MINISTERE DE L'INTERIEUR

DIRECTION DES SYSTÈMES D'INFORMATION ET DE COMMUNICATION (DSIC)
Ministère de l'Intérieur

MatchID
Lenvenshtein Automaton

Author:
Manon RIVOIRE

Mentor:
Fabien ANTOINE

May 3, 2019

Contents

Chapter 1

Objectives of the Internship

1.1 Main Objective

The main objective of this internship is to be able to add to MatchID [1], a finite state automaton of Levenshtein, more efficient than the one currently in place in Python. In a specific way to MatchID, the vocabulary of the automaton is mainly composed of name and surname, or of places and birth countries, or even of birth dates.

The aim would be to manage to carry out a fuzzy indexation on the whole set of fields, by, on the one hand storing the finite states automaton in each field and on in other hand by composing them with finite-states match-multi-fields automata for instance in the aim to allow the absence of second and third surnames or even the inversion and confusion between the fields "places" and birth countries.

The finite states automata are supposed to be able to index a large number of recordings (100M) but only with a small number of states (4 fields with the Levenshtein distances).

The stakes of the performance :

- Loading of a 100M fields vocabulary even if the vocabulary is only 100k (eg. column of names x user ID).
- Factorizing the automaton.
- Carrying out the product of the automaton with the serie of names of another source in order to make a fuzzy joint, as quick as possible.

1.2 Levenshtein Finite-State Automata

- Make an inventory of all the Finite-Automata in C or Python librairies.
- Find an efficient library able to on the one hand, treat finite states automata and on the other hand to compile (factorization).

The simple delivery in C of the Lucen / Lucene Levenshtein automaton would be excellent.

1.3 Theoretical Formulation of the Problem

1. Formulate the problem of the finite state automaton the problem of the mono-word Levenshtein finite states automaton merging.

2. Gneralize this to a mon-word Levenshtein finite states automaton merging to a multi-word Levenshtein finite states automaton merging (where the automaton would model the inversions between fields, the optional character and so on...).

Chapter 2

Essential Notions

2.1 Levenshtein Distance Or Edit Distance

What is the Levenshtein Distance or Edit Distance ?

[12] "In Information Theory, Linguistics and Computer Sciences, 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 (insertions, deletions or substitutions) required to change one word into the other."

"It is named after the Societ Mathematician Vladimir Levenshtein, who considered this distance in 1965."

[12] For example, the Lenvenshtein 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 3 edits.

1. Kitten $\rightarrow$ Sitten (substitution of "s" for "k")
2. Sitten $\rightarrow$ Sittin (substitution of "e" for "i")
3. Sittin $\rightarrow$ Sitting (insertion of "g" at the end)

2.2 Levenshtein Automaton

What is a Levenshtein Automaton ?

[11] "In Computer Science, a Levenshtein Automaton for a string w and a number n is a Finite State Automaton that can recognize the set of all strings whose Levenshtein Distance from w is at most n. That is a string x is in the formal language recognized by the Levenshtein Automaton if and only if x can be transformed into w by at most n single-character transformations (insertions, substitutions, deletions)."

NFA = Non Deterministic Finite Automata

2.3 Burkhard Keller Tree Or BK Tree

[4] "A BK Tree or Burkhard Keller Tree is a data structure that is used to perform check based on Edit Distance (Levenshtein Distance) concept. BK Trees are also used for approximate string matching."

Chapter 3

Levenshtein Distance Calculation

3.1 Levenshtein Algorithm for Distance Calculation

As previously explained on [12], "the Levenshtein Distance is a string metric for measuring the differences between two sequences. Informally, the Levenshtein Distance between two words is the minimum number of single-character edits (insertions, substitutions or deletions) required to change one word into the other."

3.1.1 Algorithm

In order to compute the Levenshtein Distance, several algorithms have been developed. I have more particularly studied one of them which consists in the calculation of the Levenshtein Distance between two words that is the number of transformations that we have to perform on one word to obtain the other. In this aim, we build a grid where the columns correspond to each letter of the hypothesis word and the rows correspond to each letter of the reference word. If the number of letters of the hypothesis word is equal to m and the number of letters of the reference word is equal to n then the obtained matrix used to compute the Levenshtein distance between the two words is of size $(n+1)(m+1)$.

A different weighting is assigned to each transformation :

- Let w_1 be the weighting specific to the substitution transformation.
- Let w_2 be the weighting specific to the insertion transformation.
- Let w_3 be the weighting specific to the deletion transformation.
- Let i be the index which crosses the rows Levenshtein matrix.
- Let j be the index which crosses the columns Levenshtein matrix.

Init The first row and the first column respectively correspond to the number of the row i or the number of the column j multiplied by the weight of the transformation. $d(i,0) = i * w_2 \ \forall i$, $d(0,j) = j * w_2 \ \forall j$ (or w_3 ?).

Once we have filled the first row and the first column, we have to fill the center of the grid by following some rules.

Rule 0 We cross the grid from the top to the bottom and from the left to the right.

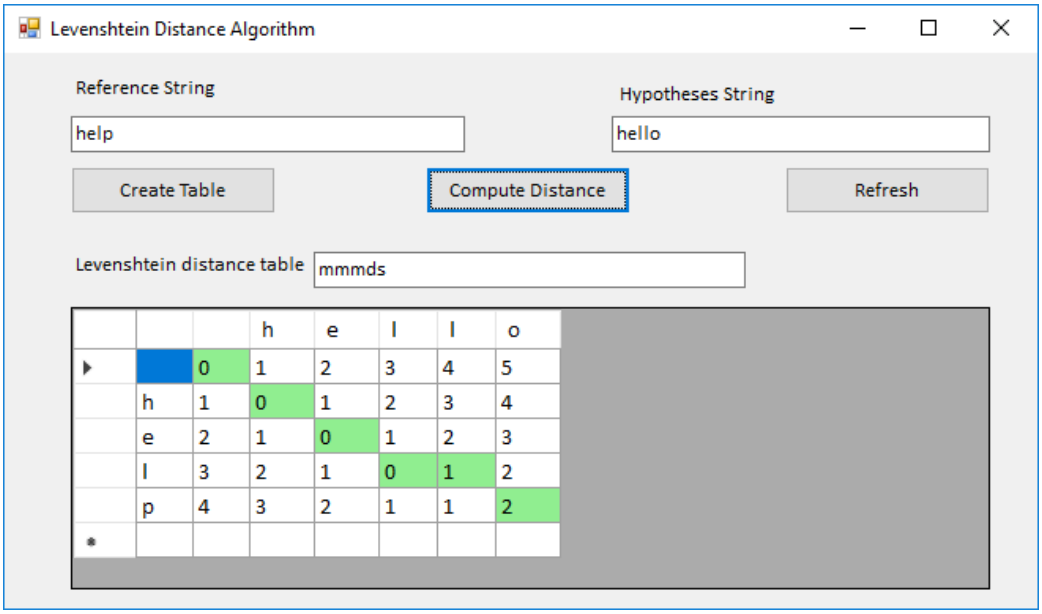
Rule 1 If the 2 letters (one of each word) are the same, then $d(i,j) = d(i-1, j-1)$ that is the score assigned to the cell is the same than the one included in the cell in diagonal.

Rule 2 We compute the cost of the different transformations in the following way :

- For a substitution : $c_1 = d(i-1, j-1) + w_1$
- For an insertion : $c_2 = d(i-1, j) + w_2$
- For a deletion : $c_3 = d(i, j-1) + w_3$

$d(i, j) = \min(c_1, c_2, c_3)$

3.1.2 Interpretation



			h	e	l	l	o
▶		0	1	2	3	4	5
	h	1	0	1	2	3	4
	e	2	1	0	1	2	3
	l	3	2	1	0	1	2
	p	4	3	2	1	1	2
*							

Figure 3.1: Exemple for "help" as reference and "hello" as hypothese, $w_1 = w_2 = w_3 = 1$. See [2] to find the program used.

For now, the reading of the Levenshtein Algorithm for the Calculation of a Distance that I can give is the following :

We fill the matrix linking the hypothesis word to the reference word with the scores corresponding to the different transformations (substitution, insertion and deletion) that we have to carry out in order to pass from the hypothesis word to the reference word.

Substitute If we have to carry out a **substitution** of a letter of the reference word with a letter of the hypothesis word we proceed as follows :
We consider the score that we led to obtain both the previous letter in the reference word (previous row i-1) and the previous letter in the hypothesis word (previous column j-1), and we add to this score the cost corresponding to the transformation which consists in the substitution of the respective following letters in each word (we increase both the index of the rows and the index of the columns).

Insert If we have to perform an **insertion** of a letter of the hypothesis word in the reference word we have to proceed as follows :
We consider the score that we led to obtain the previous letter in the reference word (previous row in the reference word i-1 and same column j, cf. Figure 5.1) and we add to this score the cost corresponding to the transformation which consists in the insertion of the letter located in the column j of the hypothesis word, just at the following of the letter located in the i-1 row of the reference word.

Delete If we have to lead a **deletion** of a letter in the hypothesis word to obtain the reference word, we have to proceed as follows :
We consider the score that we have led to obtain the previous letter in the hypothesis word (previous column in the hypothesis word j-1 and same row i, cf.Figure 5.1) and we add to this score the cost corresponding to the transformation which consists in the deletion of the letter located in the column j of the hypothesis word.

3.2 Opening

Weights Fitting We have seen that for each transformation we assign some weight that is own to the type of the transformation. We can wonder if there exist some optimization methods allowing to fit these weights so that they might be as representative as possible of the nature of the transformation (for instance high if the transformation is difficult to perform and low if the transformation is easy to perform).

Improvement of the Levenshtein Algorithm We have realized that the Levenshtein Distance was equal to the minimal number of transformations required to pass from an hypothesis string to a reference string. However, the calculation of the Levenshtein Distance is neither linked to the closeness between the letters in the alphabet nor to the similarities between the letter from a graphic point of view. Do some methods of supervised or unsupervised learning exist to take into account these parameters so as to improve the results of the Levenshtein algorithm only based on the minimal number of transformations carried out to pass from a string to another.

