**Introduction**

Damerau-Levenshtein is a string metric algorithm used in computer science to measure how different two string sequences, or simply their edit distance. There are numerous types of editing sequences that can be made in a string; a variety of the edit distance algorithm addresses to a selection of allowed type of edits. It aims to retrieve the minimum number of operations required to change the first string into the second. The 4 basic operations are insertion, deletion, substitution, and transposition of two adjacent characters.

The definition of the Damerau-Levenshtein distance between two strings *a* and *b,* and a function da,b(i,j), whose value is a distance between the symbol prefixes *i* and *j*, is:

* refers to the deletion operation (*a to b*)
* refers to the insertion operation (*a to b*)
* compares whether the respective symbols   
  match or mismatch.
* refers to the transposition of two adjacent symbols

The algorithm was named after Frederick J. Damerau and Vladimir I. Levenshtein, on which the latter introduced the edit distance concept and algorithm, while the former further improved the evaluation by integrating the comparison and transposition between adjacent characters. The improved algorithm admits all of edit operations from the former type, having in mind that the cost of a swap will be reduced to one instead of performing both deletion and insertion. It has enabled a fast-dynamic approach of solution to the problem. Damerau thought that transposition is also one of the most common operations in human misspellings.

The common applications of this algorithm include taking on a significant role in natural language processing. In natural languages, the number of errors usually do not exceed 2 as the strings involved in this processing are short. These circumstances put great impact on the difference between restricted and the real edit distance, making them almost similar. Another application is DNA sequencing and checking. Most of the time, the DNA gets edited and manipulated through insertions, deletions, substitutions and transpositions, and usually all of them occur at the same time. Damerau-Levenshtein can be used to compare strands of DNA to the target DNA to check the variation.

**Algorithm**

*Serial Implementation in Python*

Python is flexible when working with lists, which is why we chose this for the serial implementation of Damerau-Levenshtein. Since the language does not have an inherent multi-dimension array class, we have just created similar one, using a list. In the code, it shows the class we have constructed to pose as a multi-dimension array. The multi-dimension array constructed will have an equal number of columns and rows, by default if the number of columns is not assigned with a value. Each block of the list is initialized to zero thereafter.

|  |
| --- |
| class multidimen\_array(list):  def \_\_init\_\_(self, rows, cols=-1, blank=None):  if cols == -1:  cols = rows  for x in range(0, rows + 1):  self.append(list())  for y in range(0, cols + 1):  self[x].append(blank)  self.rows = rows  self.cols = cols |

Figure 1. Implementation of the multi-dimension array class in Python

To start off the implementation of the algorithm itself, a list of all the alphabet letters involved are retrieved and are indexed uniquely to form a “dictionary”. The alphabet dictionary will be hold the copy of the words inputted before being manipulated. In addition, it will also serve as basis for the next steps to determine where the changes took place in the string.

Sample Input-Output: (['a', 'b', 'c'], ['c', 'd']) -> {'a':0, 'b':1, 'c':2, 'd':3}

|  |
| --- |
| def getAlphabet(words):  alphabet = {}  i = 0  for wordList in words:  for word in wordList:  if word not in alphabet:  alphabet[word] = i  i = i + 1  return alphabet |

Figure 2. Implementation of the alphabet dictionary

The succeeding code below will be the actual implementation of the edit distance.

An array that will be used as the holder of the Damerau-Levenshtein edit distance between the first *i* characters of (a) the source string and the first *j* characters of (b) the comparing string. The maximum distance will then be stored at the last row and column of the array.

|  |
| --- |
| maxDist = len(a) + len(b) + 1    // assures all arrays are of same length  h = multidimen\_array(len(a) + 1, len(b) + 1, 0)  for i in range(0, len(a) + 1):  h[i + 1][1] = i  h[i + 1][0] = maxDist for j in range(0, len(b) + 1):  h[1][j + 1] = j  h[0][j + 1] = maxDist |

Following the pseudocode above religiously, the code below determines the distance of the substring. In Damerau-Levenshtein, transposition is computed by looking further backwards to find the match compared to the former Levenshtein algorithm. It specifically looks for:

1. The last row that contains the character of the current column
2. The last column in the said row previously where characters have matched

If both cases exist, they will define a cell together, or in our implementation, a state. The holder will then be referred to the earlier state of the process, before the transposition takes place. For the other operations, there are a few assumptions followed:

1. For deletion, simply count the characters between the characters that were transposed in the source string.
2. For insertion, simply count the characters between the characters that were transposed in the comparing string.

