Finding the Neighborhood of a Query in a Dictionary

Danny Dolev* Yuval Harari Michal Parnas

Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, ISRAEL

Abstract

Many applications require the retrieval of all words from a fixed dictionary D, that are "close" to some input string. This paper defines a theoretical framework to study the performance of algorithms for this problem, and provides a basic algorithmic approach. It is shown that a certain class of algorithms, which we call D – oblivious algorithms, can not be optimal both in space and time. This is done by proving a lower bound on the tradeoff between the space and time complexities of D – oblivious algorithms. Several algorithms for this problem are presented, and their performance is compared to that of "Ispell", the standard speller of Unix. On the Webster English dictionary our algorithms are shown to be faster than "Ispell" by a significant factor, while incurring only a small cost in space.

1 Introduction

Many applications require the retrieval of all words from a fixed dictionary D, that are "close" to some query string. In some cases the query string may be erroneous, and the task is to identify all possible corrections. In other cases, the goal is to list all words related in a certain way to the query. We will refer to this as the *Approximate Query Retrieval* problem.

This problem can arise in many different fields. The most obvious examples come from spellers and speech-recognizers, while variations of the problem can be found in fields such as the analysis of DNA sequences and proteins in Chemistry and Biology and even the study of bird-singing (see [SK] for a detailed description of these and many more applications).

The problem of string matching with k-differences defined by Landau and Vishkin ([LV1]) reminds the approximate query retrieval problem. In [LV1] a text and a pattern are given, and the goal is to find all occurrences of the pattern in the text, each with at most k differences. Algorithms for string matching typically scan the whole text for each pattern (see [LV1], [LV2]). This approach may be used also for approximate query retrieval when the size, $|D|$, of the dictionary is small enough. Then it is feasible to search all the dictionary for all the alternative similar words of a given input word. However, when the dictionary is large, a more efficient approach is required.

Although there are many practical applications to the approximate query retrieval problem, only limited theoretical research has been done so far. One of the main contributions of this paper is to define a theoretical framework to study algorithms for this problem, and provide a basic algorithmic approach.

Informally, we are given a dictionary of words D over some alphabet Σ. We consider algorithms that store D in buckets, where each word may be mapped to one or more buckets, and preprocessing of D is allowed. Then the algorithm should be able to answer queries of the sort "What are all the words in D at some given bounded distance from a query word u?". The sequence of buckets probed by the algorithm in order to answer a given query u, will be called the *search sequence* of u.

An algorithm for which the mapping of words into buckets and the search sequences are fixed for any dictionary D, will be called *D-oblivious*. Such algorithms are of course preferred, since they are usually easier to design and implement. But more important they are able to handle a dynamically changing dictionary D, without having to change the data structure every time a word is added to D or deleted from it.

However, we show that no D – *oblivious* algorithm can be both space optimal (i.e., use space $|D|$) and time optimal (i.e., answer a query in time linear in the answer size). Furthermore, we give a lower bound on the tradeoff between the space and time complexities

*The research was supported in part by the Yeshaya Horowits Association.

0-8186-3630-0/93 $3.00 © 1993 IEEE

of such algorithms (see Section 3). Optimal space $D-$ *oblivious* algorithms are further studied in [DHLNP].

More efficient algorithms must therefore take into account the size of the dictionary D and the distribution of words in D. For example, in Chemistry and Biology applications, many words have the same prefix. The English dictionary contains many words that share the same suffix "ing" and so on. These observations may help design a more efficient algorithm for the specific application at hand. Another major concern should be whether the dictionary is static or dynamic. If the dictionary is static then preprocessing may be allowed, so that the best time-space tradeoffs are achieved; and even if the dictionary is dynamic, there may be some pre-knowledge on the possible distribution of words, that should be taken into account when designing the algorithm.

A similar observation is true for hashing algorithms. It was shown (e.g., [DKMMRT],[FS]) that if the dictionary is dynamic then a deterministic hashing algorithm cannot achieve constant search time using linear memory. [FKS] showed that if the dictionary is static, then it is possible to achieve constant search time using only linear memory.

Given the above, we next present four algorithms that take into account the size of the dictionary D and the distribution of words in it. We start by introducing two algorithms that are based on error correcting codes (see [PW],[MS] for definitions of error correcting codes). Both algorithms map the dictionary into buckets by exploiting the property of error correcting codes that partition the space into spheres around code words. We show how to choose the code as to optimize the time and space tradeoff for a given dictionary D (see Section 4).

We then describe two algorithms, *Sub* and *Part*, that are based on the observation that two words are similar if they have a common subword. Algorithm *Sub* is $D-$ *oblivious* and has low time and space complexity if the words are relatively short (see Section 5). Algorithm *Part* is more adaptive to D, and can be designed to have almost optimal time and space complexity (see Section 6). The performance of both algorithms was tested on the Webster English dictionary (i.e., $|\Sigma| = 26$), and compared to that of "Ispell", the standard speller of Unix. Both algorithms were found to be significantly faster than "Ispell", while incurring only a small cost in space. For example, "Ispell" needs time $\Omega(n|\Sigma|)$ to find all the words in the dictionary of Hamming distance at most one from a given query of length n, while *Part* gives an answer in time $O(1)$. These results are described in Section 7.

2 Problem Statement

Let U be a set of strings over an alphabet Σ. The strings may be all of the same length n or of different lengths. Let $D \subset U$ be a dictionary of words, which is usually a sparse subset of U.

