6CCS3PRJ Final Year

String Sanitization: A Novel Strategy for Minimizing Data Utility Loss

Final Year Indiviual Project

string sanitization

Author: Ye Mao

Supervisor: Dr.Grigorios Loukides

Student ID: 1836436

Programme of Study: BSc Computer Science with Robotics

King's College London

April.09 2021

Abstract

String data are often disseminated to support applications such as location-based service provision or DNA sequence analysis [9]. However, confidential knowledge of data might accidentally be exposed during this dissemination. Such confidential knowledge is modelled by a sensitive pattern (*i.e.,* substring). The existing algorithms have proven that they can conceal all occurrences of sensitive patterns.

However, the current sanitization methods also make a sanitized version of data highly distorted. High distortion in data is malicious to the utility of data and possibly generates many spurious patterns that harm the accuracy of frequent pattern mining. Consequently, the primary computational problem is to guarantee that the sanitized data has a low distortion of the non-sensitive pattern, especially low distortion of the spurious pattern while avoiding re-exposure of sensitive patterns. Our contribution here is fourfold. First, we proposed a new problem of deleting characters in a string to reduce distortion and enhance data utility. Also, we developed a new score function named ***R-score*** to rank all deletions. Secondly, we proposed a greedy algorithm, **GD-ALGO**, to construct a local-optimum string to reduce distortion level and the number of spurious patterns sharply. Thirdly, we proposed **ELLS-ALGO** *(Efficient Low-Utility Loss Sanitizing)*, which integrates some Monte Carlo Tree Search ideas to more efficiently find suboptimal string under a specified number of deletions. Lastly, we designed a **python-based GUI** and a **Command Line Interface** to allow the user to manipulate parameters and variables within the algorithm without having to understand the programming details behind it. An extensive experimental study on various datasets has shown that both algorithms can effectively and significantly mitigate distortion without reinstating sensitive patterns.

Key Words: data sanitization, frequent pattern mining, data privacy, spurious patterns, distortion

Originality Avowal

I verify that I am the sole author of this report,except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copy of the submitted work for purposing of marking, plagiarism detection and archival, and upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceeds 25,000 words.

Ye Mao

April 09, 2021

Acknowledgements

I want to thank my supervisor Dr Grigorios Loukides, whose expertise was invaluable in formulating the research questions and methodology. His papers and guidance pushed me to sharpen my thinking and deepen my understanding of the research area.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Project Motivation

A string is a sequence of *letters* over some alphabet Σ that is widely used to represent individual data in domains ranging from transportation to web analytics and bioinformatics [8]. For instance, it may represent an essential part of a patient's DNA sequence, with *letter* corresponding to DNA bases [9]. Analyzing and exploiting such strings is important in applications, including product recommendation and DNA sequence analysis [9]. As a result, such strings are often disseminated outside of the collecting party.

However, the process of disseminating strings often leads to the exposure of confidential knowledge. For example, many big companies often anticipate user privacy information when disseminating data because of management omissions. Thus, it is necessary for us to sanitize the string before its dissemination so that the confidential knowledge can be fully concealed [9]. Simultaneously, when we sanitize the data, we should maintain its utility, which leads to the result that such data can still be accurately mined using any mining technique, such as frequent pattern mining. Current methods have proven to be effective in hiding sensitive patterns (*confidential knowledge*) in the string. However, when concealing such patterns, the data may be distorted so that its mining utility will be significantly impacted. Currently, concealing confidential knowledge while minimizing distortion (*i.e.,* data utility loss) remains a computational challenge [8]. This unresolved problem motivates me to have a deeper study of this field.

1.2 Project Aim & Objective

Aim:

This project aims to investigate a strategy for improving the data utility of the sanitized string by reducing distortion of spurious or non-spurious patterns while not exposing any confidential knowledge.

Objective:

The first objective is to determine the relationship between pattern occurrences and data distortion. Second, investigate the effect of symbol deletion on pattern occurrences and then determine if it can be used to minimize distortion or reduce spurious patterns. Third, based on the impact of deletion on data distortion, design a score function to quantify each deletion. The delete that results in the most significant utility enhancement should always have the lowest score. Furthermore, develop two algorithms that use such a score function to select the *sequence* of most suitable symbols to remove, resulting in the maximum spurious or non-spurious pattern distortion reduction. In addition, using publicly available datasets from various research domains, we evaluate the performance of both algorithms in terms of efficiency and effectiveness. The final objective is to create a graphical interface and a command-line interface for more convenient use and understanding of algorithms.

1.3 Summary Contributions

In this project, we define a new problem, referred to as LUL (Low Utility Lost) for *String Sanitization*, which aims at constructing a string $\mathcal{H}$ to minimize spurious pattern in sanitized string by deleting a certain number of *letters*. For LUL, we developed two different algorithms: GD-ALGO and ELLS-ALGO. Both algorithms use the feature of deletion to clear up spurious (*i.e.,* τ-ghost) patterns that are generated in sanitized string Z and reduce data distortion. To ensure that the deletion will not further deteriorate the utility of the data, we also designed a proper score function R-*score* to rank deletions. The score function rewards deletions that minimize distortion while heavily punishing deletions that reinstate sensitive sub-

strings. GD-ALGO iteratively find optimal delete at this moment until finding all suitable *letters* to delete. ELLS-ALGO applies Monte Carlo tree search (*i.e.,* a type of heuristic method) to reduce the search space while improving the the quality of result. Finally, we developed a python-based GUI and CLI(Command-line interface) for evaluating the performance of two algorithms.

Previous approaches mainly focused on concealing confidential knowledge while preserving data utility. However, some papers [8] already provide solid evidence that this problem cannot be solved in polynomial time and is hard to approximate. Therefore, without any utility loss when sanitizing data is extremely challenging to achieve. The main breakthrough of this project is that we proposed a new research direction to solve the problem of utility loss in sanitization methods. Rather than preserving data utility during sanitization, we devise a post-optimization method for optimizing the sanitized string result after sanitization. Furthermore, we established a new problem relevant to this research subject and two corresponding algorithms to solve it.

Chapter 2

Preliminaries & Literature Review

2.1 Preliminaries

An alphabet Σ is a finite set whose elements are called *letters*. We denote the set of all length-k substring over Σ by Σ^k. We fix a string $X = X[0], X[1]...X[n-1]$ of length $|X| = n$ over Σ. We call any substring $X[i..i+k-1]$ with length-k a pattern. The start position and end position of substring are denoted by i and j. $X[i..j] = X[i]...X[j]$ is the substring of X. For any substring *U* in X, we denote the number of occurrences of U by $Freq_X(U)$. The number of occurrences is also called frequency. A substring $X[i..i+k-1]$ is referred to as *sensitive* if and only if $X[i..i+k-1] \in \mathcal{S}$. Therefore, we call $\mathcal{S}$ is the set of the occurrences of sensitive substrings. Also, the set of all patterns, which cannot be mined from *string* W but can be mined from *string* Z, at a given threshold τ is denoted by $\mathcal{P}_{W,Z}$. Such patterns are also referred to as $spurious$ patterns.

Each deleted *letter* is denoted by x, and i represents its index. Given a string Z and deleted *letter* x, $\mathcal{M}_Z(x)$ denotes all substrings in Z, whose number of occurrences changes as a result of deletion. Such patterns are also called affected patterns. $\mathcal{J}_Z(x)$ represents the set of patterns in Z whose frequency increases after deleting letter x. We denote the set of patterns in Z whose frequency decreases by $\mathcal{D}_Z(x)$.

By $d_{X,Y}(U)$ we denote the frequency difference (*i.e.,* $Freq_X(U) - Freq_Y(U)$) of string U between two strings X and Y. Such a frequency difference is also referred to as frequency shift. Also, we denote the *R-score* of deleting letter x by $R(x)$ or $R(i)$ where i is the index of x.

All other notations and terminology are standard. We denote the absolute value of the number x by $|x|$. We also use normal functions min(A), max(A), sum(A)

(*i.e.,* A is an array). Also, we use the usual math calculation operation, ordering relations, and set *letters*.

2.2 Data Privacy and Data Mining

Data privacy or information privacy is a branch of data security concerned with the proper handling of data – consent, notice, and regulatory obligations [18]. There are two reasons why data privacy is one of the most significant issues in our industry. Firstly, data plays an important role in large companies. A clear trend demonstrates that there is an increasing number of companies generating revenues by sharing and using data. These companies process a large amount of data every day. How to collect, store and use data is a particularly critical issue. Any company that does not comply with the applicable privacy laws, such as GDPR, will be punished severely today. For example, in 2018, Facebook faced a huge fine, due to the fact that it violated General Data Protection Regulation (GDPR). Secondly, privacy is the right of a citizen to be protected from uninvited surveillance. Obtaining permission to use the customer's private data is critical in a democratic society.Even if authorized, posting any private information that harms people's reputation is also not allowed. Data mining has attracted more and more attention in recent years,

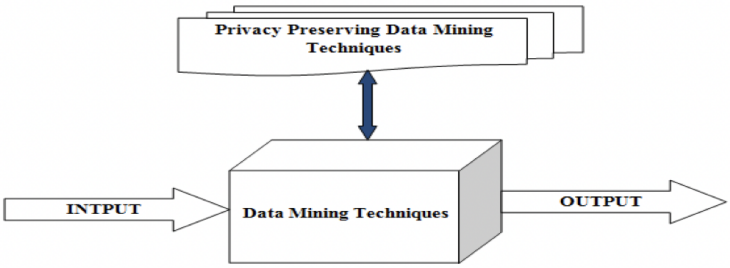


Figure 2.1: Privacy preserving data mining model [2]

probably because of the popularity of the "big data" concept [4]. Despite that the information discovered by data mining can be precious to many applications, people have shown increasing concern about the other side of the coin, namely the privacy threats posed by data mining. To deal with the issue, a sub-field of data mining, referred to as privacy-preserving data mining (PPDM), has achieved breakthroughs

[4].

The primary purpose of current PPDM algorithms or models is to hide sensitive data from certain mining operations. However, most algorithms often lead to tradeoffs between privacy protection and information loss which may affect the accuracy of the mining algorithm.

2.3 Frequent Pattern Mining

2.3.1 Introduction of Frequent Pattern Mining

In recent years, frequent pattern mining (FPM) has been a core topic of data mining. The original purpose of FPM was to analyze the purchase behaviour of customers by finding the correlations between different customers' preference items. This mining technique is now used in various board applications, including encompassing clustering, classification, software bug detection, and recommendation system design. The main idea of FPM is to search for frequent patterns, associations from a given data set found in various categories of databases (*i.e.,*transactional database). Frequent patterns can be an itemset, subsequence, or substructures that appear in a large-scale data set with a frequency higher than a predefined minimum support value. A frequent itemset is a group of objects, such as a computer, iPad, and mobile phone, that appear frequently in the same transaction dataset. A subsequence, such as purchasing first a PC, then a digital camera following this a memory card, if it frequently occurs in a shopping history database, is a (frequent) sequential pattern [7]. A substructure can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences [7]. If a substructure frequently occurs in a graph database, it is called a (frequent) structural pattern [11].

Discovering frequent patterns is critical in the search for associations, or interesting relationships between data. In this research topic, it is important to propose an algorithm for efficiently searching for a complete set of frequent patterns.

2.3.2 Frequent Pattern Mining Techniques

Throughout history, many efficient algorithms were proposed for searching a complete set of frequent patterns. Agrawal and Srikant proposed the **Apriori** algorithm in 1994 [1]. *Apriori* is a common FPM algorithm that is primarily used for association rule analysis. This method applies an iterative approach or level-wise search, in which frequent subsets are extended one item at a time. This step is known as candidate generation. The algorithm can only be terminated if no further extensions are found. The main feature of the Apriori algorithm is to assume that all subsets of a frequent itemset must be frequent; if an itemset is infrequent, all its supersets will be infrequent. This assumption effectively reduces the search space. However, the main limitation of this approach is that the database has to be scanned multiple times to ensure the accuracy of mining. Therefore, the algorithm is very inefficient when the database is reasonably large. **FP-Growth** [12] algorithm is an improved version of the Apriori algorithm. By pattern fragment growth, it searches for a full collection of frequent patterns. Rather than using candidate generation in Apriori, this method uses the prefix-tree structure, also known as FP-tree, to compact essential knowledge about frequent trends. FP-growth, in comparison to Apriori, is a more scalable and efficient technique.

2.3.3 Frequent String Mining

String data is the most common form of data type, including texts, nucleotide bases 'A', 'G', 'C' and 'T' in DNA sequences, or amino acids for protein sequences [20]. Sometimes, we may try to concentrate on discovering frequent string patterns. In bioinformatics, such frequent strings are referred to as "motifs". The typical problem is to enumerate all the frequent motifs or frequently closed motifs from a given string. This problem is known as *Frequent String Mining*, which is a crucial problem in the field of Frequent Pattern Mining.

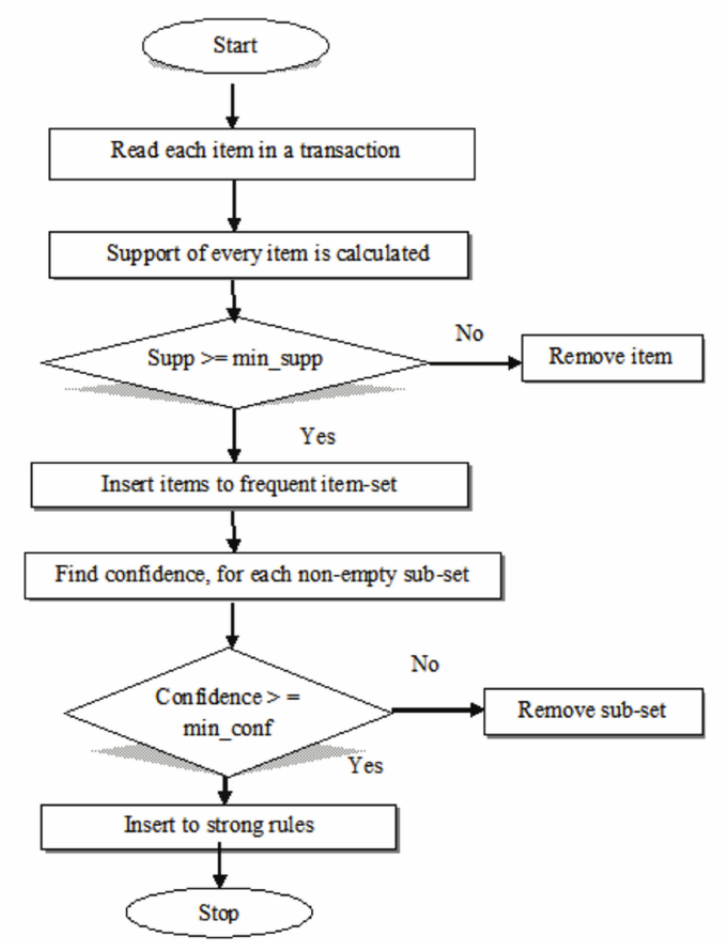


Figure 2.2: Apriori-algorithm flowchart [1]

2.4 String Sanitization

2.4.1 Introduction of String Sanitization

In many domains, such as bioinformatics and transportation, data is extensively modelled as *String* (*i.e.,* a sequence of characters over a finite alphabet Σ) [9]. Data in this format is often disseminated to support applications such as location-based service provision or DNA sequence analysis [9]. This dissemination, however, may expose sensitive patterns that model confidential knowledge (e.g., trips to mental health clinics from a string representing a user's location history) [9]. Therefore, to prevent the exposure of that confidential knowledge, it is necessary to sanitize string data before disseminating it. Simultaneously, during the sanitizing procedure, it is equally essential to preserve the utility of the string. Therefore, to resolve this privacy-utility tradeoffs problem, a new concept, referred to as string sanitization, was proposed.

