CHAPTER 4

BACKGROUND WORK

4.1 A FINGERPRINT

Skin on human fingertips contains ridges and valleys which together forms distinctive patterns. These patterns are fully developed under pregnancy and are permanent throughout whole lifetime. Prints of those patterns are called fingerprints. Injuries like cuts, burns and bruises can temporarily damage quality of fingerprints but when fully healed, patterns will be restored. Through various studies it has been observed that no two persons have the same fingerprints, hence they are unique for every individual.

![Ridge and Valley Diagram]

Figure 4.1 Finger printing
Especially in law enforcement where they have been used over a hundred years to help solve crime. Unfortunately fingerprint matching is a complex pattern recognition problem. Manual fingerprint matching is not only time consuming but education and training of experts takes a long time. Therefore since 1960s there have been done a lot of effort on development of automatic fingerprint recognition systems. Automatization of the fingerprint recognition process turned out to be success in forensic applications. Achievements made in forensic area expanded the usage of the automatic fingerprint recognition into the civilian applications. Fingerprint Identification is the method of identification using the impressions made by the minute ridge formations or patterns found on the fingertips. No two persons have exactly the same arrangement of ridge patterns, and the patterns of any one individual remain unchanged throughout life. Fingerprints offer an infallible means of personal identification. Other personal characteristics may change, but fingerprints do not. Fingerprints can be recorded on a standard fingerprint card or can be recorded digitally and transmitted electronically to the FBI for comparison. By comparing fingerprints at the scene of a crime with the fingerprint record of suspected persons, officials can establish absolute proof of the presence or identity of a person. Fraud detection is a topic applicable to many industries including banking and financial sectors, insurance, government agencies and law enforcement, and more in banking, fraud can involve using stolen credit cards, forging checks, misleading accounting practices. For example in the banking sector credit card fraud has increased nowadays. Credit applications are Internet or paper based forms with written requests by potential customers for credit cards, loans, and personal loans.

Credit application fraud is a specific case of identity crime, involving identity fraud and real identity theft then to remove duplicates in the credit card applicants. Theft rate should be reduced by Duplicates (or
matches) refer to applications which share common values. There are two types of duplicates: exact (or identical) duplicates have the same values (near or approximate) duplicates have some same values (or characters), some similar values with slightly altered spellings. This paper argues that each successful credit application fraud pattern is represented by a sudden and sharp spike in duplicates within a short time, relative to the established baseline level. Duplicates are hard to avoid from fraudster view because duplicates increases their’ success rate.

The main objective of achieving the resilience by adding two new, real time, data mining-based layers to complement the two existing non data mining layers proposed system utilizes real time data mining-based security layers (CD and SD) for identity crime detection. The first new layer is Communal Detection (CD): the white list-oriented approach on a fixed set of attributes. To complement and strengthen CD, the second new layer is Spike Detection (SD): the attribute-oriented approach on a variable-size set of attributes. The CD and SD layers are continuously updated. Data are traditionally based on a binary representation in which discrete information is assumed (even in continuous data, range representations are possible) and so the operations involve “modifying” bits without concern for any underlying semantics. In dealing with text data, representing the linguistic knowledge is an important issue in which traditional binary coding is insufficient, and so new representation schemes should be investigated. Quality data are highly desirable for data mining and data quality can be improved through the real time removal of data errors (or noise). The detection system has to filter duplicates which have been re-entered due to human error or for other reasons. It also needs to ignore redundant attributes which have many missing values. Fraud detection can be easily found the following way.
Figure 4.2 Architecture diagram for fraud detection

4.1.1 Algorithm Implementation

CD Algorithm

1. Multi-attribute link: More applications are compared using the link types. Multi attribute link score to focus on a single link between two applications, not on matching of attributes between the values.

2. Single-link score value with average score: Average score is being created based on the user input value.

3. Parameter value change: Determine same or new parameter value by comparing inputs.
4. White creating list the valid user details are stored in the white list and others are rejected and new white list is created.

**SD Algorithm**

After processing the information the data are then Send for the SD value calculation then the weights are calculated by comparing inputs.

1. Single-step scaled counts: Determine the value exceeds the time difference between each process.

2. Single value spike detection: Calculate current value score based on weighted scaled match values.

3. SD attribute selection: Each attribute weight is automatically updated at the end of the processing data stream. SD algorithm is the calculation of every current applications score using all values score and attribute weights.

4. CD attribute weights change: At the end of every current discrete data stream process, SD algorithm calculates and updates the attribute weight for CD. CD and SD is CD is provided by attribute weights by SD layer. So both these layers together provides the single score for detection any illegal user and their scores are listed for the identity crime identification (similar values of the user) is done.

4.1.2 Levenshtein Distances

**Normalized edit distance**

The conventional (un-normalized) edit distance is used to find the distance between X and Y and then normalizing this value by the length of the
corresponding editing path. In order to compute normalized edit distances, an algorithm that can be implemented to work in $O(mn^2)$ time and $O(n^2)$ memory space is proposed, where $m$ and $n$ are the lengths of the strings under consideration, and $m \geq n$. The definition of normalized edit distance seems possible in terms of edit transformations. For instance, if $y$ is zero for certain pairs of symbols, then for any two strings $X$, $Y$, there could be infinitely long sequences of elementary edit operations with normalized weight equal zero. The appropriateness of the correctly normalized edit distance versus that of post-normalized or un-normalized edit distances, a practical pattern recognition problem that consists of hand-written digit recognition through edit-distance based nearest-neighbor classification has been considered.

**Weight Function**

The weight function $y$ required for the elementary edit operations was obtained through a (dynamic-programming-based) learning technique known as the Error Correcting Grammatical Inference (ECGI). The ECGI technique gives a probability matrix for substitutions of any pair of symbols of the alphabet, as well as for insertions and deletions of any symbol. This weight function was used to compute edit distances between samples in three different ways: un-normalized, post-normalized, and normalized.

A correct procedure for computing normalized edit distances has been presented with a linear increase in computational complexity with respect to the classical un-normalized edit distance procedure.

1. Levenshtein dialect distance is a method of measuring phonetic distances between Dutch dialects. Levenshtein distance is used as a tool for measuring linguistic distances between language varieties. Levenshtein distance (LD) is a measure of the similarity between two strings. Dialects areas
should be considered as continua and not as areas separated by sharp borders. To validate the Levenshtein distance, the results for Dutch may be impressive, but the Dutch dialect area is a flat, regularly populated landscape. In contrast with this, the Norwegian dialect area is less regular, due to the mountains. This may make the test harder but more revealing.

(2) Dictionary lookup methods are a simple and effective way for approximate string matching. Significant research has been done in the field of Optical Character Recognition (OCR). Dictionary lookup methods take a string as input, and try to find the closest match (es) with entries in the dictionary. This is particularly effective, when the OCR fails to recognize the word correctly, and the string can be taken as input for the dictionary lookup methods. This effort observes the effectiveness of a specific dictionary lookup technique, but in this technique is applied for only two or three words. Within dictionary lookup methods, there are some issues which need to be addressed, such as:

- Value additions to dictionary lookup have resulted in calculating apriority probabilities, which increases the overhead and complexity.
- A short dictionary might not be sufficient to match the word in context.
- A large dictionary increases the cost of searching to an enormous degree.

