Université Jean Monnet

Machine Learning & Data Mining

Advanced Algorithms

Longest Common Subsequence

Edward Beeching
Jérémie Blanchard
Anthony Deveaux
Josselin Marnat
Joseph Renner

December, 12th 2016

Table of Contents

1 Introduction

The goal of this project is to build a plagiarism detector by using algorithms studied in class to find the Longest Common Subsequence (LCS) between two input texts. The Longest Common Subsequence is the longest subsequence that occurs in both texts. A subsequence is defined as a sequence that can be obtained from another only by deletion of elements, and not reordering. The elements need not be consecutive in the original text.
These are the algorithms implemented to find the LCS between two texts (described in detail in section 2.1):

- Dynamic Programming
- Divide and Conquer (linear space)
- Recursive
- Recursive with Branch and Bound

The project also includes the implementation of algorithms studied in class to solve the printing neatly problem. The printing neatly problem refers to the problem of printing a text while minimizing the number of empty spaces at the end of each line. Formally, the text is composed of n words where the number of characters in each word are represented as $l_1, l_2, \ldots, l_n$. The text is to be printed so each line holds a maximum of M characters. The number of extra spaces printed at the end of a line which contains words i through j is defined as follows:

$$M - j + i - \sum_{k=i}^{j} l_k$$

We want to minimize the sum of the cube of this non-negative value over all lines but the last. This is implemented using the following algorithms (described in detail in section 2.2):

- Dynamic Programming
- Greedy
- Recursive with Branch and Bound

Lastly, a study of the behavior of LCS algorithms and printing neatly algorithms is presented, including runtime analysis, space analysis, experimental runtime analysis and comparisons, and scalability.

2 Algorithms presentation

2.1 LCS

2.1.1 Dynamic programming

The dynamic programming algorithm for computing the LCS between two input texts (X, Y, with lengths m, n, respectively) is found as follows (adapted from Amaury Habrard's slides):

→ Algorithm 1 [Appendix]

→ Algorithm 2 [Appendix]

→ Algorithm 3 [Appendix]

Algorithm for building LCS from b matrix:

→ Algorithm 4 [Appendix]

LCS Linear Space Forwards: The following algorithm finds the length of the LCS of two input strings using dynamic programming with linear space complexity.

→ Algorithm 5 [Appendix]

LCS Linear Space Backwards: The following algorithm is a linear space implementation of the LCS dynamic programming algorithm, except it starts at the end of texts and works towards the beginning.

→ Algorithm 6 [Appendix]

2.1.2 Divide and Conquer

LCS Divide and Conquer: The following algorithm uses the LCS linear space forward and backward algorithms to compute the LCS of two input spaces in linear space, with the functionality to return the actual LCS (not just the length).
The algorithm is based on the following 2 properties:

1. The size of the LCS that passes through entry (i, j) is sum of the LCS from $(0, 0)$ to (i, j) (denoted $c[i, j]$) and the LCS from (i, j) to (m, n) (denoted g[i,j])

2. Let k be any number from 0 to n, let q be the number that maximizes $c[q, k] + g[q, k]$. There is an optimal solution to the LCS problem (from $0, 0$ to m, n) that passes through q, k

Combining these properties and the linear space algorithms presented above, we can formulate a divide and conquer algorithm that requires linear space.

→ Algorithm 7 [Appendix]

2.1.3 Recursive

LCS Recursive: The recursive algorithm for finding the LCS is a depth first search through the search space, enumerating all possible solutions and returning the largest LCS found.

→ `Algorithm 8 [Appendix]`

2.1.4 LCS Branch and bound

Bounds: The bounds used for this implementation were taken from the paper An Effective Branch-and-Bound Algorithm to Solve the k-Longest Common Subsequence Problem.

- The upper bound is completed as follows:

$$\mathrm{UB}_c = \sum_{\sigma\in\Sigma} \min_{i=1}^{k}(\text{number of characters } \sigma \text{ in sequence } x^{(i)})$$

- The lower bound is completed as follows:

$$\mathrm{LB}_c = \max_{\sigma\in\Sigma} \min_{i=1}^{k}(\text{number of characters } \sigma \text{ in sequence } x^{(i)})$$

To ensure to bound is computed in O(n) time, three hash tables (Python dicts) are used for fast lookup:

1. To count the occurrence of letters in list of words 1
2. To count the occurrence of letters in list of words 2
3. To count the common letters between hash table 1 and 2

For example for lists of letters [A,B,C,A] and [B,C,A,A], the common letters are A:2, B:1, C:1, the upper bound is found to be 2+2+1 = 4 and the lower bound is 2.

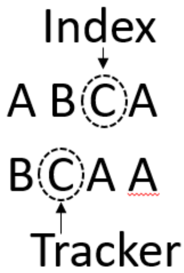
Implementation: The algorithm is implemented as a search tree, with a last in first out queue used for a depth first approach. The tree is explored node by node, with the upper bound evaluated for the two children at each step. Children with the higher upper bounds are placed on the queue last, to ensure the solutions with the most potential are explored first. Items are not placed on the queue if their bound is smaller than the best solution found so far. As elements are popped off the queue, if the bound exceeds the best solution found so far the node is explored. The items in the queue are a quadruplet containing (solution, index, tracker_index, upper_bound)
The solution is enumerate in binary as an integer, for example for strings:

- A,B,C,F
- B,C,D,E

A solution would be 0110 which would be represented as the integer 6, using a binary representation ensures the elements in the queue are as memory efficient as possible.
The "index" and "tracker" index in the quadruplet are the respective indexes of the in the two sets of strings.

→ `Algorithm 9 [Appendix]`

→ `Figure 1 [Appendix]`



source: https://www.researchgate.net/publication/220837450

2.2 Printing Neatly

Greedy Solution: The greedy solution simply prints as many words as possible on a line before going to the next. The solution it gives is not guaranteed to be optimal.

→ `Algorithm 10 [Appendix]`

Dynamic Programming: The dynamic programming approach computes an optimal solution in polynomial time. First, we define extras$[i, j]$ to be the number of extra spaces on a line containing words i through j. Using this, we can compute the cost of a line containing the same words:

1. lc$[i, j] = \infty$ if extras$[i, j]$ ¡ 0 (if words do not fit on the line)
2. lc$[i, j] = 0$ if $j = n$ and extras$[i, j] \geq 0$ (if it is the last line and the words fit)
3. lc$[i, j] =$ (extras$[i, j])^3$ otherwise

Using this, we can compute the optimal arrangement (minimal cost) of words 1..j = c[j] :

1. $c[j] = 0$ if $j = 0$
2. $c[j] = \min((1 \leq i \leq j), (c[i-1] + lc[i, j]))$ if $j > 0$

→ `Algorithm 11 [Appendix]`

Recursive: The recursive solution iterates through all the possible sequences and returns the optimal score and solution.

→ `Algorithm 12 [Appendix]`

Branch and Bound: Two branch and bound methods were compared, in both cases an initial estimate of the upper bound was evaluated using the the cost of a greedy approach. The first method then evaluates solutions in the same manner as the recursive, but when the cost of the current branch exceeds the bound given by the greedy solution, the recusion is cut and the algorithm back-tracks. The second algorithm makes an estimate of the cost of the remaining solution by using a greedy approach to find the cost of any remaining words, this enables to algorithm to cut potentially worse solutions early. The second approach is faster than the dynamic approach but unfortunately does not alway produce an optimal solution.

3 Preprocessing techniques

Introduction: We came up with four different preprocessing techniques to aid in detecting plagiarism. Each technique can be implemented on the two input texts and then the LCS is computed on the results of the preprocessing.

3.1 Light Preprocessing

The most basic preprocessing replaces certain symbols to make the texts more universal. In this technique, directional quotes (quotation marks that indicate if a quote is starting or finishing) are replaced by straight quotes, and extra newline characters and spaces are removed.

3.2 Advanced Preprocessing

Advanced preprocessing deletes all non-ascii characters and all symbols that are not periods or commas, and makes everything lower case. This is done to make the LCS more robust to small changes in punctuation. For example, the following is an text before preprocessing:

> This kind of relationship can be visualized as a tree structure, where 'student' would be the more general root node and both 'postgraduate' and 'undergraduate' would be more specialized extensions of the 'student' node (or the child nodes).

And here would be the same text after advance preprocessing:

> this kind of relationship can be visualized as a tree structure, where student would be the more general root node and both postgraduate and undergraduate would be more specialized extensions of the student node or the child nodes.

Stop Word Removal: For stop word removal preprocessing, an input text is run through the advanced preprocessing technique seen above and then passed to a function that deletes all occurrences of stop words. Stop words are any words that do not carry any significance to the similarity of two texts. We chose to remove them as a possible preprocessing technique to make the LCS robust to students simply adding in meaningless words to cover plagiarism. The stop words are listed in a file. Examples of stop words include "furthermore", "regardless", and "somehow".
Below is the same example sentence from above after stop word removal:

> kind relationship visualized tree structure, student would general root node post-graduate and undergraduate specialized extensions student node child nodes

Word Ordering Word ordering preprocessing starts by feeding an input text to the advanced preprocessing technique from above. Then, each sentence in the text is reordered so that the words in the sentence are arranged alphabetically. This is done so that texts that have copied and rearranged content from an original text are still detected as plagiarism.
Below is the same original sentence from above, after the word ordering preprocessing:

> a and and as be be be both can child extensions general kind of more more node nodes of or postgraduate relationship root specialized structure, student student the the the this tree undergraduate visualized where would.

4 Graphical User Interface

4.1 Plagiarism detector

The Graphical User Interface (GUI) uses a Python library called `PyQt4`.
In this first tab called 'LCS Unit', we have many separated areas splitted as follows:

→ `Figure 2 [Appendix]`

1. on the left: LCS inputs:
   (a) option bar:
   - `Task` : task from the corpus
   - `Text` : text from the corpus
   - `LCS preprocess` can be selected among:
     - `Raw`
     - `Advanced preprocessing`
     - `preprocessing`
     - `swr (stop words removal) preprocessing`
     - `WordOrgering preprocessing`
   - `LCS algorithm` can be selected among:
     - `LCS` (classic)
     - `LCS Sentence` (by sentence)

   (b) the text to compute from the corpus

   (c) the 'wiki' text to compare with

2. on the right: LCS results:
   (a) printing neatly section :
   - max-width slider (from 20 to 60)
   - algorithm chooser (Dynamic or Greedy)
   - the LCS printed neatly according to the width and the algorithm

   (b) some numbers resulted from the computation:
   - the original length of the text from the corpus
   - the LCS length
   - the plagiarism score, defined as follow :
     let T be the original text, and s_i the subsentences of T where words from the LCS are side-by-side in T (words in bold in the GUI). Then we can compute: score $= \sum_i (|s_i|)^2$
   - the actual running time to find the LCS
   - then, a pie chart showing the percentage of plagiarized text *vs.* the original one.

