String Clustering: Finding Edit Similarity over Large Dataset

An Honours Thesis

submitted in partial fulfillment of the requirements for the degree of

B.Comp (Hons) in Computer Science

presented to

the Department of Computer Science

School of Computing

National University of Singapore

Asst. Prof. Diptarka Chakraborty, Supervisor

Asst. Prof. Warut Suksompong, Evaluator

by

Audrey Felicio Anwar

April 5, 2023

Acknowledgements

First and foremost, I am immensely grateful to my supervisor Assistant Professor Diptarka Chakraborty for his unwavering support, valuable guidance, and constructive feedback throughout the entire research process. Without his expertise, encouragement, and many times valuable insights, this thesis would not have been possible. It has truly been an honor to have him as my supervisor for this research.

I would also like to thank my family, friends, and colleagues for their constant encouragement, motivational words, and unwavering belief in me. Their continuous emotional and moral support has kept me motivated and strong throughout the most difficult times of this thesis. This thesis could not have been completed without the support of all of you.

Lastly, I would like to express my gratitude to National University of Singapore for their support on my journey into this research and for providing me with the required resources, which have been essential to the completion of this thesis.

Abstract

Finding a representative of a huge set of data has been a fundamental problem in big data analysis. As an example, one may be interested in finding the median of a certain dataset as a representative of the whole data. This problem finds numerous applications in fields such as computational biology, DNA storage system, and speech recognition.

In this thesis, we will explore various works that have pushed the boundaries of this research interest. Specifically, we will be looking at a special problem of finding the median string under the Ulam metric, where all the data are permutations and the distance between two data is defined as the minimum number of symbol moving (deletion and then insertion) needed to transform one into the other.

We shall delve into attempts made to improve the state-of-the-art polynomial time approximation algorithm for this problem. In addition to it, some attempts on proving the hardness of the problem by means of finding a chain of reduction from NP-Hard problems are also shown. Lastly, some implementation of the state-of-the-art approximation algorithms and heuristical algorithms such as stochastic local search are provided to help bridge the gap between theory and practice by verifying the theorems empirically.

Contents

Chapter 1

Notations and Preliminaries

1.1 Notations

- $[n]$ denotes the set of numbers $\{1, 2, \cdots, n\}$.
- S_n denotes the set of all permutations over the set $[n]$.
- $ed(x, y)$ denotes the edit distance between two strings x, y.
- $ul(x, y)$ denotes the Ulam distance between two permutations x, y.
- $red(x, y)$ denotes the restricted edit distance between two strings x, y.
- $LCS(x, y)$ denotes the longest common subsequence between two strings (or permutations) x, y.
- $x[a : b]$ denotes the substring from index a to index b of a given string x.

1.2 Preliminaries

Definition 1.2.1 (Permutation)**.** A permutation of length n is a set of n distinct integers $\{1, 2, ..., n\}$ in any arbitrary order. As an example $\{3, 2, 1, 4\}$ is a permutation of length 4 but $\{2, 2, 1\}$ is not a permutation since 2 appears twice.

Definition 1.2.2 (Metric Space)**.** A metric space is an ordered pair (M, d) where M is a set and $d : M \times M \to \mathbb{R}$ a function such that

(1) d(x, x) = 0

(2) If $x \neq y$, $d(x, y) > 0$

(3) $d(x, y) = d(y, x)$

(4) $d(x, z) \leq d(x, y) + d(y, z)$

Definition 1.2.3 (Edit Distance)**.** Given two strings x, y over some alphabet Σ, the edit distance between them is the minimum number of character insertions, deletions, or substitutions needed to transform x into y.

Definition 1.2.4 (Edit Metric)**.** A metric space identified by the ordered pair (S, ed) where S denotes a set of strings over some alphabet Σ and ed denotes the edit distance.

Definition 1.2.5 (Restricted Edit Distance)**.** Given two strings x, y over some alphabet Σ, the restricted edit distance between them is the minimum number of character insertions or deletions needed to transform x into y. Character substitutions are not allowed.

Definition 1.2.6 (Restricted Edit Metric)**.** A metric space identified by the ordered pair (S, red) where S denotes a set of strings over some alphabet Σ and red denotes the restricted edit distance.

Definition 1.2.7 (Ulam Distance)**.** Given two permutations $x, y \in S_n$, the Ulam distance between them is the minimum number of character moves to transform x into y. A character move on a permutation p is defined as removing that character from p and immediately inserting it back at any location in p. As an example $[1, 2, 3, 4] \to [1, 3, 4, 2]$ is produced by one move of the character 2.

Definition 1.2.8 (Ulam Metric)**.** A metric space identified by the ordered pair (S_n, ul).

Definition 1.2.9 (Longest Common Subsequence (`LCS`))**.** Given a set of strings S over some alphabet Σ, the longest common subsequence is a string l over the same alphabet Σ such that it is a common subsequence of any $s \in S$ and it's length is maximum.

The decision version of this problem is that given some constant k, decide whether there exist a common subsequence of S such that it's length is $\geq k$.

Theorem 1.2.10. *Given 2 permutations $x, y \in S_n$, $ul(x, y) = n - |LCS(x, y)|$.*

Definition 1.2.11 (Median of Metric Space)**.** Given a metric space (M, d) and a set of data $S \subseteq M$, the median of the dataset is defined as $m^* \in M$ such that $m^* = \arg\min\limits_{m \in M} \left(\sum\limits_{s \in S} d(m, s) \right)$.

Definition 1.2.12 (`Edit Median`)**.** Given a set S of strings over some alphabet Σ, find the median of S with respect to the Edit metric. The decision version is given k, decide whether there exists some string s' over Σ such that $\sum\limits_{s \in S} ed(s, s') \leq k$.

Definition 1.2.13 (`Ulam Median`)**.** Given a set $S \subseteq S_n$ of permutations, find the median of S with respect to the Ulam metric. The decision version is given k, decide whether there exists some permutation p' over Σ such that $\sum\limits_{p \in S} ul(p, p') \leq k$.

Definition 1.2.14 (k-Median of Metric Space)**.** Given a metric space (M, d) and a set of data $S \subseteq M$, the k-median of the dataset is defined as a set $X^* \subseteq M$ such that $|X^*| = k$ and $X^* = \arg\min\limits_{X \subseteq M, |X| = k} \left(\sum\limits_{s \in S} \left(\min\limits_{x \in X} d(x, s) \right) \right)$

Definition 1.2.15 (Center of Metric Space)**.** Given a metric space (M, d) and a set of data $S \subseteq M$, the center of the dataset is defined as $m^* \in M$ such that $m^* = \arg\min\limits_{m \in M} \left(\max\limits_{s \in S} d(m, s) \right)$.

Definition 1.2.16 (`Ulam Center`)**.** Given a set $S \subseteq S_n$ of permutations, find the center of S with respect to the Ulam metric. The decision version is given k, decide whether there exists some permutation p' over Σ such that $\max\limits_{p \in S} ul(p, p') \leq k$.