Chapter 4

Burkhard and Keller Tree or BK Tree

4.1 What is a BK Tree ?

[19] "A BK Tree is a data structure created by Burkhard and Keller in 1973. It is used for spell checking based on the Levenshtein Distance between two words, which is basically the number of changes you need to make a word to turn it into another word".

4.2 Building of a BK Tree

First, we have to build the BK Tree that we will then use to compare a given word to the ones included in the tree and find the words that are the closest to the given word in relation to the Levenshtein Distance.
How to build this BK Tree ?
We have to consider a given set of words. Among these words, we choose one as the root of the tree. Then, we compute the Levenshtein Distance between this root and each word of the set. If all the computed Levenshtein Distances are different from one word to another then we only build bisections from the root to each word of the set and on each branch of the tree we indicate the Levenshtein Distance pulling each word apart the root. If on the contrary, some computed Levenshtein Distances are identical, then we cannot build several bisections from the root to each word with the same Levenshtein Distance that is why to solve this problem, we compute the Levenshtein Distance between the words whose the Levenshtein Distance to the root is the same and we build bisections between them always based on the Levenshtein Distances. For instance, let's consider the following set of words :
$\{book \quad , books \quad , boo \quad , boon \quad , cook \quad , cake \quad , cart \quad , cape\}$

1. We choose as root : Root = "book"
2. We compute the Levenshtein Distances between the root and each of the other words :
   - d(book, books) = 1 : insertion of s
   - d(book, boo) = 1 : deletion of k
   - d(book, boon) = 1 : substitution of k with n
   - d(book, cook) = 1 : substitution of b with c
   - d(book, cake) = 4 : 4 substitutions
   - d(book, cart) = 4 : 4 substitutions
   - d(book, cape) = 4 : 4 substitutions

We can easily realize that several words of the set have the same Levenshtein Distances to the root : we only have two different Levenshtein Distances (1 and 4). In this case, we build 2 bisections from the root "book" : one going to a word whose the Levenshtein Distance to the root is equal to 1 ("books") and another going to a word whose the Levenshtein Distance to the root is equal to 4 ("cake"). We obtain two subsets of words : one for which the new root is "books" and in which the Levenshtein Distance pulling each word apart the old root "book" is equal to 1 and another for which the new root is "cake" and in which the Levenshtein Distance pulling each word apart the old root "book" is equal to 4. For each of these two subsets of words, we compute the Levenshtein Distance between each word of the subset and the new root.
For the first subset :

- d(books, boo) = 2 : 2 insertions
- d(books, boon) = 2 : 1 substitution of k with n and one deletion of s
- d(books, cook) = 2 : 1 substitution of b with c and one deletion of s

For the second subset :

- d(cake, cart) = 2 : 2 substitutions
- d(cake, cape) = 1 : 1 substitution of k with p

For the first subset of words, we can easily see that the three words "boo", "boon" and "cook" have the same Levenshtein Distance to their root "books" equal to 2. Therefore, we build a bisection from the root "books" to one of them ("boo") with a Levenshtein Distance equal to 2 and we compute again the Levenshtein Distances between each of the other words of the subset and their new root "boo".

- d(boo, boon) = 1 : 1 insertion of n at the end of the word
- d(boo, cook) = 2 : 1 substitution of b with c and 1 insertion of k at the end of the word

At this stage, the Levenshtein Distances between each word of the subset and their root are different so we can build one bisection from the root to each word of the subset, in other words : one bisection from the root "boo" to the word "boon" with a Levenshtein Distance equal to 1 and one bisection from the root "boo" to the word "cook" with a Levnshtein Distance equal to 2.

For the second subset, each word "cart" and "cape" has its own Levenshtein Distance to the root "cake". Therefore, we can build a bisection from the root "cake" to the word "cart" with a Levenshtein Distance equal to 2 and a bisection from the root "cake" to the word "cape" with a Levenshtein Distance equal to 1.

4.3 Search of matches

Once we have built the BK Tree thanks to a given set of words basing on the Levenshtein Distance computation, we can choose any word and crawl the BK Tree step by step in order to find the words the closest to the given word regarding the Levenshtein Distance.
How to perform the search of the words that are the closest to the given word regarding the Levenshtein Distance ?

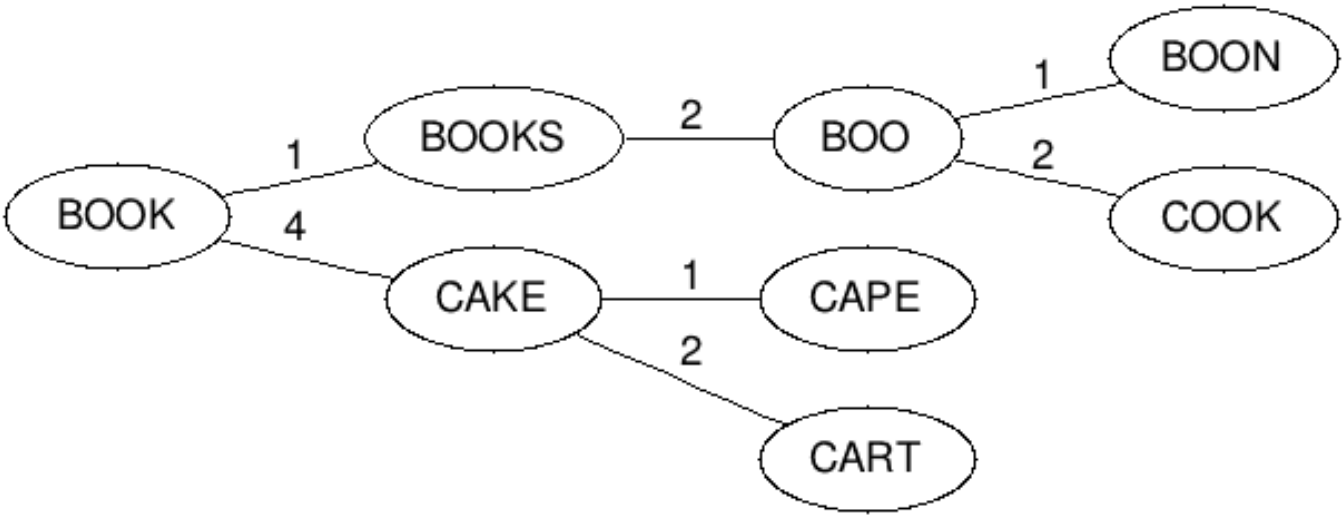


Figure 4.1: BK Tree Building.
See [19] to find the program used.

1. We start to compute the Levenshtein Distance between the given word w and the root of the BK Tree.

2. We compare the obtained Levenshtein Distance to the Levenshtein Distances assigned to the branches derived from the root with a certain tolerance d.

3. We choose to crawl the bisection for which the interval $[LevenshteinDistance-d, LevenshteinDistan$ $d]$ is the closest to the Levenshtein Distance between the given word w and the root.

4. Than, we compute once again the Levenshtein Distance between the root of the chosen bisection and the given word.

5. For each bisection that derives from this root we compute the following interval $[LevenshteinDistance$ $d, LevenshteinDistance + d]$ and we choose to crawl the bisection for which the interval of distance is the closest to the Levenshtein Distance between the root of the bisection and the given word.

6. In such a way we continue to iterate until we reach a leaf of the tree. In this case we have reached the end of the process and we can pick up the words of the path whose the Levenshtein Distance to the given word is lower than the given tolerance. These words are the closest regarding the Levenshtein Distance to the given word.

Complexity of the Algorithm :
Let's consider a set of ordered data. There are two main methods to cross the data :

1. **The Linear Search** : It consists to cross the ordered data until reaching the data that we are looking for.

2. **The Dichotomous Search** : We split the ordered data set into two equal parts and we compare the given data to the middle data, if the given data is lower than the middle data then we equate the upper bound of the interval to the middle, otherwise if the given data is upper than the middle data then we equate the lower bound of the interval to the middle data and we iterate a number n of times until the error between the given data and the middle of the interval might be lower than the fixed tolerance. With this

method, we iterate n times and at each time we divide by a factor 2 the previous interval. Therefore, between two successive iterations we reduce by half the number of data. At the end of the algorithm the final interval has a size equal to the size of the initial interval divided by 2^n.

For each of these methods it exists a better case and worse case.

For the Linear Search, the better case is the one for which the first data founded in our path is the searched data, in this case the searched data is immediately discovered. The worst case is the one for which the searched data corresponds to the last data introduced in our path, the given data is then discovered after having crawled all the data. The complexity of the algorithm corresponds to the worst case, for a dataset including n data, the complexity is in $\mathbf{O(n)}$ what means that in the worst case, the cost of calculation has the same order of magnitude than n : we have to cross all the data at least one time.

For the Dichotomous Search, the better case is the one for which the middle of the initial interval corresponds to the searched data, in this case the searched data is immediately discovered. The worst case is the one for which the searched data corresponds to the middle of the last interval, the searched data is then discovered after having divided by 2^n the initial interval.

Let's remind how the Dichotomous Search is working. The Dichotomous Search involves at each iteration of the algorithm to compute the middle of the current interval and to update one of the two bounds of the interval by equating it to the middle data : if the searched data is located in the first half of the interval that is the given data is lower than the middle then we update the upper bound by equating it to the middle, otherwise if the searched data is located in the second half of the interval that is the searched data is upper than the middle then we update the lower bound by equating it to the middle. Therefore as at each iteration we divide by 2 the current dataset (interval), this implies that between two successive iterations we reduce our dataset by half and we then only cross a dataset with a size divided by two in relation to the previous dataset. And, between the first iteration and the last iteration we have divided by 2^n the initial dataset (interval) and then we only cross a dataset whose the size is equal to the size of the initial dataset divided by 2^n instead of the whole dataset.

