Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 324 (2016) 107–122

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Practical Semi-External Memory Method for Approximate Pattern Matching [1]

Daniel Saad Nogueira Nunes [2]

Departamento de Ciência da Computação
Universidade de Brasília
Instituto Federal de Educação Ciência e Tecnologia de Goiás

Mauricio Ayala-Rincón[3]

Departamentos de Ciência da Computação e Matemática
Universidade de Brasília

Abstract

The approximate pattern matching problem ($\mathcal{APM}$) consists in locating all occurrences of a given pattern P in a text T allowing a specific amount of errors. Due to the character of real applications and the fact that solutions do not have error-free nature in Computer Science, solving $\mathcal{APM}$ is crucial for developing meaningful applications. Recently, Ilie, Navarro and Tinta presented a fast algorithm to solve $\mathcal{APM}$ based on the well-known Landau-Vishkin algorithm. However, the amount of available memory limits the usage of their algorithm, since it requires all the answer array be in memory. In this article, a practical semi-external memory method to solve $\mathcal{APM}$ is presented. The method is based on the direct-comparison variation from Ilie *et al.*'s algorithm. Performance tests with real data of length up to 1.2 GB showed that the presented method is about 5 times more space-efficient than Ilie *et al.*'s algorithm and yet, has a competitive trade-off regarding time.

Keywords: Semi-External,Approximate Patten Matching, String-Matching, Space-Efficient, Landau-Vishkin

1 Introduction

Exact pattern matching is an important problem which gives birth to a variety of applications in Computer Science, such as: Document Retrieval, Text Edition, Compiler tools, and many others.

When a fixed number of errors is allowed, interesting applications, with different complexities, arise (*cf.* [17,21,5]), from which the following can be mentioned:

[1] work funded by grant CNPq Universal 476952/2013-1.

[2] Email: daniel.saad@ifg.edu.br

[3] Email: ayala@unb.br. Author partially supported by a CNPq highg productivity grant.

http://dx.doi.org/10.1016/j.entcs.2016.09.010

- Fragment Mapping and assembling in Computational Biology: A high throughput sequencer gives as output billions of short fragments. In order to do accurate genomic/transcriptomic analysis, it is often necessary to map or to assemble all the fragments in a reference genome. Since errors are inherent within the sequencer technology, an exact approach is limited, once it does not deal with nucleotide mismatches, deletions or insertions.
- Sequence Alignment with respect to a score: given a set of sequences, one must identify the similarities (or the lack of them) among them in order to achieve relevant results. This can be a crucial step towards the construction of phylogenetic trees, which estimate the evolutionary distance between organisms.
- Signal processing: noise in channels is inherent to the nature of signal processing. Hence, more robust methods which can deal with errors are indispensable.
- Document Retrieval: when allowing errors, one can execute more complex queries for the retrieval of documents, which is difficult when depending only on exact pattern matching.

Several algorithms have been proposed in the literature to solve $\mathcal{APM}$ (*cf.* [22,20,15,2]). Among them, the Landau-Vishkin algorithm relies on the extension of the diagonals in a dynamic programming table to find approximate occurrences of a pattern in a text increasing the number of admissible errors in each extension [13]. However, this algorithm has a high consumption of memory, since it is based on complex data structures such as Suffix Trees or Suffix Arrays to execute the diagonal extension in constant time [3]. A variation of this algorithm proposed by Ilie *et. al* [7] has been reported to be faster and more space-economical than the classical version, since it does not rely on complex data structures, for it is based on a brute-force, but reliable way to extend the diagonals.

Despite of being efficient, the amount of available memory limits the usage of their algorithm, since it requires all the answer array be in memory. For instance, in an ordinary computer with 4 GB of RAM, it could only manipulate texts of length up to 750 MB. Therefore, a more space-efficient strategy would be necessary.

This work presents a practical semi-external memory method to solve $\mathcal{APM}$ by using the direct-comparison variation from [7]. Performance tests with real data of length up to 1.2 GB were done. It was shown that this approach is $\approx 5\times$ more space-efficient than the pure direct-comparison variation and yet, has a competitive trade-off regarding time. Hence, by using both memory and disk in a clever way, it is feasible to manipulate larger files which were impossible to be treated before.

This article extends previous work presented at WEIT 2015 (see the eight page full paper in [18]) by adding Manzini's *corpus* for experiments regarding time and memory consumption. It also includes a more detailed explanation of the Landau-Vishkin algorithm, related work techniques and algorithms.

The rest of the article is organized as follows. Section 2 introduces basic concepts and notation, Section 3 presents related works, Section 4 describes the proposed method, Section 5 presents experimental results and Section 6 concludes the article and presents possible future works.

2 Background

Let Σ^* denote the set of all strings over the finite alphabet $\Sigma = \{a_0, a_1, \ldots, a_{\sigma-1}\}$, such that $|\Sigma| = \sigma$. The empty string is denoted by ϵ, which has length $|\epsilon| = 0$.

The i^{th} symbol of a given string $X \in \Sigma^*$ is denoted by $X[i]$. Substrings of X are denoted by $X[i,j] = X[i]X[i+1]\ldots X[j]$, $0 \leq i \leq j < |X|-1$, otherwise $X[i,j] = \epsilon$. Suffixes $X[i,|X|-1]$ are denoted by X_i.

Two particular strings, called the text and the pattern, are denoted respectively by T and P, with $|T| = n$ and $|P| = m$.

Definition 2.1 [Edit Distance] For given feasible operations on strings, the edit distance between any two strings X and Y, denote as $\delta(X,Y)$, is the number of required operations which turns X into Y.

A common distance function used is the Levenhestein distance, whose operations are based on insertion, deletion and substitution of symbols, each with cost 1. From now on, this will be the standard distance.