Definition 1.2.17 (k-Center of Metric Space)**.** Given a metric space (M, d) and a set of data $S \subseteq M$, the k-center of the dataset is defined as a set $X^* \subseteq M$ such that $|X^*| = k$ and $X^* = \arg\min\limits_{X \subseteq M, |X| = k} \left(\max\limits_{s \in S} \left(\min\limits_{x \in X} d(x, s) \right) \right)$

Definition 1.2.18 (Alignment)**.** Given two strings (permutations) x and y of lengths n_1 and n_2 respectively, an alignment g is a function from from $[n_1]$ to $[n_2] \cup \{*\}$ which satisfies:

- $\forall i \in [n_1]$, if $g(i) \neq *$, then $x(i) = y(g(i))$
- For any two $i \neq j \in [n_1]$ such that $g(i) \neq *$ and $g(j) \neq *$, if $i > j$ then $g(i) > g(j)$

We say that g aligns a character $x(i)$ with some character $y(j)$ iff $j = g(i)$. An optimal alignment between x, y would be the $LCS(x, y)$.

Definition 1.2.19 (NP)**.** A decision problem D is in NP if and only if there exists some certificate C of polynomial size with respect to the input encoding of D and there exists some verifier algorithm A that decides whether C is a YES-instance of D in polynomial time with respect to the input encoding of D.

Definition 1.2.20 (NP-Hard)**.** A decision problem D is NP-Hard if and only if all the problems $L \in$ NP can be reduced in polynomial time to D.

Definition 1.2.21 (NP-Complete)**.** A decision problem D is NP-Complete if and only if it is in both NP and NP-Hard.

Definition 1.2.22 (`Max-Clique`)**.** Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, determine the largest clique $C \subseteq V$. A clique of size k is defined to be a complete graph on k vertices.

The decision version of this problem is that given k, determine if there exists a clique of size $\geq k$ in G.

Definition 1.2.23 (`LCSO`)**.** Given a positive constant k and n strings $w_1, w_2, \cdots, w_n$ each of length $2k$ over some alphabet Σ, decide whether there exists a string w such that $|w| \geq k$ and w is a subsequence of each w_i.

Chapter 2

Introduction and Literature Review

Finding a representative for a given dataset is one of the most common aggregation tasks in the field of data analysis. Two of the most popular representative would be the median and the center. Finding such representatives under certain metrics proved to have a lot of applications in other fields of Computer Science and even other science disciplines such as Biology [Nav01; Gus97].

One of the most well-known metrics is the Edit metric (1.2.4) because it finds vital applications in Biology such as finding sequence alignment between DNAs to find evolutionary relationships between genes and proteins. The edit distance is firstly defined by Levenshtein in his finding of correcting binary codes [Lev65]. Over time the edit distance is adapted toward fields and problems that require a concept of similarity as well as some algorithm to compute it.

There exists a well-known dynamic programming algorithm (3.0.7) that computes the exact distance between two instances in the general edit distance. Although it is not too efficient, it is the most flexible one as it can easily be adapted to different distance functions. Improvements to the worst-case time complexity of the dynamic programming approach have been made over the years, from the original $O(mn)$ down to $O(\min(n, m)k)$, where n, m are the length of the two strings, k is the edit distance [Ukk85]. The idea is to compute and store only around the diagonals of the dynamic programming table. For small $k < m^{1/4}$, Cole and Hariharan developed a faster algorithm that runs in $O(n(1+\frac{k^c}{m}))$ where $c = 3, 4$ is some constant dependant on the input [CH98].

Finding the edit distance over an arbitrary set of strings is computationally hard due to a result by Jiang and Li. Jiang and Li showed that approximating Longest Common Subsequence, which corresponds to exactly finding the minimum edit distance is NP-Hard by a reduction from another NP-Hard problem `Max-Clique` [JL95]. As a result, further reduction also shows that aggregation tasks such as finding the median (and consequently center) string in the edit metric are also NP-Hard [HC96].

Some recent work in the field has been on discovering a fast approximation algorithm for edit distance. The idea is to allow errors within a tolerable margin to get a better-performing algorithm. A recent breakthrough in this area is a sub-quadratic time $\tilde{O}(n^{2-2/7})$ algorithm that approximates the edit distance within a constant factor [Cha+18]. The breakthrough is then improved upon by Andoni and Nosatzki to $O(n^{1+\epsilon})$ for any constant $\epsilon > 0$ [AN20]. The randomized algorithm achieves a $f(\frac{1}{\epsilon})$ factor approximation. These two breakthroughs are ground-breaking as a (strongly) sub-quadratic time algorithm has not been discovered for decades.

To find insights, people have looked into other metric spaces as well such as Hamming, Jaccard, Kendall's tau, and many more. Some metric spaces are known to be computationally easy. As an example, finding the median string of size n over the Hamming distance is as simple as taking a majority character over each index in $[n]$ which can be done in polynomial time (3.0.6). Some metric spaces on the other hand are also known to be computationally hard. As an example, finding Kendall's tau median over a set of permutations S is NP-Hard even for $|S| = 4$ [Dwo+01].

One such metric space that is closely related to Edit metric is Ulam metric (1.2.8). In Ulam metric, we have a restriction that each symbol (character) can only appear once in the string, hence they are permutations of the alphabet Σ. Traditionally, the distance in Ulam metric is defined as the minimum number of character moves needed to transform one permutation to the other. However, researchers have also looked into the more general version of Ulam metric, where Levenshtein distance is used instead (also called permutation edit distance) [CK06].

There are various reasons why Ulam metric is of particular interest. The first reason is that Ulam metric captures the same core difficulties that are found in edit metric, namely the existence of misalignments between two strings (or permutations) [AN10]. Thus it is hoped that the core ideas that are useful to prove important results in Ulam metric can also be used and extended to the Edit metric.

The other reason is that the Ulam metric also finds practical applications in real-life. A popular use case is for rank aggregation problems, such as ranking webpages, employment, college admissions, voting, etc [Cha+22; Mar14]. As a side note, Kendall's tau is also a popular metric for rank aggregation problems. Another novel application is to design a data representation scheme for nonvolatile memory such as flash memory that can correct translocation errors [FM14; Jia+09; FSM13].

The `Ulam Median` (1.2.13) is simply the median string (1.2.11) problem under the (traditional) Ulam metric. Unlike the median string problem (over Edit metric), as of the writing of this thesis, it is unknown if it is in NP-Hard or not. Despite the uncertainty of the hardness of the problem, researchers have been trying to come up with polynomial time approximation algorithms for the `Ulam Median` problem.

For decades, the only known best approximation algorithm is simply a folklore algorithm that produces a 2-approximate solution under any metric space, that is outputting the best input (3.0.2). No polynomial time approximation algorithms have broken through the 2-approximate barrier until recently when Chakraborty et. al. come up with the following result in 2020 [CDK21].

