Recognizing Acronyms and their Definitions

Kazem Taghva*and Jeff Gilbreth
Technical Report 95-03
Information Science Research Institute
University of Nevada, Las Vegas

June 1995

Abstract

This paper introduces an automatic method for finding acronyms and their definitions in free text. The method is based on an inexact pattern matching algorithm applied to text surrounding the possible acronym. Evaluation shows both high recall and precision for a set of documents randomly selected from a larger set of full text documents.

*Email: taghva@isri.unlv.edu.

1 Introduction

Our interest in acronyms started with the development of a post processing system (PPS) for the improvement of text output from optical character recognition (OCR) devices[12]. Originally, acronyms were a nuisance–words that almost never appeared in dictionaries but, of course, were known to be valid strings. The most fundamental part of the PPS involved finding and correcting misrecognized words. So our first acronym finder removed these words from the text to alleviate erroneous clustering and correction by the PPS.

Recently, as a part of our research on the issues associated with retrieval from OCR text[14, 13, 15], we observed that OCR devices generally have lower accuracy rates for certain groups of words such as proper names. This is due to the fact that these devices rely heavily on the use of lexicons and statistical distribution of character n-grams. Unfortunately, these groups of special words are also identified as having higher retrieval value[10, 9, 4].

There are many procedures to extract features from documents in order to populate databases[10, 2, 1]. Since the acronyms are found in the text with their definitions, the probability that they are correct is quite high; they can be used to build a database of acronyms automatically and locate instances of these acronyms in the current document or other documents. These extracted features can also be used to enhance retrieval and/or identify associations and relationships to be used for a hypertext browsing system.

In a hypertext context, acronyms can be used to link documents which are related to each other. These links can be used to identify all documents written on a specific project or about a particular government agency. Furthermore, since government documents contain a large number of acronyms, a useful tool for the reader would be a routine that can provide acronym definitions immediately. This routine would consist of clicking the mouse if the links between acronyms and their definitions exist.

The program that recognizes acronyms and acronym definitions does not rely on a lexicon for validation of words (except for the short list of stopwords). This means that the spelling of a word is of little concern to the acronym finder. Most modern OCR devices are especially good at correctly recognizing most common words[11] anyway, so misspelled stopwords are not a major concern.

2 Background

When the project started, we were building a system designed to filter out garbage from error-prone OCR output. Our system for identifying acronyms in a set of terms was fairly primitive. It was at this time that we decided to look at `FERRET`, a text skimming system by Mauldin[9]. The `FERRET` system used complex lexical analysis to tokenize special words, quantities, dates, and other textual objects. This system influenced us to build a simple parser to identify acronyms in free text. While `FERRET` uses `Lex`[8] for its implementation, our *acronym finding program*, `AFP`, was designed specifically for finding acronyms.

Our next influence would have to be the company name recognition work by Rau[10]. Upon seeing the various methods and approaches applied to the recognition of company names, we tried some proper name and acronym parsing using some of these methods. The company name variation scheme involved the generation of acronyms based upon a previously extracted company name, in order to find alternative name forms. Considering the process in reverse, we surmised that one could use a candidate acronym to find a plausible definition. If found, we could be more certain that the candidate was indeed an acronym. As a side effect of this process, we would have a list of acronyms and their definitions.

It was also the company name project that inspired us to deal with stopwords in an intelligent way. Stopwords are words that have high frequency in documents, but have low retrieval value (e.g., “the”, “a”, “and”, “or”, “in”). Stopwords are normally ignored in retrieval applications, but we found that they could not be ignored in AFP. By examining the approaches taken for recognizing company names[10], we found a solution for handling stopwords in acronyms.

3 Definition of an Acronym

Webster’s 7th Dictionary defines “acronym” as:

> a word (as radar or snafu) formed from the initial letter or letters of each of the successive parts or major parts of a compound term.

Our working definition of an acronym candidate, however, is simply an upper-cased word from 3 to 10 characters in length. This is straightforward except for the length restriction. The lower bound is a compromise between recall (acronyms of 2 characters do exist) and precision (approximate matching on anything less than 3 characters is very error prone). The upper bound is an arbitrary but reasonable assumption. Acronyms longer than 10 characters are quite rare.

4 Outline of the Acronym-Definition Finding Program

The program consists of four phases: initialization, input filtering, parsing the remaining input into words, and the application of the acronym algorithm.

