Internal Report 2012–17 August 2012

Universiteit Leiden
Opleiding Informatica

The Assignment Problem

on

Sets of Strings

Thijs van Ommen

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction

In this thesis, we will develop a tool that compares two text files to get an overview of their differences, in a way similar to what the `diff` tool does. In some scenarios, the usefulness of `diff` may be limited by the fact that the only changes it detects are insertions and deletions of lines. If a line was moved to a different position in the file, then `diff` will consider that as one line being deleted and another being inserted elsewhere, without realizing that these are the same line.

To formalize the problem, we treat each file as a (multi)set of strings, namely its lines. We want to assign each element in one set to a unique element in the other set, in such a way that pairs of elements assigned to each other are as alike as possible. To quantify the similarity of a pair of strings, we need to choose a distance measure on strings. Then the problem is to find an assignment that minimizes the total distance between the pairs of strings.

An important property of this formalization is that by treating the files as sets rather than sequences of lines, we completely ignore their original ordering. This puts us at the other end of the spectrum from `diff`, which strictly adheres to the ordering of the lines. Hence it depends on the application which of the two tools is more appropriate for the comparison of two files. For instance, when comparing two source code files, it is possible that some lines were moved to different places in the file, but in general the ordering of lines within the file is too important to ignore it altogether, so that `diff` is still the better choice.

One example where our problem has practical application is for two files representing snapshots of data from a database, but sorted according to different criteria. Then if we have no good way of reconstructing the sorting criteria, or if too many changes occurred in the database column on which the files could have been sorted, `diff` is of limited use. The tool we intend to develop in this thesis will help us to get an overview of changes in the database based on just these two files.

The straightforward solution to the described problem would be to first compute a table of distances from each element of the first set to each element of the second set, and then run a standard algorithm for the assignment problem (described in Section 3) on this table. While this would yield the desired answer, computing all these distances is likely to be prohibitively expensive in terms of computing power. Therefore, most of this thesis will be concerned with the computational efficiency of our solution.

We will find that the resulting program can be a great help in the right situation: for certain pairs of files that `diff` has trouble with, our program finds the desired assignment of lines, and does so in very little time. However, its applicability is restricted to those cases where the order of lines within the files is irrelevant.

This thesis is structured as follows. In Sections 2 and 3, we study related work, on approximate string matching and the assignment problem respectively. We describe our optimized algorithm in Section 4. Section 5 shows the performance of this algorithm in practice, and Section 6 concludes.

The present work was written as a bachelor thesis in computer science, at Leiden University (LIACS), under the supervision of dr. W.A. Kosters.

2 Approximate string matching

In many applications in which strings are involved, exact comparison of strings is not sufficient and some notion of distance between strings is required. Some examples of such applications are spelling correction, information retrieval (where either the query or the text being searched may contain spelling mistakes), file comparison (where we usually wish to find what lines are inserted or deleted), and computational biology (where various questions may be asked of very long strings of DNA). In this section, we look at several such applications, and begin by looking at the various distance measures that have been used in them.

2.1 String distance measures

We will review several classes of string distance measures. We do not intend to be exhaustive, but want to show a glimpse of their variety, which reflects the different applications for which these measures were designed.

2.1.1 Edit distance and variants

A basic distance measure is the *edit distance* (also called *Levenshtein distance*, see [WM91]). It counts the smallest number of edit operations that will convert one string to another, where the editing operations are inserting a character, deleting a character, and substituting one character for another. This distance measure is very appropriate if we are interested in dealing with typographic or spelling mistakes, as it tends to assign small distances only to those pairs of strings that match with few such errors.

Many other string distance measures can be viewed as generalizations of the edit distance. One obvious way to generalize it is by assigning *nonuniform costs* to the different editing operations [CGK06]. For example, we could make insertions and deletions three times more expensive than substitutions, or we could assign different costs to substitutions based on how close the corresponding characters are on a keyboard.

A further variation of the edit distance that is commonly used in computational biology employs an *affine gap penalty*. The alignment of two sequences of nucleotides or amino acids can be visualized by printing each

sequence on its own line, and leaving space in the sequences so that matching elements are above each other. Then each series of insertions or deletions shows up as a *gap* in one of the two sequences. Normal edit distance can be seen as assigning linear costs to gaps: the cost of each gap is some constant times the size of the gap. As observed in [NW70], other choices are possible. In particular, an affine gap penalty results if the cost for starting a gap (the first insertion or deletion) is different from the cost of extending a gap (each subsequent insertion or deletion).

We can also modify the notion of edit distance by disallowing substitutions (or equivalently, assigning them a cost of two or greater). This is the notion of distance used by `diff`. (In `diff`, the sequence of lines in a file is treated as the characters of the string to be compared.) Then finding the distance between two files corresponds to finding the longest common subsequence of lines in those files. Therefore this distance is known as the *longest common subsequence distance* [Na01]. In principle, `diff` wants to minimize the numbers of insertions and deletions, but by default, it uses a heuristic and may produce suboptimal output if the input is complex enough [GNU11].

It is obvious that all variations of the edit distance described above satisfy the mathematical properties of a metric:

- they assign nonnegative distances to any pair of strings and zero distance only to equal strings;
- they are symmetric (if, in case of the generalized edit distance, insertions and deletions have the same (nonzero) cost and the matrix of substitution costs is symmetric);
- the triangle inequality holds (also for the generalized edit distance): if we know sequences of edit operations that transform string x into y and y into z, then those two sequences combined will transform x into z, so that the distance between x and z can be no larger than the sum of the other two distances.

2.1.2 Token-based distances

Some distance measures are based on treating the strings as (multi)sets of tokens. These tokens may be the words occurring in a string (or possibly just their stems) or the set of its q-grams (all its substrings of length q, see [CGK06]). The former is used in [HK06] to measure distances and similarities between documents, while the latter occurs in contexts dealing with shorter strings.

A simple way to measure the similarity of two sets S_1 and S_2 is the Jaccard coefficient, which is given by

$$\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}.$$

It is 1 if $S_1 = S_2$, and 0 if they share no common elements. Note that for this measure and the following, larger values correspond to more similar strings.

If the frequencies with which all tokens appear in each of two strings are given as two vectors v_1 and v_2, another way to express the similarity between those two strings is the *cosine similarity*

$$\frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|}, \tag{2.1}$$

where the numerator contains the vector dot product and the denominator the Euclidean norm. The name derives from the fact that this equals the cosine of the angle between the two vectors. It is again between 0 and 1: 0 if the vectors are perpendicular (which must again mean that the strings share no common tokens), 1 if they are colinear (in which case all tokens occur with the same relative frequencies).

In some applications, it is useful to adjust these measures in the following way. Given a set of strings (seen as documents in a corpus), we weigh each token occurring in those strings by the inverse of its "document frequency": the number of documents in the corpus in which this token appears. (The actual formula for the inverse document frequency given in [HK06] applies a logarithm to this inverse frequency). Then the $|\cdot|$s in the Jaccard coefficient become sums of such weights, and the elements in the vectors used to define cosine similarity become term frequency times inverse document frequency (*TF-IDF*). This has the result of ignoring stop words that occur in all documents (probably in large numbers) but that do not tell us anything about the topic of the document, while emphasizing terms that are specific to small sets of documents.

2.1.3 Block edit distances

A possible shortcoming of the edit distance and its variants is that they fail to recognize the similarity between the pair of strings `hello world` and `world hello`: they can only recognize one of the two words as occurring in both strings, and must treat the other as being deleted from one string and inserted into the other. Token-based distances are on the other side of the spectrum: they depend very little (or not at all, if the tokens do not overlap) on the order in which parts of a string are arranged, and may assign very large similarity to strings that are very different but happen to have similar sets of tokens. Block edit distances can be seen as a halfway point on this spectrum. They are best explained as edit distances that allow an operation that either moves or copies substrings.