Define the $c-$ *neighborhood* of $u \in U$ with respect to U to be $N_c(u) = \{v | v \in U,\ d(u,v) \leq c\}$, where $d(u,v)$ is some distance measure. The $c-$ *neighborhood* of $u \in U$ with respect to D is defined in a similar way as $N_c(u,D) = \{v | v \in D,\ d(u,v) \leq c\} = N_c(u) \cap D$. Let $N_c = |N_c(u)|$ be the number of words in the $c-$ *neighborhood* of any $u \in U$.

The *Approximate Query Retrieval* problem is to find an algorithm that stores D using as little memory as possible, and answers quickly queries of the sort "Given a query $u \in U$, return as answer the set $N_c(u,D)$." Such an algorithm will be called an *Approximate Query Retrieval* (AQR) algorithm.

2.1 Distance Measures

Different applications require different distance measures. We will use two distance measures, the *Hamming Distance* and the *Levenstein Distance* (see [SK]).

The Hamming distance can be used only on words u, v of the same length, and is simply the number of coordinates in which u and v differ.

The Levenstein distance can be used on words u, v of any given length, and is defined as the minimal number of *edit operations* that transform u to v. The edit operations that can be used are:

- **Insert**: Insert a letter to u.
- **Delete**: Delete a letter from u.
- **Exchange**: Exchange a letter of u with any letter from Σ.

It is possible also to add the operation **Swap** (i.e., swap two adjacent letters in u). Note that each one of the operations exchange and swap can be achieved by just performing a delete followed by an insert operation, while only doubling the distance.

2.2 The Model

Any AQR algorithm A performs two basic steps: storing the dictionary D in memory and answering queries. These two steps can be described formally as follows:

1. **Pre-process** D**:** Use a multi-valued-function $STORE : U \mapsto \{B_1, ..., B_m\}$ to map each word in D to one or more buckets B_i. Notice that the function $STORE$ may depend on the dictionary

D. It is possible for A to store in memory only buckets that are not empty, using some hashing scheme (e.g., [FKS]).

2. **Answer Queries:** Use a multi-valued-function $SEARCH : U \mapsto \{B_1, ..., B_m\}$ to associate a sequence of buckets S_u with every word $u \in U$, such that $N_c(u, D)$ is contained in this sequence of buckets. The sequence S_u will be called the *search sequence* of u. Then A can answer a query u by searching the buckets B_i specified by the search sequence S_u, and returning as answer only the words that are in $N_c(u, D)$.

If the functions $STORE$ and $SEARCH$ are oblivious to D, that is, the mapping into buckets and the search sequences are fixed and independent of D, then the algorithm A is called *D-Oblivious.*

2.3 Complexity Measures

The performance of an AQR algorithm A can be measured by the space it uses to store D, and the time it needs to answer a query $u \in U$. We define these measures as:

$$SPACE(A, D) = \sum_i |B_i|,$$

$$TIME(A, D, u) = |S_u| + \sum_{i \in S_u} |B_i|.$$

Notice that we add the length $|S_u|$ to the time complexity, since the algorithm has to access $|S_u|$ buckets, and each such access can result in a separate costly disk access.
Algorithm A is *space optimal* if $SPACE(A, D) = O(|D|)$, for every dictionary D. Algorithm A is *time optimal* if $TIME(A, D, u) = O(1+|N_c(u, D)|)$, for every dictionary D and for every query $u \in U$ (we add one, for one disk access).
Remark: It is possible also to measure the complexity of computing the functions $STORE$ and $SEARCH$. Note, that it is more important to have an easy to compute function $SEARCH$, since the algorithm should be able to answer queries fast. Whereas, the function $STORE$ will usually be used while pre-processing D, and therefore can be allowed to be more complicated to compute.

2.4 Two Basic AQR Algorithms

In order to demonstrate the above definitions and concepts, we introduce two simple AQR algorithms and state their performance.

- **Optimal space algorithm:** There is a bucket B_u for each $u \in D$, defined by $B_u = \{u\}$. Given a query u, the algorithm checks for each word $i \in N_c(u)$ if the bucket B_i is not empty and if so returns the word i as part of the answer [1]. Thus: $|S_u| = |N_c(u)| = N_c$, **Time:** $N_c + |N_c(u, D)|$, **Space:** $|D|$ (optimal).

- **Optimal time algorithm:** There is a bucket B_u for each word $u \in U$, defined by $B_u = N_c(u, D)$. Given a query u, the algorithm returns as answer all the words in the bucket B_u. Thus: $|S_u| = 1$, **Time:** $1+|N_c(u, D)|$ (optimal), **Space:** $\sum_{u \in U} |N_c(u, D)| = \sum_{u \in D} |N_c(u)| = N_c|D|$.

These two algorithms are two extreme members of the class of $D - oblivious$ algorithms. Each one of them is optimal in one aspect (i.e., space or time), but is inefficient in the other. In the next section we show that this is not a coincidence, and indeed there is a tradeoff between the time and space complexities of $D - oblivious$ algorithms.