4.2 Data table

We used an Excel file (`corpus-final09.xls`) to store informations concerning the execution of all the algorithms on all the texts from the corpus.

→ Figure 3 [Appendix]

- [A] the name of the file
- [B:D] group, person and task
- [E:F] the category of plagiarism, native or not
- [G:H] knowledge and difficulty
- LCS Length and Ratios for:
  - [I:J] no preprocessing
  - [K:L] light preprocessing
  - [M:N] advanced preprocessing
  - [O:P] by sentence, advanced preprocessing
  - [Q:X] without stop words (again with no,advanced,light preprocessing, and alphabetical ordering)

4.3 Graphs

General graph: In this third tab, we are simply showing the LCS ratios (LCS length over text length) for all the files, and for the different preprocessing techniques. We can see that the more advanced the preprocessing, the longer the LCS: that is exactly what we wanted.

→ Figure 4 [Appendix]

Graph by task: The fourth tab shows the LCS Ratio for each task (and for each text of the corpus referring to this task), labeling with different colors the plagiarism category given by the Excel file. The labels are confirmed by this graph: we see that the 'cut' category as a much higher LCS ratio than the 'light' and 'heavy' ones, and the 'non' are very low.

→ Figure 5 [Appendix]

Repartition by category: This last tab is here to display the repartition of each LCS ratio over the plagiarism categories (non plagiarized, heavy modified, light modified and cut from the text)

→ Figure 6 [Appendix]

5 Experimental study

5.1 Time complexity

LCS Dynamic Programming: The dynamic programing algorithm for finding the length of the LCS requires $\mathcal{O}(mn)$ time, where m and n are the lengths (number of words) of the two input texts. This can be observed from the algorithm itself, as there is a for loop from 1 to n nested inside a for loop from 1 to m. The recursive algorithm used to build the solution from the b matrix requires $\mathcal{O}(m+n)$ time. Each run of the algorithm takes constant time, and each recursive call to the algorithm decreases at least one of the lengths by one. Combining these two results gives a time complexity of $\mathcal{O}(mn)$ to compute the length and build the LCS.

LCS Linear Space (Forwards and Backwards): The linear space versions of the dynamic programming algorithm do the same computations, with an added step at copying the row of the matrix after each row iteration. As a result, the asymptotic time complexity remains $\mathcal{O}(mn)$.

LCS Divide and Conquer: Let $T(m,n)$ denote the run time of the algorithm. At each call of the algorithm, it performs two $\mathcal{O}(mn)$ calls to the space-efficient algorithm. Then it makes two recursive calls on strings of size q and $\frac{n}{2}$, and $m-q$ and $\frac{n}{2}$, respectively. Then, for constant c, we have:

$$\begin{aligned} T(m,n) &\leq cmn + T(q,\frac{n}{2}) + T(m-q,\frac{n}{2}) \\ T(m,2) &\leq cm \\ T(2,n) &\leq cn \end{aligned}$$

If we assume $m = n$ and $q = \frac{n}{2}$, then we have:

$$T(m,n) \leq 2T(\frac{n}{2}) + cn^2$$

From the master theorem, we have $a = 2, b = 2, f(n) = n^2$, and $\log b(a) = 1$. Then, since $f(n)$ is $\Omega(n^c)$ where $c = 2 > \log b(a) = 1$, and the regularity condition holds, since $2(\frac{n^2}{4}) \leq kn^2$ with $k = \frac{1}{2}$, then it follows that $T(m,n) = \Theta(f(n)) = \Theta(n^2)$. So, when m=n, the run time is $\Theta(n^2)$. We can assume that when $m \neq n, T(m,n) \leq kmn$, and prove it by induction: assume $T(m',n') \leq km'n'$ for $m' < m$ and $n' < n$:

$$\begin{aligned} T(m,n) &\leq cmn + T(q,\frac{n}{2}) + T(m-q,\frac{n}{2}) \\ &\leq cmn + kq(\frac{n}{2}) + k(m-q)(\frac{n}{2}) && \text{(by induction hypothesis)} \\ &= cmn + kq(\frac{n}{2}) + km(\frac{n}{2}) - kn(\frac{n}{2}) \\ &= (c + k/2)mn. \end{aligned}$$

So, if $c = k/2$, then the proof works. Thus, the LCS divide and conquer has a time complexity of $\mathcal{O}(mn)$

LCS Recursive: The recursion for the recursive solution is as follows:

$$\begin{aligned} T(m,n) &= T(n,m-1) + T(n-1,m) + c \\ &= (T(n,m-2) + T(n-1,m-1) + T(n-1,m-1) + T(n-2,m)) + c' \\ &\geq 2T(n-1,m-1) \end{aligned}$$

Therefore the algorithm is exponential, since it is $\mathcal{O}(2^{\min(n,m)})$

LCS Branch and Bound: In the worst case, the branch and bound solution take the same time as the recursive. Thus, it is exponential. However, in practice, it is faster (see runtimes).

Printing Neatly Greedy Solution: In the greedy algorithm, there is just one iteration over the whole text, as a decision is made to print a word on the same line if possible or move to the next line. As a result, the time complexity is $\mathcal{O}(n)$ where n is the number of words in the text.

Printing Neatly Dynamic Programming: The dynamic programming algorithm can be broken into 3 sections: first, computing the extras matrix; next, computing the line cost matrix; and finally, computing the optimal solution.

1. The extras matrix is an $n \times n$ matrix where each element is computed in constant time. Thus, it is $\mathcal{O}(n^2)$.
2. The line cost matrix is also an $n \times n$ matrix with constant cost for each element, so it is $\mathcal{O}(n^2)$.
3. The optimal cost matrix is an array of length n, but the cost to compute the element at index i is a function of i. Thus, it is $\mathcal{O}(n^2)$.

Since all three main parts of the algorithm are $\mathcal{O}(n^2)$, the algorithm is $\mathcal{O}(n^2)$.

Printing Neatly Recursive: The recursive solution iterates through all possible solutions of lines that have at least one word on each line and have 0 or more spaces on the end (that fit). Let i be the number of words that can fit on the first line, then we have the following recursion:

$$T(n) = T(n-1) + T(n-2) + \cdots + T(n-i)$$

Expanding each recursion on the right hand side of the equation gives an exponential growth.

Printing Neatly Branch and Bound: In the worst case, the branch and bound extension behaves just as the plain recursive algorithm, thus it is also exponential. However, the bounding can limit the number of unneeded computations (see runtimes).

5.2 Runtime analysis

5.2.1 LCS

We have run the different algorithms (Branch & Bound, Recursive, Divide & Conquer and Dynamic) with a different number of words, to study their time complexity. Then, we plotted this data into three different graphs, in the goal to better see what happens.

Running times for different algorithms in linear scale / |words|: in this graph, the two scales are linear, and we can already observe the speed of each algorithms, and say that the Branch & Bound and the Recursive are much slower than the Divide & Conquer and the Dynamic algorithms.

→ `Figure 8 [Appendix]`

Running times for different algorithms in logarithmic scale / |words|: The have a better view of the complexity of the slower algorithms, we draw the same graph, but changing the time scale from linear to logarithmic. Now, we see that the Branch & Bound and the Recursive are linear in this scale, then we can deduce that, graphically, this two algorithms should be exponential.

→ `Figure 9 [Appendix]`

Running times for different algorithms in logarithmic scale / (|words|)^2: This time, we keep the time scale linear, but the x-axis becomes the number of words **squared**. We demonstrate that the running time of the dynamic algorithms is far more efficient than the recursive and branch and bounds methods, and the time complexity is quadratic in n (where $m = n$)

→ `Figure 10 [Appendix]`

5.2.2 Printing Neatly

→ `Figure 11 [Appendix]`

→ `Figure 12 [Appendix]`

→ `Figure 13 [Appendix]`

5.3 Space complexity

LCS Dynamic Programming : In the dynamic programming algorithm, there are two $m \times n$ matrices (m is length of first input text, n is length of second), one for the c matrix denoting optimal solutions, and one for the b matrix that is used to build the solution. These matrices are in addition to the two input word arrays and LCS word array. Thus, the space complexity is $\mathcal{O}(mn)$.

LCS Linear Space (forward and backward): In the linear space versions of the dynamic programming algorithm, we don't use the b matrix, and the c matrix is reduced to just 2 rows by n columns. Thus, the space complexity is $\mathcal{O}(n)$.

LCS Divide and Conquer: The divide and conquer algorithm uses the linear space algorithm on each half of the first input text and the full second text. The space complexity for these algorithms alone is $\mathcal{O}(n)$; however, we must save the full column indexed at $j = \frac{n}{2}$ in order to find the optimal row to divide our problem. Since there are m rows, the space needed to save the two columns (one for forwards LCS, one for backwards LCS) is $2m$. Adding this to the $\mathcal{O}(n)$ from the linear space algorithm, we have a space complexity of $\mathcal{O}(m + n)$.

LCS Recursive: Since the algorithm does a depth first search of the search space, a search will go all the way to a leaf node (base case) before backtracking back up the call tree. Each call to the function needs the two input texts that get smaller as the search gets closer to the base case. In terms of space used in addition to the input texts, the space complexity is constant. However, if a new copy of each input text is made for each recursive call, the space complexity becomes $\mathcal{O}(n^2 + m^2)$ in the worst case, which is when a leaf node is found in the search space and both input texts have been reduced to a length of 2 or less:

$$\begin{aligned} S(m,n) &= (m + (m-1) + ... + 1) + (n + (n-1) + ... + 1) \\ &= \frac{(m^2+m)}{2} + \frac{(n^2+n)}{2} \\ &= \mathcal{O}(m^2) + \mathcal{O}(n^2) \\ &= \mathcal{O}(m^2 + n^2) \end{aligned}$$

However, if pointers are used and new copies of the input texts are not made for each recursive call, then the space complexity become linear with respect to the input texts lengths.

LCS Branch and Bound: The space complexity for the the branch and bound extension is the same as recursive algorithm, except that a Queue of possible solutions to explore is also kept, which could potentially get quite large.