2.4.2 Existing Sanitization Model

Combinatorial String Dissemination (CSD) [9] is one of the most typical current state-of-art sanitization approaches, which aims to sanitize a string of data by concealing the occurrences of all sensitive patterns. At the same time, this model can also perform well in preserving data utility. More specifically, this model works by enabling string data to meet a series of constraints and desirable properties while being disseminated. For instance, the constraints aim to capture privacy requirements, and the properties aim to capture data utility considerations.

- *Constraints:*

  (C1): For an integer $k > 0$, no given length-k sub-string (also called patterns), which models confidential knowledge occurs in sanitized string X. Such a k-length sub-string is named *sensitive pattern.*

- *Properties:*

  (P1): The order of appearance of all other length-k sub-strings (also called *non-sensitive patterns* is the same in the original String W and X.

  (P2): The frequency of other length-k substrings (non-sensitive patterns) is the same in W and in X.

To satisfy the above constraints and properties, this model defined three problems, which are TFS (Total Order, Frequency, Sanitization), PFS (Partial Order, Frequency, Sanitization), MCSR (Minimum-Cost Separators Replacement) problem. The CSD model creatively solves these problems by proposing **TFS-ALGO** [9], **PFS-ALGO** [9] and **MCSR-ALGO** [9].

1. **TFS** [9]: *Given an input String W, a pattern length, k, the set of occurrences of sensitive patterns $\mathcal{S}$, and a different set $\mathcal{I}$* construct the shortest string X.

   (a) X does not contain any sensitive patterns in W.

   (b) $\mathcal{I}_{\mathcal{W}} \equiv \mathcal{I}_{\mathcal{X}}$

   (c) $Freq_X(U) = Freq_W(U)$, *for all non-sensitive patterns* $U \in \Sigma^k$

2. **TFS-ALGO** [9]: TFS-ALGO reads and traverses all characters in W, from left to right, and appends them to X depends on two rules.

   (a) Rule 1: Replacing the last letter of sensitive pattern U with $\#$ and subsequently appending the longest prefix V of the succeeding non-sensitive pattern U right after $\#$.

   (b) Rule 2: When $k-1$ *letters* before $\#$ is equal to $k-1$ *letters* after $\#$, simplify string by removing $k-1$ *letters* after $\#$ and replacement $\#$. (*i.e.* $abb\#abb \rightarrow abb$)

   Pros and Cons: TFS-ALGO has proven to be able to construct the shortest string to meet P1, P2 without violating C1. Also, the time complexity of the algorithm is $\mathcal{O}(n)$ time. However, the replacement letter generated from TFS-ALGO may reveal the exact position of the sensitive pattern. Therefore, sensitive substring may also be recovered by the "undo" rule.

3. **PFS** [9]: *Given $W, k, \mathcal{S}$,and $\mathcal{I}$ construct a sanitized string Y.*

   (a) Y does not contain any sensitive patterns in W.

   (b) Partial Order (*i.e.*,the order of appearance only for sequences of successive non-sensitive length-k substrings that overlap by $k-1$ *letters*.) of String U in W is equivalent to the Partial Order in Y.

   (c) $Freq_Y(U) = Freq_W(U)$ *, for all non-sensitive patterns* $U \in \Sigma^k$

4. **PFS-ALGO** [9]: The main idea of the algorithm is to preserve the partial order of non-sensitive patterns rather than a total order. Therefore, a shorter string will be generated by further application of Rule 2.

   However, this algorithm still cannot eliminate all separator *letters* $\#$. Therefore, sensitive substring can also be recovered.

5. **MCSR** [9]: *Given Y over an alphabet $\Sigma \cup \{\#\}$ with $\#, \delta > 0$, and parameter τ and θ, construct a new string Z, by replacing separator $\#$ with letters from Σ*

- τ: The user-specified minimum support. In frequent pattern mining, the pattern with a frequency higher than the minimum support value will be recognized as a frequent pattern.
- τ-*ghost*: A length-k string over Σ, whose frequency is lower than τ in Y but at least τ in Z.
- θ: The maximum allowable replacement distortion level.
- δ: The number of ghost pattern.

6. **MCSR-ALGO** [9]: MCSR-ALGO is a heuristic algorithm, which seeks to solve the MCSR problem by creating an instance of *Multiple Choice Knapsack Problem* (MCK). Each # can be replaced by all *letters* in Σ and empty string (*i.e.,* delete letter). Furthermore, the functions *ghost()* and *Sub()* are proposed to measure the cost of τ-ghost and distortion level, respectively, and the results calculated by the two functions are set as the input of MCK. MCSR-ALGO eventually solves the instance of MCK and translates the solution back to a sub-optimal solution of the MCSR problem.

   In a nutshell, although all algorithms mentioned above can successfully conceal all sensitive substrings and remove replacement separators, they still have some downsides. There is a possible risk of incurring distortion of the string due to the replacement strategy. Such distortions can incur the incidence of τ-ghost. Maintaining the data utility of data mining (*i.e.*, Frequent Pattern Mining, Sequential Pattern Mining) will be a complex problem if the number of τ-ghost patterns is high. Furthermore, since minimizing the τ-ghost pattern is an NP-hard problem, no general solution can guarantee effectiveness and performance simultaneously [8].

2.4.3 Spurious Pattern *versus* Data Utility

The frequency of the pattern may be distorted during the sanitization process. Such frequency changes can make some patterns no longer frequent or no longer non-frequent. Such patterns are referred to as *spurious* patterns, and can significantly influence the accuracy of FPM. In this situation, using the

same mining threshold may mine completely different patterns, which are comprised of many false-positive patterns or true-negative patterns. Avoiding the occurrences of a spurious pattern is vital to preserving data utility.

Example: Let W be a orginal string $abbbabaaba$, Z be a sanitized string $aabbbbabb$, threshold θ = 2, length of pattern k = 3. Note that, Z contains no sensitive pattern. Then, we can get a set of all length-k frequent patterns F (*i.e.,* $Freq_W U \geq \theta$) in W, and a set of all length-k frequent patterns $\mathcal{S}$ (*i.e.,* $Freq_Z U \geq \theta$), in Z. $F = \{aba\}$, $\mathcal{S} = \{abb, bbb\}$. We can find $F \neq \mathcal{S}$. It means aba no longer be frequent in sanitized string and abb/bbb can be mined by FPM.

The intuitive approach to eliminating spurious patterns is to set a higher minimum support threshold. However, such a technique may filter out frequent patterns too. Therefore, string sanitization is a tradeoff between hiding and mining data. Concealing # without utility loss is still a challenge in this field because of its hardness and inapproximability [8].

2.5 Combinatorial Optimization Problem

Combinatorial optimization is a topic that consists of finding an optimal item from a finite set of items. Such problems are NP-hard problems and intractable in an exhaustive search. Some typical combinatorial optimization problem examples:

- Traveling Salesman Problem(TSP)[1]: Given a set of cities' locations C, find the shortest possible path P that visits each city exactly once.
- Subset Sum Problem(SSP)[2]: Given a set of n positive integers, and a target value t, determine if there is a subset of the given set with a sum equal to t.

Combinatorial Optimization Problem may also occur in *String Sanitization*. For example, MCSR-ALGO (see in 2.4.2) seeks to find the optimal replacements of # (recall that # is used to replace sensitive pattern). Each replacement letter was

[1] Hoffman, K. L., Padberg, M., Rinaldi, G. (2013). Traveling salesman problem. Encyclopedia of operations research and management science, 1, 1573-1578.

[2] https://en.wikipedia.org/wiki/Subset_sum_problem

given weight in order to avoid reinstating the sensitive pattern. Total weight cannot exceed the distortion bound level. The cost of replacement is simply the number of resulted ghosts (*i.e.,* spurious patterns). Therefore, this problem is a variant of the knapsack problem [19] (*i.e.,* a type of subset sum problem). However, because the space of possible solutions is typically too ample to search for, finding a global solution is an arduous task.

Traditionally, the greedy algorithm is an effective way to search for the solution to optimization problems. However, this algorithm is merely used to find local optimum outcomes. With increasing problem complexity, the quality of the outcome decreases. To prevent a local-optimal solution, heuristic algorithms using probability and statistics techniques were proposed to find the "estimated best solution".

2.5.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic algorithm guided by Monte Carlo simulation. Coulom originally proposed MCTS (2006) [5] as an algorithm to play the two-person zero-sum game with perfect information called Go. MCTS handles problems with a high branching factor effectively. As it gains information (*i.e.,* learning process), MCTS increasingly favors more promising moves that have better long-term reward, making its search *asymmetric*[3]. Another good feature is that MCTS balances the exploration and exploitation ratio during the search process based on the Upper Confidence bounds applied to Trees (UCT[4]) and Gradient Boosted Decision Trees (GBDT[5]). UCT formula is given by,

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

where w_i is the number of wins, n_i is the number of simulation, N_i is the total number of simulations, c is the exploration parameter.

For problems with higher complexity, MCTS outperforms other popular heuristic algorithms, such as the Genetic algorithm [15] and ant colony optimization algo-

[3] https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238

[4] Kocsis, L., Szepesvári, C. (2006, September). Bandit based monte-carlo planning. In European conference on machine learning (pp. 282-293). Springer, Berlin, Heidelberg.

[5] https://c3.ai/glossary/data-science/gradient-boosted-decision-trees-gbdt/

rithm [6]. The MCTS process can be subdivided into four steps which are repeated iteratively: selection, extension, simulation, and back-propagation (See Fig.2.3). The selection phase recursively selects the node with the maximum UCT value until the leaf is reached. The expansion phase expands the search space. The simulation step runs a simulated rollout until the result is seen. The back-propagation phase finally updates all nodes' values with the result from the simulation. Furthermore, this algorithm is effective and has strong flexibility. As a consequence, it can be easily incorporated into the solution of a combinatorial optimization problem.

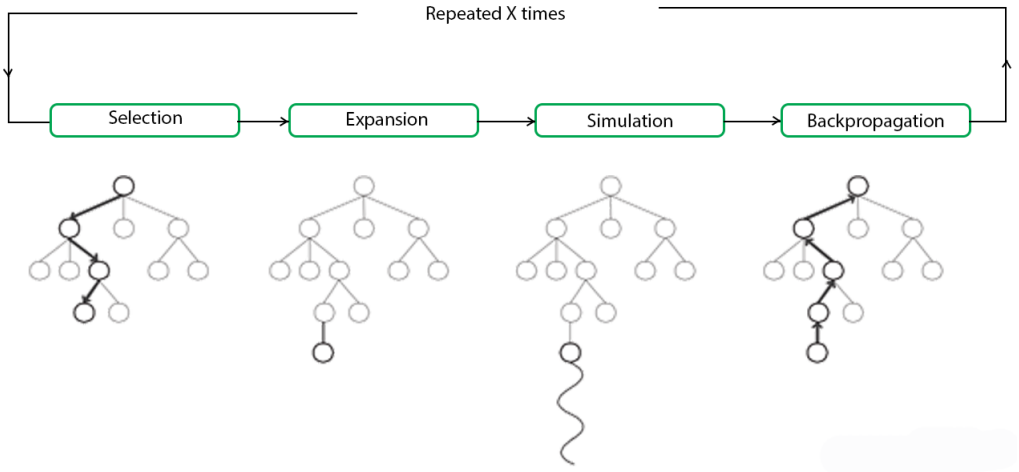


Figure 2.3: Monte Carlo Cycle diagram[6]

Chapter 3

Contributions

3.1 Parameter Descriptions

- W : A string data for sanitization.
- Z : The sanitized version of W consisting of a sequence of substring over the alphabet of W.
- k : The length of each pattern.
- $\mathcal{S}$: A set of sensitive patterns occurs in W.
- R : The output score of R-$score$ function.
- $\mathcal{L}$: A new string formed by deleting a single *letter* in Z.
- $\mathcal{G}$: The output string of GD-ALGO.
- $\mathcal{H}$: The output string of ELLS-ALGO or CSD-PLUS model.
- δ : The number of allowable deletions to reduce distortion.
- τ : The user-specified threshold to determine if a pattern is frequent.
- τ-ghost (*i.e.,* a type of spurious pattern) : The frequency of pattern is smaller than τ in W, but greater than or equal to τ in Z.
- τ-lost (*i.e.,* a type of spurious pattern) : The frequency of pattern is more significant than τ in W but smaller than or equal to τ in Z.
- ω : The weight for balancing the distortion reduction of the non-spurious pattern and spurious pattern. If $w = 0$, optimal delete only focus on the reduction of τ-ghost or τ-lost patterns. Distorting other non-sensitive patterns will not be penalized. If $w = 1$, optimal delete should minimize the distortion of the spurious pattern and the non-spurious pattern at the same time. Later experiments demonstrate how this parameter impacts the result of algorithms.

- max-simulations: The number of simulations required to select the best-deleted letter in ELLS-ALGO.
- C_p : The exploration parameter in the UCT formula used in ELLS-ALGO.
- E : The parameter in ELLS-ALGO to prune unnecessary nodes.

3.2 Problem

We define the LUL problem, which seeks to produce a string $\mathcal{H}$, by deleting a series of *letters* from the alphabet of Z so that: no sensitive pattern in $\mathcal{H}$; maximise the reduction of distortion. The first requirement is to meet privacy constraints. The second requirement is to improve the data utility of string data. By showing that the deletion function is not a monotone and submodular function, deleting one letter may indirectly affect the quality of other deletions. In addition, we show that the LUL problem is an NP-hard problem via a polynomial reduction of the *subset sum problem* to LUL.