3 Lower Bounds for $D - oblivious$ Algorithms

In [DHLNP] a tradeoff for *optimal space* $D -$ *oblivious* AQR algorithms is proved, between the length k of the search sequences and the size of the buckets. It is shown that the size of the largest bucket grows exponentially in n/k, where n is the length of the words. From their results it is possible to show that for any optimal space $D -$ *oblivious* algorithm A, there exist a dictionary D and a query $u \in U$, for which the search time is $TIME(A, D, u) = \Omega(nN_c(u, D))$.
We further strengthen their results and give a series of tradeoffs between the space and time complexities of $D-$*oblivious* AQR algorithms that are not necessarily space optimal.

3.1 Time-Space Tradeoffs

Define for any AQR algorithm A two parameters T_A and S_A, that bound the performance of A compared to that of an optimal space and time AQR algorithm:

$$T_A = \max_{u,D} \frac{TIME(A, D, u)}{|N_c(u, D)| + 1},$$

$$S_A = \max_{D} \frac{SPACE(A, D)}{|D|}.$$

For the rest of this section, assume that $U = \Sigma^n$ (i.e., all words are of length n), and that the Hamming distance is used. Thus, the size of a $c-$*neighborhood* in U is $N_c = \sum_{i=0}^{c} \binom{n}{i} (|\Sigma|-1)^i$. Let $I_k = |N_c(u_1) \cap N_c(u_2)|$

[1] The "Ispell" standard speller of Unix is based on such an algorithm.

be the number of words in the intersection of the $c-neighborhoods$ of two queries $u_1, u_2 \in U$ for which $d(u_1, u_2) = k$. Note that the size of I_k depends only on the distance k between u_1 and u_2, and not on u_1, u_2. A few extreme values of I_k for the Hamming distance are specified bellow (the exact value of I_k for a general k is computed in Appendix A):

- $I_1 = \sum_{i=0}^{c-1} \binom{n-1}{i}(|\Sigma|-1)^i|\Sigma|$.
- $I_{2c} = \binom{2c}{c}$.
- $I_k = 0$ for $k \geq 2c+1$.

The main theorem we prove in this section is:

Theorem 3.1 *Let A be a $D-oblivious$* AQR *algorithm with parameters T_A and S_A.*
If $S_A \leq N_c/2(|\Sigma|^{k-1}+1)$ then $T_A \geq (\sqrt{I_k^2+4N_c} - I_k)/4$ for any $k > 0$ (and vice versa).

Corollary 3.1 *Let A be a $D-oblivious$* AQR *algorithm with parameters T_A and T_S. Then:*

- *If A is space optimal, that is $S_A = O(1)$, then $T_A = \Omega(\sqrt{N_c})$.*
- *If A is time optimal, that is $T_A = O(1)$, then $S_A = \Omega(N_c)$.*

Notice that the space complexity of the optimal time algorithm of Section 2.4 matches this lower bound, while there is a gap of $\sqrt{N_c}$ between the lower bound and the time complexity of the optimal space algorithm presented there.

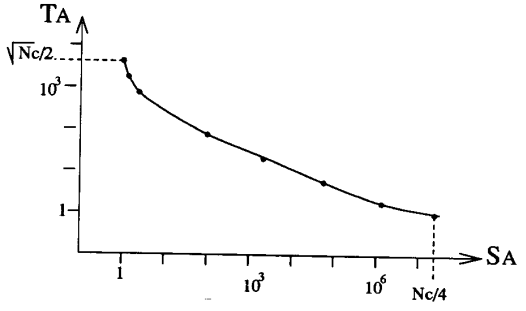


Figure 1: The tradeoff between T_A and S_A for $n = 10$, $\Sigma = 26$, $c = 4$.

The underlying idea of the proof is to show that if the space used is small, then many queries access the same bucket, but each one may find in that bucket many irrelevant words. Thus saving in space results in an increased time complexity. The proof itself is composed of two parts. First, we show that for any $D-oblivious$ algorithm there exists a bucket B, such that $N_c/(2T_A)$ words are mapped to B, and the search sequences of $N_c/(2S_A)$ queries include B. Next, we show a tradeoff between the number of words mapped to a bucket and the number of queries that access that bucket. Then by comparing the quantities, a lower bound on T_A and S_A is derived.
We start with the first part. Let $u \in U$. We will say that u *finds* a word $w \in N_c(u)$ in bucket B, if w is mapped by the function $STORE$ to the bucket B, and the search sequence S_u of u includes the bucket B. The bucket B will be called *big*, if at least $N_c/(2T_A)$ words are mapped to it by the function $STORE$. Otherwise B will be called *small*.

Lemma 3.1 *Let A be a $D-oblivious$ algorithm with parameters T_A and S_A. Then, $\forall u \in U$,*

$$|\{w|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a big bucket}\}| \geq N_c/2.$$

Proof: The length of the search sequence of any query u, satisfies: $|S_u| \leq T_A$. Otherwise, $TIME(A,D,u)/(|N_c(u,D)|+1) > T_A$, for a dictionary D for which $N_c(u,D) = \emptyset$.
Assume the claim is false. Thus, u finds more than $1/2$ of the words of $N_c(u)$ in small buckets. But this implies that the length of the search sequence of u is larger than T_A, in contradiction to the above. □

Lemma 3.2 *Let A be a $D-oblivious$ algorithm with parameters T_A and S_A. Then, $\exists w \in U$,*

$$|\{u|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a big bucket}\}| \geq N_c/2.$$