The complexity of the algorithm corresponds to the number of operations that we have to perform to reach the searched data in the worst case that is the case for which the searched data corresponds to the middle data of the n^{th} interval. In order to reach this stage, we have to divide by 2, n successive datasets (intervals), or in other words to divide by 2^n the initial dataset. Yet, so as to be able to divide by 2 the current dataset, the size of the current dataset has to be a multiple of 2 that is the number of data included in the current dataset has to be formed by an exactly number integer of 2 data, in other words the size of the current dataset has to be even. We are able to divide by 2 the inital dataset and then the successive datasets until that the size of the current dataset is not even anymore (odd). If we consider, that the size of n successive datasets is even, the size of the $(n+1)^{th}$ dataset is odd and that the error between the searched data and the middle of the n^{th} dataset is under the tolerance then, we are able to exactly divide by 2 the n times the initial dataset and the following subdatasets obtained at each iteration. Therefore, the number of operations that we have to perform is equal to the number of divisions by 2 of the successive datasets. The number of divisions by 2 that we are able to perform to reach the searched data in the worst case corresponds to the number of power of 2 that exists in the number of data included in the initial dataset. To define the maximal number of power of 2 that exists in the size of the initial dataset, we have to divide by 2 the number of data included in the initial dataset and in the successive subdatasets until that the obtained number of data included in the current dataset after a number integer n of divisions by 2 does not any more divisible by 2. At this stage we have reached the maximal

power of 2 that forms the number of data :

$$\begin{cases} N = 2^n + 1 \quad if \quad the \quad number \quad of \quad data \quad included \quad in \quad the \quad initial \quad dataset \quad is \quad odd \\ N = 2^n + 0 \quad if \quad the \quad number \quad of \quad data \quad included \quad in \quad the \quad initial \quad dataset \quad is \quad even \end{cases}$$

The reciprocal function of the function power of 2 is the function $\log_2$, in other words the function log_2 gives the number of power of 2 that exists in a given number. Therefore, the complexity of the algorithm of Dichotomous Search corresponds to the number of operations carried out to reach the searched data in the worst case. In other words, it corresponds to the number of divisions by 2 performed to find the searched data in the worst case that is the maximal number of power of 2 that exists in the number of data included in the initial dataset. This maximal number of power of 2 that exists in the number of data included in the initial dataset corresponds to $log_2(N)$.

In our case, we have a given word and we want to pick the words the closest to this given word up regarding the Levenshtein Distance. In this aim, at each floor of the BK Tree, we compute the Levenshtein Distance between the given word and the word corresponding to the root of the chosen bisection regarding the Levenshtein Distance. Then we compute the corresponding interval of distance taking into account the fixed tolerance d. We look at the Levenshtein Distances assigned to each child bisection and we check if these distances belong or not to the interval of distance. Then, we choose to crawl the bisection for which the Levenshtein Distance belongs to the interval of distance and is as close as possible to the Levenshtein Distance computed between the given word and the root of the bisections. We cross the tree in such a way until reaching the leaves what means that we have reached the end of the process and we just have to pick the word whose the Levenshtein Distance to the given word is the lowest up. These words form a subset corresponding to the matches. As for each node or root it generally derives two bisections, and as for each chosen root we choose to crawl the bisection for which the Levenshtein Distance is the closest to the Levenshtein Distance computed between the root and the given word, then at each iteration we approximately divide by 2 the number of data that compose our current dataset, in other words, we reduce by half the size of our current dataset. Thus, once we have built the whole path from the root of the BK Tree to one of the leaves by choosing at each iteration the bisection of the tree for which the Levenshtein Distance is the closest to the Levenshtein Distance computed between the root of the bisections and the given word, then we have divide the number of data included in the initial dataset by 2^n if we consider that the BK Tree is formed by n floors. Therefore, building a path from the root of the BK Tree to one of the leaves, by selecting at each iteration the bisection for which the Levenshtein Distance is the closest to the Levenshtein Distance computed between the current root and the given word, implies that at each iteration we reduce by half the amount of data so it is equivalent to carry out a Dichotomous Search. As the complexity of this algorithm corresponds to the maximal number of choices of bisections that we have to perform to go from the root of the BK Tree to one of the leaves and as this number corresponds to the number divisions by 2 that we are able to perform in the number of data included in the initial dataset that corresponds to the maximal number of power of 2 that exists in the number of data included in the intial dataset then the complexity is in $\mathbf{O(\log_2(N))}$.

Example of Search : Let $w =$ "*caqe*" a given word and $d = 1$ a fixed tolerance. We want to define the set of words that are the closest to this word regarding the Levenshtein Distance.

1. We compute the Levenshtein Distance between the root "book" and the given word $w =$ "*caqe*" : d(book, caqe) = 4 (4 substitutions).

2. We compute the interval of Levenshtein Distance taking into account the tolerance :

[4-1,4+1] = [3,5].

3. d(book, cake) = 4 ∈ [3,5] whereas d(book, books) = 1 ∉ [3,5] therefore we choose to crawl the bisection linking "book" to "cake".

4. We compute the Levenshtein Distance between the new root "cake" and the given word $w =$ "*caqe*" : d(cake, caqe) = 1 (1 substitution of k with q).

5. We compute the interval of Levenshtein Distance taking into account the tolerance : [1-1,1+1] = [0,2].

6. d(cake, cart) = 2 ∈ [0,2] and d(cake, cape) = 1 ∈ [0,2] but d(cake, cape) = d(cake, caqe) therefore we choose to crawl the bisection linking "cake" to "cape".

7. We have reached a leaf of the BK Tree "cape" so it is the end of the algorithm.

8. The path that we have built by crawling the tree from the root "book" to the leaf "cape" is formed by the following words {*book*, *cake*, *cape*} where d(caqe, book) = 4, d(caqe, cake) = 1, d(caqe, cape) = 1.

9. **Conclusion** : The matches are given by the following set of word : {*cake*, *cape*}.

4.4 Implementation of the BK Tree

What is the architecture of the code ?
As we have previously seen, the implementation of the BK Tree is composed of two main steps: the building of the BK Tree and the process of research of the matches for a given word.

4.4.1 Code Structure of the BK Tree Building

We create 2 classes :

- The Node Class: In this class, we define all the objects that we need to build the BK Tree especially the nodes containing a word or a key, but also the properties such as the getter and the setter which enable to access in reading and writing to the nodes and even the methods required to build the tree like the methods allowing to create a child to a given node.
- The BK Tree Class : In this class we define 4 methods that are intended to build the tree thanks basing on the Node Class and especially to perform researches of matching words in relation to a given word. This Class will be detail in the following subsection.

This method returns the Levenshtein Distance which corresponds to the minimal number of transformation that we have carried out on the hypothesis word or on the reference word so that the hypothesis word might be able to turn it into the reference word.

4.4.2 Code Structure of the Research Process

How the BK Tree Class is it organized ? The BK Tree Class is composed of 4 methods that are the following:

- Add : This method takes as parameter a word of type string. It allows to:

  1. Create a new node containing the word given in parameter.
  2. Compute the Levenshtein Distance between the root of the tree and the word included in the created node.
  3. Add the new child at the fair location in the BK Tree regarding the Levenshtein Distance thanks to the method allowing to add a child in the BK Tree belonging to the Node Class which takes as parameters the Levenshtein Distance between the root and the new node that we have to add and the word as a string.

- Search : This method takes as parameters a given word, a word for which we want to find matches, and a distance tolerance d. It gives back a list composed of the matching words regarding the Levenshtein Distance. So as to build the path from the root of the BK Tree to the leaves composed of the words that are the closest to the given word regarding the Levenshtein Distance this method calls the Recursive Search method.

- Recursive Search : This method a given word, a distance tolerance, a node and a list of string. It allows to build the path from the root of the BK Tree to the leaves of the BK Tree composed of the words that are the closest to the given word regarding the Levenshtein Distance in a recursive way. This method is called in the Search method which gives back the list of matching words built thanks to the Recursive Search method.

- Levenshtein Distance : This method takes as parameters two strings that correspond to the two words between which we want to compute the Levenshtein Distance. It gives back an integer corresponding to the Levenshtein Distance computed between the two words given as parameters. In this method we build a matrix of distances between the letters of the two words. The matrix has a number of rows equal to the number of letters of the first word plus 1 and a number of columns equal to the number of letters of the second word plus 1. We fill the first row (row 0) of the matrix with the index equal to the position of the given letter in the first word and the first column (column 0) of the matrix with the index equal to the position of the given letter in the second word. Concerning the center of the matrix, each cell corresponds to the transformation that we have to carry out so that the hypothesis word might be able to turn it into the reference word. Each transformation is represented by a number corresponding to the Levenshtein Distance specific to the transformation.

  - Let w_1 be the weighting specific to the substitution transformation.
  - Let w_2 be the weighting specific to the insertion transformation.
  - Let w_3 be the weighting specific to the deletion transformation.

  We can for example equate to 1 the weights respectively associated to the insertion and the deletion transformations and equate the weight associated to the substitution transformation to a boolean variable which is equal to 1 if the two previous letters both in the hypothesis word and in the reference word are the same what means that no transformation is required to turn the hypothesis word into the reference word thus we just have to report the same score in the following cell, and 0 if the two previous letters both in the hypothesis word and in the reference word are different what means that we have to

perform some transformations either in the hypothesis word or in the reference word so that the hypothesis word might become the reference word.

- Let i be the index which crosses the rows Levenshtein matrix.
- Let j be the index which crosses the columns Levenshtein matrix.

Once we have filled the first row and the first column, we have to fill the center of the grid by following some rules. A different weighting is assigned to each transformation :

Rule 0 We cross the grid from the top to the bottom and from the left to the right.

Rule 1 If the 2 letters (one of each word) are the same, then $d(i,j) = d(i-1,j-1)$ because the score assigned to the cell (i,j) is the same than the one included in the cell in the high left diagonal.

Rule 2 We compute the cost of the different transformations in the following way :

- For a substitution : $c_1 = d(i-1,j-1) + w_1$
- For an insertion : $c_2 = d(i-1,j) + w_2$
- For a deletion : $c_3 = d(i,j-1) + w_3$

$d(i,j) = \min(c_1, c_2, c_3)$

The Levenshtein Distance method returns the last cell of the matrix of distances (the cell located at the bottom right of the grid) which corresponds to the Levenshtein Distance between the two words given as parameters (hypothesis word and reference word). Indeed, the number included in this cell corresponds to the minimal cost of all the transformations carried out on the hypothesis word and on the reference word so that the hypothesis word might be able to turn into the reference word. The Levenshtein Distance method is called in the Add method which is intended to add a new node containing a given word to the BK Tree at the fair location regarding to the Levenshtein Distance.

You can find the entire code at the following source [19].

