurrences of symbol α in M. If there exists substring $M^\lambda = M_{a_1-r} \ldots M_{a_1-1}$ in M with $r \geq 1$ and a substring $T^\lambda = T_{n-k-r} \ldots T_{n-k-1}$ in T such that $M^\lambda = T^\lambda$, then there exists an optimal solution to $\langle T, M\rangle$ in which each symbol in the substring M^λ has a mapping.*

Proof. Let $\langle T, M\rangle$ be an instance of BSSC with nonempty strings T^λ, M^λ as defined above such that $T^\lambda = M^\lambda$. Reversed-string instances of BSSC are equivalent, by Observation 1; therefore we can create the equivalent instance $\langle T^R, M^R\rangle$ and apply Lemma 8 above. □

3.14 Mapping of Symbols from B^M_1 and B^M_{last}

Some instances of BSSC allow us to automatically assign a certain number of symbols from the first and last blocks of M to their first and last occurrences in T, respectively. Essentially, if the interior blocks of M do not contain enough symbols to match each symbol in T, the use of symbols from B^M_1 or B^M_{last} is automatic, and the exact mapping of symbols in T to these extremal symbols in M in any optimal solution is also known.

Lemma 10 (Mapping of Symbols from B_1^M and B_{last}^M). *Let $\langle T, M\rangle$ be a reduced instance of BSSC, and let M^I be an exact copy of M, but with blocks B_1^M and B_{last}^M omitted. If $\#_\alpha(M^I) < \#_\alpha(T)$, then at least $c = \#_\alpha(T) - \#_\alpha(M^I)$ α symbols from B_1^M or B_{last}^M will be mapped in any solution to $\langle T, M\rangle$. Further, from these c symbols, the a α symbols mapped from B_1^M will be mapped to the first a occurrences of α in T, and the b α symbols mapped from B_{last}^M will be mapped to the last b occurrences of α in T, in any optimal solution to $\langle T, M\rangle$.*

Proof. Since $\langle T, M\rangle$ is a reduced instance, it is solvable; therefore, since M^I contains insufficient symbols to match the entirety of T, some of the n symbols in T that will be matched to n symbols in M must be matched to symbols in B_1^M or B_{last}^M, by the pigeonhole principle.

To show that a `'0'` symbols (without loss of generality) from B_1^M must be matched to the first a `'0'` symbols in T, we use contradiction. Assume that in some optimal solution $S(T, M)$, one of these `'0'` symbols M_r is matched to symbol $T_e =$ `'0'` in T which is not one of the first a occurrences of `'0'` in T. This means that there exists a symbol $T_d =$`'0'` in T which is one of the first a `'0'` symbols in T, and is mapped to a symbol M_s not in B_1^M. So $\xi(e) = r$ and $\xi(d) = s$; but since $d < e$ and $r < s$ we incur a swap of identical symbols, contradicting the optimality of S by Observation 2.

An identical argument for the b mapped α symbols from B_{last}^M establishes the same forced mapping of these symbols to the last b occurrences of α in T in any optimal solution. □

3.15 Independence of Operations

We present the important result that for the BSSC problem, the swap and delete operations required for obtaining an optimal solution can be performed in any order.

Theorem 4 (Independence of Operations). *Let $\langle T, M\rangle$ be an instance of BSSC, and assume there is an optimal solution that requires k swaps and $m-n$ deletions to transform M into T. Then, for any permutation P of k swaps and $m-n$ deletions required to transform*

M into T, there exists a solution S in which the $k+m-n$ operations are in the exact order of P.

Proof. We first examine the deletions-only and swaps-only instances of the problem.

Deletions-only Instance

First, for $S(T, M)$ containing no swaps, proof is by inspection. The order of operations is irrelevant; exactly $m - n$ deletions must be performed, regardless of order. For example, if boxed symbols are to be deleted:

0 0 1 $\boxed{0}$ 0 0 $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{0}$ 1 1 $\boxed{0}$ 1 1 0 0 0 1 1 0

Deleting the boxed symbols in any order will always produce the same result:

0 0 1 0 0 1 1 1 1 0 0 0 1 1 0

Swaps-only Instance

Theorem 2 (Unidirectional Movement of Symbols) demonstrates that any swap required in an optimal solution is productive in the sense that *both* symbols involved in the swap are always moved closer to their ξ mapping. Therefore, since any swap σ that is required by the rearrangement map ξ for an optimal solution $S(T, M)$ always improves the positioning of *both* affected symbols and does not change the position of any other symbols, the swaps may be performed in any order, without increasing the number of operations required.

Mixed Swaps and Deletions Instance

We now examine cases $S(T, M)$ containing both deletions and swaps. Consider the rearrangement map ξ and some adjacent pair of symbols M_i and $M_{i+1} \in M$ with $1 \leq i < m$. See Figure 3.4 for an example.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\xi(i)$	3	4	6	9	11	10	12	15	13						
T :	‘0	0	1	1	1	0	1	0	1’						
M :	‘1	1	0	0	0	1	0	0	1	0	1	1	1	0	0’
$\xi^{-1}(i)$			1	2		3			4	6	5	7	9		8

Figure 3.4: Example instance $\langle T, M\rangle$ with $n = 9, m = 15$ and optimal solution $S(T, M)$ mapped by ξ.

If M_i and M_{i+1} are both mapped (there exist $\xi(x), \xi(y)$ such that $\xi(x) = i$ and $\xi(y) = i+1$), we will swap them if $x > y$ and otherwise perform no operations on them (as per the swaps-only argument above); in either case, no other symbols in M will be affected.

If M_i and M_{i+1} are both unmapped (there exists no $\xi(x)$ such that either $\xi(x) = i$ or $\xi(x) = i + 1$), one or both of these symbols may be deleted at any time, without affecting any other deletions (by the deletions-only case above) or affecting any swaps (by Theorem 1, the Swap Adjacency Theorem).

Now suppose without loss of generality that M_i is mapped but M_{i+1} is not. Here, three cases can occur.

(1) If $\xi^{-1}(i) = i$, then M_i is in its correct position (no action will be performed on it); delete M_{i+1} at any time without affecting other symbols, as in Figure 3.5 on the next page.

(2) If $\xi^{-1}(i) < i$, then M_i might later be swapped with some symbol to the left of M_i, but it will certainly not swapped with any symbol to the right of it since this would be non-optimal. Thus the swap $\sigma(i)$ is not an option, and deletion of M_{i+1} at any point in time (before or after the leftward swap of M_i) will not affect any operation on M_i. See Figure 3.6.

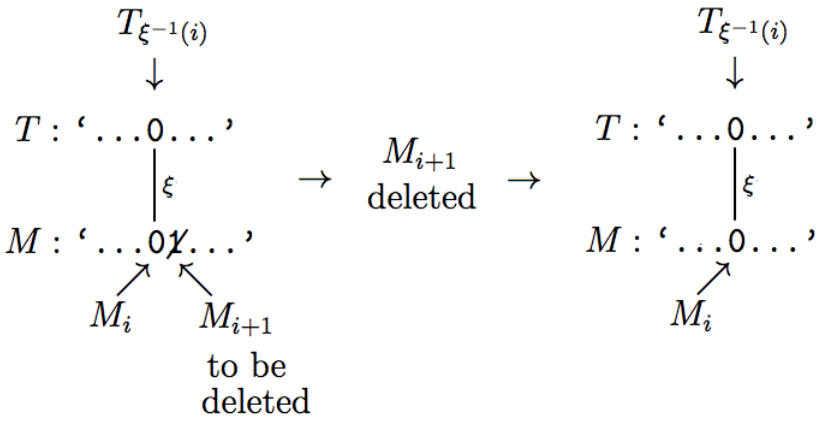

Figure 3.5: Depiction of case where $\xi^{-1}(i) = i$.

(3) If $\xi^{-1}(i) > i$, then we invoke the Swap Adjacency Theorem, as follows.

If $M_i = M_{i+1}$ we can reassign M_i to be deleted and M_{i+1} to be saved and swapped to the right, by the Swap Adjacency Theorem; or we can simply choose to delete the unmapped M_{i+1} prior to swapping M_i without affecting M_i in any way. See Figure 3.7.

If $M_i \neq M_{i+1}$, we can still choose either swap or deletion without affecting the solution. We first create a new mapping ξ' which is identical to ξ except that we reassign the deletion of M_{i+1} to the next identical symbol M_j to the right of M_{i+1}, by the Swap Adjacency Theorem. Now, if we choose to perform swap $\sigma(i)$ first, we will not affect the pending deletion of M_j in any way; and if we delete M_j prior to the swap at index i, this deletion will not affect the swap in any way See Figure 3.8.