Proof:

$$|\{(u,w)|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a big bucket}\}|$$
$$= \sum_{u \in U} |\{w|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a big bucket}\}|$$
$$\geq |U|N_c/2,$$

where the last inequality follows from Lemma 3.1. Suppose the claim is false, then

$$|\{(u,w)|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a small bucket}\}|$$
$$= \sum_{w \in U} |\{u|d(u,w) \leq c,\ u \textit{ finds } w \textit{ in a small bucket}\}|$$
$$> |U|N_c/2.$$

But since the total number of pairs is $|\{(u,w)|d(u,w) \leq c\}| = |U|N_c$, a contradiction is derived. □

Corollary 3.2 *Let A be a $D-$ oblivious algorithm with parameters T_A and S_A. Then there exists a bucket B, such that $N_c/(2T_A)$ words are mapped to B, and the search sequences of $N_c/(2S_A)$ queries include B.*

Proof: By Lemma 3.2, there exists a word w, for which at least $1/2$ of the queries $u \in N_c(w)$ find w in a bucket with at least $N_c/(2T_A)$ words.
The word $w \in U$ is mapped to at most S_A buckets. Else, $SPACE(A,D)/|D| > S_A$, for a dictionary D that includes only w.
Therefore by the pigeon hole principle, one of the buckets to which w is mapped, includes at least $N_c/(2T_A)$ words and at least $N_c/(2S_A)$ queries access it. □

We now turn to the second part of the proof. That is, we show a tradeoff between the number of words mapped to a bucket and the number of queries accessing that bucket.

Lemma 3.3 *Let A be a $D-$ oblivious algorithm with parameters T_A and S_A. If $|\Sigma|^{k-1}+1$ queries access a bucket B, then at most $I_k + 2T_A$ words are mapped to B.*

Proof: In any set of $|\Sigma|^{k-1}+1$ words there exist two words u_1, u_2 with distance $d(u_1,u_2) \geq k$. For these two words $|N_c(u_1) \cap N_c(u_2)| \leq I_k$.
It is also easy to see that if A is a $D-$ *oblivious* algorithm with parameter T_A, and u is some query that accesses a bucket B, then at most T_A of the words mapped to B do not belong to $N_c(u)$. Otherwise, let D be a dictionary for which $N_c(u,D) = \emptyset$, but that includes all other words mapped to B. Then, $TIME(A,D,u)/(|N_c(u,D)|+1) > T_A$.
Therefore, at most $I_k + 2T_A$ words are mapped to B. Else, there are more than T_A words that do not belong to $N_c(u_1)$ or to $N_c(u_2)$. □

By combining the two parts we can now prove Theorem 3.1:
Proof of Theorem 3.1:
By Corollary 3.2, there exists a bucket B, such that $N_c/(2T_A)$ words are mapped to B and $N_c/(2S_A)$ queries access B. Hence by Lemma 3.3, if $N_c/(2S_A) \geq |\Sigma|^{k-1}+1$ then $N_c/(2T_A) \leq I_k + 2T_A$. Solving these two inequalities proves the claim. □

4 Algorithms Based on Error Correcting Codes

We present two AQR algorithms $CODE1$ and $CODE2$ that use error correcting codes as a main building block. Both algorithms exploit the structure of code words and spheres around them, in order to map the space U efficiently into buckets. For the rest of this section assume the Hamming distance is used, and all words are of length n.
Let C be an error correcting code over U, where $U = \Sigma^n$. The code C is called an $(n,t,d)-$ *code* if t is the maximum value such that the spheres of radius t around code words are disjoint, and d is the minimum value such that the spheres of radius $t+d$ cover U (see [PW] or [MS] for a comprehensive description of error-correcting codes).
The following AQR algorithms are a generalization of the two basic algorithms described in Section 2.4 (for $t = d = 0$).

4.1 Informal Description of $CODE1$

Let C be an $(n,t,d)-$*code*. There is a bucket B_x for each code word $x \in C$. The function $STORE$ maps a word $w \in D$ to the bucket of the nearest code word.
The function $SEARCH$ associates with each query u a search sequence S_u, that includes all the buckets of code words at distance at most $t+d+c$ from u. The $c-$*neighborhood* of u may have been mapped to these code words (and only to them), since each word is at distance $\leq t+d$ from some code word.

4.2 Informal Description of $CODE2$

Let C be an $(n,t,d)-$*code*. There is a bucket B_x for each code word $x \in C$. The function $STORE$ maps a word $w \in D$ to the buckets of all the code words at distance at most $t+d+c$ from w.
The function $SEARCH$ associates with each query u a search sequence S_u, that includes just the bucket of the nearest code word to u. This bucket includes the $c-$ *neighborhood* of u, by the nature of the function $STORE$.

4.3 Complexity of Algorithms $CODE1$ and $CODE2$

The following lemma will be used in the analysis of the algorithms:

Lemma 4.1 *Let C be an (n,t,d)-code. Then the number of code words at distance at most $t+d+c$ from some word $u \in U$ is $O((\frac{n}{i})^{d+c}(|\Sigma|-1)^{d+c})$.*

Proof: Denote by Z_i the number of code words at distance $t+i$ from u, where $i = 0, 1.., d+c$.
There are $\binom{n}{i}(|\Sigma|-1)^i$ words at distance i from u. Call them the $i-$ *neighbors* of u. Let x be some code word at distance $t+i$ from u. Then, exactly $\binom{t+i}{i}$

of the $i-neighbors$ of u belong to $N_t(x)$. Since the spheres of radius t around code words are disjoint,

$$Z_i \leq \binom{n}{i}(|\Sigma|-1)^i / \binom{t+i}{i} \leq (\frac{n}{t})^i(|\Sigma|-1)^i.$$