The first article we could find that attempts to define such a distance measure is [He78], which treats the problem of creating edit scripts (like `diff` does). He allows a move operator in addition to deletion and insertion operators. The (implicit) distance he tries to minimize assigns unit cost to

deleting and inserting lines, while moving lines is free. This can easily be seen to be a metric by regarding it as the Manhattan (or city block) distance on the frequency vectors as used in (2.1). His algorithm prefers to move contiguous elements together, but this preference is not made precise, so that the distance behaves more like a token-based one. The exact preferences of the algorithm are hard to make precise because the algorithm he describes will fail to find matching lines and count them as insertions and deletions, unless there are sufficiently many lines that occur exactly once in each file. Nevertheless, the article was a good starting point for the investigation of block edit distances.

A slightly different notion of distance is given by [Ti84]: he allows a new string to be constructed from left to right as a sequence of insertions and block copies from an original string. The problem he discusses is to minimize the number of such block copy operations (which he calls "moves", even though the original string is not changed by this operation), given that only characters that did not occur in the original string may be inserted. This can be seen to correspond to minimizing the cost of a sequence of editing operations where insertions cost M per character and copies cost 1 per block, for some sufficiently large constant M. After these operations, the original string is always deleted at no cost and the remainder must equal the target string. Efficient algorithms to find the minimizing sequence of edit operations are also given. Because unlike in the previous distance, block copies come at a nonzero cost, this distance penalizes pairs of strings that share symbols but have them in a different order. However, it is not a metric. In particular, it is clearly asymmetric, and it violates the triangle inequality, for example for

$$d(\mathtt{a}, \mathtt{ab}) = M + 1 \qquad d(\mathtt{ab}, \mathtt{abab}) = 2 \qquad d(\mathtt{a}, \mathtt{abab}) = 2M + 2. \tag{2.2}$$

In [EH88], the authors define $\mathrm{diff}(x, y)$ as the smallest number of characters of the string x that must be marked so that each intervening substring of x is also a substring of y. This leads to something very closely resembling the previous distance for constructing x from y, though $\mathrm{diff}(x, y)$ will sometimes force characters to be inserted into x just as a way of marking the boundary between two copy operations. The authors proceed to show that $\lg((\mathrm{diff}(x, y) + 1)(\mathrm{diff}(y, x) + 1))$ is a metric. It is clear how this transformation serves to get symmetry, and the logarithm is necessary to maintain the triangle inequality in examples such as (2.2), where the relation between the distances is multiplicative rather than additive.

Finally, [LT97] considers the following problem: Given a pair of strings, we want to associate substrings of one with substrings of the other so that the total distance of these pairs of substrings is minimized. The distances between substrings may be given by an arbitrary function, which may even take negative values. The authors discuss several variations with different requirements on the substrings, the most natural of which seems to be the one

where the substrings of one string are required to be a disjoint cover, while there are no requirements on the other string's substrings. This resembles creating the first string from the second by a sequence of copy operations, except that the copies may then be further edited. The authors show that this problem is NP-complete if requirements of disjointness or coveredness are made on both strings. For the other cases, they give polynomial algorithms, though these are significantly less efficient than the algorithms occurring in the other papers mentioned in this section.

2.1.4 Specialized distances

Some distance functions are not intended for comparing general strings, but for strings from specific domains. We mention Soundex, described in [Kn98], which is a method for encoding surnames by mapping them to a sequence of a letter and three digits. This mapping throws away much of the information in the original string, in such a way that names that sound similar are mapped to the same code. This defines an equivalence relation on strings. It can be seen as a distance function that assigns distance 0 to strings in the same equivalence class, and distance 1 otherwise. Hence it satisfies the properties of a metric except one: it may assign distance 0 to unequal strings. This makes it a pseudometric.

2.1.5 Compression-based distances

According to the Minimum Description Length (MDL) principle [Gr07], it is possible to draw various kinds of conclusions about arbitrary data by seeing how well this data can be compressed. The motivation is that if we can compress data well, then we must have seen regularity in the data, which corresponds to having "learned" from the data. The ideal measure of compressibility is *Kolmogorov complexity*, which is the length of the shortest computer program that reproduces the data. However, this theoretical notion has several problems when we try to apply it in practice: the exact length depends on the programming language we use (though this difference is a constant for Turing-complete languages), and the problem of finding (the length of) the shortest program that produces some output is uncomputable. Therefore, to apply this idea in practice we must restrict ourselves to less general schemes of encoding data. We want such an encoding scheme to be good at compressing the types of regularity we expect to find in our data.

Compressibility corresponds to a distance measure between two strings in the following intuitive sense. Given some function L that assigns to a string the length of its encoding (which we will measure in bits) and two strings x and y, $L(y|x) := L(xy) - L(x)$ is the code length we need to encode y after having seen x. If x and y are similar (for example, if they share long common substrings), then xy can be encoded by first encoding x and then

encoding y with the help of x. Then $L(y|x)$ will be small. However, if x and y share little similarity, then encoding xy will be about as expensive as encoding x and y separately, so that $L(y|x)$ will be large (close to $L(y)$).

The expression $L(y|x)$ is not a metric. It is clearly asymmetric: for $x = \epsilon$ (the empty string) and $y =$ `this is a long string`, $L(x|y)$ will be small but $L(y|x)$ large. If a symmetric measure is desired, this can be accomplished by taking $d(x, y) = \max\{L(y|x), L(x|y)\}$ [Ci07]. The other properties of a metric do hold approximately for Kolmogorov complexity, and thus might also be hoped for in a practical compressor. In [Ci07], the conditions on a compressor such that d is approximately a metric are formalized in the definition of *normal compressor*. Finally, [Ci07] suggests normalizing the distance through dividing $d(x, y)$ by $\max\{L(x), L(y)\}$. The result is still a metric, and assigns distance approximately 1 to pairs of strings which are completely dissimilar.

We can now choose a practical string compression algorithm with the properties referred to above and use it to define a distance function. One popular family of compressors with these properties is the Lempel-Ziv family. Roughly speaking, such a compressor encodes a string s as a sequence of words from its dictionary, which initially contains all single characters in the alphabet, but which is updated with substrings from the part of s already encoded. The first member of this family was introduced in [ZL77] and is referred to as `LZ77`. The main difference between `LZ77` and its successors is that the dictionary in `LZ77` contains all substrings of s (up to some maximum length and with starting point not too far back) of which the first character has already been encoded, whereas subsequent algorithms only contain a subset of those substrings.

The encoded output produced by `LZ77` consists of an alternating sequence of pointers to previous substrings and literal characters. These pointers are encoded with two uniform codes: one over all possible starting positions, and one over all possible lengths. Then the triplet of two pointers and a literal character takes $\lceil \lg w \rceil + \lceil \lg L_s \rceil + \lceil \lg |\Sigma| \rceil$ bits, where w is the size of the sliding window in which the original substring must start, L_s is the maximum length, and Σ is the source alphabet.

2.2 Approximate string matching algorithms

The problem of approximate string matching is to find a substring in a large text which is within some specified distance threshold k of a small pattern string [Na01] according to a chosen distance measure. There are two major variations of this problem: the online version (where no preprocessing of the text is allowed), and the version where the text can first be preprocessed to create an index on it. We start with some online algorithms.

2.2.1 Online algorithms

For biological sequences, [NW70] described the dynamic programming solution to compute several variants of the edit distance between two strings. The cell at position (i, j) of the dynamic programming table represents the distance between prefixes of lengths i and j of the two strings respectively. Then the value of each cell can be computed from the cells $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$ with a simple recurrence. The algorithm in [SW81] is a small adaptation of this to find approximately matching substrings rather than the edit distance between the two full strings. In fact, the algorithm goes further than approximate string matching: given two long sequences, it can find a substring in each, such that these substrings have a good "similarity score" (which is like an edit distance, except that matching characters improve the score; otherwise two empty substrings would be the optimal solution).

A very efficient solution using a different approach is the *bitap* algorithm by [WM91]. Its base form finds all strings in the text which are within edit distance k of the pattern, but many extensions are described, such as using the edit distances with nonuniform costs (as long as the costs are small integers), or using a regular expression rather than a single string as the pattern. The algorithm is based on a state machine which tracks how some substring of the text can be matched against the pattern. It gets a lot of its speed from working with bitwise operations, which essentially allow it to track many match states in parallel as long as the pattern's length is small compared to the machine's word length. It is implemented in the UNIX tool `agrep`.