3.3 Algorithms to solve the problem

3.3.1 *R-score* function

We design a function *R-score*, for measuring the goodness of each deletion. We prove that the time complexity of *R-score* is worst-case $\mathcal{O}(n + |X| + |\mathcal{S}|)$ time, where $|X|$ is the length of the affected substring before deletion, $|S|$ is the length of a sensitive pattern set. The output of the *R-score* is a float number that corresponds to the reduction of the distortion caused by the input deleted character to the sanitized string. The lower the R-score, the greater the distortion reduction. Due to the deletion property, each deleted letter can only affect at most $2k - 1$ substrings, where k is the length of the substring. The *R-score* function thus significantly reduces the search space by applying this property.

Example 1. Let W = abbabbbabbbbababba, sanitized string Z = bbabbbbabbbbbbabbbb, $k = 4$, $\tau = 2$ and set of sensitive patterns be {abab}, w = 0 (*i.e.,* the impact of the non-spurious pattern distortion is negligible). The index of deleted letter i is 3. The *R-score* $R = -1.0$. Then, constructing a new String $\mathcal{L}$ =bbabbbabbbbbabbbb by deleting letter at i. Note that, no sensitive pattern occurs in $\mathcal{L}$, while decreasing the frequency of τ-ghost pattern bbbb by one. Also, note that, neither new τ-ghost nor τ-lost occurs in $\mathcal{L}$ in W.

Example 2. Let W = abbabbbab, Z = abbaabbaab, $k = 4$, $\tau = 2$, and set of sensitive patterns be {babb}, $i = 4$, $w = 0$. The string after deletion $\mathcal{L}$ = abbabbaab. The *R-score* $R = \infty$, because $\mathcal{L}$ contains one sensitive pattern.

Example 3. Let W = abbabbbab, Z = abbaabbaab, $k = 3$, $\tau = 2$, and set of sensitive patterns be {babb}, $i = 5$, $w = 1$. After deletion, we construct a new string $\mathcal{L}$ = abbaabaab. Note that, the overall distortion of non-spurious patterns increases by 5, while no sensitive pattern occurs in $\mathcal{L}$. Hence, the output of *R-score* function $R = 5.0$.

3.3.2 GD-ALGO

To solve the LUL problem, we propose an algorithm GD-ALGO, for iteratively finding optimal deletion at current timestep in worst-case $\mathcal{O}(\delta|Z|\mathcal{T}(n, X, Y))$ time, where δ is the number of deletions, $|Z|$ is the length of Z. Also, we prove that GD-ALGO is linear scalable. The output of GD-ALGO is a string $\mathcal{G}$ consisting of a sequence of substrings over the alphabet of Z, and the total R-score of a series of deletions.

Example 4. Let $\delta = 2$, W = abbabbbabbbbababba, Z = bbabbbbabbbbbabbbb, $\mathcal{S}$ = {abba}, $\tau = 2$, $k = 4$, $w = 1$ (*i.e.,* Eliminating spurious patterns is equally important to minimise distortion of non-spurious patterns).

A string $\mathcal{G}$ = bbabbabbbbbabbbb is generated, by deleting letters at position 3 and 8. $R_{total} = -2$. Note that, no sensitive pattern occurs in $\mathcal{G}$, while decreasing the overall distortion by two. Also, note that, both deleted *letters* scored the lowest at time 0 and time 1 respectively.

3.3.3 ELLS-ALGO

Since GD-ALGO can only generate a greedy result via local search, we also devised a heuristic algorithm based on Monte Carlo Tree Search to avoid local minima. A critical feature of this algorithm is to balance exploration and exploitation by the UCT formula. Another good feature is that ELLS-ALGO is an *anytime* algorithm: it can make effective use of any number of iterations, and running for longer generally yields better decisions.

Example 5. (*Cont'd from Example 4*) Recall that W = abbabbbabbbbababba, Z = bbabbbbabbbbbabbbb, $k = 3$, $\tau = 3$, $S = \{baa\}$. Let $\delta = 4$. A string $\mathcal{H}$ = bbabbbabbbbabb is produced by deleting *letters* in $i \in [3, 8, 14, 15]$. A string $\mathcal{G}$ = is produced by deleting *letters* at position 3, 8, 9, and 15. Note that, score p of GD-ALGO result is -2.0, while p of $\mathcal{H}$ is -4.0. Also, $\mathcal{H}$ does not reinstate any sensitive patterns, while returning optimal distortion reduction. In addition, $\mathcal{G}$ prevents local-optimum by selecting the letter at position 14 rather than in 9 (*i.e.,* position with the local optimal p), when deleting the third letter.

$\mathcal{G}$: bbabbbbabbbbbabbbb $\rightarrow$ babbbabbbabbbb

$\mathcal{H}$: bbabbbbabbbbbabbbb $\rightarrow$ bbabbbabbbbabb

3.3.4 Optimized ELLS-ALGO

While ELLS-ALGO often performs well with a random playout policy, if the number of iterations is high, this method can consume computational resources and memory usage. Thus, we improve the efficiency of ELLS via some optimizations such as *pruning* (*i.e.,* a memory bounding technique) sim-

ulation information backup, which converts ELLS-ALGO into a *anyspace* algorithm. Finally, we show that the optimized ELLS-ALGO is much more effective than the naive ELLS-ALGO.

3.4 Experiments

To show that our algorithm can substantially reduce the distortion of non-sensitive patterns when processing large data sets, we designed a baseline algorithm and an exhaustive search algorithm to evaluate the performance of our model in terms of efficiency and effectiveness. By comparing our model to the baseline, we demonstrated that our model outperforms the baseline in any data size and for any k. In addition, we also demonstrate that both ELLS-ALGO and GD-ALGO are linearly scalable with data size and the number of deletions. Furthermore, by comparing their output to exhaustive search output, we show that both algorithms are very likely to find global optima results. Finally, we presented some experiments that analyzed the effect of various algorithm parameters on the output of our model.

Chapter 4

Problem Statements & Main Results

Problem Statements:

Problem 1 (LUL). *Given W, Z, δ, k, τ, and $\mathcal{S}$, construct a string $\mathcal{H}$ which minimising distortion by deleting δ letters and satisfies the following:*

1. No pattern $U \in \mathcal{S}$ occurs in $\mathcal{H}$.
2. $\sum_U d_{W,Z}(U)^2 < \sum_U d_{W,\mathcal{H}}(U)^2$, where $U \in \Sigma^k$ or $U \in \mathcal{P}_{W,\mathcal{Z}}$

LUL requires constructing a string X in which all sensitive patterns from W cannot be reinstated **(1)**. Simultaneously, in the string after deletions $\mathcal{H}$, both its total distortion and its spurious pattern distortion should be lower than in the undeleted version **(2)**. The non-spurious pattern may be a victim if distorting them can eliminate more spurious patterns.

Main Results:

Theorem 1 *The* $\text{LUL}_{\delta=1}$ *problem can be solved in worst-case* $\mathcal{O}(n + |X| + |\mathcal{S}|)$ *time.*

Theorem 2 *The* $\text{LUL}_{\delta\neq1}$ *problem is an NP-hard problem.*

Theorem 3 *Let W be a string of length n over* $\Sigma = \{1, ..., n^{\mathcal{O}(1)}\}$. *Given a string* Z *of length* l *over the alphabet of* W, $k < n, k < l$, $\mathcal{S}$ *and parameters* τ, δ. *GD-ALGO can find local optimal solution of Problem 1 in the worst-case* $\mathcal{O}(\delta|Z|\mathcal{T}(n, X, \mathcal{S}))$ *time, where* $\mathcal{T}(n, X, \mathcal{S})$ *is the running time of R-score function.*

Chapter 5

Algorithm Description

5.1 *R-score*

This section will cover more descriptions about how the *R-score* function works. In this project, we use the following formula to measure the (frequency) distortion of non-sensitive patterns [9],

$$\sum_{U}(Freq_W(U) - Freq_L(U))^2 \tag{5.1}$$

where W is original string data (*i.e.,* before sanitization), L is the *string* after sanitization or deletion, $U \in \Sigma^k$.

1. *Objective of R-score function*

   R-score function seeks to search for the best-deleted letter, which minimizes distortion.

2. *Deletion Strategy Analysis*

   Given the length of each pattern is k. Deleting a letter at index i from String X can delete at most k patterns and generate up to $k-1$ patterns. For example, let $X =$ abcde, all length-3 patterns are {abc, bcd, cde}. By deleting the letter c, a new string $Y =$ abde is produced. The set of all length-3 patterns in Y is {abd, bde}. It can be clearly observed that the frequency of abc, cde and bcd decreases, while the frequency of abd and bde increases. We can thus also find that deletion can affect up to $2k-1$ patterns.

   However, the deletion function is a double-edged sword. Ideally, we expect that all affected patterns are beneficial for reducing distortion without compromising the utility of string. In other words, they need to satisfy the following requirements:

Given *string* W and sanitized string Z and deleted letter x,

(a) If $Freq_W(U) < Freq_Z(U)$, where $U \in \Sigma^k$, then $U \in \mathcal{D}_Z(x)$, and $U \notin \mathcal{S}$. (Recall that $\mathcal{D}_Z(x)$ denotes patterns whose frequency decreases after deletion.)

(b) If $Freq_W(U) > Freq_Z(U)$, where $U \in \Sigma^k$, then $U \in \mathcal{J}_Z(x)$ and $U \notin \mathcal{S}$. (Recall that $\mathcal{J}_Z(x)$ denotes patterns whose frequency increases after deletion.)

(c) If $Freq_W(U) = Freq_Z(U)$, where $U \in \Sigma^k$, then $U \notin \mathcal{M}_Z(x)$ and $U \notin \mathcal{S}$.

In **(a)**, as the pattern's frequency in Z is small than its frequency in W, we expect that delete can decrease its frequency. In **(b)**, as the pattern's frequency in Z is greater than its frequency in W, the optimal deleted letter should lead to an increment in its frequency. In **(c)**, as the pattern's frequency is equal in W and Z, the best-deleted symbol should not modify its frequency.

However, in most cases, it is unlikely to search for such a letter. We thus propose a reward mechanism to evaluate the goodness of each legal letter. For example, if the *letters* meet the above criteria, they will be rewarded.Otherwise, they will be penalized. In addition, sometimes, we concentrate more on minimizing the distortion of τ-ghost or τ-lost pattern. Therefore, we also need to check if affected patterns are spurious or not (See Fig. 5.1).

The score of each affected pattern, defined as

$$P_U = \Big(d_{\mathcal{H},Z}(U) + d_{Z,W}(U)\Big)^2 - \Big(d_{Z,W}(U)\Big)^2 \tag{5.2}$$

where $U \in \Sigma^k$. This function seeks to calculate the reduction of distortion of a single pattern. The first part of the formula is defined as the distortion of such a pattern after deletion. The second part calculated the distortion before deletion. The function is useful regardless of whether U is generated pattern or not.

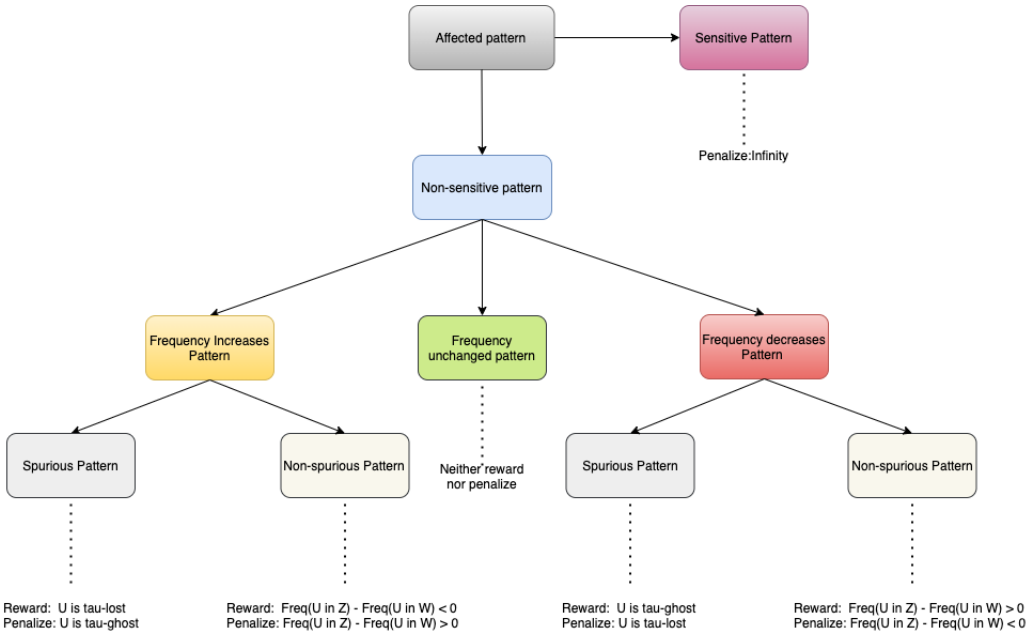


Figure 5.1: *R-score* reward mechanism

Given a string for deletion Z and a deleted letter x,

$$R(x) = \sum_{X} P_X + \omega \sum_{Y} P_Y + \sum_{U} Sens(U) \tag{5.3}$$

where $X \in \mathcal{M}_Z(x) \cap \mathcal{P}_{W,Z}$ is a spurious pattern in Z which distortion changes after deletion. $Y \in \mathcal{M}_Z(x)$, $Y \notin \mathcal{P}_{W,Z}$ and $U \in \mathcal{M}_Z(x)$. $Sens(U) = +\infty$ if and only if U is a sensitive pattern; otherwise, 0.

The function consists of three components. The first component measures the total score of spurious patterns, and the second part is to measure the score of non-spurious patterns. The weight ω is a float number in the range [0..1] used to control the weight of non-spurious pattern utility enhancement. The third part is to preserve privacy constraints. All deletions that can reinstate sensitive patterns will be filtered out. Fig. 5.2 shows how the *R-score* function calculates the total distortion reduction, total reduction is calculated as $(6-4)^2+(5-4)^2+(2-2)^2+(2-2)^2$.