Hence the number of code words at distance at most $t+d+c$ from u is:

$$\sum_{i=0}^{d+c} Z_i = O((\frac{n}{t})^{d+c}(|\Sigma|-1)^{d+c}).$$

□

Complexity of Algorithm $CODE1$

The space complexity of algorithm $CODE1$ is optimal, since each word $w \in D$ is mapped to exactly one code word. Thus

$$SPACE(CODE1, D) = |D|.$$

The time needed to answer a query u is composed of two parameters: the number of buckets that are accessed and the number of words found in each bucket. The length of search sequence of u is bounded by the number of code words at distance $\leq t+d+c$ from u. Also, if x is a code word, then the bucket B_x includes at most $|N_{t+d}(x, D)|$ words. Thus by Lemma 4.1:

$$TIME(CODE1, D, u) \leq$$

$$O((\frac{n}{t})^{d+c}(|\Sigma|-1)^{d+c}) + \sum_{\substack{x \in C \\ d(u,x) \leq t+d+c}} |N_{t+d}(x, D)|$$

Complexity of Algorithm $CODE2$

The length of the search sequence of any query u is exactly one, and includes the bucket of the nearest code word to u. Denote this nearest code word by x_u. The words that are mapped to the bucket of x_u are all the words at distance $\leq t+d+c$ from the code word x_u. Hence the time complexity is:

$$TIME(CODE2, D, u) = 1 + |N_{t+d+c}(x_u, D)|.$$

To compute the space complexity we again use Lemma 4.1, since each word $w \in D$ is mapped to all code words at distance $\leq t+d+c$ from it:

$$SPACE(CODE2, D) = O(|D|(\frac{n}{t})^{d+c}(|\Sigma|-1)^{d+c}).$$

Remarks:

- As already stated, each access to a bucket can result in a costly disk access. Thus algorithm $CODE2$ has an advantage in this sense over algorithm $CODE1$, since each query accesses exactly one bucket. The advantage of $CODE2$ grows when the dictionary D is sparse, because then its time complexity decreases. Where as the time complexity of $CODE1$ stays large, since each query accesses a large number of buckets in any case.
- The complexity of the functions $STORE$ and $SEARCH$ of both algorithms depends on the efficiency of decoding the code. Finding the nearest code word to some word, can be done by simply using the decoding function of the code. Again, algorithm $CODE2$ has some advantage over $CODE1$, since the function $SEARCH$ is easier to compute in $CODE2$.

4.4 Choosing a Good Code

The complexity of algorithms $CODE1$ and $CODE2$ depends highly on the parameters t and d of the code. Since it is always possible to find a code with $d \leq t$, we concentrate first on choosing an optimal t. We then describe briefly some bounds on d.

4.4.1 Choosing t

The dictionary D is usually sparse in U, and thus we can assume that most buckets will include significantly less words than were mapped to them. If we assume also that D is uniformly distributed, then we can choose t as to optimize the space and time complexity of the algorithms. The details are omitted and can be found in [H].

4.4.2 Bounds on d

Both algorithms use a general $(n,t,d)-code$, and their efficiency depends also on d. Hence an important question is to try to determine an upper bound on d that would guarantee the existence of an $(n,t,d)-code$, for any given n and t. An upper bound of $d \leq t$ is easy to show, but no better general bound is known. This question is of independent interest to the study of error correcting codes, and some bounds were given for special types of codes (see [CKMS]).
A special subclass of codes are the *perfect codes*. An *(n,t)-perfect code*, is a code for which the spheres of radius t are disjoint and every vector is at distance at most t from some code word (i.e., $d=0$). Perfect codes exist for only very restricted values of n and t.

The Hamming code is a binary perfect code for $t = 1$ and $n = 2^r - 1$. The Golay code is perfect for $n = 23$ and $t = 3$. It was proven that no other interesting perfect codes exist ([V]).
An *(n,t)-quasi perfect code* is a code for which spheres of radius t around code words are disjoint, and every vector is at distance at most $t + 1$ from some code word (i.e., $d = 1$). For $t = 1$ there exists a binary quasi perfect code for any $n = 2^r - 2$; for $t = 2$ there exists a binary quasi perfect code for any $n = 4^r - 1$ (see [GS]). Many other quasi-perfect codes were found (see [MS]).
A list of all known best cods for $n \leq 128$ can be found in [B]. For most practical applications these codes will do.

5 Algorithm *Subwords – Subqueries*

In this section we present an *AQR* algorithm called *Subwords – Subqueries* (*Sub*), which is based on the observation that words are similar if they have a common subword. The space and time complexity of this algorithm depends on the length of the words (and not on Σ), and thus the algorithm is efficient when the words and queries are relatively short. The algorithm can find the neighbors of a given query for either the Levenstein or the Hamming distance.
A word x will be called an $i -$ *subword* of w, if it is possible to derive x from w by i delete operations. Thus of course $d(x, w) \leq i$.