4.1 Initialization

The input for the algorithm is composed of several lists of words, with the text of the document as the final input stream. These inputs are:

1. A list of *stopwords*—commonplace words that are often insignificant parts of an acronym (e.g., “the”, “and”, “of”). It is important to distinguish these stopwords from regular words for the algorithm to make good matches with the definitions. This list is required.

2. A list of *reject* words—words that are frequent in the document, or in general, but are known not to be acronyms (e.g., “TABLE”, “FIGURE”, Roman Numerals). The fewer acronym candidates there are, the more efficient the program, and in turn, the fewer *coincidental matches*. This list is optional.

3. A database of acronyms and their accompanying definitions. This information can be used to either override the program’s searching routine or as a fall-back mechanism when a search is fruitless. We did not use any databases in the evaluation of the program. This database is optional.

4. The text of the document (or collection) to be searched.

4.2 Filtering the input

The input is pre-processed to disregard lines of text that are all uppercase (e.g., titles and headings). Upon identifying an acronym candidate, the reject word list is consulted before subsequent processing. If the candidate does not appear in the reject list, then an appropriate *text window*[2] surrounding the acronym is searched for its definition. The text window is divided into two subwindows, the *pre-window* and the *post-window*. Each subwindow's length in words is set to twice the number of characters in the acronym.

4.3 Word parsing

In order for this algorithm to find a reasonable number of acronym definitions, a precedence has to be assigned to different types of words. Currently, these types are limited to (1) stopwords, (2) hyphenated words, (3) acronyms themselves, and (4) ordinary words that do not fall into any of the above categories. The following gives the philosophy behind categorizing the words into types.

Stopwords — Normally ignored in traditional text retrieval applications, stopwords cannot be eliminated from the definition search process. If the algorithm ignores stopwords completely, many acronyms are not found. Similarly, if stopwords are not ignored, many acronyms will not be correctly identified. Precedence of non-stopwords over stopwords in the matching process helps resolve these problems. For example,

`stopwords must be counted`	`Department of Energy (DOE)`
	`as low as reasonably achievable (ALARA)`
`stopwords must be ignored`	`Office of Nuclear Waste Isolation (ONWI)`

Hyphenated Words — Hyphenated words are treated as a special case. Acronym definitions often contain hyphenated words in which either the first, or all of the word parts of the hyphenated word correspond to letters of the acronym. Both cases must be checked to find the best match. For example,

`first word part matches`	`X-ray photoelectron spectroscopy (XPS)`
	`high temperature gas-cooled reactor (HTGR)`
`all word parts match`	`non-high-level solid waste (NHLSW)`
	`June-July-August (JJA)`

Acronyms — Acronyms sometimes occur within short word distances of each other. Since acronyms sometimes include other acronyms in their definitions, we don't want to abort processing if this situation occurs. What we can do is to abort processing if the acronym encountered is the same as the one we are trying to define. For example,

what we want to find:

```
    ARINC Communications and Reporting System (ACARS)
```

what we don't want to find:

```
with SIMS. In most cases, separate SIMS profiles were
```

Normal Words — Words that don't fall into any of the above categories are considered normal words. These words make up the majority of the words in acronym definitions and require no special handling.

When a subwindow is parsed, we generate two symbolic arrays for that window: the *leader* array, consisting of the first letter of each word, and the *type* array, consisting of the type of each word in the subwindow. For simplicity, we use the characters `s, H, h, a,` and `w` to denote stopwords, the initial part of hyphenated words, following parts of hyphenated words, acronyms, and normal words, respectively. These abstractions simplify the main engine since it becomes unnecessary to scan the text strings. We can systematically search through the text windows for matches of the first letters of words and the acronym letters.

Example 1

Given the text:

```
spent fuel and recycling the recovered uranium and
plutonium results in the generation of transuranic
(TRU) non-high-level solid waste (NHLSW). Volumes
and characteristics of these wastes, and methods for
```

the pre-window for the acronym `NHLSW` is:

```
[results in the generation of transuranic (TRU)
non-high-level solid waste]
```

The leader and type arrays are:

```
[r i t g o t t n h l s w] leaders
[w s s w s w a H h h w w] types
```

4.4 Applying the algorithm

The algorithm identifies a common subsequence of the letters of the acronym and the leader array to find a probable definition. Following [3], A *subsequence* of a given sequence is just the given sequence with some elements removed. For two sequences X and Y, we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y. For example, if $X = acbceac$ and $Y = cebaca$, then *cba* is a common subsequence of X and Y of length 3. Observe that *ceac* and *cbca* are also common subsequences of X and Y (length 4), and there are no common subsequences of length greater than 4 (i.e., *ceac* is a common subsequence of maximum length). The *longest common subsequence* (LCS) of any two strings X and Y is a common subsequence with the maximum length among all common subsequences. We also want to point out that LCS *ceac* can be generated from X by indices [2, 5, 6, 7] or indices [4, 5, 6, 7]. The need for this distinction will be apparent shortly.