Definition 2.2 [$\mathcal{APM}$] The Approximate Pattern Matching Problem can be formulated as, given T, P and a number of errors k, return all positions:

$$Occ = \{j | 0 \leq i \leq j < n \wedge \delta(P, T[i,j]) \leq k\} \tag{1}$$

Occ stands for the positions where P ends in T with at most k errors.

Figure 1 illustrates the occurrences of $P = ACA$ in $T = ACTAGACATAGCAA$ allowing at most one error (insertion, deletion or mismatch).

$$\begin{array}{ll}
AC{-}TAGACATAGCAA & ACTAGACATAGCAA \\
ACA & ACA \\
\\
ACTAGACATAGCAA & ACTAGACATAGCAA \\
AC{-}A & \quad ACA \\
\\
ACTAGAC{-}ATAGCAA & ACTAGACATAGCAA \\
\quad ACA & \quad ACA \\
\\
ACTAGACATAGCAA & ACTAGACATAGCAA \\
\quad AC{-}A & \quad ACA \\
\\
ACTAGACATAGCAA & ACTAGACATAGCA{-}A \\
\quad A{-}CA & \quad ACA
\end{array}$$

Figure 1. Ocurrences of $P = ACA$ in $T = ACTAGACATAGCAA$ allowing at most one error.

Definition 2.3 [LCE] The longest common string (LCE) of any given two strings X and Y corresponds to the length of the maximal prefix shared by X and Y:

$$\text{LCE}(X,Y) = \max\{k | X[0,k-1] = Y[0,k-1]\} \tag{2}$$

The Suffix Tree [23] is a fundamental text indexing data structure, as shown by [5]. However, its space consumption turns their usage unfeasible for large strings. The suffix array [14] is one of the most important space-efficient alternative which has been used to solve efficiently (in time and space) many string processing problems (*cf.* [19]).

Definition 2.4 [SA] A Suffix Array, SA for short, of a text T is an array A of integers containing the position of the Suffixes in lexicographical order induced by the order of the alphabet symbols. Hence:

$$T_{A[0]} < T_{A[1]} < \ldots < T_{A[n-1]}$$

Definition 2.5 [ISA] The inverse Suffix Array of a text T is an array A^{-1} of integers containing in the i^{th} entry, the lexicographical position of the i^{th} suffix among the others.

While in the Suffix Array $A[i]$ is concerned about the i^{th} suffix in lexicographical order, $A^{-1}[i]$ corresponds to the lexicographical position of the suffix T_i among the other suffixes of T. Therefore $A[A^{-1}[i]] = i$.

Definition 2.6 [LCP] The Longest-Common-Prefix is defined as the length of the maximal prefix shared by two consecutive entries in the SA A. Formally:

$$\mathrm{LCP}(i) = \begin{cases} 0, & i = 0 \\ \mathrm{LCE}(T_{A[i-1]}, T_{A[i]}), & i > 0 \end{cases} \tag{3}$$

Table 1 shows a SA A augmented with A^{-1} and the LCP for the text $T' = ACTAGACATAGCAA\#ACA\$$.

Definition 2.7 [RMQ] The Range-Minimum-Queries over an array V, denoted as RMQ_V, is defined as:

$$\mathrm{RMQ}_V(i,j) = \min\{\arg\min\{V[k] \mid i \leq k \leq j\}\} \tag{4}$$

Thus, $\mathrm{RMQ}_V(i,j)$ holds the leftmost position in which occurs the minimum value on $V[i,j]$.

Suffix Arrays and their inverses can be build in $\Theta(n)$ time, as shown by Kärkkäinen and Sanders [9]. The LCP information also can be computed in $\Theta(n)$ time [10]. The RMQ_{LCP} support data structure can be computed in $\Theta(n)$ as well while allowing $\Theta(1)$ queries (*cf.* [8,4]).

3 The Landau-Vishkin Algorithm

The Landau-Vishkin Algorithm, proposed originally in [13], solves the $\mathcal{APM}$. This algorithm is based on a dynamic programming technique which, at the k^{th} iteration, obtains the maximal extension of diagonals of the dynamic programming table

Table 1
Suffix array for $T' = T\#P\$ = ACTAGACATAGCAA\#ACA\$$.

i	$T'_{A[i]}$	$A[i]$	$A^{-1}[i]$	LCP$[i]$
0	$\#ACA\$$	14	7	0
1	$\$$	18	14	0
2	$A\#ACA\$$	13	17	0
3	$A\$$	17	8	1
4	$AA\#ACA\$$	12	15	1
5	$ACA\$$	15	6	1
6	$ACATAGCAA\#ACA\$$	5	13	3
7	$ACTAGACATAGCAA\#ACA\$$	0	10	2
8	$AGACATAGCAA\#ACA\$$	3	18	1
9	$AGCAA\#ACA\$$	9	9	2
10	$ATAGCAA\#ACA\$$	7	16	1
11	$CA\$$	16	12	0
12	$CAA\#ACA\$$	11	4	2
13	$CATAGCAA\#ACA$	6	2	2
14	$CTAGACATAGCAA\#ACA\$$	1	0	1
15	$GACATAGCAA\#ACA\$$	4	5	0
16	$GCAA\#ACA\$$	10	11	1
17	$TAGACATAGCAA\#ACA\$$	2	3	0
18	$TAGCAA\#ACA\$$	8	1	3

allowing at most k errors. The diagonals refers to the Table of the classical Dynamic Programming technique which computes the minimal edit distance [6].