Knowing that the value of each of the cell in the array are the lowest cost of any of the operation, the differences happening in the middle of the transposition are needed not to be checked whether they were an insertion or a deletion. Instead, we just counted from both ends, and if both are not zero, the cost will be ignored and will be assumed to be too high.

From the inferences above, we have concluded that a transposition can be calculated as:

* – cost before transposition
* – distance between rows
* – distance between columns

|  |
| --- |
| for i in range(1, len(a) + 1):  db = 0 //previous index of the substring   for j in range(1, len(b) + 1):  i1 = da[alphabet[b[j - 1]]] //row of the character in current col  j1 = db  cost = 1  if (a[i - 1] == b[j - 1]): //where characters have matched  cost = 0  db = j   substitutionScore = h[i][j] + cost   insertionScore = h[i + 1][j] + 1  deletionScore = h[i][j + 1] + 1  transpositionScore = h[i1][j1] + (i - i1 - 1) + 1 + (j - j1 - 1)  //looks for the minimum cost for change  h[i + 1][j + 1] = min(  substitutionScore,  insertionScore,  deletionScore,  transpositionScore  )   da[alphabet[a[i - 1]]] = i  return h[len(a) + 1] |

*Parallel Implementation in Java*

The parallel implementation’s base code followed the algorithm mentioned above. The difference is that it traverses per column. It creates a thread for each element in each column; however, the program cannot move on to the next column without finishing the current one. This is to avoid errors in dependencies.

|  |
| --- |
| for(int i = 1; i <= m; i++){  ExecutorService es = Executors.newCachedThreadPool();  for (int j = 1; j <= n; j++) {  es.execute(new Thread(new DamerauThread(n, alphaNum, alphabet,  first, second, i, j, d), "i : " + i));  }  es.shutdown();  try {  boolean finshed = es.awaitTermination(1, TimeUnit.MINUTES);  Damerau.displayMatrixD(i+"");  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  } |

In parallel implementations, it is possible for it to use the wrong values to evaluate the current value. In order to avoid this scenario, we added another loop inside the thread to fine tune the values generated after each run of the thread.

**Analysis**

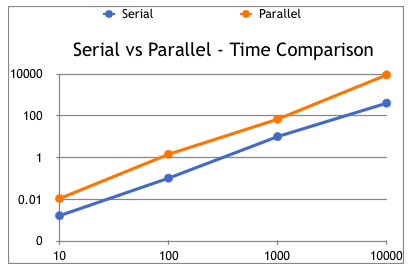
For each implementation, we did 3 trials for 10, 100, 1000 and 100000 characters and recorded the time it took for it to execute. The test cases is made of random characters generated by an online string generator.

*Time Consumption of Serial Implementation of the Algorithm*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of Characters** | **Trial 1** | **Trial 2** | **Trial 3** | **Average Time (in secs)** |
| 10 | 0.001999855042 | 0.001000165939 | 0.002000331879 | 0.001666784287 |
| 20 | 0.00600028038 | 0.004000425338745 | 0.004000425338745 | 0.004667043686 |
| 100 | 0.1060061455 | 0.1050057411 | 0.1030056477 | 0.1046725114 |
| 1000 | 7.821447372 | 11.87167907 | 11.1076355 | 10.26692065 |
| 10000 | 419.1546823 | 403.1654321354 | 411.654132746321 | 411.3247491 |

*Time Consumption of Parallel Implementation of the Algorithm*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of Characters** | **Trial 1** | **Trial 2** | **Trial 3** | **Average Time (in secs)** |
| 10 | 0.001999855042 | 0.001000165939 | 0.002000331879 | 0.001666784287 |
| 20 | 0.00600028038 | 0.004000425338745 | 0.004000425338745 | 0.004667043686 |
| 100 | 0.1060061455 | 0.1050057411 | 0.1030056477 | 0.1046725114 |
| 1000 | 7.821447372 | 11.87167907 | 11.1076355 | 10.26692065 |
| 10000 | 419.1546823 | 403.1654321354 | 411.654132746321 | 411.3247491 |

****

In the chart above, it is evident that the parallel implementation took a longer time to execute than the serial implementation. This is because the parallel implementation contains 3 loops whereas the serial implementation has 2 loops; making it O(n^3) and O(n^2) respectively. The programming language used should also be taken into consideration because the program used for serial and parallel implementation is Python and Java respectively. Being a list oriented language, Python can use and execute lists faster than Java. Java, being an object oriented program, can handle threads much better than Python.

**Conclusion**