Lemma 1. *Let W be a string of length n over Σ. Z be a sanitized string. Given $k < n$, deleted letter x, R-score function calculates the score of removing letter x such that the following hold:*

1. *The output is $+\infty$ as long as deletion generates sensitive pattern.* $(P1)$

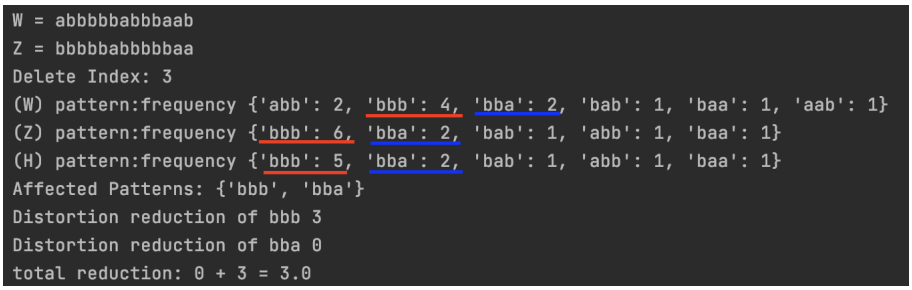


Figure 5.2: *R-score* calculation example

2. *When* $\omega = 1$ *and* $U \notin \mathcal{S}$, *R(x) is equal to the distortion reduction.* $(P2)$

Proof. **P1:** In Line 8, we first search a set of patterns $\mathcal{I}$ whose frequency changes and appears in the string after deletion. In addition, we only consider patterns that occur in the string after deletion rather than the original string, because we cannot guarantee that the original string does not contain a sensitive pattern. Subsequently, we check if the $\mathcal{I} \cap \mathcal{S}$ is an empty set $\emptyset$. Deletion is legal if and only if the intersection is $\emptyset$. Otherwise, it will be penalized $+\infty$. **P2:** When $\omega = 1$, $U \notin \mathcal{S}$, *R-score* function can be simplified into $\sum_X P_X + \sum_Y P_Y$. By substituting function P, $R(x) = \sum_X \Big(d_{\mathcal{H},Z}(X) + d_{Z,W}(X)\Big)^2 - \sum_Y \Big(d_{Z,W}(Y)\Big)^2$. Also, when $\omega = 0$, the *R-score* function treats spurious patterns and non-spurious patterns equally
We thus finally obtain the relationship:

$$\begin{aligned} R(x) &= \sum_U \Big(d_{\mathcal{H},Z}(U) + d_{Z,W}(U)\Big)^2 - \Big(d_{Z,W}(U)\Big)^2 \\ &= \sum_U \Big(Freq_{\mathcal{H}}(U) - Freq_Z(U) + Freq_Z(U) - Freq_W(U)\Big)^2 - \Big(d_{Z,W}(U)\Big)^2 \\ &= \sum_U \Big(Freq_{\mathcal{H}}(U) - Freq_W(U)\Big)^2 - \Big(Freq_Z(U) - Freq_W(U)\Big)^2 \end{aligned} \tag{5.4}$$

For example, let $W = aba$, $Z = bab$, given delete index x = 2, k =1, we have $\mathcal{M}_Z(x) = \{a\}$. So we construct a new string $\mathcal{H}$ after deletions and $Freq_{\mathcal{H}}(a) = 1$. In addition, we already know $Freq_Z(a) = 1, Freq_W(a) = 2$ from W and Z. We thus can get $R_{k=1}(x) = ((1-2)+(2-1))^2 - 1^2 = (1-1)^2 - (2-1)^2 = -1$. □

Theorem 1 *The* $LUL_{\delta=1}$ *problem can be solved in worst-case* $\mathcal{O}(n + |X| + |S|)$ *time.*

Proof. As $\delta = 1$, the *R-score* function is only called once. Furthermore, from the

Algorithm 1: *R-score*

input : $k, W, Z, \mathcal{S}, \tau, \omega, i$;
output: *R-score*;

1 **if** $i - k < 0$ **then**
2 **sub-before** $\leftarrow Z[0..i+k]$; `// The longest string in Z can change pattern frequency. (Reduce Search Space)`
3 **sub-after** $\leftarrow Z[0..i] + Z[i+1..i+k]$; `// The longest string in deleted string can change pattern frequency.`
4 **else**
5 **sub-before** $\leftarrow Z[i-k+1..i+k]$;
6 **sub-after** $\leftarrow Z[i-k+1..i] + Z[i+1..i+k]$;
7 **sp-shift** $\leftarrow 0.0$; **nsp-shift**$\leftarrow 0.0$;
`// Scores for spurious affected patterns and non-spurious affected patterns`
8 **if** *SearchNsens(sub-after,k)* $\cap\, S \neq \emptyset$ **then**
9 **Report** ∞**;** `// Penalize infinity`
10 $\mathcal{M} \leftarrow SearchNsens($**sub-before**$, k) \cap SearchNsens($**sub-after**$, k)$;
`// All length-k patterns affected by deletion`
11 **foreach** U *in* $\mathcal{M}$ **do**
12 **shift** $\leftarrow Freq(U,$**sub-before**$) - Freq(U,$**sub-after**$)$;
13 **distZtoW** $\leftarrow Freq(U, Z) - Freq(U, W)$;
14 $P \leftarrow$**(distZtoW + shift)**^{2} **+distZtoW**^{2};
15 **if** $Freq(U,W) < \tau \leq Freq(U, Z) \vee Freq(U,Z) < \tau \leq Freq(U, W)$ **then**
16 **sp-shift** $\leftarrow$ **sp-shift +** P; `// Add to spurious pattern score`
17 **else**
18 **nsp-shift** $\leftarrow$ **nsp-shift +** P;`// Add to non-spurious pattern score`
19 ***R-score*** $\leftarrow$**sp-shift** $+ \omega \times$ **nsp-shift**;
20 **Report *R-score*;**

pseudocode, we can clearly observe no nested loops in the *R-score* function. By applying data structure, finding the intersection of two sets in Line 8 can effectively be solved in $\mathcal{O}(|X| + |S|)$, where $|X|$ is the length of the affected substring in $\mathcal{H}$. Line 10 loops through $|X| - k$ times to search for all non-sensitive patterns in X. Also, it is executed $|X| - k + 1$ times to find all non-sensitive patterns in the affected substring in Z, whose length = $|X| + 1$ (*i.e.,* add one letter to X). We use the same variable to represent because two strings are dependent. Therefore, Line 10 can be realized in $\mathcal{O}(2|X| - 2k + 1) = \mathcal{O}(|X|)$ time (*i.e.,* k is constant). The for loop in line 11 is executed n times, where n is the length of total affected patterns. So it runs in $\mathcal{O}(n)$. In addition, each operation inside the loop costs $\mathcal{O}(1)$. Function Freq() also a basic operation, if we store all non-sensitive patterns beforehand (*i.e.,* before entering the for each loop) into a dictionary. Therefore, LUL $_{\delta=1}$ problem is solvable in worst-case $\mathcal{O}(n + |X| + |S|)$ time. □

5.2 GD-ALGO

Lemma 1 states that the *R-score* function can accurately calculate the reduction of distortion. Furthermore, all deletions that can reinstate sensitive patterns will result in a $+\infty$ score. A lower *R-score* will result from better deletion. GD-ALGO is proposed for finding the best-δ *letters* to delete from Z locally, where $\delta > 1$. The algorithm iteratively finds the current best candidate *letter* to delete until δ *letters* have been searched or a deletion that can avoid the generation of sensitive patterns cannot be found. The most straightforward method is to search for the best-δ *letters* all at once. However, by proving the following theorems, we show that it is not achievable.

Definition 5.2.1 (Monotone[1]). *A set function $f : 2^S \to \mathbb{R}$ is monotone if $A \subset B \subset S$ implies $f(A) \leq f(B)$.*

Definition 5.2.2 (Submodular[2]). *A set function $f : 2^S \to \mathbb{R}$ is submodular if $\forall A \subset B \subset S$ and $s \in S \setminus B$, we have $f(B + s) - f(B) \leq f(A + s) - f(A)$.*

Lemma 2. *Let f: $X \to \mathbb{R}$ be the function from $f(X) = \sum_{x \in X} R(x)$, where X is a set of deleted letters, lower R(x) is more preferable.*

1. *f(X) is not monotonic.*
2. *f(X) is not submodular.*

Proof. Let X be a set of deleted *letters*, x be a new letter to delete. **(1)** If $f(X)$ function is not a monotone function, $f(X \cup u) \geq f(X)$ for any X and u should not hold. We thus introduce a counterexample to prove it. Given W = abb, Z= ababa, $k = 1$, $\tau = 2$ and $\omega = 1$ (i,e, consider all affected patterns). In addition, $X = \{a_2\}, u = a_3$. We thus have $f(\{a_2\} \cup a_3) < f(\{a_2\})$, because $(-3.0 + (-1.0)) < -3.0$. Constructing a new string $\mathcal{H} = Z =$ abb. Deleting two *letters* clearly have better performance. Therefore, $f(X)$ is not monotonic. **(2)** Proving f(x) is not submodular, we should have an example $f(X \cup u) - f(X) < f(Y \cup u) - f(Y)$, where $Y \subset X$. Let $W =$ abbabaa, Z = babbabba, two sets of indices

[1] https://en.wikipedia.org/wiki/Monotonic_function
[2] https://en.wikipedia.org/wiki/Submodular_set_function

that correspond to deleted *letters* $X = \{0, 1, 2\}$, $Y = \{0, 1\}$, new deleted index $u = 3$ and parameters $k = 3$. The counterexample in Fig.5.3 shows that the total distortion of $f(X \cup u) - f(X) = -1$ and $f(Y \cup u) - f(Y) = 1$. Hence, $f(X)$ is not submodular.

```
X:{0,1,2}
Y:{0,1}
u:3
f(X) = 2 f(Y) = 3
f(X union u) = 3 f(Y union u) = 2
f(X union u) - f(x) = 1 f(Y union u) - f(Y) = -1
```

Figure 5.3: Counterexample to prove *R-score* function is not submodular

□

According to the proof of Lemma 2, the function used to solve the LUL problem is neither monotonic nor submodular. Hence, extending the range of deleted *letters* does not always result in better results. Furthermore, it is difficult to predict and estimate a letter's potential *R-score* (*i.e.,* the goodness of deletion). This is because removing one letter may indirectly influence other *letters*' *R-score* in the next step.

Theorem 2 *The* LUL $_{\delta \neq 1}$ *problem is an NP-hard problem.*

To prove the LUL problem is NP-Hard, we will have to reduce a known NP-Hard problem to this problem. We will carry out a reduction from the *Subset Sum Problem*(SSP) to the LUL problem.

- LUL problem:

  Instance: A set of R scores, corresponding to each possible deleted letter $\mathcal{R} = \{r_1, r_2, r_3 ... r_n\}$, θ. Delete each *letter* has a cost 1, thus we have costs $\{c_1, c_2, c_3\}$, and a bound δ.

  Questions: Is there a subset of *R-score*s with total *R-score* at most θ, such that the corresponding cost is equal to δ?

- Subset Sum Problem:

  Instance: A set of non-negative integer numbers $S = \{s_1, s_2, s_3 ... s_n\}$, and target value t.

  Question: Is there any subset $T \subseteq S$ such that $\sum_{i \in T} s_i = t$?

Given an input set, checking if the total *R-score* is at most θ and if the corresponding cost is exactly δ can be solved in linear time.

To reduce an instance of Subset Sum problem to an instance of Knapsack problem, we create such an LUL problem that:

$$\begin{cases} s_i = c_i = r_i, \\ \theta = \delta = t. \end{cases} \tag{5.5}$$

Then, for any subset $T \subseteq S$

$$\sum_{i \in T} s_i = t \Longleftrightarrow \sum_{i \in T} r_i = \sum_{i \in T} s_i \leq t \wedge \sum_{i \in T} c_i = \sum_{i \in T} s_i = t \tag{5.6}$$

If we have a solution to the right part of the equation, we will have a solution to the left part. Hence, the LUL problem is an NP-hard problem.

Algorithm 2: GD-ALGO

```
   input : k, W, Z, S, τ, ω;
   output: Tour, Total-R;
1  Tour ← ∅;
2  Total-R ← 0;
3  for i ← 0 to δ do
4      B ← ∞;
5      foreach i in Z do
6          r ← R-score(k, W, Z, S, τ, ω, i);
7          if r < B then
8              B ← r;
9      if B == ∞ ∨ (i == δ ∧ B > 0) then
10         break; // Best letter can reinstate sens or last deletion
                  can harm utility.
11     Total-R ← Total-R + B;
12     Tour ← Z[0..i] + Z[i + 1..];
13 Report Tour,Total-R;
```

Theorem 3 *Let W be a string of length n over $\Sigma = \{1, ..., n^{\mathcal{O}(1)}\}$. Given a string Z of length l over the alphabet of W, $k < n, k < l$, $\mathcal{S}$ and parameters τ, δ, GD-ALGO finds local optimal solution of Problem 1 in the worst-case $\mathcal{O}(\delta|Z|\mathcal{T}(n, X, \mathcal{S}))$ time, where $\mathcal{T}(n, X, \mathcal{S})$ is the running time of R-$score$ function.*

Proof. GD-ALGO is a standard greedy local search algorithm. The pseudocode clearly shows that there is only one nested loop in lines 3-10. The outer loop at line

3 is executed δ. The inner loop runs $|Z|$, where $|Z|$ is the sanitised string's length. All operations except line 6 are basic $\mathcal{O}(1)$ operations. The running time in line 6 is precisely the running time of the *R-score*. Therefore, it clearly illustrates that the local solution of problem 1 can be searched by GD-ALGO in $\mathcal{O}(\delta|Z|\mathcal{T}(n, X, \mathcal{S}))$ time. □

5.3 ELLS-ALGO

GD-ALGO has proven to be able to find local optima of LUL problem. However, the greedy selection of the best-deleted letter with the lowest *R-score* will potentially "blind" the algorithm. Therefore, sometimes we may sacrifice short-term gains for long-term gains. Monte Carlo Tree Search (MCTS) is a type of random-based search algorithm to search for sub-optimal results without any problem-specific knowledge. We thus propose ELLS-ALGO, which exploits some features of the MCTS algorithm to escape from local optima. ELLS-ALGO can keep looking for other "good" deleted *letters* whilst validating how good the current best deletion is. We have provided a detailed description of ELLS-ALGO below.