There are well known and efficient algorithms to find an LCS of two sequences[3][7]. Most of these algorithms only find one LCS. To fully explain `AFP`, we first introduce the

LCS algorithm as described in [3], then we present an algorithm to generate all possible LCS's. Finally, we give our algorithm to locate the acronym definition.

We use the notation $X[1 \ldots i]$ to denote the prefix of length i in the string $X[1 \ldots m]$. Now, for two strings $X[1 \ldots m]$ and $Y[1 \ldots n]$, let $c[i,j]$ be the length of an LCS of the sequences $X[1 \ldots i]$ and $Y[1 \ldots j]$. We observe that when either X or Y are empty sequences, then the LCS is an empty string and $c[i,j] = 0$. We also know that $c[i,j]$ can be obtained from the following recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } X_i = Y_j \\ max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } X_i \neq Y_j \end{cases} \quad (1)$$

This recursive equation states that in order to compute the LCS of X and Y (in notation $LCS(X,Y)$), we should test to see if $X[m] = Y[n]$. In case the equality holds, calculate $LCS(X[1 \ldots m-1], Y[1 \ldots n-1])$, otherwise choose the larger of $LCS(X[1 \ldots m], Y[1 \ldots n-1])$ and $LCS(X[1 \ldots m-1], Y[1 \ldots n])$.

```
*build-LCS-matrix*(X, Y)

m $\leftarrow$ *length*[X];
n $\leftarrow$ *length*[Y];
**for** i $\leftarrow$ 1 **to** m **do**
    c[i, 0] $\leftarrow$ 0;
**for** j $\leftarrow$ 1 **to** n **do**
    c[0, j] $\leftarrow$ 0;
**for** i $\leftarrow$ 1 **to** m **do**
    **for** j $\leftarrow$ 1 **to** n **do**
        **if** X[i] = Y[j] **then**
            c[i, j] $\leftarrow$ c[i−1, j−1] + 1;
            b[i, j] $\leftarrow$ "$\nwarrow$";
        **else if** c[i−1, j] $\geq$ c[i, j−1] **then**
            c[i, j] $\leftarrow$ c[i−1, j];
            b[i, j] $\leftarrow$ "$\uparrow$";
        **else**
            c[i, j] $\leftarrow$ c[i, j−1];
            b[i, j] $\leftarrow$ "$\leftarrow$";

**return** c and b;
```

Figure 1: The `build-LCS-matrix` routine.

Figure 1 shows a dynamic programming algorithm [3] of the recursive equation 1. The algorithm computes the length of an LCS for strings X and Y and stores this value in $c[m,n]$. If this LCS length falls below the confidence level threshold, no further processing for this acronym will be done. The calculation of this confidence level will be explained in more detail in section 4.4.1. The LCS construction method in [3] utilizes the matrix b to show the path from which an LCS can be constructed.

A "↖" entry in $b[i,j]$ asserts that $X[i] = Y[j]$, and $c[i-1, j-1] + 1$ is the selected value in equation 1. A "↑" or "←" in $b[i,j]$ asserts that $X[i] \neq Y[j]$, and $c[i-1,j]$ or $c[i,j-1]$ is the selected value in equation 1, respectively.

Example 2

Consider the following text:

```
This work was conducted as part of the Department
of Energy's (DOE) National Waste Terminal Storage
program under the management of the Office of
Nuclear Waste Isolation (ONWI). A primary objective
of the program is to develop and demonstrate the
technology for safe disposal of nuclear waste
including spent commercial reactor fuel.
```