In [Uk92], two algorithms are described for approximate string matching using different distance functions: a distance based on q-grams, and [Ti84]'s block edit distance.

2.2.2 Algorithms using preprocessing

We discuss two types of indices. The first is based on q-grams. It is useful even if the distance function we are interested in is not directly based on q-grams: many algorithms can be sped up by first finding occurrences of the pattern's q-grams in the text, then ignoring parts of the text containing no matching q-grams, and running a different algorithm to take a closer look at the parts that do [Na01]. Depending on q, the size of the pattern, and the allowed distance, such an approach can be made precise in ways that guarantee that no approximate matches will be missed.

Because we want to compare q-grams efficiently, we usually hash them. A particularly useful type of hash function in many applications is a *rolling hash*. Such a hash function is also used in [KR87]'s algorithm for exact string matching. Given the hash of some substring, we can efficiently compute the

hash of a substring of the same length starting one character later.

Another category of indices is based on suffix trees and variants [Gu97]. Such data structures store all suffixes of the text, and can be constructed in linear time and space [McC76]. The ability to quickly locate any substring is obviously valuable to an approximate string matching algorithm, though they tend to be more expensive than q-gram based indices because of larger constant factors.

Two concrete examples of approximate string matching algorithms with edit distances, one using a suffix automaton and the other using an index of q-grams, are given in [JU91].

2.3 Other problems involving approximate string matching

Several applications of string distance measures have already been mentioned. Here we mention a few more that are of interest, and discuss how they relate to our assignment problem.

2.3.1 File comparison

In the problem of file comparison, we are not primarily interested in the distance between two files, but rather in the similarities and differences determining this distance. If the two files were similar, these differences can be described in a *patch* file that is significantly smaller than the two files being compared. This leads to a secondary application of file comparison tools: if we also have a tool that, given the first input file and the patch, reconstructs the second file, then these patches can be used to efficiently store many versions of one file in a *versioning system*.

We already mentioned the well-known `diff` utility in our discussion of the longest common subsequence distance. Its implementation (at least, of the GNU version, see [GNU11]) is based on the algorithm described in [Me86] and [MM85].

File comparison utilities could conceivably become more powerful if they considered block edit distances: they could capture more similarities in the files (where the user of `diff` will have to find such similarities by himself), and they might make patches even more efficient. This was indeed the motivation behind most of the articles referred to in Section 2.1.3.

The problem we want to solve in this thesis is also very similar, the only difference being that our primary interest is not the ability of reconstructing the second file in its original order, but only modulo a permutation of its lines. But with the trivial modification of adding the original ordering of the lines to the difference file which (as required in our problem setting) already describes the assignment of lines from one file to the other and the changes made to those lines, such a file would constitute a patch.

In this light, we take a closer look at two of the block edit distances and how they compare to our problem. First, [He78] also wants to find an assignment between the lines of the two files. However, the author only assigns two lines to each other if they are identical (like `diff`), and the solution found by the described algorithm may be arbitrarily far removed from the optimal solution. On the other hand, [LT97] allows the file to be split up into substrings in arbitrary ways rather than always splitting it at every line end, and allows those substrings to be compared using an arbitrary distance measure. However, the conclusion is that if we require for both files that their substrings form a disjoint cover (which would make it most similar to our assignment problem), then the problem becomes NP-complete. This gives us hope that our problem, being halfway in complexity between these two problems, will admit efficient computation, at least for certain choices of the string distance function used to compare lines.

A final application that should be mentioned in this section is `rsync` [Tr99], another popular tool in UNIX environments. The part of its functionality that we are interested in is creating a patch, but with the caveat that the two files being compared are separated by a low-bandwidth high-latency connection. The crucial ingredient of the algorithm is a rolling hash, as used in [KR87].

2.3.2 Record linkage

A common operation when working with relational databases is the *join*. This is a subset of the Cartesian product of two tables, filtered to contain only those pairs of records which match in some way; for example, referring to the same customer or product. These are usually recognized by comparing *keys* in the database. *Record linkage* concerns itself with such tasks when no key is available but we still need to find all pairs of records that might refer to the same entity, maybe because data from independent sources need to be integrated. This may be accomplished by a *similarity join*, where all pairs of records that are "close" in some sense are returned [CGK06]. Measuring the distance between two records often involves measuring string distances. To get good results, it is important to choose the distance functions properly. Specialized distances like Soundex play a large role here.

It is also useful to match a database with itself in this way, in order to find records referring to the same entity but not recognized as such because of a misspelling in a name or a customer changing address.

The problem of record linkage is similar to our assignment problem, though in record linkage, one typically does not assume that as many records from each table as possible must be matched to a record in the other table. A further difference is that we want an algorithm that returns at most one match per record, while a similarity join may return more than one as long as all matches are below the chosen threshold.

2.3.3 Inference

Based on the Minimum Description Length principle, [Ci07] explored various types of statistical inference tasks that can be done using compression-based distances. These tasks included classification (for example, recognizing whether an email message is spam) and clustering (for example, grouping musical pieces together based on their similarity, so that pieces by the same composer end up being grouped together). That such tasks can be performed in this way demonstrates the usefulness of compression-based distances. The similarity of these applications to ours suggests that such a distance measure would also be useful for our purposes.

3 The assignment problem and the Hungarian algorithm

The assignment problem is an optimization problem, with input consisting of an $n \times n$ matrix of nonnegative integers. The rows of this matrix represent jobs that need to be done; the columns represent the workers that may be assigned to these jobs. The workers have different aptitudes for the various jobs: the number in the matrix at row i, column j is the cost of assigning worker j to job i. An *independent set* of cells of the matrix is a set for which each row and each column contains at most one cell in the set. Such a set describes a (partial) assignment of workers to jobs, where each worker is doing at most one job and each job is being done by at most one worker. The goal of the assignment problem is to find an assignment of each of the n workers to a unique job, such that the total cost of the worker-job pairs is minimized. In other words, we want to find an independent set of n cells which minimizes the sum.

		worker		
		Alice	Bob	Carol
job	A	4	4	1
	B	3	2	1
	C	5	3	7

For example, in the table above, we see that Alice, Bob and Carol need to be assigned to the three jobs A, B and C. Carol is very good at jobs A and B, but we want to avoid assigning her to job C; because both Alice and Bob are better at job B than at job A, we choose to assign Carol to job A. Of the remaining costs, that of assigning Bob to job B is smallest. However, that would leave Alice with job C. A better solution is to assign Bob to job C and Alice to job B, for a total cost of 7.

3.1 Theory and basic form of the Hungarian algorithm

Polynomial-time algorithms for this problem were first found by [Ku55] and [Mu57]. The algorithm is based on two observations. First, if we pick a row or column and add a (possibly negative) constant to each of its elements, then this does not change the solution of the problem: each independent set contains exactly one modified element, so all candidate solutions change by the same value.

The other observation is a key theorem from graph theory. We first need to define two terms.

Definition 3.1 *A* matching *is a subset of the edges in an undirected graph such that no vertex is the endpoint of more than one of those edges.*

Definition 3.2 *A* vertex cover *is a subset of the vertices in an undirected graph such that for each edge, at least one of its endpoints is in that subset.*

Now we can state the theorem.

Theorem 3.3 (Kőnig's theorem) *In a bipartite graph G, the size of a largest matching is equal to the size of a smallest vertex cover.*

If we view the rows and columns of the matrix as $2n$ vertices in a graph, and join two of these vertices by an edge if the corresponding cell in the matrix contains a 0, then we obtain a bipartite graph. Applying the theorem to this graph tells us that a largest set of independent zeros (which corresponds to a set of edges which share no endpoints) has the same size as a smallest set of rows and/or columns covering all zeros.

Kuhn named his algorithm "the Hungarian algorithm" in [Ku55], after Kőnig's nationality, reflecting the importance of these theoretical results to the development of the algorithm.