As the number of words in the dictionary grows, it can be to keep in mind the optimal balance between the size of the dictionary and the
overhead needed to compute, in order to make these dictionary lookup methods more effective. From the semantic web viewpoint, while uploading/downloading scanned manuscripts, this facility can be used as a web service. Dictionary lookup service approach can make the process simpler for the user.

(3) An abundance of vision algorithms which, provided with a sequence of images that have been acquired from sufficiently close successive 3D locations, are capable of determining the relative positions of the viewpoints from which the images have been acquired. In this work, concerning with the problem of determining the relative positions and orientations of the viewpoints corresponding to a set of unordered central panoramic images. This problem is hereafter referred to as unordered panoramic image localization and arises naturally. Most of the existing research on SaM recovery has approached the problem focusing on image sequences. The underlying assumption is that images that have been acquired close in time have viewpoints that are also close in space and, therefore, can be processed by repeatedly applying short baseline algorithms. Forward a novel approach for determining the relative locations and orientations of a set of unordered panoramic images, along with certain structure information regarding their imaged surroundings. Simultaneously localizing method can be readily extended to matching corresponding epipolar curves, thus covering the whole available visual field.

(4) The edit distance is defined as minimum number of edit operation needed to convert one string in to another string there are two methods taken in to consideration they are Levenshtein edit distance and Damerau edit distance. The
Levenshtein edit distance allows three operations they are used to insert, delete and substitution. In Damerau edit distance operations are performed as same in the above edit method it add one operation that is transposition.

Levenshtein edit distance and the Stoilos distance defined in for ontologies. Levenshtein distances defined as the minimum number of elementary operations that are required to transform a string S1 into a string S2. There are three possible transactions: replacing, deleting or adding a character. This measure takes its values in the interval $[0, \infty]$.

**Automatic correction of spelling errors**

Automatic correction of spelling errors is one of the most important areas of natural language processing. Spell checking is to find the word closest to the erroneous word and words in the lexicon. This approach is based on the similarity and the distance between words. This approach associates for each comparison between two words a weight, which is a decimal number and not an integer. This weight allows the better and perfect scheduling solutions proposed by the correcting system of the spelling errors.

**Edit distance algorithms**

There are a number of metrics available to achieve the string matching tasks but the basic metrics are based on ED metrics. Various ED metrics have been developed so far to decrease the penalty for the most possible transcription errors (Jogan & Leonardis 2003, Micusik & Pajdla 2006). The main problem is how to select or combine multiple orthographic measures (Lourakis & Argyros 2004) in order to achieve desired results. The basic EDA’s are based on dynamic programming including Smith-Waterman, Levenshtein Distance and Needleman Wunseh (Micusik & Pajdla 2006).
The dynamic programming algorithms needs \( O (m \times n) \) operations to calculate the edit distance between two strings, where ‘m’ and ‘n’ are the lengths of string1 and string2, respectively. Dynamic programming generates the \((m + 1) \times (n + 1)\) matrix and compute all values of \(D(i, j)\) by using a recursive function and stores the result in a table, where ‘i’ and ‘j’ represents all strings from ‘1’ to \(m/n\).

The LEDA counts the minimum number of edit operations required to transform one string to another (Rousseeuw 1984, Micusik & Pajdla 2006). It is also referred as basic Levenshtein (BLev) EDA. The LEDA allows three basic edit operations as given below:

1) Insert: \(D(i-1, j) + 1\)

2) Delete: \(D(i, j-1) + 1\)

3) Substitute: \(D(i-1, j-1) + \text{Cost}\)

If \(a_i = b_j\) then \(\text{Cost}=0\) and if \(a_i \neq b_j\) then \(\text{Cost}=1\)

Introducing an additional new edit operation, that is, ‘exchange of vowels’ (a, e, i, o, u). This edit operation is proposed to find the most commonly occurring orthographic and typographical errors especially in person names. The ‘exchange of vowels’ edit operation is introduced to account for the most commonly occurring spelling mistakes of vowels due to the converting names from one language to another.

The Damerau-Levenshtein distance metric arose from research by Damerau and Levenshtein on spelling errors. (Damerau 1964, Levenshtein 1966). The Damerau-Levenshtein distance metric is a function, from finite strings drawn from an alphabet, to the integers. It is a distance metric in the sense. That given strings \(s_1, s_2, s_3\), the following conditions apply.
• Non-negativity: \( d(s_1, s_2) \).
• Non-degeneracy: \( d(s_1, s_2) = 0 \) if and only if \( s_1 = s_2 \).
• Symmetry: \( d(s_1, s_2) = d(s_2, s_1) \).
• Triangle Inequality: \( d(s_1, s_2) + d(s_2, s_3) = d(s_1, s_3) \).

The distance \( d(s_1, s_2) \) is defined as follows. One “operation” can be inserting a character, deleting a character, substituting one character for another in one location, or transposing two adjacent characters. There may be many combinations of these four operations that can convert the string \( s_1 \) to \( s_2 \), but the length of the shortest sequence is the distance between the two strings. It should be noted that the four operations can be given weights other than one, but do not consider that here. Damerau showed that 80% of typographical errors are distance 1 in this model (Damerau 1964). A highly related-metric is the Levenshtein metric (Levenshtein 1966), which excludes the transposition, and is frequently used in spelling checkers. Note that a transposition can be thought of as an insertion followed by a deletion, and so transposition errors are still covered by the Levenshtein metric. The Levenshtein metric is much easier to compute, where \( d(a, b) \) can be computed in time \( O(|a||b|) \), with several techniques available (Wagner & Fisher 1974, Ukkonen 1985) for acceleration in the case of calculating many \( d(a, b) \) for the same \( a \) and all \( b \) in a dictionary (as would be the case in this paper’s scheme). However, since transposition errors are probably very likely in entering a password, selecting to use the Damerau-Levenshtein metric instead. The complexity of computing this metric is \( O(|a||b|\max(|a|, |b|)) \) with the naive algorithm, and \( O(|a||b|) \) with the algorithm of Lowrance and Wagner (Lowrance & Wagner 1975). Proving the four metric properties in and discuss the algorithms used to compute the distance and their complexities. Also there are some empirical tricks to accelerate calculation in the special case of this problem.
4.2 IMAGE MATCHING

The Levenshtein Distance

The Levenshtein distance, also known as the edit distance, is a measure of the similarity between two strings of arbitrary lengths [Rishin Haldar and Debajyoti Mukhopadhyay, 2011]. Given a pair of strings referred to as the source (s) and target (t), the LD corresponds to the minimum number of one-step operations (defined as letter deletions, insertions and substitutions), that are necessary to transform s into t. For example, for s=“VISION” and t=“VISITOR”, LD(s; t) = 2 since two changes (i.e., insert ’T’ before ’O’ and substitute ’R’ for ’N’) suffice. The LD can be computed in O(|s|+|t|) time by a dynamic programming technique, known as the Levenshtein algorithm. As a byproduct, this algorithm returns the pairs of letters that have been matched while computing the LD. A property of the Levenshtein algorithms that it preserves the order of matched letters. In other words, if a letter at position i in s matches the letter at position j in t, then letters in s at positions k > i can only match letters in t that are at positions l > j. The LD has been employed in various domains in need of pattern matching, such as spell checking, pattern recognition, speech recognition, information theory, cryptology, bioinformatics, etc. Regarding computer vision, use of the LD has been rather limited and has concerned the comparison of graph structures under edit operations.

