The handle http://hdl.handle.net/1887/29716 holds various files of this Leiden University dissertation.

Author: Schraagen, Marijn Paul
Title: Aspects of record linkage
Issue Date: 2014-11-11
Chapter 4

Indexing edit distance

In this chapter an algorithm is described for efficient computation of edit distance in a large search space given a distance threshold. The chapter is based on the paper *Complete coverage for approximate string matching in record linkage using bit vectors* [113].

4.1 Introduction

In Section 2.3 the concept of blocking is described, which is used to limit the number of pair-wise record comparisons when performing record linkage on large datasets. Blocking has the disadvantage that pairs of similar records from different blocks are not found. This issue, referred as the coverage problem, is addressed by the method described in this chapter which computes all pairs of records within a threshold of Levenshtein edit distance. Complete coverage is provided without using standardization of data or selection of possible links. The main idea behind the method is to use a tree-based index on single characters contained in a record. The index is small and fast to compute by using a fixed-size binary representation of records. The method can be extended to other similarity measures, such as Jaro distance. Using this method, blocking is no longer necessary for practical levels of the distance threshold.

This chapter is structured as follows: in Section 4.2, an overview of related work is provided. Section Section 4.3 describes the tree construction and traversal algorithms.
A proof is provided that this algorithm finds all pairs of records within a given threshold on Levenshtein distance. Sections 4.4 and 4.5 discuss various optimizations to limit the number of tree traversal steps without loss of coverage. Section 4.6 provides results from an experiment on the Genlias dataset. A comparison to other complete coverage methods is provided in Section 4.7. Additionally, this section contains a discussion of combining blocking and tree indexing which combines the efficiency of both methods at the loss of complete coverage. In a third set of experiments, described in Section 4.8, tree indexing is applied to Jaro distance. Section 4.9 concludes.

4.2 Related work

The method described in this chapter applies a tree structure to organize the data. Tree indexing is a well-known technique for efficient database querying. The B-tree [9] is a data structure that partitions the values of database keys in a tree structure permitting logarithmic search. Full-text indexing can be achieved efficiently with a trie [41] where strings are represented by a path in a tree with consecutive characters as path nodes. Tries can be compressed or combined with other data structures to save storage space or to increase retrieval performance (see, e.g., [56]).

Tree indexing for approximate string matching has been studied in the context of matching relatively small patterns against relatively long texts, such as DNA sequence matching or document retrieval (see, e.g., [88] for an overview). The approximate matching utility Agrep divides the pattern into $k + 1$ parts for a maximum error of $k$, such that at least one of the parts must match in an exact way. Then, the bitap algorithm (also called shift-or) is used to find occurrences of the exact matching subpattern in the text [140]. A combination of filtering and hashing is Locality Sensitive Hashing [46], where a record (or point) is hashed multiple times using different dimension reductions. The different hash functions are chosen such that at least one of the functions is likely to provide the same hash key for two similar records.

A similar pattern division filtering technique, but allowing for small errors in the partitions, is the basis of the $q$-gram tree approach by [90] which is similar to earlier approaches using suffix automata [66] and suffix trees [89]. A standard dynamic programming algorithm is used to check for partition errors. A key element in this approach
is sampling. The pattern is divided into $q$-grams such that in every sequence of $n$ pattern $q$-grams, there is a maximum of $m$ $q$-grams with a large error compared to the text, and all other parts have no or only small errors. The text is also divided into $q$-grams, and for every $n$ text $q$-grams a selection of $m+1$ $q$-grams is used to check against the pattern $q$-grams. The number of samples ensures that at least 1 text $q$-gram can be found with a small error to a pattern $q$-gram, which can save a large number of comparisons.

Indices using suffix trees can be quite large (up to 65 times the size of the indexed text), but space efficiency of the tree structure can be improved [76] to lower memory requirements.

Sampling is not useful in record linkage, because there has to be at least one sample from every record, which means that there is no reduction in the number of record comparisons. Filtering, in contrast, can be used in record linkage. A comparison of filtering with the new indexing method presented in this work is provided in Section 4.7.

4.3 Approach

The record linkage problem can be stated as follows: given two sets of records $A_1, A_2$, a similarity measure\(^1\) $s$ and a similarity threshold $t$, find the set $B \subseteq A_1 \times A_2$ of pairs $(p, q)$ for which $s(p, q) \geq t$.

The set $B$ can be found efficiently using an index that is flexible enough to incorporate the similarity measure. An index on textual data can become large and relatively inefficient during lookup due to the size of the alphabet. This problem can be resolved in part by a transformation of the original data, which will be discussed in Sections 4.3.1 and 4.4. The sets of source records $A_1$ and target records $A_2$ can be equal, for example in a customer database where multiple entries for the same customer should be merged. The source and target sets can also originate from different sources, if for example patients from a hospital should be linked to patients from another hospital. However, the general record linkage problem is the same in both situations.