Though the details differ between the algorithms in [Ku55] and [Mu57], both algorithms follow the same general pattern. First, we find a set of lines (i.e., rows and/or columns) that cover all zeros in the matrix. If this requires n lines, then by Kőnig's theorem there is an independent set of n zeros, so we have a complete assignment and the algorithm terminates. Otherwise, let h be the value of a smallest uncovered element of the matrix. Then for each covered column, add h to all its elements, and for each uncovered row, subtract h from all its elements. Repeat these steps until the assignment is complete.

Each time the matrix is modified, this happens in such a way that the solution is unchanged (by the first observation above). The net result of the modification is that all uncovered cells are reduced by h (so that at least one of them becomes a zero), all cells for which both the row and column are covered are increased by h, and other cells are unchanged. This ensures that all cells remain nonnegative.

To show that the algorithm terminates, it suffices to show that the sum of all cells decreases with each modification step. Because the number of covered lines was less than n, the number of uncovered rows must be larger than the number of covered columns, and the number of uncovered columns must be larger than the number of covered rows. Hence the number of cells whose value decreases is larger than the number of cells whose values increases, so that their total decreases each time.

3.2 Improvements of the algorithm

Algorithm 1 Pseudocode of [Mu57]'s Hungarian algorithm

```
initialize
repeat
    uncover all columns; cover all rows containing a starred zero
    repeat
        if there are uncovered zeros then
            choose an uncovered zero
            cover its column
            if that column contained a starred zero then
                uncover the starred zero's row
            end if
        else
            find the smallest uncovered element
            modify matrix
        end if
    until we covered a column that contained no starred zero
    star a larger set of zeros
until the assignment is complete
```

In order to understand some changes to the algorithm we will describe later, we first need to take a closer look at the version of the algorithm described in [Mu57], shown in Algorithm 1. This algorithm tracks an independent set of zeros by marking them with stars. Initially, only rows are used when covering all zeros. Then when another uncovered zero appears (due to the matrix being modified or due to rows being uncovered), its column is covered. If this column contained a starred zero, that zero's row is uncovered (possibly causing new zeros to become uncovered). If the column did not contain a starred zero, then it can be seen from Kőnig's theorem that there must be an independent set of $k + 1$ elements, where k is the number of zeros that already have a star. This new set can also be found efficiently.

At most $k+1$ uncovered zeros are chosen and at most k modifications to the matrix are made until we star an additional zero. Every time a zero is

found, all columns are uncovered and we again cover each row that contains a starred zero. After n repetitions of this, we have n independent zeros and the algorithm terminates. All this requires $O(n^4)$ operations on matrix elements.

A refinement in [La76] allows the algorithm to be performed in $O(n^3)$ steps by tracking the minimum value in each column incrementally.

Further improvements to reduce the amount of work performed by the algorithm are described in [JV86]. These improvements are:

- Rather than looking for the smallest element in the entire matrix, only look in one uncovered row at a time. Until an additional zero is starred, only covered rows and that one uncovered row are considered when looking for the minimum.

- The algorithm consists of an outer loop (which stars an additional zero in each iteration) and an inner loop (which identifies an uncovered zero). Matrix modifications in principle happen within the inner loop, but can be postponed to outside this loop where several can be applied together.

These modifications to the Hungarian algorithm reveal an interesting connection with the minimum-cost maximum-flow problem and a certain class of algorithms for it. This latter problem is a generalization of the assignment problem. In graph-theoretical terms, the assignment problem considers a bipartite graph with costs on the edges, and asks for the cheapest way to transport each of n units of some commodity from a distribution center, via a vertex in one part, to a vertex in the other part, and from there to a common destination. The minimum-cost maximum-flow problem allows an arbitrary directed graph with two special vertices marked as the source and the sink, where each edge additionally has a capacity that limits how many units may be transported along that edge. (In the assignment problem, all edges have capacity one.)

One class of algorithms for the minimum-cost maximum-flow problem repeatedly augments its solution by a shortest path from the source to the sink. Such a path may travel along edges that have not been used to capacity yet. It may also travel backwards along edges used in the solution. This represents undoing that part of the solution, hence the cost of such backward travel is the negative of the normal cost. If no augmenting path exists, the solution is optimal.

A straightforward algorithm based on augmenting paths applied to the assignment problem would take at least $O(n^4)$. An $O(n^3)$ algorithm of this type for the assignment problem was described in [To71]. To accomplish this, we want to use Dijkstra's algorithm to find each shortest augmenting path. However, if the augmenting path follows an edge that was already in the assignment, then its cost needs to be subtracted from rather than added to

the total cost of the path. Dijkstra's algorithm does not allow such negative edge costs. But the modifications to the matrix in [JV86] correspond to modifications to the edge costs so that edges in the assignment always have cost 0, and the highly efficient Dijkstra's algorithm can still be used. In fact, the modifications in [JV86] make the Hungarian algorithm equivalent to a shortest augmenting path algorithm. This equivalence also allows a concise explanation of the procedure used in Algorithm 1 to increase the set of starred zeros: this is accomplished by following the shortest augmenting path in the graph, unstarring the starred zeros and starring those that were unstarred.

4 Our algorithm

We intend to write a program that solves the assignment problem on the matrix of distances between two sets of strings. Because such distances may be expensive to compute, we would like to avoid as many such computations as possible. To know the optimal solution of the assignment problem, for many elements of the matrix it will suffice to know a lower bound rather than the exact value of the distance. To use this idea and solve the assignment problem more efficiently, we need a way to identify quickly which cells are likely to contain small distances, and need guaranteed lower bounds on the rest. Both can be accomplished by creating an index on one of the files. Then for a given line in the other file, we can quickly determine which lines in the first file are most similar. Later in this section, we discuss in detail how the Hungarian algorithm can be adapted to use such lower bounds, but first we need to choose a distance measure.

4.1 Distance measure

The usefulness of our program relies on an appropriate choice of the underlying distance measure. We want this measure to be able to reflect many types of similarity between two lines, so that two lines that look related to a human reader will also have a small distance by our measure. If our distance measure fails to satisfy this, then the optimal assignment found by our algorithm may often look wrong.

Edit distances capture mostly small editing operations, as used when correcting typographical errors. This is one kind of similarity we wish to capture, but certainly not the only one. For example, rephrasing an English sentence may cause parts of it to be moved elsewhere, which would be assigned too large a cost by an edit distance. This makes edit distances an unsuitable choice for our purposes.

Specialized distances can be discarded immediately: we wish to write a general purpose program which can be applied to strings from many different domains, and a specialized measure would limit this wide applicability.

We can imagine that token-based distances might generate good results, but as discussed in Section 2.1.3, they may treat strings as similar even when they are not, as long as their sets of tokens are similar.

Of the types of string distance measures discussed in Section 2.1, this leaves block edit distances and compression-based distances. Both seem like good choices, as they can capture many kinds of similarity between pairs of strings. We limited our treatment of compression algorithms to the Lempel-Ziv family; with that restriction, these two classes are very similar. Their main differences are:

- Compression-based distances have a firmer theoretical basis for assigning numerical values to the costs of editing operations; the block edit distances seem to use relatively ad-hoc numbers;
- Block edit distances create the target string from substrings of the source string; compression-based distances can take their copies not just from the source string, but also from the already encoded part of the target string.

The first is definitely an advantage of compression-based distances. The second is not a clear advantage to either: it may mean that compression-based distances satisfy the triangle inequality at least approximately, while on the other hand, searching both strings for substrings to copy may increase the computational cost and make it harder to establish lower bounds on the distance. Because the importance of the triangle inequality to our application is not evident, we choose a distance measure that is a mixture of the two: a block edit distance with costs based on compression, or equivalently, a compression-based distance that only copies substrings from the source string.

For natural language strings, `LZ77` achieves better results than later members of the Lempel-Ziv family except when we are dealing with really long strings (over 1 MB) [MRS98]. Therefore, we choose to base our distance on `LZ77`: we consider all substrings of the source string as candidates for copying, rather than using some more limited subset of these as a dictionary.

