UNIVERSIDADE FEDERAL DA FRONTEIRA SUL

FEDERAL UNIVERSITY OF FRONTEIRA SUL
CAMPUS OF CHAPECÓ
COURSE OF COMPUTER SCIENCE

GABRIEL BATISTA GALLI

ON THE MARRIAGE OF STRINGS

CHAPECÓ
2018

GABRIEL BATISTA GALLI

ON THE MARRIAGE OF STRINGS

Final undergraduate work submitted as requirement to obtain a Bachelor's degree in Computer Science from the Federal University of Fronteira Sul.
Advisor: Emílio Wuerges
Co-advisor: José Bins

CHAPECÓ
2018

Galli, Gabriel Batista

On the marriage of strings / Gabriel Batista Galli. – 2018.
49 pp.: il.

Advisor: Emílio Wuerges.
Co-advisor: José Bins.
Final undergraduate work – Federal University of Fronteira Sul, course of Computer Science, Chapecó, SC, 2018.

1. ICCAD. 2. Bipartite matching. 3. String distance. 4. Suffix array. 5. q-grams. I. Wuerges, Emílio, advisor. II. Bins, José, co-advisor. III. Federal University of Fronteira Sul. IV. Title.

GABRIEL BATISTA GALLI

ON THE MARRIAGE OF STRINGS

Final undergraduate work submitted as requirement to obtain a Bachelor's degree in Computer Science from the Federal University of Fronteira Sul.

Advisor: Emílio Wuerges
Co-advisor: José Bins

This final undergraduate work was defended and approved by the examination committee on: December 6, 2018.

EXAMINATION COMMITTEE

[illegible]
Emílio Wuerges – UFFS

[illegible]
José Bins – UFFS

[illegible]
Denio Duarte – UFFS

[illegible] por vídeo conferência
Marcelo Cezar Pinto – UNILA

ABSTRACT

Given two sets L and R of strings such that R is the result of applying unknown transformations to L, match every string in L to its corresponding transformed string in R. This problem was proposed at the 2018 International Conference on Computer-Aided Design (ICCAD) CAD Contest, to which this work studies a deterministic solution using graph theory and approximate string matching. From graph theory, bipartite matching and stable marriage are reviewed; from approximate string matching, suffix array, edit distance, q-gram distance and q-gram index are considered. Three other algorithms are proposed based on these concepts, two of which only serve the purpose of being baselines to the others. The proposed solution itself is divided in three steps: the construction of filters (or indexes) on L and R with suffix array or q-gram index; the calculation of the adjacency (preference) lists for the graph $G = (\{L \cup R\}, E)$ with edit or q-gram distance as edge weights; and, finally, the computation of the bipartite matching. A framework for benchmarking the run time and accuracy of this approach was built. It also allows an easy switching between the available algorithms. The most significant outcome of the benchmarks is that the filtering step is the bottleneck of the proposed methodology. It affects accuracy and prevents a proper reasoning about the impact of some of the compared algorithms. Nevertheless, the works of the winning teams of the contest managed to achieve 100% of accuracy. When they are published, it will be possible to study their solutions and perhaps better understand the compromises of our choices in order to propose improvements to our approach.

Keywords: ICCAD. Bipartite matching. String distance. Suffix array. q-grams.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

It is common that compilers must deal with name collisions, for example because of the existence of modules or function overloading (1). To solve this problem, among other things, they usually implement name mangling. Basically, this technique consists of a series of transformations to the names present in the code to make them unique. For instance, one could prepend the module name to a function name so that it does not conflict with a namesake in another module; or add return and arguments type information to the function name so that the language allows overloading. This last example can be seen in Fig. 1, in which the mangled function name has the character 'i' of the return and parameter type "int" preprended and appended to it, along with underscores as simple delimiters.

Figure 1 – Example of code mangling

(a) Original code

```
int f(int x) {
    return 2 * x;
}
```

(b) Mangled code

```
int _i_f_i(int x) {
    return 2 * x;
}
```

Source – the authors

In the case of a compiler, the task of demangling a name is easily accomplished and there are several tools to do this (for instance, `undname.exe` in Robertson et al. (1)). That is because the mangling rules are commonly well defined and documented and do not change frequently (Rossum; Warsaw; Coghlan (2), for example). So if a name is given, it will be mangled by just following the rules. Likewise, given a mangled name, it will be demangled by just following the rules in reverse. However, if the rules are unclear or incomplete, the developer of the demangler will have a hard time building a tool that correctly covers all the corners; and if they keep changing, the tool will demand constant updating. Figure 2 illustrates an undocumented change in mangling rules.

Figure 2 – Example of different mangling rules

(a) Original mangling

```
int _i_f_i(int x) {
    return 2 * x;
}
```

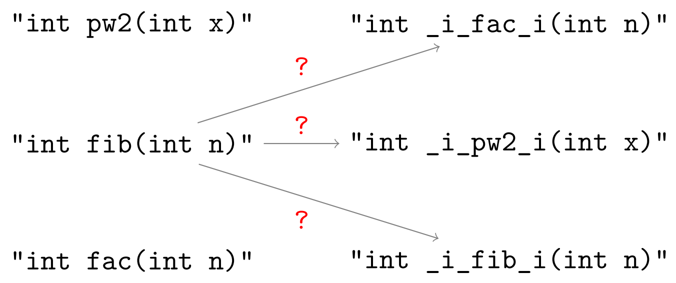
(b) New mangling

```
int _g_@4z_f@4z(int x) {
    return 2 * x;
}
```

Source – the authors

Cadence Design Systems Inc. introduced a similar problem at the 2018 International Conference on Computer-Aided Design (ICCAD) CAD Contest (3). They report that optimization tools change the names of design components, such as modules, ports and nets, to comply to implementation rules (such as 4, p. 179). The optimizations are continuously applied from one stage to another, for example when moving from logic design to circuit design, and the rules they follow may change, either because it is a different tool or a different stage. Wu; Huang; Hsu (3) also claim that mapping these changes of names is important for formal equivalence checking and engineering change orders. This would be an easy task for humans, but there are too many names for a human to map, as recent CPU designs have billions of transistors (5). Hence their proposal for the contest, in which we receive two sets L and R of strings such that R is the result of applying an unknown transformation to L. We assume without loss of generality that it is a single transformation, but it may be the composition of multiple ones. Additionally, $|L| = |R| = n$. The task is to match every string l in L to its resulting transformation r in R. Having no match between any two strings in L and any two strings in R, these sets can be seen as the two partitions of a complete bipartite graph $G = (\{L \cup R\}, E)$. An example of an instance of this task can be seen in Fig. 3 (only a few edges were drawn for the sake of clarity). Solving it means choosing the edges in E that compose the correct matching of the input strings. In order to decide what edges to choose, a string distance function is used to assign weights to them and the ones with lowest values are considered the correct choice.

Figure 3 – Example of an instance of the problem



Source – the authors

However, additional challenges arise when we analyze the test cases provided by the problem authors. They may be downloaded at `http://iccad-contest.org/2018/Problem_A/cases_all.tgz`. Table 1 enumerates them along with their size and minimum, maximum, arithmetic mean (or average; μ), standard deviation (σ), and quartiles[1] of the lengths of their strings (values are for both L and R). Notice that they are numbered starting from 1 in this work. The 24 *training* test cases are "labeled" (they are JSON files containing the correct matching) and therefore allow us to compute the accuracy of our solution. We say "training" because Wu; Huang; Hsu hinted at solutions that use supervised machine learning (ML). We are not using ML and we have no training step.

[1] Q_2 is the median of the data

Still, we approached it in a more general way by not taking advantage of any intrinsic feature of the test cases and their strings.