Theorem 2.0.1. *There exists a deterministic algorithm that given an input a set of m permutations $S \subseteq S_n$, computes a $(2-\delta)$-approximate median in $O(nm^2 \log n + n^2 m + n^3)$ time.*

The δ is a very small constant $\approx 2^{-40}$ but it still breaks the 2-approximation barrier.

The idea behind the work is to look into when does the folklore algorithm fails to achieve an approximation better than 2. Turns out the folklore algorithm yields a good approximation when the optimal objective value is large. Otherwise, the structure of the input permutations must be to some extent exploited. The main idea to exploit is to divide the set of inputs into two categories, those that are far from the median and those that are close, and respectively find an upper bound for both sets of inputs.

Another novel idea that is used is to analyze differently based on the number of bad symbols. A symbol is defined to be bad if it misaligns (1.2.18) in almost all of the input strings with respect to the median. In doing so the idea that the folklore algorithm finds a good approximation when objective value is large can be leveraged further.

As a small result, the paper also showcased that it is indeed possible to compute an exact median when the input is of only three permutations.

Theorem 2.0.2. *There exists a deterministic algorithm that given an input 3 permutations $s_1, s_2, s_3 \in S_n$, computes the median in $O(n^4)$.*

The main idea is to compute by Dynamic Programming an exact median string of the 3 input permutations under the restricted form of edit metric (1.2.6), where character substitutions are not allowed, and then carefully convert it into a permutation. It can be shown that if a permutation x' is the median of the restricted edit metric, then it must also be the median under Ulam metric (3.0.5).

Let x' be the restricted edit median. To carefully convert it to a permutation without reducing it's optimality, we fix some optimal alignment (1.2.18) with all of the input strings, remove all the duplicate symbols that don't maximally align, and finally add the missing symbols back by aligning with any of the strings. As a byproduct, by following the same steps, given a fixed constant m denoting the number of permutations, then we can always get a 1.5-approximation of the median in $O(2^{m+1}n^{m+1})$.

Theorem 2.0.3. *There exists a deterministic algorithm that given an input m permutations $s_1, s_2, ..., s_m \in S_n$, computes a 1.5-approximate median in $O(2^{m+1}n^{m+1})$.*

The idea of fixing an exact optimal solution under a more "general" metric can be applied also to the center string problem (1.2.15). Chakraborty et. al. came up with the following result for computing an approximate `Ulam center` (1.2.16) [CGJ21].

Theorem 2.0.4. *There exists a deterministic algorithm that given an input a set of permutations $S \subseteq S_n$, where $|S| = m$, computes a $(\frac{3}{2} - \frac{1}{3m})$-approximate center under the Ulam metric in $n^{O(m^2 \ln m)}$ time.*

As an extra result, the running time can be improved to $O\left(n^{3m} + n^{O(\frac{\ln m}{\epsilon^2})}\right)$ for any $\epsilon > 0$ by reducing the optimality approximation to $(\frac{3}{2} + \epsilon - \frac{1}{m})$-approximate center.

The main idea follows the 1.5-approximate median for Ulam metric. The first step is to fix an optimal n-length center by Dynamic Programming. The second step is to carefully convert the optimal n-length center to a permutation without decreasing too much of the optimality.

To carefully convert the optimal n-length center to a permutation, the duplicate symbols must be deleted in a balanced manner as a deletion will increase the distance to some $S' \subseteq S$. A reduction to a generalized closest string problem is used to produce a balanced deletion.

This result is a breakthrough as there was no known below 2-approximation algorithm for the `Ulam Center` problem. The best approximation algorithm was the folklore output the best input which achieves 2-approximation (3.0.3).

A natural extension to the median problem would be to allow to choose not only one median but k medians where $k \geq 1$ is some constant. This variant is called the k-median problem (1.2.14). The distance between an input s would be defined as the distance to the closest median among all the k medians chosen. In this variant, given an input S over some metric space M, the objective is to choose a subset $X \subseteq M$ such that $|X| = k$ and the Ulam distance between X and S is minimized.

To extend the previous breakthrough results to support k medians, a new framework is used. The result is the following theorems due to Chakraborty et. al [CDK22].

Theorem 2.0.5. *There exists a deterministic algorithm such that given an input $S \subseteq S_n$ where $|S| = m$, outputs a 1.999-approximate median in $O(m^5 n^3)$ time.*

Theorem 2.0.6. *There exists a deterministic algorithm such that given an input $S \subseteq S_n$ where $|S| = m$, outputs a 1.999-approximate k-median in $O(m^{O(k)} n \log n + m^5 n^3)$ time.*

The main idea of this new framework is to split the input into three categories. The first two categories are borrowed from the older framework, that is when some of the inputs are already a good enough approximation, either because they are close to a large cluster or they are close to the actual optimal medians. The last case is where the main difference lies with the old framework. Instead of analyzing the bad symbols, we simply compute an approximate median for every subset of 5 input permutations and among those approximate medians, we pick the k best.

Chapter 3

Basic and Miscellaneous Results

This chapter aims to provide some basic results and formally prove them. These basic results are the foundations in which most progress in the field has been made, including the attempts made in the later chapters. Some miscellaneous findings that don't fit in the rest of the chapters are also provided here along with the proofs.

Theorem 3.0.1 (Proof of Theorem 1.2.10)**.**

Proof. Consider a sequence of optimal moves $M = m_1, m_2, \cdots, m_k$ that is used to transform x into y. Let $I = \{i_1, i_2, \cdots, i_t\}$ be the indexes of the characters that are not moved by the moves M. Notice that the subsequence $x[i_1]x[i_2]\cdots x[i_t]$ appears in both x and y since they are not moved and x eventually transforms into y. Hence $x[i_1]x[i_2]\cdots x[i_t]$ is a common subsequence of x and y which implies $t \leq |LCS(x, y)|$.

Observe that $k \geq n - t$ since the rest of the characters whose index is not in I must be moved at some point. We have $ul(x, y) = k \geq n - t \geq n - |LCS(x, y)|$.