Most compression algorithms use a greedy method for choosing a substring to copy. This may not yield optimal results; for example, instead of greedily encoding the obvious match, it may be possible to first encode a single literal character, then copy a much larger match. This results in a smaller overall cost, assuming that encoding a literal is cheaper than encoding a match. For large files, compressing with a greedy algorithm will be faster than trying to find an optimal encoding of the file, while the difference in code length will usually be negligible compared to the total size. However, we are interested in an accurate measure of string similarity; using a greedy algorithm introduces undesirable noise into this measurement. Therefore, we

choose to use an optimal algorithm. It will turn out in Section 4.4 that an efficient algorithm can be found to compute this optimal code length.

To encode a target string given a source string, we will transmit a sequence consisting of literal characters, pointers to substrings of the source string, and eventually a symbol denoting that the target string has ended. We will use the following code lengths for these:

- To encode a literal, we transmit a zero bit, followed by the eight bits of the literal character. (We make no assumptions about the character encoding used by the files, and treat the strings as sequences of bytes.)
- To encode a substring from the source string, we transmit a one bit, followed by a sequence of bits identifying the beginning and end of the substring.
- To encode the end of the target string, we again transmit a one bit, followed by a special sequence of bits.

The second and third of these require us to encode which option out of a number of options we choose. The number of possible ways to choose a nonempty substring from a source string of length n is $n(n+1)/2$. We want to allow all of these to be encoded. With the end symbol, there are $n(n+1)/2+1$ options to encode. It seems reasonable to assign the same cost to all options, and thus in particular to all possible substrings, regardless of their lengths; we have no reason to assign smaller costs to shorter substrings. Uniform code lengths also simplify the computation of distances and lower bounds. The length of the uniform code is $\lg(n(n+1)/2+1)$ bits. As is standard when working with minimum description lengths, we do not round this number to an integer.

This distance measure is clearly not symmetric, but that may be the way it should be: the cost of editing one string to make another may be quite different from the cost of reversing that operation, and it may be desirable that our algorithm uses the cost for the appropriate direction.

The distance measure is also always positive, even for identical strings. However, the following similar property does hold: for fixed t, the distance from s to t is minimized when $s = t$.

We also want to know costs for inserting or deleting an entire line. Deleting a line is free, as we are interested in the cost of encoding all lines in the target file, and a deleted line is not in there. The cost of inserting a line into the target file without a corresponding line in the source file is taken to be equal to the cost of encoding the target line using the empty string as source line. The result is that such a target line will be encoded as a sequence of literals, with an interleaved sequence of zeros and eventually a one to announce the end of the string.

4.2 Hashing

To speed up the determination of a minimum cost assignment, we will index the first file. For this purpose, we will use two hash tables. For the first, we will use a hash function on complete lines; for the second, a hash function on q-grams. The index on lines will allow us to determine quickly if a line b in the second file has an identical copy a in the first file. If such a line a exists, then very likely b will be assigned to it, and we may not have to compute distances to b from any other line in the first file. If no identical copy of b exists in the first file, the index on q-grams will help us identify good candidates for assigning to b: non-identical but similar lines in the first file.

As discussed in Section 2.2.2, using a rolling hash function for the index on q-grams will allow us to compute the hash values of overlapping q-grams more efficiently. We use a hash function of the same form as used in [KR87]:

$$H(s) \equiv \sum_{i=1}^{|s|} s_i \cdot a^{|s|-i} \pmod{M}.$$

Here the string s to be hashed is treated as a sequence of integers. Our choices for the modulus M and the multiplier a will be discussed below.

If we want to compute hash values of all q-grams of the string s, then we can compute the hash value of the q-gram starting at character $t+1$ (denoted s_{t+1}^{t+q}) from the hash value of the q-gram at t with

$$H(s_{t+1}^{t+q}) \equiv (H(s_t^{t+q-1}) - s_t \cdot a^{q-1}) \cdot a + s_{t+q} \pmod{M}.$$

This expression first removes the character s_t from the hash value, then multiplies by a so that the remaining characters have the new correct multiplier (using distributivity), and finally adds the character s_{t+q}.

In [KR87], the modulus M is chosen at random from a certain range of primes, which they show has a high probability of producing a hash function with few collisions on the input. We do not use this randomized approach, but for computational convenience choose M to be a power of two large enough to facilitate the number of entries we expect in the hash table. The quality of H as a hash function then depends on the apparent randomness of the sequences of the form $c, c \cdot a, c \cdot a^2, \ldots \pmod{M}$. Because these sequences are exactly the output of a multiplicative congruential generator with multiplier a for different seeds c, we choose $a = 69069$, which is mentioned in [Kn97, pages 106–107] as a good choice for such generators when the modulus is a power of two.

It is not obviously best to store all q-grams of the first file in the index: all these q-grams contain a lot of redundant information due to their overlap. A more general approach is to store only every hth q-gram. For $h = 1$, this

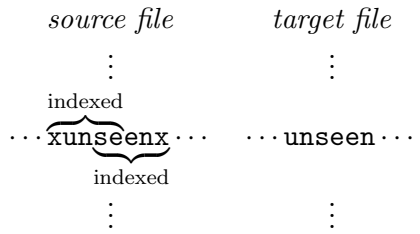

Figure 1: The common substring `unseen` is unnoticed by the index for $q = 5$, $h = 3$

gives the old approach of collecting all q-grams; for $h \geq q$, it leaves non-overlapping q-grams, which are called q-samples in [ST96]. We allow the parameters q and h to be adjusted freely.

4.3 Computing lower bounds on distances

Using the index of q-grams of the first file, we want to be able to compute lower bounds on the distance from string a to string b. This will be done as follows: given b, we look up all its q-grams in the index, retrieving for each q-grams a list of locations where it occurs. (If $h = 1$, these lists will contain all occurrences; otherwise, they will contain only subsets.) Each location consists of a line number and a position within that line. Traversing these lists simultaneously, we visit all source lines a for which at least one matching q-gram was found. We call these matching q-grams *fingerprints*. For each such line a, we use the fingerprint information to compute a lower bound on the distance from a to b. For source lines for which no fingerprints were found, such lower bounds can be computed without referring to the index.

If a substring shared by a and b is sufficiently long, then it will be detected by the index. To be precise, the largest substring that can go undetected is of length $q+h-2$: as illustrated in Figure 1, such a substring would have to start in a position in a one character after a position where a fingerprint was taken, and extend until one character before the end of the next fingerprint (which starts h characters later). Thus in a region of b for which no fingerprints were found in a, we know that no matches of length greater than $q + h - 2$ exist. We are looking for a lower bound, so we will be optimistic and assume such matches exist at all positions in b. To encode a region of b where no fingerprints were found, we use as many matches of size $q + h - 2$ as will fit, then encode any remaining characters either as literals or with another match.

We will use the following information from the index when establishing a lower bound on a distance for which fingerprints were found:

- The number of positions in b for which fingerprints were found;

- The first and last positions in b for which a fingerprint was found;
- The number of *extending pairs* of such positions: two positions in b that differ by exactly h, and for which corresponding q-grams in a occur in the same order, at positions that also differ by exactly h.

All these statistics can be determined efficiently when traversing the lists obtained from the index. We use these statistics as follows to establish lower bounds:

- Each fingerprint may indicate a matching substring of length at most $q + 2h - 2$ (if that substring extends $h - 1$ characters to either side of the fingerprint), or k fingerprints may work together to allow the existence of a match of length $q + (k + 1)h - 2$;
- The first and last positions with fingerprints in b limit how much of b can be encoded with the help of matches longer than $q + h - 2$; the rest is encoded with matches of length at most $q + h - 2$ as described above;
- Regardless of how they are used exactly, each fingerprint allows an additional h characters to be encoded by means of matches rather than literals. However, the shortest code lengths are achieved if we can encode b with just a few long matches. If k fingerprints are all due to the same matching substring, then each neighbouring pair of fingerprints must be an extending pair. This way, the number of extending pairs limits how long a single match can be.

Now the algorithm for computing a lower bound using these statistics boils down to the following: first, greedily make a single match that is as large as allowed by the constraints; then, make further matches of length $q+2h-2$, supported by one fingerprint each. Continue with matches of length $q + h - 2$ not supported by fingerprints, and finally encode the remainder of b with literals. The algorithm must also check for several boundary cases, for example when the optimal encoding requires match lengths not of the form $q + kh - 2$. We will not go into such details here.