\(^1\)The range of $s$ is generally $[0,1]$ for real similarity measures and $\mathbb{N}^*$ for distance measures such as Levenshtein distance (which can be used as similarity measures by reversing the threshold condition). However, any range may be used in the general formulation of the record linkage problem.
4.3.1 Algorithm

In the current indexing approach records are represented as a set of characters. A binary vector (or bit vector) is constructed for each target record, where vector positions represent the presence of characters. Note that the order and frequency of characters is discarded, which allows for a more compact representation. As an example of vector construction consider an alphabet \{a, b, c\} and a record \textit{cacca}, for which mapping characters to vector positions in alphabetical order would result in the vector [1, 0, 1]. Using a bit vector, the alphabet size is reduced to 2, which permits an efficient implementation. A binary tree index is constructed on the vectors, where the presence of characters is queried in the nodes. The leaf nodes contain the actual records. The maximum number of nodes is bounded by \(2^{d+1}\) for tree depth \(d\), however the size of the tree is also linearly bounded by the number of records which quickly becomes less than the exponential bound for larger values of \(d\) (for example a tree of depth 32 for 4 million records with full branching at the top, which maximizes the number of nodes, runs out of records to create additional branches around depth 22 which leaves 10 levels of non-branching for an approximate total of \(2^{32} \cdot 2 + 10 \cdot 2^{22}\) = 48 million nodes, while the exponential maximum equals \(2^{32} \cdot 2 = 8.5\) billion nodes). The actual number of nodes is much smaller because of shared paths between records, which occur frequently because of the skewed distribution of values in many datasets (including historical civil certificate data).

In Figure 4.1 a small vector tree is presented containing a number of strings from the alphabet \{a, b, c\} (a more extensive example is discussed in Section 4.6.1). In this example the depth of the tree (defined by the length of the vector) is equal to the size of the alphabet, however in general any \(n\) to \(n\) mapping procedure can be used. The eight records in the example are located in three branches of the tree. Two other branches are terminated because no corresponding records exist, indicated by square nodes in Figure 4.1. If all records with a maximum distance of 1 need to be located for the query record \textit{cabca}, which is assigned the bit vector [1, 1, 1], then only the 101 and 111 branches are checked in order to find the matching record \textit{cacca}. The 100 branch can be discarded because the bit difference between this branch and the query vector 111 is larger than the distance threshold. Terminated branches (110 and 0) are discarded by default. The construction of the tree is outlined in Algorithm 4.1. Initially, the tree consists of a single root node. A bit vector for the first target record is constructed as
Chapter 4: Indexing edit distance

The algorithm checks whether an edge exists from the root node to another node \( n \) labeled with the value of the first bit. If this is the case, the algorithm proceeds with the second bit from node \( n \). Otherwise, a new node is added to the tree and the algorithm proceeds from that node with the second bit. This process is repeated for all

**Algorithm 4.1** Tree construction on target records.

```plaintext
for all target records \( t \) do

    \([b_0, b_1, \ldots, b_n]\) ← BITVECTOR\((t)\)

    current node \( c \) ← 0

    for \( i = 0 \) to \( n \) do

        if EdgeExists\((c, b_i, c')\) then

            \( c \) ← \( c' \)

        else

            AddEdge\((c, b_i, c_{new})\)

            \( c \) ← \( c_{new} \)

        if LastBit\((b_i)\) then

            AddRecord\((c, t)\)

```

![Figure 4.1: Example vector tree.](image)
Algorithm 4.2 TreeTraversal(node n, bit vector V, vector position p, error count $e_1, e_2$)

1. candidate set $C ← ∅$
2. if $p =$ last position in $V$ then
   3. return records in node $n$
4. $b ← V_p$
5. incorrect node $n_{err} ←$TreePath($n, 1 - b$)
6. $e'_1 ← e_1 + b$
7. $e'_2 ← e_2 + (1 - b)$
8. if $e'_1 ≤ \text{error}_{\text{max}}$ and $e'_2 ≤ \text{error}_{\text{max}}$ then
   9. $C ← C ∪ $TreeTraversal($n_{err}, V, p + 1, e'_1, e'_2$)
10. correct node $n_{corr} ←$TreePath($n, b$)
11. $C ← C ∪ $TreeTraversal($n_{corr}, V, p + 1, e_1, e_2$)
12. return $C$

bits, and the record is added to the final node. The bit vectors for the other target records are processed using the same procedure.