Now consider the $LCS(x, y)$. For any character in x that is not in $LCS(x, y)$, we can use one move operation to put that character in index j where $y[j]$ equals that character. We spend $n - |LCS(x, y)|$ number of moves. After those moves are done, notice that the characters in $LCS(x, y)$ need not be moved anymore as they are already in the correct positions. Hence, $ul(x, y) \leq n - |LCS(x, y)|$. Combining, we have $ul(x, y) =$

$n - |LCS(x, y)|$ as desired. □

Theorem 3.0.2 (Folklore 2-Approximation of Metric Median)**.** *There exists a deterministic algorithm such that given an input $S \subseteq M$ over some metric space (M, d), computes a 2-approximate median of S in polynomial time given that d is computable in poly$(|S|)$.*

Proof. Let $m \in M$ denote the optimal median of S and $\texttt{OPT} = \sum_{s \in S} d(m, s)$ be the optimal objective value. Let $s^* \in S$ denotes the input such that $d(s^*, m) \leq d(s, m)$ for any $s \in S$. By the triangle inequality, we have

$$\sum_{s \in S} d(s^*, s) \leq \sum_{s \in S} d(s^*, m) + d(m, s) \leq 2 \sum_{s \in S} d(m, s) = 2\texttt{OPT}$$

Now consider the input $x \in S$ such that $\sum_{s \in S} d(x, s) \leq \sum_{s \in S} d(y, s)$ for any $y \in S$. Notice that we have

$$\sum_{s \in S} d(x, s) \leq \sum_{s \in S} d(s^*, s) \leq 2\texttt{OPT}$$

Hence an input that minimizes the total distance to the other inputs is a 2-approximate median. Below is the algorithm pseudocode to find such an input:

```
def best_median_input(S):
  best_input = None
  best_distance = Infinity
  for s1 in S:
    total_distance = 0
    for s2 in S:
      total_distance += d(s1, s2)
    if total_distance < best_distance:
      best_distance = total_distance
      best_input = s1
  return best_input
```

Listing 3.1: Best Median Input

Overall the algorithm runs in $O(|S|^2 D)$ where D is the time to compute the distance function d, which is assumed to be computable in $poly(|S|)$. Hence, the algorithm runs in polynomial time. □

Theorem 3.0.3 (Folklore 2-Approximation of Metric Center)**.** *There exists a deterministic algorithm such that given an input $S \subseteq M$ over some metric space (M, d), computes a 2-approximate center of S in polynomial time.*

Proof. Let $c \in M$ denote the optimal center of S and $\texttt{OPT} = \max_{s \in S} d(c, s)$ be the largest distance from the center c to the inputs S. Notice that for any $x, y \in S$, we have by the triangle inequality

$$d(x, y) \leq d(x, c) + d(c, y)$$

For any input $x \in S$

$$\max_{y \in S} d(x, y) \leq \max_{y \in S}(d(x, c)+d(c, y)) \leq d(x, c)+\max_{y \in S} d(c, y) \leq \max_{x \in S} d(x, c)+\max_{y \in S} d(c, y) = 2\texttt{OPT}$$

Hence any input $x \in S$ is immediately a 2-approximate center. The algorithm is to just output any input in S, which can be done in $O(|S|)$ which is polynomial time. □

Theorem 3.0.4 (2-Approximation of Metric k-center)**.** *There exists a deterministic algorithm such that given an input $S \subseteq M$ over some metric space (M, d), computes a 2-approximate k-center of S in polynomial time given that d is computable in $poly(|S|)$.*

Proof. The following pseudocode denotes the algorithm:

```
def k_center(S, k):
  C = [ S[0] ]
  is_chosen = { 0: True }
  for i in range(k - 1):
    max_distance = 0
    max_index = 0
    for j in range(|S|):
      if is_chosen[j]:
          continue
        current_distance = Infinity
        for candidate in C:
          current_distance = min(current_distance, d(candidate, S[j]))
        if current_distance > max_distance:
          max_distance = current_distance
          max_index = j
      is_chosen[max_index] = True
      C.append(S[max_index])
  return C
```

Listing 3.2: Hamming Median

Fix an optimal k-center C^* and let OPT be the optimal objective value, that is $d(s, C^*) \leq$ OPT for any $s \in S$. Now consider the approximate centers C that are produced by the greedy algorithm. Notice that by definition, for each $s \in S$ we can always find a $c^* \in C^*$ such that $d(s, c^*) \leq \texttt{OPT}$. If for that c^* there exists a $c \in C$ such that $d(c^*, c) \leq \texttt{OPT}$, then by triangle inequality:

$$d(s, c) \leq d(s, c^*) + d(c^*, c) \leq 2\texttt{OPT}$$

Otherwise, because $|C^*| = |C| = k$, by pigeonhole principle there must exists some $s' \in C^*$ such that $d(s_i, s'), d(s_j, s') \leq \texttt{OPT}$ where $s_i, s_j \in C$. Without Loss of Generality assume that s_i is chosen first before s_j by the algorithm. Let C' denote the set of medians chosen right before s_j is added. By construction of the algorithm, for every $s \in S \backslash C'$, $d(s, C') \leq d(s_i, s_j)$ since s_j maximizes the distance to C' among everything in $S \backslash C'$. By triangle inequality again

$$d(s_i, s_j) \leq d(s_i, s') + d(s', s_j) = 2\texttt{OPT}$$

Hence, the set C is a 2-approximate k-center as desired. The algorithm runs in $O(|S|k^2 D)$ where D is the time needed to compute the distance function d, which is assumed to be computable in $poly(|S|)$. Hence it runs in polynomial time. □

Theorem 3.0.5 (Median of Restricted Edit is Median of Ulam)**.** *Let $S \subseteq S_n$ be a given set of permutations. Let x' be the Restricted Edit median over S. If x' is a permutation then it is also an Ulam median over S.*

Proof. Notice that every Ulam move can be modeled with a single deletion followed by a single insertion. We observe that given two strings x, y, it must be the case that $2ul(x, y) = red(x, y)$.

Let x' be the Restricted Edit median that is also a permutation and let p be some permutation such that it is an Ulam median. By definition, we have $\sum\limits_{s \in S} ul(x', s) \geq \sum\limits_{s \in S} ul(p, s)$. We also have $\sum\limits_{s \in S} red(x', s) \leq \sum\limits_{s \in S} red(p, s) \implies \sum\limits_{s \in S} 2ul(x', s) \leq \sum\limits_{s \in S} 2ul(p, s)$. Hence we have $\sum\limits_{s \in S} ul(x', s) \leq \sum\limits_{s \in S} ul(p, s)$. Concluding, we have $\sum\limits_{s \in S} ul(x', s) = \sum\limits_{s \in S} ul(p, s)$ which means that x' is an Ulam median. □

Theorem 3.0.6 (Median over Hamming Distance)**.** *There exists a deterministic algorithm such that given an input S of strings of equal length n over some alphabet Σ, computes the Hamming median of S in $O(n|S|)$ time.*

Proof. Let `OPT(X)` denote the optimal objective value over the set X. Let S' be the set where all the first characters of every string in S are removed. Let $f_S(c, i)$ denote the number of occurrences of the character c at index i in the set S. The Hamming median problem has the following optimal substructure:

$$\texttt{OPT(S)} = \min_{c \in \Sigma} \left(|S| - f_S(c, 1) + \texttt{OPT(S')} \right)$$