Considering k as the number of errors, and i as the i^{th} diagonal, the dynamic programming technique is based on the recurrence relation $L(i,k)$:

$$L(i,k) = \begin{cases} \text{For } (k=0) \wedge (0 \leq i \leq n), \\ \quad L(i,0) := \text{LCE}(P_0, T_i) \\ \\ \text{For } (-(m-1) \leq i < 0) \wedge (k = -i), \\ \quad L(i,k) := \text{let } j = (L(i+1,k-1)+1) \text{ in} \\ \quad\quad \text{if } j \geq m+i \text{ then } m+i \\ \quad\quad \text{else } max(\text{LCE}(P_{-i}, T_0), j + \text{LCE}(P_{-i+j+1}, T_{j+1})) \\ \\ \text{For } (-(m-1) \leq i \leq 0) \wedge (k = -i+1) \\ \quad L(i,k) := \text{let } j = max(L(i,k-1)+1, L(i+1,k-1)+1) \text{ in} \\ \quad\quad \text{if } j \geq m+i \text{ then } m+i \\ \quad\quad \text{else } j + \text{LCE}(P_{-i+j+1}, T_{j+1}) \\ \\ \text{For } (0 < k \leq m) \wedge (-(m-1) \leq i < n), \text{where} \\ \quad \text{for } (i \geq 0), (k+i \leq n) \wedge \text{ for } (i<0), (k > -i+1) \\ \quad L(i,k) := \text{let } j = \max(L(i-1,k-1), L(i,k-1)+1, \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad L(i+1,k-1)+1) \text{ in} \\ \quad\quad \text{if } j \geq m \vee i+j \geq n \text{ then } \min(m, n-i) \\ \quad\quad \text{else } j + \text{LCE}(P_{j+1}, T_{i+j+1}) \end{cases} \tag{5}$$

Thus, by this relation, $L(i,k)$ indicates that $P[0, L(i,k)-1]$ occurs at the position $i + L(i,k) - 1$ of T with at most k errors.

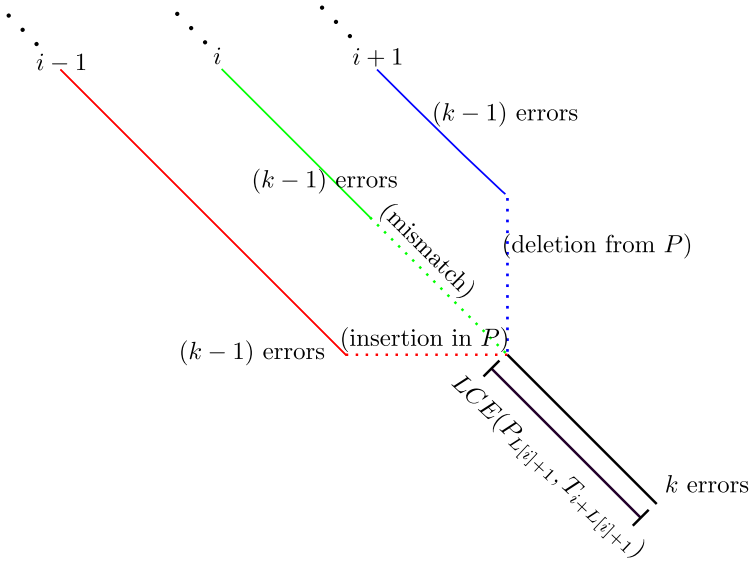


Figure 2. i^{th} diagonal extension in the Landau-Vishkin Algorithm via diagonal $i-1$.

This recurrence states that, for the diagonal i, allowing k errors, firstly, one has to look for the maximal extensions in diagonals $i-1$, i and $i+1$ allowing $k-1$ errors; afterwards, one has to introduce an error, and finally, to extend the diagonal maximally once again. Graphically this is represented in the Figure 2.

The implementation details of the dynamic programming technique are given in the Algorithm 1. One does not need to represent the entire Dynamic Programming Table, only the array L is necessary. The Table 2 shows a snapshot of table L for $T = ACTAGACATAGCAA$ and $P = ACA$ allowing at most $k = 1$ errors.

The Landau-Vishkin Algorithm in the worst-case takes $\Theta(nk \cdot t_{LCE})$, where t_{LCE} stands for the time needed to compute the LCE$(P_{L[i]+1}, T_{i+L[i]+1})$ [13].

Table 2
Snapshot of $L[i]$ for $T = ACTAGACATAGCAA$ and $P = ACA$

	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$L[0]$		1	-1	-1	0	-1	2	-1	0	-1	0	-1	-1	0	0
$L[1]$	2	2	1	2	2	2	2	2	2	1	1	2	2	1	1

3.1 Classical Approach

The classical approach computes LCE$(P_{L[i]+1}, T_{i+L[i]+1})$ with the aid of Suffix Trees [13]. First, a generalized Suffix Tree is build for the text $T' = T\#P\$$ and then, Lowest Common Ancestor (LCA) queries are done between leaves in order to obtain the LCE value. Since LCA queries can be done in $\Theta(1)$ time with a $\Theta(n)$ time preprocessing [1], it is possible to extend a diagonal maximally in $\Theta(1)$ time.

However, Suffix trees have a high consumption of memory in practice. In fact, considering the worst case behavior, Suffix Trees have a consumption between $10\times$ and $15\times$ the text input size [12]. This huge factor turns the manipulation of long text

unfeasible due the restricted amount of memory available in ordinary computers.

Algorithm 1. Generic Landau-Vishkin Algorithm

Input: P, T, k
Output: $\{j | P$ ends in $T[j]$ with at most k errors$\}$
$L[i] \leftarrow -2, -k \leq i \leq n$
for $e \leftarrow 0; e \leq k; k++$ **do**
  $prev \leftarrow -2$
  $cur \leftarrow -2 + e$
  $next \leftarrow L[-e + 1]$
  for $i \leftarrow -e; i < n; i++$ **do**
    $L[i] \leftarrow \max(prev, cur + 1, next + 1)$
    $L[i] \leftarrow \max(L[i], m - 1)$
    if $(i + L[i] + 1 < n)$ **then**
      $L[i] \leftarrow L[i] + \text{LCE}(P_{L[i]+1}, T_{i+L[i]+1})$
    $prev \leftarrow cur$
    $cur \leftarrow next$
    $next \leftarrow L[i + 2]$
for $i \leftarrow -k; i < n; i++$ **do**
  if $L[i] \geq |P| - 1$ **then**
    ReportMatch$(T_{i+|P|-1})$

3.2 Using Suffix Arrays

A variation proposed by Miranda and Ayala-Rincón [3] uses a variant of Suffix Arrays. The used data structure are essentially Suffix Arrays augmented with additional information: the inverse Suffix Array (ISA), the Longest-Common-Prefix (LCP) information and a support data structure for Range-Minimum-Queries (RMQ).