Table 1 – Sizes of the test cases provided by Wu; Huang; Hsu (3)

		String lengths (m)						
Test case	Size (n)	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	1.1×10^5	3	150	64.99	21.22	52	65	79
2nd	1.1×10^4	2	287	84.73	60.50	36	64	116
3rd	1.1×10^4	3	287	87.16	60.44	36	66	116
4th	6.3×10^4	3	213	92.66	35.60	66	99	114
5th	3.4×10^4	2	107	22.64	9.71	17	21	26
6th	5.6×10^4	3	183	60.73	26.42	38	58	80
7th	2.4×10^4	4	115	39.24	22.93	17	36	57
8th	2.2×10^4	4	115	39.12	23.39	17	36	58
9th	5.6×10^4	3	101	47.84	16.17	37	47	57
10th	5.8×10^4	2	162	37.56	24.03	15	29	55
11th	7.1×10^4	1	93	38.51	15.33	24	36	49
12th	7.3×10^4	3	218	55.40	35.47	29	48	64
13th	8.0×10^4	1	1608	148.1	355.3	29	37	112
14th	8.5×10^4	2	266	65.04	30.42	40	65	78
15th	7.6×10^4	21	103	57.83	18.79	39	60	74
16th	8.0×10^4	3	147	36.38	16.00	30	35	39
17th	5.5×10^4	3	94	35.27	20.53	21	24	47
18th	2.4×10^5	1	263	96.76	52.57	54	89	145
19th	1.9×10^5	26	263	117.1	41.05	81	110	151
20th	5.4×10^4	2	112	42.27	16.13	32	40	53
21st	5.1×10^4	2	94	41.23	18.42	28	38	51
22nd	6.3×10^4	2	207	32.23	23.14	19	23	33
23rd	6.5×10^4	2	110	35.02	19.43	18	31	51
24th	5.6×10^4	2	92	15.35	9.26	11	13	13

Source – the authors

The first challenge is the size (n) of the datasets, which ranges from roughly 10^4 to 10^5, meaning that the biggest graph will have 2×10^5 vertices and each one of them will have a 10^5-sized adjacency list. Therefore, because of the $O(n^2)$ space complexity of storing all complete adjacency lists, 2×10^{10} integers are needed in memory to represent them. Considering the regular 32-bit integer, this totals approximately 74 GiB of RAM. Another challenge is the fact that the vertices represent strings and we need to compare them to build the weighted adjacency lists. For example, a distance function like the edit distance that is $O(m^2)$, m being the length of the longest of the two strings involved in the comparison, will lead to a time complexity of $O(n^2m^2)$ to compare all elements in L with all elements in R. Indeed, our initial implementation in Python 3 did not terminate within one and a half hours of running the first test case on an Intel® Core™ i5-6300HQ (6) with 16 GB of RAM. This means that it did not even get to compute 20% of all preference lists or else it would have run out of memory.

Hence there is a need to avoid the construction of complete adjacency lists. In other words, avoid the inherent cartesian product comparison of the complete bipartite graph.

This means that we need to *filter* which of its n^2 edges should be considered to compute the string distance and then be matched against. The concepts and algorithms related to string distance, filtering and bipartite matching that we used in our attempt to solve this problem are reviewed in Chapter 2. In Chapter 3 are proposed additional algorithms that are based on the review or will be used as a baseline to our results. Chapter 4 is where we describe our methodology and more precisely define the three steps of our solution. Then in Chapter 5 we present and analyze our results and in Chapter 6 we give our final remarks.

2 LITERATURE REVIEW

In the following sections of this Chapter are presented the concepts and algorithms relevant to the three steps of our approach. First, two string distance functions are described and illustrated, the edit and q-gram distances, which may be used to compute the weight of each edge in the problem graph; second, two indexing algorithms from approximate string matching to filter the edges from the problem graph, the suffix array and the q-gram index; last, an overview of the theory behind bipartite matching and what is called stable marriage, wherein the matching algorithms derive from.

2.1 STRING DISTANCE

2.1.1 Edit distance

We consider the edit distance as seen in Wagner; Fischer (7). It is defined as follows: the distance between strings s_1 and s_2 is the number of operations needed to transform s_1 into s_2. The allowed operations are insertion, deletion and substitution and are defined for a single character only. All operations are performed on s_1. A function $\textsc{Cost}(a, b)$ (called $\gamma(a \to b)$ by the authors) is also needed to assign a cost to these operations, wherein a and b are characters or ϵ (the empty string). If both are characters, the operation is a substitution (from a to b); if only a is a character, the operation is a deletion (of a); if only b is a character, the operation is an insertion (of b).

Algorithm 1 – Edit distance algorithm

```
function WF($s_1, s_2$)
    $D[i, 0] \leftarrow i$ for $0 \leq i \leq |s_1|$
    $D[0, j] \leftarrow j$ for $0 \leq j \leq |s_2|$
    for $1 \leq i \leq |s_1|$ do
        for $1 \leq j \leq |s_2|$ do
            $substitution \leftarrow D[i-1, j-1] + \textsc{Cost}(s_1[i-1], s_2[j-1])$
            $deletion \leftarrow D[i-1, j] + \textsc{Cost}(s_1[i-1], \epsilon)$
            $insertion \leftarrow D[i, j-1] + \textsc{Cost}(\epsilon, s_2[j-1])$
            $D[i, j] \leftarrow \textsc{Min}(substitution, deletion, insertion)$
    return $D[|s_1|, |s_2|]$
```

Source – the authors adaptation of Wagner; Fischer (7, p. 171, 172)

For this setting, the authors provide an $O(|s_1||s_2|)$ algorithm using dynamic programming (or $O(m^2)$ for short, wherein $m = \textsc{Max}(|s_1|, |s_2|)$), which is presented in Algorithm 1. Let us call it WF after Wagner; Fischer. It uses a table named D with $|s_1| + 1$ rows and $|s_2| + 1$ columns to store the state of the dynamic programming. The last column of the last row contains the final edit distance. Table D can also be used to retrieve the operations that would transform s_1 into s_2, if they are needed. Our cost function is simply

$\textsc{Cost}(a, b) = 1$ if $a \neq b$ and 0 if $a = b$ (same characters, no operation). This method is also sometimes called Levenshtein distance or simple edit distance (8, p. 32, 37). A few examples are given below:

a) WF(*"french", "fries"*) $= 0 + 0 + 1 + 0 + 1 + 1 + 1 = 4$, which are for keeping "fr", inserting 'i', keeping 'e', substituting 'n' by 's' and deleting "ch";

b) $\mathrm{WF}(\epsilon, s) = |s|$ for any string $s \neq \epsilon$ for inserting all its characters;

c) $\mathrm{WF}(s, \epsilon) = |s|$ for any string $s \neq \epsilon$ for deleting all its characters.

2.1.2 *q*-gram distance

Looking for alternatives to the edit distance and mainly encouraged by its costly time complexity, Ukkonen (9) studied approximate string matching with two other distance measures. One of them was the q-gram distance, which is considered here.

A q-gram g is a substring of some string s such that the length of g is q. The set $P_q(s)$ of all q-grams found in s is called its profile, and $P_q(s)[g]$ is the number of occurrences of g. Thus, the q-gram distance $D_q(s_1, s_2)$ between strings s_1 and s_2 is described by Eq. (1).

$$\begin{aligned} D_q(s_1, s_2) = & \sum_{g \in \{P_q(s_1) \cap P_q(s_2)\}} |P_q(s_1)[g] - P_q(s_2)[g]| \\ & + \sum_{g \in \{P_q(s_1) \setminus P_q(s_2)\}} P_q(s_1)[g] \\ & + \sum_{g \in \{P_q(s_2) \setminus P_q(s_1)\}} P_q(s_2)[g] \end{aligned} \tag{1}$$

As an example, given $s_1 =$ "001010" and $s_2 =$ "110101", then

$$\begin{aligned} P_2(s_1) &= \{\text{“00”} : 1, \text{“01”} : 2, \text{“10”} : 2\} \\ P_2(s_2) &= \{\text{“01”} : 2, \text{“10”} : 2, \text{“11”} : 1\} \\ D_2(s_1, s_2) &= |2 - 2| + |2 - 2| + 1 + 1 = 2. \end{aligned}$$

Different q-grams may overlap, but are all contiguous. Thus it is possible to use a sliding window algorithm to compute $P_q(s)$. It is a single pass of a window of width q ranging from $s[i]$ to $s[i + q - 1]$ while $0 \leq i \leq |s| - q$. By using hash tables to represent profiles, this computation may be done in $O(|s|)$. To evaluate $D_q(s_1, s_2)$, it suffices to calculate the profiles of s_1 and s_2 and then perform the summation presented in Eq. (1). Therefore the total complexity is $O(|s_1| + |s_2|)$.

2.2 FILTERS

2.2.1 Suffix array

Motivated by the problem of finding all occurrences of some string p (often called pattern) in a large text T, Manber; Myers (10) introduced the suffix array. It is a data structure that stores a sorted array of all suffixes of T. It is an alternative to suffix trees[1] with a construction that is slower, yet much simpler. Furthermore, the lower space complexity and very competitive query time complexity makes it a better option in many applications.

The simplest algorithm to build a suffix array over a text T is a direct encoding of its description: compute all suffixes of T and sort them. Hence, its time complexity is $O(m^2 \log m)$, wherein $m = |T|$. A suffix array does not need to *actually* store the suffixes, as integer references to T are sufficient. Since any T has m suffixes, the space complexity is $O(m)$. Although a suffix tree also has linear space complexity, its hidden constant is larger (16, p. 54).

However, if T is actually an n-sized list of strings, compute all suffixes of all strings in T and sort them. Thus, the time complexity becomes $O(nm^2 \log nm)$ because we have nm suffixes that take $O(m)$ time to compare, m now being the length of the longest string in T. For all s in T, we store a pair $\{j, k\}$ such that $T[j] = s$ and $s[k \ldots |s|]$ is the $(k+1)$-th suffix of s. This yields a space complexity of $O(S)$, wherein $S = \sum_{s \in T} |s|$.