By cut-and-paste argument, suppose that there exists a better solution `OPT*(S')` $<$ `OPT(S')`, then by replacing `OPT(S')` with `OPT*(S')` we get another solution `OPT*(S)` $<$ `OPT(S)`, but this is a contradiction with the optimality of `OPT(S)`. Hence the optimal substructure must hold. From the optimal substructure equation, we can do algebraic manipulation as such:

$$\begin{aligned}\texttt{OPT(S)} &= \min_{c\in\Sigma}(|S| - f_S(c,1)) + \texttt{OPT(S')}\\ &= \left(|S| - \max_{c\in\Sigma} f_S(c,1)\right) + \texttt{OPT(S')}\\ &= \sum_{i=1}^{n}\left(|S| - \max_{c\in\Sigma} f_S(c,i)\right) \text{ (By unrolling the recursion)}\\ &= n|S| - \sum_{i=1}^{n}\max_{c\in\Sigma} f_S(c,i)\end{aligned}$$

Hence, to get the Hamming median, we just need to pick the majority character that occurs at every index $i \in [n]$. Below is a pseudocode that implements the algorithm:

```
def hamming_median(S, n):
  median_string = ""
  for i in range(n):
    dictionary = {}
    maximum_count = 0
    for s in S:
      dictionary[s[i]] += 1
      maximum_count = max(maximum_count, dictionary[s[i]])
    for key, value in dictionary:
      if value == maximum_count:
        median_string += key
        break
  return median_string
```

Listing 3.3: Hamming Median

The algorithm runs in $O(n|S|)$ time as desired. □

Theorem 3.0.7 (Folklore Dynamic Programming for Edit Distance)**.** *Given two strings x, y of length n, m respectively over some alphabet Σ, there exists a $O(mn)$ algorithm to compute the edit distance.*

Proof. Suppose that the last character of x and y is the same, then $ed(x, y) = ed(x[1 : n-1], y[1 : m-1])$. If the last character of x and y are different, then $ed(x, y) =$

$1 + min(ed(x[1 : n - 1], y), ed(x, y[1 : m - 1])$ since we need to spend one operation to either delete the last character of x or the last character of y. We have an optimal substructure on *ed*. We can use Dynamic Programming since there are overlapping subproblems.

Below is the pseudocode of the implementation:

```
def edit_dp(x, y):
  n = len(x)
  m = len(y)
  dp_table = [[0 for _ in range(m + 1)] for _ in range(n + 1)]
  for i in range(n):
    for j in range(m):
      if x[i] == y[j]:
        dp_table[i + 1][j + 1] = dp[i][j]
      else:
        dp_table[i + 1][j + 1] = 1 + min(dp[i - 1][j], dp[i][j - 1])
  return dp_table[n][m]
```

Listing 3.4: Edit DP

The algorithm runs in $O(mn)$ as desired. $\square$

Chapter 4

Approximation Algorithms

In this chapter, we discuss findings that can improve the approximation algorithms that we have seen in chapter 2. For context, the problem that we are considering is when there are exactly four input permutations each of length n and we want to find the Ulam median over those input permutations. Direct application of [CDK21] and Theorem 2.0.3 gives a $\frac{3}{2} - \frac{1}{8} = \frac{11}{8}$-approximate median in $O(n^5)$ time.

Theorem 2.0.3 makes use of a dynamic programming subroutine to compute a median over the Restricted Edit Metric where only character deletions and substitutions are allowed. The median is then carefully adjusted to be a permutation by removing duplicate characters and adding them back in a balanced manner such that they do not increase the distance by much. We shall try to add restrictions to the dynamic programming subroutine in hopes of getting a better approximation ratio.

4.1 Limiting the Number of Duplicate Characters

One interesting assumption that can be made is to limit the number of duplicate characters in the result of the dynamic programming subroutine. We make the assumption that we can find some black-box algorithm that outputs a restricted edit median with $\geq 1 - \epsilon n$ distinct characters. In this case, the number of duplicate characters to be deleted in other strings is $C \leq \epsilon n$.

This idea didn't work well because in the analysis, the bulk of the approximation comes from the total amount of removals as a whole over all the input strings and not between only two strings or characters. The restriction cares about the number of removals at a per-character level instead of on a whole-input level. Following the same analysis in [CDK21] we are only replacing C with ϵn at best and this does not yield any interesting implication or better approximation bound.

4.2 Removing the N Length Restriction

Another interesting restriction is to allow the dynamic programming subroutine to produce a restricted edit median of length $< n$. Assume that we modify the subroutine such that it doesn't have to produce a string of length n. In this case we have the following property:

Lemma 4.2.1. *For every character c in the restricted edit median x', it must be aligned with $\geq \frac{m}{2}$ of the input permutations, where m is the number of input permutations.*

Proof. Suppose that c is aligned in exactly k of all the inputs. Then for the rest of the $m-k$ permutations, we need to move c to the correct position such that it becomes aligned with x'. Each move operation translates to one deletion and one insertion. In this case, we will pay a total cost of $2(m-k)$.

If c doesn't appear at all in x', then we have to pay one deletion for each of the m permutations to delete c in them. Hence in this case we will pay a total cost of m. Since the dynamic programming subroutine chooses to include c in x', we must have

$$2(m-k) \leq m \implies k \geq \frac{m}{2}$$

□

As a corollary, when n is odd, x' should not have duplicate symbols, and when n is even, x' should have at most one duplicate for each symbol. This is because each duplicate will be aligned in $\leq \frac{m}{2}$ input permutations.

Let x' be the modified dynamic programming output and y' be the original dynamic programming output. We have

$$\sum_{i\in[m]} red(x', x_i) \leq \sum_{i\in[m]} red(y', x_i)$$

where $x_1, x_2, \cdots, x_m$ are the input permutations.

The hard part is that adding the missing symbols back to x' and making sure that it doesn't exceed too much compared to $\sum_{i\in[m]} red(y', x_i)$. No good strategies on how to further relate the two have been found.

Chapter 5

Hardness Results

Besides coming up with better approximation algorithms as discussed in the previous chapter, it is also of great interest to determine the computational hardness of the `Ulam Median` problem. In this chapter, several attempts and approaches to finding a reduction chain to try to prove that `Ulam Median` is NP-Hard will be discussed.

5.1 Reduction From LCS

The main motivation for this reduction is due to existing past works that showed that the `Edit Median` problem is NP-Complete. The proof follows from the polynomial time reduction from `Max-Clique` to `LCS` by Jiang and Li and then chained together with the polynomial time reduction from `LCS` to `LCS0` and then from `LCS0` to `Edit Median` [JL95; HC96]. We shall start our finding by reproducing the proof that `LCS` is NP-Hard with more details as Jiang and Li didn't explicitly prove the reduction.

Theorem 5.1.1 (`LCS` is NP-Hard)**.**

Proof. We prove this by a reduction from `Max-Clique` which is known to be NP-Hard [Kar75].