Definition 5.3.1 (Tree Data Structure)**.** *A tree data structure (See in Fig. 5.4) can be defined recursively as a collection of nodes. Each node holds a value or piece of information, and it may or may not have a child node. The topmost node in a tree is called the root node. A leaf is a node that does not have a child node in the tree. Path refers to the sequence of nodes along the edges of a tree.*

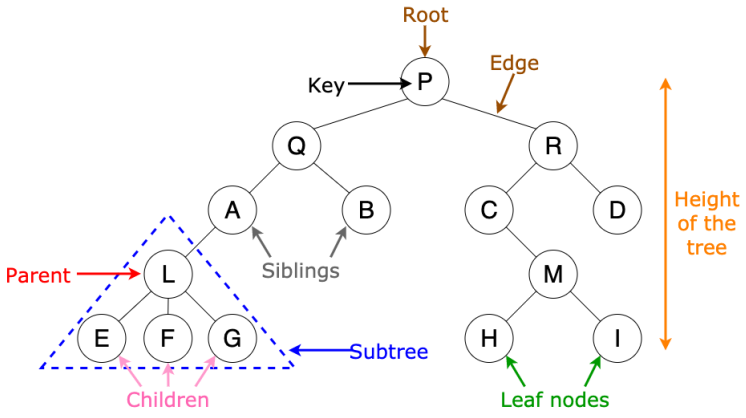


Figure 5.4: Tree structure diagram [3]

Definition 5.3.2 (Terminal State)**.** *Indicate the state that all delete selections have been completed.*

Definition 5.3.3 (Playout)**.** *The process of randomly deleting letter from a given start string until it reaches the terminal state.*

Definition 5.3.4 (Branching Factor)**.** *The branching factor is the number of child nodes generated by a given node. A tree has a high branching factor while each node in the tree has a large number of children.*

5.3.1 Mapping between ELLS-ALGO and LUL Problem

The primary goal of the LUL problem is to construct a string to maximize distortion reduction by deleting δ *letters*. Therefore, choosing deleted *letters* can be viewed as a sequential decision-making process. We thus can find the solution to the LUL problem by constructing a decision tree with height δ. Every possible string is stored in a node of the tree. Whenever we delete a letter, we branch the tree and generate a new node. The length of all paths in the tree should be δ (*i.e.,* the number of deletions). The solution to the LUL problem is the path that generates the lowest total *R-score*. However, because of the large number of possible deletions, a full decision tree cannot be efficiently generated to find the best-deleted *letters*. For instance, we have a sanitized string Z. If $\delta = |Z|$ (the length of the sanitized string), we will build at most $|Z|!$ ($|Z| \times |Z| - 1.. \times 1$) nodes to fully expand the trees. ELLS-ALGO is a type of tree search technique that favors more promising nodes resulting in an asymmetric tree over time. In other words, the partial tree is biased toward regions that are more promising and thus more important. This feature can significantly reduce the size of the tree. After fully constructing the tree, we select the value stored in the node that maximizes the reward as the resulting string (*i.e.,* The string after removing the best candidate *letter*). Simultaneously, this node will become the root node for searching for the next deleted letter. The algorithm will terminate when the value of the root node with the length of $|Z| - \delta$ (*i.e.,* complete all deletions). Therefore, we can use ELLS-ALGO to address the LUL problem.

5.3.2 ELLS-Node Structure

ELLS-Node is used to store all the necessary information for ELLS-ALGO to perform deleted letter selection.

- *state* : The *string* data of node. The state of root node is Z.
- *move* : An action (*i.e.,* deletion) in which the parent node generates child nodes. More precisely, each node can use "move" to position specific child node.
- *score* : The *R-score* of getting the state of node.
- *isExpanded* : The boolean expression that defines whether or not the tree can be extended at this node.
- *parent* : The parent node of the node. Defining the parent node can improve the efficiency of looping backward in the tree. The root node does not have a parent node.
- *visits* : The total number of times the node has been accessed.
- *reward* : The cumulative rewards of node. Each reward is created as a result of a simulation step (Negative total *R-score* of a random deletion sequence). The node with a higher reward is more promising.
- *children* : A dictionary for storing all child nodes and their corresponding moves. The node is a leaf node if the dictionary is empty.

5.3.3 Selection

To select the best node in the tree to perform simulations, the key challenge is to strike a balance between exploiting the current most promising node and exploring alternate nodes that could provide better long-term gain. In other words, if we overuse the current best node, our model will have a fast convergence but will miss the global optimala solution.Alternatively, if we explore nodes (*i.e.,* possible deletions) as much as possible, our model will eventually require an extremely

expensive computational resource to converge. Therefore, this problem represents a tradeoff between speed and accuracy.

To deal with this problem, we utilize the feature of the UCT formula to balance the dilemma of exploration *versus* exploitation. Each time a node i is selected in ELLS, its choice maximizes the following formula:

$$i = \underset{i \in children(I)}{\arg\max} \frac{R(i)}{v(i)} + c_p \sqrt{\frac{\ln v(I)}{v(i)}} \tag{5.7}$$

where $R(i)$ is the reward of node i, and $v(i)$ is visits of node i, $children(I)$ represents all child nodes in node I, c_p is a constant.

5.3.4 Expansion

After reaching the expandable node, we create n child nodes where n is the number of possible deletions, to expand the tree. The expanded node will no longer be expandable and store all child nodes by constructing a dictionary consisting of (x: child) pairs, where x is the deleted letter. For example, given an expanded node is $node(ab)$, we will generate a dictionary $\{(b : node(a)), (a : node(b))\}$.

5.3.5 Simulation

In this step, we randomly select the child nodes of the expand node as the start node C to perform a random playout (See in Fig 5.5). During the playout, The characters that incur any sensitive patterns will not be allowed to be added into the playout, because this will cause our final simulation result to be $+\infty$. No matter how good the simulation results of the *letters* in the simulation are in the future, their overall reward will be unchanged. Therefore, this will undoubtedly interfere with the algorithm's selection of the most promising nodes in the future.

5.3.6 Backpropagation

Following the simulation phase, we should tell the algorithm how good the current path is. Therefore, we traverse the tree backward, appending the simulation reward to each simulation node's "reward" attribute and increasing each simulation node's

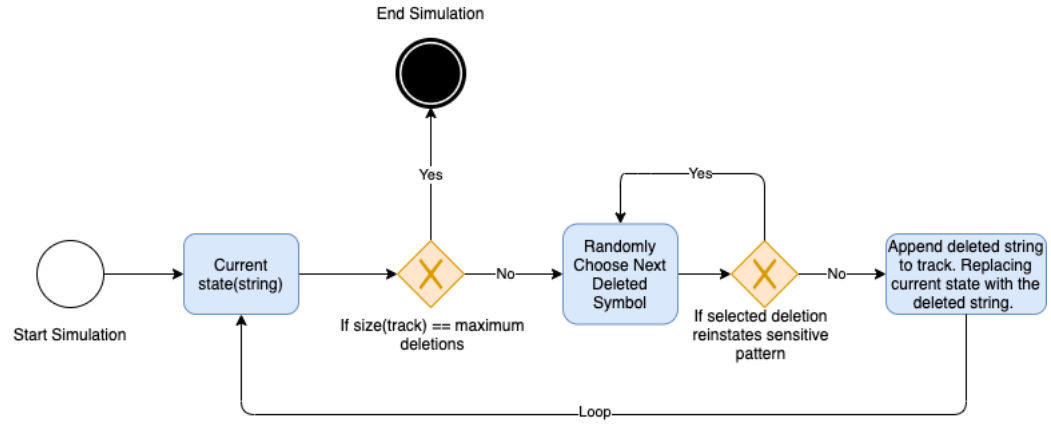


Figure 5.5: Simulation process flowchart

visit count by one until reaching the root node.

Note: Section 5.3.3 - 5.3.6 is visualized in Fig. 5.6.

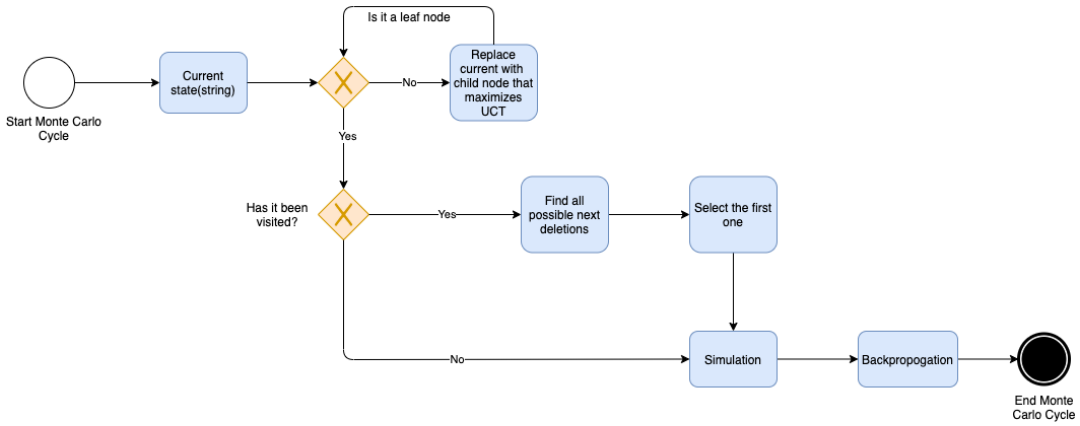


Figure 5.6: Monte Carlo Iteration Flowchart

5.3.7 Tuning Exploration Parameter

In section 5.3.3, we exploits a exploration parameter C_p to balance exploration and exploitation. In this section, we will discuss more about the features of C_p. The value of C_p is strongly associated with the number of iterations. Setting a too high C_p when the number of iterations is low would make the result difficult to converge. Setting a low C_p for a large number of iterations, on the other hand, runs the risk of dropping into the local optimum.

5.3.8 Computational Budget

If we have an infinite amount of computing power, we can iteratively run steps 1-4 (See section 5.3.3 - 5.3.6) to enhance the quality of the result. However, we must set up a limited computational budget for real-world applications. We thus use a parameter θ to limit the number of iterations. When computational budge is met, ELLS-ALGO terminates the searching process and returns the best deletions associated with the root node. One advantage of ELLS is that it can stop the growth of trees at any time.

5.3.9 Determine Deleted letter

After ELLS-ALGO meets termination criteria (See section 5.3.8), we will select the best-deleted letter by applying one of three criteria in [3].

- *Max-Child:* Select child node at the root with the highest reward.

$$i = \underset{i \in C(root)}{\arg\max}\, R(i) \tag{5.8}$$

  where $C(root)$ is the set of children of the root node.

- *Robust-Child:* Select child node at the root with the highest visits count.

$$i = \underset{i \in C(root)}{\arg\max}\, v(i) \tag{5.9}$$

- *Max-Robust-Child:* Select the child node at the root node with the highest sum of visits and rewards.

$$i = \underset{i \in C(root)}{\arg\max}(R(i) + v(i)) \tag{5.10}$$

We subsequently using the selected node as the root node of the next decision tree, if the length of the node's state is greater than $|Z| - \delta$ (*i.e.,* all deleted *letters* are found).

5.3.10 Efficiency and Memory Optimization

Unfortunately, the basic ELLS-ALGO performs relatively poorly when dealing with the huge dataset. This is because we usually necessitate a large number of simulations to gather an accurate result. Whether the strategy that UCT identifies is reliable heavily depends on the number of visits.

An unbounded increase in simulations will lead to expensive memory usage and computational resources. Each ELLS-ALGO operation is fast on its own. However, when we run thousands of Python operations, such as calculating *R-score*, looping the monte Carlo tree, the whole thing becomes inevitably slow. We thus exploit the following strategies to optimize naive ELLS-ALGO.

- *Pruning Search Tree*

  In order to reduce memory usage, we implement a simple pruning filter to expand the nodes if and only if, whose *R-score* is within a certain range. We first set the minimum *R-score* $min(R)$ as the lower bound of the set. The range of the set is a user-specified parameter E. Therefore, we only consider the deletions whose *R-score* within the range $[min(R), min(R) + E]$. Also, if we consider deleting a letter at index i, and $X[i] = X[i+1]/X[i-1]$, we can say this deleted letter is useless because deleting such a letter or its neighbors will lead to the exact same result. For example, $X = abba$, deleting first b and second b will generate the same new string aba, which leads to the same distortion. These two approaches can effectively limit the growth of search space while improving the quality of the result, because in most cases if the current deletion results in a very poor *R-score*, the overall *R-score* is also likely to be poor. Especially for small k and δ, deletion will not have a significant impact.

- *Expansion Delay*

  We also limit the growth rate of the tree by avoiding excessive expansion. Instead of expanding the tree every time we reach the leaf node, we expand the tree if and only if it has been sampled k times. This method is initially proposed by Rémi Coulom in 2007 [5], and it is very effective at reducing

memory usage.

- *Backup Simulation Data*

  Naive ELLS-ALGO has to gather all non-sensitive patterns and calculate their *R-score* for each simulation and expansion step. Even though all operations can be completed in linear time (See in Theorem 1), the unbounded simulation will also lead to expensive computational costs and memory usage. To deal with this task, we exploit a dictionary (*i.e.,* A collection to store key: value pair) to store all simulation-related data, such as *R-score*s and deleted *letters*. As a result, for n simulations, all operations only need to be performed once. For example, current string X = abbab, we aim to calculate the *R-score* to delete the second a. If this simulation has already been performed in previous simulations, we can retrieve its data from the global dictionary immediately. At the same time, we may delete keys in the dictionary that have not been used for a long time when our algorithm meets the memory limit.

- *Search Tree Reuse*

  Naive ELLS-ALGO must discard the old tree and reconstruct a new tree by selecting a single deleted letter. This method is very stubborn if we require to delete many *letters*. Suppose that ELLS built up a search tree in a previous letter selection $t-1 \geq 0$, and we finally decided to delete letter x_{t-1}. The entire subtree rooted in the node corresponding to that best-deleted letter can still be considered relevant for the new search process in the current letter selection t. Hence, instead of initializing ELLS-ALGO by only a root node, it can be initialized with a part of the tree built in the previous decision-making process (See in Fig.5.7). This method can significantly enhance the efficiency of the program. For example, given max-simulations Max, the number of iterations of the next round is $Max - \sum_{i \in C(root)} v(i)$, where $C(root)$ denotes the children of the root node.