After construction of the index the tree can be used to compute record matches. For every source record, the tree is traversed using the recursive algorithm outlined in Algorithm 4.2. A bit vector is constructed for the source record and the tree is traversed, starting at the root node, by following the edges corresponding to the bit values of the vector. If this path exists, the records in the leaf node are added to the set of candidate matches. This procedure is implemented by the second recursive call to TreeTraversal in Algorithm 4.2: the search continues from the next node using bit $p + 1$. To incorporate approximate matches, the algorithm allows for a limited number of incorrect branch traversals. This is implemented by the first recursive call to TreeTraversal. The error count for deletion (bit $b$ changes from 1 to 0) or insertion ($b$ changes from 0 to 1) is increased, and if both error counts are still below the threshold (implemented by the constant error$_{\text{max}}$) the search continues following the incorrect branch. The recursive procedure collects all candidates from leaf nodes of the various branches. The original string representation of target records is compared to the original string representation of the source record using the string similarity measure $s$ to determine all actual matching records within the error threshold.
4.3.2 Similarity matches

The Levenshtein string edit distance, or Levenshtein distance, between string $p$ and string $q$ is defined as the minimal number of operations needed to transform $p$ into $q$. These operations can be insertion, deletion and substitution of characters (see Section 2.2 for a formal definition). For example, the string $abc$ can be transformed into $abdc$ by insertion of the character $d$, therefore the Levenshtein distance between the two strings is 1. The difference in bit vectors for two records is related to the Levenshtein distance, as stated in the following theorem:

**Theorem.** Let $p$ and $q$ be strings, $P$ and $Q$ binary vectors corresponding to these strings as constructed by the above method, and $e_1$, $e_2$ the number of vector positions with value 1 in $P$ and value 0 in $Q$, or value 0 in $P$ and value 1 in $Q$, respectively. Then, \( \max(e_1, e_2) \) is a lower bound for the Levenshtein distance between $p$ and $q$.

**Proof.** First, we consider $e_1$. For each vector position with value 1 in $P$ and value 0 in $Q$, by definition of the vector construction the character associated to that position is present in $p$ but not in $q$. Therefore, either a deletion or a substitution of the character has occurred. Similarly, for $e_2$ either an insertion or a substitution has occurred. A substitution can be counted twice, both in $e_1$ and in $e_2$. The minimum Levenshtein distance between $p$ and $q$ is equal to the sum of the deletions counted in $e_1$, the insertions counted in $e_2$, and the substitutions counted in both. This number is at least $\max(e_1, e_2)$, if all operations from $e_1$ are substitutions which are also counted in $e_2$ or vice versa. \( \square \)

An incorrect 0 edge or 1 edge corresponds to an increment in $e_1$ or $e_2$, respectively. Therefore, an algorithm that permits at most $n$ incorrect edge traversal steps of either type for a given source record finds all target records for which the Levenshtein distance is at most $n$. The tree traversal algorithm applied on the full set of source records finds all pairs of records within the distance threshold.

Note that the upper bound for the Levenshtein distance might be higher than the lower bound computed by the algorithm. Therefore, the actual Levenshtein distance needs to be computed for the candidate pairs found using the tree traversal algorithm.
4.4 Model parameters

Various parameters influence the memory and time requirements of the algorithm. The proof in Section 4.3.2 remains valid using different parameter settings.

4.4.1 Subvectors per record

A bit vector used as a bag of characters is position and frequency independent. In practise, this means that many highly dissimilar records will be represented using the same or similar bit vectors. This increases the processing time for comparing candidate target records, which becomes problematic for higher distance thresholds. However, this disadvantage can be limited by introducing some position dependence into the bit vector. Records used for linkage are likely to contain several fields, such as first name, last name, address, etc. A separate bit vector can be assigned to each of these fields, generating a number of subvectors for a record. The subvectors are concatenated to obtain a single vector which is used to build and traverse the search tree. This modification causes a direct comparison between corresponding fields without interference from other fields, which reduces the search space significantly.

4.4.2 Characters per node

In Section 4.3.1 a bit vector is constructed using a distinct vector position for every character in the alphabet. Using the lower case Latin alphabet with a few additional characters (such as spaces or hyphens), the vector would be around 30 positions long. If multiple vectors per record are used, as described above, the number of vector positions is multiplied by the number of vectors per record, which leads to a larger tree. To counter this effect, multiple characters can be assigned to the same vector position. This in turn causes more vector overlap for distinct strings, but with proper tuning the disadvantage of overlapping vectors is small compared to the efficiency gain of smaller vectors. Section 4.5 discusses a learning approach to determine an efficient character assignment.