In order to compute the value $\text{LCE}(P_{L[i]+1}, T_{i+L[i]+1})$ in $\Theta(1)$ time, one needs to build a complex data structure in the preprocessing phase. Building this data structure consist of: creating the Suffix Array A for $T' = T\#P\$$, computing its inverse A^{-1}, calculating its LCP information and building a support data structure RMQ_{LCP}.

The answer of a given $\text{RMQ}_{LCP}(k, l)$ can be obtained in constant time after that data structure is built. Thus, $\text{LCE}(P_{L[i]+1}, T_{i+L[i]+1}) = \text{LCP}[\text{RMQ}_{LCP}(k + 1, l)]$, where $k = \min\{A^{-1}[L[i] + n + 2], A^{-1}[L[i] + i + 1]\}$ and $l = \max\{A^{-1}[L[i] + n + 2], A^{-1}[L[i] + i + 1]\}$. Hence, the LCE query is supported in constant time.

For example, in the Table 1, $\text{LCE}(AGCAA\#ACA\$, ACA\$)$ corresponds to $\text{LCP}[\text{RMQ}_{LCP}(6, 9)] = \text{LCP}[8] = 1$.

The resulting procedure is given by Algorithm 2.

Since RMQ_{LCP} queries take $\Theta(1)$ time, $t_{LCE} \in \Theta(1)$. Therefore, the Landau-

Algorithm 2. Extension of diagonals by using LCP and RMQ queries
Input: $P, T, i, L[i], A^{-1}, \text{LCP}, \text{RMQ}_{LCP}$
Output: $\text{LCE}(P_{L[i]+1}, T_{i+L[i]+1})$
$i_1 \leftarrow A^{-1}[|T| + L[i] + 2]$
$i_2 \leftarrow A^{-1}[i + L[i] + 1]$
if $i_1 > i_2$ **then**
  SWAP(i_1, i_2)
return $LCP[RMQ_{LCP}(i_1 + 1, i_2)]$

Vishkin algorithm takes $\Theta(kn)$ in the worst case when using this collection of data structures.

3.3 The Direct Comparison Variation

Despite assuring a $\Theta(kn)$ worst case time, the classical solution is often considered unpractical, due to the large factors involved in the construction of Suffix Arrays, LCP information and the RMQ support data structure.

Moreover, according to Ilie *et. al*, LCE values tend to be quite small on average [7]. In fact, there is a proof that, the average $\text{LCE}(S, R)$ considering all strings S and R of length n, is $\leq \frac{1}{\sigma - 1}$.

Due to this behavior, RMQ queries do not go well, since despite taking time in $\Theta(1)$, a meaningful overhead is present. A brute-force approach to compute LCE values is faster in practice, since they are usually small.

Ilie *et. al* showed that the Landau-Vishkin algorithm turns from a unpractical one to a practical one when direct-comparisons are made in order to compute $\text{LCE}(P_{L[i]+1}, T_{i+L[i]+1})$.

The Algorithm given in 3 shows the direct-comparisons version of the diagonal extension.

Algorithm 3. Extension of diagonals by using direct-comparisons
Input: $P, T, L[i]$
Output: $\text{LCE}(P_{L[i]+1}, T_{i+L[i]+1})$
$c \leftarrow 0$
while $P[L[i] + 1 + c] = T[i + L[i] + 1 + c]$ **do**
  $c++$
return c

The extension by direct-comparisons turned out to be very effective [7]. Besides, it uses the computer resources in a more appropriate way, especially with respect to cache memory. This is due to the great locality of reference, which was poor in the RMQ based queries of the classical variation.

Asymptotically, when using the direct-comparisons, $t_{LCE} = O(m)$ in the worst case. However, on average, $t_{LCE} = \frac{1}{\sigma - 1} \in \Theta(1)$. Therefore, a Landau-Vishkin approach, in the average case using direct-comparisons, would take $\Theta(kn)$ time.

Table 3
Memory, data structures and time required for Landau-Vishkin variations

	Variations		
	Classical	Direct-Comparisons	Semi-External
Data Structures	P, T, A, A^{-1}, LCP, RMQ and L	P, T and L	P and T
Memory (bytes)	$\approx 17.875n + m$	$5n + m$	$n + m + \Theta(1)$
Worst-Case	$\Theta(kn)$	$\Theta(knm)$	$\Theta(knm)$
Average-Case	$\Theta(kn)$	$\Theta(kn)$	$\Theta(kn)$