4.2.1 Horizon Line Matching

Assuming color images acquired by a central panoramic camera confined to move at a constant height from a planar ground and with its optical axis perpendicular to it. A panoramic image can be unfolded with a polar-to-Cartesian transformation that gives rise to a cylindrical image. Such an image is represented by a rectangular grid whose vertical coordinates axis
corresponds to a longitude that will refer to as the image or viewpoint orientation. It can easily be verified that under the assumed motion model, the vanishing line of the ground plane corresponds to a straight horizontal line (i.e. a line of fixed y-intercept) in the unfolded image, which will hereafter be referred to as the horizon line. Moreover, the assumed camera motion guarantees that in the absence of occlusions, if an environment point projects on the horizon line of one view, then it appears on the horizon of any other view. Stated differently, the epipolar constraint for all points on the horizon of a panoramic image confines them to lie on the horizon line in any other panoramic image acquired under the assumed camera motion. Prior to extracting a horizon line, linear color normalization is performed separately to each color band to account for possible illumination changes. Furthermore, in order to allow for some tolerance in the case that the image plane of the panoramic camera is not exactly parallel to the ground, horizon lines are extracted through convolution with an 1D Gaussian filter of \( \sigma = 2 \), oriented vertically and centered on the line's expected location. Considering the effects camera motion has on the image horizon, pure translation is expected to expand the areas around the focus of expansion (resulting in pixel insertions), shrink areas around the focus of contraction (resulting in pixel deletions) and shift pixels in other locations by an amount dependent on scene structure. Pure rotation is expected to introduce a constant, horizontal shift to all horizon pixels.

General motion will have a combined effect. Pixel substitutions are also expected because of illumination changes, occlusion effects and imaging deformations. Before applying the LD to the comparison of strings consisting of horizon pixels, the costs incurred by each edit operation should be defined. In this work, pixel deletions and insertions are assumed to have unit cost. The cost of a pixel substitution depends on the absolute differences of the RGB components of the pixels being compared. If any of these differences exceeds
a certain threshold, the substitution is assigned a fixed cost of two. Otherwise, the cost of substitution increases proportionally with the sum of the three cubed differences and assumes values in the range [0, 2]. A threshold value of 25 has produced good results in practice. The previous definition allows for some smoothness in the cost of substitutions and assigns low values when replacing pixels whose values differ slightly due to image noise and quantization effects. As defined, the LD compares linear strings that have certain first and last letters. Panoramic horizons, however, are inherently cyclic and their origins in cylindrical images are arbitrary. Had the relative orientations of image viewpoints been known, this could have been remedied by circularly rotating all horizon strings so that their origins corresponded to the same absolute direction. Since the proposed approach does not make any assumption on the relative poses of panoramic views, the LD should be extended to account for the arbitrary linearization of horizons extracted from unfolded panoramic images.

The problem of cyclic sequence matching has attracted considerable interest and several algorithms have been proposed for efficiently solving it [Damien Michel, Antonis A. Argyros and Manolis I.A. Lourakis, 2007]. For the purposes of this work, the technique described next has proved to perform well in practice. The target horizon string is first duplicated next to itself, thus ensuring that it can be matched with the source string without having to wrap around at string ends. Nevertheless, target string duplication introduces a new problem, specifically the possibility that both a target string pixel and its duplicate are matched to different source pixels, thus violating the uniqueness stereo property. To deal with this problem, the source horizon string is repetitively matched with a substring of the duplicated target string that is aligned with the part of the latter yielding the most pixel matches and whose length is being progressively shrunk until it becomes equal to that of the source string. The two sample images of
dimensions that were captured about 50cm apart. Superimposed lines indicate horizon pixel pairs matched between the two views as described above. Typically, the number of pixels matched between two images of this resolution is from 900 to 1000. It is worth pointing out that the order-preserving property of the Levenshtein algorithm automatically enforces the stereo ordering constraint when matching horizon strings, ensuring that the order of matches is preserved along horizon lines.

4.2.2 Angular Alignment of Images

This section is concerned with estimating the relative rotation between two cylindrical images whose optical centers lie on the plane $Z = 0$. The shows two panoramic views at locations $(X_c; Y_c)$ and $(X_c; Y_0)$. At first, recovering the angle that makes the two views parallel. Following this, also interested in recovering the angle - that permits the alignment of both views with the direction of their relative translation. Assume that the horizon lines of the image pair have been matched. The disparities of horizon pixels have two components: The first, which varies from pixel to pixel, depends on the relative translation of the two images and the structure of the environment. The second depends on the relative orientation between the images and is the same for all pixels regardless of the environment. Thus, assuming that the average of positive translational disparities is approximately equal to the average of negative ones, the mean of all disparities approximates the disparity due to rotation. The assumption that positive and negative disparities cancel out boils down to an implicit assumption regarding scene structure. Nevertheless, experimental evidence indicates that this assumption is valid even in settings with considerable depth variations of no particular structure.

A circular shift of the second horizon line with the estimated rotation roughly aligns it with the first. Then, the shifted horizon line is rematched with the first using the Levenshtein algorithm and the
The aforementioned procedure is repeated for refining the estimated rotation. The procedure terminates when the change in the estimated rotation becomes too small. Having canceled the rotation between the two images, the direction of the translational motion of one with respect to the other can be estimated based on the following observation. When a camera moves along a straight path without rotating, horizon pixels move so that positive and negative disparities define two half circles. These half circles are separated by the foci of expansion and contraction, which define the direction of translation. Therefore, for two matched horizon lines with no relative rotation, the two antidiometric points separating the horizon pixels into two groups with opposite disparity signs yield the direction of translation - as the direction of the line passing through them. As will shortly become clear, the angles and achieving the angular alignment of images are needed only when localizing an initial pair of reference images. However, observed that the matches produced by the Levenshtein algorithm are of better quality when the images being matched are aligned. Therefore, prior to computing the final matches for two images, they are aligned by rotational shifting according to their estimated and angles.