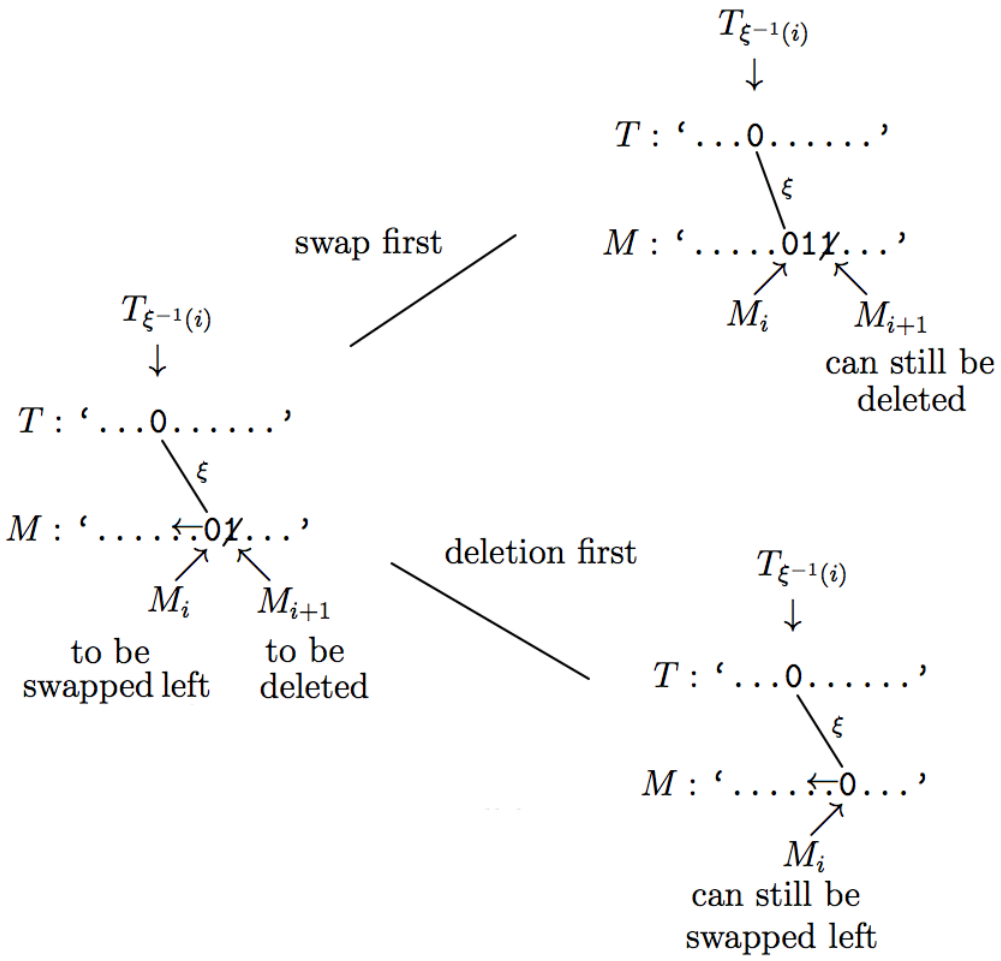

Figure 3.6: Depiction of case where $\xi^{-1}(i) < i$. The left figure is the initial condition; the two right figures indicate the condition after initiating a swap first (top) and alternately, initiating a deletion first (bottom).

Finally, note that for the opposite assumption in which M_i is unmapped but M_{i+1} is mapped, a mirrored argument of the three cases above establishes an identical result.

In all of these cases, the execution of any required swap (even if one of the swapped symbols is unmapped) is *productive* in the sense that it does not create the need for additional

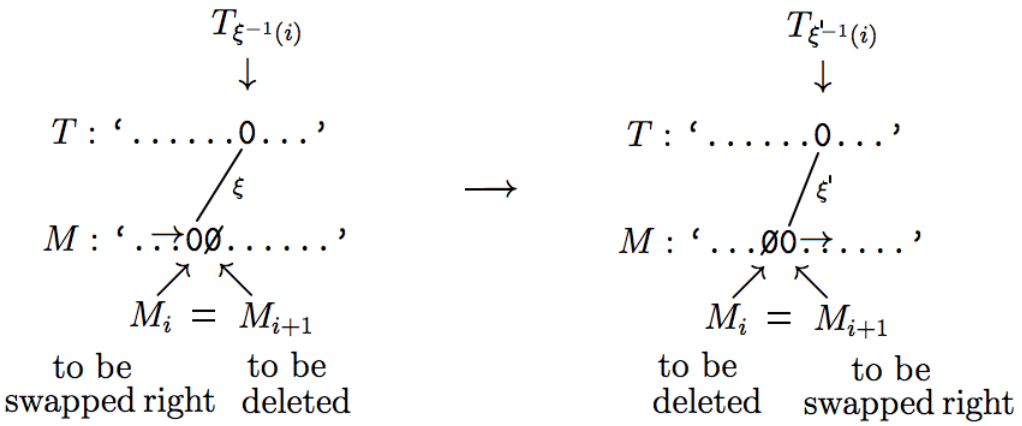

Figure 3.7: Depiction of case where $\xi^{-1}(i) > i$ with $M_i = M_{i+1}$, before and after invoking the Swap Adjacency Theorem.

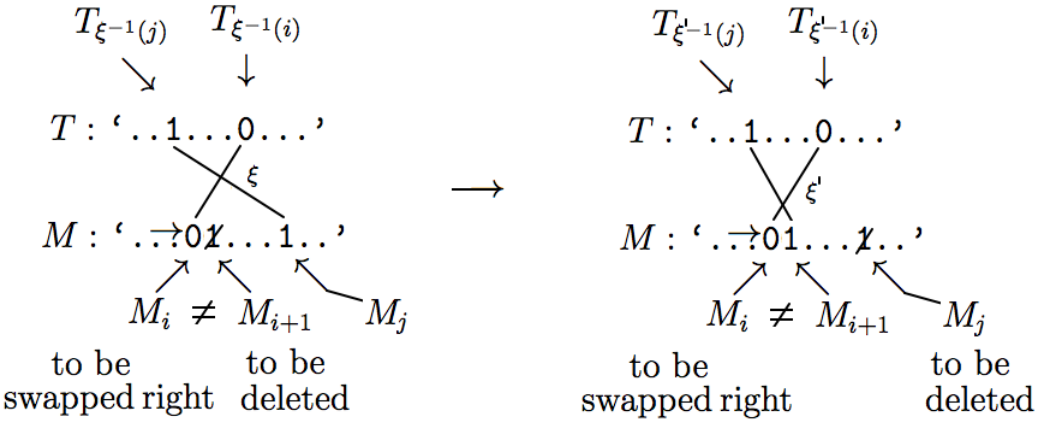

Figure 3.8: Depiction of $\xi^{-1}(i) > i$ case with $M_i \neq M_{i+1}$, before and after invoking the Swap Adjacency Theorem.

swaps. Finally, exactly $m - n$ deletions will always be performed; this number is necessarily invariant, and in no case will the deletion of a symbol adversely affect any swaps that may still need to be performed.

Therefore the k swaps and $m - n$ deletions required in the solution $S(T, M)$ to some instance $\langle T, M \rangle$ may be performed in any order, without affecting the number of operations in S. □

3.16 LCS Length and Optimality

LCS is a key factor in determining similarity of strings T and M. So a natural question arises: can we make decisions about the optimality of a single operation, based on the immediate effect of that operation on the length of an LCS of T and M?

In this section, we explore every possible effect on LCS length after executing an operation. We then show by example that in fact, no direct correlation can be made between change in LCS length and optimality during the course of solving a problem.

In all cases, let $\langle T, M \rangle$ be an instance of BSSC, let l be the length of the LCS of T and M, and let M_k be some symbol in M at index k for which a swap or deletion operation is being considered. We state without proof that deletion of symbol M_k, since it removes a symbol from M, never increases LCS length l; it can only leave l unchanged or decrement l. Of our two allowed operations, the swap of symbol M_k with M_{k+1} is the only operation that can increment l, but it can also leave l unchanged, or even decrement l. Thus there are $2 \times 3 = 6$ cases to consider. See Table 3.2 below.

For the two cases which can never occur, we provide a sketch proof as to why this is true: if deletion of a symbol reduces LCS length, its position with respect to the symbols on each side of it was crucial to preserving the length of any LCS, and therefore a swap of that symbol cannot increment LCS length; and if deletion of a symbol leaves the LCS length unchanged, then either it or an identical symbol next to it were not in the LCS, and therefore a swap of that symbol cannot decrease LCS length.

Δl (DELETE)	Δl (SWAP)	Intuition	Reality
-1	-1	don't SWAP or DELETE M_k	Case 1 below
-1	0	SWAP M_k, don't DELETE	Case 2 below
-1	+1	-	this case can never occur
0	-1	-	this case can never occur
0	0	none	Case 4 below
0	+1	SWAP M_k, don't DELETE	Case 3 below

Table 3.2: Effect of DELETE or SWAP operation on symbol M_k on the length l of the LCS of T and M.

We next provide examples[2] for each of the possible cases, to show that the intuitive operation in each case is not the correct choice.

Case 1: Deletion of symbol M_k decreases l, and swap of M_k also decreases l
This does not imply that M_k should have no action done to it. The following example demonstrates that a symbol to be deleted or swapped might also decrease LCS length if either operation is performed on it:

T : `'011001110111000000'`
M : `'11010100101010111100001'`

Here, swap or deletion of symbol M_8 = `'0'` will cause a reduction in l from 16 to 15. However, deletion $\delta(M, 8)$ is optimal.

In the same example, swap or deletion of M_{14} = `'0'` will cause a reduction in l from 16 to 15. However, swap $\sigma(M, 14)$ is optimal.

Case 2: Deletion of symbol M_k decreases l, swap of M_k leaves l unchanged
This does not imply that M_k should always be saved. The following example demonstrates that a symbol to be deleted might also decrease LCS length:

[2]The examples for each of these cases were generated by program `OpAnalyze`; see Appendix A.

T : ‘011100110001’
M : ‘1100000110000000101110’

Here, deletion of M_2 in block B_1^M causes a reduction in l from 10 to 9, and $\sigma(M, 2)$ effects no change in l; however, deletion of M_2 (or, equivalently, M_1) is optimal.

Case 3: Deletion of symbol M_k leaves l unchanged, swap of M_k increases l
This does not imply that M_k should always be saved (and swapped). The following example demonstrates that a symbol to be deleted can increase LCS length if it is experimentally swapped:

T : ‘01011001000110010 10’
M : ‘1011001100010101101010 01’

Here, swap $\sigma(M, 1)$ increases l from 17 to 18, while deletion $\delta(X, 1)$ leaves l unchanged; but deletion of symbol M_1 is optimal.