5.1 Informal Description of *Sub*

There is a bucket B_x for each x that is an $i -$ *subword* of some word in D, for $i = 0, 1, ..., c$. The function *STORE* maps a word $w \in D$ to all the buckets of its $i -$ *subwords*, for $i = 0, 1, ..., c$.
The function *SEARCH* associates with each query $u \in U$ a search sequence S_u, that includes the buckets of all the $i -$ *subwords* of u, for $i = 0, 1, ..., c$. The buckets accessed by S_u include all the words at distance at most c delete and c insert steps from u. Thus they also include all words that are at most c exchange or swap steps from u.
If we want to find only neighbors for the Hamming distance, then it is enough to store and search only the $c -$ *subwords* of each word (since two words of the same length at distance at most c from each other, must have a common $c -$ *subword*).

5.2 Complexity of Algorithm *Sub*

Each word $w \in D$ is mapped to the buckets of all its $i -$ *subwords*, for $i = 0, 1, ..., c$. If the length of w is $|w|$, then w has $\sum_{i=0}^{c} \binom{|w|}{i}$ such subwords. Denote by D_j the set of all words in D of length j, and let *max* be the length of the longest word in D. Then the total space used is:

$$SPACE(Sub, D) = \sum_{j=1}^{max} |D_j| \sum_{i=0}^{c} \binom{j}{i} \leq |D| \sum_{i=0}^{c} \binom{max}{i}.$$

Given a query $u \in U$, we search the buckets of all its $i -$ *subwords*, for $i = 0, 1, ..., c$. Thus the length of the search sequence of u is $\sum_{i=0}^{c} \binom{|u|}{i}$. Denote by $\tilde{N}_c(u, D)$ the set of all words in D at distance at most c delete operations and c insert operations from u. The buckets searched include only words from $\tilde{N}_c(u, D)$ (at most c delete steps to get from u to the subword x that represents a given bucket B_x, and at most c insert steps to get from x to another word in B_x). Hence:

$$TIME(Sub, D, u) < \sum_{i=0}^{c} \binom{|u|}{i} + |\tilde{N}_c(u, D)| \sum_{i=0}^{c} \binom{|u|}{i}.$$

The analysis for the Hamming distance is similar.

6 Algorithm $k -$ *Partition*

We now present an *AQR* algorithm called $k -$ *Partition* (*Part*), which is based on ideas similar to algorithm *Sub*, that is, storing and searching subwords. The worst time and space complexity of this algorithm may be as bad as that of the algorithm *Sub*, but in most practical applications its performance is much better.
The basic idea is to partition each word and query into k parts, using some fixed partition P, where $k > c$. Any two words u, w with distance $d(u, w) \leq c$, have at least $k - c$ identical parts, and can be compared using these parts. The partition P will be called a *k-partition*. A word x will be called a $(P, k, c) -$ *subword* of w, if it is composed of **exactly** $k - c$ parts of w, under the partition P. We first show how to find neighbors of Hamming distance at most c from a given query, and then generalize the algorithm for the Levenstein distance (see Appendix B).

6.1 Informal Description of *Part* for the Hamming Distance

Denote by *max* the length of the longest word in D. For all the words of length i ($i = 1, 2, ..., max$) choose a fixed partition P_i that partitions words of length i into k parts. Any partition into k parts is allowed; that is, each part can include any set of letters (not necessarily consecutive) but the parts must be disjoint. Define a bucket B_x for each word x that is a $(P_i, k, c) -$ *subword* of some word of length i in D.
The function *STORE* maps a word $w \in D$ to all the buckets of its $(P_{|w|}, k, c) -$ *subwords*.

The function $SEARCH$ associates with each query $u \in U$ a search sequence S_u that includes the buckets of all the $(P_{|u|}, k, c) - subwords$ of u.

6.2 Complexity of Algorithm *Part*

Each word $w \in D$ is mapped to the buckets of all its $(P, k, c) - subwords$, for some partition P that partitions w into k parts. The number of $(P, k, c) - subwords$ is exactly $\binom{k}{c}$ (i.e., all the ways to choose $k - c$ parts from a total of k parts). Therefore:

$$SPACE(Part, D) = \binom{k}{c}|D|.$$

The length of the search sequence of any query u is $\binom{k}{c}$ for the Hamming distance, and slightly more for the Levenstein distance. The problem is to determine the total number of words in the buckets accessed by the search sequence of u, and the number of words in these buckets that do not belong to $N_c(u, D)$.

The performance of the algorithm can be improved significantly by choosing a partition that does not create large buckets. If large buckets do occur, then it is possible to use a second partition to distribute the words in them to sub-buckets. This process can continue until every bucket contains only a constant number of words. Recall that these computations can be done during the preprocessing stage of D.

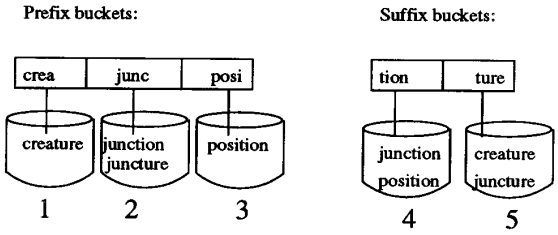
In Section 7 it is shown that for the English dictionary, there exists a partition for which the average bucket includes only a constant number of words. Thus the average time complexity for the English dictionary is $TIME(Part, D, u) = O(\binom{k}{c} + |N_c(u, D)|)$. However, there are dictionaries for which any partition creates large buckets (e.g., a dictionary that includes all the words in some large sphere). It is an open problem to determine for what dictionaries there exists a good partition.