A bit-vector algorithm for computing levenshtein and damerau edit distances: The edit distance between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. The Levenshtein edit distance allows three types of operations: an insertion, a deletion or a substitution of a character. The Damereau edit distance allows the previous three plus in addition a transposition between two adjacent characters. To best knowledge the best current practical algorithms for computing these edit distances run in time $O(dm)$ and $O(\sigma+[m/w]n)$, where $d$ is the edit distance between the two strings, $m$ and $n$ are their lengths ($m \leq n$), $w$ is the computer word size and $\sigma$ is the size of the alphabet. In this paper an algorithm that runs in time $O(\sigma + [d/w]m)$. The
structure of the algorithm is such that in practice it is mostly suitable for testing whether the edit distance between two strings is within some pre-determined error threshold. Also presented some initial test results with threshold edit distance computation. The algorithm works faster than the original algorithm of Myers. This paper drawsbacks the oldest, but most flexible in terms of permitting different edit operations and/or edit operation costs, algorithms for computing edit distance are based on dynamic programming and run in time $O(mn)$. This paper Proposed Algorithm that runs in time $O(\sigma + [d/w] m)$. Combining one of the $O(dm)$edit distance algorithms of Ukkonen with the bit-parallel algorithm of Myers to obtain a faster algorithm.

- **Disadvantages:** Lack of comprehensive experimental comparison of the relative performance between different algorithms for computing edit distance.

- **Future Work:** Composing a fairly comprehensive survey on algorithms for computing edit distance. The survey will also include a more comprehensive test with algorithm.

**Levenshtein distance technique in dictionary lookup methods: an improved approach:** Dictionary lookup methods are popular in dealing with ambiguous letters which were not recognized by Optical Character Readers. However, a robust dictionary lookup method can be complex as apriori probability calculation or a large dictionary size increases the overhead and the cost of searching. In this context, Levenshtein distance is a simple metric which can be an effective string approximation tool. After observing the effectiveness of this method, an improvement has been made to this method by grouping some similar looking alphabets and reducing the weighted difference among members of the same group. The results showed marked improvement over the traditional Levenshtein distance technique. This paper
drawbacks. Within dictionary lookup method, there are some issues which need to be addressed, such as:

1) Value additions to dictionary lookup have resulted in calculating apriori probabilities, which increases the overhead and complexity.

2) A short dictionary might not be sufficient to match the word in context.

3) A large dictionary increases the cost of searching to an enormous degree.

This paper proposed Levenshtein distance is a simple metric which can be an effective string approximation tool. After observing the effectiveness of this method, an improvement has been made to this method by grouping some similar looking alphabets and reducing the weighted difference among members of the same group. The results showed marked improvement over the traditional Levenshtein distance technique.

The semantic web viewpoint, while uploading / downloading scanned manuscripts, this facility can be used as a web service.

Websites generate a random sequence of alphanumeric characters which needs to be entered by the user as a means of authentication. Many a times, those characters are unclear, in those cases this approach can make the process simpler for the user.

**Computation of normalized edit distance and applications:** Given two strings X and Y over a finite alphabet, the normalized edit distance between X and Y, \( d( X , Y ) \) is defined as the minimum of \( W(P)/L(P) \), here P is an editing path between X and Y , \( W(P) \) is the sum of the weights of the
elementary edit operations of $P$, and $L(P)$ is the number of these operations (length of $P$). In this paper, it is shown that in general, $d\left( X, Y \right)$ cannot be computed by first obtaining the conventional (un normalized) edit distance between $X$ and $Y$ and then normalizing this value by the length of the corresponding editing path. In order to compute normalized edit distances, an algorithm that can be implemented to work in $O(mn^2)$ time and $O(n^2)$ memory space is proposed, where $m$ and $n$ are the lengths of the strings under consideration, and $m \geq n$. Experiments in hand-written digit recognition are presented, revealing that the normalized edit distance consistently provides better results than both un normalized or post-normalized classical edit distances. In general, $d\left( X, Y \right)$ cannot be computed by first obtaining the conventional (un normalized) edit distance between $X$ and $Y$ and then normalizing this value by the length of the corresponding editing path.

This paper proposed Compute normalized edit distances for handwritten digit recognition, revealing that the normalized edit distance consistently provides better results than both un normalized or post-normalized classical edit distances. Suboptimal techniques are computationally cheaper than the optimal. Future investigation should also address the problem of fast computation of the normalized edit distance.

**Spell-checking queries by combining Levenshtein and Stoilos distances:** Proposed in this paper a simple yet efficient method in order to correct misspellings of queries submitted by users to an online search tool in medicine. In addition to exact phonetic term matching, test two approximate string comparators: the string distance metric of Stoilos and the Levenshtein edit distance. Proposed here to combine them. At a threshold comparator score of 0.2, the normalized Levenshtein algorithm gives the highest recall of 76% but the highest precision 94% is obtained by combining the two distances of Levenshtein and Stoilos. Despite the well-known good
performance of the normalized edit distance of Levenshtein, paper shows that its combination with the Stoilos algorithm improves the results for misspelling correction of user queries. This method may be applied to text documents in Electronic Health Records or clinical documents. Vocabulary is difficult to handle by non-professionals. Misspellings of queries submitted by users to an online search tool in medicine. In addition to exact phonetic term matching, test two approximate string comparators: the string distance metric of Stoilos and the Levenshtein edit distance. Proposed Good performance of the normalized edit distance of Levenshtein, this paper shows that its combination with the Stoilos algorithm improves the results for misspelling correction of user queries. This method may be applied to text documents in Electronic Health Records or clinical documents. Future Work Phonemisation cannot correct all errors: it can only be applied when a query and an entry term of the vocabulary sound alike. However, when there is reversal of characters in the query, it is an error of another type, the sound is not the same and then the similarity distances can be exploited.

Localizing unordered panoramic images using the Levenshtein distance:
This paper proposes a feature-based method for recovering the relative positions of the viewpoints of a set of panoramic images for which no a priori order information is available, along with certain structure information regarding the imaged environment. The proposed approach operates incrementally, employing the Levenshtein distance to deduce the spatial proximity of image viewpoints and thus determine the order in which images should be processed. The Levenshtein distance also provides matches between images, from which their underlying environment points can be recovered. Recovered points that are visible in multiple views permit the localization of more views which in turn allow the recovery of more points. The process repeats until all views have been localized. Periodic refinement of the reconstruction with the aid of bundle adjustment, distributes the
reconstruction errors among images. The method is demonstrated on several unordered sets of panoramic images obtained in an indoor environment. This paper Drawback the problem focusing on image sequences. The underlying assumption is that images that have been acquired close in time have viewpoints that are also close in space and, therefore, can be processed by repeatedly applying short baseline algorithms. When applied to a set of unordered images, SaM(fundamental structure and motion) estimation becomes more challenging since a suitable order for processing images has to be automatically determined. Proposed Operates incrementally, employing the Levenshtein distance to deduce the spatial proximity of image viewpoints and thus determine the order in which images should be processed. The Levenshtein distance also provides matches between images, from which their underlying environment points can be recovered. Recovered points that are visible in multiple views permit the localization of more views which in turn allow the recovery of more points. Future Work Extended to matching corresponding epipolar curves, thus covering the whole available visual field.