4.4.3 Pruning

The search tree can be pruned while preserving complete coverage. Every node in the tree can be considered the root of a subtree, with an associated set of leaf nodes. The leaf nodes contain the actual records, often represented by separate fields as above. If records contain a first name, for example, then for every node in the tree a list can be compiled containing all first names in the associated leaf nodes. When a source record is traversing the tree, the first name in this record can be compared to the list of first names associated to each node. If the similarity measure rejects all names in the list, the subtree can be pruned without loss of coverage.

Obviously, comparing all records from a subtree to the current source record for pruning means that the linkage problem is moved from the subtree to the current node. However, the distribution properties of a single field can allow for an efficient comparison as opposed to comparison of full records. For person names in particular (which are present in many application domains), a relatively small number of names accounts for a large portion of the data. String distances between pairs of popular names and all other names in the data set are fast to precompute, and the results can be stored in the nodes of the search tree. Now, a constant time lookup can determine whether a source record containing a popular name has any possibility to match a target record in a given subtree. This saves a substantial amount of tree traversal as well as a reduction in edit distance computations.

Not all nodes are equally useful for pruning. Nodes high in the tree usually contain all popular names, therefore no pruning can be performed. Nodes down the tree contain only a few names, but there is not much to be pruned in these nodes because most of the tree traversal has already been performed. The name lookup action is constant, but with billions of tree traversal steps the performance of the algorithm is notably affected. Therefore, the most efficient way of pruning is to attempt the procedure only at a selection of nodes in the middle levels of the tree. The selection of nodes and the amount of popular names can be optimized for a given data set by measuring pruning performance on a sample of the data.
Chapter 4: Indexing edit distance

4.5 Vector assignment

The assignment of characters to vector positions is crucial for the performance of the algorithm, because it controls the distribution of records over the paths in the tree. The interplay between character distribution, tree size and tree traversal efficiency is complex, therefore finding the optimal assignment is non-trivial. A Simulated Annealing (SA) algorithm [72] is used to determine an efficient assignment from training data. The tree is constructed using 100,000 records, and matching is performed for 1,000 test records. As parameters a vector of $4 \cdot 8$ bits and a Levenshtein threshold of 3 are used (cf. Section 4.6). Candidate assignments are computed by changing the assigned position of a single character to an adjacent position, or by switching positions between two adjacent characters. The bit vectors and the tree are adjusted to the candidate assignment and the number of visited nodes is computed for the training data. The candidate assignment with the lowest number of visited nodes is selected as starting point for the next iteration. The tree adjustment and traversal are computationally intensive (both operations are linear in the number of records), however the SA optimization only needs to be performed once prior to the application of the algorithm to the full dataset. In the current experiments the SA algorithm has been terminated after several hours of running, however virtually all score improvement was achieved within the first hour.

The resulting assignment (see Table 4.1) is tested on the full dataset, and compared against a random assignment and a manual assignment. In the manual assignment the distribution of characters over vector positions is approximately even according to char-

<table>
<thead>
<tr>
<th>position</th>
<th>frequency based</th>
<th>simulated annealing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{x,q,z,y,comma,p,space,other}</td>
<td>{e,g}</td>
</tr>
<tr>
<td>1</td>
<td>{g,c,f,v,w}</td>
<td>{a,l,q,h,j,x,comma}</td>
</tr>
<tr>
<td>2</td>
<td>{b,u,l,j,h}</td>
<td>{r,p,v}</td>
</tr>
<tr>
<td>3</td>
<td>{k,d,n,m}</td>
<td>{n,space}</td>
</tr>
<tr>
<td>4</td>
<td>{s,t,o,r}</td>
<td>{i,u,w}</td>
</tr>
<tr>
<td>5</td>
<td>{i}</td>
<td>{d,f,c,m,z}</td>
</tr>
<tr>
<td>6</td>
<td>{a}</td>
<td>{t,s,y}</td>
</tr>
<tr>
<td>7</td>
<td>{e}</td>
<td>{o,b,k,other}</td>
</tr>
</tbody>
</table>

Table 4.1: Distribution of characters over bit vector positions.
acter frequency. Figure 4.2 shows the proportion of records for different amounts of visited nodes. For the SA assignment, a higher proportion of records is checked using relatively small amounts of visited nodes as compared to the other assignments. The mean number of visited nodes for the SA assignment shows a 6% improvement over the manual assignment. This result is obtained on the Genlias dataset, however similar improvements are expected for other datasets under the assumption that the type of character distribution in Genlias is common for natural language data.