6.3 Examples of $k - partitions$

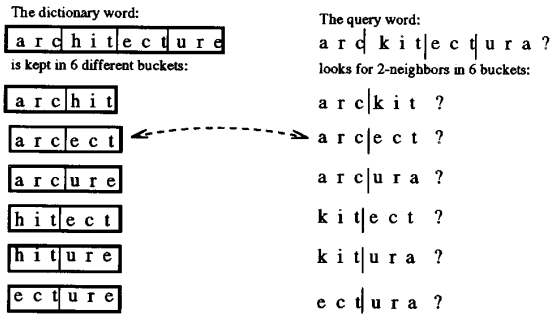
The simplest possible 2 - *partition* splits each word to a prefix and suffix of equal length. Such a partition can find neighbors at distance at most one from a given query. Consider the following dictionary and the set of buckets created for it:

creature	junction	juncture	position

The query "pisition" will be searched by its prefix and suffix. Its prefix, "pisi", does not match any of the prefix-buckets, but its suffix "tion", matches bucket 4. Therefore, we search bucket 4 for neighbors and find the word "position".



Now consider a partition to $k = 4$ parts, such that each subword includes 2 parts. Such a partition can find neighbors at distance at most 2 from a given query, as shown in the following figure:



7 Comparison of Algorithms

In this section we give a short comparison of the performance of three AQR algorithms on the Webster English dictionary (i.e., $|\Sigma| = 26$). The Webster dictionary includes 234,936 words. The algorithms checked were: "Ispell" , *Sub* and *Part*. All three algorithms were supposed to find all the neighbors at distance at most one (i.e., $c = 1$) from a given query. The space used by each algorithm is specified in Table 1. "Ispell" uses an algorithm similar to the Optimal space algorithm described in Section 2.3, and thus uses optimal space.

Algorithm *Sub* stores a word of length j in j buckets for the Hamming distance (the buckets of all its $1 - subwords$), and $j + 1$ buckets for the Levenstein distance (the buckets of all its $1 - subwords$ and the bucket of the word itself). Denote by D_j the set of English words of length j, and by max the length of the longest English word. Then the total space used by *Sub* is $\sum_{j=1}^{max} j|D_j|$ for the Hamming distance and $|D| + \sum_{j=1}^{max} j|D_j|$ for the Levenstein distance (see Section 5.2).

The partition used for algorithm *Part* was a partition into $k = c + 1 = 2$ parts. Therefore each word was stored in 2 buckets, thus using a total space of $2|D|$.

In order to compute the time complexity, we checked for each algorithm how many words were mapped to each bucket (worst and average case), and how many buckets each algorithm accessed in order to answer a query.

Algorithm	Space used for Hamming Dist.	Space used for Levenstein Dist.
Ispell	$\|D\|$	$\|D\|$
Sub	$6.96\|D\|$	$7.96\|D\|$
Part	$2\|D\|$	$2\|D\|$

Table 1: Space used for the English dictionary for $c = 1$, where $|D| = 234,936$ words.

Algorithm	# buckets accessed	largest bucket	average bucket	average time
Ispell	226	1	1	226
Sub	9	7	1.06	18.54
Part	2	27	1.45	4.90

Table 2: Performance results for the Hamming distance, for queries of length $n = 9$ and for $c = 1$.

Algorithm	# buckets accessed	largest bucket	average bucket	average time
Ispell	495	1	1	495
Sub	10	7	1.06	20.66
Part	6	312	2.96	23.76

Table 3: Performance results for the Levenstein distance, for queries of length $n = 9$ and for $c = 1$.

The average time needed to answer a query was then computed by adding the number of buckets accessed (i.e., the length of the search sequence) to the average number of words found in all buckets accessed. These results are described in Tables 1 and 2. Note that all the units in the tables are units of words.

The length of the search sequence of "Ispell" for a given query u is $|N_c(u)|$. Thus for $c = 1$, the length of the search sequence is $1 + n(|\Sigma| - 1)$ for the Hamming distance, and $1 + n(|\Sigma| - 1) + n + (n + 1)|\Sigma|$ for the Levenstein Distance. We used the length of the search sequence as a lower bound on the time needed by "Ispell" to answer a query; although in practice the time complexity may be even larger when the algorithm actually finds neighbors and has to read them.

The length of the search sequence of algorithm *Part* for $c = 1, k = 2$, is 2 for the Hamming distance, and 6 for the Levenstein distance. The best partition into 2 parts that was found for words of length 9 is: the 1st, 3rd, 6th, 8th letters in one part, and the 2nd, 4th, 5th, 7th, 9th letters in the second part. For the Levenstein distance, each word was partitioned to a prefix of length 4 and a suffix of length 5. Note that the results for *Part* are better for the Hamming distance, when it is possible to choose the partition.

The length of the search sequence of algorithm *Sub* for queries of length j, is j for the Hamming distance and $j + 1$ for the Levenstein distance.

In general, the results show that for the Hamming distance and $c = 1$, "Ispell" uses space $|D|$ and needs $\Omega(n|\Sigma|)$ time to answer a query. Algorithm *Sub* uses space $n|D|$ and average time $O(n)$, while algorithm *Part* uses space $2|D|$ and average time $O(1)$. For the Levenstein distance algorithm *Sub* is slightly faster than *Part*, and both are superior to "Ispell".

References

[B] A.Brouwer. Best Known Codes for $n \leq 128$. *To be published in IEEE Transactions on Inf.*, 1993.