**Eliminating Fuzzy Duplicates In Data Warehouses:** The duplicate elimination problem of detecting multiple tuples, which describe the same real world entity, is an important data cleaning problem. Previous domain independent solutions to this problem relied on standard textual similarity functions (e.g., edit distance, cosine metric) between multi-attribute tuples. However, such approaches result in large numbers of false positives if want to identify domain-specific abbreviations and conventions. In this paper, developed an algorithm for eliminating duplicates in dimensional tables in a data warehouse, which are usually associated with hierarchies. Exploits hierarchies to develop a high quality, scalable duplicate elimination algorithm, and evaluate it on real datasets from an operational data warehouse. The problem of detecting and eliminating duplicated data is one of the major problems in the broad area of data cleaning and data quality. The
problem of detecting and eliminating multiple distinct records representing
the same real world entity as the fuzzy duplicate elimination problem.
Duplicate elimination is hard because it is caused by several types of errors
like typographical errors, and equivalence errors equivalence error may result
in several duplicate tuples. Proposed Domain-specific information when
available complements these techniques. Previous domain independent
methods for duplicate elimination rely on textual similarity functions
predicting that two tuples whose textual similarity is greater than a pre-
specified similarity threshold are duplicates. Rely on hierarchies to detect an
important class of equivalence errors in each relation, and to significantly
reduce the number of false positives. Dimensional hierarchies in data
warehouses to develop a high quality, scalable, and efficient algorithm for
detecting fuzzy duplicates in dimensional tables. It is support only single
hierarchies do not support multiple hierarchies for detecting fuzzy duplicates

Efficient Similarity Joins For Near Duplicate Detection: With the
increasing amount of data and the need to integrate data from multiple data
sources, a challenging issue is to find near duplicate records efficiently. In this
paper, it focus on efficient algorithms to find pairs of records such that their
similarities are above a given threshold. Several existing algorithms rely on
the prefix filtering principle to avoid computing similarity values for all
possible pairs of records. Proposed a new filtering techniques by exploiting
the ordering information, they are integrated into the existing methods and
drastically reduce the candidate sizes and hence improve the efficiency.
Experimental results show that the proposed algorithms can achieve up to
2.6x–5x speed-up over previous algorithms on several real datasets and
provide alternative solutions to the near duplicate Web page detection
problem. This paper drawbacks Near duplicate data bear high similarity to
each other, yet they are not bitwise identical. There are many causes for the
existence of near duplicate data: typographical errors, versioned, mirrored, or
plagiarized documents, multiple representations of the same physical object, spam emails generated from the same template. The hypothesis problem might be weakened for problems with a low similarity threshold or with a restricted feature domain. This paper proposed exact similarities join algorithms with application to near duplicate detection. Propose a positional filtering principle, which exploits the ordering of tokens in a record and leads to upper bound estimates of similarity scores. The algorithms provide efficient solutions for an array of applications, such as duplicate Web page detection on the Web. It shows that positional filtering and suffix filtering are complementary to the existing prefix filtering technique. They successfully alleviate the problem of quadratic growth of candidate pairs when the data grows in size.

An Efficient Data Mining For Credit Card Fraud Detection Using Finger Print Recognition: Today there are millions of credit card transactions being processed and mining techniques are highly applied to amount transaction and processing then the data’s are highly skewed. Mining such massive amounts of data requires highly efficient techniques that scaled that can be extend transactions are legitimate than fraudulent fraud detection systems were widely used but this document gives the detection techniques. This paper contains multilayered techniques for providing the security for the credit card frauds. The first layer is communal detection and second is Spike detection layers that highly provides security for detection of frauds like probe resistant and mark the illegal user through their input details and mark it in a list. Then it removes attacks like defense in depths on cards and by removing the data redundancy of the attributes and it is being processed with millions of the credit cards. This paper drawbacks in the banking sector credit card fraud has increased nowadays. Credit applications are Internet or paper based forms with written requests by potential customers for credit cards, loans, and personal loans. Credit application fraud is a specific case of
identity crime, involving identity fraud and real identity theft then to remove duplicates in the credit card applicants. Theft rate should be reduced by Duplicates (or matches) refer to applications which share common values. This paper Proposed To achieve resilience by adding two new, real time, data mining-based layers to Complement the two existing non data mining layers Proposed system utilizes real time data mining- based Security layers (cd and sd) for identity crime Detection. a real-time credit application fraud detection system With the help of the finger print biometric process fraud detection can be easily found then data mining with accuracy in the existing applicants with new ones so frauds can be traced and they can be ignored without getting the card.

**Improving Duplicate Elimination In Storage Systems:** Elimination of redundant data has become a critical concern in the design of storage architectures. Content addressable storage engines eliminate data at the block level by mapping data blocks with the same content to the same physical storage location. Intelligent object partitioning techniques leverage block level content addressing in order to improve duplicate elimination. In this paper, proposed a novel object partitioning technique- fingerdiff that is designed to improve storage consumption of existing object partitioning techniques while at the same time reducing associated costs. Presented a detailed evaluation of fingerdiff and other existing object partitioning schemes, and shows that fingerdiff meets its design goals as it improves the effectiveness of block level duplicate elimination while reducing overhead costs. This paper drawback To improve storage consumption of CDC will result in greater number of chunks and smaller chunk sizes. As storage hardware moves towards increasing disk block sizes in order to improve system throughput, it is unreasonable to reduce chunk sizes in order to improve duplicate elimination alone. Further increasing the number of chunks impose metadata overheads on the storage engine. These observations
motivated us to discover a chunking technique that would improve duplicate elimination over existing CDC techniques without increasing its overheads. This paper Proposed To overcome this problem a new chunking algorithm fingerdiff that is designed to improve upon the storage consumption of CDC while lowering the overheads it imposes on the storage system. Finger diff is able to minimize the size of new chunks introduced with every version, while keeping the average size of all blocks relatively large. This in turn allows it to provide the best storage consumption for the least overhead costs.

A Single-Chip Fingerprint Sensor And Identifier: The popular Biometric used to authenticate a person is Fingerprint which is unique and permanent throughout a person’s life. A minutia matching is widely used for fingerprint recognition and can be classified as ridge ending and ridge bifurcation. In this paper projected Fingerprint Recognition using Minutia Score Matching method (FRMSM). For Fingerprint thinning, the Block Filter is used, which scans the image at the boundary to preserves the quality of the image and extract the minutiae from the thinned image. The false matching ratio is better compared to the existing algorithm. This paper drawback A good quality fingerprint contains 25 to 80 minutiae depending on sensor resolution and finger placement on the sensor. The false minutiae are the false ridge breaks due to insufficient amount of ink and cross-connections due to over inking. It is difficult to extract reliably minutia from poor quality fingerprint impressions arising from very dry fingers and fingers mutilated by scars, scratches due to accidents, injuries.

Proposed Fingerprint Recognition using Minutia Score Matching method. Fingerprint matching is used in FRMSM. The pre-processing the original fingerprint involves image binarization, ridge thinning, and noise removal. Fingerprint Recognition using Minutia Score Matching method is
used for matching the minutia points. The proposed method FRMSM gives better FMR values compared to the existing method.