Printing Neatly Greedy: There are no additional matrices or arrays used to compute the greedy solution, as the input text is the only array stored. Thus, the space needed is linear, $\mathcal{O}(n)$ where n is the number of words in the input text.

Printing Neatly Dynamic Programming: The dynamic programming algorithm uses two matrices (one for extras, one for line cost) of size $n \times n$, where n is the number of words in the input text. Additionally, an array of size n is used to compute the optimal solution. Thus, the asymptotic space complexity is $\mathcal{O}(n^2)$.

Printing Neatly Recursive: There are no additional matrices or arrays used for the plain recursive solution. Thus, as long as pointers are used instead of copying a new input text array for each recursive call, the space complexity is $\mathcal{O}(n)$. If copies are made, then the space complexity would be $\mathcal{O}(n^2)$ (see LCS recursive space complexity analysis).

Printing Neatly Branch and Bound: The space complexity is that of the recursive solution plus an array of possible solutions to explore. This array is created at each level of recursion, and contains at most every word in the input. Since there are at most $n-1$ levels of recursion (in the case where there is just one word per line), the worst case space complexity would be $\mathcal{O}(n^2)$.

5.4 Outliers

If we look again at the Figure 5: Graph by task, we can see that there is three 'cut' texts that have a LCS ratio less than 70% (in task b and c). After looking each files (for example `g4pD_taskb.txt` with `orig_taskb.txt`), we concluded that this is an error by the original classification which was given to us.

5.5 Scalability

LCS: For input texts that are very large, it is clear that the LCS Divide and Conquer strategy would scale the best out of all of them. It runs in pseudo-polynomial time which is much faster than the recursive or branch and bound solutions. Also, the space improvement over the classic dynamic programming algorithm would be needed when the input texts are very large.

Printing Neatly: If an input text is very large, the recursive and branch and bound solutions would take too long. Furthermore, since the dynamic programming approach takes polynomial space and time, it would be unsuitable for extremely large input texts. A only solution left is the greedy solution, which takes linear space and time. It offers the best chance of scaling reasonably with large input texts. However, the solution is not guaranteed to be optimal. If an optimal solution was needed, the dynamic programming algorithm would offer the best scalability.

6 Project planning & difficulties

Planning

Difficulties: The main goal of coding the LCS algorithm was to implement what we saw during the course.
One difficulty that we ran into using Python to implement our algorithms was that Python is zero-indexed, so the first element in a list is indexed at 0. This was a problem because all the algorithms were implemented in the slides using one-indexed pseudo-code. We overcame this by having an empty element in the beginning of every list, to make them line up. Another difficulty we had was combining all of our work into one product. This was not a huge problem, but it did take some time to get all the functions working in different parts of the software.
Moreover most of the Algorithms in the slides of the course were very simple, and needed a lot of improvement and reflection before working in Python.

7 Conclusion

In term of algorithms this project was a good way to exploit each algorithm and approach, to determine which one is better than another and why. It was helpful to understand LCS algorithms in the goal to optimize them, and get an idea of which approach can be the most efficient to be implement in our user interface.
To sum up every member of our group were very concerned by the project. For some of us the language Python was unknown, and we can't say that we could have succeeded without the help of each other.
The interesting part was to manage the project: everything that was planned have been implemented. So thanks to our teamwork everybody gave his best.

Appendices

List of Algorithms

List of Figures

Algorithm 1: Optimal substructure

```
Data: texts X, Y with lengths m, n
for an LCS Z =< z_1, ··· , z_n > do
    if X_m = Y_n then
        X_m = Y_n = z_k;
        z_{k-1} is the LCS of X_{m-1} and Y_{n-1};
if X_m ≠ Y_n then
    z_k ≠ X_m ⇒ Z is the LCS of X_{m-1} and Y_n;
if X_m ≠ Y_n then
    z_k ≠ Y_n ⇒ Z is the LCS of X_m and Y_{n-1};
```

Algorithm 2: Recursive solution

```
Data: texts X, Y with lengths m, n
if i = 0 or j = 0 then
    c[i, j] = 0;
if i, j > 0 and X_i = Y_j then
    c[i, j] = c[i − 1, j − 1] + 1;
if i, j > 0 and xi ≠ yj then
    c[i, j] = max(c[i − 1, j], c[i, j − 1])
```

Algorithm 3: LCS-Length(input: X, Y)

```
m = |X|;
n = |Y|;
b[1..m, 1..n], c[0..m, 0..n];
for i ∈ 1..m do c[i, 0] = 0;
for j ∈ 0..n do c[0, j] = 0;
for i ∈ 1..m do
    for j ∈ 1..n do
        if xi = yj then
            c[i, j] = c[i − 1, j − 1] + 1;
            b[i, j] =′ d′ #for diagonal;
        else if c[i − 1, j] ≥ c[i, j − 1] then
            c[i, j] = c[i − 1, j];
            b[i, j] =′ u′ #for up;
        else
            c[i, j] = c[i, j − 1];
            b[i, j] =′ l′ #for left;
return c and b
```

Algorithm 4: Print-LCS(input: $b, X, |X|, |Y|$)

```
if i = 0 or j = 0 then
    return;
if b[i, j] = ‘d′ then
    Print-LCS(b, X, i − 1, j − 1)
else if b[i, j] = ‘u′ then
    Print-LCS(b, X, i − 1, j)
else
    Print-LCS(b, X, i, j − 1)
```

Algorithm 5: LCS_LSF (input: X, Y)

```
m = |X|;
n = |Y|;
c[1..2, 1..n];
col = [] #for use in divide and conquer alg (see next section);
for i ∈ 1..m do
    c[1, 0] = 0;
    for j ∈ 1..n do
        if xi = yj then
            c[1, j] = c[0, j − 1] + 1;
        else if c[1, j − 1] > c[0, j] then
            c[1, j] = c[1, j − 1];
        else
            c[1, j] = c[0, j];
    c[0, :] = c[1, :];
    col.append(c[1, n]);
return c[1, length_Y], col #return length of LCS, column
```

Algorithm 6: LCS_LSB(input: X, Y)

```
m = |X|;
n = |Y|;
c[1..2, 1..n];
col = [] #for use in divide and conquer alg (see next section);
for i ∈ m − 1..0 do
    for j ∈ n − 1..0 do
        if X[i + 1] = Y[j + 1] then
            c[1, j] = c[0, j + 1] + 1;
        else if c[1, j + 1] > c[0, j] then
            c[1, j] = c[1, j + 1];
        else
            c[1, j] = c[0, j];
    c[0, :] = c[1, :];
    col.insert(0, c[1, 0]);
return c[1, 0], col #length of LCS, column
```

Algorithm 7: LCS_DC(input: X, Y)

$m = |X|$;
$n = |Y|$;
if $m * n = 0$ **then**
  return [] `#if one of the texts is empty, return an empty LCS`
else if $m \leq 2$ *or* $n \leq 2$ **then**
  return *LCS_DyProg(X,Y)* `#if one of the texts is small, can do normal Dynamic Prog`
else
  breakpt = floor($n/2$);
  length, c = LCS_LSF(X, Y[0 : breakpt]);
  length2, $c2$ = LCSLSB(X, Y[breakpt : n]);
  q = maxindex($c, c2$) `#q is the index that maximizes (`$c[q] + c2[q]$`);`
  LCSL = LCS_DC($X[0..q], Y$[0..breakpt]);
  LCSR = LCS_DC($X[q+1, m], Y$[breakpt + 1, n]);
  return $LCSL + LCSR$ `#return concatenation of sub LCS`

Algorithm 8: LCS_Recursive(input: X, Y)

$m = |X|$;
$n = |Y|$;
if $X[m] = Y[n]$ **then**
  return *LCS_Recursive(*$X[0 : m-1], Y[0 : n-1]$*)*;
else
  LCS1 = LCS_Recursive($X[0 : m-1], Y[0 : n]$);
  LCS2 = LCS_Recursive($X[0 : m], Y[0 : n-1]$);
  if $|LCS1| > |LCS2|$ **then** **return** *LCS1* ;
  else **return** *LCS2* ;

Algorithm 9: Brand_and_bound(inputs: X, Y)

```
max_lcs_length = min(X.length, Y.length);
best = find_lower_bound(X, Y) − 1;
starting_solution = (1 << max_lcs_length) − 1;
best_solution = 0;
index = -1;
tracker = -1;
LIFO_queue.put([starting_solution, index, tracker, best]);
while LIFO_queue is not empty do
    node = LIFO_queue.get();
    if node.index = max_lcs_length and node.best > best then
        best = node.best;
        best_solution = node.solution;
    else
        if node.bound > best then
            child1 = node.child1 child2 = node.child2
        if find_upper_bound(child1) > find_upper_bound(child2) then
            if child2.bound > best then LIFO_queue.add(child2);
            if child1.bound > best then LIFO_queue.add(child1);
return best_solution;
```

Algorithm 10: PN_Greedy(input: X (list of words to be printed), L (number of spaces per line))

```
lines = [];
currentLine = [];
currentLineSpace = 0;
for i in X do
    if currentLineSpace + (|X[i]| + 1) ≤ L then
        currentLineSpace += |X[i]| + 1 #update space taken on current line (for
          space between words);
        currentLine += " " + X[i];
    else
        lines.append(currentLine);
        currentLineSpace = |X[i]|;
        currentLine = X[i];
lines.append(currentLine);
return lines;
```

Algorithm 11: PN_DP(input: X, L)

Compute extras$[i, j]$,$1 = i, j = n$;
Compute lc$[i, j]$,$1 = i, j = n$;
$c[0] = 0$;
for $j \in 1..n$ **do**
  $c[j] = \min((1 \leq i \leq j), (c[i-1] + lc[i, j]))$;
  $p[j] = k$ s.t. $\min(1 \leq i \leq j)(c[i-1] + lc[i, j]) = c[k-1] + lc[k, j]$ `#store where the cuts are`;
return p

Algorithm 12: PN_Recursive(input: X, L)

result = [] `#where the lines will be stored`;
max_words = the maximum number of words that can fit on current line `#what the greedy solution would return`;
if *max_words* $\geq |X|$ **then**
  return $0, X[0 : |X|]$ `#base case: where all words inputed will fit on current line`;
best_score = ∞;
for $i \in 1..$*max_words* **do**
  subscore = 0 `#current line space taken`;
  for $j \in 0..i-1$ **do**
    subscore += $|X[j]| + 1$ `#add current word and space`;
  subscore -= 1 `#remove last space`;
  subscore = $(L - \text{subscore})^3$ `#compute cost of space left over`;
  score, res = PN_Recursive($X[i : n], L$);
  if *score + subscore ¡ best_score* **then**
    result.append($X[0..i]$);
    result += res;
return *best_score, result*

