Parallel implementation of Edit Distance in Java and OpenMP

Bianucci Stefano*

University of Florence
stefano.bianucci@stud.unifi.it

September 5, 2019

Abstract

In this paper, I explore an approache to parallelizing the edit distance problem and the related approximate string matching problem. The edit distance is a measure of the number of individual character insertions, deletions, and substitutions required to transform one string into another string. In the canonical dynamic programming solution to the edit distance, a chain of dependencies renders parallelization extremely difficult; thus, I investigate on a new implementation that uses the java threads and OpenMp library to improve performance.

I. Introduction

In computational linguistics and computer science, edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other.

Edit distances find applications in natural language processing, where automatic spelling correction can determine candidate corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question.

In bioinformatics, it can be used to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.

Given two strings a and b on an alphabet Îč the edit distance d(a, b) is the minimum-weight series of edit operations that transforms a into b.

The valid operations are:

- Insertion: ab -> axb
- Deletion: abc -> ac
- Substitution: abc -> xbc

*A thank you or further information

II. The sequenzial algorithm Idea

How to trasform this roles in sequenzial algorithm?

The algorithm starts with the upper left-hand corner of an two-dimensional array indexed in rows by the letters of the source word, and in columns by the letters of the target. It fills out the rest of the array while finding all the distances between each initial prefix of the source on the one hand and each initial prefix of the target. Each [i,j] cell represents the (minimal) distance between the first i letters of the source word and the first j letters of the target.

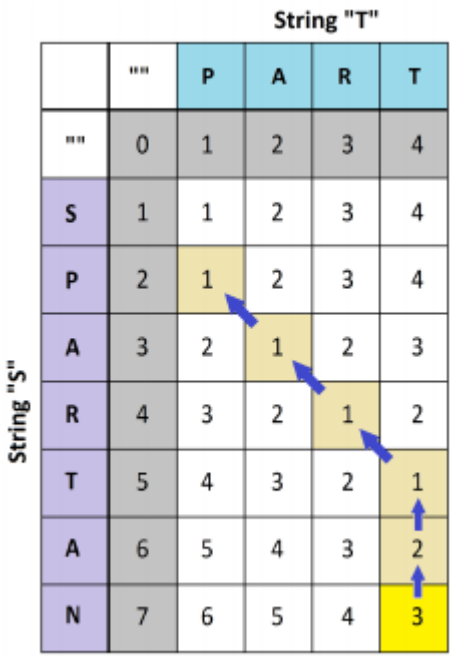
You can fill in the value of a cell only in case the values of all its neighbours upward and to the left have been filled in.

Top horizontal row is always 1, 2, etc in progress: this rapresents the cost of insertions

Left vertical column is always 1, 2, etc in progress: this rapresents the cost of deletions

Once cells up and to the left have been filled in, we can fill in the adjoining cell. We simply calculate all three values which might appear there – the result of deleting the letter indexing that column (added to the value directly above), the result of inserting the letter indexing the

row (added to the value to the left), and the result of replacing the column letter by the row letter (added to the value diagonally upward and to the left. [lower diagram above]



We can complete the entire array in this way, and once we have, we know the least costly set of operations that map one string to another. This involves some tracing back through the array, but the relevant cells are coloured contrastively in the demonstration. By following where the least costly values were found, we can see which operations are assumed to have resulted in the target string.

III. The sequenzial algorithm

I show my implementation where all is in EditDistance class in Java. Logical and text is very similar from java to cpp.

In Java implementation i need this attributes:

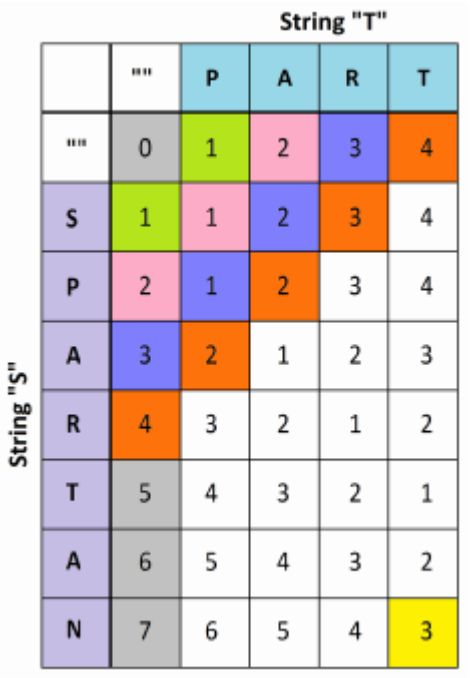
```
private String fs;

private String ss;

private int fsl;

private int ssl;

private int eM[][];
```