Having built the suffix array, we can search it for all occurrences of p in T via two binary searches: one for the lower bound and one for the upper bound. More specifically, one for the smallest and one for the largest index i such that *the prefix* of the $(i+1)$-th suffix matches p. Each binary search is $O(|p| \log nm)$, which is a dramatic improvement over a naive $O(|p|nm)$ search.

Table 2 illustrates the suffix array (SA) for $T = [\text{“GATA”}, \text{“GACA”}]$ and the result (in **bold**) of a search for $p = \text{“GA”}$ (j and k are explicit for convenience).

Table 2 – Example of suffix array and results of a search

i	SA[i]	T[j]	$s[k \ldots |s|]$
0	$\{j:0, k:3\}$	GATA	A
1	$\{j:1, k:3\}$	GACA	A
2	$\{j:1, k:1\}$	GACA	ACA
3	$\{j:0, k:1\}$	GATA	ATA
4	$\{j:1, k:2\}$	GACA	CA
5	$\boldsymbol{\{j:1, k:0\}}$	**GACA**	**GACA**
6	$\boldsymbol{\{j:0, k:0\}}$	**GATA**	**GATA**
7	$\{j:0, k:2\}$	GATA	TA

Source – the authors adaptation of Halim; Halim (17, p. 260)

[1] For more information on suffix trees, see (11, 12, 13, 14, 15).

2.2.2 *q*-gram index

The idea of the q-gram index is based on Navarro; Baeza-Yates (18). For every s in a list T of strings, the index i_s of s in T and the number of occurrences of each q-gram g in s's profile are stored in a hash table indexed by g. In other words, the profile $P_q(s)$ is computed for all s in T and a pair $\{P_q(s)[g], i_s\}$ is stored in a hash table indexed by g for all g in $P_q(s)$. Every resulting list, one for each q-gram, is then sorted in descending order to speed up queries.

Querying the index starts with the construction of the profile $P_q(p)$ of a query p. Then the index is searched for all g in $P_q(p)$, which returns a possibly empty list of candidate matches. By the construction of the index, all candidates have g in common with p. A candidate is immediately disregarded if its number of occurrences of g is less than the occurrences of g in the query multiplied by a configurable constant $threshold_g$, that is $threshold_g \times P_q(p)[g]$.

Candidates also have their appearances counted. Starting from 0, a candidate has its appearance count increased every time it appears in a search result and is not immediately disregarded. An appearance count *count* of a candidate c means that p and c have *count* q-grams in common (satisfying the threshold). More specifically, $|P_q(p) \cap P_q(c)| = count$.

After searching for all g in $P_q(p)$, candidates with a total appearance count less than $threshold_p \times |P_q(p)|$ are also disregarded, that is, candidates with less than a factor of $threshold_p$ of q-grams in common with p are dismissed. The indices pointed by the candidates that satisfied both thresholds are returned as the results of the query.

Table 3 illustrates the q-gram index (QGI) for

$$T = [\text{“0010100”}, \text{“1101011”}, \text{“0010001”}, \text{“111”}, \text{“111011”}]$$

with $q = 2$ and the result (in **bold**) of a search for $p =$ “00010” with $threshold_g = 0.5$ and $threshold_p = 1.0$.

Table 3 – Example of q-gram index and results of a search

g	QGI[g]
“00”	$\mathbf{\{3,2\},\{2,0\}}$
“01”	$\{2,0\},\{2,1\},\{2,2\},\{1,4\}$
“10”	$\{2,0\},\{2,1\},\{1,2\},\{1,4\}$
“11”	$\{3,4\},\{2,1\},\{2,3\}$

Source – the authors

Notice that $P_2(p) = \{\text{“00”} : 2, \text{“01”} : 1, \text{“10”} : 1\}$. Consequently, indices 0 and 2 are returned as the results of the query because the q-gram count of T[0] and T[2] for

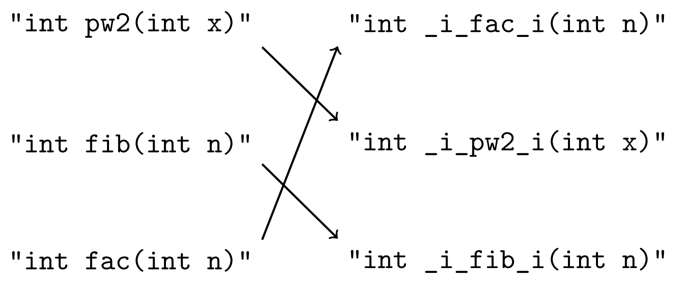
"00", "01" and "10" and the fact that they have these three q-grams in their profile make them pass both thresholds. This is in contrast to the other indexed strings that fail at the second one, because no other string has q-gram "00" and $threshold_p = 1.0$ requires that all candidates have at least all q-grams that p has.

2.3 MATCHING

As seen in Diestel (19), the problem of finding the largest matching of one set of independent[2] vertices to another is called maximum cardinality bipartite matching. More precisely, a matching maps exactly one vertex in the first set to one vertex in the second set and a maximum matching has the largest possible number of edges (or mappings; or matched vertices). By Hall's theorem (1935), the necessary and sufficient condition for a matching to exist in a bipartite graph $G = (\{L \cup R\}, E)$ is that $|S| \leq |N(S)|$ for every $S \subseteq L$ (19, p. 38), wherein $N(S)$ is the set of neighbour (adjacent) vertices of S (also called its neighbourhood). A *complete* bipartite graph clearly satisfies this condition.

In our case, though, the vertices are not *indifferent* of their match (see Fig. 4: the drawn edges compose the only correct answer). This leads us to the concept of stable matchings (or stable marriages). They are commonly presented as follows: we have a set of n men and a set of n women and each one has a preference list ranking *every* person of the opposite sex. The task is to marry every man to a woman in a way that there is no pair of people in which both prefer each other to their current partners (20, 21, 22). The classic famous work that accomplishes this is Gale; Shapley (20). The authors proved that there is at least one matching that obeys these rules for every instance of this problem (therefore there always exists a stable marriage) and their $O(n^2)$ solution, presented in Algorithm 2, is asymptotically optimal in time complexity.

Figure 4 – Example of a correct matching



Source – the authors

However, an important thing to realize is that this classic algorithm does not handle the case of incomplete preference lists (22, 23). The required ranking of every person of the opposite sex leads to a complete bipartite graph, as preference lists can be seen as the

[2] "[...] a set of vertices or of edges is called *independent* if no two of its elements are adjacent." (19, p. 3)

adjacency lists (neighbourhoods) of each vertex, satisfying Hall's theorem. Gale; Shapley proved that, given this condition, a stable marriage always exists no matter how preference lists are arranged. But if they are incomplete, Hall's theorem is not necessarily satisfied and there is no guarantee for a stable marriage and even a matching to exist. For these reasons, an alternative algorithm is proposed in Chapter 3.

Algorithm 2 – Stable marriage algorithm

```
assign each person to be free
while some man $m$ is free do
    $w \leftarrow$ first woman in $m$'s preference list
    if some man $m'$ is engaged to $w$ then
        set $m'$ free
    $m$ becomes engaged to $w$
    for each successor $m'$ of $m$ in $w$'s preference list do
        remove $m'$ from $w$'s preference list
        remove $w$ from $m'$'s preference list
output engagements
```

Source – Irving (22, p. 264)

3 PROPOSED ALGORITHMS

Apart from the algorithms introduced in the literature review, in this Chapter are presented three more algorithms: Left Greedy Matcher (LGM), based on the review; cheating matcher, a baseline to LGM; and simple index, a baseline to suffix array and q-gram index.

3.1 LEFT GREEDY MATCHER

With the limitations of the classic stable marriage algorithm in mind, the Left Greedy Matcher (LGM) is obtained by applying simple modifications to the Gale; Shapley algorithm to make it handle incomplete preference lists. It is represented in Algorithm 3 and works as follows: all people are initially free and we seek a stable marriage while there is a free man m, as in the original algorithm. Then we get and remove the most preferred woman w from m's preference list. If w is already married to some other man m', we check if marrying her to m is *less costly* than leaving her with m' and we change them if it is. Notice that we consider the cost of marrying m to w, not only the order in which w appears in m's preferences. Also, if we changed the marriage, we only set m' free if his preference list is not already empty. Finally, if w was not originally engaged, we simply marry her to m.

Algorithm 3 – Left Greedy Matcher algorithm

```
assign each person to be free
while some man $m$ is free do
    $w \leftarrow$ first woman in $m$'s preference list
    remove $w$ from $m$'s preference list
    if some man $m'$ is engaged to $w$ then
        if cost of $\{m, w\} < \{m', w\}$ then
            set $m'$ free if his preference list is not empty
        else
            continue
    $m$ becomes engaged to $w$
output engagements
```

Source – the authors

The cost of a couple may be determined in many ways. In this work, it is through the rating that each person gives to everyone in his or her preference list (through the string distance functions already described and in accordance to what is explained in Chapter 4). Despite that, in this specific algorithm, only the ratings of m are taken into account and a marriage is changed as soon as a less costly one is found (hence the name).