- *NumPy and Cython*

  We also improve the efficiency of the algorithm by utilizing the NumPy and cython libraries. This is because Python was not originally intended for nu-

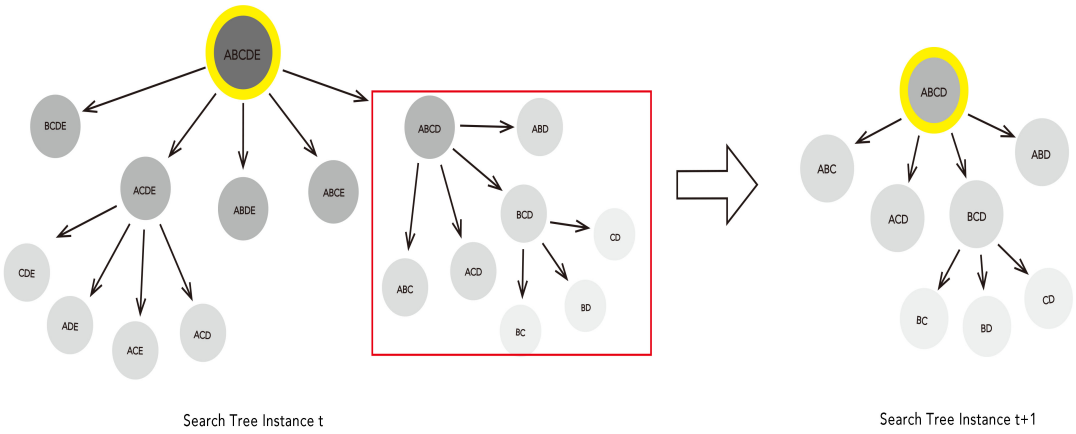
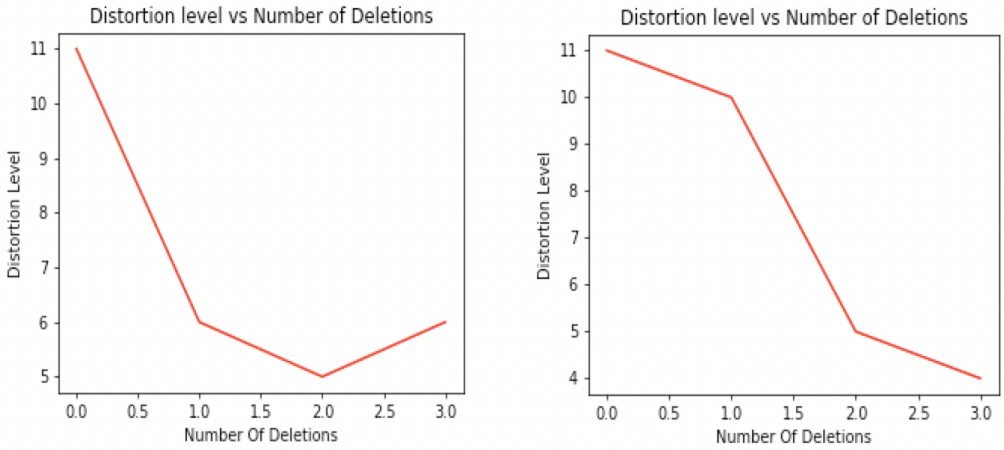


Figure 5.7: Search tree reuse example

merical computation. The Numpy operation is more compact and can be effectively mapped to highly optimized C code. We converted our Python code to C code using Cython. Unlike Python, Cython code is closer to machine code and does not require line-by-line interpretation.

5.3.11 Correctness of Algorithm

Example 6. Let W = abbaaababaa, Z = ababbabbaa, $\delta = 3$, $k = 4$, $\tau = \infty$, Given UCT parameters $C_p = 150$. $\mathcal{S} = \{aaab\}$. See the results of GD-ALGO and ELLS-ALGO in Fig.5.8. When performing the first deletion, GD-ALGO selects the deleted letter with the lowest *R-score* = -5, while ELLS-ALGO selects a letter with *R-score* = -1, but with better long-term benefits.



(a) GD-ALGO (b) ELLS-ALGO

Figure 5.8: Deleted letter selection process

5.4 Tkinter Graphic Interface

To compare the output of the two algorithms and discover intuitively how different parameters influence the results. We created a simple python-based GUI to visualize the entire deleted *letter* selections operation. This application uses native Tkinter libraries and provides useful graphical feedback when tuning parameters. Also, users can run some basic tests on a small dataset to verify the correctness and performance of our algorithms. (See in Fig. 5.9) The detailed user guide will be covered in the appendix.

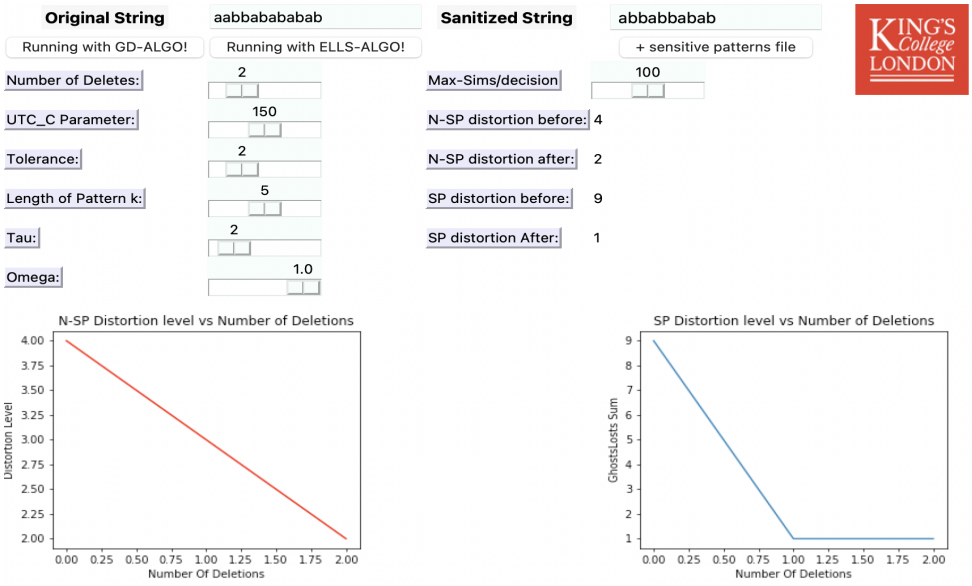


Figure 5.9: Main control window of Python GUI

5.5 Command-line Interface

We also designed a text-based user interface for allowing users to reproduce experimental results or apply it to broader data sets. The user can run the project by providing the file name of the data set and appropriate parameters. If the user runs command "python runner.py" without any suffix, the default test case will be triggered automatically. The detailed user guide will be covered in the appendix.

```
Usage:
    USAGE:      python runner.py <options>
    EXAMPLES:   (1) python runner.py
                    - starts deletion strategy test case


Options:
  -h, --help            show this help message and exit
  -w W_FILENAME, --originalFile=W_FILENAME
                        The string for sanitization(W) [Default:
                        test/test_w.txt]
  -z Z_FILENAME, --sanitizedFile=Z_FILENAME
                        The string for sanitization(Z) [Default:
                        test/test_z.txt]
  -t TAU, --tau=TAU     The tau value to identify spurious pattern [Default:
                        1]
  -o OMEGA, --omega=OMEGA
                        The weight of non-spurious pattern [Default: 1]
  -s SENSITIVE_PAT, --sensitivePatterns=SENSITIVE_PAT
                        A file that consists of all sensitive patterns in W
                        [Default: test/sen_pattern_test.txt]
  -k K                  The length of each pattern [Default: 4]
  -c C                  The exploration parameter for UCB1 formula [Default:
                        200]
  -d DELTA, --delta=DELTA
                        The number of deletions [Default: 5]
  -e TOLERANCE, --E=TOLERANCE
                        The pruning parameter for ELLS-ALGO [Default: 100]
  -m MAX_SIMULATIONS, --max=MAX_SIMULATIONS
                        The number of iterations per selection in ELLS-ALGO
                        [Default: 3]
```

Figure 5.10: Command-line Interface Application

Chapter 6

Professional Issues

6.1 British Computing Society Code of Conduct

In the United Kingdom, the British Computer Society (BCS)[1] offers sets of guidelines, ethical practice to consider when computing. Any project should follow a code of ethics, and failing to do so can have significant legal and ethical implications. Additionally, using resources or products without authorization would also be considered unethical. Throughout the development of this project, the British Computing Society's rules and professional issues were followed. The majority of the code in this project is my own work. For the exhaustive search algorithm, we used some Open-Source code from stack-overflow to speed up our algorithm that has been explicitly mentioned as commenting on the code.

[1] https://www.bcs.org/

Chapter 7

Experimental Evaluation

In this chapter, we evaluate our algorithms in terms of effectiveness and efficiency. The effectiveness is measured based on the distortion reduction of spurious patterns and non-spurious patterns. Efficiency is measured based on the running time of our algorithms. Also, we perform some experiments to demonstrate how different parameters influence the final result.

7.1 Evaluated Algorithms

Our algorithms are performed in the following order: GD-ALGO $\rightarrow$ ELLS-ALGO. The combination of the two algorithms is also referred to as CSD-PLUS. We firstly utilize the output of GD-ALGO as the lower bound of our outcome. Subsequently, we run ELLS-ALGO to boost our results on this basis. The final output is a string $\mathcal{H}$ with the length $|Z| - \delta$.

First, we compared CSD-PLUS against a baseline algorithm, in terms of effectiveness. We aim to demonstrate that the CSD-PLUS model is a robust and powerful algorithm that outperforms the baseline for any data size and any pattern length k.

Second, in terms of efficiency, we compared ELLS-ALGO to naive-ELLS-ALGO. Naive-ELLS does not support search tree reuse and simulation information backup mechanisms. As a result, we plan to demonstrate that such optimizations will actually speed up the algorithm.

Lastly, we designed two experiments to present how various parameter and selection policies (See section 5.3.9) affect the effectiveness of our algorithms. Also, we consider comparing the performance of GD-ALGO and ELLS-ALGO with *exhaustive search* (*i.e.,* a very general problem-solving method to find the global op-

timum solution by considering all possible permutations) to demonstrate that both algorithms can find the global optimum in many situations. In addition, ELLS-ALGO outperforms GD-ALGO in some challenging cases.

7.1.1 Description of baseline

Baseline algorithm (See in pseudocode 3), referred to as **BA**, like our algorithm principle, seeks to improve data utility by deleting redundant *letters*. To initialize the algorithm, it requires original data W, sanitized data Z, $|\mathcal{S}|$, δ, and the length of pattern k. To enhance the reliability of our experimental result, the baseline should also select deleted *letters* by following some rules rather than randomly delete. For each deletion operation, the algorithm initially sorts all patterns by their distortions. Subsequently, it traverses the letters from the most distorted pattern until finding a 'victim' letter that does not incur any sensitive patterns. Such 'victim' is the deleted letter. Unlike the CSD-PLUS model, the baseline algorithm cannot distinguish spurious and non-spurious patterns. Hence, it may be poor to balance the distortion of spurious patterns and non-spurious patterns. Furthermore, it is unable to evaluate the effect of each deletion on a particular pattern.

7.2 Experimental Data

We conduct our experiment using the publicly accessible datasets used in [10, 17]: trucks (TRU), DNA substrings (DNA), and synthetic data (uniformly random strings referred to as SYN). The table below demonstrates the properties of these datasets in experiments:

7.3 Experimental Setup

For each dataset, we used τ mentioned in Table 7.1 to determine if a pattern is frequent. All sensitive patterns should be with length-k and randomly selected from the maxmotif mining algorithm's output in [13, 14]. Moreover, if we want to carry out our experiments, we need both the original and sanitized datasets. To get sani-

Algorithm 3: Baseline-ALGO

input : $k, W, Z, \mathcal{S}$;
output: Z;

```
1  W-nsens ← SearchNsens(W, k);
   // Search for all non-sensitive patterns in W
2  Z-nsens ← SearchNsens(Z, k);
   // Search for all non-sensitive patterns in Z
3  for i ← 0 to δ do
4      candidates ← ∅;
5      for j ← 0 to |Z| − k + 1 do
6          distZtoW ← Freq(U,Z-nsens) - Freq(U,W-nsens);
7          candidates ← (distZtoW, [j, j + k]);
8      end
9      Sort(candidates) // Sort the candidates in reverse order by
           their first column values
10     foreach candid in candidates do
11         foreach victim in candid do
12             if Sens(victim,S) then
                   // check if removing 'victim' reinstates any
                      sensitive patterns
13                 Z ← delete(Z,victim);
14                 break;
15             end
16         end
17     end
18 end
19 Report Z;
```

Dataset	Data domain	Length n	Alphabet size Σ	Sensitive patterns	Total number |S|	Delete δ	pattern length k
TRU	Transportation	5763	100	[20,80]	[612,2148]	20	[3,6]
DNA	Genomic	10000	4	[20,80]	[568,2484]	20	[2,5]
SYN	Synthetic	1000	5	[5,20]	[49,148]	[0,50]	[2,5]
SYN_{Big}	Synthetic	50000	5	5	342	[2,10]	[5,25]

Table 7.1: Characteristics of dataset

tized data, we first input all the data into TFS-ALGO and PFS-ALGO (See section 2.4.2), following replace all separator *letters* # with *letters* chosen at random from Σ that will not reinstate sensitive patterns. When comparing CSD-PLUS *versus* BA, we used $\omega = 0.5$ in all experiments since we want to concentrate more on eliminating spurious patterns. We also allow both baseline and CSD-PLUS to delete the same number of letters, ensuring a fair comparison. In terms of efficiency evaluation, we aim to preserve all nodes rather than applying the pruning technique in order to guarantee both algorithms are not interfered with by the number of pruned nodes.

To capture the improvement of data utility, we used the distortion reduction

measure:

$$\sum_{U}(Freq_W(U) - Freq_Z(U))^2 - \sum_{U}(Freq_W(U) - Freq_{\mathcal{H}}(U))^2 \tag{7.1}$$

where $U \in \Sigma^k$ is a non-sensitive substring, W is the original string, Z is sanitized string, and $\mathcal{H}$ is the string after deleting. More precisely, we measured the distortion reduction of both spurious patterns and non-spurious patterns. A pattern U is spurious if $Freq_W(U) < \tau < Freq_{Z/H}(U)$ or $Freq_{Z/H}(U) < \tau < Freq_W(U)$. A large number of spurious patterns may significantly affect the accuracy of frequent pattern mining. However, this does not mean that we should omit the impact of non-spurious trends. During deletions, neither the BA algorithm nor CSD-PLUS is permitted to reinstate any sensitive patterns.

All experiments were carried out on a MacBook Pro with an Apple M1 chip and 32GB of RAM. Our source code is written in Python and is available at URL: https://github.com/Yebulabula/String-Sanitization-Project. Because of the randomness of algorithms, we get our result by running each experiment at least 20 times and removing outliers.

7.4 CSD-PLUS *versus* Baseline

Data & parameters:

Data set	Deletions	Iterations	tau	omega	E(pruning)	C_p
SYN	10	[1000,3000]	3	0.5	[5,10]	[25,50]
TRU	10	1000	10	0.5	[5,15]	50
DNA	10	[1000,3000]	10	0.5	[5,10]	[50,80]

Table 7.2: Experimental Data & parameters for CSD-PLUS *versus* Baseline (varying pattern length k)

Data set	Deletions	k	Iterations	tau	omega	E(pruning)	C_p
SYN	10	3	[1000,3000]	3	0.5	[5,10]	50
TRU	10	4	1000	5	0.5	[5,15]	25
DNA	10	4	1000	10	0.5	[5,10]	50

Table 7.3: Experimental Data & parameters for CSD-PLUS *versus* Baseline (varying number of sensitive patterns)

Note that: we used a small pruning parameter E in this experiment because we have already tested that a larger pruning parameter did not perform better.

- **Spurious Pattern Distortion Reduction:** The bar charts in Fig.7.1&7.2 compares the performance of BA and CSD-PLUS in reducing spurious pattern distortion.

  Varying number of sensitive patterns:

  For a varying number of sensitive patterns, Fig.7.1 clearly shows that CSD-PLUS reduces distortion more than the BA over all datasets. Of the three datasets, CSD-PLUS reduces the enormous amount of distortion on TRU. In addition, Fig.7.1a & 7.1b clearly demonstrates that BA performed poorly in the SYN and DNA dataset, even aggravates the distortion to a certain extent ($e.g.$, In Fig.7.1b, the distortion reduction of the baseline is only positive when $|S| = 60$; otherwise, it is negative.) However, from the figures, our experimental results clearly reveal that CSD-PLUS implies no distortion increase and outperforms BA for all types of dataset. Furthermore, the model's distortion reduction in the SYN and DNA data sets is at least four times that of the BA.