and this is the principal method that implementes the idea:

```
for (i = 0; i < this.fsl + 1; i++) {
   for (j = 0; j < this.ssl + 1; j++) {
      if (i == 0 || j == 0) {
         this.eMx[i][j] = Math.max(i, j);
      } else {
         this.eM[i][j] = min3(
               this.eM[i - 1][j] + 1,
               this.eM[i][j - 1] + 1,
               this.eM[i - 1][j - 1] +
                    (this.fs.charAt(i - 1)
                    != this.ss.charAt(j - 1)
                    ? 1 : 0));
      }
   }
}
```

IV. The parallel algorithm Idea

The key observation to our approach is that the entries of a single diagonal row of the DP table can be evaluated independently, provided that previous diagonals have been computed. To see this more clearly, refer to the next image. As can be seen, the entries of the blue diagonal (size 4) can be compuated independently once the values of the green and pink diagonal (sizes 2 and 3) have been evaluated. Similarly, the entries of the orange diagonal can be computed in parallel once the blue diagonal is complete. Our goal was the exploit this avenue of parallelism and implement a multi-threaded algorithm to compute the edit distance between very large strings.



Similar to before, only three diagonals are required at any point in order to obtain the edit distance. Since the length of the longest diagonal is equal to the length of the smaller input, we again have reduced our space requirements to ÎŸ(minM, N).

V. The parallel algorithm

I will show you how change from two difference linguages

Java threads:

```
case 2:

   EditThread oneTwo = new
        EditThread(editMatrix, 1,
        this.chunckHeight, 1,
        this.chunckWidth, firstString,
        secondString);
   oneTwo.start();
   while (oneTwo.isAlive()) {
   }

   EditThread twoTwo = new
        EditThread(editMatrix,
        this.chunckHeight + 1,
        this.firstStringLength, 1,
        this.chunckWidth, firstString,
         secondString);
   EditThread threeTwo = new
        EditThread(editMatrix, 1,
        this.chunckHeight, this.chunckWidth
        + 1, this.secondStringLength,
        firstString,
         secondString);
   twoTwo.start();
   threeTwo.start();
   while (oneTwo.isAlive() ||
        twoTwo.isAlive()) {
   }

   EditThread fourTwo = new
        EditThread(editMatrix,
        this.chunckHeight + 1,
        this.firstStringLength,
        this.chunckWidth + 1,
         this.secondStringLength,
             firstString, secondString);
   fourTwo.start();
   while (fourTwo.isAlive()) {
   }

   break;
```

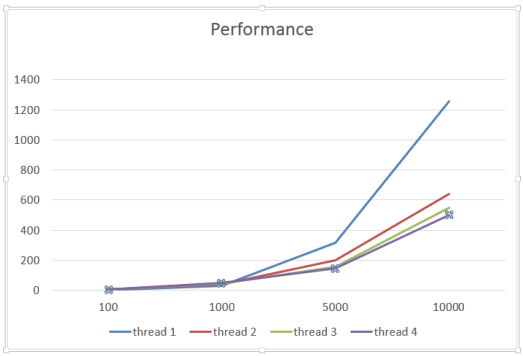
With OpenMP instead:

```
clock_t tStart = clock();
ed.calculate(0, 750, 0, 750);
#pragma omp parallel sections
{
    #pragma omp section
    ed.calculate(751, 1500, 0, 750);
    #pragma omp section
    ed.calculate(0, 750, 751, 1500);
}
ed.calculate(751, 1500, 751, 1500);
cout << "\n";
printf("Time taken: %.5fs\n",
     (double)(clock() -
     tStart)/CLOCKS_PER_SEC);
return 0;
```

VI. Results

The following is the speedup graph for our parallel diagonal implementation of edit distance.

The input sizes of both strings were fixed at 100, 1k, 5k 10k and measurements were taken with respect to a serial row-order implementation.



VII. Conclusions

In the end, I see that performance worsens on little data due to the costs of managing the process. With a lot of data this cost is irrelevant and we can see an important improvement even if linear.

VIII. Future work

Unfortunately, increasing the number of threads increases the difficulty of the algorithm and does not improve performance too much. We need to study new ways to improve further.

References

[Wikipedia] https://en.wikipedia.org/wiki/Edit$_d$*istance*