The biggest difference between LGM and the original stable marriage algorithm is that LGM is clearly not guaranteed to find a stable marriage, because incomplete preference lists do not necessarily satisfy Hall's theorem. Beside that, a woman is removed from a man's preference list exactly in the iteration she is proposed. These removals and the fact that a recently disengaged man is not again considered free if his preference list has become empty guarantees the termination of the algorithm. Additionally, its complexity is $O(n^2)$ because at every iteration an element is removed from a preference list and any of the n preference lists has at most n elements.

3.2 CHEATING MATCHER

On the other hand, the cheating matcher (CM) is an extra matching algorithm that is built to know the answer of the matching from an already mapped input. It always marries the correct couple $\{m, w\}$ if w is present in m's preference list. Therefore it represents an upper bound for any matching algorithm because it always yields the optimal possible marriage. Consequently, it allows the assessment of the results of a filter (because it cannot be responsible for a bad result) and the decrease of accuracy caused by LGM.

3.3 SIMPLE INDEX

There is also a lower bound for the filtering step and it is called simple index. Conceptually, it just stores a sorted list of the n input strings and searches for candidates with the same binary searches as the suffix array. In fact, it is implemented as a suffix array without the suffixes, resulting in a time complexity of $O(nm \log n)$ to build it and of $O(|p| \log n)$ to search it, wherein m is the length of the longest indexed string and p is a query string. Being the simplest technique, its results will be compared to the other two filters to analyze their cost-benefit.

4 METHODOLOGY

All the code for this work can be found in `https://github.com/ggabriel96/mapnames`. It is written in Python 3 and is explained below.

A framework for benchmarking and comparing techniques was built. It allows us to easily switch between algorithms and set their parameters. It is illustrated by Algorithm 4 and works as follows: we first parse the command-line arguments to get the settings that will determine: dataset; bipartite matcher; filter; string distance function; parameters of the algorithms; and output destination. Then we call the actual benchmarking function that returns the results and we output them. The code for both the framework and the actual benchmarking function are in the file named `benchmark.py`. This is the entry file and all available options and their descriptions may be consulted running `python benchmark.py --help`.

Algorithm 4 – Framework algorithm

```
1 settings ← ParseCommandLine()
2 dataset ← LoadTestCase(settings)
3 Matcher, Filter, Distance ← SelectAlgorithms(settings)
4 results ← Benchmark(dataset, Matcher, Filter, Distance)
5 Output(results, settings)
```

Source – the authors

There are several algorithms available to choose from and they work and can be set independently. For bipartite matcher, they are LGM and CM; the filter is one of simple index, suffix array or q-gram index; and the string distance function may be edit distance or q-gram distance. Matchers can be found in the file `mapnames/graph.py` and filters and string distances in `mapnames/string.py`.

Algorithm 5 shows the benchmarking function. It starts by splitting the dataset, which is a JSON file, into the left (L) and right (R) sets of strings. Right before the main part, it stores the current time so we can later evaluate the total run time of our solution. The main part is where the actual proposed approach takes place, divided in three steps: the computation of the filters, preference lists and matching. Then the total run time and accuracy are also calculated and we return the results. The accuracy is calculated by counting the number of correct matches and dividing that by the size of the input (n). Unmatched strings are counted as wrong matches.

In the first step of our approach we build indexes of the input strings: one on L to be queried with an r from R and one on R to be queried with an l from L. Each query returns a list of indices pointing to the indexed strings. All indices in the result of a query are considered admissible for the second step: the computation of preference lists (Algorithm 6). It is quite simple: call the distance function with every u in U paired with each admissible v in the result of filtering V and sort the list. Parameters U and

Algorithm 5 – Benchmarking algorithm

```
function BENCHMARK(settings, MATCHER, FILTER, DISTANCE)
    $L, R \leftarrow$ SPLIT(dataset)
    start $\leftarrow$ CURRENTTIME()
    FILTERONL $\leftarrow$ FILTER($L$)
    FILTERONR $\leftarrow$ FILTER($R$)
    BUILDPREFERENCES($L$, $R$, FILTERONR, DISTANCE)
    BUILDPREFERENCES($R$, $L$, FILTERONL, DISTANCE)
    matching $\leftarrow$ MATCHER($L$, $R$)
    time $\leftarrow$ start – CURRENTTIME()
    accuracy $\leftarrow$ ACCURACY(matching, dataset)
    return matching, accuracy, time
```

Source – the authors

V alternate between L and R in the calls from Algorithm 5. The value returned by the distance function represents the rating that u gives to v. Finally, in the third step, the bipartite matching algorithm is invoked with L and R.

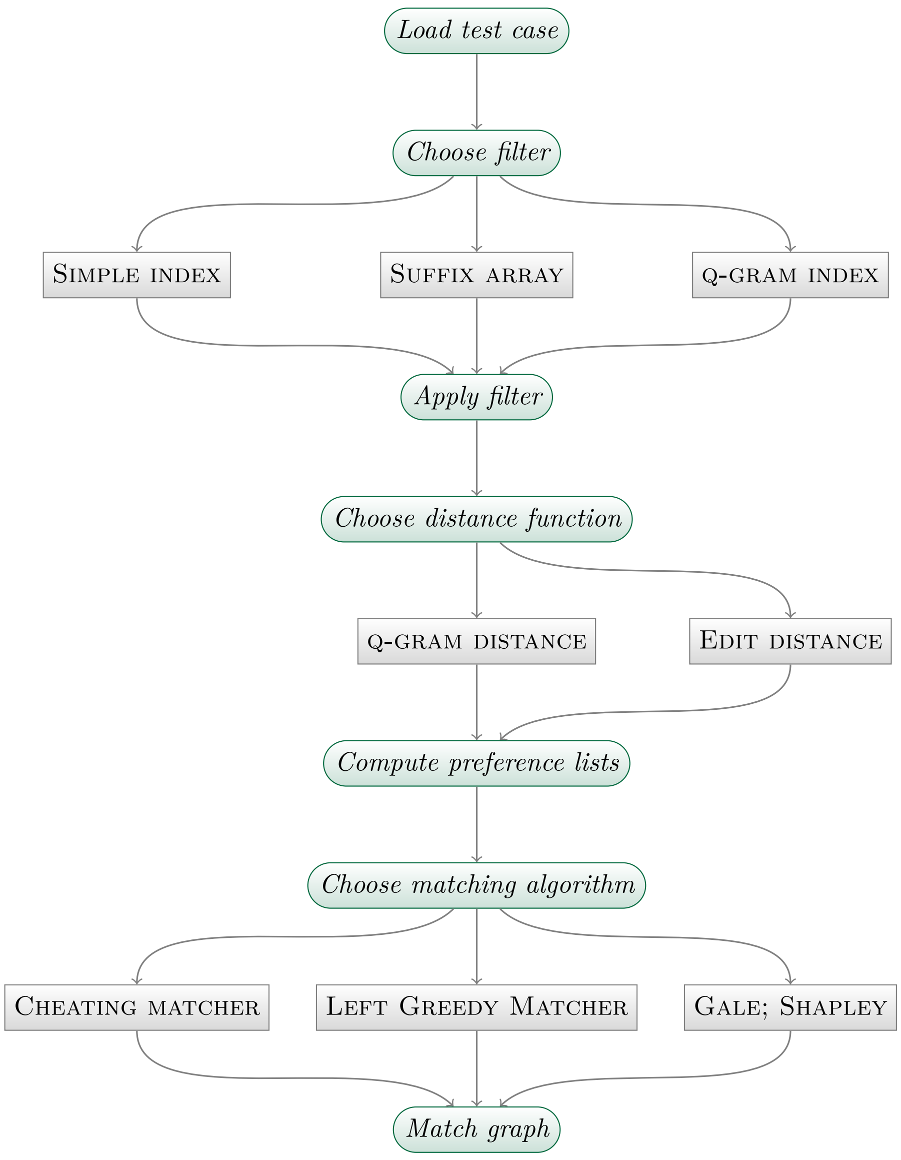
Algorithm 6 – Algorithm to build preference lists

```
function BUILDPREFERENCES($U$, $V$, FILTERONV, DISTANCE)
    for all $u \in U$ do
        admissible $\leftarrow$ FILTERONV($u$)
        $u_p \leftarrow$ DISTANCE($u$, $v$) for all $v \in \{V \cap admissible\}$
        SORT($u_p$)                              » $u_p$ is the preference list of $u$
```

Source – the authors

Figure 5 illustrates the logic procedure of the main part of our approach with its three steps, namely filtering vertices, computing preference lists with string distances, and bipartite matching.