4.4 Computing distances exactly

Though a lot of work goes into avoiding this, our program will have to compute distances between some pairs of strings exactly. If we are unlucky and the lower bounds only rule out a few source strings as candidates for assigning to a target string, we may still end up computing a large fraction of these distances. Therefore, we would like such computations to be fast. Since we want to find an optimal encoding of the target string rather than a greedy one, it is not obvious that this efficiency is possible.

The problem of finding the optimal encoding of the target string b as a sequence of literal characters and substring matches from the source string a can be translated to a problem of finding the shortest path in a graph. This graph has vertices $v_0, v_1, \ldots, v_{|b|}$ which correspond to the prefixes of b of the lengths indicated by the subscripts, and a directed edge between two vertices for each encoding operation. We want to find the cheapest path from vertex v_0 to vertex $v_{|b|}$. Because all operations have nonnegative costs, we can use Dijkstra's algorithm to solve this problem.

But we can do even better than in the general shortest path problem with nonnegative edge costs. First, we know that the cost of encoding the first k characters of b is nondecreasing in k. Hence, if Dijkstra's algorithm has visited vertices v_0 up to v_{k-1}, then the next vertex to visit will be vertex v_k; no priority queue is necessary. Second, the edges in the graph are highly structured: each vertex v_k other than $v_{|b|}$ has a (literal) edge to v_{k+1} and these edges all have the same cost c_1; similarly, all other (match) edges share a single cost c_2, and if vertices v_i and v_ℓ are connected by a match edge, then so are vertices v_j and v_k for all $i \leq j < k \leq \ell$. Assume that $c_1 < c_2$, as otherwise greedily using any available match is optimal. Then it follows that when Dijkstra's algorithm is currently visiting vertex v_k, there are only two types of vertices for which the shortest path may come from v_k:

- the vertex v_{k+1};
- those vertices which can be reached from v_k by a single match and which have not been reached before.

Vertices that have been reached before are disqualified from the second set: among all shortest paths to the vertices visited so far, the path to v_k is longest, so any vertex that has been reached before via a vertex other than v_k will have been reached at least as efficiently as it could be reached through v_k. It follows from the structure of the match edges that the vertices in the second set form a contiguous sequence, extending from the first unreached vertex to the last vertex reachable from v_k by a match. Thus, Dijkstra's algorithm needs to do very little work when visiting a vertex.

In order to quickly find which vertices are connected by matches, we create a small index on the ℓ-grams of a, where ℓ is the smallest number of characters for which encoding them as a match is cheaper than encoding them as a sequence of literals: matches shorter than ℓ are not worthwhile. Using this new index is likely to be more efficient than using the previously created index on the entire source file: that index allows us to quickly retrieve a list of all occurrences of a particular q-gram in the first file, but not to quickly filter out just the ones in line a. Additionally, it is likely that $q > \ell$, in which case that index would not help us find matches smaller than q.

4.5 Adapting the Hungarian algorithm

We now describe what changes we made to the Hungarian algorithm to exploit the hash tables. Our starting point is the version of the algorithm in [JV86]. Pseudocode of this algorithm is given in Algorithm 2; our algorithm will differ very little from it at this level of detail.

Algorithm 2 Pseudocode of [JV86]'s Hungarian algorithm

```
initialize
repeat
    uncover all columns; cover all rows which are already assigned
    determine which unassigned row will be considered
    repeat
        find the smallest uncovered element among the considered rows
        cover its column
        if augmenting path not found yet then
            partially modify matrix
            uncover row and add it to consideration
        end if
    until an augmenting path is found
    finish modifying matrix
    augment assignment
until the assignment is complete
```

One modification has already been applied to Algorithm 2, because it is a simplification. Since we chose a real-valued distance function, we do not expect many cells in the matrix to contain exactly the same distance value. Therefore, whenever we modify the matrix, we expect only one element to becomes zero, namely the minimum that determined by how much we updated the matrix. Rather than looking for zeros, our algorithm assumes there are none and looks for the minimum, then modifies the matrix to turn it into a zero. Of course, the minimum may have already been a zero, but the (superfluous) update operation takes very little computational work because it is performed lazily.

4.5.1 Finding the minimum

In order to find the smallest cell in some region of the matrix without having to consider all cells, we use a system of priority queues. For each row in the matrix (corresponding to a line in the (unindexed) target file), we maintain a *row queue* which contains all exact distance values and lower bounds that have been computed so far for cells in that row. There is also a *master queue*: each element in this queue corresponds to an entire row, and uses that row's minimum as its key.

One optimization introduced by [JV86] was to consider only some subset of the rows when looking for a minimum. We use this same optimization, and only have elements in the master queue for those rows under consideration for finding the minimum. Also, unlike the row queues, the master queue is cleared every time we augment the assignment. Adding a row to consideration is accomplished simply by adding an element to the master queue.

Now whenever we need to find the smallest value in some region of the matrix, we first pop the minimum from the master queue to find out which row contains the smallest element. Then popping an element from that row's queue will tell us what column that element comes from, and whether it is an exact distance or a lower bound. If we find an exact distance, then this is the minimum and we are done. However, if we find a lower bound, then we need to compute the corresponding exact distance. It may be equal to the lower bound, in which case we are still done. But in general, it may of course be larger, in which case we insert a new element with the exact distance into the row queue, insert a new element with the row's new minimum into the master queue, and repeat.

In every iteration of the inner loop, a column is covered, and its cells should no longer be considered when looking for the minimum value. We do not look through all row queues to remove elements corresponding to a column when it gets covered. Instead, we solve the problem when such an element is popped from a row queue. Then we temporarily remove it from its row queue, and proceed as when we encountered a lower bound. When all columns are uncovered at the start of the outer loop, we return all these temporarily removed elements to their row queues.

4.5.2 Labels

After the minimum element has been determined, we need to modify the matrix to make the minimum equal to zero. Like almost all implementations of the Hungarian algorithm, we do not actually recompute all cells in a row or column that we modify, but only change a row or column *label*: row labels track by how much all cells in that row have been decreased, and column labels track the increase of their cells.

Whenever we add an element to a row queue, we use as its key not just (the lower bound on) the distance, but that value plus the value of its column label. Because all elements in one row queue share the same row label, we do need to take row labels into account here. This is done when a row is added to the master queue: then the row label's value is subtracted from the minimum value in the row queue.

Like [JV86], we do not update the labels immediately after determining the minimum element, but we postpone part of the label update until an augmenting path has been found and the independent set can be extended.

However, we perform our updates slightly differently, to be compatible with the priority queues. We have the requirement that the elements among which we are looking for the minimum must have the correct values relative to each other whenever we need to determine that minimum. (This is not required in [JV86], because there the minimum is found in a different way.) But the relative values can be maintained by only updating one column and one row label after finding a minimum: for the column just covered (instead of for all columns that were already covered), and for the row about to be uncovered (instead of for all currently uncovered rows under consideration). After the inner loop, we need to get all modified rows and columns back in line with the unmodified ones, but this is a cheap operation when done at that time.

A row label will only be updated just before that row is added to the master queue, and again after the inner loop is done. This means that during the time that a row is represented in the master queue, its row label is constant. Hence, we never have to deal with outdated row label values in the master queue.

On the other hand, when a column label is changed, this will usually affect many elements spread out over the row queues. Similar to how we handle covered columns, we handle this situation when an element is popped from a row queue and the column label used for its key turns out to be outdated. Then we recompute its key using the current value of the column label and proceed as with a lower bound. Because modifications to column labels will only increase matrix cells as time passes, the outdated keys are still lower bounds on the actual values, so that this procedure guarantees that we find the minimum.

4.5.3 Creating the row queues

For two similar files, we expect it to happen often that a given line in the target file has one very similar or even identical twin in the source file, while all other lines in the source file are quite different. This is the situation we hope to take advantage of with the row queues. In such a situation, where only a very small portion of a row will be considered, we would like to avoid wasting time filling up the row queue with lower bounds for every column. So when constructing the row queues during initialization, for each line b in the second file, we use the index on the first file's whole lines to determine if any are identical to that line. If such lines are found, then their exact distances are added to the row queue. For all remaining lines, we add a single common lower bound. The value of this lower bound is attained by a string a which contains b as a substring, but is one character longer: for this pair, it holds that $a \neq b$, but b can still be encoded by a single match which is only slightly more expensive than when $a = b$.