Consider an instance of `Max-Clique`, a graph $G = (V, E)$ where $V = \{v_1, v_2, \cdots, v_n\}$ is the set of vertices and E is the set of edges. Pick any vertex v_i, let $v_{i_1}, v_{i_2}, \cdots, v_{i_q}$ be the neighbours of the vertex v_i, where $i_1 < i_2 < \cdots < i_q$. Let p be the largest index such that $i_p < i$. We construct two sequences as such:

$$x_i = v_{i_1} v_{i_2} \cdots v_{i_p} v_i v_1 v_2 \cdots v_{i-1} v_{i+1} v_{i+2} \cdots v_n$$

$$x'_i = v_1 v_2 \cdots v_{i-1} v_{i+1} \cdots v_n v_i v_{i_{p+1}} v_{i_{p+2}} \cdots v_{i_q}$$

Let $S = \{x_i, x'_i | 1 \leq i \leq n\}$ be the instance of `LCS`.

$(\rightarrow)$ If the graph G has a clique of size k, then let $v_{c_1}, v_{c_2}, \cdots, v_{c_k}$ be the vertices in the clique of size k, where $c_1 < c_2 < \cdots < c_k$. Notice that the sequence

$$V = v_{c_1} v_{c_2} \cdots v_{c_k}$$

is a subsequence of each sequence $s \in S$. Hence the Longest Common Subsequence of S must have length of at least k.

$(\leftarrow)$ If S has a common subsequence of length k, let the common subsequence be

$$V = v_{c_1} v_{c_2} \cdots v_{c_k}$$

Pick any vertex v_{c_i} and consider any v_{c_j} such that $c_j < c_i$. Now consider x_{c_i}, inside that string there's only 2 possible contiguous substring that it can possibly match in, that is:

$$S_1 = v_{c_{i_1}} v_{c_{i_2}} \cdots v_{c_{i_p}}$$

$$S_2 = v_1 v_2 \cdots v_{c_i - 1}$$

Notice that in between S_1 and S_2 in x_{c_i}, there is the character (vertex) v_{c_i}. Suppose now that v_{c_j} matches in S_2, then consider the string x'_{c_i}. v_{c_j} can only possibly appear before v_{c_i} in x'_{c_i} by construction, but by matching with S_2, v_{c_j} appears after v_{c_i} in x_{c_i}. Regardless of where both v_{c_i} and v_{c_j} are in V, we cannot have them both appear in V

as a common subsequence of everything in S as they will not align with one of x_{c_i} or x'_{c_i}. Hence it must be the case that v_{c_j} matches in S_1. We conclude that v_{c_j} must be a neighbour of v_{c_i}.

Using similar reasoning, for any v_{c_j} such that $c_j > c_i$, we can show that it must be matched in the substring $v_{c_{i_{p+1}}} v_{c_{i_{p+2}}} \cdots v_{c_{i_q}}$ in x'_{c_i}. Hence v_{c_j} must also be a neighbour of v_{c_i}. By repeating the argument over all vertex v_{c_i}, we get that V forms a clique of size k in G.

It takes $O(n \log n)$ time for each vertex v_i to produce x_i, x'_i as it needs to traverse its neighbour list and sort them. Overall the reduction runs in $O(n^2 \log n)$ time as we have n vertices in G. This runs in polynomial time with respect to the encoding of G. □

As a corollary of the previous reduction, we have a special instance of `LCS` that is NP-Hard. This special instance has the property that each character appears at most twice in each input string. The high-level idea is to adapt the chain of reductions from `LCS` to `Edit Median` to conclude that `Ulam Median` over strings which allows repetition at most once for each character is NP-Hard.

There are a few reasons why the reductions are hard to adapt to Ulam. The first reason is that we must ensure that the number of occurences of each character in each input strings to be the same. This constraint is needed to have a well-defined Ulam distance. Such a constraint is not needed in the case of Edit. As a corollary, we must also have that each input in Ulam must have the same length, while it is not necessary in the case of Edit.

The other reason is that in Ulam if we try to extend the alphabet Σ and add new characters into Σ, we must ensure that each character have to appear in the input strings as well. Combining both we see that the chain of reductions `LCS` $\leq_p$ `LCS0` $\leq_p$ `Edit Median` is not adaptable (at least directly) to `Ulam Median` as the reduction makes heavy use of alphabet extension and creating shorter strings as inputs.

Chapter 6

Implementation and Empirical Results

In this chapter, we shall discuss various strategies for implementing some of the algorithms seen in earlier chapters. To further bridge the gap between theory and practice, some heuristic algorithms such as local search are also implemented. Finally, we try to see the tightness of the theoretical results by benchmarking with custom datasets.

All the source codes along with the information on how to build and run can be found in the following public repository: `https://github.com/AudreyFelicio/FYP`

6.1 Algorithm Implementation

The algorithms implemented is strictly to solve the `Ulam Median` problem. The input is a set of permutations $S \subseteq S_n$ where $|S| = m$. All the algorithms are implemented using `C++` as it is a systems language which means that it is good for tasks that require fast computations and low-level optimizations.

6.1.1 Bruteforce Solver (`Optimal`)

Bruteforce solver for the optimal permutation under Ulam metric which runs in $O(n \log n * n! * m)$ time. The solver simply iterates over all $n!$ possible permutations and outputs two lines. The first one is the optimal objective value and the second one the optimal permutation that minimizes the total Ulam distance to all m inputs. It terminates within a reasonable time ($\leq$ 2 hour) for $n \leq 10$ and $m \leq 5000$.

To leverage on modern CPU architectures with multicore processors and simultaneous multi-threading, the Bruteforce Solver is further optimized using multi-threading to have a factor of C speedup, where C is the number of logical cores available in the running machine. The multi-threading strategy is to spawn n threads and each thread will iterate through exactly $(n-1)!$ different permutations. Finally all the result from the threads will be aggregated back to the `main()` method and the optimal median will be chosen from the local median of the threads. A Master-Worker pattern is used for this solver.

To allow for novel and clean implementation, the i-th thread spawned will be responsible to iterate through all the permutations from $[i, 1, 2, \cdots, i-1, i+1, \cdots, n]$ to $[i, n, n-1, \cdots, i+1, i-1, \cdots, 1]$. The implementation can be found in the file `bruteforce.cpp`.

6.1.2 Best From Input (`BestInput`)

It is a direct implementation of the pseudocode shown in 3.0.2 which runs in $O(m^2 n \log n)$ time. The factor m^2 is due to iterating over all pairs of input permutations and $n \log n$ is due to computing the Ulam distance between each pair of input permutation. The algorithm outputs 2 lines, the first one containing the objective value of the permutation and the second line containing the best input permutation itself. The implementation can be found in the files `best_from_input.cpp, best_from_input.hpp`.