Case 4: l is sometimes entirely unaffected, regardless of chosen operation
Consider the following case:

T : ‘0011101111001’
M : ‘1110000010101101000110’

Here, no swap or deletion operation on *any* symbol in M will change l, demonstrating that the LCS length can sometimes be completely unaffected by any operation, regardless of the choice of operation made (and the symbol operated on).

3.17 Monotonicity of LCS Length

We show that for instances of BSSC for which an optimal solution is known, an operation sequence can be found for which the length of the longest common subsequence (LCS) is

monotone nondecreasing as a solution is implemented. We begin with some definitions that will help to facilitate this result.

Let $\langle T, M \rangle$ be an instance of BSSC, and let L be some LCS of T and M. A swap of two adjacent dissimilar symbols in M for which not both of the symbols are in L is a *passive swap*; it is a *perturbing swap* if both of the symbols are in L.

Let $\langle T, M \rangle$ be a reduced instance of BSSC, let L be some LCS of T and M, and let M_b, M_a (with $a = b + 1$) be two symbols in M that form a perturbing swap based on the LCS L and some optimal solution S. Let T_α be the match for M_a and T_β be the match for M_b in S. Let T_θ be the symbol in L corresponding to M_a in L. If $\theta < \beta$, the swap is a *left-oriented swap*; otherwise, $\beta < \theta$ and the swap is a *right-oriented swap*.

Figures 3.9 and 3.10 help illustrate left-oriented and right-oriented perturbing swaps.

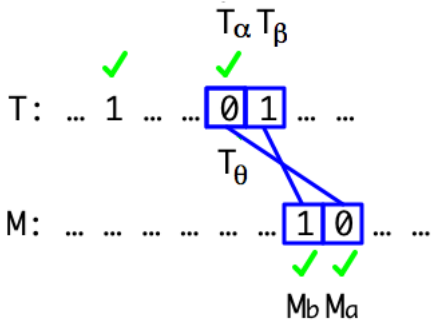

Figure 3.9: Left-oriented perturbing swap. Green checkmarks denote matching members of the LCS in T and M.

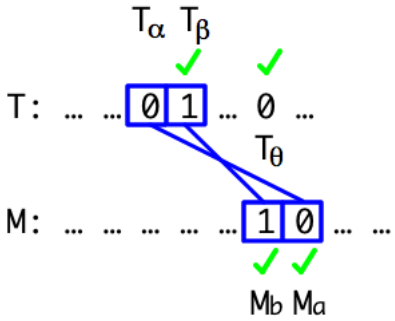

Figure 3.10: Right-oriented perturbing swap.

With these definitions in place we are now ready to proceed with the theorem.

Theorem 5 (Monotonicity of LCS Length). *Let $\langle T, M\rangle$ be a reduced instance of BSSC, and let S be some optimal solution to $\langle T, M\rangle$. Then there exists a particular sequence $\mathcal{O}$ for the operations in S in which the length of the LCS of T and M during the execution of $\mathcal{O}$ is monotone nondecreasing at all times.*

Proof. Our strategy is to avoid deletion and swap operations that may reduce the LCS length. It is sufficient to show that for any reduced instance of BSSC, there exists an LCS involved in a passive swap. By locating these LCS's as we go, we may carefully perform all swaps required in the solution without reducing LCS length; subsequently performing all deletions will then trivially preserve LCS length since T will be a subsequence of M.

If L is such that any of the swaps in S are passive, we can perform them without a reduction in LCS length. Therefore, assume L is chosen such that all optimal swaps are perturbing swaps. Furthermore, assume without loss of generality that the instance contains at least one right-oriented swap; if all of the swaps are left-oriented, reverse the strings and examine the equivalent instance $\langle T^R, M^R\rangle$ (Observation 1), which will have all right-oriented swaps.

Consider the rightmost right-oriented swap of adjacent symbols in M. Call the left symbol from this pair M_b and the right symbol M_a. They are matched with two symbols T_α and T_β in the solution S. Let T_λ be the symbol in T which is the LCS match for symbol M_a in L.

Since the swap is right-oriented, $\beta < \lambda$. Finally, let M_l be the symbol in M to which T_λ is matched in the solution S.

Now consider symbol M_l. If it is part of a swap, that swap is perturbing by assumption and thus M_l is in the LCS. The LCS match for M_l lies to the right of T_λ, implying a right-oriented swap; but this contradicts our assumption that the rightmost right-oriented swap was the one involving M_b, M_a. Therefore M_l is a static symbol. See Figure 3.11.

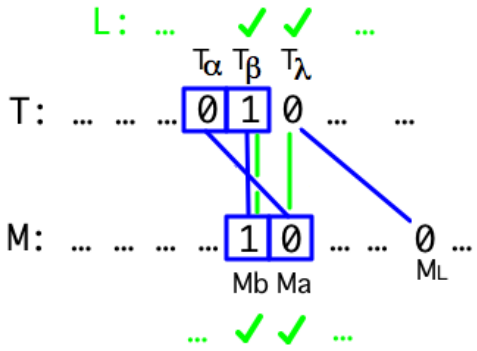

Figure 3.11: Notation example with $T_\alpha =$ '0', $T_\beta =$ '1'.

M_l forms part or all of a block M' of static and/or deletion symbols in M. The end of this block M' is found by reaching the next pair of swap symbols in M, or the end of the string. Each of the k static symbols in M' is matched with a consecutive symbol in T in the solution; call this run of k consecutive symbols T'. (See Figure 3.12)

Since $T_\lambda \in T'$ has as its LCS counterpart a symbol to the left of M' (namely M_a), the only way that all static symbols in M' can be in the LCS is if there exists an LCS counterpart to one of them which lies right of T'; but this would imply the existence of another right-oriented swap, a contradiction. Therefore at most $k-1$ static symbols from M' are part of the LCS. That is, there exists at least one static symbol M_x in M' that is not part of L.

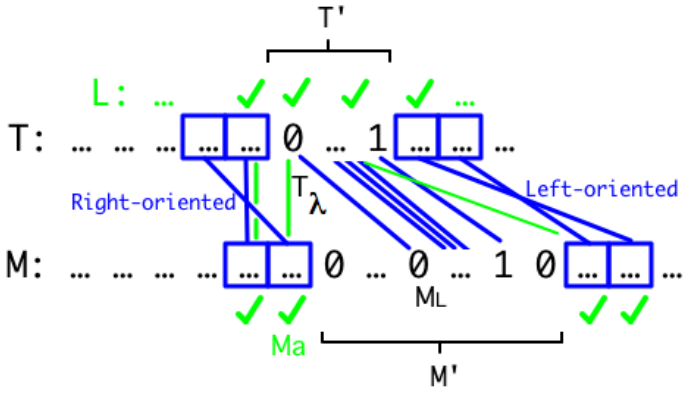

Figure 3.12: Selected LCS matchings in original LCS L.

This allows us room to shift the first of the LCS assignments in M' (those from M_l to M_{x-1}) right one symbol, so that symbol M_a can be excluded from the LCS by having its assignment shifted to M_l, as follows.

Create a new LCS L' by first copying L. We will modify the portion of it that contains symbols from T'. For this portion, match each symbol in T' which is part of L to its matching static symbol in M'. The LCS match for T_λ will now be M_l (instead of symbol M_a) and we have located a passive swap (namely, M_b and M_a) since only M_b remains part of L' (See Figure 3.13).

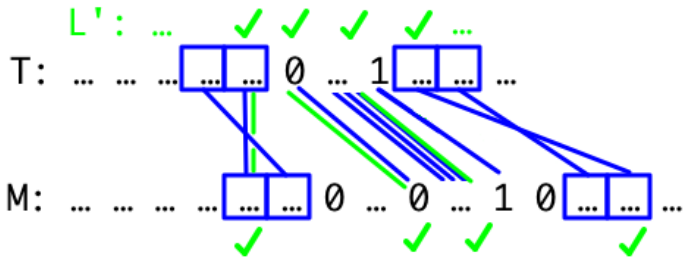

Figure 3.13: LCS matchings in new LCS L'.

Once the LCS L' is created and a passive swap has been found, execute the passive swap. Repeat the process on the resulting instance until T is a subsequence of M. Finally, perform all optimal deletions in any order. Since T is already a subsequence of M, no optimal deletion will decrease the length of the LCS.

Note that we perform all swaps in the solution S first (in a particular order), then all deletions (in any order). This is a valid sequence, since we have previously shown that for BSSC, the set of operations in an optimal solution can be performed in any order, without affecting the optimality of the solution (Theorem 4).

Therefore, for any instance of BSSC in which an optimal solution is known, the set of swaps and deletions can be performed without reducing the length of the LCS. □

Chapter 4

Polynomial-Time Cases

In this chapter we introduce polynomial-time algorithms for solving certain cases of BSSC. Any instances of BSSC for which binary strings T and M take on the forms specified in the following sections can be solved optimally in polynomial time.[1]

4.1 Equal-Zeroes or Equal-Ones Case

We examine here the cases where, for some reduced instance $\langle T, M \rangle$, we have that $\#_1(T) = \#_1(M)$ (without loss of generality). Under these conditions, we can utilize a special case of integer linear programming to find an efficient solution. The *transportation problem* or TP is a classic integer linear programming problem that models the real-world problem of transportation of goods from supply centres to their destinations. Suppose Acme Computer Co. has k assembly plants and l stores that sell Acme Computers. Acme wants to transport its computers from the assembly plants to the stores as cheaply as possible. The route from a particular plant k_i to a particular store l_j has a cost associated with it, based on miles traveled, the kind of vehicle required (ferry, truck etc) and many other factors.