Chapter 5

DFA : Deterministic Finite Automaton

5.1 Definition of the Main Notions

5.1.1 Symbol

"A symbol, in computer programming is a primitive data type whose instances have a unique human readable form. Symbols can be used as identifiers. In some programming languages they are called atoms." [17]

5.1.2 Alphabet

"In formal language theory, a string is defined as a finite sequence of members of an underlying base set : this set is called the Alphabet of the string or collection of strings. The members of the set are called symbols and are typically thought as representing letters, characters or digits."[5]
Example :
A common Alphabet is the Binary Alphabet $\{0, 1\}$.
A Binary String is a string drawn from the Alphabet $\{0, 1\}$ such as "0101".
An infinite sequence of letters may be constructed from an alphabet as well.

5.1.3 Formal Language

"In Mathematics, Computer Sciences and Linguistics, a Formal Language consists of words whose letters are taken from an Alphabet and are well-formed according to a specific set of rules. The Alphabet of a Formal Language consists of symbols, letters or token that concatenate into strings of the language. Each string concatenated from the symbols of this Alphabet is called a Word, and the words that belong to a particular formal language are sometimes called Well-Formed Words or Well-Formed Formulas. A Formal Language is often defined by means of a formal grammar such as a regular grammar or context-free grammar, which consists of its formation rules."[9]

5.1.4 Grammar

"In Formal Language theory, a Grammar (when the context is not given, often called a Formal Grammar for clarity) is a set of production rules for strings in a Formal Language. The rules describe how to form strings from the language's Alphabet that are valid according to the language's syntax. A Grammar does not describe the meaning of the strings or what can be done with them in whatever context, only their form."[8]

5.1.5 Well-Formedness

"Well-Formedness is the quality of a clause, word or other linguistic element that conforms to the grammar of the language of which it is a part. Well-Formed words or phrases are grammatical, meaning they obey all relevant rules of grammar. In contrast, a form that violates some grammar rule is ill-formed and does not constitute part of the language."[18]

5.1.6 Regular Expression

"A Regular Expression, regex or regexp (sometimes called a Rational Expression) is a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithm for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in Theoretical Computer Science and Formal Language Theory."[15]

5.2 What is a DFA ?

"In the Theory of Computation, a branch of Theoretical Computer Science, a **D**eterministic **F**inite **A**utomaton (**DFA**), also knows as (aka) **D**eterministic **F**inite **A**cceptor (**DFA**), **D**eterministic **F**inite **S**tate **M**achine (**DFSM**), or **D**eterministic **F**inite **S**tate **A**utomaton (**DFSA**), is a Finite State Machine that accepts or rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. Deterministic refers to the uniqueness of the computation."[6]

Formal Definition of a DFA[6]
A Deterministic Finite Automaton is M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, consisting of :

- A finite set of states Q.
- A finite set of input symbols called the Alphabet Σ.
- A transition function: $\delta : Q \times \Sigma \longrightarrow Q$.
- An initial or start state $q_0 \in Q$.
- A set of accept states $F \subseteq Q$

Let $\omega = a_0 a_2 \dots a_n$ be a string over the alphabet Σ. The automaton M accepts the string if a sequence of states $r_0, r_1, \dots, r_n$ exists in Q with the following conditions :

- $r_0 = q_0$: The machine starts in the inital state w_0.
- $r_{i+1} = \delta(r_i, a_{i+1}) \quad \forall i = 0, \dots, n-1$: Given each character of the string ω, the machine will transition from state to state according to the transition function δ.
- $r_n \in F$: The machine accepts ω if the last input of ω causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton rejects the string. The set of strings accepted by M is the language recognized by M and this language is denoted by **L(M)**.

Reading:

$M = (Q, \Sigma, \delta, q_0, F)$ where :

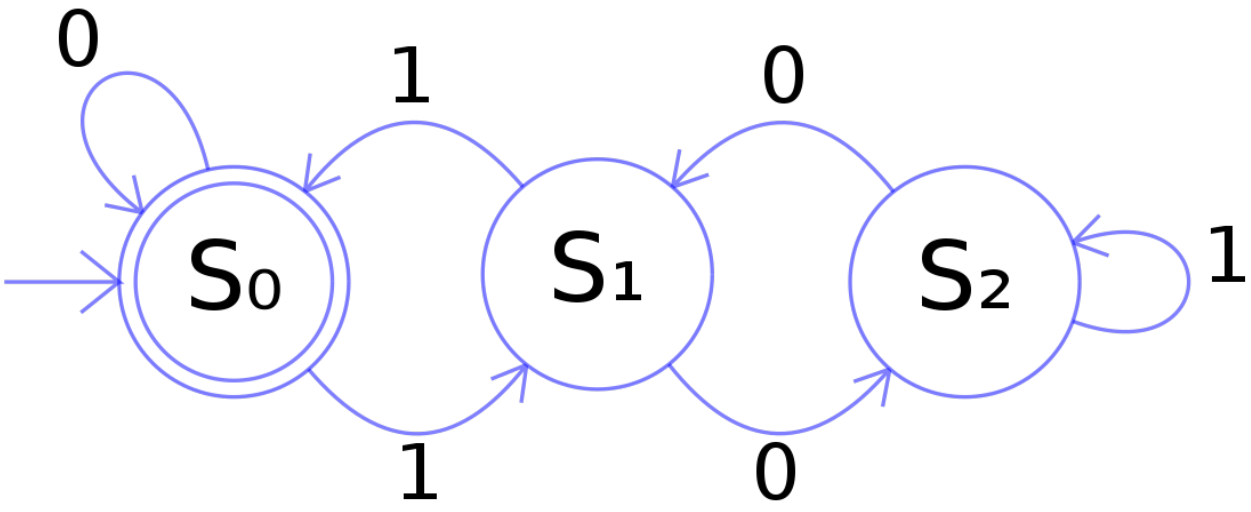


Figure 5.1: Exemple of DFA.
See [6]

- $Q = \{S_0, S_1, S_2\}$
- $\Sigma = \{0, 1\}$
- δ is defined by the following transition table :

	0	1
S_0	S_0	S_1
S_1	S_2	S_1
S_2	S_1	S_2

- $q_0 = S_0$
- $F = \{S_0\}$
- S_0 is both the initial state and the final state of the automaton.
- When we are in the state S_0 we can either remaining in S_0 through the transition "0" or going in S_1 with the transition "1".
- When we are in the stazte S_1 we can either going in S_2 through the transition "0" or coming back in s_0 through the transition "1".
- When we are in the state S_2 we can either remaining in S_2 through the transition "1" or coming back in S_1 through the transition "0".
- The sum of the gains of the arrows arriving on each transition state is equal to 1.
- A state from which we cannot go anymore is called an absorbent state or a well.

5.3 What is the DFA Minimization ?

"In Automata Theory (A Branch of Theoretical Computer Science), DFA Minimization is the task of transforming a given Deterministic Finite Automaton (DFA) into an equivalent DFA

that has a minimum number of states. Here, two DFAs are called equivalent if they recognize the same regular language. There exist several algorithms accomplishing this task."[7]

Minimum DFA

For each regular language, there also exists a Minimal Automaton that accepts it, that is, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching. There are two classes of states that can be removed or merged from the original DFA without affecting the language it accepts to minimize it.

- **UnreachableUnreachable States** are the states that are not reachable from the initial state of the DFA, for any input string.
- **Nondistinguishable States** are those that cannot be distinguished from one another for any input string.

DFA Minimization is often done in 3 steps, corresponding to the removal or the merger of the relevant states. Since the elimination of Nondistinguishable states is computationally the most expensive one, it is usually done as the last step.

5.4 How to proceed in order to minimize a DFA ?

In order to minimize a DFA, we proceed as follows :

1. We remove the Unreachable states from the initial state, for any input string.
2. We identify the remaining states which plays an identical role from the point of view of the recognition. These states are called the Nondistinguishable states.

Myhill-Nerode Theorem

Let L be a rational language.
Among all the DFA that recognize L, it exists a unique one which has a minimal number of states.

Examples of Minimization Process of DFAs
Example n°1

Definition of the Automaton :
$M = \{Q, \Sigma, \delta, q_0, F\}$ where :

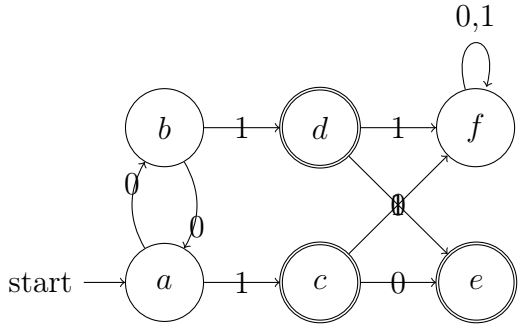
- $Q = \{a, b, c, d, e, f\}$
- $\Sigma = \{0, 1\}$
- δ is defined by the following transition table :

	0	1
a	b	c
b	a	d
c	e	f
d	e	f
e	e	f
f	f	f

- $q_0 = a$

- $F = \{c, d, e\}$

DFA



Minimization

	a	b	c	d	e	f
0	b	a	e	e	e	f
1	c	d	f	f	f	f

Equivalence $\sim_0$:
Initially we can distinguish 3 classes as follows :

1. The absorbent States Class : $[f]_0$
2. The Final States Class : $[c]_0$
3. The Other States Class : $[a]_0$

	a	b	c	d	e	f
$\sim_0$	$[a]_0$	$[a]_0$	$[c]_0$	$[c]_0$	$[c]_0$	$[f]_0$

Equivalence $\sim_1$:

	a	b	c	d	e	f
$\sim_0$	$[a]_0$	$[a]_0$	$[c]_0$	$[c]_0$	$[c]_0$	$[f]_0$
0	$[a]_0$	$[a]_0$	$[c]_0$	$[c]_0$	$[c]_0$	$[f]_0$
1	$[c]_0$	$[c]_0$	$[f]_0$	$[f]_0$	$[f]_0$	$[f]_0$
$\sim_1$	$[a]_1$	$[a]_1$	$[c]_1$	$[c]_1$	$[c]_1$	$[f]_1$

Equivalence $\sim_2$:

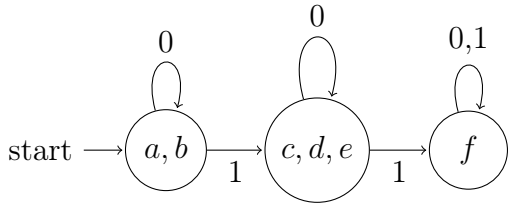
	a	b	c	d	e	f
$\sim_1$	$[a]_1$	$[a]_1$	$[c]_1$	$[c]_1$	$[c]_1$	$[f]_1$
0	$[a]_1$	$[a]_1$	$[c]_1$	$[c]_1$	$[c]_1$	$[f]_1$
1	$[c]_1$	$[c]_1$	$[f]_1$	$[f]_1$	$[f]_1$	$[f]_1$
$\sim_2$	$[a]_2$	$[a]_2$	$[c]_2$	$[c]_2$	$[c]_2$	$[f]_2$

We can easily realize that the Equivalence $\sim_1$ and the Equivalence $\sim_2$ are the same what means that we have reached a stationary state. Therefore, the Undistinguishable States are the following :

- $\{a, b\}$
- $\{c, d, e\}$

- $\{f\}$

The Minimum DFA is thus given as follows :



Example n°2

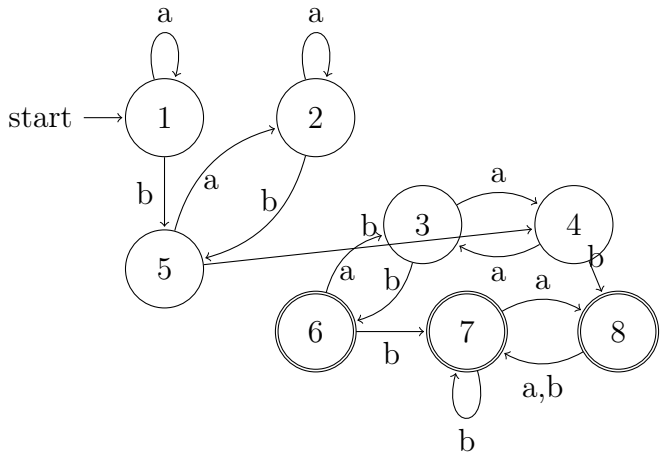
Definition of the Automaton :
$M = \{Q, \Sigma, \delta, q_0, F\}$ where :

- $Q = \{1, 2, 3, 4, 5, 6\}$
- $\Sigma = \{a, b\}$
- δ is defined by the following transition table :

	a	b
1	1	5
2	2	5
3	4	6
4	3	8
5	2	4
6	3	7
7	8	7
8	7	7

- $q_0 = 1$
- $F = \{6, 7, 8\}$

DFA



Minimization

	1	2	3	4	5	6	7	8
a	1	2	4	3	2	3	8	7
b	5	5	6	8	4	7	7	7

Equivalence $\sim_0$:
Initially, there are only 2 classes :

1. The Final States Class : $[6]_0 = \{6, 7, 8\}$
2. The Other States Class : $[1]_0 = \{1, 2, 3, 4, 5\}$

	1	2	3	4	5	6	7	8
$\sim_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[6]_0$	$[6]_0$	$[6]_0$

Equivalence $\sim_1$:

	1	2	3	4	5	6	7	8
$\sim_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[6]_0$	$[6]_0$	$[6]_0$
a	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[1]_0$	$[6]_0$	$[6]_0$
b	$[1]_0$	$[1]_0$	$[6]_0$	$[6]_0$	$[1]_0$	$[6]_0$	$[6]_0$	$[6]_0$
$\sim_1$	$[1]_1$	$[1]_1$	$[3]_1$	$[3]_1$	$[1]_1$	$[6]_1$	$[7]_1$	$[7]_1$

Equivalence $\sim_2$:

	1	2	3	4	5	6	7	8
$\sim_1$	$[1]_1$	$[1]_1$	$[3]_1$	$[3]_1$	$[1]_1$	$[6]_1$	$[7]_1$	$[7]_1$
a	$[1]_1$	$[1]_1$	$[3]_1$	$[3]_1$	$[1]_1$	$[3]_1$	$[7]_1$	$[7]_1$
b	$[1]_1$	$[1]_1$	$[3]_1$	$[7]_1$	$[3]_1$	$[7]_1$	$[7]_1$	$[7]_1$
$\sim_2$	$[1]_2$	$[1]_2$	$[3]_2$	$[4]_2$	$[5]_2$	$[6]_2$	$[7]_2$	$[7]_2$

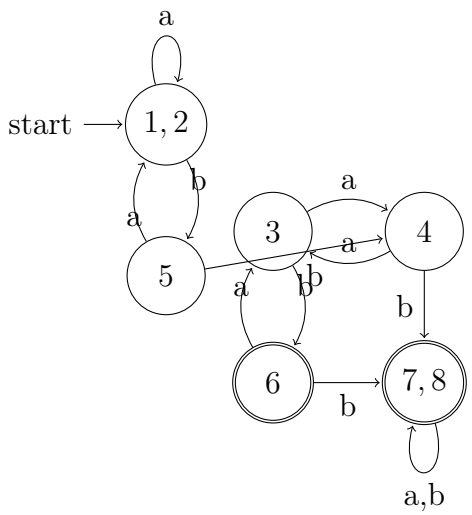
Equivalence $\sim_3$:

	1	2	3	4	5	6	7	8
$\sim_2$	$[1]_2$	$[1]_2$	$[3]_2$	$[4]_2$	$[5]_2$	$[6]_2$	$[7]_2$	$[7]_2$
a	$[1]_2$	$[1]_2$	$[4]_2$	$[3]_2$	$[1]_2$	$[3]_2$	$[7]_2$	$[7]_{2s}$
b	$[5]_2$	$[5]_2$	$[6]_2$	$[7]_2$	$[4]_2$	$[7]_2$	$[7]_2$	$[7]_2$
$\sim_3$	$[1]_3$	$[1]_3$	$[3]_3$	$[4]_3$	$[5]_3$	$[6]_3$	$[7]_3$	$[7]_3$

We can easily realize that the Equivalence $\sim_2$ and the Equivalence $\sim_3$ are the same what means that we have reached a stationary state. Therefore, the Undistinguishable States are the following :

- $\{1, 2\}$
- $\{3\}$
- $\{4\}$
- $\{5\}$
- $\{6\}$
- $\{7, 8\}$

The Minimum DFA is thus given as follows :



5.5 Test Automaton Plot

Reference String : var = b o o k
Levenshtein Distance : d

Chapter 6

Fuzzy Matching

6.1 What is Fuzzy Matching ?

"In Computer Science, Approximate String Matching (often called Fuzzy String Searching) is the technique of finding strings that match a pattern approximately (rather than exactly). The problem of approximate string matching is typically divided into two sub-problems: finding approximate sub-string matches inside a given string and finding dictionary strings that match the pattern approximately."[10]

6.2 How to determine the closeness of a match ?

[10] The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. This number is called the Edit Distance or Levenshtein Distance between the string and the pattern.
What are the primitive operations so that a given string might be able to exactly turn it into the pattern ?
The main primitive operations on the strings are given as follows:

- **Insertion:** cot $\longrightarrow$ co**a**t
- **Deletion:** co**a**t $\longrightarrow$ cot
- **Substitution:** co**a**t $\longrightarrow$ co**s**t

These 3 primitive operations may be generalized as forms of substitutions by adding a NULL character that we can symbolized by a star: $*$ wherever a character has been inserted or deleted. If we come back to the previous example we obtain:

- **Insertion:** co$*$t $\longrightarrow$ co**a**t
- **Deletion:** co**a**t $\longrightarrow$ co$*$t
- **Real Substitution:** co**a**t $\longrightarrow$ co**s**t

Some approximate matchers also treat transposition, in which the positions of 2 letters in the string are swapped, to be a primitive operations.

- **Transposition:** co**st** $\longrightarrow$ co**ts**

The transposition can also be seen as two consecutive substitutions:

1. **Initial String:** cost
2. **First Substitution:** cos**t** $\longrightarrow$ co**t**t
3. **Second Substitution:** co**t**t $\longrightarrow$ cot**s**
4. **Final String:** cots

Different constraints can be imposed on the matching process according the chosen matcher. What are these different constraints ?

- A single global unweighted cost: All operations count as a single unit of cost and the limit is set to one. In this case the Edit Distance exactly corresponds to the total number of primitive operations necessary to convert the match to the pattern.
  Example:
  - **c**oil $\longrightarrow$ **f**oil : 1 Substitution
  - coil$*$ $\longrightarrow$ coil**s** : 1 Insertion
  - **c**oil $\longrightarrow$ $*$oil : 1 Deletion
  - **c**o**i**l $\longrightarrow$ **f**o**a**l : 2 Substitutions

  If we consider that all primitive operation counts as a single unit and if the limit is set to one then, the strings "foil", "coils" and "oil" are counted as matches whereas the string "foal" does not count as match as its Edit Distance is equal to 2.

  According to this weighting method, we can consider the following thing :

  - **c**oil$*$ $\longrightarrow$ $*$oil**s** : 1 Deletion and 1 Insertion

  We does not distinguish the costs of each type of primitive operations so we combine all the costs to obtain a total cost which is given by : **Total Weight:** 2.

- A single global cost but with different weights assigned each type of primitive operations. According this weighting method, we fix a cost specific to each type of primitive operations.
  Example:
  - For each Substitution: $w_1 = 1$ (A Substitution requires to perform first a Deletion and then an Insertion that is why we fix a cost twice upper than the costs assigned to an Insertion or to a Deletion).
  - For each Insertion: $w_2 = 0.5$
  - For each Deletion: $w_3 = 0.5$

  Special Case:
  - **b**ook$*$ $\longrightarrow$ **l**ook**s** : 1 Substitution and 1 Insertion

  Total Weight: 1 + 0.5 = 1.5

- The number of operations of each type is specified separately.
  Example:

  – **c**oil$*$ $\longrightarrow$ $*$oil**s** : 1 Deletion and 1 Insertion

  According to this weighting method, we distinguish the costs of each type of primitive operations and we do not gather the costs together, we keep apart the costs of each type of primitive operations and we compare these costs with each given threshold (This method implies to fix a threshold for each type of primitive operation).