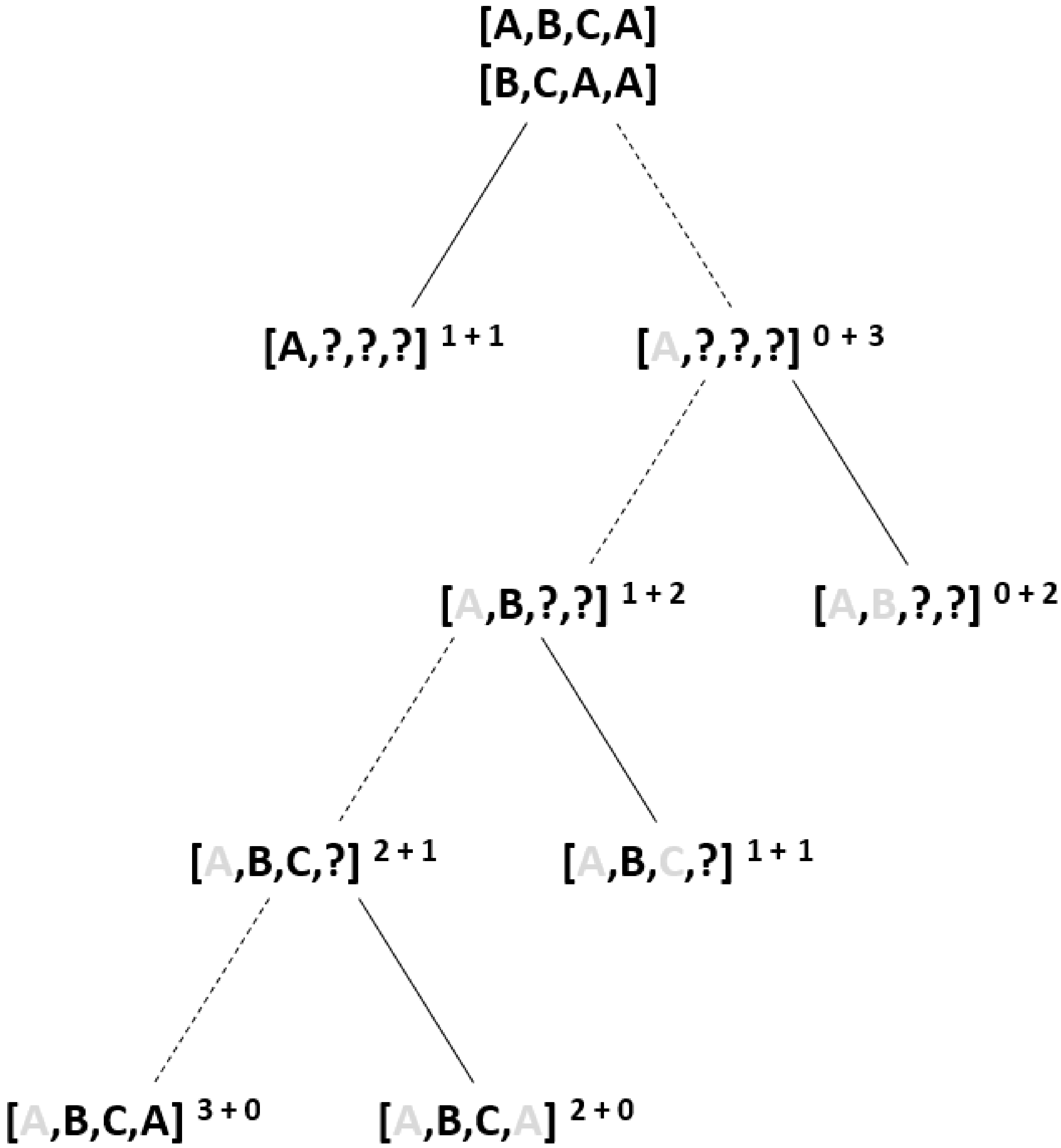


Figure 1: Path to the solution of the LCS of [A,B,C,A] and [B,C,A,A]

The path explored first shown as a dashed line, the super-script numbers at the top right show the score and upper bounds computed at each stage.

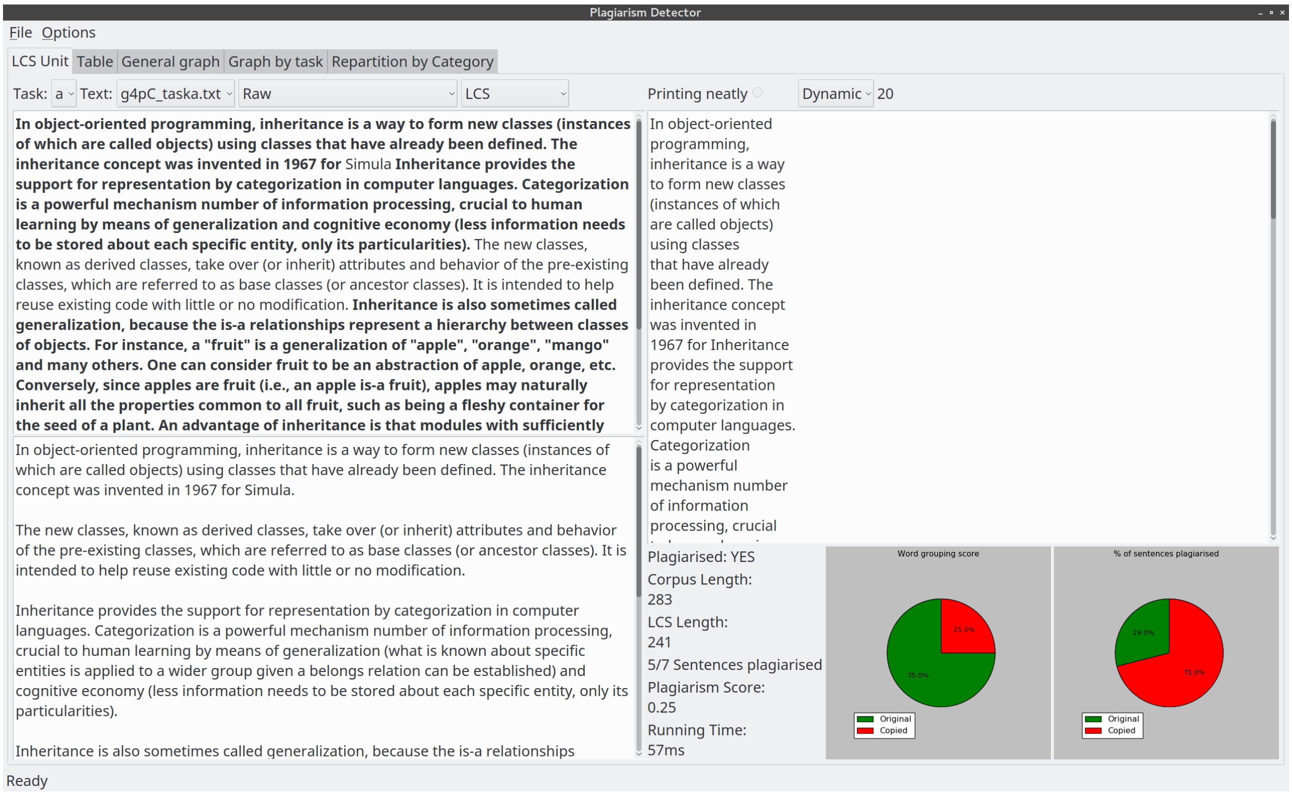
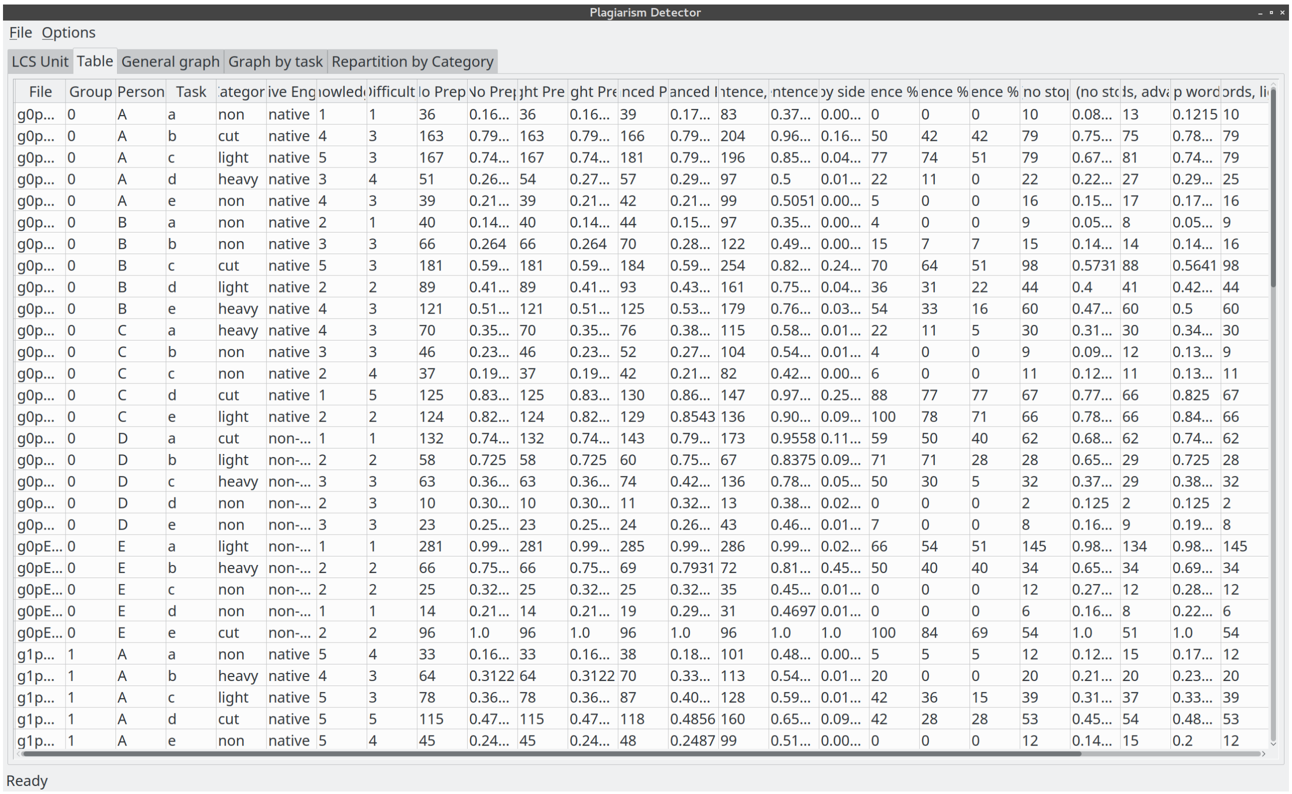