Figure 4.2: Efficiency of character assignments. The mean of each assignment is indicated with a vertical line.

### 4.6 Experiment

To test the efficiency of the method, marriage certificates from the Dutch Genlias historical civil database are used to perform a record linkage task. A marriage certificate
contains three couples: the bride and groom, the parents of the bride and the parents of the groom. The linkage task used in the experiment is to link a certificate where a couple is listed as parents to a certificate where this couple is listed as bride and groom. Figure 4.3 gives an overview of the linkage process: marriage certificates (5.1 million couples) are transformed into bit vectors; all vectors are combined into a vector tree; single records traverse the tree to find candidate links and using the similarity measure real links are established. The model uses 4 subvectors per record (total vector length: 32). For each type of name (male first name, female first name, family name) the 100 most frequent names are used for pruning. The search is pruned at tree depths 16, 20 and 24. All experiments are performed on a 3.16 GHz dual core CPU with 6GB memory using 32-bit Linux. All programs are written in C++ and compiled with the -O2 optimization flag.

4.6.1 Example

To clarify the procedure, the steps of the algorithm are outlined for a single record. The vector for this record is 11010001110101001111101011111100. The full tree on the Genlias dataset contains 17.3 million nodes, a sample is shown in Figure 4.4 to illustrate various situations in tree traversal. The nodes contain the deletion and insertion counts $e_1$ and $e_2$, respectively. Edges are labeled + for correct traversals and – for incorrect traversals of the tree for the example vector. The maximum distance for this example is 2.

- **Branching.** The first vector position is 1. Starting from the root node, the right edge (labeled 1+) is correct. The algorithm will however also try the incorrect branch, assuming deletion of the character corresponding to the first vector position. The deletion error count $e_1$ is increased for this branch of the search.
- **Error count threshold.** In node 001, the correct edge is 1 (indicated by 1+). The algorithm is not allowed to branch to edge 0 (deletion), because this would exceed the threshold for error count $e_1$.

- **Branch termination.** In the node labeled ...001, the correct edge is 1. However, this branch does not exist in the tree, indicating that no records exist with the vector prefix associated to the current path. Therefore, this path is not examined.

![Diagram](image.png)

Figure 4.4: Example tree traversal.
further and the algorithm performs backtracking to the last point of choice (in this case node 0011).

- **Pruning.** Assume that the source record contains the popular name *Jansen* (English: Johnson). The node labeled . . . 111 is a pruning node, which contains a list of all popular names that could possibly match a name in one or more records in any leaf node below the current node. The name *Jansen* is not on the list for this node, therefore pruning is performed on this branch.

- **Leaf node termination.** The last situation depicted in Figure 4.4 occurs in the node labeled . . . 11100. This is a leaf node that contains a record which is added to the candidate set. Again, backtracking is performed to find the remaining candidates.

Using the standard dynamic programming algorithm Section 2.2, Levenshtein edit distance is computed between the source record and all records in the candidate set to determine actual matches.

In Figure 4.5 an example tree is provided for a set of 10 names. This example uses the character distribution from Table 4.1. The full tree used in the experiment described in this section consists of 32 levels, divided into four layers of 8 bits for each part of a record (i.e., the first name of the bridegroom, the last name of the bridegroom, the first name of the bride and the last name of the bride). The current example shows one layer of 8 bits. The bit vectors for the names in the example are provided in Table 4.2.

In Figure 4.5 various aspects of the behaviour of the algorithm are illustrated. The bit vector assignment does not consider character frequency or order, therefore *Aaltje* and *Altje* are assigned the same vector. This means that these two names are considered as candidates for comparison regardless of the threshold used. In this case the comparison is justified, given that the Levenshtein distance between the two names is 1 and the names are actual onomastical variants. In general this aspect of the algorithm captures the heuristic that many name variants are cases of character duplication or character shift, such as *Gretien–Gertien*. Only for uncommon cases such as anagrams (e.g.,

---

2Character shift is not very common for onomastical name variants, however this phenomenon is frequent if typing errors are taken into account.
Chapter 4: Indexing edit distance

<table>
<thead>
<tr>
<th>name</th>
<th>bit vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aafje</td>
<td>11000100</td>
</tr>
<tr>
<td>Aafien</td>
<td>11011100</td>
</tr>
<tr>
<td>Aalie</td>
<td>11001000</td>
</tr>
<tr>
<td>Aaltgijn</td>
<td>11011010</td>
</tr>
<tr>
<td>Aaltijn</td>
<td>01011010</td>
</tr>
<tr>
<td>Aaltje</td>
<td>11000010</td>
</tr>
<tr>
<td>Altje</td>
<td>11000010</td>
</tr>
<tr>
<td>Andrea</td>
<td>11110100</td>
</tr>
<tr>
<td>Lisa</td>
<td>01001010</td>
</tr>
<tr>
<td>Liza</td>
<td>01001100</td>
</tr>
</tbody>
</table>

Table 4.2: Bit vectors for an example name set.