Figure 5 – Illustrative diagram of our approach



Source – the authors

5 RESULTS

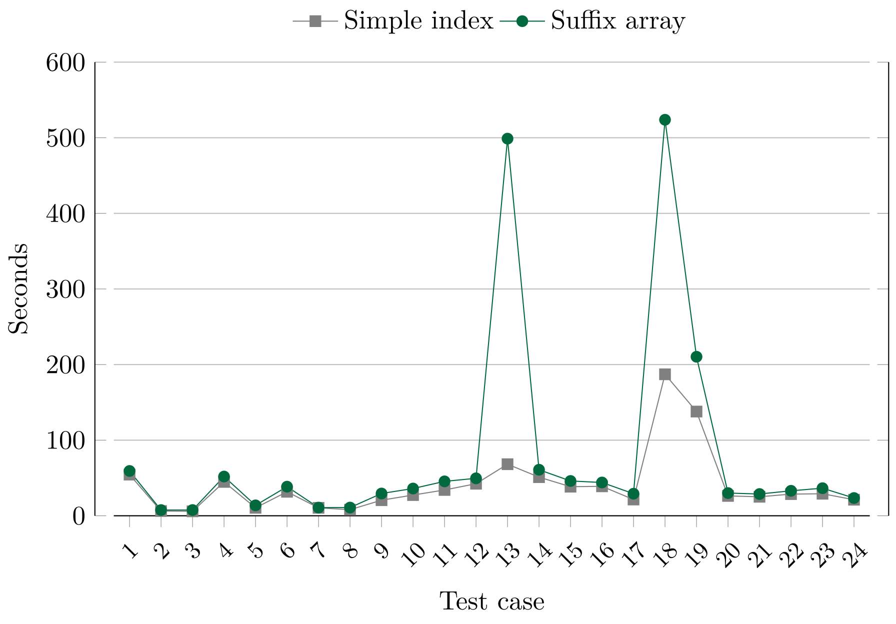
In order to analyze the impact of filtering and matching algorithms independently, the results of the cheating matcher (CM) with the three possible filters are discussed first, in Sections 5.1 and 5.2. The results of the Left Greedy Matcher (LGM) will be compared to CM with suffix array in Section 5.3. Even though all tests were run with q-gram distance as string distance function (it is *much* faster than edit distance), it will only interfere with results when matching with LGM because CM always chooses the correct answer if it is available.

An utility executable script named `benchmark.sh` was created to automatically execute the benchmark for all test cases. The command that was issued inside its loop to obtain the results are listed alongside them. All benchmarks were executed in a server provided by the Federal University of Fronteira Sul with an Intel® Xeon® E7-4850 (24) and 128 GB of RAM.

5.1 SUFFIX ARRAY

Figure 6 presents a comparison of the total run time in seconds between filtering with simple index and suffix array. We can see that the run time overhead of the suffix array is mainly negligible, with the exceptions of test cases 18 and 19, which are the largest ones in size (n) and are among the largest in string length (m); and test case 13, the largest in *both* maximum and average string length. Larger run time gaps are due to longer strings, which evince the difference in time complexity between the algorithms.

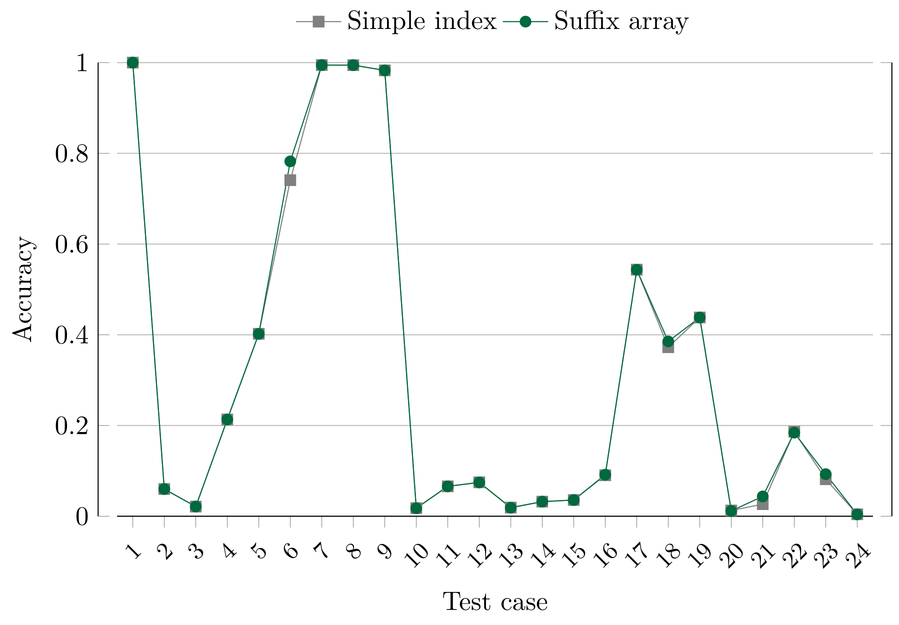
Figure 6 – Run time of CM with simple index and suffix array



Source – the authors

Figure 7 presents a comparison of the achieved accuracy between filtering with simple index and suffix array. The plot shows us that both algorithms are not being able to consistently filter candidates without also discarding the correct answers to queries. Despite that, Tables 5 and 6 in Appendix A show us that, for both filters, at least 75% of the preference lists of all test cases have size less than or qual to 2 and the average size of preference lists is almost always less than 3 (with the exceptions of suffix array in test cases 13 and 18 with averages of 10.28 and 6.19, respectively). This is an immensely desirable behaviour if only they were not filtering out the correct answers, as previously mentioned. The main differences between them is that simple index has lower maximums and standard deviations (σ).

Figure 7 – Accuracy of CM with simple index and suffix array



Source – the authors

Results for the simple index were obtained by issuing the command below. Meanwhile, for the suffix array, the only difference is `--filter sa`.

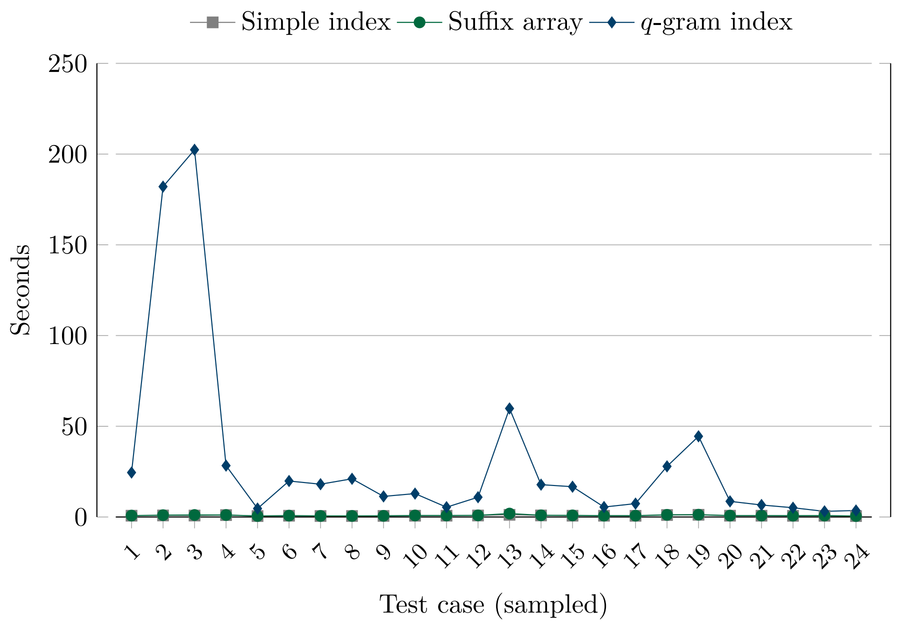
```
python benchmark.py ${folder}/${file} \
    --comparison --matcher c --filter si --distance qg -q 2 \
    --outdir cmpf
```

5.2 Q-GRAM INDEX

The second comparison is between all three filters. It is separated from the previous one because the q-gram index is much slower to query than simple index and suffix array. Therefore, this comparison was run for a random subset of size 1500 of the test cases, sampled with the seed 7814958244. Parameters $threshold_g$ and $threshold_p$ are fixed at 0.5 and $q = 2$.

Figure 8 presents a comparison of the total run time in seconds and Fig. 9 of the achieved accuracy between filtering with simple index, suffix array and q-gram index, while Tables 7 to 9 in Appendix A respectively describe their resulting preference list sizes. Those sizes are evidence that q-gram index is much more "sensitive" to variations in the input, in the sense that there are many more empty preference lists and standard deviations are overall much higher than simple index and suffix array. And although it scored a higher accuracy in 14 of the 24 test cases, the run time comparison suggests that its advantage is not scalable.