4 Proposed Semi-External Memory Method

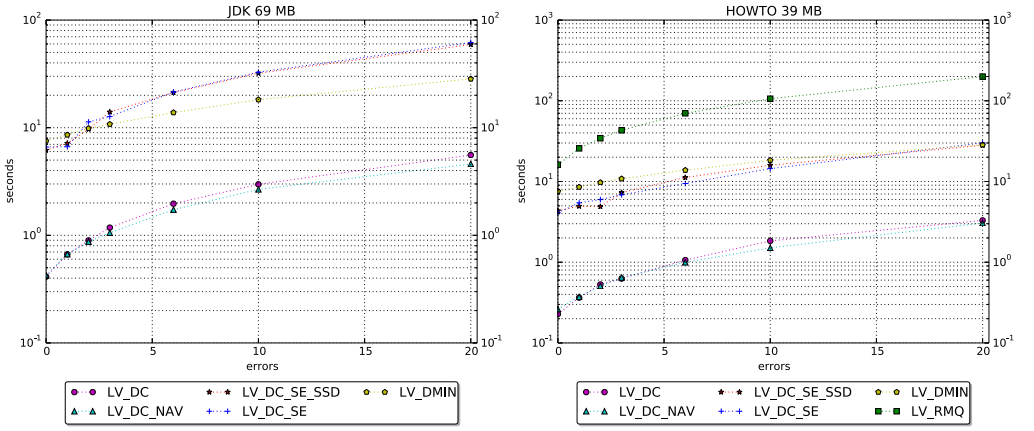
A shared problem between the classical and the direct-comparison versions of the Landau-Vishkin algorithm is the amount of memory used. The straightforward classical variation needs $4\,n$ bytes for A, $4\,n$ bytes for A^{-1}, $4\,n$ bytes for the LCP information, $4\,n$ bytes for the L array, n bytes for the text, m bytes for the patten and $\approx \frac{7}{8}n$ bytes for the RMQ_{LCP} support data structure based on [4]. The faster and more space-economical direct-comparison variation does not need any preprocessed data structure, only n bytes for the text, m bytes for pattern and $4\,n$ bytes for the L array.

Thus, an ordinary computer with 4 GB of RAM could only manipulate texts of $\approx$ 200 MB in the classical variation and $\approx$ 750 MB in the pure direct-comparison variation. Therefore, long texts cannot be treated by ordinary approaches. A more space-efficient strategy is necessary.

Semi-external algorithms are in between internal memory algorithms and external memory algorithms. An algorithm is named semi-external if the input is dependent to the amount of available memory, but it also uses external memory. In practice, a semi-external algorithm accesses the data in the disk sequentially and maintains in memory the data which needs to be accessed randomly.

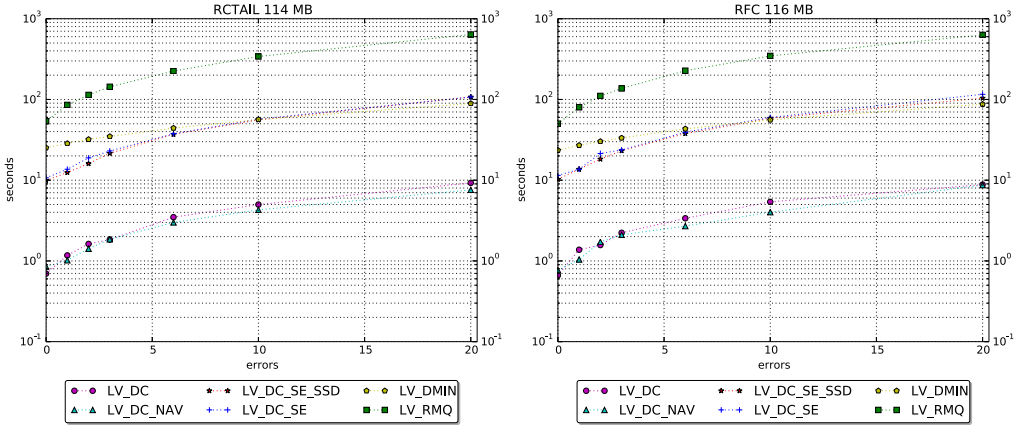
This work proposes a practical semi-external memory method based on the direct-comparison variation from [7] to solve $\mathcal{APM}$. This method explores the fact that the array L is not entirely needed in memory. Thus, one can keep a block $B = L[j, j + |B| - 1]$ in main memory to process the diagonals $j + 1 \ldots j + |B| - 2$. When the next block of diagonals need to be computed, one simply stores B into disk and loads the new chunk $B' = L[j+|B|-1, j+2|B|-2]]$ from the same disk into memory. Since these values would occupy a contiguous space in the disk, seeks are minimized, for the data tend to be in the same track, and hence, the technique does not struggle from the disk access bottleneck. Choosing $B \in \Theta(1)$, the overall space required is $n+m+\Theta(1)$, that is about $5\times$ less than the original direct-comparisons variation ($5n + m$ bytes).

The Table 3 summarizes the issues discussed until now.



(a) 69 MB JDK File (b) 39 MB HOWTO file.

Figure 3. Manzini *corpus* files.



(a) 114 MB RCTAIL file. (b) 116 MB RFC file.

Figure 4. Manzini *corpus* files.

5 Experimental Results

In order to evaluate the proposed semi-external variations, experiments were performed considering collections of long texts from Pizza&Chili [4] and Manzini [5] *corpora*. Table 4 shows a description of the used texts. Each text has its own particularities and specific alphabets.

Comparisons were done with the respective Landau-Vishkin algorithm variations:

(i) `LV_RMQ`: classical Variation with Suffix Arrays. Uses A, A^{-1} , LCP and

[4] http://pizzachili.dcc.uchile.cl/

[5] http://people.unipmn.it/~manzini/lightweight/corpus/

[5] http://dblp.uni-trier.de/db/

[6] ftp://ftp.ebi.ac.uk/pub/databases/swissprot/release_compressed/

Table 4
Texts used in the experiments.

Text	Size (MB)	Description
Manzini *Corpus*		
JDK	69.72	HTML file documenting Java 2
HOWTO	39.42	Describes DFX graphics accelerator chip support for Linux
RCTAIL	114.71	XML file containing news from Reuters
RFC	116.42	A request for comments file
Pizza&Chili *Corpus*		
DNA	403.92	DNA texts collected from Gutenberg project
DBLP	293.13	XML files containing bibliographical information from DBLP [6]
English	1073.74	Natural Language texts collected from Gutenberg Project
Proteins	1184.05	Sequence of proteins obtained from Swissprot [7]

RMQ_{LCP}. To build A, `libdivsufsort` was used [16]. The LCP was constructed by the authors using Kasai *et. al* method [10]. For the support RMQ_{LCP} data structure, code from [4] was used.