6.3 What are the main objectives of Fuzzy Matching ?

[10] Once we have fixed a Levenshtein distance considered as a tolerance of dissimilarity between the reference string and the hypothesis string, we can focus on the two classical algorithmic problems which are the following :

1. **Problem n°1:**
   Given a long text, a pattern and an integer k corresponding to the tolerance of Edit Distance, the objective is to find the positions in the text where the pattern is approximately present, that is, the text contains a string at a distance at most k from the searched pattern.

2. **Problem n°2:**
   Given a string and a dictionary, the objective is to find a string in the dictionary which minimizes the Edit Distance to the given word in entry.

6.4 What are the scope of the Fuzzy Matching ?

[10] The most common application of approximate matchers until recently has been spell-checking.
What is spell-checking ?
In software, a spell checker (or spell check) is a software feature that checks for mispellings in a text.
With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application.
Approximate matching is also used in spam filtering.
String matching cannot be used for most binary data, such as images and music. They require algorithms, such as acoustic fingerprinting.

6.5 Some Definitions

- **String**
  "In computer programming, a string is traditionally a sequence of characters, either as a literal constant or as some kind of variable. The latter may allow its elements to be mutated and the length changed, or it may be fixed (after creation). A string is generally considered as a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. String may also denote more general arrays or other sequence (or list) data types and structures."[16]

- **Pattern**
  "A pattern is a regularity in the World, man-made design, or abstract ideas. As such, the elements of a pattern repeat in a predictable manner. A geometric pattern is a kind of pattern formed of geometric shapes and typically repeated like a wallpaper design."[13]

Chapter 7

Record Linkage

7.1 What is Record Linkage ?

"Record Linkage (RL) is the task of finding records in a data set that refer to the same entity across different data sources (e.g. data files, books, websites, and databases). Record Linkage is necessary when joining data sets based on entities that may or may not share a common identifier (e.g. database key, URI, national identification number), which may be due to differences in record shape, storage location, or curator style or preference. A data set that has undergone Record Linkage oriented reconciliation may be referred to as being cross-linked. Record Linkage is called data linkage in many jurisdictions, but is the same process."[14]
It exists several types of Record Linkage, we are going to focus ourselves on one of them which is the Probabilistic Record Linkage.

7.2 What is the Probabilistic Record Linkage ?

Chapter 8

MatchID Project Presentation

8.1 MatchID Background

We have a database containing all the personal records of the French drivers and each personal record is defined by the following characteristics : last name, first name, date of birth and place of birth.
Nevertheless, a problem arises from this context : a withdrawal of driving licence points is performed on some driving licences whereas people are dead. In order to solve this problem, we have to withdraw the personal records of the dead people from our driving licences database.

Stake:
How to efficiently withdraw the personal records of the dead people from the French driving licences database ?

8.2 What are the methods used to reach this aim ?

We have to carry out joints between the database containing all the personal records of the French drivers and the database containing the drivers who are dead. There are several types of joints : the stringent joints, the fuzzy joints and the mixed joints (stringent and fuzzy). After having carried out joints between the two databases, we obtain a sub-database containing all the drivers shared by the two databases that is the French drivers who are dead also called the matches. If we only perform stringent joints between the 2 databases, we only obtain 33% of the matches that we would have to obtain. When in addition to the stringent joints, we carry out a preparation of the data which consists in the correction of the writing of the towns names (such as the correction of the dash presence or absence, the accented characters, the abbreviation ...), we obtain 70% of the matches that we would have to obtain. And if besides the stringent joints, we perform a preparation of the data as well as a fuzzy matching, then we obtain 95 to 98% of the matches that we would have to obtain. Thus, mixed joints both composed of stringent joints and fuzzy matching can be very interesting in order to improve the results of the matching.

8.3 How to proceed ?

First, we proceed to a data cleansing phase which consists in the preparation of the database in order to improve the results of the following treatments. This data preparation phase is composed of :

- A writing correction step: we check that all the birth towns names be consistent with their corresponding ID, when it is not the case, we rewrite the birth towns names so that the correction might be able to match with the corresponding ID (for instance, we remove the dash, we take away the abbreviation, we take down the accents ...)

- Once we have rewrite the birth towns names so that they might be able to be consistent with their ID, we add a field to the characteristics of the drivers which is the INSEE code associated to the birth town ID.

- A Fuzzy Matching between the birth towns ID of the database containing the personal records of all the French drivers and the birth towns ID of the database containing the dead drivers that we want to remove from the global database : This Fuzzy Matching allows to perform fuzzy joints (a fuzzy joint has a most important tolerance than a stringent joint: we can for instance permit a Levenshtein Distance equal to 2 between the matches) on the birth towns ID in order to obtain the real birth town ID of the driver.

Once we have carried out the data preparation phase, we have to perform the data treatment whose the aim is to execute the joints between the database containing the personal records of all the French drivers and the database containing the dead drivers that we want to remove from the global database. This process aims at withdrawing the obtained matches which correspond to the French dead drivers from the global database to avoid a withdrawal of license points to a dead driver. In order to accomplish this task we perform a Fuzzy Matching between the first names of French drivers from the global database and the first names of the dead French drivers. The Fuzzy Matching, as previously quoted, allows to execute joints between two databases, in relation to certain fields (the first names of the drivers in our case), while permitting some error (we fix a Levenshtein Distance which corresponds to the error term between the matches) so as to maximize the performance of the joints carried out. The sub-database obtained after having carried out fuzzy matching between our two databases is composed of the personal records of the French dead drivers whose the first names are matching to the nearest given Levenshtein Distance. This sub-databases includes about 95% to 98% of the matching that we would have to really obtain.

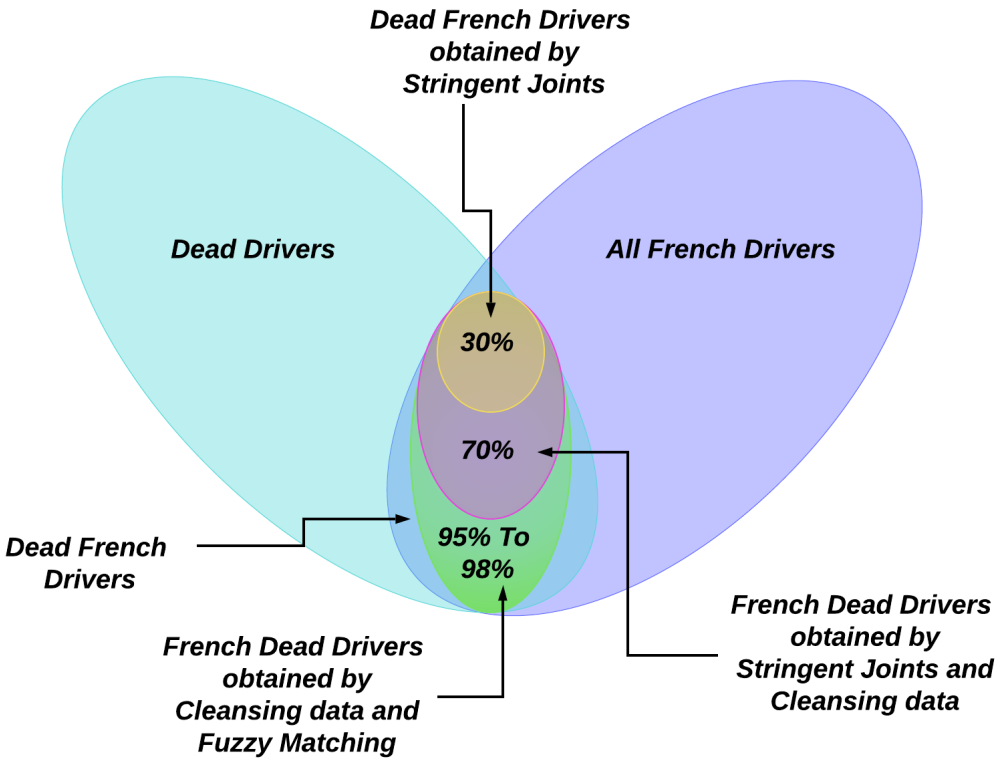


Figure 8.1: Set Diagram.

Chapter 9

Libraries used in the MatchID Project

9.1 Bisect Module in Python

cf.[3]

9.1.1 What does the use of the Bisect Module bring ?

To use the Bisect Algorithm, we have to have at our disposal a list of items in a sorted order (if the items are numbers, they have to be sorted either in an increasing order or in a decreasing order, otherwise, if the items are letters or strings, they have to be sorted either in alphabetical order or in the opposite of the alphabetical order). Once we have sorted the list of items that we have at our disposal, we are able to use the Bisect Algorithm implemented in Bisect Module in Python.
What is the aim of the Bisect Algorithm ?
The Bisect Algorithm aims at finding the position, in the list of items that we have at our disposal, of a given item so that this insertion might be able to keep the items of the list in a sorted order taking into account the item newly added.
This Bisect Algorithm is based on 3 main Bisection Functions given as follows :

1. **The Bisect Function : bisect(list, itemToInsert, begin, end)**
   The parameters of this function are the following :

   - **list** is the sorted list that we have at our disposal.
   - **itemToInsert** is the item that we want to insert in our list without disturbing the order of the item.
   - **begin** is the position of the first item of the sorted list.
   - **end** is the position of the last item of the sorted list.