Arnold–Roland) the resulting behaviour is undesired, in which case the names are considered for comparison and subsequently rejected in the actual distance computation.

The paths in the tree for the name Lisa, represented with the bit vector 01001010, can be traced for various values of the distance threshold. Starting at root node 0 the algorithm checks if the name contains e or g, i.e., the first bit of the bit vector is examined. The value is 0, but for a threshold of 1 the algorithm continues in both nodes on the second level, i.e., nodes 1 and 2. In the branch to node 2 the insertion count \( e_2 \) is increased, corresponding to an insertion of either e or g or both. After processing of the second bit the search proceeds to nodes 4 and 6 without increasing the error counts of the branches. Branching to nodes 3 and 5 is also permitted (which would increase the deletion count \( e_1 \) in these nodes), but in the example graph both nodes are empty and are not considered further. On the next level nodes 7 and 9 can be reached without an increase in error counts, however node 10 is not available for a threshold of 1 because this path would increase the insertion count \( e_2 \) to 2 (corresponding to an insertion of r, p or v). This means that the subtree below node 10 can be skipped for this threshold, removing in the case the name Andrea for consideration as variant of Lisa. For a threshold of 2 this subtree would be skipped in the next step, as also the traversal from node 10 to node 16 would increase the insertion count (corresponding to an insertion of n). This is the desired behaviour: a name which is very dissimilar (the edit distance between Lisa and Andrea is 5) is no longer considered after just 3 tree traversal steps.
Figure 4.5: Tree index for an example name set.
Consider node 18, in which both error counts are 0 for the name Lisa as this node is on the path to leaf node 57 which contains this name. Given a threshold of 1, the algorithm can continue to node 28 which increases the insertion count $e_2$ to 1 and subsequently node 41 can be reached which increases the deletion count $e_1$ to 1. Finally, the algorithm proceeds to node 59 without increasing the error counts and Liza is found as a candidate for comparison with Lisa. This situation shows that similar names with different characters (as opposed to similar names with equal characters, as in node 63) are discovered by the algorithm.

However, the grouping of characters can lead to the comparison of dissimilar names as well. In this example the name Aaltijn (node 61) can be reached when searching for the name Lisa with only a single increase in the error count in node 7, while the edit distance is 5. One reason for the overlap in bit vectors is that the two names share three characters (l,i,a) but two of the three remaining differences are grouped in the same nodes, which means that the algorithm cannot differentiate these characters during tree traversal (s and t are grouped in nodes 27/28 and j is grouped with the shared characters a and l in node 1). Only the difference in the character n leads to different branches in node 7.

The same tree traversal procedure can be used for names which are not part of the tree. If for example the name Aaltjen is encountered, an exact search would fail in node 23, but with a threshold of 1 nodes 61, 63 and 69 can be reached which results in the discovery of the only variant Aaltje for this threshold.

In Sections 4.6.2 and 4.7 the results of the algorithm on the Genlias dataset are discussed. These results indicate that the amount of unnecessary comparisons is relatively small, which means that the algorithm can efficiently be applied in this case.

### 4.6.2 Results for Levenshtein distance

The algorithm processes 1648 records per second on average for a maximum Levenshtein distance of 2 (see Table 4.3), which is less than 1 hour for 5 million records. This is suitable for real-time systems (cf. [129, 31]). Blocking methods are based on heuristics, therefore the coverage of a blocking method depends on the quality of the heuristic and the distribution of variation in the data. The current indexing method, in contrast, provides full coverage for the Levenshtein distance on any dataset.
Note that Table 4.3 reports on the number of matches according to a threshold on the Levenshtein edit distance, which is most likely different from the number of actual links between records. The Genlias dataset does not provide any verified links, therefore a complete verification of the matching results is hard to perform. However, the aim of this chapter is not to assess the quality of the Levenshtein edit distance, but to provide a practical algorithm for complete coverage without the disadvantages of blocking.