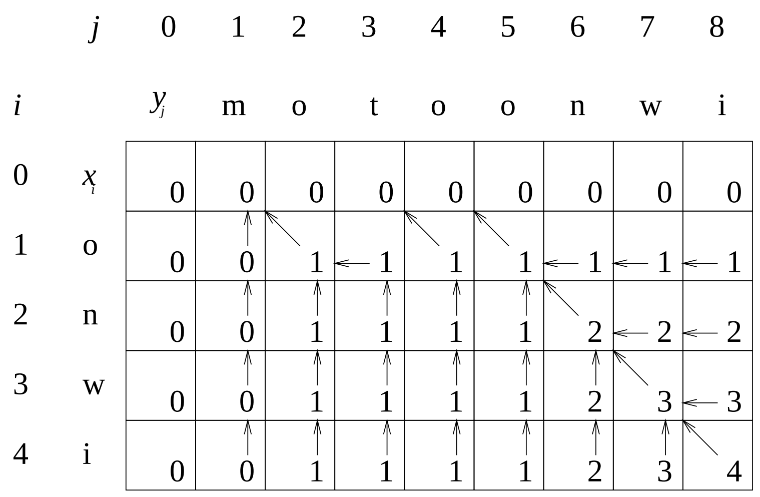
the pre-window for the acronym `ONWI` is:

```
[management of the Office of Nuclear Waste
Isolation]
```

the leader and type arrays are:

```
[ m o t o o n w i ] leaders
[ w s s w s w w w ] types
```

Then `build-LCS-matrix`("onwi", "motoonwi") will produce the b and c matrices in Figure 2. Matrix b is superimposed over c to show their relationship. The length of LCS("onwi", "motoonwi") is 4.



	j	0	1	2	3	4	5	6	7	8
i		y_j	m	o	t	o	o	n	w	i
0	x_i	0	0	0	0	0	0	0	0	0
1	o	0	↑ 0	↖ 1	← 1	↖ 1	↖ 1	← 1	← 1	← 1
2	n	0	↑ 0	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	← 2	← 2
3	w	0	↑ 0	↑ 1	↑ 1	↑ 1	↑ 1	↑ 2	↖ 3	← 3
4	i	0	↑ 0	↑ 1	↑ 1	↑ 1	↑ 1	↑ 2	↑ 3	↖ 4

Figure 2: The c and b matrices computed by `build-LCS-matrix` on $X = onwi$ and $Y = motoonwi$.

```
parse-LCS-matrix(b, start_i, start_j, m, n, lcs_length, Stack, Vectorlist)

1   for i ← start_i to m do
2       for j ← start_j to n do
3           if b[i, j] = "↖" then
4               s ← build-stack(i, j);
5               push(Stack, s);
6               if lcs_length = 1 then
7                   vector ← build-vector(Stack, n);
8                   add(Vectorlist, vector);
9               else
10                  parse-LCS-matrix(b, i+1, j+1, m, n, lcs_length−1, Stack, Vectorlist);
11              pop(Stack);
12  return;
```

```
build-vector(Stack, n)

1   v ← allocate-vector(n);
2   for j ← 1 to n do
3       v[j] ← 0;
4   s ← Stack;
5   while s ≠ NIL do
6       v[j[s]] ← i[s];
7       s ← next[s];
8   return v;
```

Figure 3: The `parse-LCS-matrix` and `build-vector` routines.

The matrix b is used to construct an LCS by starting from the lower right-hand corner; each "↖" corresponds to an entry where $X[i] = Y[j]$. The LCS construction method used in [3] only finds one LCS. For `AFP`, we are interested in *all* ordered arrangements of indices leading to an LCS. We developed the procedures `parse-LCS-matrix` and `build-vector` in Figure 3 to accomplish this goal. Let $b[i, j]$ be an entry in the matrix b with value "↖", then the procedure limits its search to the sub-matrix $b[i+1 \ldots m, j+1 \ldots n]$ to build the rest of the LCS. The procedure uses a stack to store the partial sequences leading to the LCS. Finally, the procedure uses the indices of the LCS to construct a vector representation of a possible definition for the acronym.

Earlier we showed that `LCS`("onwi", "motoonwi") was found to be of length 4. The `parse-LCS-matrix` routine will produce the following ordered lists of indices (or equivalently, the stacks built by this routine):

(1,2), (2,6), (3,7), (4,8)
(1,4), (2,6), (3,7), (4,8)
(1,5), (2,6), (3,7), (4,8)

The notation (i, j) indicates that the jth leader entry matches the ith letter of the acronym. The `build-vector` routine creates the vectors by setting the jth entry to the