If Acme has 3 assembly plants and sells its computers by the box to 5 computer stores,

[1]We note that for very small instances of BSSC (string sizes of approximately $|T| \leq 9$, $|M| \leq 15$) we have developed an algorithm that optimally solves these cases in $O(n^2)$ time, regardless of the form that the strings take; see Appendix A for details.

its monthly transportation costs might be summarized by a table like Table 4.1 below:

From ↓ To →	Store 1	Store 2	Store 3	Store 4	Store 5	Plant Supply
Plant 1	\$5/box	\$4/box	\$6/box	\$5/box	\$3/box	280 boxes
Plant 2	\$4/box	\$5/box	\$7/box	\$5/box	\$4/box	250 boxes
Plant 3	\$6/box	\$5/box	\$8/box	\$4/box	\$3/box	150 boxes
Store Demand	60 boxes	90 boxes	85 boxes	115 boxes	110 boxes	

Table 4.1: Cost matrix for a sample transportation problem.

Acme's problem, then, is to minimize the overall cost of delivering shipments of computers from the *sources* (assembly plants) to the *sinks* (computer stores) based on these *weighted edges* (the cost of transport for one box between a given plant and a given store).

Mathematically we can model this problem as follows: let the demand at store i be represented by d_i, the supply at plant i be s_i, and the cost of transport of one box between plant i and store j as c_{ij}. Finally, let x_{ij} indicate the number of boxes transported from plant i to store j. Then, the total cost of transport from k assembly plants to l stores will be

$$\text{Total Cost} \;\; C \;=\; \sum_{i=1}^{k} \sum_{j=1}^{l} c_{ij} x_{ij}.$$

To minimize C, we must solve the following problem:

Minimize

$$C \;=\; \sum_{i=1}^{k} \sum_{j=1}^{l} c_{ij} x_{ij}$$

with each integer $x_{ij} \geq 0$, subject to supply constraints

$$\sum_{j=1}^{l} x_{ij} \leq s_i \;\; \text{for} \;\; 1 \leq i \leq k$$

and demand constraints

$$\sum_{i=1}^{k} x_{ij} \geq d_i \quad \text{for} \quad 1 \leq j \leq l,$$

which implies that the supply at least equals or exceeds demand. When supply is exactly equal to demand, the inequalities become strict equalities and a system of $k + l$ linear equations results; this is called a *balanced* transportation problem. In fact, any TP can be converted into a balanced TP by introducing a "dummy" sink which absorbs all unused supply at some arbitrary cost (often 0).

The transportation problem has several excellent-quality polynomial-time algorithms for finding the minimum-cost solution available in the literature. Tardos found a strongly polynomial-time algorithm for finding an optimal solution to transportation problems in 1986 [15]; improvements to the Tardos algorithm were made by Orlin in 1988 [14] and by Kleinschmidt and Schannath in 1995 [7].

Thus, if we can construct a TP that represents our BSSC optimization problem, we can find an optimal solution efficiently. The basic idea for our translation from TP to BSSC is to consider the number of swaps required to transport a symbol from its initial position in M to its mapped position in $M^{(C)}$ (an exact copy of T) in some solution $S(T, M)$. This number of swaps, or distance, can be seen as equivalent to the transport cost coefficient in TP.

The "dummy" sink is also neatly modelled in BSSC, since we need to reduce the size of M to exactly the size of T by deleting unmapped symbols; thus mapped symbols represent transport of supply from sources to sinks, while the deletions can be sent (at a cost of 1 each, our unit cost for the deletion operation) to the dummy sink, which we will call a "trash" sink since the symbols are to be deleted (discarded).

Construction is as follows:

Let $k = \#_1(T) = \#_1(M)$, and let $\#_0(\alpha_i^M)$ be the number of `'0'` symbols between the i^{th} and $i+1^{\text{th}}$ `'1'` in the string M. Define k *sources* $S_0 \ldots S_k$ as follows: source S_i conceptually lies between the i^{th} and $i+1^{\text{th}}$ occurrence of `'1'` in M. Between these `'1'`s could be between 0 and $(|M| - k)$ `'0'` symbols; these `'0'`s represent the supply available from that particular source. Thus assign the supply at each source S_i to be $s(S_i) = \#_0(\alpha_i^M)$. In this way, there will be one source for each of the $k-1$ positions in M lying between successive `'1'` symbols. Additionally we add one at each extreme end of M (before the first `'1'` and after the last `'1'`), for a total of $k+1$ sources (see Figure 4.1).

Define k *sinks* $Z_0 \ldots Z_k$ analogously to sources, but embedded instead within the string T instead of M, and with the number of `'0'` symbols between the i^{th} and $i+1^{\text{th}}$ `'1'` representing the *demand* at sink Z_i. So the demand at a particular sink Z_i is represented by $d(Z_i) = \#_0(\alpha_i^T)$.

To construct the problem as a balanced TP (for which computation of a solution is much more straightforward), we add a "trash" sink $Z_{k+1} = Z_\omega$; all `'0'` symbols in M that are not allocated to one of the k regular sinks after all demand is satisfied can be allocated to our trash sink Z_ω at cost 1 per symbol, the cost of a deletion in BSSC. See Figure 4.1.

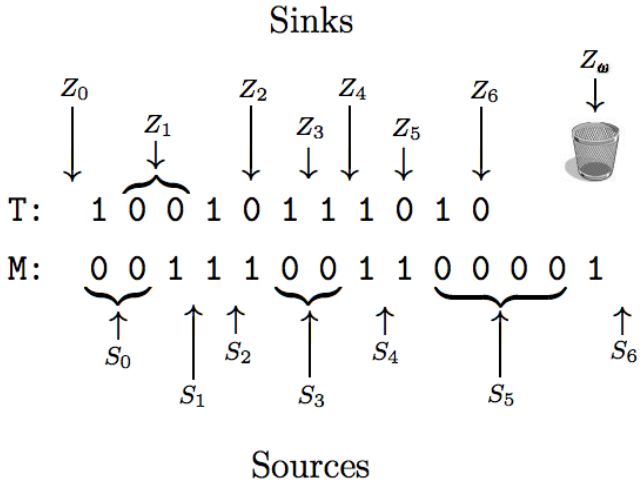

Figure 4.1: Example instance $\langle T, M \rangle$ with $\#_1(T) = \#_1(M)$ and depicted as a transportation problem, with sources and sinks annotated.

The exact number of '0's that are transported (swapped) from source S_i to sink Z_j is recorded by non-negative integer variable X_{ij}.

The cost of transporting a '0', represented by the coefficients in the TP formulation, is $|i-j|$ for a given single transport operation from S_i to Z_j (with the exception of the trash sink). The $|i-j|$ transport cost is due to the fact that the swap operation must swap the '0' over precisely $|i-j|$ '1's in order to reach its target position at Z_j, and since each swap operation is assigned a cost of 1, the total cost for $|i-j|$ swaps is $|i-j|$.

A transportation problem cost matrix for the above example is as follows:

From ↓ To →	Z_0	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	$Z_7 = Z_\omega$	Supply at S_i
S_0	0	1	2	3	4	5	6	1	2
S_1	1	0	1	2	3	4	5	1	0
S_2	2	1	0	1	2	3	4	1	0
S_3	3	2	1	0	1	2	3	1	2
S_4	4	3	2	1	0	1	2	1	0
S_5	5	4	3	2	1	0	1	1	4
S_6	6	5	4	3	2	1	0	1	0
Demand at Z_i	0	2	1	0	0	1	1	8-5=3	

Table 4.2: Cost matrix for example instance from Figure 4.1.

Then, to find the minimum cost $\Psi(T, M)$ of transforming M into T using swaps and deletions, we have the following formulation:

Minimize

$$|S(T,M)| \;=\; \sum_{i=0}^{k}\sum_{j=0}^{k}|i-j|X_{ij} + \sum_{i=0}^{k} X_{i(k+1)}$$

with each integer $X_{ij} \geq 0$, subject to supply constraints

$$\sum_{j=0}^{k+1} X_{ij} = s(S_i) \quad \text{for} \quad 0 \leq i \leq k$$

and demand constraints

$$\sum_{i=0}^{k} X_{ij} = d(Z_j) \quad \text{for} \quad 0 \leq j \leq k+1.$$

We next prove that the cost of this balanced transportation problem with k supply nodes $(S_1 \dots S_k)$, $k+1$ demand nodes $(Z_1 \dots Z_{k+1})$, and $k(k+1)$ directed edges (the transport paths that exist from every source to every sink) is equal to the cost of an optimal solution to the Equal-Zeroes or Equal-Ones (E0E1) case of BSSC. Our method is to show that the cost of an optimal solution to our TP formulation is equal to the cost of an optimal solution to the BSSC instance upon which it was modelled.

We first show that the BSSC solution cost is less than or equal to the TP cost. We have set up the cost matrix such that the cost to transport a symbol from source S_i to sink Z_j is $|i-j|$. Consider that in the E0E1 BSSC, the cost to swap a `'0'` symbol (without loss of generality) from index j to index i is dependent on the number of `'1'` symbols that lie between i and j. Assume without loss of generality that $i < j$. Thus M_j must be swapped left. As it moves left, if it encounters symbol M_z to its left which is also a `'0'`, the swap may be reassigned to M_z (since swaps of identical symbols are never optimal as per Observation 2).

Thus swaps are incurred only when `'1'` symbols are encountered; and since we have set up the TP such that each source lies between the `'1'` symbols in M, there are exactly $|i-j|$ `'1'` symbols between M_i and M_j in M, and thus the `'0'` symbol at M_j will swap with no more than $|i-j|$ `'1'` symbols to move to index i. See Figure 4.2 next page for an example.