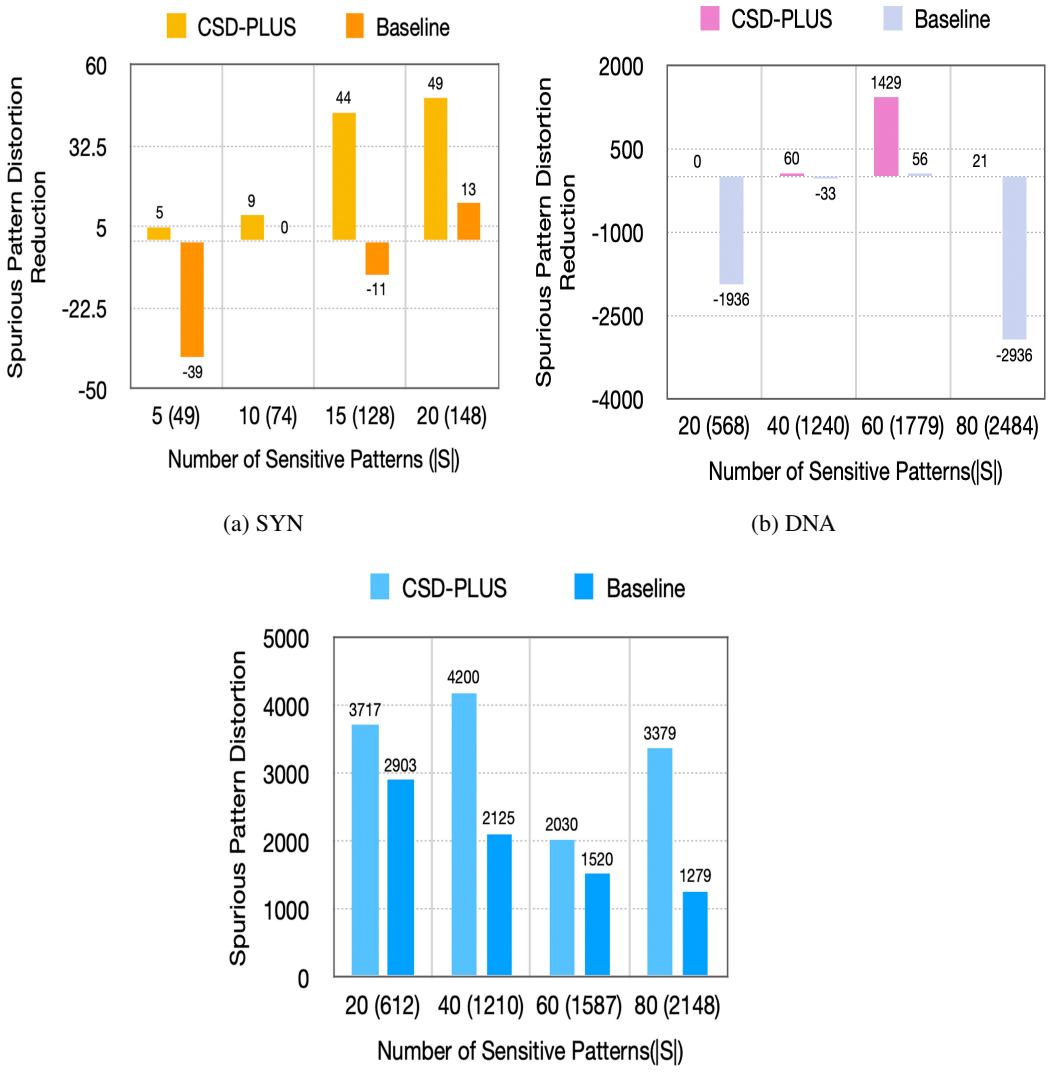
  Varying pattern length k:

  In Fig.7.2, we compare two algorithms by varying pattern length k. On all datasets, the charts clearly indicate that CSD-PLUS performs better than BA. The biggest disparity between the two algorithms occurs in DNA ($e.g.$, For the second bar in Fig.7.2b, CSD-PLUS reduces 6986 more distortions than BA). Moreover, except $k = 5$ in TRU, in all other cases, the CSD-PLUS distortion reduction was at least 3 (and up to 24) times larger than the BA distortion reduction. Also, CSD-PLUS implies only utility improvement.

- **Non-spurious Pattern Distortion Reduction:** Since we set a low ω in our experimental setting, our model's non-spurious pattern distortion reduction may be inferior to spurious pattern distortion reduction.

  Varying number of sensitive patterns:

  As the number of sensitive patterns varies, our result in Fig.7.3 shows that compared to BA, CSD-PLUS still has the better performance in all cases. Fig.7.3b shows that both algorithms are effective, and the distortion reduction



(a) SYN

(b) DNA

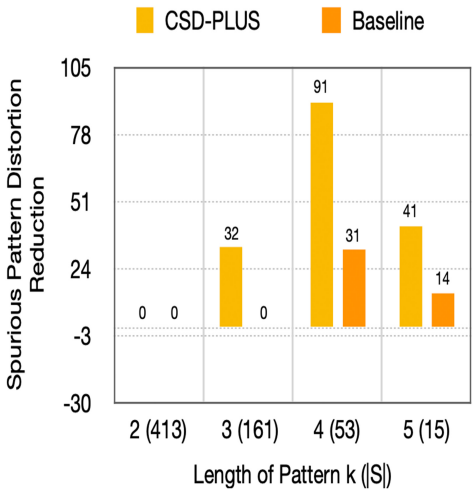
(c) TRU

Figure 7.1: Spurious pattern distortion reduction $versus$ number of sensitive pattern(and $|S|$), where |S| is the total number of sensitive patterns

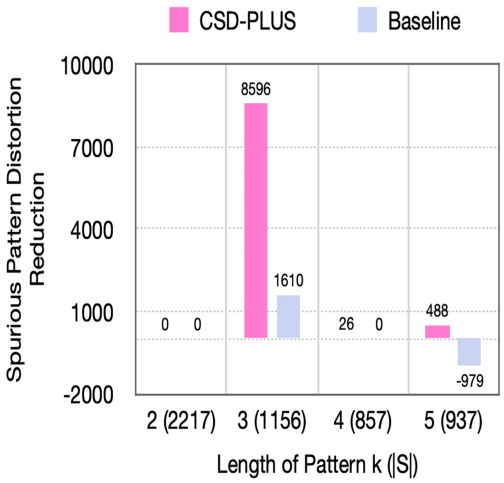
between them is close in SYN. However, from Fig.7.3b I Fig.7.3c, the results clearly demonstrate that BA usually leads to more severe utility loss. At the same time, CSD-PLUS can decrease pattern distortion in most situations.

Varying length of pattern k***:***

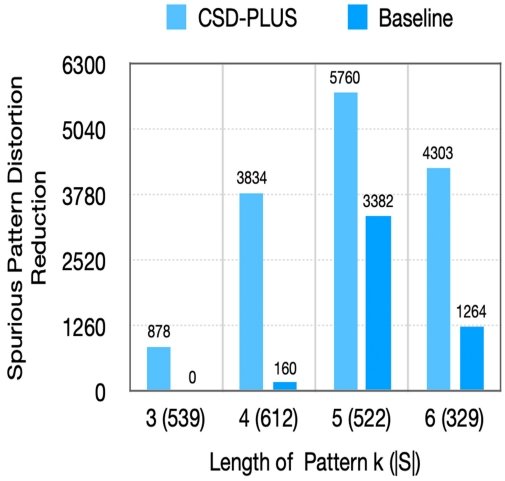
We also evaluated two algorithms by varying the length of pattern k. Fig.7.4 clearly shows that CSD-PLUS incur more significant distortion reduction than baseline in all cases. Furthermore, CSD-PLUS guaranteed that removing symbols would not negatively impact the data utility. In Fig.7.4c, BA led to an increase of distortion in any pattern length k. Hence, the BA algorithm performed poorly on handling challenging dataset (*i.e.,* TRU).



(a) SYN



(b) DNA



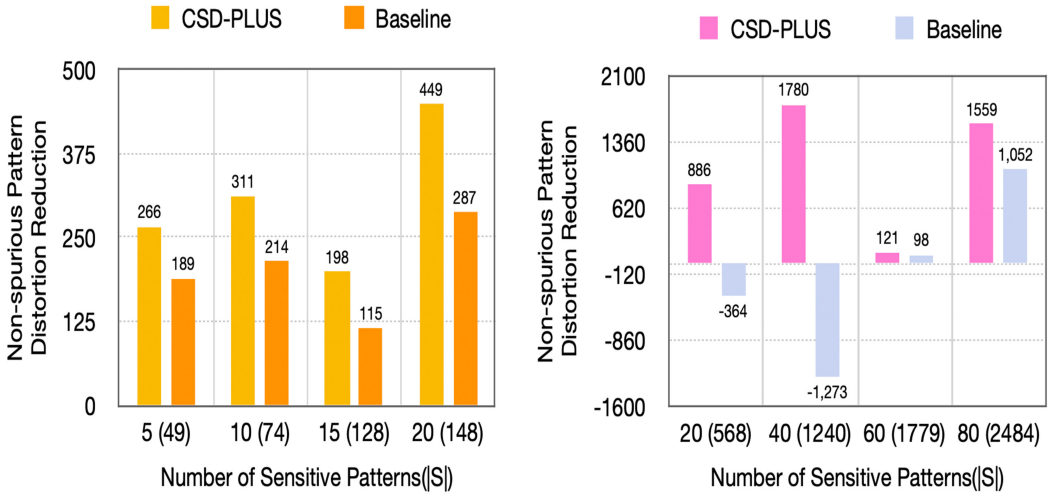
(c) TRU

Figure 7.2: Spurious pattern distortion reduction $versus$ pattern length k (and $|S|$)

($e.g.$, In Fig.7.4c, BA raises distortion by 22613)

- **Total Distortion:** For a varying number of δ, the result in Fig.7.5 clearly shows that CSD-PLUS can guarantee increasing deletions can continuously improve the data utility. However, the reduction in BA distortion would rapidly slow down as δ increases. It also indirectly shows that CSD-PLUS can find a close optimal solution per each deletion.

At last, we can conclude from the above experimental results that CSD-PLUS is a robust model and outperforms BA in any size data set, any length pattern, and any number of sensitive patterns. CSD-PLUS decreases the distortion for both types of non-sensitive patterns in most cases. Although our model may incur a small



(a) SYN

(b) DNA

(c) TRU

Figure 7.3: Non-spurious pattern distortion reduction $versus$ number of sensitive pattern(and $|S|$)

increment in non-spurious pattern distortion in the specific case, such distortion is nearly negligible in contrast to the decrease in distortion in spurious patterns. Furthermore, it can produce competitive results even in complicated datasets ($e.g.$, TRU).

7.5 ELLS-ALGO *versus* Naive-ELLS

In this section, we used an SYN_{Big} to measure the running time of Naive-ELLS and ELLS-ALGO, whose length $n = 50000$. Furthermore, both algorithm performs 2000 iterations, and $E = \infty$ (i.e., expand all nodes without pruning) , $C = 50$.

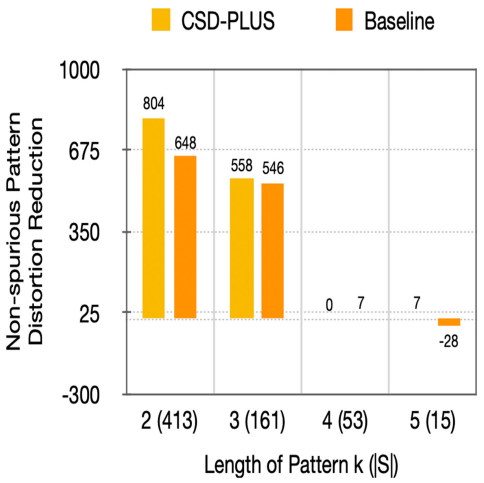
1. *Running time versus deletions* δ:

   (Data: Substring of SYN_{Big}, whose $n = 20000$)

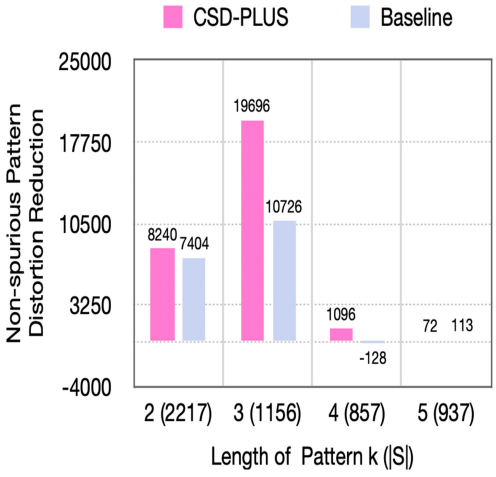
   Fig.7.6 shows that the running time of ELLS-ALGO increases linearly as the number of deletions δ increases, while the running time of Naive-ELLS grows exponentially. As $\delta = 10$, ELLS-ALGO is nearly 30 seconds faster than its naive version. That makes sense because when we remove more *letters*, our optimization effect, such as search tree reuse, would be more evident.

2. *Running time versus data size* n:

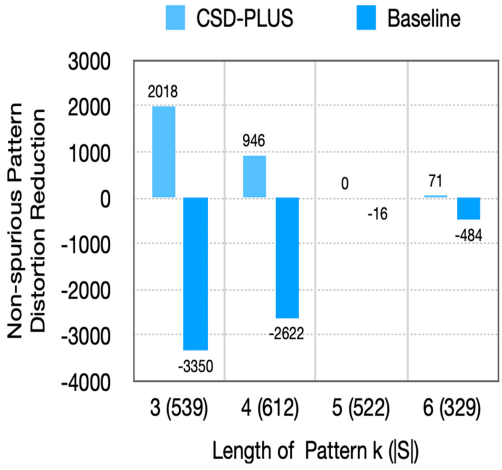
   In Fig.7.7, we compare the two algorithms' efficiency by deleting 5 *letters*



(a) SYN



(b) DNA



(c) TRU

Figure 7.4: Non-spurious pattern distortion reduction $versus$ pattern length k (and $|S|$)

from data of varying sizes. The result clearly shows that Naive-ELLS is quite sensitive about the rise of data size. As running 50k *letters*, Naive-ELLS already have to spend around 700s. In contrast, CSD-PLUS scaled linearly with n.