6.1.3 Theorem 2.0.1 (`RelativeOrder`)

It is a direct implementation of the pseudocode shown in [CDK21]. The implementation runs in $O(nm^2 \log n + n^2 m + n^4)$. The main difference with the pseudocode is that to remove a cycle of minimum length, a BFS based implementation that runs in $O(n^3)$ is used instead of a Floyd-Warshall based approach. The idea behind the BFS based approach is that by doing a single BFS from a vertex v, one can find the cycle of minimum length that contains v. This is because if we remove an edge (u, v) in some minimum length cycle C, then the path that remains from breaking the cycle must be a shortest path from u to v and we are able to find it using BFS since the graph is unweighted. We simply run BFS for every vertex v in the graph to find the global minimum cycle.

The algorithm outputs two lines, the first one indicating the total distance of the output permutation to the input permutations and the second one indicating the approximate median permutation produced. The implementation can be found in the files `relative_order.cpp, relative_order.hpp`.

6.1.4 Theorem 2.0.5 (`ApproxMedian`)

It is a direct implementation of the pseudocode shown in [CDK22], more specifically we implement the `ApproxMedian` algorithm. The implemnetation runs in $O(m^5n^4)$ time. The minimum length cycle removal procedure uses the same BFS idea as the previous `RelativeOrder` algorithm.

The algorithm outputs two lines, the first one indicating the total distance of the output permutation to the input permutations and the second one indicating the approximate median permutation produced. The implementation can be found in the files `approx_median.cpp, approx_median.hpp`.

6.2 Local Search

Some local search strategies are also implemented as a possible way to provide a better approximation algorithm. The state-of-the-art local search based approach has an approximation ratio of $(2.836 + \epsilon)$ for solving the k-median problem over any metric space [Coh+21]. We give a better approximation local search for solving the `Ulam median` problem.

Theorem 6.2.1. *There exists a local search algorithm that given an input $S \subseteq S_n$, outputs a 2-approximate median over the Ulam metric.*

Proof. The idea is to simply start the local search from a 2-approximate permutation. This is possible by running the algorithm `BestInput` to produce the initial candidate solution for the local search. In the end, we take the best between the initial candidate solution and the final candidate solution after the local search algorithm has been stopped. This yields a 2-approximate median. □

We give three implementations of Theorem 6.2.1 each using a different metaheuristic. For generating the neighbors of a candidate solution, a 2-index swap heuristic is used for all of the implementations. More formally, we define 2 permutations p, q as neighbors if they differ in exactly two positions. A permutation has exactly $\frac{n(n-1)}{2}$ neighbors under this definition. The idea of a 2-index swap is borrowed from the 2-exchange neighbourhood of the `Travelling Salesman Problem`.

To not waste computation resources and to leverage on multicore and multithreaded architectures, each implementation runs in parallel C number of local searches where C is the number of logical cores. At least one of those local searches has `BestInput` as the initial candidate hence the 2-approximation is preserved. This is one of the most powerful aspect of local search as it can easily be run in parallel since the tasks are independent of one another.

6.2.1 Gradient Descent

This is an implementation of the naive gradient descent heuristic. This variant of local search simply takes in the neighbor that decreases the objective value the most as the new candidate solution. The process continues until the time limit is up or a local optimum is hit. The implementation can be found in the file `naive_local_search.cpp`.

6.2.2 Tabu Search

This implementation uses the tabu search metaheuristic which high-level idea is produced in the pseudocode below:

```
def tabu_search():
  determine initial candidate solution s
  while temination criteria is False:
    determine set N' of non-tabu neighbours of s
    for s' in N':
      if s' is an improving candidate solution:
        with probability P ignore this and continue to the next
   neighbour
        update tabu table based on s'
        s = s'
        break
```

Listing 6.1: Tabu Search

We define a neighbour as tabu if it is in the tabu table. Essentially if a neighbour is tabu, then we are not allowed to choose that neighbour as the next candidate solution. Each banned neighbour is only kept in the tabu table for a bounded amount of time where a single unit of time is defined as the number of outer while loop iterations it has persisted. The time is determined by a constant called Tabu Tenure. In this implementation, we use a fixed Tabu Tenure value and the time each entry in the tabu table is kept is equal to the Tabu Tenure value. The probability 1 - P of accepting the improving candidate is called the acceptance probability.

By banning recent moves and accepting better neighbours randomly, the local search can reliably escape from a local optimum. The implementation can be found in the file `tabu_search.cpp`.

6.2.3 Iterated Local Search

This implementation uses the iterated local search metaheuristic which high-level idea is produced in the pseudocode below:

```
def iterated_local_search():
  determine an initial candidate solution s
  perform subsidiary local search on s
  While termination criteria is False:
    r = s
    perform perturbation on s
    perform subsidiary local search on s

    if s is worse than r:
      revert s back to r
```

Listing 6.2: Iterated Local Search

For the subsidiary local search, the simple `Gradient Descent` strategy is used. For perturbation strategy, we implemented 2 random swaps where each swap is a random 2-index swap. This perturbation strategy is to model the 4-exchange perturbation (double bridge move) found in highly successful local search for `Travelling Salesman problem`.

The main idea for the perturbation is to make some changes such that it is hard for the subsidiary local search to revert it. This corresponds to "jumping" to a far away candidate solution, which is a good move if we are stuck in a local optimum. The subsidiary local search done next is to hopefully guide the local search to another better local optimum. The implementation can be found in the file `iterated_local_search.cpp`.

6.3 Benchmarks

6.3.1 Data and Results

We give some empirical results of running the non local search implementations and compare them to the optimal objective value. The actual datasets can be found in `datas` folder. Each dataset has the file name `n_m.txt` where n is the length of each permutation and m is the number of permutations. The actual optimal median if it can be found within reasonable time is provided inside the `expected` folder with the same file name.

Constraints	`Optimal`	`BestInput`	`RelativeOrder`	`ApproxMedian`
$n = 3, m = 6$	6	6	6	6
$n = 5, m = 30$	57	59	59	57
$n = 10, m = 10$	43	49	49	49
$n = 10, m = 30$	147	157	157	157
$n = 12, m = 10$	56	61	61	61

To benchmark local search implementations, we compare their outputs to the `BestInput` instead of `Optimal` since their input is order of magnitudes bigger which `Optimal` cannot compute within a reasonable amount of time. Each local search has a maximal time limit of 1 minute. The tabu tenure = 7 and acceptance probability = 0.7 for `Tabu Search`.

Constraints	`BestInput`	`Gradient Descent`	`Tabu`	`Iterated`
$n = 10, m = 100$	540	528	540	526
$n = 10, m = 1000$	5579	5549	5548	5535
$n = 15, m = 100$	911	884	885	878
$n = 15, m = 1000$	9328	9260	9229	9217
$n = 50, m = 100$	3799	3700	3699	3686

6.3.2 Commentary

Since all the inputs are randomly generated, the results for the non local search are expected. It is well known that the expected Longest Increasing Subsequence for a randomly generated permutation of length n is $O(\sqrt{n})$. This means that the distance between the median and an input permutation is expected to be around $\approx n - \sqrt{n}$. `BestInput` indeed provides a very good approximation when the objective value is very large and since the rest of the algorithm (besides `Optimal`) makes use of `BestInput`, the result is dominated by it. This verifies the result of the Chakraborty et. al. [CDK21].