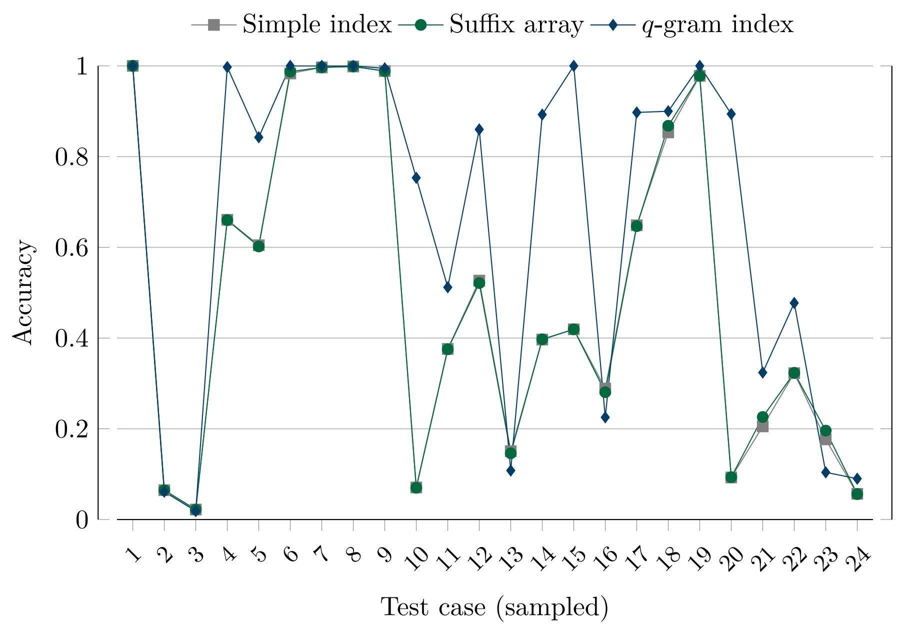
Figure 8 – Run time of CM with all filters (random sample)



Source – the authors

Experimental tests varying $threshold_g$ and $threshold_p$ indicate that, in its current form, q-gram index generally maintains higher accuracies but never gets comparably fast as the other two filters and still has an accuracy upper bound that is *not* 1.0. For example, this happens with test case 11 with a peak accuracy of 0.706 with the same command as above. This behaviour is observed by setting (in code) both thresholds to zero, which means that there is no restriction to a candidate beyond just having a q-gram in common with the query. We say that this is an indication because these tests were not run on whole test cases, but the program would nevertheless take too long to run (and probably consume too much memory) otherwise.

Figure 9 – Accuracy of CM with all filters (random sample)



Source – the authors

The results above were obtained by issuing the command below, but varying `--filter` over `si`, `sa` and `qg`.

```
python benchmark.py ${folder}/${file} \
    --comparison --matcher c --filter qg --distance qg -q 2 \
    --outdir cmpf/rand --size 1500 --seed 7814958244
```

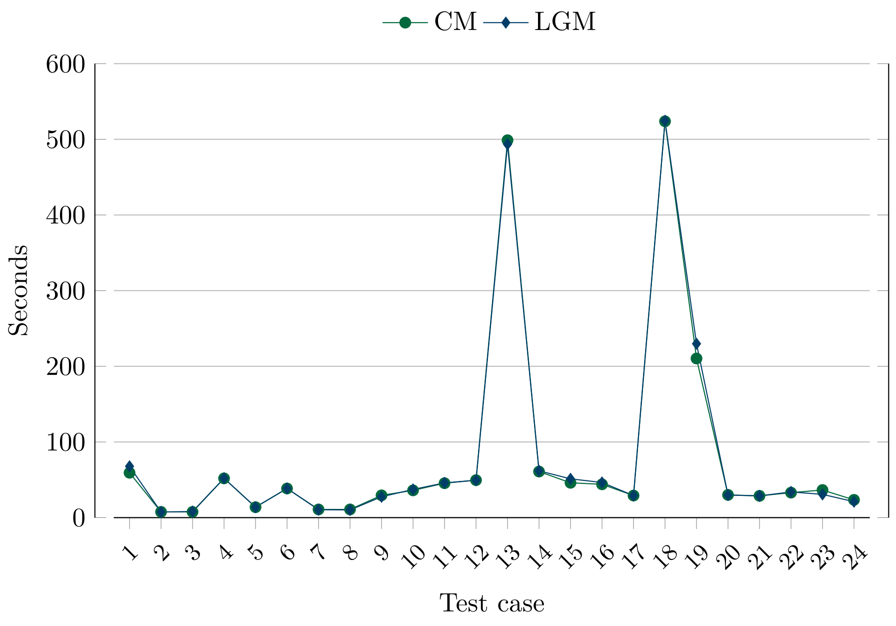
5.3 LEFT GREEDY MATCHER

Finally, in this section LGM is compared to CM. The last benchmark, of LGM, was executed as below (CM has already been benchmarked with suffix array in our first comparison).

```
python benchmark.py ${folder}/${file} \
    --comparison --matcher lgm --filter sa --distance qg -q 2 \
    --outdir cmpm
```

Figure 10 presents a comparison of the total run time in seconds between maching with CM and LGM. It is noticeable that they barely differ, indicating that the run time cost of matching algorithms is negligible when compared to that of computing preference lists (which include building and invoking the filters and calculating string distances). Hence the cost of the matching step is not of our concern.

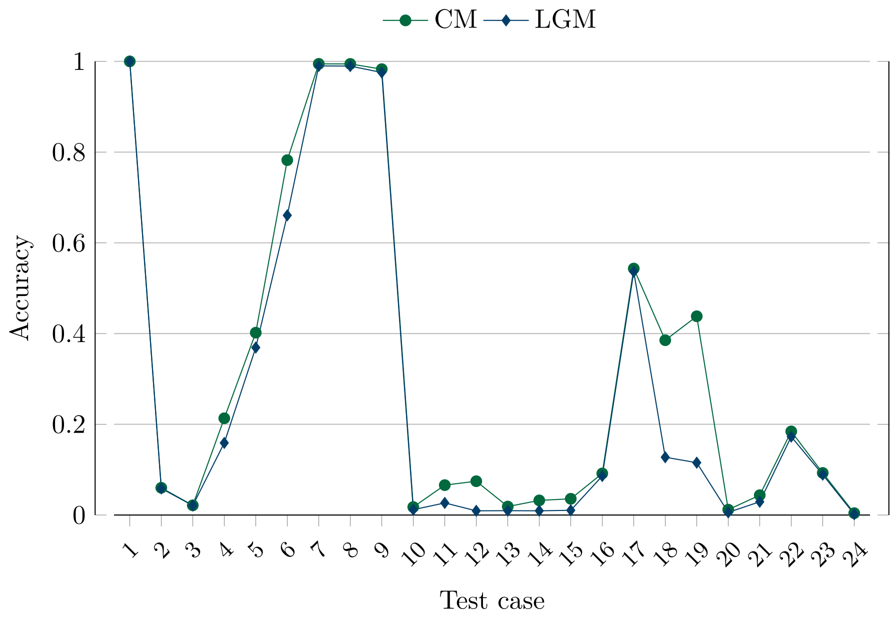
Figure 10 – Run time of CM and LGM with suffix array



Source – the authors

Figure 11 presents a comparison of the achieved accuracy between maching with CM and LGM. Generally, 19 test cases have a decrease in accuracy below 5%. The ones that have it fallen beyond 5% are test cases 4, 6, 12, 18 and 19 (see Table 4; values are rounded). However, the accuracy limitations caused by the poor results of the filtering step inhibited a proper evaluation of the accuracy compromises caused by LGM or q-gram distance.

Figure 11 – Accuracy of CM and LGM with suffix array



Source – the authors

Table 4 – Test cases with accuracy difference beyond 5%

	Accuracy	
Test case	CM	LGM
4th	0.21	0.16
6th	0.78	0.66
12th	0.07	0.01
18th	0.39	0.13
19th	0.44	0.12

Source – the authors

6 CONSTRAINTS AND CONCLUSIONS

At the 2018 ICCAD CAD Contest was presented a problem (3) that, given two sets L and R of design component names such that R is the result of applying an unknown transformation to L, required the matching of every name in L to its corresponding transformed name in R with the aid of artificial intelligence (more specifically, supervised machine learning). In this work, we presented a deterministic approach that used graph theory and approximate string matching to try to compute the correct matchings. We have not used supervised machine learning and disregarded solutions that took more than 15 minutes (900 seconds) to run or used more than approximately 16 GiB of RAM. This is why a filtering step was needed before the matching and q-gram distance was used instead of edit distance. Furthermore, parallel computing was not explored because we kept the rule from the contest that did not allow it.