T : ‘1 0 0 1 0 1 1 1 0 1 0’

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 1 1 0 0 0 0 1’ Initial Condition

(pick arbitrary ‘0’ from source S_5 to be swapped to S_1)

(this mimics transport of ‘0’ from source S_5 to sink Z_1 in T)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 1 1 0 0 0 0 1’ Reassign to adjacent ‘0’ (no cost)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 1 1 0 0 0 0 1’ Reassign to adjacent ‘0’ (no cost)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 1 0 1 0 0 0 1’ Swap with ‘1’ (cost 1)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 0 1 1 0 0 0 1’ Swap with ‘1’ (cost 1)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 0 1 1 0 0 0 1’ Reassign to adjacent ‘0’ (no cost)

M : ‘0 0 1 $\boxed{S_1}$ 1 1 0 0 0 1 1 0 0 0 1’ Reassign to adjacent ‘0’ (no cost)

M : ‘0 0 1 $\boxed{S_1}$ 1 0 1 0 0 1 1 0 0 0 1’ Swap with ‘1’ (cost 1)

M : ‘0 0 1 $\boxed{S_1\ 0}$ 1 1 0 0 1 1 0 0 0 1’ Swap with ‘1’ (cost 1)

Figure 4.2: Example instance $\langle T, M\rangle$ showing steps in swapping an arbitrary ‘0’ from S_5 (coloured red) to S_1 (shipping to demand at Z_1 coloured blue), incurring 4 swaps at cost $4 = |1 - 5|$.

Thus, since swapping a ‘0’ from S_i to S_j cannot incur more than $|i-j|$ swaps, the BSSC cost is less than or equal to its associated entry in our TP cost matrix. We next show that the TP solution cost is less than or equal to the BSSC optimal solution cost. Each entry in the TP cost matrix for our problem is set to $|i - j|$, where i, j are the indices of the corresponding source S_i and sink Z_j respectively. The only deviation from this cost calculation is for the cost of transport to the trash sink Z_ω; here we have set a fixed unit cost. Consider that we have already shown that there can be no more than $|i - j|$ ‘0’ symbols between symbol M_i and symbol M_j and therefore the cost of the swap of a ‘0’ symbol from S_i to S_j (mimicking transport of the symbol from source S_i to sink Z_j) cannot be greater than $|i - j|$, and thus the $|i - j|$ entry in the TP formulation cannot exceed the associated swap cost. In the case of a deletion, the TP matrix allocates a cost of 1 for transport to the trash sink Z_ω, which is the same as the cost of a deletion under BSSC. Thus the cost of our TP formulation cannot exceed that of the BSSC solution.

Therefore, since we have shown both that our BSSC optimal cost is less than or equal

to the TP cost, and that our TP cost is less than or equal to our BSSC cost, the two values are identical and our formulation is valid.

Hence E0E1 cases of BSSC can be solved efficiently using one of the many methods available for solving transportation problems [7, 14, 15]. The `'0'` symbols chosen for transport under the transportation problem represent `'0'`s in M that are to be saved (and potentially swapped); unused `'0'` symbols from the supply sources in M which are placed into the trash sink Z_ω represent optimal deletions.

The fastest technique currently available is due to Kleinschmidt and Schannath [7]; they developed the strongly polynomial algorithm `STP` for optimally solving a transportation problem with s supply nodes, t demand nodes and k directed edges, as per our model. `STP` runs in time $O(s \log s(k + t \log t))$ where $s \geq t$ without loss of generality. In our case, $s = t = \#_1(T) = \#_1(M)$.

4.2 1, 2, and 3-Block Cases

This section will show that all cases of BSSC for which $\beta(T) \leq 3$ or $\beta(M) \leq 3$ are polynomial-time solvable, regardless of the number of blocks in the accompanying string in the instance.

4.2.1 1-Block Case

1-block cases are extremely simple instances of BSSC for which T or M consists of a string of n `'0'` symbols (or, equivalently, `'1'` symbols). To solve, simply choose any n occurrences of the `'0'` symbol in M (without loss of generality) and delete all other symbols, in linear time.

4.2.2 2-Block Case

We first examine the case where T contains only two blocks: B_1^T and B_{last}^T. Lemmas 6 and 7 respectively map each of these blocks in polynomial time; thus the entirety of T can be

mapped in polynomial time. A simple algorithm to solve this case is as follows (assume without loss of generality that T consists of k `'0'` symbols followed by $n-k$ `'1'` symbols): by choosing the symbols in M such that the maximum number of `'0'` symbols occur prior to the `'1'` symbols, any swaps that remain will be minimized. Thus, choose the first (leftmost) k occurrences of `'0'` and the last (rightmost) $n-k$ `'1'` symbols from M. Delete all others, and perform any swaps that remain.

In the case where M contains only two blocks, and given k `'0'` symbols and $n-k$ `'1'` symbols in T: choose any k `'0'` symbols and $n-k$ `'1'` symbols in M, and delete the remaining symbols from M. The fact that M contains only two blocks means that deletions will not change the block structure of M in any way, and therefore the choices for deletions can be arbitrary.

4.2.3 3-Block Case

We begin with the case where the string T contains 3 blocks. Without loss of generality let B_1^T and B_{last}^T consist of `'0'` symbols and let B_2^T consist of `'1'` symbols. Let k be the size of B_1^T, and let l be the size of B_{last}^T. Map B_1^T and B_{last}^T in polynomial-time by Lemmas 6 and 7 respectively. Delete all other `'0'` symbols in M. At this point we can revert to the TP solution given by the Equal-Zeroes or Equal-Ones Case described earlier; or use the following straightforward (and faster) technique. After mapping all `'0'` symbols in T to M and deleting the extraneous ones, we have a partial mapping of T to M with T_k, the rightmost symbol in block B_1^T, mapped to $M_{\xi(k)}$ and T_{n-l+1}, the leftmost symbol in block $B_3^T = B_{last}^T$, mapped to $M_{\xi(n-l+1)}$. Note that $\xi(k) < \xi(n-l+1)$ since B_1^T and B_{last}^T consist of the same symbols, and crossed mappings cannot occur by Lemma 3.

Now, to map the `'1'` symbols in B_2^T, first choose every available `'1'` symbol from substring $M_\mu = M_{\xi(k)+1} \ldots M_{\xi(n-l)}$. If the supply of `'1'` symbols from M_μ is insufficient to map each of the `'1'` symbols in B_2^T, choose `'1'` symbols in M that are closest to the left or right edge of M_μ (that is, repeatedly choose the `'1'` symbol that requires the fewest swaps to be repositioned within M_μ). If M_μ is empty (that is, $\xi(n-l+1) - \xi(k) = 1$) we can still use the same technique, but we choose the `'1'` symbol that requires the fewest swaps to be

repositioned at index $k = n - l$. See Figure 4.3 for an example.

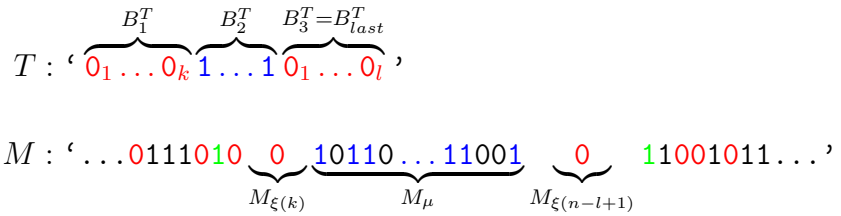

Figure 4.3: Illustrative example of notation in 3-block case. The ‘1’ symbols in T should be mapped first to those lying within M_μ (coloured blue), followed by those ‘1’ symbols nearest to M_μ (coloured green).

This polynomial-time technique is correct because only one destination is possible for the unmapped ‘1’ symbols in T, and their selection is therefore straightforward. The only place that ‘1’ symbols are required in T is B_2^T, and this corresponds to swapping the required number of ‘1’ symbols in M into position to the right of $M_{\xi(k)}$, but left of $M_{\xi(n-l)}$; that is, within M_μ.

For the case where M has 3 blocks: without loss of generality, suppose block B_2^M contains all ‘1’ symbols, and that T contains k ‘1’ symbols. Map k symbols from block B_2^M to the k occurrences of ‘1’ in T. Delete the remaining occurrences of ‘1’ from block B_2^M. T and M now contain an equal number of ‘1’ symbols, and we can revert to the Equal-Zeroes or Equal-Ones case (section 4.1) in polynomial time.

4.3 4-Block Case

We show that all instances of BSSC for which $\beta(T) = \beta(M) = 4$ are solvable in low-order polynomial time.

Without loss of generality, let $\langle T, M \rangle$ be an instance with the following form:

$$T : \text{`}1\ldots10\ldots01\ldots10\ldots0\text{'}$$
and
$$M : \text{`}0\ldots01\ldots10\ldots01\ldots1\text{'}$$

The above color-separated contiguous blocks of symbols we label as $B_1^T, B_2^T, B_3^T, B_4^T$ and $B_1^M, B_2^M, B_3^M, B_4^M$ respectively:

$$T : \text{`}\overbrace{1\ldots1}^{B_1^T}\overbrace{0\ldots0}^{B_2^T}\overbrace{1\ldots1}^{B_3^T}\overbrace{0\ldots0}^{B_4^T}\text{'}$$

$$M : \text{`}\underbrace{0\ldots0}_{B_1^M}\underbrace{1\ldots1}_{B_2^M}\underbrace{0\ldots0}_{B_3^M}\underbrace{1\ldots1}_{B_4^M}\text{'}$$

With a 4-block instance as above there are 3 cases to consider.