(ii) `LV_DMIN`: direct-comparison variation aided by LCP information. It uses direct-comparisons only if the distance in the Suffix Array from the suffixes $P_{L[i]+1}$ and $T_{i+L[i]+1}$ is large enough (suffixes which are close in the Suffix Array tend to share more symbols). Otherwise, a linear scan picking the minimum LCP value between these two suffixes is used. In this variation RMQ_{LCP} is not used.

(iii) `LV_DC`: our direct-comparison variation. Only needs P, T and L.

(iv) `LV_DC_NAV`: Ilie's *et al.* direct-comparison variation from [7] [8].

(v) `LV_DC_SE`: our semi-external variation using a SATA 500 GB, 7200 RPM hard disk drive.

(vi) `LV_DC_SE_SSD`: our semi-external variation using a 64 GB, SATA SSD disk.

The code is based on `C++11` standard and all tests ran in an `Ubuntu 12.04 64-bit`, `core i5-750`, 4GB 1333 Mhz RAM machine. All codes were compiled using `g++ 4.8` with the `-O2` flag for optimization purposes. The block size used for the Semi-External variation was $B = 1$ MB. The code is available on https://github.com/danielsaad/Semi-External-Landau-Vishkin.

Each experiment was executed 3 times and the lowest wall clock time was chosen. A strong statistical analysis regarding more executions were not done once the Landau-Vishkin variations differ one from another orders of magnitudes considering time. At every experiment, a pattern with length 50 was chosen from the text randomly, as was done in [7], since the overall time showed to be largely independent of P and m. Errors from the set $\{0, 1, 2, 3, 6, 10, 20\}$ were considered in the experiments. Prefix files from the *corpora* were also considered. For measuring the used memory, the `malloc_count` [9] tool was used.

Figures 3 and 4 show the experiments considering Manzini *corpus*. On every

[8] The direct-comparisons Landau-Vishkin code was made available by Gonzalo Navarro.

[9] https://github.com/bingmann/malloc_count

experiment, one can see that both the direct-comparison variations (`LV_DC` and `LV_DC_NAV`) are faster than any other variation. The slower one is the classical approach using RMQ queries, which behaves poorly for every file, confirming the high constants involved in the algorithm. The semi-external variations (`LV_DC_SE` and `LV_DC_SE_SSD`) have a good trade-off between space and time, for they are a order of magnitude slower if compared with the direct-comparison variations, but they require $\approx 5\times$ less memory. Finally, the hybrid `LV_DMIN` variation shows to be as competitive as the semi-external variations, since the data structures required taking a considerable amount of time to be built, however, their space requirements are much higher than the semi-external variations.

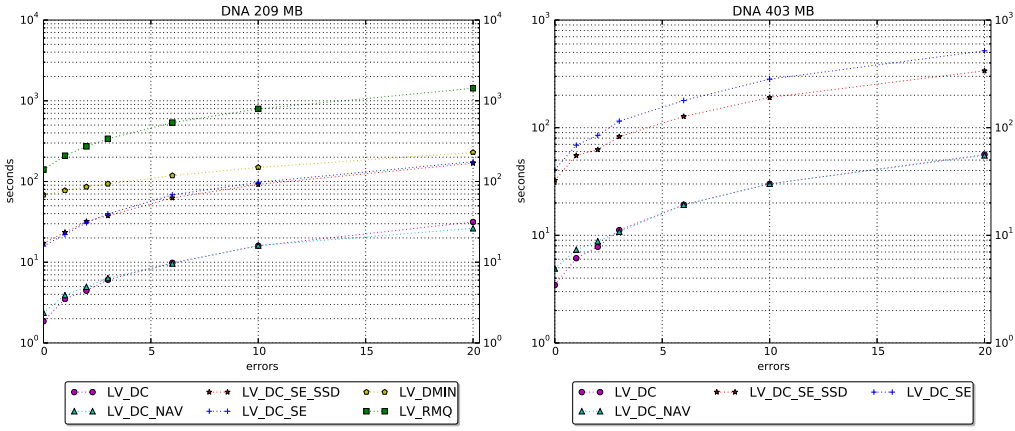
We shall consider now experiments under Pizza&Chili *Corpus*.

For DNA files, Figure 5 shows the experiments performed. Considering the 209 MB prefix file from Figure 5a, it is clear that the classical variation is unpractical, taking two orders of magnitude more than the direct-comparison variations. Once again, the semi-external variations show to have a good compromise between time and space, being only one order of magnitude slower than the direct-comparison variation. Again, `LV_DMIN` shows a similar behavior with respect to the semi-external variations, but using much more memory. Figure 5b shows a situation where the classical and the `LV_DMIN` variations do not fit in memory. Once again, the semi-externals variations showed to be very competitive with respect to time/space trade-off. Also, it can be noticed that the `LV_DC_SE_SSD` variation has a significant difference in relation as the `LV_DC_SE` variation, which shows the potential of the SSD technology for larger files.

Considering now the XML alphabet, Figures 6a and 6b show the experiments performed for the 209 MB prefix and the 293 MB XML-DBLP files. As discussed before, the semi-external variations have an acceptable trade-off between time and space. Besides, Figure 6b shows that both the classical and the `LV_DMIN` variations are unfeasible for larger files, since they do not fit in main memory.