The most significant result of this work is that the filtering step is the bottleneck of the proposed methodology. Matching algorithms that require a complete bipartite graph cannot be used because of time or memory. Meanwhile, the ones that work with an incomplete bipartite graph have failed to achieve 100% of accuracy because the index that filtered the edges that should be considered for matching were inaccurate. This is demonstrated by the poor results of the cheating matcher (CM). As we have seen, CM is built from the input and thus always matches the correct pair of strings. Its purpose was to allow us to independently assess the impact of a filter and of a different matcher. By cheating and always choosing the correct pair of strings, CM achieves 100% of accuracy on a complete bipartite graph and would also achieve it on an incomplete bipartite graph if the edges connecting the correct pair of strings were present. Hence, if CM does not yield the final correct matching, it means that the graph does not have those edges. Since the index is responsible for filtering the edges, it is the cause of the lack of accuracy of the whole solution. In addition, it prevents a proper reasoning about the impact of the Left Greedy Matcher (LGM) and the string distance functions (mainly q-gram distance) because we cannot know if they would perform better with an index that did not filter out the edges connecting the correct pair of strings.

In Annex A are listed a few pairs of strings from test case 11 that the filters presented in this work cannot deal with. Notice that there is no apparent relationship from the original to the transformed strings. There are also cases where original strings are very long and transformed strings are very short, causing the loss of too much information. This happens, for instance, in test case 13 with strings as long as 1600 characters being reduced to just 60. Entries like these are plenty in the input files. Despite that, the works of the winning teams of the contest managed to achieve 100% of accuracy. When they are published, it will be possible to study their solutions and perhaps better understand the compromises of our choices in order to propose improvements to our approach.

REFERENCES

1 ROBERTSON, Colin et al. **Decorated Names**. Microsoft Corporation. 4 Nov. 2016. Available from: <`https://docs.microsoft.com/en-us/cpp/build/reference/decorated-names`>. Visited on: 2 May 2018.

2 ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. **Style Guide for Python Code**. PEP 8. 1 Aug. 2013. Available from: <`https://www.python.org/dev/peps/pep-0008/`>. Visited on: 11 May 2018.

3 WU, Chi-An; HUANG, Ching-Yi; HSU, Chih-Jen. **Problem A: Smart EC: Program-Building for Name Mapping**. Version 2018-04-25. CAD Contest @ ICCAD. Cadence Design Systems, Inc. 2018. Available from: <`http://iccad-contest.org/2018/Problem_A/2018ICCADContest_ProblemA.pdf`>. Visited on: 11 May 2018.

4 BHATNAGAR, Himanshu. **Advanced ASIC Chip Synthesis**: Using Synopsys® Design Compiler™ Physical Compiler™ and PrimeTime®. 2. ed. [S.l.]: Springer US, 2002. 328 pp. ISBN 978-0-306-47507-8. DOI: `10.1007/b117024`.

5 QUALCOMM Datacenter Technologies Announces Commercial Shipment of Qualcomm Centriq 2400 – The World's First 10nm Server Processor and Highest Performance Arm-based Server Processor Family Ever Designed. Qualcomm Technologies, Inc. 8 Nov. 2017. Available from: <`https://www.qualcomm.com/news/releases/2017/11/08/qualcomm-datacenter-technologies-announces-commercial-shipment-qualcomm`>. Visited on: 5 May 2018.

6 INTEL® Core™ i5-6300HQ Processor. Intel Corporation. 2015. Available from: <`https://ark.intel.com/products/88959/Intel-Core-i5-6300HQ-Processor-6M-Cache-up-to-3_20-GHz`>. Visited on: 3 June 2018.

7 WAGNER, Robert A.; FISCHER, Michael J. The String-to-String Correction Problem. **Journal of the ACM**, ACM, New York, NY, USA, v. 21, n. 1, p. 168–173, 6 pp., Jan. 1974. ISSN 0004-5411. DOI: `10.1145/321796.321811`.

8 NAVARRO, Gonzalo. A Guided Tour to Approximate String Matching. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 33, n. 1, p. 31–88, Mar. 2001. ISSN 0360-0300. DOI: `10.1145/375360.375365`.

9 UKKONEN, Esko. Approximate string-matching with q-grams and maximal matches. **Theoretical Computer Science**, v. 92, n. 1, p. 191–211, 1992. ISSN 0304-3975. DOI: `10.1016/0304-3975(92)90143-4`.

10 MANBER, Udi; MYERS, Gene. Suffix Arrays: A New Method for On-Line String Searches. **SIAM Journal on Computing**, v. 22, n. 5, p. 935–948, 1993. DOI: 10.1137/0222058.

11 APOSTOLICO, Alberto. The Myriad Virtues of Subword Trees. In: ________. **Combinatorial Algorithms on Words**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985. p. 85–96. ISBN 978-3-642-82456-2.

12 GALIL, Z; GIANCARLO, R. Data structures and algorithms for approximate string matching. **Journal of Complexity**, v. 4, n. 1, p. 33–72, 1988. ISSN 0885-064X. DOI: 10.1016/0885-064X(88)90008-8.

13 UKKONEN, Esko. Approximate string-matching over suffix trees. In: ________. **Combinatorial Pattern Matching**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 228–242. ISBN 978-3-540-47732-7.

14 GUSFIELD, Dan. **Algorithms on Strings, Trees, and Sequences**: Computer Science and Computational Biology. [S.l.]: Cambridge University Press, 1997. DOI: 10.1017/CBO9780511574931.

15 COLE, Richard; GOTTLIEB, Lee-Ad; LEWENSTEIN, Moshe. Dictionary Matching and Indexing with Errors and Don't Cares. In: PROCEEDINGS of the Thirty-sixth Annual ACM Symposium on Theory of Computing. Chicago, IL, USA: ACM, 2004. (STOC '04), p. 91–100. ISBN 1-58113-852-0. DOI: 10.1145/1007352.1007374.

16 ABOUELHODA, Mohamed Ibrahim; KURTZ, Stefan; OHLEBUSCH, Enno. Replacing suffix trees with enhanced suffix arrays. **Journal of Discrete Algorithms**, v. 2, n. 1, p. 53–86, 2004. The 9th International Symposium on String Processing and Information Retrieval. ISSN 1570-8667. DOI: 10.1016/S1570-8667(03)00065-0.

17 HALIM, Steven; HALIM, Felix. **Competitive Programming**: The New Lower Bound of Programming Contests. 3. ed. [S.l.]: Lulu, 2013. 447 pp. Available from: <https://cpbook.net>.

18 NAVARRO, Gonzalo; BAEZA-YATES, Ricardo. A Practical q-Gram Index for Text Retrieval Allowing Errors. **CLEI Electronic Journal**, v. 1, n. 2, 1998.

19 DIESTEL, Reihard. **Graph Theory**. 5. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2017. v. 173. 429 pp. (Graduate Texts in Mathematics). ISBN 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3.

20 GALE, D.; SHAPLEY, L. S. College Admissions and the Stability of Marriage. **The American Mathematical Monthly**, Mathematical Association of America, v. 69, n. 1, p. 9–15, 1962. DOI: 10.2307/2312726.

21 GUSFIELD, Dan. Three fast algorithms for four problems in stable marriage. **SIAM Journal on Computing**, SIAM, v. 16, n. 1, p. 111–128, 1987. DOI: 10.1137/0216010.

22 IRVING, Robert W. Stable marriage and indifference. **Discrete Applied Mathematics**, v. 48, n. 3, p. 261–272, 1994. DOI: 10.1016/0166-218X(92)00179-P.

23 IWAMA, Kazuo et al. Stable Marriage with Incomplete Lists and Ties. In: ________. **Automata, Languages and Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 443–452. ISBN 978-3-540-48523-0.

24 INTEL® Xeon® Processor E7-4850. Intel Corporation. 2011. Available from: <https://ark.intel.com/products/53574/Intel-Xeon-Processor-E7-4850-24M-Cache-2-00-GHz-6-40-GT-s-Intel-QPI->. Visited on: 25 Nov. 2018.