1) $|B_1^T| \geq |B_2^M|$:

By Lemma L (Leading Matches), all `'1'` symbols in B_2^M have predetermined assignments. Thus we may assign all `'1'` symbols in T, delete any remaining `'1'` symbols from B_4^M, and revert to the TP solution for the Equal-Zeroes or Equal-Ones case (Section 4.1) in linear time.

2) $|B_4^T| \geq |B_3^M|$:

By Lemma 4 (Trailing Matches), all `'0'` symbols in B_3^M have predetermined assignments. Map all `'0'` symbols in T, delete any remaining `'0'` symbols from B_1^M, and revert to the TP solution for the Equal-Zeroes or Equal-Ones case (Section 4.1) in linear time.

3) $|B_1^T| < |B_2^M|$ and $|B_4^T| < |B_3^M|$:

This third case is the most interesting, and we discuss its solution as follows. This case

means that, after assigning all symbols in B_1^T and B_4^T by Lemma L and Lemma 4 respectively, there exists at least one unclaimed `'1'` in B_2^M, and at least one unclaimed `'0'` in B_3^M. Thus we will have choices in the blocks in M from which we will map symbols to the as-yet unmapped symbols in B_2^T, B_3^T.

With the assignments to B_2^T, B_3^T in mind, the cost to use exclusively symbols from blocks B_2^M and B_3^M is:

$$|B_2^T| \times |B_3^T| \tag{1}$$

This is because (due to the characteristics of T and M that we assumed for this section) all symbols mapped from B_2^M and B_3^M must be swapped, and they must be swapped the maximum possible distance.

The generalized cost for using *only* outer blocks B_1^M, B_4^M as sources for B_2^T, B_3^T is:

$$(|B_1^T| \times |B_2^T|) + (|B_3^T| \times |B_4^T|) \tag{2}$$

Special cases can also occur:

If $|B_2^M| \geq |B_1^T| + |B_3^T|$, we can obtain a cost of $|B_1^T| \times |B_2^T|$; similarly, if $|B_3^M| \geq |B_2^T| + |B_4^T|$, we can obtain a cost of $|B_3^T| \times |B_4^T|$.

Both of these special-case costs are clearly lower than (2), but a simple constant-time comparison would be needed to determine if they are lower than the middle-block swap cost of equation (1). In general: if $|B_2^M| < 2|B_1^T|$, use B_3^M as much as possible to map `'0'` symbols to B_2^T, and the rest from B_1^M. Similarly, if $|B_3^M| < 2|B_4^T|$, use B_2^M as much as possible to map `'1'` symbols to B_3^T, and the rest from B_4^M.

In all situations outlined above, the solutions have been found in linear time.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The Binary String-to-String Correction Problem has proved to have many surprising and intriguing properties. BSSC is ostensibly a restricted subset of the general String-to-String Correction Problem, and yet it is clearly still a problem that requires a sophisticated approach to a polynomial-time solution, if it exists.

In this thesis we have presented the first detailed examination of Binary String-to-String Correction. We have presented several new results about it (some of which are unique to BSSC). These include: determined the effect of both swap and delete on the number of blocks in M; that crossed matchings cannot occur in an optimal solution; that symbols must only ever move in one direction in an optimal solution; established an upper bound on the number operations for a given instance of BSSC; determined the number of swaps required for any equal-length instance; that any two symbols that are to be swapped can always be mapped such that they are adjacent (i.e. without deletions between them) in an optimal solution; that not only leading matches, but trailing matches can be stripped from the problem without changing it; that the symbols from the first and last blocks of T (B_1^T and B_{last}^T) can always be mapped to their first and last occurrences in M, respectively, in an optimal solution; that matching contiguous symbols immediately after B_1^T and immediately prior to B_{last}^T can also be mapped; that extremal symbols in M have defined mappings if the interior blocks have insufficient symbols to match all symbols in T; that the order of operations is

irrelevant; that when examining a single operation, there is no link between its effect on LCS length and the optimality of that operation; and finally that an optimal solution can always be found for which the length of the LCS of T and M is monotone nondecreasing as the solution is implemented.

In addition to theoretical results, we have also demonstrated several forms of the BSSC problem for which a polynomial-time solution exists (other than trivial single-type operation instances). In particular: polynomial-time solutions exist for any instance in which $\beta(T) \leq 3$; if $\beta(T) = \beta(M) = 4$; if $\#_0(T) = \#_0(M)$; or if $\#_1(T) = \#_1(M)$.

5.2 Future Work

There are several opportunities for further study on BSSC:

1. Development of a polynomial-time algorithm for instances of BSSC in which $\beta(T) = 5$ and $\beta(M) = 4$, or $\beta(T) = 4$ and $\beta(M) = 5$. When considering instances with polynomial-time solutions, such a result would be an incremental improvement on the number of blocks in strings T and/or M from $\beta(T) = \beta(M) = 4$, which was the result of this thesis.

2. Initial study of the Ternary String-to-String Correction (TSSC) problem; that is, String-to-String Correction under a 3-symbol alphabet using swaps and deletions only. In particular, its differences from the binary-alphabet case could be compared, since it is likely that the two problems have very different characteristics and form a kind of dividing line in terms of complexity. For instance, it is clear that the order of operations, which we have shown to be irrelevant under BSSC, is crucial under TSSC: transformation of `'021'` into `'10'`, for example, is not possible to do in two operations (one deletion, one swap) unless the deletion of symbol `'2'` is performed first.

3. Perhaps the most interesting future work related to this thesis is discovering the complexity of BSSC in general. We have shown in this thesis that polynomial-time solutions exist for certain instances of BSSC, but the overall complexity of the problem, namely whether the problem is in P or is NP-complete, is as yet unknown; and testing on

even moderately large string sizes is problematic, since our current method for finding an optimal solution, while streamlined as much as possible, still runs in time $O(\binom{m}{n})$. Thus, a potential solution for an instance for which, say, $|T| = 50$ and $|M| = 100$ is unverifiable with current processing power. Our heuristics (see Appendix A) proved successful on instances of BSSC with small string sizes (lengths less than approximately 9 symbols for T and 15 symbols for M) but began to exhibit worse than optimal solutions for larger string sizes, and we estimate that the percentage of optimal solutions for instances with fixed string sizes $n = |T|$ and $m = |M|$ would decrease as n and m increase.

Our intuition tells us to consider the possibility that BSSC is intractable for large-sized string inputs despite its seeming simplicity. Our efforts in this thesis concentrated on finding polynomial-time algorithms for solving BSSC, but future work might concentrate on finding a reduction of BSSC from some suitable NP-complete problem.

Conjecture (Complexity of Binary String-to-String Correction). *The BSSC problem is NP-hard.*

Chapter 6

Bibliography

[1] Abu-Khzam, F., Fernau, H., Langston, M.A., Lee-Cultura, S, and Stege, U. Charge and Reduce: A Fixed-parameter Algorithm for String-to-String Correction. *Discrete Optimization 8*, 41-49, 2011.

[2] Bergroth, L., Hakonen, H., and Raita, T. A Survey of Longest Common Subsequence Algorithms. *Proceedings of the Seventh International Symposium on String Processing Information Retrieval*, 39-48, 2000.

[3] Damerau, F.J. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM 7 (3)* 171-176, 1964.

[4] Edmonds, J. and Karp, R. M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of ACM 19* 248-264, 1972.

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

[6] Hamming, R.W. Error Detecting and Error Correcting Codes. *Bell Systems Technical Journal 29 (2)* 147-160, 1950.

[7] Kleinschmidt, P. and Schannath, H. A Strongly Polynomial Algorithm for the Transportation Problem. *Mathematical Programming 68* 1-13, 1995.

[8] Landau, G.M. and Myers, E.W. Incremental String Comparison. *SIAM Journal on Computing 27* 557-582, 1998.

[9] Lee-Cultura, S. An FPT Algorithm for String-to-String Correction. *M.Sc Thesis.* University of Victoria, 2011.

[10] Levenshtein, V.I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Sov. Phys. Dokl. 163 (4)* 845-848, 1966.

[11] Lowrance, R. and Wagner, Robert A. An Extension of the String-to-String Correction Problem. *Journal of ACM 22* 177-183, 1975.

[12] Maier, D. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of ACM 25 (2)* 322-336, 1978.

[13] Nijenhuis, A. and Wilf, H. *Combinatorial Algorithms for Computers and Calculators.* 2nd ed. pp 28-31, New York: Academic Press, 1978.

[14] Orlin, J. B. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Proc. of the 20th Annual ACM Symposium on Theory of Computing* 377-387, 1988.

[15] Tardos, E. A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs. *Oper. Res. 34* 250-256, 1986.

[16] Wagner, Robert A. On the Complexity of the Extended String-to-String Correction Problem. *STOC '75 Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, 218-223, 1975.

[17] Wagner, Robert A. and Fischer, M. The String-to-String Correction Problem. *Journal of ACM 21* 168-173, 1974.

[18] Watt, Nathaniel. String to String Correction Kernelization. *M.Sc. Thesis.* University of Victoria, 2013.

Appendix

Appendix A

Selected Implementations

This appendix provides a short description of selected algorithms which were developed and utilized for research on the BSSC problem. For brevity the code is not included here, but is available from the author upon request at `tspreen@csc.uvic.ca`. All implementations are written in standard ANSI C and will compile on Windows, Macintosh and UNIX platforms.

A.1 R.R.I.G.