   This function splits the sorted list into two parts containing or not the same number of items that is containing or not a number of items equal to the half of the total number of items in the sorted list, and returns the position between the two subsets obtained after the split which corresponds to the position of the items that we want to insert into the list so that the resultant list might remain sorted after the insertion of the new item.
   If the item that we want to insert into the list is already present in the list, then the function returns the right most position in the list allowing to keep the list sorted after the insertion of the new item.
   The two subsets obtained after the split contain the same number of items that is a number of items equal to the half of the total number of items in the list, if and only if the

position in the list where we have to insert the new item so that the list might remain sorted after the insertion, corresponds to the middle point of the list also called in this case the median of the list as the latter is sorted. In this case, each of the two subsets contains the same number of items which corresponds to the half of the total number of items in the whole list.
If the position of the item that we want to insert in the list so that the resultant list might remain sorted after the insertion, does not correspond to the middle point of the list, then the two subsets obtained after the split do not contain the same number of items, one of them contains a number of items lower than the half of the total number of items in the list and the other contains a number of items greater than the half of the total number of items in the list.
Example :
Let $l = 5, 2, 3, 1, 4$ a list.
We have to sort the list, in this aim we can use several function in Python as follows :
$sorted\_list = sorted(l) = 1, 2, 3, 4, 5$ or $sorted\_list = l.sort = 1, 2, 3, 4, 5$
Let $new\_item = 2.5$ the new item that we want to insert in the list l in such a way that the resultant list l might remain sorted even after the insertion of the new item. In order to find the position where we have to insert the new item so that the resultant list might remain sorted, we split the initial list into two subsets between which there is the position of the new item allowing to the resultant list to remain sorted after the insertion. The two subsets that we obtain after the split are the following :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Therefore, the position of the new item is located between 2 and 3 that is at the 3^{rd} position in the list.
The resultant list is then the following : $resultant\_list = 1, 2, 2.5, 3, 4, 5$ and this list is always sorted in ascending order.
If now the new item that we want to insert in the list l is the following $new\_item = 3$, this item is already present in the list so the position where we have to insert the new item in the list is the most right position allowing to keep a sorted list. In our case the most right position allowing to keep a sorted list is the 4^{th} position in the list.
The resultant list is then the following : $resultant\_list = 1, 2, 3, 3, 4, 5$
The initial list l contains an odd number of items, $n = 2k + 1 = 2 \times 2 + 1$ so the number of items is not a multiple of 2 that is the number of items is not divisible by 2. Then we can constitute 2 subsets containing each 2 items and it remains 1 item between the 2 subsets that is the middle point of the list and that corresponds in our case to 3. This is the median of the list as the list is sorted in ascending order. When we insert a new item to the list, we have to split the list into two subsets between which we can find the position in the list where inserting the new item so that the list might remain sorted even after the insertion. As the initial list contains an odd number of items, then when we split the list into two subsets we never can obtain two subsets containing exactly the same number of items, when we approach at most to the equilibrium we obtain one subset containing a number of items equal to the half of the total number of items plus one item $lower\_subset = 1, 2$ and the other containing a number of items equal to the half of the total number of items $upper\_subset = 3, 4, 5$, so the new item never can be the middle point of the resultant list.
If we consider now an initial list $l = 2, 3, 1, 4$ containing an even number of items then the number of items included in the initial list is exactly formed by an integer group of 2 items : $n = 2k = 2 \times 2$ so the number of items is a multiple of 2 that is the number of items is divisible by 2. Then we can constitute exactly 2 subsets containing each 2 items without it remains additional item. In this case, as after the split there is no additional

item, the middle point of the list does not belong to the list that is the median is not an item of the list, it corresponds to the average between the last item of the first half and the first item of the second half. Therefore, when we split the list into two subsets in order to find the position where we have to insert the new item so that the resultant list might remain sorted even after the insertion of the new item, in some particular case we can obtain two subsets exactly containing the same number of items equal to the half of the total number of items and in this particular case the new item is the middle point or the median of the resultant list.
For instance, if we have to insert the following item : $new\_item = 2.5$ in the sorted list $sorted\_list = 1, 2, 3, 4$ then we split the sorted list as follows :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4$
We obtain two subsets containing each two items and we have to insert the new item between the two halfs of the initial list that is at the 3^{rd} position. Therefore, the new item is the middle point or the median of the list.

2. **The Bisect Left Function : bisect_left(list, itemToInsert, begin, end)**
The parameters of this function are the following :

   - **list** is the sorted list that we have at our disposal.
   - **itemToInsert** is the item that we want to insert in our list without disturbing the order of the item.
   - **begin** is the position of the first item of the sorted list.
   - **end** is the position of the last item of the sorted list.

   This function splits the sorted list into two parts containing or not the same number of items that is containing or not a number of items equal to the half of the total number of items in the sorted list, and returns the position between the two subsets obtained after the split, which corresponds to the position of the items that we want to insert into the list so that the resultant list might remain sorted after the insertion of the new item.
   If the item that we want to insert into the list is already present in the list, then the function returns the left most position in the list allowing to keep the list sorted after the insertion of the new item.
   The two subsets obtained after the split contain the same number of items that is a number of items equal to the half of the total number of items in the list, if and only if the position of the item that we want to insert into the list so that the resultant list might remain sorted after the insertion of the new item, corresponds to the middle point of the sorted list also called the median of the list as the latter is sorted. In this case, the two subsets contain the same number of items which corresponds to the half of the total number of items in the list.
   In the case where the position of the item that we want to insert into the list so that the resultant list might remain sorted after the insertion, does not correspond to the middle point of the list, then the two subsets obtained after the split do not contain the same number of items, one of them contains a number of items lower than the half of the total number of items in the list and the other contains a number of items greater than the half of the total number of items in the list.
   Example :
   Let $l = 5, 2, 3, 1, 4$ a list.
   We have to sort the list, in this aim we can use several function in Python as follows :
   $sorted\_list = sorted(l) = 1, 2, 3, 4, 5$ or $sorted\_list = l.sort = 1, 2, 3, 4, 5$

Let $new\_item = 2.5$ the new item that we want to insert in the list l in such a way that the resultant list l might remain sorted even after the insertion of the new item. In order to find the position where we have to insert the new item so that the resultant list might remain sorted, we split the initial list into two subsets between which there is the position of the new item allowing to the resultant list to remain sorted after the insertion. The two subsets that we obtain after the split are the following :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Therefore, the position of the new item is located between 2 and 3 that is at the 3^{rd} place in the list. The resultant list is given by : $resultant\_list = 1, 2, 2.5, 3, 4, 5$ and remains sorted in ascending order.
If now the new item that we want to insert in the list l is the following $new\_item = 3$, this item is already present in the list so the position where we have to insert the new item in the list is the most left position allowing to keep a sorted list. When we split into two subsets our initial list according this rule we obtain the following subsets :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Therefore, the left most position where we can inserting the new item so that the resultant list might remain sorted is between 2 and 3 that is at the 3^{rd} position in the list. We obtain the following resultant list :
$resultant\_list = 1, 2, \mathbf{3}, 3, 4, 5$
The initial list l contains an odd number of items, $n = 2k+1 = 2 \times 2+1$ so the number of items is not a multiple of 2 that is the number of items is not divisible by 2. Then we can constitute 2 subsets containing each 2 items and it remains 1 item between the 2 subsets that is the middle point of the list and that corresponds in our case to 3. This is the median of the list as the list is sorted in ascending order. When we insert a new item to the list, we have to split the list into two subsets between which we can find the position in the list where inserting the new item so that the list might remain sorted even after the insertion. As the initial list contains an odd number of items, then when we split the list into two subsets we never can obtain two subsets containing exactly the same number of items, when we approach at most to the equilibrium we obtain one subset containing a number of items equal to the half of the total number of items and another containing a number of items equal to the half of the total number of items plus one item :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Thus the new item can never be the middle point of the resultant list.
If we consider now an initial list $l = 2, 3, 1, 4$ containing an even number of items, then the number of items included in the initial list is exactly formed by a number integer group of 2 items : $n = 2k = 2 \times 2$ so the number of items is a multiple of 2 that is the number of items is divisible by 2. Then we can constitute exactly 2 subsets containing each 2 items without that it remains additional item. In this case, as after the split there is no additional item, the middle point of the list does not belong to the list that is the median is not an item of the list, it corresponds to the average between the last item of the first half and the first item of the second half. Therefore, when we split the list into two subsets in order to find the position where we have to insert the new item so that the resultant list might remain sorted even after the insertion of the new item, in some particular case we can obtain two subsets exactly containing the same number of items equal to the half of the total number of items and in this particular case the new item is the middle point or the median of the resultant list.
For instance, if we have to insert the following item : $new\_item = 2.5$ in the sorted list $sorted\_list = 1, 2, 3, 4$ then we split the sorted list as follows :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4$
We obtain two subsets containing each two items and we have to insert the new item

between the two halfs of the initial list that is at the $3rd$ position. Therefore, the new item is the middle point or the median of the resultant list : $resultant\_list = 1, 2, \mathbf{2.5}, 3, 4$.

3. **The Bisect Right Function : bisect_right(list, itemToInsert, begin, end)**
The parameters of this function are the following :

   - **list** is the sorted list that we have at our disposal.
   - **itemToInsert** is the item that we want to insert in our list without disturbing the order of the item.
   - **begin** is the position of the first item of the sorted list.
   - **end** is the position of the last item of the sorted list.

   This function splits the sorted list into two parts containing or not the same number of items that is containing or not a number of items equal to the half of the total number of items in the sorted list, and returns the position between the two subsets obtained after the split, which corresponds to the position of the item that we want to insert into the list so that the resultant list might remain sorted after the insertion of the new item.
   If the item that we want to insert into the list is already present in the list, then the function returns the right most position in the list allowing to keep the list sorted after the insertion of the new item.
   the two subsets obtained after the split contain the same number of items that is a number equal to the half of the total number of items in the list, if and only if the position of the item that we want to insert into the list so that the resultant list might remain sorted even after the insertion of the new item, corresponds to the middle point of the list also called in this case the median of the list as the list is sorted. In this case, the position in the list of the new items shares the items of the sorted list into two subsets of equal number and we obtain two subsets containing a number of items equal to the half of the total number of items in the list.
   In the case where the position of the item that we want to insert into the list so that the resultant list might remain sorted after the insertion, does not correspond to the middle point of the list, then the two subsets obtained after the split do not contain the same number of items, one of them contains a number of items lower than the half of the total number of items in the list and the other contains a number of items greater than the half of the total number of items in the list.
   The Bisect Function performs by default the same task than the Bisect Right Function.
   Example :
   Let $l = 5, 2, 3, 1, 4$ a list.
   We have to sort the list, in this aim we can use several function in Python as follows :
   $sorted\_list = sorted(l) = 1, 2, 3, 4, 5$ or $sorted\_list = l.sort = 1, 2, 3, 4, 5$
   Let $new\_item = 2.5$ the new item that we want to insert in the list l in such a way that the resultant list l might remain sorted even after the insertion of the new item. In order to find the position where we have to insert the new item so that the resultant list might remain sorted, we split the initial list into two subsets between which there is the position of the new item allowing to the resultant list to remain sorted after the insertion. The two subsets that we obtain after the split are the following :
   $lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
   Therefore, the position of the new item is located between 2 and 3 that is at the 3^{rd} place in the list. The resultant list is given by : $resultant\_list = 1, 2, 2.5, 3, 4, 5$ and remains sorted in ascending order.