4.3 DISTANCE METHOD

4.3.1 Damerau–Levenshtein Distance

In the Damerau–Levenshtein distance is a "distance" between two string i.e., finite sequence of symbols, given by counting the minimum number of operations needed to transform one string into the other, where an operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters. In the seminal paper, Damerau not only distinguished these four edit operations but also stated that they correspond to more than 80% of all human misspellings. Damerau's paper considered only misspellings that could be corrected with at most one edit operation. The corresponding edit distance, i.e., dealing with multiple edit operations, known as the Levenshtien distance was introduced by Levenshtein, but it did not include transpositions in the set of basic operations. The name Damerau–Levenshtein distance is used to refer to the edit distance that allows multiple edit operations including transpositions, although it is not clear whether the term Damerau–Levenshtein distance is sometimes used in some sources as to take into account non-adjacent transpositions or not.

4.3.2 Euclidean Distance

In the Euclidean distance or Euclidean metric is the "ordinary" distance between two points that one would measure with a ruler, and is given by the pythagorean formula. By using this formula as distance, Euclidean space becomes a metric space. The associated norm is called the euclidean matric. Older literature refers to the metric as Pythagorean metric.
The Euclidean distance between points $p$ and $q$ is the length of the line segment connecting them ($\overline{pq}$).

In cartesian co-ordinate, if $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ are two points in Euclidean $n$-space, then the distance from $p$ to $q$, or from $q$ to $p$ is given by:

$$d(p, q) = d(q, p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$ (4.1)

The position of a point in a Euclidean $n$-space is a Euclidean vector. So, $p$ and $q$ are Euclidean vectors, starting from the origin of the space, and their tips indicate two points. The euclidean norm, or Euclidean length, or magnitude of a vector measures the length of the vector:

$$||p|| = \sqrt{p_1^2 + p_2^2 + \cdots + p_n^2} = \sqrt{p \cdot p}$$ (4.2)

where the last equation involves the dot product.

A vector can be described as a directed line segment from the origin of the Euclidean space (vector tail), to a point in that space (vector tip). Consider if that its length is actually the distance from its tail to its tip, it becomes clear that the Euclidean norm of a vector is just a special case of Euclidean distance: the Euclidean distance between its tail and its tip.

The distance between points $p$ and $q$ may have a direction (e.g. from $p$ to $q$), so it may be represented by another vector, given by

$$q - p = (q_1 - p_1, q_2 - p_2, \ldots, q_n - p_n)$$ (4.3)
In a three-dimensional space \((n=3)\), this is an arrow from \(p\) to \(q\), which can be also regarded as the position of \(q\) relative to \(p\). It may be also called a displacement vector if \(p\) and \(q\) represent two positions of the same point at two successive instants of time.

The Euclidean distance between \(p\) and \(q\) is just the Euclidean length of this distance (or displacement) vector:

\[
||q-p|| = \sqrt{(q-p)\cdot(q-p)}
\]  \(\text{(4.4)}\)

which is equivalent to equation 1, and also to:

\[
||q-p|| = \sqrt{||p||^2 + ||q||^2 - 2p\cdot q}
\]  \(\text{(4.5)}\)

### 4.3.3 Jaro–Winkler Distance

The Jaro distance \(d_j\) of two given strings \(s_1\) and \(s_2\) is

\[
d_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3} \left( \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) & \text{Otherwise} \\ \end{cases}
\]  \(\text{(4.6)}\)

where:

- \(M\) is the number of matching characters (see below);
- \(t\) is half the number of transpositions (see below).

Two characters from \(s_1\) and \(s_2\) respectively, are considered matching only if they are the same and not farther than \(\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1\)
Each character of $s_1$ is compared with all its matching characters in $s_2$. The number of matching (but different sequence order) characters divided by 2 defines the number of transpositions. For example, in comparing CRATE with TRACE, only 'R', 'A', 'E' are the matching characters, i.e. $m=3$. Although 'C', 'T' appear in both strings, they are farther than 1, i.e., floor $(5/2)-1=1$. Therefore, $t=0$. In DwAyNE versus DuANE the matching letters are already in the same order D-A-N-E, so no transpositions are needed.

Jaro–Winkler distance uses a prefix scale $p$ which gives more favourable ratings to strings that match from the beginning for a set prefix length $l$. Given two strings $s_1$ and $s_2$, their Jaro–Winkler distance $d_w$ is:

$$d_w = d_j + (lp(1-d_j))$$

(4.7)

where:

- $D_j$ is the Jaro distance for strings $s_1$ and $s_2$
- $l$ is the length of common prefix at the start of the string up to a maximum of 4 characters
- $p$ is a constant scaling factor for how much the score is adjusted upwards for having common prefixes. $p$ should not exceed 0.25, otherwise the distance can become larger than 1. The standard value for this constant in Winkler's work is $p = 0.1$

Although often referred to as a distance metric, the Jaro–Winkler distance is actually not a metric in the mathematical sense of that term because it does not obey the triangle inequality.
In some implementations of Jaro-Winkler, the prefix bonus $lp(1-d_j)$ is only added when the compared strings have a Jaro distance above a set "boost threshold" $h_i$. The boost threshold in Winkler's implementation was 0.7.

$$d_v = \begin{cases} d_j & \text{if } d_j < h_i \\ d_j + (lp(1-d_j)) & \text{otherwise} \end{cases}$$

(4.8)

### 4.3.4 Taxicab Geometry

The taxicab distance, $d_h$, between two vectors $p,q$ in an $n$-dimensional real vector space with fixed cartesian coordinate system, is the sum of the lengths of the projections of the line segment between the points onto the coordinate. More formally,

$$d_h(p-q) = ||p-q||_1 = \sum_{i=1}^{n} |p_i-q_i|$$

(4.9)

where

$$p=(p_1, p_2, ..., p_n) \text{ and } q=(q_1, q_2, ..., q_n)$$

are vectors.

For example, in the plane, the taxicab distance between $(p_1, p_2)$ and $(p_3, p_4)$ is $|p_1-p_3| + |p_2-p_4|$.

### 4.3.5 Levenshtein Distance

**Levenshtein** created a distance algorithm. This tells us the number of edits needed to turn one string into another. With Levenshtein distance, measure the similarity and match approximate strings with fuzzy logic. The Levenshtein distance is a **string metric** for measuring the difference between
two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertion, deletion, substitution) required to change one word into the other. The phrase edit distance is often used to refer specifically to Levenshtein distance. It is named after Vladimir Levenshtein, who considered this distance in 1965. It is closely related to pairwise string alignments.

**Edit Distance**

- The minimum edit distance between two strings
- Is the minimum number of editing operations
  - Insertion
  - Deletion
  - Substitution
- Needed to transform one into the another form.

Levenshtein Distance (LD) is a measure of the similarity between two strings, the source string (s) and the target string (t). The distance is the number of deletions, insertions, or substitutions required to transform s into t. The greater the Levenshtein distance, the more different the strings are. In case, the source string is the input, and the target string is one of the entries in the dictionary. Intuitively "GUMBO" can be transformed into "GAMBOL" by substituting "A" for "U" and adding "L" (one substitution and one insertion = two changes).