For the local search benchmarks, turns out that `Tabu Search` didn't perform as well as initially projected. It still lacks behind sometimes compared to the naive `Gradient Descent`. Perhaps more parameter tuning needs to be done as the performance heavily relies on a good tabu tenure constant and acceptance probability. On the other hand, as expected `Iterated Local Search` has the best results. It should have the best mechanism of escaping local optimums and hence it can jump around the state space and find better and better local optimums. Sometimes it even stumbled across global optimum in the case when $n = 10$.

Chapter 7

Extensions and Future Work

7.1 Approximation Algorithms

An open extension would be to see if the state-of-the-art of 1.999-approximate can be further extended to yield a better bound. Another possible way to approach this is to come up with a perhaps better and tighter analysis of the current existing algorithms. Empirical statistics in chapter 6 shows that in a lot of cases, the approximation ratio is way better than 1.999. This perhaps shows that the analysis is not the tightest yet and more can be gained from the existing algorithms.

7.2 Hardness Results

An open extension is to find the missing link to adapt the idea discussed in chapter 5 to Ulam. If the link indeed can be established, then another open problem is to find a polynomial time reduction from Ulam median with at most one repetition to original `Ulam Median`. Another noteworthy approach is to try a reduction from the `Feedback Arc Set` problem as NP-Hard result in Kendall's tau metric (which also works on permutations) is shown via reduction from `Feedback Arc Set` [Dwo+01].

7.3 Empirical Results

As an extension to gain better empirical results, one can consider applying other meta-heuristics for local search that are not explored in this thesis, such as Simulated Annealing and Genetic Algorithms. Another open extension is to find some other heuristics to generate the neighbours of a candidate solution. Perhaps some heuristics can be proven to converge faster to a local optimum or can be implemented more efficiently compared to the used 2-index swap heuristic.

References

[Lev65] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Doklady Akademii Nauk SSSR* 164.4 (Jan. 1965), pp. 845–848.

[Kar75] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *The Journal of Symbolic Logic* 40.4 (1975), pp. 618–619.

[Ukk85] Esko Ukkonen. “Algorithms For Approximate String Matching”. In: *Information and Control* 64 (Jan. 1985), pp. 100–118. DOI: 10.1016/S0019-[]9958(85)80046-[]2. URL: https://doi.org/10.1016/S0019-[]9958(85)80046-[]2.

[JL95] Tao Jiang and Ming Li. “On The Approximation Of Shortest Common Supersequences And Longest Common Subsequences”. In: *SIAM Journal of Computing* 24.5 (Oct. 1995), pp. 1122–1139.

[HC96] C. de la Higuera and F. Casacuberta. “Topology of Strings: Median String is NP-Complete”. In: *Theoretical Computer Science* 230 (Apr. 1996), pp. 39–48.

[Gus97] Dan Gusfield. “Algorithms on Strings, Trees, and Sequences”. In: (1997).

[CH98] Richard Cole and Ramesh Hariharan. “Approximate string matching: A simpler faster algorithm”. In: *Proceedings of the 1998 9th Annual ACM SIAM Symposium on Discrete Algorithms* (1998), pp. 463–472.

[Dwo+01] Cynthia Dwork et al. “Rank Aggregation Methods For The Web”. In: *The Tenth International World Wide Web Conference* (2001), pp. 613–622. DOI: 10.1145/371920.372165.

[Nav01] Gonzalo Navarro. “A guided tour to approximate string matching”. In: *ACM computing surveys (CSUR)* 33.1 (2001), pp. 31–88.

[CK06] Moses Charikar and Robert Krauthgamer. “Embedding the Ulam metric into l_1”. In: *Theory of Computing* 2.11 (2006), pp. 207–224. DOI: 10.4086/toc.2006.v002a011. URL: https://theoryofcomputing.org/articles/v002a011.

[Jia+09] Anxiao Jiang et al. “Rank Modulation for Flash Memories”. In: *IEEE Transactions on Information Theory* 55.6 (2009), pp. 2659–2673. DOI: 10.1109/TIT.2009.2018336.

[AN10] Alexandr Andoni and Huy L. Nguyen. “Near-Optimal Sublinear Time Algorithms for Ulam Distance”. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 76–86. DOI: 10.1137/1.9781611973075.8.

[FSM13] Farzad Farnoud, Vitaly Skachek, and Olgica Milenkovic. “Error-Correction in Flash Memories via Codes in the Ulam Metric”. In: *IEEE Transactions on Information Theory* 59.5 (2013), pp. 3003–3020. DOI: 10.1109/TIT.2013.2239700.

[FM14] Farzad Farnoud and Olgica Milenkovic. “Multipermutation codes in the Ulam metric”. In: *2014 IEEE International Symposium on Information Theory*. 2014, pp. 2754–2758. DOI: 10.1109/ISIT.2014.6875335.

[Mar14] J.I. Marden. *Analyzing and Modeling Rank Data*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. CRC Press, 2014. ISBN: 9781482252491. URL: https://books.google.com.sg/books?id=Ug5MDwAAQBAJ.

[Cha+18] Diptarka Chakraborty et al. “Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time”. In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)* (2018), pp. 979–990. DOI: 10.1145/3422823. URL: https://doi.org/10.1145/3422823.

[AN20] Alexandr Andoni and Negev Shekel Nosatzki. “Edit Distance in Near-Linear Time: It’s a Constant Factor”. In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)* (Nov. 2020). DOI: 10.1109/FOCS46700.2020.00096.

[CDK21] Diptarka Chakraborty, Debarati Das, and Robert Krauthgamer. “Approximating the median under the ulam metric”. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 761–775.

[CGJ21] Diptarka Chakraborty, Kshitij Gajjar, and Agastya Vibhuti Jha. “Approximating the Center Ranking Under Ulam”. In: *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021)*. Ed. by Mikołaj Bojańczy and Chandra Chekuri. Vol. 213. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 12:1–12:21. ISBN: 978-3-95977-215-0. DOI: 10.4230/LIPIcs.FSTTCS.2021.12. URL: https://drops.dagstuhl.de/opus/volltexte/2021/15523.

[Coh+21] Vincent Cohen-Addad et al. *An Improved Local Search Algorithm for k-Median*. 2021. arXiv: 2111.04589 [cs.DS].

[CDK22] Diptarka Chakraborty, Debarati Das, and Robert Krauthgamer. *Clustering Permutations: New Techniques with Streaming Applications*. 2022. arXiv: 2212.01821 [cs.DS].

[Cha+22] Diptarka Chakraborty et al. “Fair Rank Aggregation”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 23965–23978. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/974309ef51ebd89034adc64a57e304f2-[]Paper-[]Conference.pdf.