Figure 2: LCS Unit



File	Group	Person	Task	ategor	ive Eng	owled	ifficult	o Prep	No Pre	ht Pre	ght Pre	nced P	anced I	ntence,	ntence	y side	ence %	ence %	ence %	no sto	(no st	ls, adv	p word	ords, li
g0p...	0	A	a	non	native	1	1	36	0.16...	36	0.16...	39	0.17...	83	0.37...	0.00...	0	0	0	10	0.08...	13	0.1215	10
g0p...	0	A	b	cut	native	4	3	163	0.79...	163	0.79...	166	0.79...	204	0.96...	0.16...	50	42	42	79	0.75...	75	0.78...	79
g0p...	0	A	c	light	native	5	3	167	0.74...	167	0.74...	181	0.79...	196	0.85...	0.04...	77	74	51	79	0.67...	81	0.74...	79
g0p...	0	A	d	heavy	native	3	4	51	0.26...	54	0.27...	57	0.29...	97	0.5	0.01...	22	11	0	22	0.22...	27	0.29...	25
g0p...	0	A	e	non	native	4	3	39	0.21...	39	0.21...	42	0.21...	99	0.5051	0.00...	5	0	0	16	0.15...	17	0.17...	16
g0p...	0	B	a	non	native	2	1	40	0.14...	40	0.14...	44	0.15...	97	0.35...	0.00...	4	0	0	9	0.05...	8	0.05...	9
g0p...	0	B	b	non	native	3	3	66	0.264	66	0.264	70	0.28...	122	0.49...	0.00...	15	7	7	15	0.14...	14	0.14...	16
g0p...	0	B	c	cut	native	5	3	181	0.59...	181	0.59...	184	0.59...	254	0.82...	0.24...	70	64	51	98	0.5731	88	0.5641	98
g0p...	0	B	d	light	native	2	2	89	0.41...	89	0.41...	93	0.43...	161	0.75...	0.04...	36	31	22	44	0.4	41	0.42...	44
g0p...	0	B	e	heavy	native	4	3	121	0.51...	121	0.51...	125	0.53...	179	0.76...	0.03...	54	33	16	60	0.47...	60	0.5	60
g0p...	0	C	a	heavy	native	4	3	70	0.35...	70	0.35...	76	0.38...	115	0.58...	0.01...	22	11	5	30	0.31...	30	0.34...	30
g0p...	0	C	b	non	native	3	3	46	0.23...	46	0.23...	52	0.27...	104	0.54...	0.01...	4	0	0	9	0.09...	12	0.13...	9
g0p...	0	C	c	non	native	2	4	37	0.19...	37	0.19...	42	0.21...	82	0.42...	0.00...	6	0	0	11	0.12...	11	0.13...	11
g0p...	0	C	d	cut	native	1	5	125	0.83...	125	0.83...	130	0.86...	147	0.97...	0.25...	88	77	77	67	0.77...	66	0.825	67
g0p...	0	C	e	light	native	2	2	124	0.82...	124	0.82...	129	0.8543	136	0.90...	0.09...	100	78	71	66	0.78...	66	0.84...	66
g0p...	0	D	a	cut	non-...	1	1	132	0.74...	132	0.74...	143	0.79...	173	0.9558	0.11...	59	50	40	62	0.68...	62	0.74...	62
g0p...	0	D	b	light	non-...	2	2	58	0.725	58	0.725	60	0.75...	67	0.8375	0.09...	71	71	28	28	0.65...	29	0.725	28
g0p...	0	D	c	heavy	non-...	3	3	63	0.36...	63	0.36...	74	0.42...	136	0.78...	0.05...	50	30	5	32	0.37...	29	0.38...	32
g0p...	0	D	d	non	non-...	2	3	10	0.30...	10	0.30...	11	0.32...	13	0.38...	0.02...	0	0	0	2	0.125	2	0.125	2
g0p...	0	D	e	non	non-...	3	3	23	0.25...	23	0.25...	24	0.26...	43	0.46...	0.01...	7	0	0	8	0.16...	9	0.19...	8
g0pE...	0	E	a	light	non-...	1	1	281	0.99...	281	0.99...	285	0.99...	286	0.99...	0.02...	66	54	51	145	0.98...	134	0.98...	145
g0pE...	0	E	b	heavy	non-...	2	2	66	0.75...	66	0.75...	69	0.7931	72	0.81...	0.45...	50	40	40	34	0.65...	34	0.69...	34
g0pE...	0	E	c	non	non-...	2	2	25	0.32...	25	0.32...	25	0.32...	35	0.45...	0.01...	0	0	0	12	0.27...	12	0.28...	12
g0pE...	0	E	d	non	non-...	1	1	14	0.21...	14	0.21...	19	0.29...	31	0.4697	0.01...	0	0	0	6	0.16...	8	0.22...	6
g0pE...	0	E	e	cut	non-...	2	2	96	1.0	96	1.0	96	1.0	96	1.0	1.0	100	84	69	54	1.0	51	1.0	54
g1p...	1	A	a	non	native	5	4	33	0.16...	33	0.16...	38	0.18...	101	0.48...	0.00...	5	5	5	12	0.12...	15	0.17...	12
g1p...	1	A	b	heavy	native	4	3	64	0.3122	64	0.3122	70	0.33...	113	0.54...	0.01...	20	0	0	20	0.21...	20	0.23...	20
g1p...	1	A	c	light	native	5	3	78	0.36...	78	0.36...	87	0.40...	128	0.59...	0.01...	42	36	15	39	0.31...	37	0.33...	39
g1p...	1	A	d	cut	native	5	5	115	0.47...	115	0.47...	118	0.4856	160	0.65...	0.09...	42	28	28	53	0.45...	54	0.48...	53
g1p...	1	A	e	non	native	5	4	45	0.24...	45	0.24...	48	0.2487	99	0.51...	0.00...	0	0	0	12	0.14...	15	0.2	12