A Random Reduced Instance Generator (`R.R.I.G.`) program was developed which takes as input $|T|$ and $|M|$, and returns a *reduced instance* of BSSC with the given string sizes. Recall that we define a reduced instance as an instance of BSSC for which the following properties hold:

1. $\#_0(T) < \#_0(M)$;
2. $\#_1(T) < \#_1(M)$;
3. $T \not\subseteq M$; (T is not a subsequence of M)
4. $T_1 \neq M_1$; and
5. $T_{last} \neq M_{last}$.

Items 1 and 2 ensure that the instance is solvable, requires the deletion operation, and that we cannot apply our Equal-Zeros or Equal-Ones result from Chapter 4; item 3 ensures that we require the swap operation; and items 4 and 5 relieve us of any requirement to remove any leading or trailing matches from the instance (Lemma L and Lemma 4).

`R.R.I.G.` proved very useful as a time saver. When implemented as subroutine in an experimental heuristic, thousands of reduced (and therefore interesting) instances of BSSC were able to be tested without bothering with trivial instances. Then, if a good result was found from the heuristic based on its performance on these random reduced instances, a more thorough test was conducted in which the exhaustive list of all possible strings of a given size was tested.

A.2 Optimal Solver

Algorithm `OptimalSolver` was developed for one primary purpose: to provide the optimal solution to any instance of BSSC. Such a solution is required in order to check experimental BSSC solver algorithms for correctness. The algorithm takes as input a BSSC instance $\langle T, M \rangle$ and returns the optimal solution using brute force, finding the solution with the fewest operations from among all possible solutions. As such it runs in time $O(\binom{m}{n})$ time, where $m = |M|$ and $n = |T|$. However some optimizations are made in order to reduce the running time as much as possible.

The algorithm uses a single m-bit number b to choose which deletions are to occur, with 1 representing delete and 0 representing save. b begins with the form $0 \ldots 01 \ldots 1$ ($m - n$ 1's and n 0's) and is altered using the so-called "Revolving-Door" Binary Gray Code of Nijenhuis and Wilf [13]. This elegant recurrence generates all combinations of s 0s and t 1s in some binary string of length $s + t$. Thus b is altered until eventually it is of the form $1 \ldots 10 \ldots 0$. In this way, every possible combination of $m - n$ deletions in the instance will be considered. Each iteration of b is used to create a new instance $\langle T', M' \rangle$ with deletions determined by the populated bits in b. After the deletions occur, $|M'| = |T'|$ and the swaps are counted (see Section 3.3) and compared with the lowest swap count yet seen. If the

current swap count is lower than any yet seen, it is marked and b is stored as the current best solution B.

Once b reaches the form $1 \ldots 10 \ldots 0$, B is returned, providing a bitwise representation of the optimal deletions for the instance $\langle T, M \rangle$, and the algorithm terminates.

One optimal solution (the last one found in the list) is displayed on termination; in the displayed solution, a hash (`#`) character appears under each symbol in M which is to be deleted.

A sample of the output from algorithm `OptimalSolver` is as follows:

```
*** Instance #127 ***
|T|:12 T: 011100110001
|M|:22 M: 1100000110000000101110
T has 6 zeroes and 6 ones.
M has 14 zeroes and 8 ones.

Optimal solution is 15 (5 swaps, 10 deletions).
|T|:12 T: 011100110001
|M|:22 M: 1100000110000000101110
           # ####     ####    #
```

A.3 LCS Matrix Analyzer

Algorithm `LCSMatrix` was developed to search for any possible patterns within the LCS matrix, so that such a pattern (if it exists) might be of use in determining an optimal deletion. The input is binary strings T and M, usually generated by algorithm `R.R.I.G.` above. The output is an augmented matrix determining the LCS of T and M. The augmentations are as follows: the algorithm utilizes `OptimalSolver` to indicate the symbols in M which are to be deleted (the `'d'` demarcation in the matrix); and any row which is an exact duplicate of

the one immediately above it (preceding it) is demarcated with a 'DUPLICATE' label.

```
Instance #155 LCS MATRIX
    |  0  1  1  1  0  0  1  1  0  0  0  1
----+------------------------------------
  1 | 00 01 01 01 01 01 01 01 01 01 01 01
d 1 | 00 01 02 02 02 02 02 02 02 02 02 02
  0 | 01 01 02 02 03 03 03 03 03 03 03 03
d 0 | 01 01 02 02 03 04 04 04 04 04 04 04
d 0 | 01 01 02 02 03 04 04 04 05 05 05 05
d 0 | 01 01 02 02 03 04 04 04 05 06 06 06
d 0 | 01 01 02 02 03 04 04 04 05 06 07 07
  1 | 01 02 02 03 03 04 05 05 05 06 07 08
  1 | 01 02 03 03 03 04 05 06 06 06 07 08
  0 | 01 02 03 03 04 04 05 06 07 07 07 08
  0 | 01 02 03 03 04 05 05 06 07 08 08 08
  0 | 01 02 03 03 04 05 05 06 07 08 09 09
d 0 | 01 02 03 03 04 05 05 06 07 08 09 09 DUPLICATE
d 0 | 01 02 03 03 04 05 05 06 07 08 09 09 DUPLICATE
d 0 | 01 02 03 03 04 05 05 06 07 08 09 09 DUPLICATE
d 0 | 01 02 03 03 04 05 05 06 07 08 09 09 DUPLICATE
  1 | 01 02 03 04 04 05 06 06 07 08 09 10
  0 | 01 02 03 04 05 05 06 06 07 08 09 10
  1 | 01 02 03 04 05 05 06 07 07 08 09 10
  1 | 01 02 03 04 05 05 06 07 07 08 09 10 DUPLICATE
d 1 | 01 02 03 04 05 05 06 07 07 08 09 10 DUPLICATE
  0 | 01 02 03 04 05 06 06 07 08 08 09 10
```

A.4 Operation Analyzer

Algorithm `OpAnalyze` takes as input binary strings T and M (generated by `R.I.I.G.` above) and returns the length of the LCS for every possible initial operation on the BSSC instance $\langle T, M \rangle$. As well, the algorithm utilizes `OptimalSolver` described above to display the optimal deletions for the instance.

This gives an easy visual indication of the effect that any of the optimal swap and delete operations will have on the length of the LCS. An initial conjecture was that any optimal operation would always either leave the LCS length unchanged, or increase it by 1. In fact as we have seen in Section 3.16, the LCS length can also *decrease* after an optimal operation.

Sample output is as follows.
Assume input of $T =$`'0111000100'` and $M =$`'101000000110101'`:

```
|T|:10 T: 0111000100
|M|:15 M: 101000000110101
LCS len on swap  :  09 08 08 08 08 08 08 08 08 08 08 08 08 08
LCS len on delete:  08 08 07 08 08 08 08 08 08 08 08 07 08 07 08
                     1  0  1  0  0  0  0  0  0  1  1  0  1  0  1
                                       #  #  #           #     #
```

A.5 Heuristic Algorithm

Algorithm `BSSCSolver` was written to solve small instances of BSSC optimally, and gain insight regarding efficient solutions to BSSC problems of any size. It takes as input $|T|$ and $|M|$, and exhaustively provides a solution for every possible instance of BSSC for the given string sizes. For instance: with input `4, 6` the first instance examined would be $\langle 0000, 000000 \rangle$ and the last would be $\langle 1111, 011111 \rangle$. (Recall from Observation 3 that, for example, $\langle 1111, 110111 \rangle$ need not be solved because it is equivalent to $\langle 0000, 001000 \rangle$, which has occurred earlier in the list of instances).

`BSSCSolver` optimally solves every instance of BSSC for which $|T| \leq 9$ and $|M| \leq 15$, using a total of 34 different strategies for applying swaps and deletions to the problem. The strategies are generally based on pattern recognition; for instance, Strategy 1 is (given a reduced instance, and an excess of symbol α in M):

> *"locate the first two occurrences M_a, M_b of symbol α in M, and the first occurrence T_k of α in T. Then, if $b \geq k-1$, DELETE symbol M_a from string M."*

Strategy 2 is the same, but is performed on the reversed instance. In fact each of the strategies is "paired" in this way, performed once on the given instance and once on the reversed instance. So the 34 strategies are essentially 17 strategies.

For instances with larger string sizes (up to $|T| = 10, |M| = 18$ were tested) the program solves over 96% of all instances optimally while providing worse than optimal (but good quality, within one or two moves of optimal) solutions for the rest. It is expected that as the string sizes of T and M increase, the percentage of optimal solutions found would decrease.

The algorithm runs in time $O(2km)$, where $m = |M|$ and k is the number of strategies employed. The optimality of `BSSCSolver`'s output was verified using the previously described `OptimalSolver`.

In Tables A.1 and A.2 that follow, each table entry represents the number of instances for which a particular strategy yielded an optimal solution.