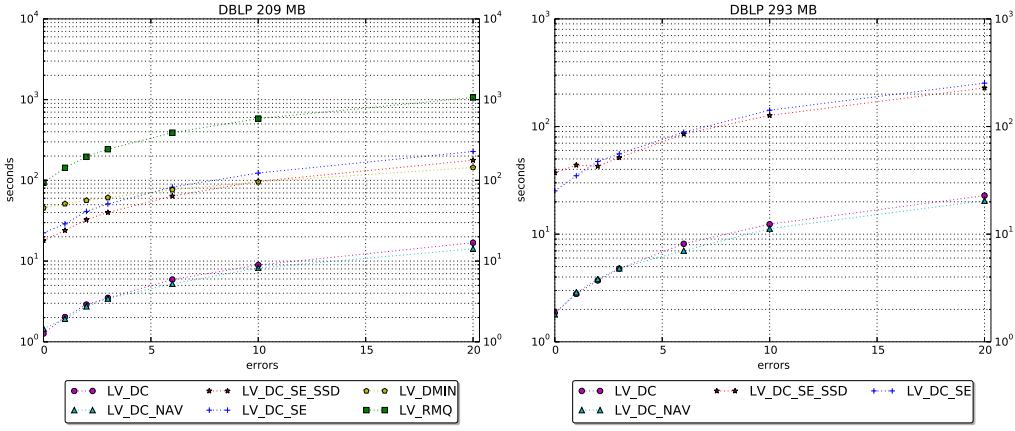
Figures 7 and 8 show the results with respect to the English and Proteins, respectively. The same behavior from the previous experiments is observed. The interesting cases are shown in Figures 7b and 8b. In these two experiments, even the direct-comparison variations do not fit in memory, thus the semi-external variations show their truly utility. Once again, the `LV_DC_SE_SSD` shows to be more efficient than the `LV_DC_SE` variation for larger files.

Regarding the peak memory consumption, Table 5 shows that `LV_DC_SE` has a much lower space consumption than the `LV_DC` version (about $5\times$). `LV_RMQ` and `LV_DMIN` have a higher space consumption which explains why such variations cannot handle bigger texts. One can see that in **Pizza&Chili *Corpus*** the results in the experiments are the same, once the used data structures are independent of the underlying alphabet. The practical results reflects the theoretical information shown by table 3.



(a) 209 MB prefix DNA file. (b) 403 MB DNA file.

Figure 5. DNA files



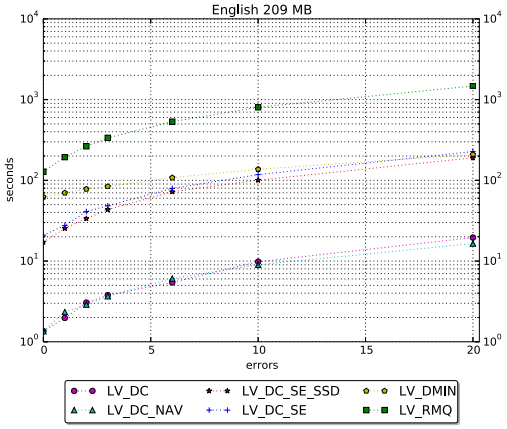
(a) 209 MB prefix DBLP-XML file. (b) 293 MB DBLP-XML file.

Figure 6. DBLP-XML files

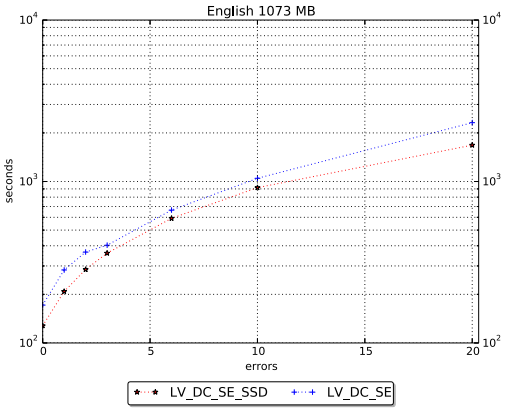
6 Conclusions and Future Work

$\mathcal{APM}$ is a very important and recurrent problem in Computer Science with applications in many areas. The Landau-Vishkin method solves this problem by using the classical dynamic programming approach by extending maximally each diagonal of the table while the number of errors is increased by one. It has been shown that the classical variation performs poorly in practice. Due to an observation from [7], it was shown that the expected LCE between any two strings with length n is $1/(\sigma-1)$. Hence, a brute-force approach to compute LCE values results faster than elaborated and complex solutions, such as the ones given in [13,3].

This work proposed a semi-external variation of the Landau-Vishkin algorithm based on direct-comparisons. Despite of being only an order of magnitude slower in the majority of the experiments than the pure direct-comparison variations, the semi-external variation is a factor of $\approx 5\times$ more space-efficient, as stated by Tables

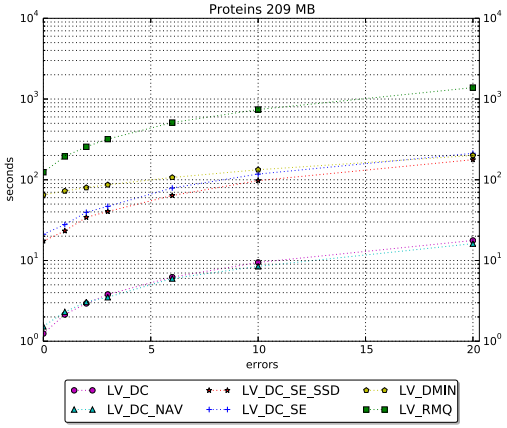


(a) 209 MB prefix English file.

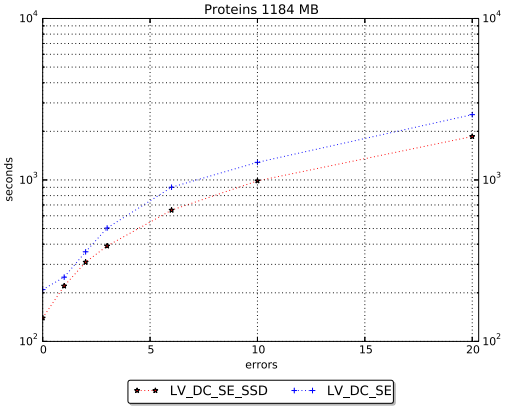


(b) 1073 MB English file.