Figure 3: Table

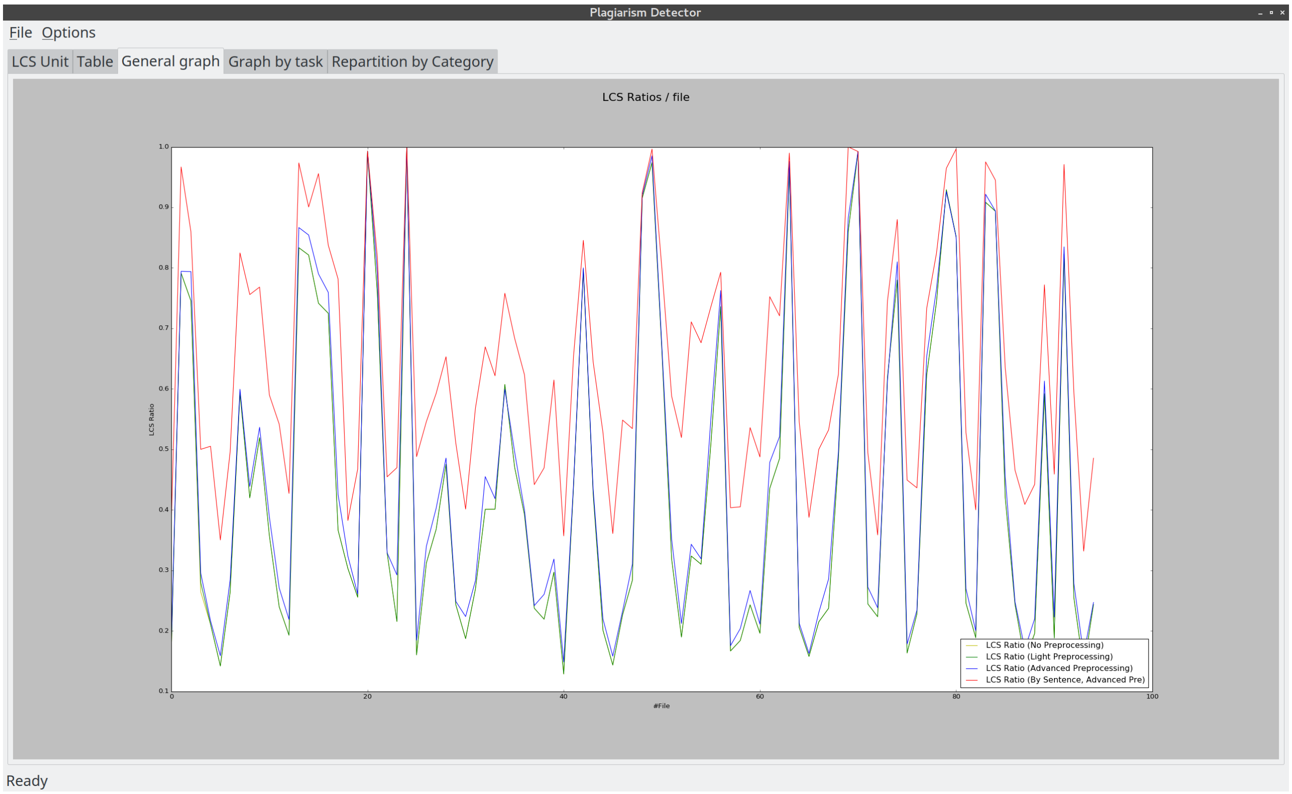


Figure 4: General graph

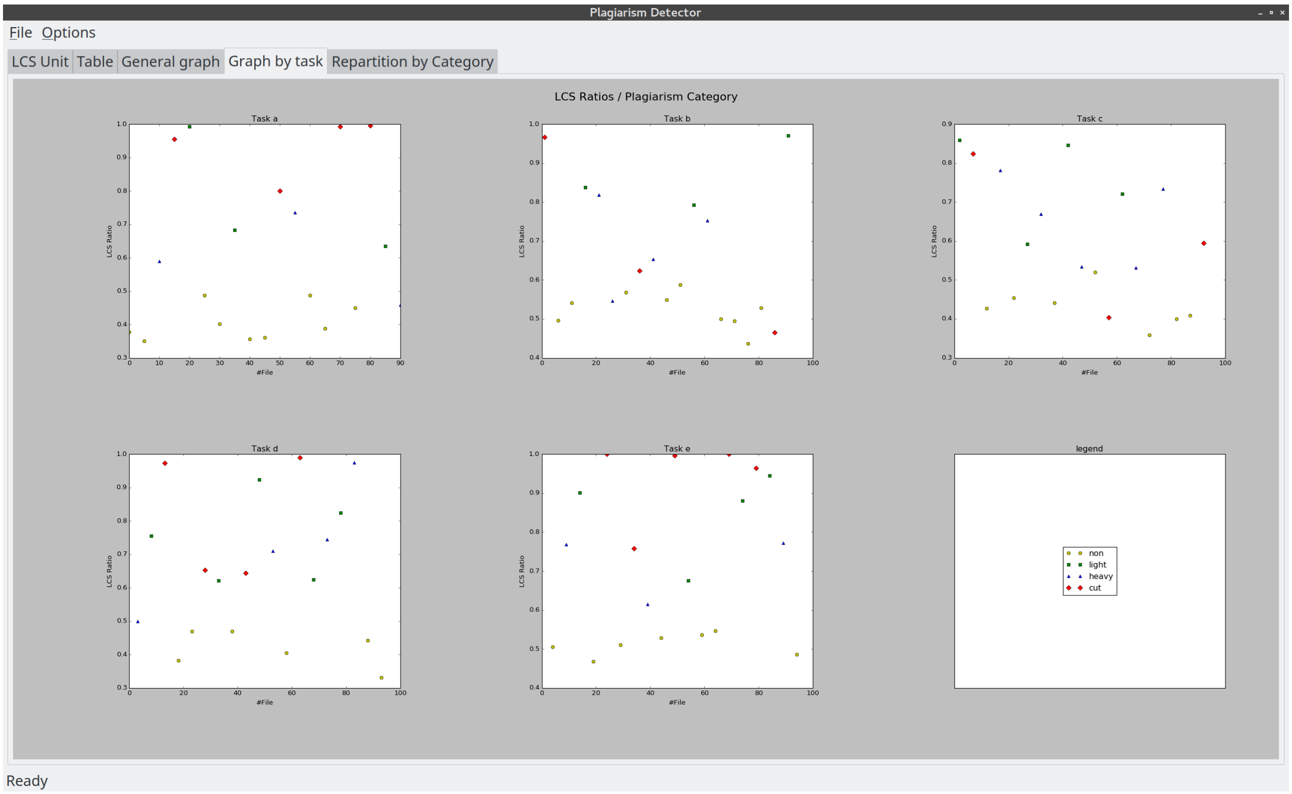


Figure 5: Graph by task

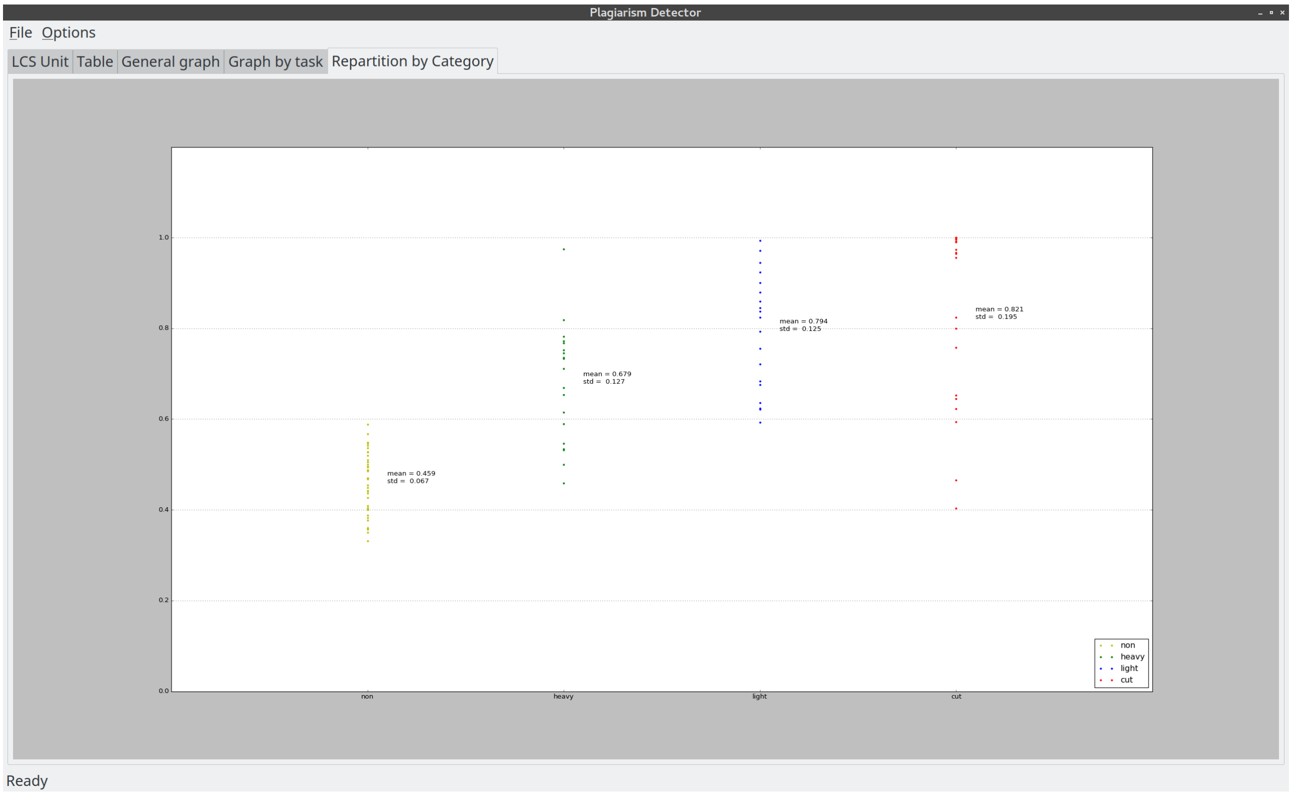


Figure 6: Repartition by Category

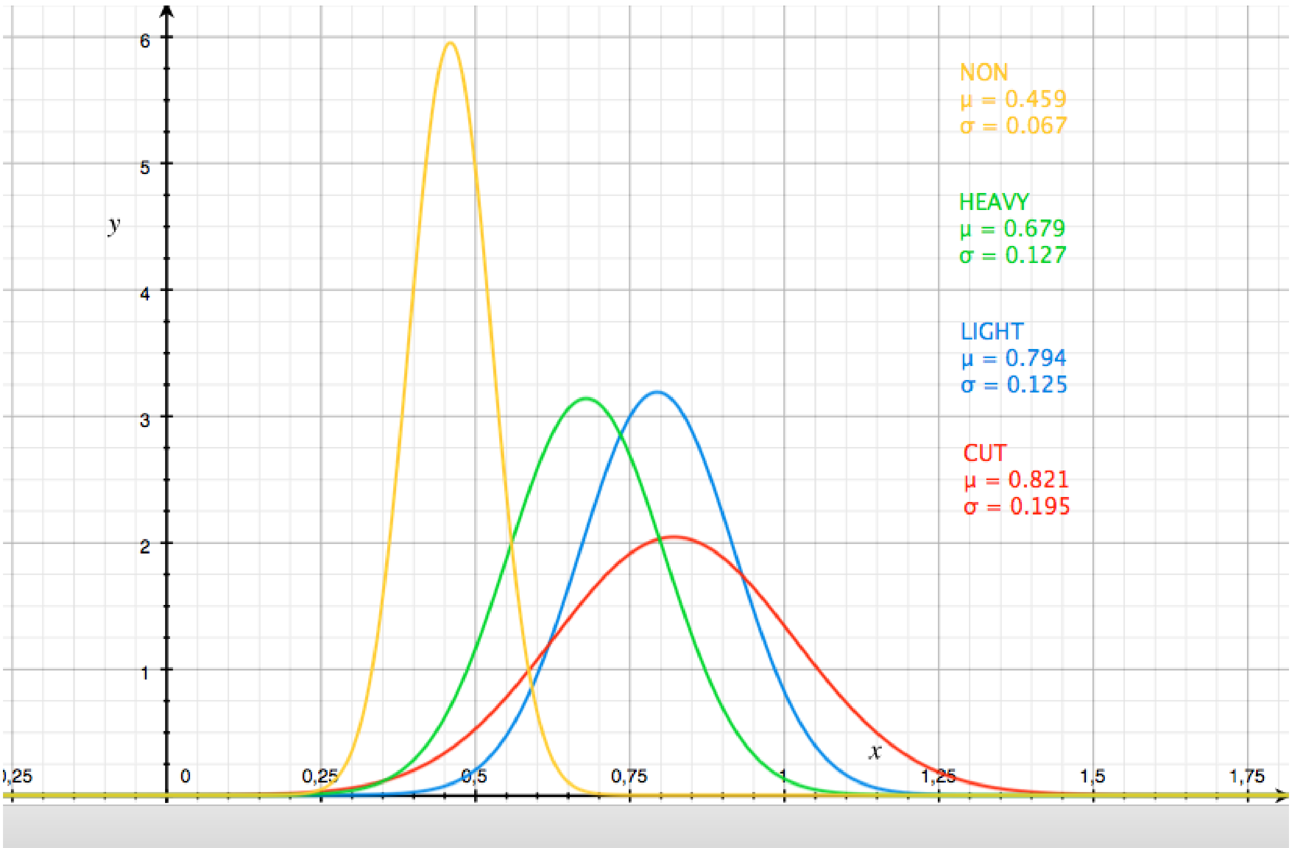


Figure 7: Normal distributions of the repartition by category / |words|

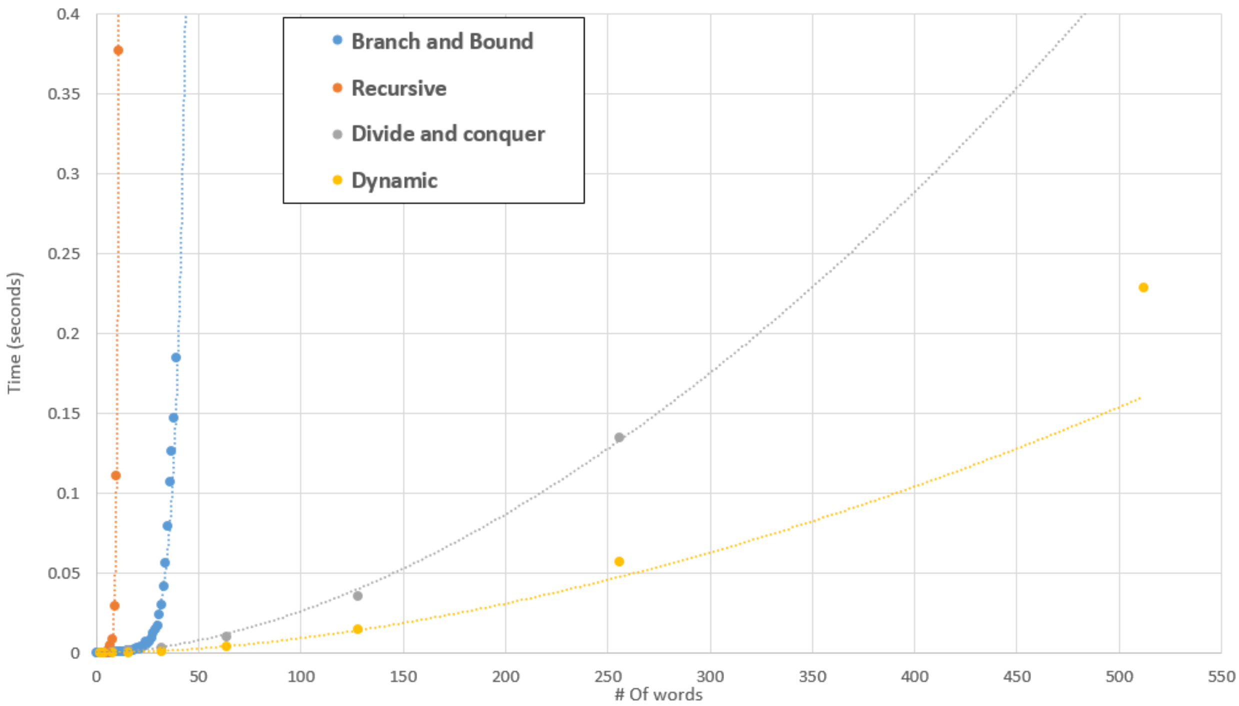