Algorithms for calculating transposition invariant distances (Hamming distance, longest common subsequence, edit distance) between strings are also given in (Their transposition invariant minimum edit distance between 'a' and y, where x and y are melodic strings and is x transposed by t)
Definition

Mathematically, the Levenshtein distance between two strings a,b is given by $\text{lev}_{a,b}(|a|,|b|)$ where

$$
\text{lev}_{a,b}(i,j)=
\begin{cases}
\max(i,j) & \text{if min}(i,j)=0, \\
\text{lev}_{a,b}(i-1,j)+1 & \text{if } a_i \neq b_j \\
\min \left\{ \begin{array}{l}
\text{lev}_{a,b}(i-1,j)+1 \\
\text{lev}_{a,b}(i,j-1)+1 \\
\text{lev}_{a,b}(i-1,j-1)+|a_i \neq b_j|
\end{array} \right. & \text{Otherwise}
\end{cases}
\tag{4.10}
$$

Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

Example

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. kitten $\rightarrow$ sitten (substitution of "s" for "k")
2. sitten $\rightarrow$ sittin (substitution of "i" for "e")
3. sittin $\rightarrow$ sitting (insertion of "g" at the end).

Instead of using dynamic programming, bit-parallelism can be employed to compare strings with at most n errors. Bit-parallel algorithms simulate classical string matching algorithms, but use bit-masks to store the number of errors allowed. Bit-parallel algorithms have the advantage of...
having faster execution times, but the disadvantage of limiting of the
maximum number of allowable errors to the word size of the bit-masks used

The following things should be noted:

- Only the first MAX_LEN characters are considered. Long
  strings differing at the end will therefore seem to match better
  than they should. A penalty is awarded if strings are shortened

- The calculation can stop prematurely as soon as it realise that
  the supplied minimum required similarity cannot be reached.
  Strings with widely different lengths give the opportunity for
  this shortcut. This is by definition of the Levenshtein distance:
  the distance will be at least as much as the difference in string
  length. Similarities lower than the supplied minimum (or the
default) should therefore not be considered authoritative

The Levenshtein algorithm calculates the least number of edit
operations that are necessary to modify one string to obtain another string.
The most common way of calculating this is by the dynamic programming
approach. A matrix is initialized measuring in the (m, n)-cell the Levenshtein
distance between the m-character prefix of one with the n-prefix of the other
word. The matrix can be filled from the upper left to the lower right corner.
Each jump horizontally or vertically corresponds to an insert or a delete,
respectively. The cost is normally set to 1 for each of the operations. The
diagonal jump can cost either one, if the two characters in the row and column
do not match or 0, if they do. Each cell always minimizes the cost locally.
This way the number in the lower right corner is the Levenshtein distance
between both words.
The Levenshtein method is static, can also improve performance, avoiding callvirt instructions.

It is stateless, which means it doesn't store instance data. It therefore can be put in a static class. Static classes are easier to add to new projects than separate methods. Advanced Levenshtein distance algorithm, and the best is $O(n^*m)$ where $n$ and $m$ are the lengths of the two strings.

Applications will, in most cases, use implementations which use heap allocations sparingly, in particular when large lists of words are compared to each other. The following remarks indicate some of the variations on this and related topics:

- Most implementations use one- or two-dimensional arrays to store the distances of prefixes of the words compared. In most applications the size of these structures is previously known. This is the case, when, for instance the distance is relevant only if it is below a certain maximally allowed distance (this happens when words are selected from a dictionary to approximately match a given word). In this case the arrays can be preallocated and reused over the various runs of the algorithm over successive words.

- Using a maximum allowed distance puts an upper bound on the search time. The search can be stopped as soon as the minimum Levenshtein distance between prefixes of the strings exceeds the maximum allowed distance.

- Deletion, insertion, and replacement of characters can be assigned different weights. The usual choice is to set all three weights to 1. Different values for these weights allows for
more flexible search strategies in lists of words. Edit distance, also known as Levenshtein distance or evolutionary distance is a concept from information retrieval and it describes the number of edits (insertions, deletions and substitutions) that have to be made in order to change one string to another. It is the most common measure to expose the dissimilarity between two strings (Levenshtein 1966).

The edit distance \( ed(x, y) \) between strings \( x=x_1...x_m \) and \( y=y_1...y_n \), where \( x, y \in \Sigma^* \) is the minimum cost of a sequence of editing steps required to convert \( x \) into \( y \). The alphabet \( \Sigma \) of possible characters \( ch \) gives \( \Sigma^* \), the set of all possible sequences of \( ch \)\( \in \Sigma \). Edit distance can be calculated using dynamic programming. Dynamic programming is a method of solving a large problem by regarding the problem as the sum of the solution to its recursively solved sub problems. Dynamic programming is different to recursion however. In order to avoid recalculating the solutions to sub problems, dynamic programming makes use of a technique called memorization, whereby the solutions to sub problems are stored once calculated, to save recalculation.

To compute the edit distance \( ed(x, y) \) between strings \( x \) and \( y \), a matrix \( M_{1...m+1,1...n+1} \) is constructed where \( M_{i,j} \) is the minimum number of edit operations needed to match \( x[i...i] \) to \( y[j...j] \). Each matrix element \( M_{i,j} \) is calculated.

Proof: As mentioned earlier, the invariant is that can transform the initial segment \( s[1..i] \) into \( t[1..j] \) using a minimum of \( d[i,j] \) operations. This invariant holds since:
• It is initially true on row and column 0 because \( s[1..i] \) can be transformed into the empty string \( t[1..0] \) by simply dropping all \( i \) characters. Similarly, \( s[1..0] \) to \( t[1..j] \) by simply adding all \( j \) characters.

• If \( s[i] = t[j] \), transform \( s[1..i-1] \) to \( t[1..j-1] \) in \( k \) operations, then do the same to \( s[1..i] \) and just leave the last character alone, giving \( k \) operations.

• Otherwise, the distance is the minimum of the three possible ways to do the transformation:
  - Transform \( s[1..i] \) to \( t[1..j-1] \) in \( k \) operations, then simply add \( t[j] \) afterwards to get \( t[1..j] \) in \( k+1 \) operations (insertion).
  - Transform \( s[1..i-1] \) to \( t[1..j] \) in \( k \) operations, then remove \( s[i] \) and then do the same transformation, for a total of \( k+1 \) operations (deletion).
  - It transform \( s[1..i-1] \) to \( t[1..j-1] \) in \( k \) operations, then do the same to \( s[1..i] \), and exchange the original \( s[i] \) for \( t[j] \) afterwards, for a total of \( k+1 \) operations (substitution).

• The operations required to transform \( s[1..n] \) into \( t[1..m] \) is of course the number required to transform all of \( s \) into all of \( t \), and so \( d[n, m] \) holds the result.

This proof fails to validate that the number placed in \( d[i, j] \) is in fact minimal, this is more difficult to show, and involves an argument by contradiction in \( d[i, j] \) is smaller than the minimum of the three, and use this to show one of the three is not minimal.
Possible modifications

Possible modifications to this algorithm include:

- Algorithm to use less space, \( \mathcal{O}(\min(n,m)) \) instead of \( \mathcal{O}(nm) \), since it only requires that the previous row and current row be stored at any one time.