When the common lower bound is popped from the row queue, then we need to spend a little more computation time to add more precise lower

bounds to the row queue. At this point, we consult the index on q-grams in the first file, following the procedure described in Section 4.3. No time is spent yet on lines in the first file without matching q-grams; for these, a new common lower bound is enqueued. Only when this lower bound is popped from the queue do we add individual lower bounds to the queue for source lines with no matching q-grams in the index. How these values can be computed was also described in Section 4.3.

4.5.4 Rectangular matrices

The assignment problem as stated in Section 3 takes a square matrix as input. However, the two files being compared by our algorithm may of course be of different size. In [BL71], it is shown that the original Hungarian algorithm can be applied to such matrices with only some small modifications to the initialization steps. The result is equivalent to applying the algorithm to the square matrix obtained by extending the actual input matrix with cells containing some large number. However, the optimization of considering only one row at a time used in [JV86] does assume square matrices. It will still work without modification if the number of columns is larger than the number of rows, as each row must then still be assigned to a column. However, if there are more rows than columns, then a number of rows equal to the difference, k, must remain unassigned in the end. In that case, considering only a single row at a time may force the algorithm to assign a row that should not be assigned. To avoid this, we consider $k + 1$ rows at the start of each iteration iteration, adding those rows to the master queue. Also, all unassigned rows should retain their correct relative values, as these may affect which rows to assign and which to leave unassigned. Therefore, we do not update the row labels of the k rows that were considered but will remain unassigned after the assignment is augmented.

When choosing among the different unassigned rows under consideration, the algorithm will prefer the row which contains the cell with the smallest cost. However, the behaviour we want is that it prefers the row with the greatest *reduction* in cost when going from not assigning to assigning that row. In effect, we should change the initial value of all cells in each row so that they reflect this cost reduction rather that the absolute cost. This can of course be accomplished easily by initializing the row labels to different values.

4.5.5 Voluntarily assigning fewer lines

Our algorithm does not always need to find a complete assignment, assigning lines from the files to each other until either file runs out. It has an alternative option: mark a line from the source file as deleted, and a line from the target file as inserted from scratch. For our distance measure, this is the better

option for example if the strings have no characters in common.

The behaviour we desire is that whenever a distance between two lines is computed, if not assigning those lines would be cheaper than assigning them to each other, we use that smaller cost instead. If those lines end up being assigned to each other, then we treat both as unassigned. We can achieve this behaviour by inserting into each row queue a single special element, with as its cost the cost of inserting that line into the target file without a corresponding line in the source file. When this element is the smallest in the row queue, it means that inserting the target line is cheaper than assigning it to any source line.

It suffices to have a single such element per row queue, which is not associated to any one column. To see why, consider the desired behaviour described above. We call a matrix cell an *insertion cell* if its value is the cost of inserting a target line anew, rather than assigning it to a source line. When an insertion cell is used for the assignment of some row i, then apparently there are no unassigned columns with non-insertion cells in row i. If in some future iteration, we consider row i again (because the column of its insertion cell contained the smallest element), then apparently there are still unassigned columns (or this iteration would not have happened); for all of these, the cell where they insersect row i must be an insertion cell. Then we would select one of those other insertion cells to complete the augmenting path at no extra cost. Hence we could have ignored what column the insertion was in originally, because we can always assume it is in a column we do not need anymore.

4.6 Theoretical performance analysis

It is hard to say theoretically how much the improvements detailed above are worth, because they are intended to provide benefit only in certain lucky cases, where for each line in the target file there is a line in the source file that is clearly more similar than other lines. Indeed, the worst case of our algorithm is slightly worse than that of the standard Hungarian algorithm: $O(n^3 \log n)$ instead of $O(n^3)$. (Here we ignore the cost of reading the files and computing exact distances—tasks which our algorithm also has to do, but which are not part of the assignment problem solved by the standard Hungarian algorithm.) This $O(n^3 \log n)$ is caused by the operations that must be done when popping row queues yields invalid elements, that is, ones for which the column is covered or the column label is outdated. Such events may happen $O(n^3)$ times during the execution of the algorithm, causing heap operations costing $O(\log n)$ each time.

One example of a particularly lucky case for our algorithm is if the two files are simply permutations of each other; in particular, this happens when the two files are identical. Then the exact distance computation is never called (because the index on lines will recognize the pairs of identical lines

and know their distance directly), and the number of times that a column gets covered is limited by the presence of identical lines within either file. Then on average (depending on the performance of the hash tables), the computational complexity is linear in the length of the files.

5 Experiments

In this section, we will look at the performance of our algorithm in practice. We are interested in two things: correctness and speed. By "correctness" we mean whether the string distance measure we use lead to assignments that correspond to our intuition of which lines of the input files correspond to which.

5.1 Correctness

The practical problem we had in mind when starting this project is that of finding a summary of the differences between two versions of a file of card information for the card game *Magic: the Gathering*® [MtG12]: one version from when the set of cards was still under development, and one containing the final, printed information. Until now, this task had to be done mostly by hand, because of the unpredictable nature of the changes between the files. Because the cards are listed in a different order in each file, `diff` would be able to match only very few cards correctly, and because the titles of many cards are changed between the two versions, sorting both files by name before running `diff` would not help much. Even a special purpose sorting method based on domain knowledge would not be robust against all the potential editing operations that could have been performed between the two files.

However, our program seems to handle the task very well: for the set of cards that was released this summer, we checked its results by hand and found that each card in the final file was correctly assigned to the earlier version of that same card in the development file. Finding this assignment would have been a lot of work if done by hand, or with tools inadequate for this purpose. This result means that our string distance measure was able to see through all the modifications made to these cards, and recognize the relevant similarities.

5.2 Speed

For the pair of files mentioned above, the optimizations proved very valuable. Even though both files contain only 229 elements and measure a modest 33 kilobytes each, computing the distance between each pair of them would take over 5 seconds. Our algorithm takes less than 150 milliseconds to find a complete assignment, by avoiding most of the work of computing distances. It was able to avoid this work because each card in the final file really was

q:	h:				
	1	2	3	4	5
1	524	800	1120	1376	1516
2	184	204	340	476	560
3	132	140	236	376	464
4	144	172	292	396	476
5	180	248	364	452	524
6	256	352	432	506	604

Table 1: Computation time in milliseconds for various values of q (the size of the q-grams used by the index) and h (the interval at which these q-grams were sampled)

produced by taking a card in the development file and making some changes to it. The connections between these pairs of cards can be recognized by looking at the files, and the index enables our program to do this quickly.

In another experiment, we considered two random files, each consisting of 100 lines of 50 characters chosen uniformly from the capital letters and the space. For these files, it does not hold true that each line in one file has a unique relative in the other file. This makes it necessary to compute a much larger fraction of exact distances. However, as we will see below, even in that difficult case, many exact distance computations could be avoided by tuning the parameters of the index.

Because the hash table of q-grams plays such a large role in the algorithm, we experimented with different values of q and h to find which values offer the best performance. The results are shown in Table 1. We see that the best results are obtained when q is 3 or 4, and $h = 1$. Increasing h is generally harmful to performance (with the exception that $q = 3$, $h = 2$ stands out as another good choice), so adding all q-grams to the index seems best in our application. Based on these results, we set $q = 3$, $h = 1$ as the defaults in our program.

In the experiment with random files, the behaviour of the index changes. For files generated this way, all q-grams over the used alphabet are equally likely, and their expected frequency drops off exponentially in q. This is very different from the behaviour we find in natural language files, where some q-grams are expected to occur very frequently, even for large q. For these random files, setting q to 1 or 2 and h to 1 proved best; for larger values, 100% of the distances had to be computed, but for $q = 2$, this was only 5%.