### 4.7 Comparison to existing methods

Various methods in information retrieval are designed for complete coverage, using techniques such as filtering and sampling (see Section 4.2). Unlike sampling, filtering can be useful in record linkage as well. If, for example, a record consists of four fields and the maximum allowed Levenshtein distance is three, then at least one of the fields will match without error for all matching pairs. Using filtering, fast exact matching can be used to generate the candidate set for a record. However, in case of person names the candidate set can be as much as 10% of all records for high-frequent names. Efficiency can be improved by computing distances in a trie structure (cf. [90]), using a single distance computation for paths in the trie that are shared between records. Note that, instead of a suffix tree, a normal trie is sufficient for this method because all matches have to start at the beginning of the record. An implementation of filtering using a trie takes around 0.2 seconds to find all matches for a single record (see Table 4.4), which is acceptable for a query system but not for batch processing. The publicly available implementation of Agrep for Linux checks a record in around 1.5 seconds. It should be noted, however, that

<table>
<thead>
<tr>
<th>Distance threshold</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>3 min.</td>
<td>3 min.</td>
</tr>
<tr>
<td>Records/sec</td>
<td>1417</td>
<td>136</td>
</tr>
<tr>
<td>with pruning</td>
<td>1648</td>
<td>200</td>
</tr>
<tr>
<td>Matches</td>
<td>895,144</td>
<td>1,217,459</td>
</tr>
<tr>
<td>Matches (blocking)</td>
<td>848,463</td>
<td>1,031,097</td>
</tr>
<tr>
<td>Improvement</td>
<td>5%</td>
<td>18%</td>
</tr>
</tbody>
</table>

Table 4.3: Experimental results on the Genlias data set.
that these methods are intended for short queries on longer texts, which is not the case in record linkage. Conversely, the text querying problem cannot be solved efficiently with the bit vector method.

4.7.1 Comparison to blocking methods

As mentioned in Section 4.1, a reduction in computation time is often accomplished by blocking or pre-grouping of possible matches. For example, only names starting with the same character (initial character blocking) can be considered, or only records from a certain time period or geographical location. The bit vector method can also be applied to a blocked data set, which results in a double efficiency increment. Using the initial character blocking results in about 23 candidates on average for each record (note that this is an average, some blocks are much larger than 23 records). Using the bit vector method, the number of candidates can be reduced to around 2 or 3 on average. Of course extra time is needed for tree traversal, but the total computation time is still over 90% faster compared to blocking only, resulting in a computation time of around 5 minutes for the full set.

This method is compared to a recent approach in record linkage, Improved suffix array blocking [129]. Suffix array blocking is a simple yet very effective method for blocking which maintains a high level of recall. The method extracts for every target record all suffixes with a minimum length \( \ell \) and sorts the suffixes in alphabetical order. In the linkage step, all suffixes with a minimum length \( \ell \) are extracted from a source record. The method selects all suffixes from the index that are equal to a source record suffix, or alphabetically adjacent to a source record suffix. If a target record and a source record are different, but the difference is not in the last \( \ell \) characters, there will be an exact suffix match for this pair of records. If the difference is located in the last \( \ell \) characters, then it is very likely that two larger suffixes are alphabetically adjacent. Take for example the records John Smith and John Snith (example from [129]). With \( \ell = 4 \), no suffixes are shared (the shortest suffixes are mith and nith). However, it is likely that for example ohnSmith and ohnSnith are alphabetically adjacent in the suffix array. An additional parameter mbs (maximum block size) removes a suffix from the suffix array if more than mbs records contain this suffix, to limit the number of record comparisons.

The bit vector method with blocking is compared to our own implementation of
Table 4.4: Method performance for Levenshtein distance $\leq 3$ on 1.5 million records.

<table>
<thead>
<tr>
<th>Blocking</th>
<th>Bit vector</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matches</td>
<td>626,205</td>
<td>659,068</td>
</tr>
<tr>
<td>Indexing</td>
<td>83 sec.</td>
<td>207 sec.</td>
</tr>
<tr>
<td>Records/sec</td>
<td>45,838</td>
<td>3,137</td>
</tr>
<tr>
<td>Memory</td>
<td>2.28 GB</td>
<td>2.91 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complete</th>
<th>Bit vector</th>
<th>Filtering</th>
<th>Agrep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records/sec</td>
<td>200</td>
<td>5.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>

suffix array blocking, using $\ell = 4$ and $mbs = 12$ (values based on the results of [129]). The number of unique suffixes for the Genlias data set is around 16 times as large as the number of records, therefore memory consumption is a problem for this method. To perform the comparison, an index is built on the first 1.5 million records for both methods. The results are outlined in Table 4.4. Suffix array blocking finds more matches, although the difference is relatively small. The bit vector method with blocking, on the other hand, is faster in indexing and matching, and consumes less memory.

### 4.8 Extension to Jaro distance

The method described in this chapter is developed for use with Levenshtein edit distance, but with a few modifications application to other similarity measures is possible. As an example this section discusses application to Jaro similarity (see Section 2.2 for a definition).