If now the new item that we want to insert in the list l is the following $new\_item = 3$, this item is already present in the list so the position where we have to insert the new item in the list is the most right position allowing to keep a sorted list. When we split into two subsets our initial list according this rule we obtain the following subsets :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Therefore, the right most position where we can inserting the new item so that the resultant list might remain sorted is between 2 and 3 that is at the 3^{rd} position in the list. We obtain the following resultant list :
$resultant\_list = 1, 2, \mathbf{3}, 3, 4, 5$
The initial list l contains an odd number of items : $n = 2k+1 = 2\times2+1$ so the number of items is not a multiple of 2 that is the number of items is not divisible by 2. Then we can constitute 2 subsets each containing 2 items and it remains 1 item between the 2 subsets that is the middle point of the list and that corresponds in our case to 3. This is the median of the list as the list is sorted in ascending order. When we insert a new item to the list, we have to split the list into two subsets between which we can find the position in the list where inserting the new item so that the list might remain sorted even after the insertion. As the initial list contains an odd number of items, then when we split the list into two subsets we never can obtain two subsets containing exactly the same number of items, when we approach at most to the equilibrium we obtain one subset containing a number of items equal to the half of the total number of items and another containing a number of items equal to the half of the total number of items plus one item :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4, 5$
Thus the new item can never be the middle point of the resultant list.
If we consider now an initial list $l = 2, 3, 1, 4$ containing an even number of items, then the number of items included in the initial list is exactly formed by an number integer of group of 2 items : $n = 2k = 2 \times 2$ so the number of items is a multiple of 2 that is the number of items is divisible by 2. Then we can constitute exactly 2 subsets containing each 2 items without that it remains additional item. In this case, as after the split there is no additional item, the middle point of the list does not belong to the list that is the median is not an item of the list, it corresponds to the average between the last item of the first half and the first item of the second half. Therefore, when we split the list into two subsets in order to find the position where we have to insert the new item so that the resultant list might remain sorted even after the insertion of the new item, in some particular case we can obtain two subsets exactly containing the same number of items equal to the half of the total number of items and in this particular case the new item is the middle point or the median of the resultant list.
For instance, if we have to insert the following item : $new\_item = 2.5$ in the sorted list $sorted\_list = 1, 2, 3, 4$ then we split the sorted list as follows :
$lower\_subset = 1, 2$ and $upper\_subset = 3, 4$
We obtain two subsets containing each two items and we have to insert the new item between the two halfs of the initial list that is at the 3^{rd} position. Therefore, the new item is the middle point or the median of the resultant list : $resultant\_list = 1, 2, \mathbf{2.5}, 3, 4$.

4. **The Insort Function : insort(list, itemToInsert, begin, end)**
   The parameters of this function are the following :

   - **list** is the sorted list that we have at our disposal.
   - **itemToInsert** is the item that we want to insert in our list without disturbing the order of the item.

- **begin** is the position of the first item of the sorted list.
- **end** is the position of the last item of the sorted list.

This function returns the resultant list after inserting the new item in appropriate position so that the resultant list might remain sorted. If the item that we want to insert into the list is already present in the list then, the new item is inserted at the right most possible position which allows to the resultant list to remain sorted even after the insertion of the new item.

5. **The Insort Left Function : insort_left(list, itemToInsert, begin, end)**
   The parameters of this function are the following :

   - **list** is the sorted list that we have at our disposal.
   - **itemToInsert** is the item that we want to insert in our list without disturbing the order of the item.
   - **begin** is the position of the first item of the sorted list.
   - **end** is the position of the last item of the sorted list.

   This function returns the resultant list after inserting the new item in appropriate position so that the resultant list might remain sorted. If the item that we want to insert in the list is already present in the list then, the new item is inserted at the left most possible position which allows to the resultant list to remain sorted even after the insertion of the new item.

6. **The Insort Right Function : insort_right(list, itemToInsert, begin, end)**
   This function performs exactly the same tasks than the insort function. The insort function is the function by default of the insort right function.

If we want to work with list of strings, we proceed in the same way but instead of comparing integers we compare the ASCII code of the letters of the strings.

9.2 Panda Library

Chapter 10

Bibliography

- MatchID Code Source :
  https://github.com/matchID-project/backend/tree/dev/code
- Lucene Code Source :
  https://github.com/apache/lucene-solr/blob/master/lucene/core/src/java/org/apache/lucene/util/a
- Article (Practical Vision)
  https://opensourceconnections.com/blog/2013/02/21/lucene-4-finite-state-automaton-in-10-minutes-intro-tutorial/
- Python Tutorial :
  `https://wiki.python.org/moin/FiniteStateMachine#FSA_-_Finite_State_Automation_in_Python`
- Levenshtein Automata :
  http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata `https://en.wikipedia.org/wiki/Levenshtein_automaton` `http://julesjacobs.github.io/2015/06/17/disqus-levenshtein-simple-and-fast.html`
- BK Tree :
  https://www.geeksforgeeks.org/bk-tree-introduction-implementation/ https://yomguithereal.github.
  tree.html http://www.martinbroadhurst.com/bk-tree.html https://gist.github.com/eiiches/2016232
  https://towardsdatascience.com/symspell-vs-bk-tree-100x-faster-fuzzy-string-search-spell-checking-c4f10d80a078 https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/
- Levenshtein Algorithm :
  https://www.codeproject.com/Tips/697588/Levenshtein-Edit-Distance-Algorithm
- DFA Minimization :
  `https://en.m.wikipedia.org/wiki/DFA_minimization` `https://en.m.wikipedia.org/wiki/Deterministic_finite_automaton`
- Symbol :
  `https://en.m.wikipedia.org/wiki/Symbol_(programming)`
- Alphabet :
  `https://en.m.wikipedia.org/wiki/Alphabet_(formal_languages)`
- Formal Language :
  `https://en.m.wikipedia.org/wiki/Formal_language`

- Formal Grammar :
  https://en.m.wikipedia.org/wiki/Formal_grammar
- Well-Formedness :

- DFA :
  https://en.wikipedia.org/wiki/Deterministic_finite_automaton https://en.m.wikipedia.org/wiki/Well-formedness
- Regular Expression :
  https://en.m.wikipedia.org/wiki/Regular_expression
- Complexity of an Algorithm :
  https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes
- Fuzzy Matching :
  https://en.wikipedia.org/wiki/Approximate_string_matching https://www.datacamp.com/community/tutorials/fuzzy-string-python
- Spell Checking:
  https://en.wikipedia.org/wiki/Spell_checker
- String:
  https://en.wikipedia.org/wiki/String_(computer_science)
- Pattern:
  https://en.wikipedia.org/wiki/Pattern
- Record Linkage (RL):
  https://en.wikipedia.org/wiki/Record_linkage http://www.linkagewiz.net/Whitepaper.pdf https://blog.couchbase.com/fuzzy-matching/
- NFA vs DFA :
  https://www.geeksforgeeks.org/theory-of-computation-conversion-from-nfa-to-dfa/
- Automaton Optimization :
  https://smart--grid.net/cours-lessons-theory/algorithm/time-complexity/
- MatchID Background :
  https://matchid-project.github.io/about
- Lucid Chart
- Python :
  http://www.xavierdupre.fr/app/teachpyx/helpsphinx/c_resume/python_sheet.html https://vincentchoqueuse.github.io/Python-pour-les-nuls/chapitre2/index.html#premier-programme https://openclassrooms.com/fr/courses/4452741-decouvrez-les-libra https://www.w3schools.com/python/python_dictionaries.asp
- Panda Library :
  http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html
- Bisect Module in Python :
  https://www.geeksforgeeks.org/bisect-algorithm-functions-in-python/

- Latex from Python :
  `https://pypi.org/project/PyLaTeX/0.6.1/`

Bibliography

[1] DSIC. Matchid code source. https://github.com/matchID-project/backend/tree/dev/code.

[2] furkanavcu. Levenshtein edit distance algorithm, on codeproject.com. https://www.codeproject.com/Tips/697588/Levenshtein-Edit-Distance-Algorithm.

[3] GeeksForGeeks. Bisect module in python. https://www.geeksforgeeks.org/bisect-algorithm-functions-in-python/.

[4] geeksforgeeks. Bk tree. https://www.geeksforgeeks.org/bk-tree-introduction-implementation/.

[5] Wikipedia. Alphabet in dfa minimization. https://en.m.wikipedia.org/wiki/Alphabet_(formal_languages).

[6] Wikipedia. Dfa. https://en.wikipedia.org/wiki/Deterministic_finite_automaton.

[7] Wikipedia. Dfa minimization. https://en.m.wikipedia.org/wiki/DFA_minimization.

[8] Wikipedia. Formal grammar in dfa minimization. https://en.m.wikipedia.org/wiki/Formal_grammar.

[9] Wikipedia. Formal language in dfa minimization. https://en.m.wikipedia.org/wiki/Formal_language.

[10] Wikipedia. Fuzzy matching. https://en.wikipedia.org/wiki/Approximate_string_matching.

[11] Wikipedia. Levenshtein automaton. https://en.wikipedia.org/wiki/Levenshtein_automaton.

[12] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance.

[13] Wikipedia. Pattern definition. https://en.wikipedia.org/wiki/Pattern.

[14] Wikipedia. Record linkage definition. https://en.wikipedia.org/wiki/Record_linkage.

[15] Wikipedia. Regular expression in dfa minimization. https://en.m.wikipedia.org/wiki/Regular_expression.

[16] Wikipedia. String definition. https://en.wikipedia.org/wiki/String_(computer_science).

[17] Wikipedia. Symbols in dfa minimization. https://en.m.wikipedia.org/wiki/Symbol_(programming).

[18] Wikipedia. Well-formedness in dfa minimization. https://en.m.wikipedia.org/wiki/Well-formedness.

[19] Xenopax'sBlog. Bk tree building. https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/.