APPENDIX A – SIZES OF PREFERENCE LISTS

Table 5 – Sizes of preference lists using simple index

	Sizes of preference lists						
Test case	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	1	2	1.48	0.50	1.00	1.00	2.00
2nd	1	2	1.95	0.22	2.00	2.00	2.00
3rd	1	2	1.98	0.14	2.00	2.00	2.00
4th	1	33	1.90	0.51	2.00	2.00	2.00
5th	1	618	1.68	3.44	1.00	2.00	2.00
6th	1	3691	1.86	16.52	1.00	2.00	2.00
7th	1	55	1.56	1.58	1.00	1.00	2.00
8th	1	55	1.55	1.62	1.00	1.00	2.00
9th	1	454	1.44	1.98	1.00	1.00	2.00
10th	1	18	2.00	0.05	2.00	2.00	2.00
11th	2	3	2.00	0.00	2.00	2.00	2.00
12th	2	2	2.00	0.00	2.00	2.00	2.00
13th	1	234	2.02	1.45	2.00	2.00	2.00
14th	1	144	2.00	0.35	2.00	2.00	2.00
15th	2	2	2.00	0.00	2.00	2.00	2.00
16th	2	21	2.00	0.05	2.00	2.00	2.00
17th	1	2	2.00	0.04	2.00	2.00	2.00
18th	1	20731	2.06	29.83	2.00	2.00	2.00
19th	2	2	2.00	0.00	2.00	2.00	2.00
20th	2	3	2.00	0.00	2.00	2.00	2.00
21st	1	2	2.00	0.01	2.00	2.00	2.00
22nd	2	2	2.00	0.00	2.00	2.00	2.00
23rd	1	750	2.02	2.22	2.00	2.00	2.00
24th	1	1000	2.24	15.46	2.00	2.00	2.00

Source – the authors

Table 6 – Sizes of preference lists using suffix array

	Sizes of preference lists						
Test case	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	1	33	1.48	0.51	1.00	1.00	2.00
2nd	1	233	1.98	2.00	2.00	2.00	2.00
3rd	1	2	1.98	0.14	2.00	2.00	2.00
4th	1	298	1.93	1.63	2.00	2.00	2.00
5th	1	1727	1.75	9.48	1.00	2.00	2.00
6th	1	3691	2.11	17.04	1.00	2.00	2.00
7th	1	429	1.58	3.20	1.00	1.00	2.00
8th	1	429	1.57	3.31	1.00	1.00	2.00
9th	1	1218	1.47	5.52	1.00	1.00	2.00
10th	1	70	2.00	0.20	2.00	2.00	2.00
11th	2	36416	2.52	137.07	2.00	2.00	2.00
12th	1	9	2.00	0.02	2.00	2.00	2.00
13th	1	109473	10.28	909.98	2.00	2.00	2.00
14th	1	1535	2.01	3.74	2.00	2.00	2.00
15th	2	2	2.00	0.00	2.00	2.00	2.00
16th	1	45	2.00	0.11	2.00	2.00	2.00
17th	1	489	2.00	1.49	2.00	2.00	2.00
18th	1	699433	6.19	1265.43	2.00	2.00	2.00
19th	2	2	2.00	0.00	2.00	2.00	2.00
20th	1	1061	2.02	4.12	2.00	2.00	2.00
21st	1	41	1.99	0.20	2.00	2.00	2.00
22nd	1	2	2.00	0.00	2.00	2.00	2.00
23rd	1	13755	2.16	38.90	2.00	2.00	2.00
24th	1	1000	2.28	16.07	2.00	2.00	2.00

Source – the authors

Table 7 – Sizes of preference lists using simple index for random sample of 1500 elements

	Sizes of preference lists						
Test case	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	1	2	1.48	0.50	1	1	2
2nd	1	2	1.95	0.22	2	2	2
3rd	1	2	1.98	0.14	2	2	2
4th	1	2	1.90	0.30	2	2	2
5th	1	41	1.69	1.12	1	2	2
6th	1	2	1.54	0.50	1	2	2
7th	1	4	1.35	0.48	1	1	2
8th	1	3	1.34	0.48	1	1	2
9th	1	2	1.39	0.49	1	1	2
10th	2	2	2.00	0.00	2	2	2
11th	2	2	2.00	0.00	2	2	2
12th	2	2	2.00	0.00	2	2	2
13th	1	2	2.00	0.02	2	2	2
14th	2	2	2.00	0.00	2	2	2
15th	2	2	2.00	0.00	2	2	2
16th	2	2	2.00	0.00	2	2	2
17th	2	2	2.00	0.00	2	2	2
18th	2	2	2.00	0.00	2	2	2
19th	2	2	2.00	0.00	2	2	2
20th	2	2	2.00	0.00	2	2	2
21st	2	2	2.00	0.00	2	2	2
22nd	2	2	2.00	0.00	2	2	2
23rd	2	2	2.00	0.00	2	2	2
24th	2	2	2.00	0.00	2	2	2

Source – the authors

Table 8 – Sizes of preference lists using suffix array for random sample of 1500 elements

	Sizes of preference lists						
Test case	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	1	2	1.48	0.50	1	1	2
2nd	1	2	1.95	0.22	2	2	2
3rd	1	2	1.98	0.14	2	2	2
4th	1	2	1.90	0.30	2	2	2
5th	1	81	1.71	2.10	1	2	2
6th	1	2	1.52	0.50	1	2	2
7th	1	4	1.35	0.48	1	1	2
8th	1	3	1.34	0.48	1	1	2
9th	1	2	1.39	0.49	1	1	2
10th	2	2	2.00	0.00	2	2	2
11th	2	2	2.00	0.00	2	2	2
12th	1	2	2.00	0.02	2	2	2
13th	1	10	2.02	0.36	2	2	2
14th	2	2	2.00	0.00	2	2	2
15th	2	2	2.00	0.00	2	2	2
16th	2	2	2.00	0.00	2	2	2
17th	2	2	2.00	0.00	2	2	2
18th	1	2	1.99	0.08	2	2	2
19th	2	2	2.00	0.00	2	2	2
20th	2	2	2.00	0.00	2	2	2
21st	1	2	1.99	0.09	2	2	2
22nd	2	2	2.00	0.00	2	2	2
23rd	1	24	2.00	0.41	2	2	2
24th	1	2	2.00	0.02	2	2	2

Source – the authors

Table 9 – Sizes of preference lists using q-gram index for random sample of 1500 elements

	Sizes of preference lists						
Test case	min.	max.	μ	σ	Q_1	Q_2	Q_3
1st	0	118	20.09	21.86	3.0	11.0	31.00
2nd	0	1088	400.66	518.17	0.0	4.0	1086.00
3rd	0	1144	442.56	550.69	0.0	2.0	1143.00
4th	0	296	19.81	25.98	4.0	11.0	27.00
5th	0	93	5.04	10.21	0.0	1.0	4.00
6th	0	281	25.61	38.57	1.0	5.0	38.00
7th	0	293	45.36	41.69	11.0	32.0	72.00
8th	0	207	52.90	49.10	11.0	32.0	91.25
9th	0	69	7.13	10.34	1.0	3.0	8.00
10th	0	862	35.01	69.86	0.0	0.0	56.00
11th	0	69	1.98	6.26	0.0	0.0	1.00
12th	0	89	6.04	15.16	0.0	0.0	3.25
13th	0	314	48.98	98.21	0.0	0.0	24.00
14th	0	172	11.28	30.69	0.0	1.0	6.00
15th	0	76	5.87	11.59	0.0	0.5	6.00
16th	0	96	1.78	9.16	0.0	0.0	0.00
17th	0	70	3.78	6.56	0.0	2.0	4.00
18th	0	176	22.74	41.76	0.0	2.0	20.00
19th	0	212	37.01	59.10	0.0	5.0	34.00
20th	0	178	9.91	20.31	0.0	0.0	11.00
21st	0	205	3.57	14.39	0.0	0.0	0.00
22nd	0	54	1.68	7.54	0.0	0.0	1.00
23rd	0	32	0.90	3.60	0.0	0.0	0.00
24th	0	315	7.61	15.23	0.0	1.0	8.00

Source – the authors

ANNEX A – SAMPLE ENTRIES FROM TEST CASE 11

The entries below were taken from lines 10839 to 10859 of test case 11[1].

```
"CAVU/fp/vum_sht[489][34]": "tendry/tendry_2/cuj0_61/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[481][55]": "tendry/tendry_23/cuj0_60/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[16][37]": "tendry/tendry_5/cuj0_2/H0/H0/Y_DS0",
"CAVU/fp/vum_sht[487][53]": "tendry/tendry_21/cuj1_60/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[14][56]": "tendry/tendry_24/cuj1_1/H0/H1/Y_DS0",
"CAVU/fp/vum_sht[327][41]": "tendry/tendry_9/cuj1_40/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[247][10]": "vaadim/tendry_10/cuj1_30/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[89][29]": "vaadim/tendry_29/cuj0_11/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[351][1]": "vaadim/tendry_1/cuj1_43/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[147][2]": "vaadim/tendry_2/cuj1_18/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[87][48]": "tendry/tendry_16/cuj1_10/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[329][62]": "tendry/tendry_30/cuj0_41/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[272][40]": "tendry/tendry_8/cuj0_34/H0/H0/Y_DS0",
"CAVU/fp/vum_sht[11][12]": "vaadim/tendry_12/cuj1_1/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[356][40]": "tendry/tendry_8/cuj0_44/H0/H1/Y_DS0",
"CAVU/fp/vum_sht[47][17]": "vaadim/tendry_17/cuj1_5/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[45][36]": "tendry/tendry_4/cuj0_5/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[109][6]": "vaadim/tendry_6/cuj0_13/H1/H1/Y_DS0",
"CAVU/fp/vum_sht[313][5]": "vaadim/tendry_5/cuj0_39/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[43][55]": "tendry/tendry_23/cuj1_5/H1/H0/Y_DS0",
"CAVU/fp/vum_sht[337][50]": "tendry/tendry_18/cuj0_42/H1/H0/Y_DS0",
```

[1] In this work, test cases are indexed from 1.