- Number of insertions, deletions, and substitutions separately, or even the positions at which they occur, which is always \( j \).

- Normalize the distance to the interval \([0,1]\).

- Smaller than a threshold \( k \), then it suffices to compute a diagonal stripe of width \( 2k+1 \) in the matrix. In this way, the algorithm can be run in \( \mathcal{O}(k) \) time, where \( l \) is the length of the shortest string.

- Give different penalty costs to insertion, deletion and substitution and also give penalty costs that depend on which characters are inserted, deleted or substituted.

- By initializing the first row of the matrix with 0, the algorithm can be used for fuzzy string search of a string in a text. This modification gives the end-position of matching substrings of the text. To determine the start-position of the matching substrings, the number of insertions and deletions can be stored separately and used to compute the start-position from the end-position.

- This algorithm parallelizes poorly, due to a large number of data dependencies. However, all the cost values can be computed in parallel, and the algorithm can be adapted to
perform the minimum function in phases to eliminate dependencies.

- By examining diagonals instead of rows, and by using lazy evaluation, it can be find the Levenshtein distance in $O(m(1+d))$ time (where $d$ is the Levenshtein distance), which is much faster than the regular dynamic programming algorithm if the distance is small.

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + \begin{cases} 2; & \text{if } S_t(i) \neq S_r(j) \\ 0; & \text{if } S_t(i) = S_r(j) \end{cases} \end{cases}$$ \hspace{1cm} (4.11)

The levenshtein function takes two words and returns how far apart they are. It's an $O(N \times M)$ algorithm, where $N$ is the length of one word, and $M$ is the length of the other.

But comparing two words at a time isn't useful. Usually find the closest matching words in a whole dictionary, possibly with many thousands of words. It uses the file /usr/share/dict/words. The first argument is the misspelled word, and the second argument is the maximum distance. It will print out all the words with that distance, as well as the time spent actually searching.

$$T = \{x_i - y_j : 1 \leq i \leq m, 1 \leq j \leq n\}$$, in other words, the set of all possible values for $t$ which would result in an alignment between $x$ and $y$. In order to calculate $ed(x', y)$, they propose using a brute force approach by calculating $ed(x + t, y)$ for all $T$. This is similar to the approach described. This obviously increases the computational complexity of the algorithm over a straightforward edit distance calculation between the two strings. In order to
speed up the calculation, they propose using sparse dynamic programming. Sparse dynamic programming was introduced in. The main idea behind these techniques is that only elements in a string associated with a match are visited. In order to achieve this, the authors propose calculating an ordered set of matching elements in $x'$ and $y$ for every value of $l$ such that $M_l = \{(i, j) \mid x_i + l = y_j\}$. Using sparse dynamic programming, the computational complexity of the transposition invariant edit distance algorithm is $O(mn \log n)$ compared to $O(mn)$ for standard edit distance. They also present a measure called "Longest Common Hidden Melody", which is a transposition invariant version of the longest common subsequence measure.

The important things to know are that it fills in a N x M sized table, like this one, and the answer is in the bottom-right square.

**Table 4.1 Transposition**

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>a</th>
<th>T</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

But wait, what's it going to do when it moves on to the next word *after* cat? In dictionary, that's "cats" so here it is:

**Table 4.2 N×M matrix for transposition**

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>a</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Only the last row changes: It can be avoided that a lot of work if it can
process the words in order, so never need to repeat a row for the same prefix
of letters. The trie data structure is perfect for this. A trie is a giant tree, where
each node represents a partial or complete word. Here's one with the words
cat, cats, catacomb, and catacombs in it. Nodes that represent a word are
marked in black.

Figure 4.3 String matching

Applications

In approximate string matching, the objective is to find matches for
short strings in many longer texts, in situations where a small number of
differences is to be expected. The short strings could come from a dictionary,
for instance. Here, one of the strings is typically short, while the other is
arbitrarily long. This has a wide range of applications, for instance, spell checkers, correction systems for optical character recognition, and software to assist natural language translation based on translation memory. The Levenshtein distance can also be computed between two longer strings, but the cost to compute it, which is roughly proportional to the product of the two string lengths, makes this impractical. Thus, when used to aid in fuzzy string searching in applications such as record linkage, the compared strings are usually short to help improve speed of comparisons. Relationship with other edit distance metrics. There are other popular measures of edit distance, which are calculated using a different set of allowable edit operations. For instance,

- the Damerau–Levenshtein distance allows insertion, deletion, substitution, and the transposition of two adjacent characters;

- the longest common subsequence metric allows only insertion and deletion, not substitution;

- the Hamming distance allows only substitution, hence, it only applies to strings of the same length.

Edit distance is usually defined as a parametrizable metric calculated with a specific set of allowed edit operations, and each operation is assigned a cost (possibly infinite).

The following shortcomings have been identified:

- **Cases sensitivity**: 'E' and 'e' are considered different characters and according differ as much as 'z' and 'e'. This is not ideal, as case differences should be considered less of a difference.
• **Diacritics:** ‘č’ and ‘ě’ are considered different characters and according differ as much as ‘ç’ and ‘é’. This is not ideal, as missing diacritics could be due to small input errors, or even input data that simply do not have the correct diacritics.

• **Similar but different words:** Words that have similar characters, but are different, could increase the similarity beyond what is wanted. The sentences “It is though.” and “It is dough.” differ markedly semantically, but score similarity of almost 85%. A possible solution is to do an additional calculation based on words, instead of characters.

**Modifications made in the algorithm:**

• Algorithm to use less space, \(O(\min(n,m))\) instead of \(O(nm)\), since it only requires that the previous row and current row be stored at any one time.

• Normalize the distance to the interval \([0,1]\).

• Smaller than a threshold \(k\), then it suffices to compute a diagonal stripe of width \(2k+1\) in the matrix. In this way, the algorithm can be run in \(O(kl)\) time, where \(l\) is the length of the shortest string.

• Given penalty costs to insertion, deletion and substitution and none for no such operations

• By initializing the first row of the matrix with 0, the algorithm can be used for fuzzy string search of a string in a text. This modification gives the end-position of matching substrings of the text. To determine the start-position of the matching substrings, the number of insertions and deletions can be
stored separately and used to compute the start-position from the end-position.

- This algorithm parallelizes poorly, due to a large number of data dependencies. However, all the cost values can be computed in parallel, and the algorithm can be adapted to perform the minimum function in phases to eliminate dependencies.

- By examining diagonals instead of rows, and by using lazy evaluation, it can be find the Levenshtein distance in \( O(m(1 + d)) \) time (where \( d \) is the Levenshtein distance), which is much faster than the regular dynamic programming algorithm if the distance is small.

\[
D(i, j) = \min \begin{cases} 
    D(i - 1, j) + 1 \\
    D(i, j - 1) + 1 \\
    D(i - 1, j - 1) + \begin{cases} 
        1, & \text{if } S_1(i) \neq S_2(j) \\
        0, & \text{if } S_1(i) = S_2(j)
    \end{cases}
\end{cases} \tag{4.12}
\]