$(\|T\|, \|M\|) \rightarrow$ Total Pairs $\rightarrow$		$(9, 12)$ 1 293 292	$(9, 13)$ 2 993 564	$(9, 14)$ 6 626 000	$(8, 15)$ 7 607 296	$(10, 14)$ 11 617 000	$(9, 15)$ 14 233 964
Strategy #							
1		1 041 334	2 403 008	5 392 512	6 881 078	8 749 902	11 856 104
2		210 198	481 756	998 834	623 960	2 204 978	1 924 030
3		23 688	48 888	88 082	30 164	279 630	146 190
4		12 138	31 508	63 400	24 004	183 308	112 092
5		2 992	9 612	23 902	13 810	67 722	50 554
6		1 232	4 904	13 216	8 118	32 216	29 118
7		204	5 030	18 176	12 560	30 398	45 532
8		204	2 902	10 346	6 276	19 418	26 320
9		530	3 602	9 508	2 824	27 672	17 716
10		6	394	4 812	1 378	3 058	5 580
11		358	860	1 677	338	8 009	3 254
12		358	860	1 673	338	7 989	3 240
13		0	42	101	1 175	571	6 161
14		0	42	101	1 159	571	5 991
15		18	39	68	14	628	346
16		18	21	68	14	446	304
17		0	21	24	11	401	146

Table A.1: Effectiveness of 34 different heuristics - `BSSCSolver`

(continued next page)

$(\|T\|, \|M\|) \rightarrow$ Total Pairs $\rightarrow$		$(9, 12)$ 1 293 292	$(9, 13)$ 2 993 564	$(9, 14)$ 6 626 000	$(8, 15)$ 7 607 296	$(10, 14)$ 11 617 000	$(9, 15)$ 14 233 964
Strategy #							
18		0	21	24	11	401	126
19		0	21	30	14	401	387
20		0	21	30	14	401	363
21		2	4	6	5	108	94
22		2	4	6	5	108	94
23		0	0	10	1	123	31
24		0	0	10	1	123	31
25		0	0	10	0	122	20
26		0	0	10	0	122	20
27		0	0	0	0	1	6
28		0	0	0	0	1	6
29		0	0	2	0	14	4
30		0	0	2	0	14	4
31		0	2	14	12	48	49
32		0	2	14	12	48	49
33		0	0	0	0	2	1
34		0	0	0	0	2	1

Table A.2: Effectiveness of 34 different heuristics - `BSSCSolver` (continued)

Appendix B

Glossary

B.1 Terms

What follows are definitions of several terms that are commonly used in this thesis. Some new terms that are specific to this document are introduced here; however, some well-known terms such as *string* and *deletion* are redefined here to ensure that the meaning of the terms are clear within the context of this document.

string a sequence of n symbols over some given alphabet Σ, where $n \in \mathbb{N}$ is finite.

binary string a string over a two-symbol alphabet such as $\Sigma_1 = \{\texttt{`0'}, \texttt{`1'}\}$ or $\Sigma_2 = \{\texttt{`a'}, \texttt{`b'}\}$. For this document we will use $\Sigma = \{\texttt{`0'}, \texttt{`1'}\}$ exclusively.

target string a binary string, denoted T, of length n, of which an exact copy is desired by transforming the mutable string M, defined below.

mutable string a binary string, denoted M, of length m, which is to be transformed into the target string T (if possible).

instance two nonempty binary strings: a target string T and a mutable string M. These strings form a BSSC problem.

swap denoted by $\sigma(X, i)$ or just $\sigma(i)$ if the string context is obvious. $\sigma(i)$ in string X is the interchange of the two adjacent symbols X_i and X_{i+1}. For example: given $X =$ `'011011'`, $\sigma(X, 3)$ produces $X' =$ `'010111'`.

deletion denoted by $\delta(X, i)$ or just $\delta(i)$ if the string context is obvious. $\delta(X, i)$ is the removal of symbol X_i from string X. For example: given $X =$ `'011011'`, $\delta(X, 3)$ produces $X' =$ `'01011'`.

in-place for an instance $\langle T, M \rangle$, a symbol M_i, the k^{th} occurrence of that particular symbol in M, is in-place if T_i is also the k^{th} occurrence of that symbol in T.

block a substring $X_r \dots X_s$ of a binary string X such that $X_i = X_r$ for $r \leq i \leq s$, and such that adjacent symbols $X_{r-1} = X_{s+1} \neq X_r$. That is, a block is a maximal run of contiguous identical symbols. See Figure B.1.

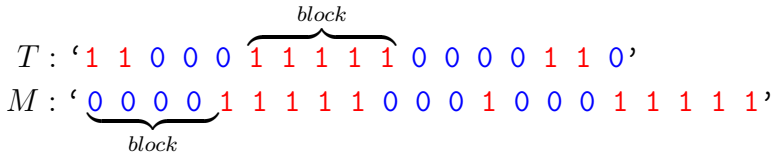

Figure B.1: Example string pair $\langle T, M \rangle$ with blocks coloured.

reduced instance An instance of BSSC for which the following properties hold:

1. $\#_0(T) < \#_0(M)$;
2. $\#_1(T) < \#_1(M)$;
3. $T \not\subseteq M$; (T is not a subsequence of M)

4. $T_1 \neq M_1$; and
5. $T_{last} \neq M_{last}$.

dynamic, static Let $\langle T, M \rangle$ be an instance of BSSC, and let S be some optimal solution for $\langle T, M \rangle$. For a given symbol M_k in M, call M_k a *dynamic* symbol if it is to be swapped or deleted in S; call it *static* otherwise.

passive swap, perturbing swap Let $\langle T, M \rangle$ be an instance of BSSC, and let L be some LCS of T and M. A swap of two adjacent dissimilar symbols in M for which not both of the symbols are in L is a *passive swap*; it is a *perturbing swap* if both of the symbols are in L.

left-oriented swap, right-oriented swap Let $\langle T, M \rangle$ be a reduced instance of BSSC, let L be some LCS of T and M, and let M_b, M_a (with $a = b + 1$) be two symbols in M that form a perturbing swap based on the LCS L and some optimal solution S. Let T_α be the match for M_a and T_β be the match for M_b in S. Let T_θ be the symbol in L corresponding to M_a in L. If $\theta < \beta$, the swap is a *left-oriented swap*; otherwise, $\beta < \theta$ and the swap is a *right-oriented swap*.

(See Figures B.2 and B.3 next page)

B.2 List of Commonly Used Symbols

$\langle T, M \rangle$ an instance of the Binary String-to-String Correction (BSSC) problem, where T is the target string and M is the mutable string.

X_i the i^{th} symbol of string X, beginning at index 1. the last (rightmost) symbol is sometimes denoted X_{last}.

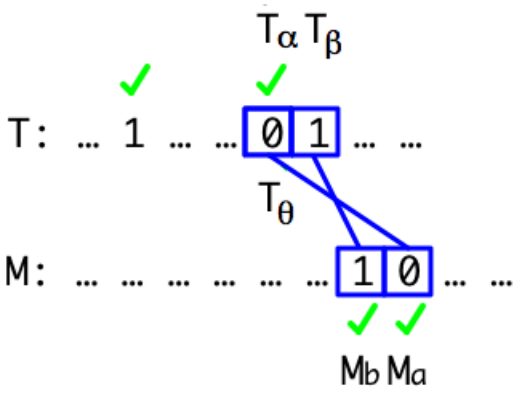

Figure B.2: Left-oriented perturbing swap. Green checkmarks denote matching members of the LCS in T and M.

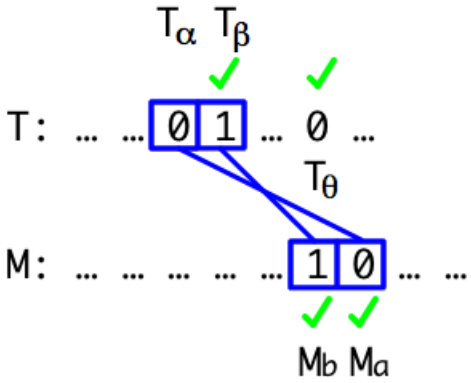

Figure B.3: Right-oriented perturbing swap.

B_i^X the i^{th} block in the binary string X, counting from left to right. The last (rightmost) block is sometimes denoted B_{last}^X.

$\beta(X)$ the number of blocks in the binary string X. For example: $\beta(000110100) = 5$.

$\#_0(X)$ the number of `'0'` symbols in the binary string X.

$\#_1(X)$ the number of `'1'` symbols in the binary string X.

$S(T, M) = (M^{(0)}, M^{(1)}, \ldots M^{(C)})$ A solution to the instance $\langle T, M \rangle$ using C operations, where $M^{(0)} = M$, $M^{(C)} = T$, and such that for any given $M^{(i)}$, $0 \leq i < C$, $M^{(i+1)}$ differs from $M^{(i)}$ by exactly one deletion or swap operation.

$|S(T, M)| = C$ above; it is the cost of (or equivalently, the number of operations in) the solution S to the instance $\langle T, M \rangle$.

$\Psi(T, M)$ The number of operations required in an *optimal* solution to the instance $\langle T, M \rangle$. That is, it is the least number of operations required to solve the instance $\langle T, M \rangle$, over all possible solutions.

ξ The *rearrangement map* for $S(T, M)$; a one-to-one function $\xi : \{1, \ldots, n\} \to \{1, \ldots, m\}$ that, for instance $\langle T, M \rangle$ and given index i, $1 \leq i \leq n$, yields the index in M to which symbol T_i is mapped in a solution $S(T, M)$ to the instance. Thus $M_{\xi(1)} M_{\xi(2)} \ldots M_{\xi(n)} = T_1 T_2 \ldots T_n$.

ξ^{-1} The partial function denoting the inverse of the rearrangement map. If $\xi(a) = b$, then $\xi^{-1}(b) = a$ and $\xi(\xi^{-1}(a)) = \xi^{-1}(\xi(a)) = a$.

$\sigma(X, i)$, $\sigma(i)$ *swap* operation of adjacent symbols X_i and X_{i+1} in the string X, exchanging their positions in X. See Section B.1.

$\delta(X, i)$, $\delta(i)$ *deletion* operation of symbol X_i from the string X. See Section B.1.

$\subseteq$ In the context of this document, $X \subseteq Y$ means that string X is a subsequence of string Y.

$\not\subseteq$ In the context of this document, $X \not\subseteq Y$ means that string X is not a subsequence of string Y.

Δ "Change in". Used variously as a prefix of some other symbol α, to indicate a change in the value or position of α.

$[\![P]\!]$ Iversonian notation. If P is a proposition, then the value of $[\![P]\!]$ is 1 if P is true and is 0 if P is false.