Figure 7. English files



(a) 209 MB prefix Proteins file.



(b) 1184 MB Proteins file.

Figure 8. Proteins files

3 and 5. Besides, the semi-external variation is able to manipulate larger texts which cannot be possibly treated by the pure direct-comparison variation, since it does not fit in main memory. Consequently, the proposed semi-external memory approach is of practical value.

Further possible improvements include the following. In order to make the semi-external memory method more space and time efficient, one could represent the alphabet in a succinct way. For example, considering a DNA alphabet, one could spend only 2 bits per symbol, instead of a byte. Bit packing can be employed as well, since when packing several symbols under a single integer, multiple comparisons can be performed at once. Compressing texts while allowing random access to them (*cf.* [11]) can be a good option in order to manipulate huge texts as well when the raw representation does not fit in main memory.

Table 5
Peak Memory Consumption.

		Peak memory for each variation (MB)			
File	Size (MB)	LV_DC_SE	LV_DC	LV_RMQ	LV_DMIN
Manzini *Corpus*					
JDK	69.72	78.163	348.67	1241.29	1185.70
HOWTO	39.42	47.85	197.13	701.30	670.48
RCTAIL	114.71	123.14	573.58	2041.83	1950.40
RFC	116.42	124.85	582.13	2072.28	1979.48
Pizza&Chili *Corpus*					
DNA	209.71	218.14	1048.60	3735.88	3565.46
DBLP	209.71	218.14	1048.60	3735.88	3565.46
ENGLISH	209.71	218.14	1048.60	3735.88	3565.46
PROTEINS	209.71	218.14	1048.60	3735.88	3565.46

References

[1] Bender, M. A. and M. Farach-Colton, *The LCA Problem Revisited*, in: G. H. Gonnet, D. Panario and A. Viola, editors, *LATIN*, Lecture Notes in Computer Science **1776** (2000), pp. 88–94.

[2] Cole, R. and R. Hariharan, *Approximate String Matching: A Simpler Faster Algorithm*, SIAM Journal on Computing **31** (2002), pp. 1761–1782.

[3] de Castro Miranda, R. and M. Ayala-Rincón, *A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays*, in: *Brazilian Symposium on Bioinformatics*, 2005, pp. 210–213.

[4] Fischer, J. and V. Heun, *A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array*, in: B. Chen, M. Paterson and G. Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, Lecture Notes in Computer Science **4614** (2007), pp. 459–470.
URL http://dx.doi.org/10.1007/978-3-540-74450-4_41

[5] Gusfield, D., "Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology," Cambridge University Press, 1997.

[6] Hirschberg, D. S., *A linear space algorithm for computing maximal common subsequences*, Commun. ACM **18** (1975), pp. 341–343.
URL http://doi.acm.org/10.1145/360825.360861

[7] Ilie, L., G. Navarro and L. Tinta, *The Longest Common Extension Problem Revisited and Applications to Approximate String Searching*, Journal of Discrete Algorithms **8** (2010), pp. 418–428.

[8] Johannes Fischer and Volker Heun, *Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE*, in: M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching*, Lecture Notes in Computer Science **4009** (2006), pp. 36–48.

[9] Kärkkäinen, J. and P. Sanders, *Simple Linear Work Suffix Array Construction*, in: J. C. M. Baeten, J. K. Lenstra, J. Parrow and G. J. Woeginger, editors, *International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science **2719** (2003), pp. 943–955.

[10] Kasai, T., G. Lee, H. Arimura, S. Arikawa and K. Park, *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in: A. Amir and G. M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, Lecture Notes in Computer Science **2089** (2001), pp. 181–192.

[11] Kreft, S. and G. Navarro, *Lz77-like compression with fast random access*, in: J. A. Storer and M. W. Marcellin, editors, *2010 Data Compression Conference (DCC 2010), 24-26 March 2010, Snowbird, UT, USA* (2010), pp. 239–248.
URL http://dx.doi.org/10.1109/DCC.2010.29

[12] Kurtz, S., *Reducing the Space Requirement of Suffix Trees*, Software: Practice and Experience **29** (1999), pp. 1149–1171.

[13] Landau, G. M. and U. Vishkin, *Fast Parallel and Serial Approximate String Matching*, Journal of Algorithms **10** (1989), pp. 157–169.

[14] Manber, U. and E. W. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal on Computing **22** (1993), pp. 935–948.

[15] Masek, W. J. and M. Paterson, *A Faster Algorithm Computing String Edit Distances*, Journal of Computer and System Sciences **20** (1980), pp. 18–31.

[16] Mori, Y., *libdivsufsort - A Lightweight Suffix Sorting Library* (2007).
URL https://code.google.com/p/libdivsufsort/

[17] Navarro, G., *A guided tour to approximate string matching*, ACM Computing Surveys **33** (2001), pp. 31–88.

[18] Nunes, D. S. N. and M. Ayala-Rincón, *A Practical Semi-External Memory Method for Approximate Pattern Matching*, in: *3rd Workshop-School on Theoretical Computer Science (WEIT 2015), Porto Alegre, RS, Brazil, October 14-16*, 2015, pp. 11–18.

[19] Ohlebusch, E., "Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction," Oldenbusch Verlag, 2013.

[20] Sellers, P. H., *The Theory and Computation of Evolutionary Distances: Pattern Recognition*, J. Algorithms **1** (1980), pp. 359–373.

[21] Setubal, J. C. and J. Meidanis, "Introduction to Computational Molecular Biology," PWS Publishing Company, 1997, I-XIII, 1-296 pp.

[22] Ukkonen, E., *Algorithms for Approximate String Matching*, Information and Control **64** (1985), pp. 100–118.

[23] Weiner, P., *Linear pattern matching algorithms*, in: *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973* (1973), pp. 1–11.
URL http://dx.doi.org/10.1109/SWAT.1973.13