[CKMS] G.D. Cohen, M.G. Karpovsky, H.F. Mattson and J.R. Schatz. Covering Radius - Survey and Recent Results. *IEEE Transactions on Information Theory,* Vol 31, No. 3, 1985.

[DHLNP] D. Dolev, Y. Harari, N. Linial, N. Nisan and M. Parnas. On Neighborhood Preserving Hashing. *Technical Report 92-31. Hebrew University.*

[DKMMRT] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert and R.E. Tarjan. Dynamic Perfect Hashing - Upper and Lower Bounds. *29th Symposium on Foundations of Computer Science,* 1988.

[FKS] M.L. Fredman, J. Komlos, E. Szemeredi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *J.ACM 31*, 1984, pp. 538-544.

[FS] M.L. Fredman and M.E. Saks. The Cell Probe Complexity of Dynamic Data Structures. *21st Annual ACM Symposium on Theory of Computing,* 1989.

[GS] J.M. Goethals and S.L. Snover. Nearly Perfect Binary Codes. In *Discrete Mathematics 3*, 1972, pp. 65-88.

[H] Y. Harari. Algorithms and Lower Bounds for Retrieval of Neighbors from a Dictionary. *M.Sc. thesis. Hebrew University, 1992. In Hebrew.*

[L] K. Lindstrom. The Nonexistence of Unknown Nearly Perfect Codes. *Ann. Univ. Turku., Ser. A, No. 169,* 1975, pp. 3-28.

[LV1] G.M. Landau and U. Vishkin. Efficient String Matching in the Presence of Errors. *Proc. 26th IEEE Symposium on Foundations of Computer Science,* 1985.

[LV2] G.M. Landau and U. Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm. *Proc. 18th Annual ACM Symposium on Theory of Computing,* 1986.

[MS] F.J. MacWilliams and N.J.A. Sloane. The Theory of Error Correcting Codes. *North-Holland Publishing Company,* 1977.

[PW] W.W. Peterson and E.J. Weldon. Error Correcting Codes. *MIT Press,* 1972.

[SK] D. Sankoff and J.B. Kruskal. Time Warps, Strings Edits and Macromolecules: The Theory and Practice of Sequence Comparison. *Addison-Wesley Publishing Company, Inc.*, 1983.

[V] J.H. Van Lint. A Survey of Perfect Codes. *Rocky Mountain Journal of Mathematics, Vol. 3, No. 2,* 1975, pp. 199-224.

A Computing I_k

Lemma A.1

$$I_k = \sum_{i=0}^{c}\sum_{s=0}^{i}\sum_{j=s}^{c}\binom{n-k}{s}(|\Sigma|-1)^s \times\binom{k}{k-i+s}\binom{i-s}{i+j-k-2s}(|\Sigma|-2)^{i+j-k-2s}$$

Proof: Let x and y be two words in U with Hamming distance $d(x,y) = k$. We want to find the number I_k of words in the intersection of their $c-$*neighborhoods*. We will do so by computing the number of words at distance i from x and j from y, and summing for all possible pairs i and j.
So let w be a word such that $d(x,w) = i$ and $d(y,w) = j$. The words x and y agree on $n-k$ coordinates and disagree on k. It is possible to construct w in the following way:
Choose s coordinates from the $n-k$ coordinates in which x and y agree and change them, and let the other $n-k-s$ coordinates of w be as they are in x and y. This can be done in $\binom{n-k}{s}(|\Sigma|-1)^s$ ways. From the k coordinates that x and y disagree on, choose $k-i+s$ to be as x and $k-j+s$ to be as y. The rest $i+j-2s-k$ coordinates of w will be different from x and y. This can be done in $\binom{k}{k-i+s}\binom{i-s}{i+j-k-2s}(|\Sigma|-2)^{i+j-k-2s}$ ways.
Putting all this together, and letting i,j and s range over the correct domain, proves the claim. □

B Algorithm *Part* for the Levenstein Distance

In a similar way to the algorithm for the Hamming distance, choose partitions P_i, for $i = 1,2,...,max$. Now, the only partitions allowed, are those that partition the words into k parts, such that each part is composed of a sequence of consecutive letters. For example, if we partition the words into $k = 2$ parts, then each word must be partitioned into a suffix and prefix (not necessarily of the same length).
The function *STORE* maps a word $w \in D$ to all the buckets of its $(P_{|w|}, k, c) -$ *subwords*.
Answering a query u is slightly more complex. We demonstrate the search procedure for $c = 1$, that is neighbors of distance at most 1 from u. The generalization to a larger c is straight forward.
Given a query u of length n, the possible neighbors of u in D are of length $n-1, n, n+1$ (for $c = 1$). Neighbors of length n, are at Hamming distance 1 from u. Thus using the partition P_n on u, it is possible to search for all these neighbors as described in Section 6.1.
A possible neighbor v of length $n+1$ has one extra letter in one of its k parts. Assume this extra letter is in part j. Construct the $(P_{n+1}, k, c) -$ *subword* of u that does not include part j, as follows:
Parts $1,2,...,j-1$ of u will be computed using partition P_{n+1}. Part j of u will be one letter shorter than it is in partition P_{n+1} and will be omitted. Parts $j+1, j+2, ..., k$ of u will thus be shifted one letter to the left, and computed according to partition P_{n+1}. Then v can be found in the bucket of this subword. This process is repeated for $j = 1,...,k$, and then a similar search is carried out for neighbors of length $n-1$.