Figure 8: Running times for different algorithms in linear scale / |words|

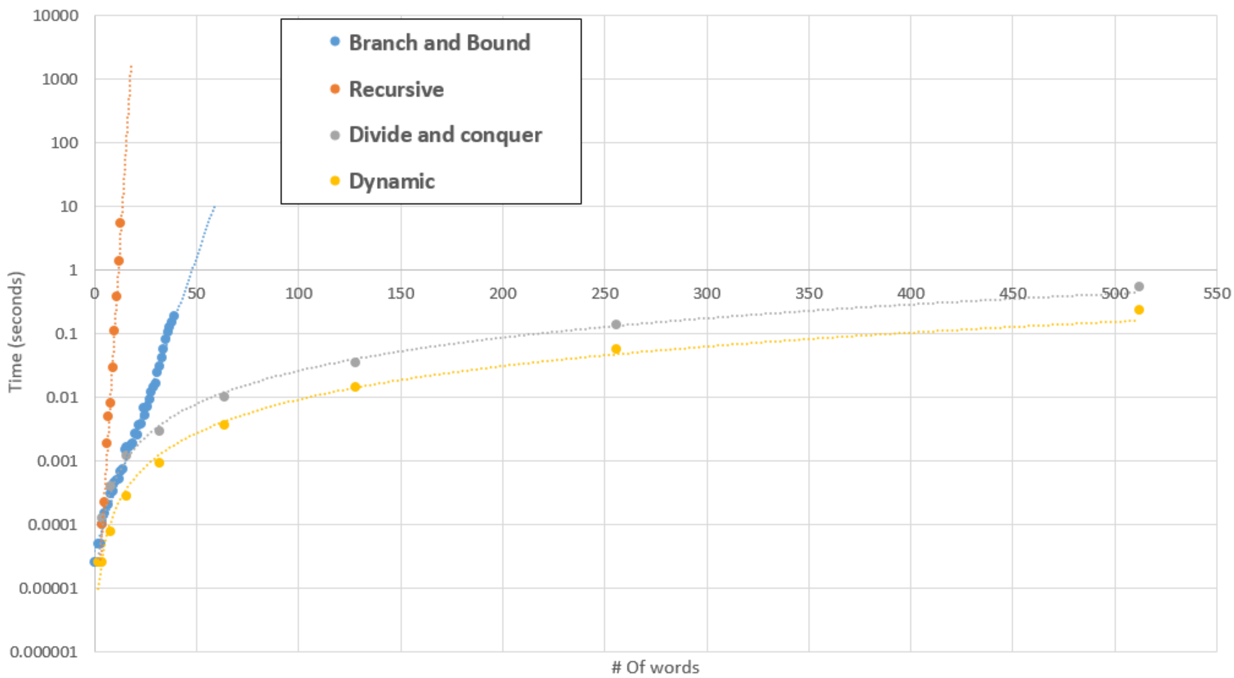


Figure 9: Running times for different algorithms in logarithmic scale / |words|

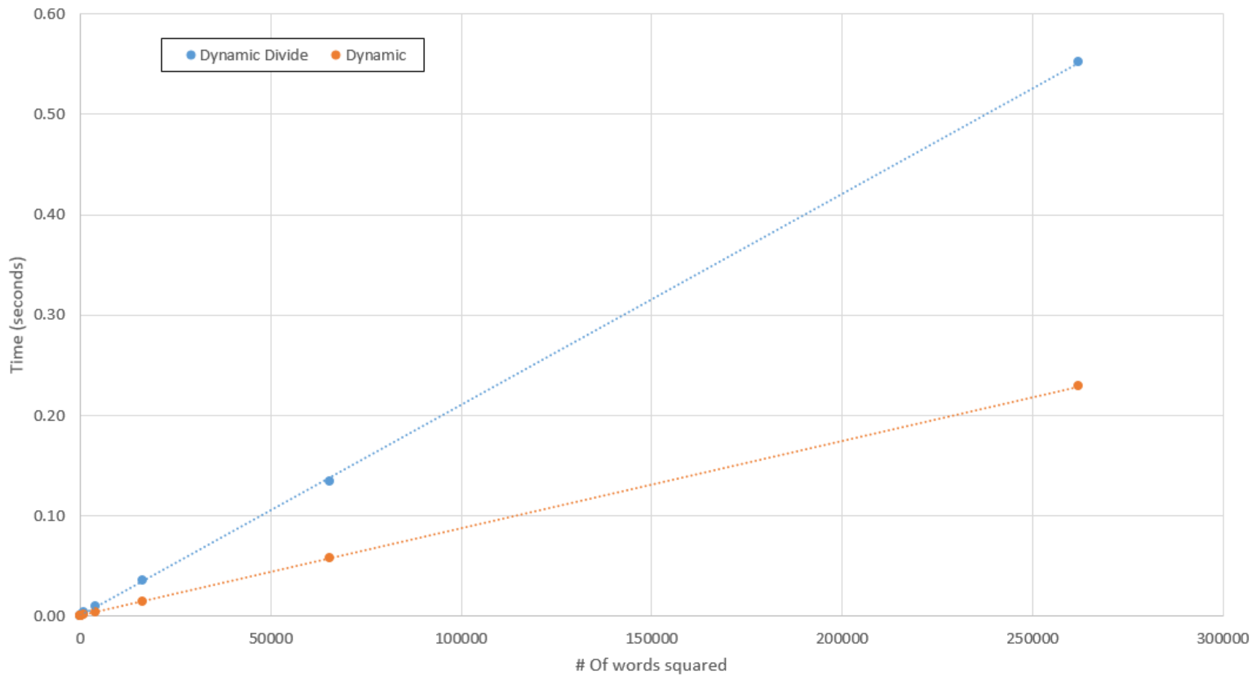


Figure 10: Running times for different algorithms in logarithmic scale / $(|\text{words}|)^2$

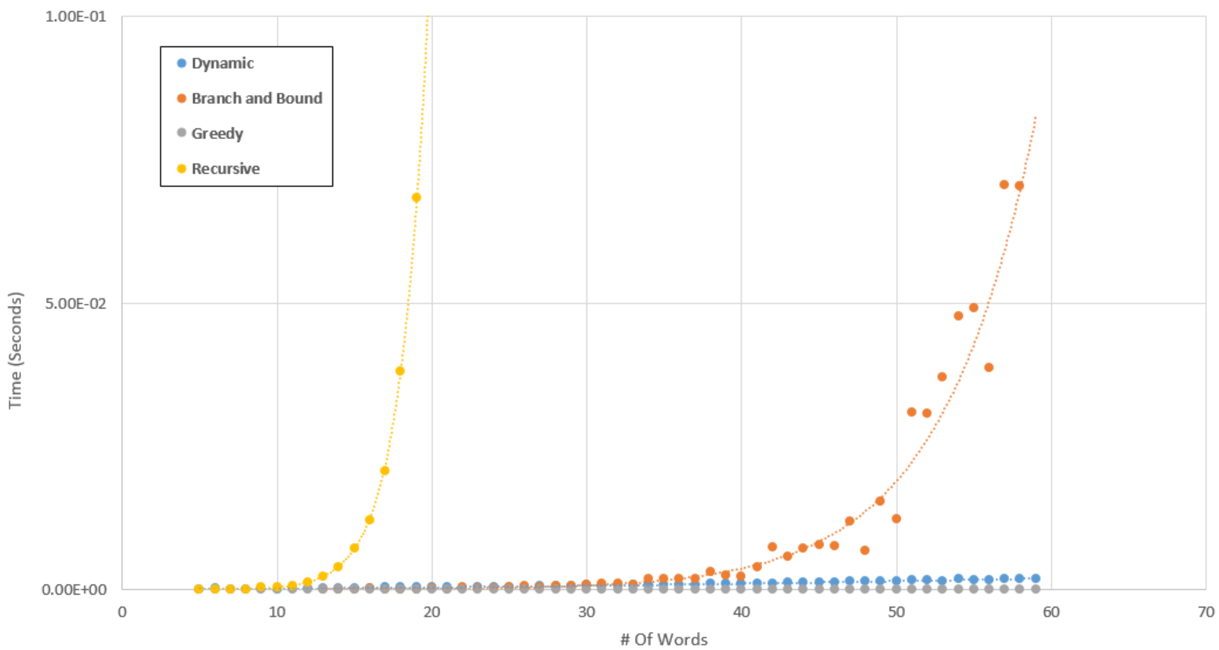


Figure 11:

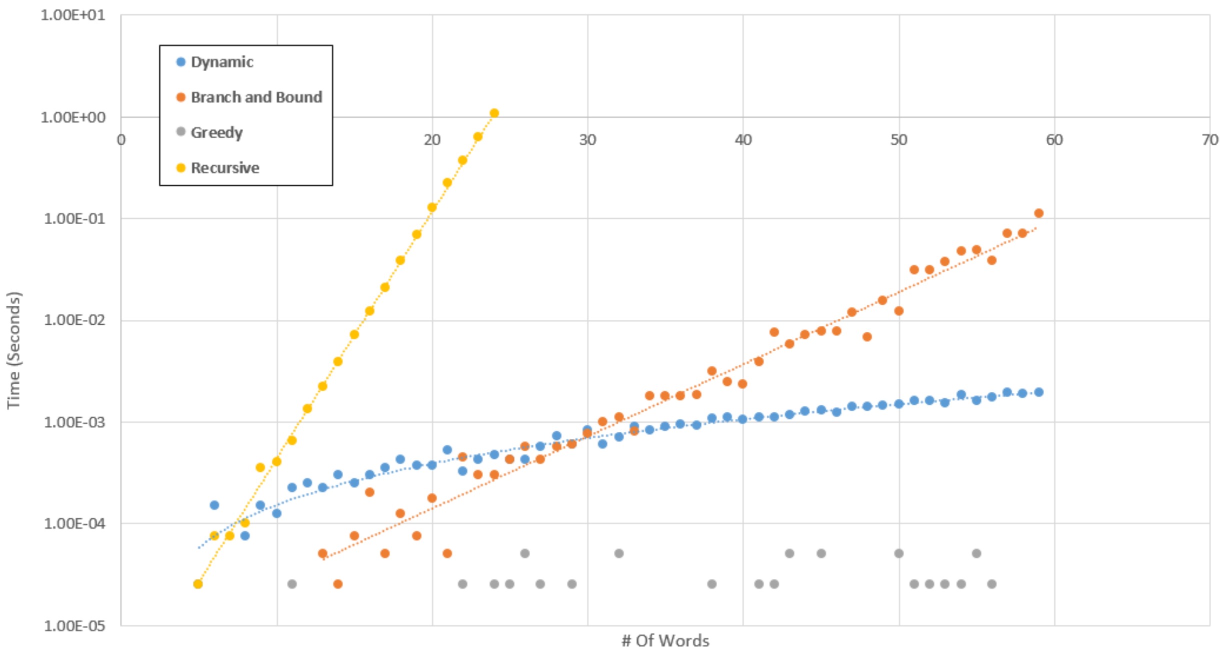


Figure 12:

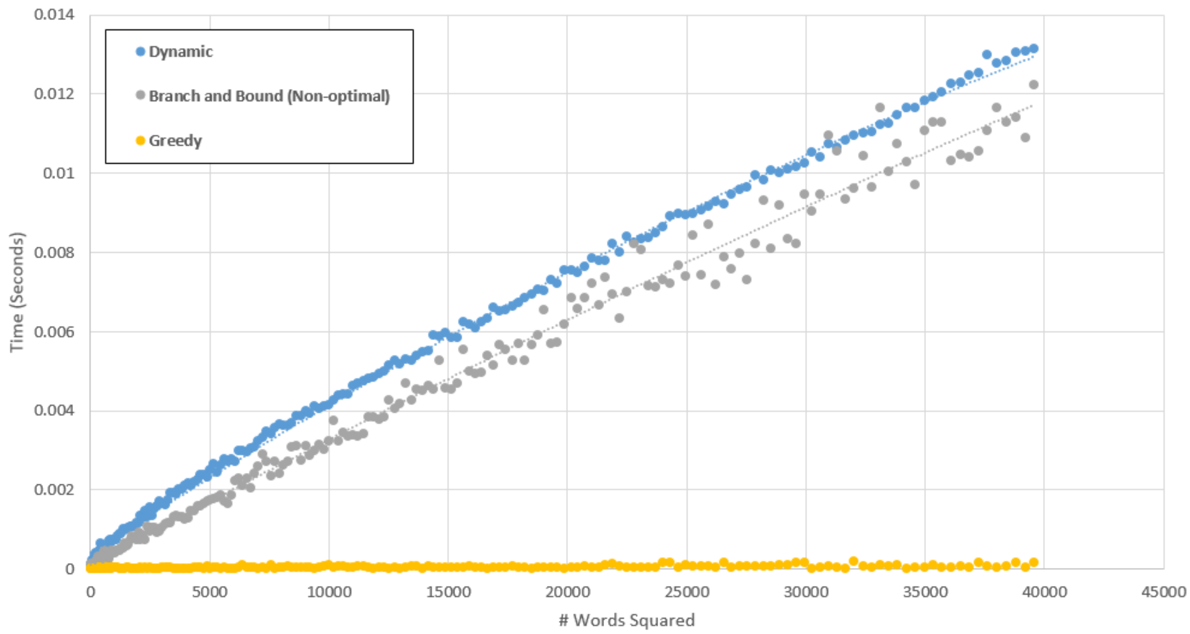


Figure 13: Comparition between the Dynamic and the Branch & Bound algorithms

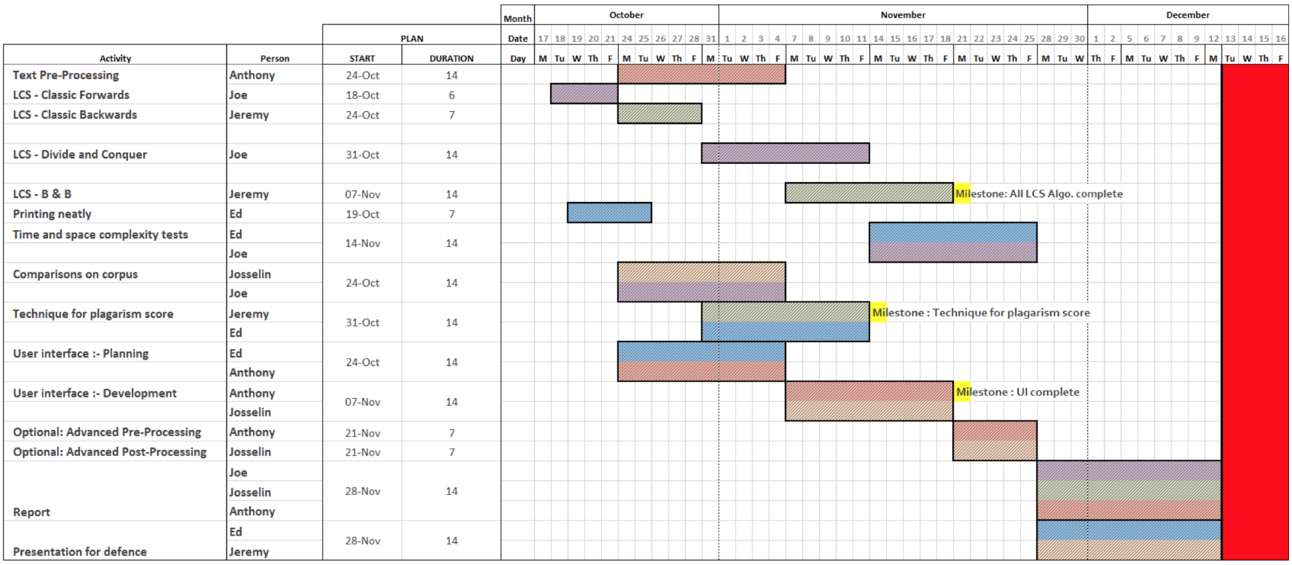


Figure 14: Planning: the plan

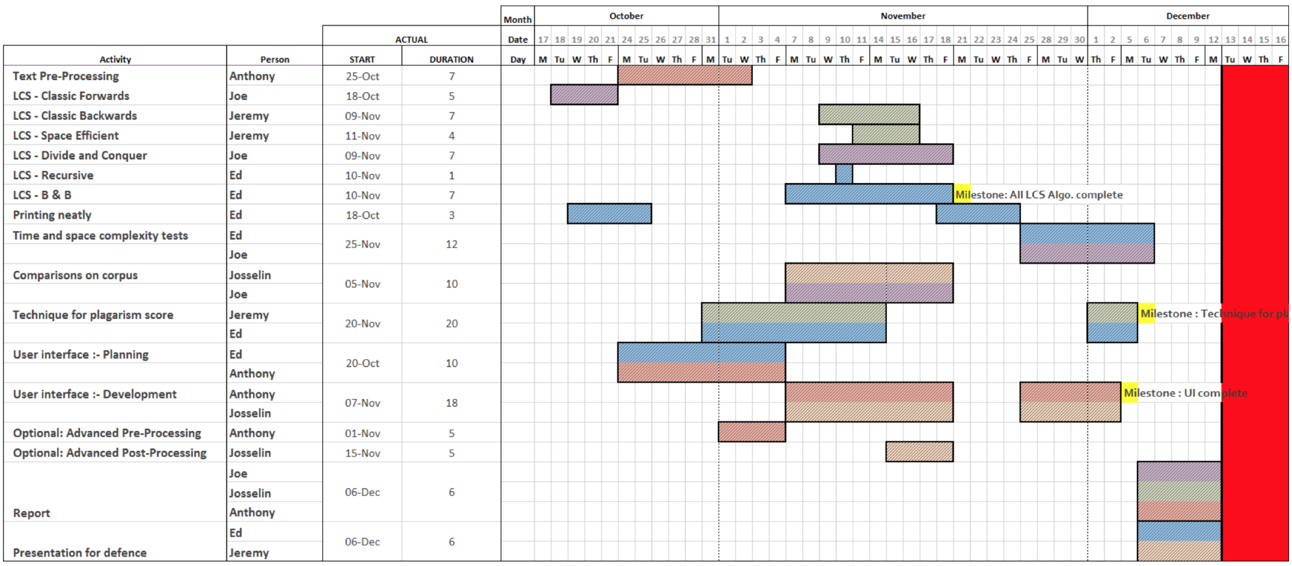


Figure 15: Planning: actual