3. *Running time versus pattern length k*:

   (Data: Substring of SYN_{Big}, whose $n = 20000$)

   Although both algorithms scaled linearly with k, the result in Fig.7.8 reveals that on average, optimized ELLS are 3s faster than naive-ELLS.

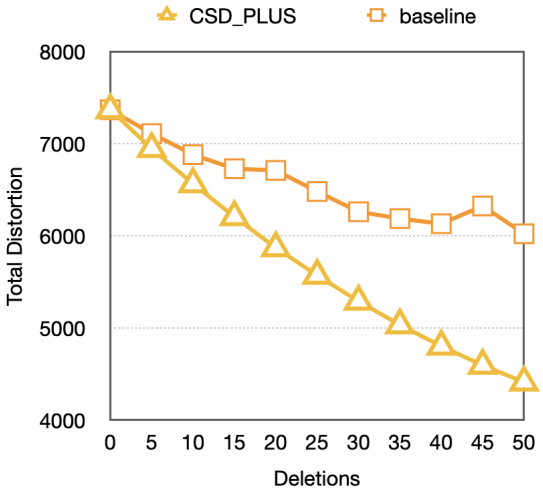


Figure 7.5: Total distortion $(versus)$ deletions δ (SYN)

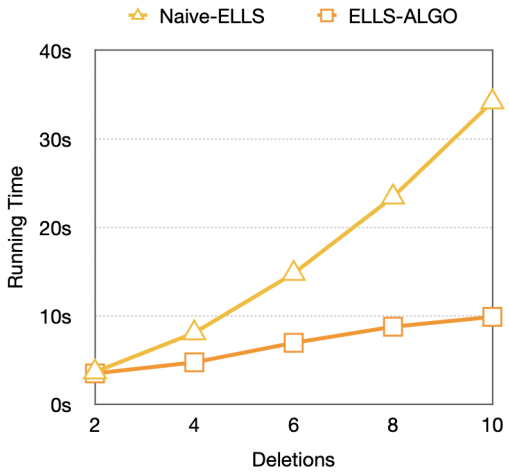


Figure 7.6: Running time $versus$ deletions δ (SYN_{Big})

7.6 The Parameter ω

This section evaluates how ω impacts the performance of the algorithm. To highlight the trend, we compare the variation of distortion in two types of patterns by calculating: $\frac{Distortion_{\mathcal{H}}}{Distortion_{Z}}$, where $\mathcal{H}$ is a string after deletions, Z is a sanitized string. Fig.7.9 clearly shows that decreasing omega does not always result in a continuous reduction in the distortion of non-spurious patterns while increasing the distortion of spurious patterns. This is because our model still prefers to delete patterns with extremely high distortion when the distortion gap between the two types of patterns is huge. However, the effect of omega on two types of patterns is predictable in general.

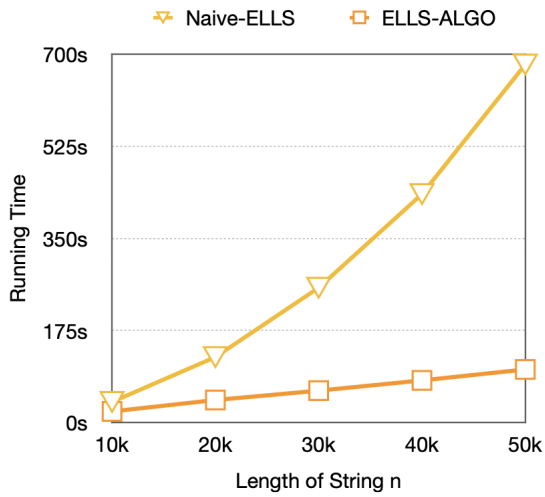


Figure 7.7: Running time $versus$ data size n (SYN_{Big})

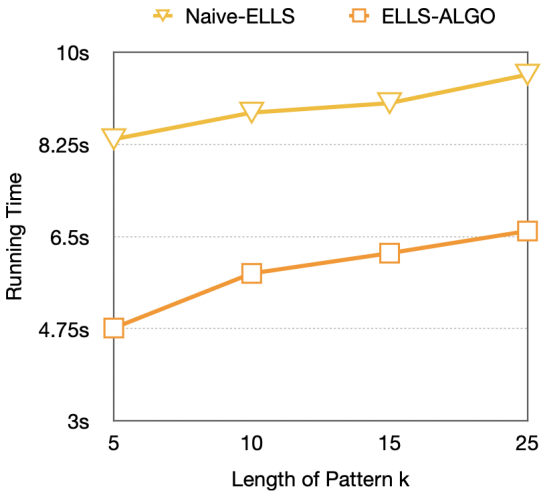


Figure 7.8: Running time $versus$ pattern length k (SYN_{Big})

7.7 Selection Criterions Comparison

In addition, we compared the effectiveness of the various action selection alternatives mentioned in section 5.3.9. The results are summarized in Fig.7.10, and indicate that:

- *Max-Child:* High convergence speed, however, the result may fluctuate until 4000 iterations.
- *Robust-Child:* Robust-Child deletes the letter that is associated with the most visited node. However, the result shows that the selected action is competitive only after 3500 iterations. Hence, this strategy requires the longest conver-

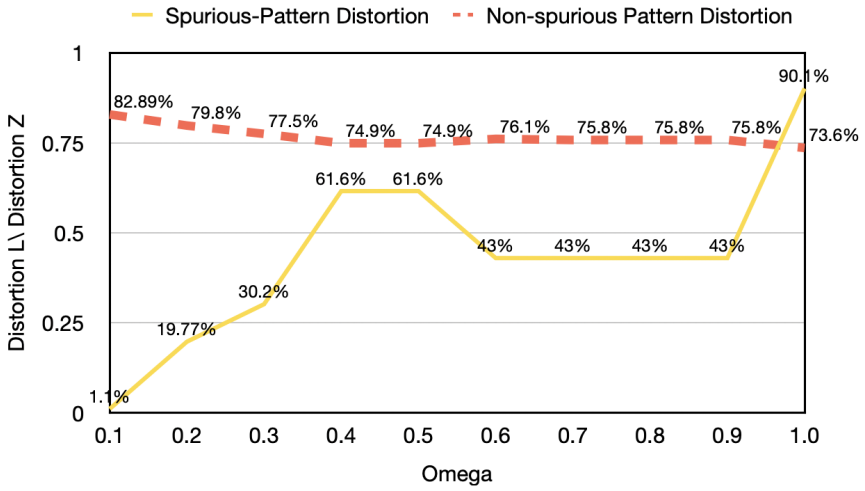


Figure 7.9: The ratio of Distortion$_{\mathcal{H}}$: Distortion$_Z$ $versus$ ω (SYN)

gence time of all strategies.

- *Max-Robust-Child:* Max-Robust-Child selects the deleted letter, which maximizes both the number of visits and the reward, and is the best-performing criterion. Fig.7.10 reveals that the suboptimal solution can be found within 3000 iterations.

7.8 ELLS-ALGO *versus* GD-ALGO

In this section, we used a substring in SYN$_{Big}$, followed this using the result of exhaustive search to evaluate the performance of each algorithm. *Parameters in experiment:*

$|w|$: 500; max-simulations: 2000; k = 4; $\tau = 10$; $\omega = 1$

	Deletions													
Algorithm	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Exhaustive Search	597	564	533	499	477	452	424	402	379	342	333	312	290	274
GD-ALGO	597	564	533	504	477	452	427	402	379	356	333	312	293	274
ELLS-ALGO	597	564	533	499	477	452	424	402	379	342	333	312	290	274

Table 7.4: Total distortion vs number of deleted *letters*, column: the number of deletions, row: type of algorithm

In most cases, the result of GD-ALGO is also a global solution, as shown in Table 7.4. The experimental results make sense because, despite the fact that the LUL problem is NP-hard, each deleted symbol can only affect $2k - 1$ patterns.

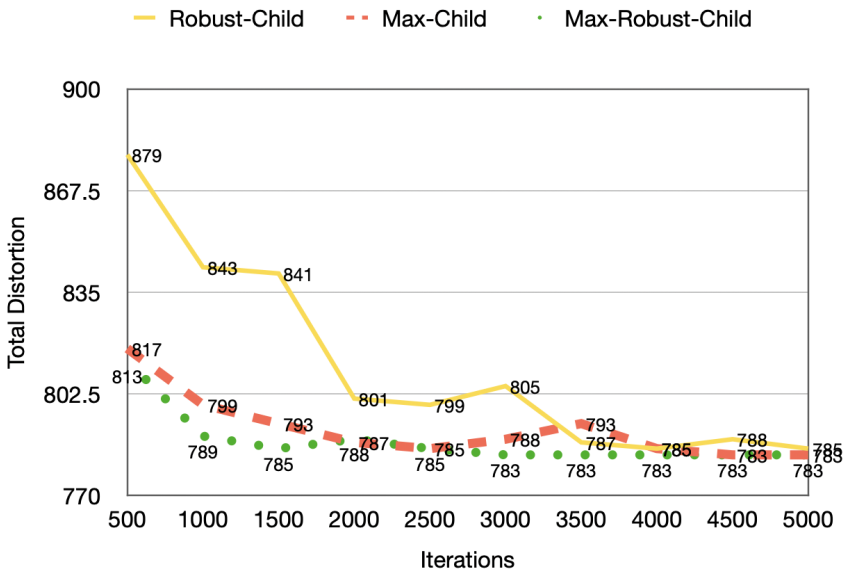


Figure 7.10: Total distortion vs iterations
(SYN)

In practice, the length of k is typically small compared to the size of the data, and not all patterns are from these $2k - 1$ patterns. Therefore, even though the long-term effect of deletion is unpredictable, its impact is generally smaller than expected. However, that does not mean that we can simplify any search steps because extreme cases must still be considered. Furthermore, no explicit result proves that the performance of GD-ALGO would decay as δ increases. In addition, ELLS-ALGO can find the optimal global solution for any number of deletions, which will be at least as good as GD-ALGO's results.

7.9 Limitations:

Due to the LUL problem's strong hardness, deleting a *letter* will affect the distortion of all patterns in the worst case. Therefore, we have to recalculate the *R-score* of each possible deleted *letter* at each step. For ELLS-ALGO, although it only expands promising branches each time, when the size of the dataset grows to a certain extent, the tree generated by our algorithm will potentially have a high branching factor. Our algorithm would require a reasonably large number of simulations to get a reliable result under this situation. For example, we are more likely to miss the optimum value if we do not visit every node in the tree at least once. Therefore,

there is a typical speed-accuracy tradeoff in our algorithm. So far, we still tend to select smaller δ and data size n to bound the depth and branching factor of the tree to guarantee the quality of the results.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this project, we presented a novel strategy for improving the utility of sanitized string, so that it can be accurately mined by frequent pattern mining technique. This approach focuses on reducing data distortion by removing redundant *letters*. We specified a problem (LUL) that aims to minimize distortion by deleting δ *letters*. We developed the *R-score* function to address this problem, which evaluates the value of each deleted letter. We also developed two algorithms, GD-ALGO and ELLS-ALGO, to reduce the total *R-score* by deleting the best-δ deleted *letters*. First of all, we discovered that the outcome of GD-ALGO is a nearly global optimum answer, even though it exploits greedy search. Secondly, our experimental results reveal that ELLS-ALGO results are at least as good as those of GD-ALGO, within an acceptable number of simulations. Furthermore, our results demonstrate that both algorithms are linearly scalable with data size and thus efficient.

8.2 Suggestions for Future Work

To perform a simulation, the current ELLS-ALGO randomly selects a sequence of *letters* to delete. However, as the size of the tree grows larger, this approach usually necessitates a large number of iterations to converge the outcome. Therefore, for future work, we will first attempt to integrate heuristic knowledge of the problem into our simulation strategy, which will significantly accelerate convergence. If the search space can be further reduced, our algorithm will have the ability to select more deleted *letters*. Secondly, in 2014, a new algorithm called EMCTS [16] (Evolutional Monte Carlo Tree Search) was proposed. This algorithm combined the advantages of a genetic algorithm and MCTS. We will determine if it can solve

the LUL problem more effectively. At the next step, we will also attempt to define a more complicated problem: minimizing the distortion of the sanitized string with the smallest edit distance.

Bibliography

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499. Citeseer, 1994.

[2] P. AnnanNaidu. An efficient approach for privacy preserving data mining using clustering methods and secure multi party computation techniques. 2019.

[3] Daniel Whitehouse Cameron Browne, Edward Powley. *A Survey of Monte Carlo Tree Search Methods*. IEEE Conference, 2021.

[4] Jian Yuan Chunxiao Jiang, Jian Wang. *Information Security in Big Data:Privacy and Data Mining*. IEEE Conference, 2014.

[5] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[6] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[7] Soumadip Ghosh, Sushanta Biswas, Debasree Sarkar, and Partha Pratim Sarkar. Mining frequent itemsets using genetic algorithm. *arXiv preprint arXiv:1011.0328*, 2010.

[8] et al. Giulia, Bernardini. *Hide and Mine in Strings: Hardness and Algorithms*. 20th IEEE International Conference on Data Mining (ICDM 2020), 2020.

[9] Alessio Conte Roberto Grossi Grigorios Loukides Giulia Bernardini, Huiping Chen. *String Sanitization: A Combinatorial Approach*. ECML PKDD, 2019.

[10] Loukides G. Gkoulalas-Divanis, A. *Revisiting sequential pattern hiding to enhance utility*. In: KDD. pp. 1316–1324, 2011.

[11] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery*, 15(1):55–86, 2007.

[12] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[13] Takeaki Uno Hiroki, Arimura. *Polynomial Space and Polynomial Delay Algorithm for Enumeration of Maximal Motifs in a Sequence.* ISAAC2005, Lecture Notes in Computer Science 3827, pp. 724-737, 2005.

[14] Takeaki Uno Hiroki, Arimura. *An Efficient Polynomial Space and Polynomial Delay Algorithm for Enumeration of Maximal Motifs in a Sequence.* to appear in Journal of Combinatorial Optimization, 2006.

[15] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[16] Simon M Lucas, Spyridon Samothrakis, and Diego Perez. Fast evolutionary adaptation for monte carlo tree search. In *European Conference on the Applications of Evolutionary Computation*, pages 349–360. Springer, 2014.

[17] Miller W. Myers, E.W. *Approximate matching of regular expressions.* Bulletin of Mathematical Biology 51(1), 5–37, 1989.

[18] JEFF PETTERS. Data Privacy Guide: Definitions, Explanations and Legislation. http://www-cs-faculty.stanford.edu/~uno/abcde.html.

[19] Harvey M Salkin and Cornelis A De Kluyver. The knapsack problem: a survey. *Naval Research Logistics Quarterly*, 22(1):127–144, 1975.

[20] Wikipedia. Sequential pattern mining. https://en.wikipedia.org/wiki/Sequential_pattern_mining.