Another factor that might impact the program's efficiency is the performance of the hash tables. The hash table of lines uses a cyclic redundancy check as a hash function, which can be trusted to give good performance. The hash table of q-grams, on the other hand, uses our custom rolling hash

	χ^2 (msb):		χ^2 (no msb):		df	95% quantile
lines	28.4468		41.0912		31	44.985
1-grams	132.000	!	3.60000		15	24.996
2-grams	1913.85		2280.43	!	2047	2153.37
3-grams	4251.33	!	4114.16		4095	4244.99
4-grams	4012.61		4003.74		4095	4244.99
5-grams	4135.17		4294.96	!	4095	4244.99

Table 2: Results of χ^2-tests for the Magic: the Gathering files

	χ^2 (msb):		χ^2 (no msb):		df	95% quantile
lines	13.1200		8.80000		15	24.996
1-grams	178.630	!	2.03704		15	24.996
2-grams	416.021		634.343	!	511	564.696
3-grams	706.634	!	553.827		511	564.696
4-grams	497.408		515.142		511	564.696
5-grams	476.814		475.923		511	564.696

Table 3: Results of χ^2-tests for the random files

function described in Section 4.2 and needed to be tested. In our experiments, we computed χ^2-scores of (groups of) buckets, using the formula from [Kn97, 3.3.1]. For the statistic to be reliable, the expected number of values in each bucket must be large enough. Therefore we join sets of sixteen buckets together into groups. Two different ways of grouping are reported in Tables 2 and 3. In the column marked "msb" we use the first four most significant bits of the binary representation of each bucket index, ignore the next four, and use the rest to determine the bucket's group. For the column marked "no msb", the four most significant bits were ignored, but all others were used.

We compared the values of this statistic against the 95% quantile of the χ^2-distribution with the appropriate degrees of freedom. When the statistic exceeds this quantile, it is marked with an exclamation point. This means that uniform random values would be this unevenly distributed only 5% of the time. All χ^2-scores for the hash table of lines are below this threshold, showing that the cyclic redundancy check is behaving as expected. However, for the hash table of q-grams, several exclamation points appear. The ones for $q = 1, 2, 3$ appear to follow a pattern. It should have been expected that the most significant bits of 1-grams would not be evenly distributed: for a 1-gram, the value of the rolling hash is just the character code of the only character, and ignoring the four least significant bits throws almost all its information away. This gives a false alarm, because the hash function is

actually perfect, mapping each 1-gram to a unique bucket. Something similar may be going on for $q = 2$ and 3: the number of hash buckets being used might simply be too small to satisfy the test. Though further investigation is warranted, enough χ^2-scores are below the threshold that we see no reason for concern in these results.

6 Conclusion

We implemented a program that finds the optimal assignment between two sets of strings. Though its applicability is not as wide as that of `diff`, our tool finds the correct answer very efficiently under appropriate circumstances.

The idea of comparing two files by finding an optimal assignment between their lines is new, to the best of our knowledge.

As a measure for the distance between two strings, we used a mixture between a compression-based distance and a block edit distance. In our experiments, this measure was able to recognize the similarity between pairs of strings that were related to each other through editing operations. This property made it a suitable choice for computing the distances to which the assignment problem could be applied.

We described and implemented a novel version of the Hungarian algorithm for the assignment problem, which uses lower bounds on distances to exclude many elements of the distance matrix from consideration, thereby saving the computational effort of computing the values of those elements exactly. Lower bounds could be computed efficiently with the help of indexing techniques from the approximate string matching literature.

We showed that an algorithm based on Dijkstra's shortest path algorithm could compute our distance measure very efficiently, because we only need to consider a subset of the matching substrings between the two input strings. Though distance measures very similar to ours exist in the literature, we found no reference to such an algorithm.

Our experiments were limited in scope. Further testing is required to tell whether the program works similarly well in a wider set of circumstances. One part of the algorithm that may be improved upon is the distance function, which can be replaced by a different one without affecting the rest of the program. Many different functions are possible, though implementing and comparing them is outside the scope of this thesis.

References

[BL71] F. Bourgeois and J.C. Lasalle, An extension of the Munkres algorithm for the assignment problem to rectangular matrices, Communications of the ACM, 14(12), 1971, 802–804

[CGK06] S. Chaudhuri, V. Ganti and R. Kaushik, A primitive operator for simlarity joins in data cleaning, Proceedings of the 22nd International Conference on Data Engineering, 2006, 5–16

[Ci07] R. Cilibrasi, Statistical inference through data compression, PhD thesis, ILLC, University of Amsterdam, 2007

[McC76] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM, 23(2), 1976, 262–272

[EH88] A. Ehrenfeucht and D. Haussler, A new distance metric on strings computable in linear time, Discrete Applied Mathematics, 20, 1988, 191–203

[GNU11] GNU diffutils manual, `http://www.gnu.org/software/diffutils/manual/`, version September 4, 2011

[Gr07] P.D. Grünwald, The Minimum Description Length principle, M.I.T. Press, Cambridge (MA), 2007

[Gu97] D. Gusfield, Algorithms on strings, trees, and sequences, Cambridge University Press, 1997

[HK06] J. Han and M. Kamber, Data mining: Concepts and techniques, Second Edition, Morgan Kaufmann, 2006

[He78] P. Heckel, A technique for isolating differences between files, Communications of the ACM, 21(4), 1978, 264–268

[JU91] P. Jokinen and E. Ukkonen, Two algorithms for approximate string matching in static texts, Proceedings Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 520, 1991, 240–248

[JV86] R. Jonker and A. Volgenant, Improving the Hungarian assignment algorithm, Operations Research Letters, 5, 1986, 171–175

[KR87] R.M. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development, 31(2), 1987, 249–260

[Kn97] D. Knuth, The art of computer programming, volume 2: Seminumerical algorithms, Third Edition, Addison-Wesley, 1997

[Kn98] D. Knuth, The art of computer programming, volume 3: Sorting and searching, Second Edition, Addison-Wesley, 1998

[Ku55] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Research Logistics Quarterly, 2, 1955, 83–97

[La76] E.L. Lawler, Combinatorial optimization: Networks and matroids, Holt, Rinehart & Winston, 1976

[LT97] D. Lopresti and A. Tomkins, Block edit models for approximate string matching, Theoretical Computer Science, 181, 1997, 159–179

[MtG12] Magic: the Gathering® website, `http://www.wizards.com/magic/TCG/Default.aspx`, retrieved August 28, 2012

[MRS98] Y. Matias, N. Rajpoot and S.C. Sahinalp, Implementation and experimental evaluation of flexible parsing for dynamic dictionary based data compression, Proceedings Workshop on Algorithm Engineering, 1998, 49–61

[MM85] W. Miller and E.W. Myers, A file comparison program, Software—Practice and Experience, 15(11), 1985, 1025–1040

[Mu57] J. Munkres, Algorithms for the assignment and transportation problems, Journal of the Society for Industrial and Applied Mathematics, 5(1), 1957, 32–38

[Me86] E.W. Myers, An O(ND) difference algorithm and its variations, Algorithmica, 1(2), 1986, 251–266

[Na01] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys, 33(1), 2001, 31–88

[NW70] S.B. Needleman and C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology, 48(3), 1970, 443–453

[ST96] E. Sutinen and J. Tarhio, Filtration with q-samples in approximate string matching, Proceedings 7th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science 1075, 1996, 50–61

[SW81] T.F. Smith and M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology, 147, 1981, 195–197

[Ti84] W.F. Tichy, The string-to-string correction problem with block moves, ACM Transactions on Computer Systems, 2(4), 1984, 309–321

[To71] N. Tomizawa, On some techniques useful for the solution of transportation problems, Networks, 1, 1971, 173–194

[Tr99] A. Tridgell, Efficient algorithms for sorting and synchronization, PhD thesis, Australian National University, 1999

[Uk92] E. Ukkonen, Approximate string-matching with q-grams and maximal matches, Theoretical Computer Science, 92, 1992, 191–211

[WM91] S. Wu and U. Manber, Fast text searching with errors, Technical Report TR 91-11, Department of Computer Science, University of Arizona, Tucson, 1991

[ZL77] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory, 23(3), 1977, 337–343