For any threshold $t$ of the Jaro distance, the number of shared characters needed to obtain a Jaro distance of at least $t$ can be computed. This number depends on the length of both strings, because the proportion of shared characters is used (unlike Levenshtein distance, where the absolute number of edit operations is measured). The minimum number of shared characters provides an upper bound on the Jaro distance, because the score can be adjusted downwards due to transpositions or shared characters outside of the window.

The tree structure as described for Levenshtein distance can be used for Jaro distance with a few modifications. For Jaro distance, the maximum number of non-shared
characters can be used as a threshold for incorrect branch traversal. This threshold is dependent on the length of the source records. Because also the length of the target records (which are in the tree) is important, the tree can be divided into subtrees according to record length. Within a subtree, the incorrect branch threshold is proportional to the length of the records, which is efficient for shorter records. Between subtrees, the threshold is usually somewhat larger but only adjacent subtrees need to be checked.

4.8.1 Results for Jaro distance

To give an idea of the performance of the method on the Jaro distance measure, a number of experiments have been carried out using 10,000 distinct random person names from the Genlias dataset. For a minimum Jaro distance of 0.8, which is already quite permissive, the number of candidate records is reduced by 94% compared to full pairwise computation. A minimum distance of 0.85 amounts to a reduction of 98%. The nodes in the tree can be sorted according to the frequency of the corresponding characters in the data set. If low-frequent characters are queried first, then the lower part of the tree has a higher density. Using the traversal algorithm on this tree leads to a time reduction of around 65% compared to pairwise computation. Further optimizations on the arrangement of parts of the tree, grouping characters or pruning might be possible. As with Levenshtein edit distance, the method is guaranteed to find all pairs of records within the Jaro threshold.

4.9 Discussion and further research

In this chapter a method is described to transform a set of records into a binary tree in order to find pairs of records within a similarity threshold. The method provides complete coverage for Levenshtein distance within a practical time limit. The algorithm uses an informed strategy for approximate matching: a possible bit vector variation is only developed if it is present in the tree (and, therefore, if corresponding records exist in the data set). This approach is more efficient than a simple generate-and-test procedure for bit vectors using a hash table. The upper bound on the number of vectors that needs to be checked is $k^d$ for vector length $k$ and maximum distance $d$. In practise, this number
drops to only a small percentage of the possible vectors for increasing values of $d$ due to the sparseness of points in multidimensional space. However, for generate-and-test all hash functions that might select error-free overlapping substrings need to be checked. For efficiency reasons generally a subset of hash functions is selected with a high probability of finding record matches (cf. min-wise hashing, e.g., [24]) at the expense of complete coverage, which is preserved in the current approach.

An extension to Jaro distance is discussed, as well as a combination of the method with blocking techniques. The thresholds used in the experiments are already useful for applications, however scaling the method to larger distances is problematic because the part of the search tree that must be visited increases. The situation can be improved with domain-specific optimizations such as the pruning procedure as described in Section 4.4.3.

Jaro distance can be used with this method because the number of non-shared characters between strings is directly linked to the upper bound of the distance score. This monotonicity property also holds for Levenshtein distance. If the threshold of non-shared characters is exceeded, then the presence or absence of other characters will not enable the distance to reach the threshold. The Winkler modification does not have this property, because the Jaro-Winkler score is adjusted upwards for shared prefixes. Therefore, any number of non-shared characters can theoretically be compensated by adding prefix characters. In practice, the increase in score is limited, because the strings have a finite length. Additionally, Winkler limits the application of the prefix adjustment in his implementation of the similarity measure. Therefore it is possible to define non-shared character thresholds for strings of given lengths also for Jaro-Winkler distance, but the upper bound will be less useful in this case.

If it is sufficient to find a single match for each record, located anywhere in the data set (i.e., without using blocking), tree traversal can first try promising paths and stop searching once a match has been found. This can lead to a large efficiency improvement.

Searching promising paths only can be compared to Locality Sensitive Hashing [46], a technique in which two similar records are assigned a value by a number of hash functions such that the probability of a collision for at least one hash function is high. The vector tree approach can be thought of as generating a new very small family of hash functions (tree paths) for each data point which contains only functions relevant for
this point. Additionally, the structure of the tree allows for efficient switching between different tree paths and memory-efficient storage of the index. The current data set, for example, needs to store around 14 integer values for each 32-dimensional data point due to shared paths between data points.

In future work, the simulated annealing algorithm generating the distribution of characters over vector positions can be investigated in more detail. Different settings in vector size or subvectors per record can be used to attempt further efficiency improvements. Extensions to various other similarity measures could also be investigated.