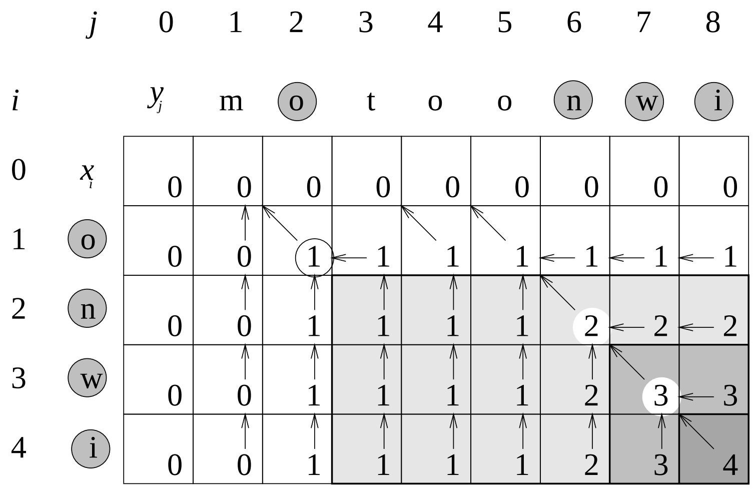


Figure 4: The parsing of the c and b matrices by `parse-LCS-matrix`. To reconstruct the elements of all LCS's, start at the upper left-hand corner; searching for the first (i, j) such that $X[i] = Y[j]$, indicated by an "$\nwarrow$" entry in the matrix. When a matching (i, j) is found, recursion is used to parse the sub-matrix $b[i+1 \ldots m, j+1 \ldots n]$ (shaded). Every matching is processed in this way; increased shading is used to illustrate the recursive processing of sub-matrices.

value i for all (i, j) entries in the stack, with the remaining entries set to 0. For this example, the corresponding vectors are:

```
[0 1 0 0 0 2 3 4]
[0 0 0 1 0 2 3 4]
[0 0 0 0 1 2 3 4]
```

Referring back to our leader array in example 2, the second vector indicates that for acronym `ONWI`, the letters *o, n, w,* and *i* occur as the leaders of the 4th, 6th, 7th and 8th words in the pre-window.

The last part of the algorithm deals with selecting the appropriate definition for the acronym from the vectors generated by `parse-LCS-matrix`. The procedure `vector-values`(V) in Figure 5 calculates the following four values for each vector:

1. *misses*$[V]$:
   The number of zero entries in the vector; disregarding leading zeros, trailing zeros, and those zero entries corresponding to words of types `s` or `h`. Gives the number of words in the definition that do not match a letter of the acronym.

2. *stopcount*$[V]$:
   The number of stopwords that will be used in the acronym definition if the vector is selected.

3. *distance*$[V]$:
   The index of the last non-zero entry. This value measures the proximity of the definition to the actual acronym.

4. *size*$[V]$:
   The number of entries in the vector after removing leading and trailing zeros. This value represents the length of the definition in words.

Finally, the procedure `compare-vectors`(A, B) will choose one of two input vectors by comparing the vector values of A with the vector values of B. The procedure chooses a vector by priority processing. If all conditions fail to resolve the comparison, the procedure will return vector A. In practice, this situation is rare (we have not seen one). The following type array and vectors are constructed artificially to illustrate that this last case *can* occur:

```
[w H h w H h w w s] types

[0 1 2 0 3 0 4 5 0] vector A
[0 1 0 2 3 4 0 5 0] vector B
```

Example 3

Recall that in example 2, the `parse-LCS-matrix` routine generated the following vectors:

```
[0 1 0 0 0 2 3 4] vector A
[0 0 0 1 0 2 3 4] vector B
[0 0 0 0 1 2 3 4] vector C
```

The values calculated by the `vector-values` routine are as follows:

	A	B	C
misses	1	0	0
stopcount	1	0	1
distance	0	0	0
size	7	5	4

The call `compare-vectors`(A, B) will return B, since *misses*$[A]$ $>$ *misses*$[B]$. The call `compare-vectors`(B, C) will return B since *stopcount*$[B]$ $<$ *stopcount*$[C]$. Therefore, vector B is chosen, producing the definition:

"Office of Nevada Waste Investigations".

4.4.1 Confidence Level

Once the length of the $LCS(acronym, leaders)$ is known (found in `build-LCS-matrix`), the next step in the definition searching process is to compute the confidence level of the current acronym candidate. The confidence level is simply:

$$\frac{\text{length of LCS}}{\text{\# of acronym letters}} + (\textit{error percentage})$$

```
vector-values(V)

i ← 1;
while i < length[V] and V[i] = 0 do
    i ← i + 1;
first ← i;
i ← length[V];
while i > 0 and V[i] = 0 do
    i ← i − 1;
last ← i;
size[V] ← last − first;
distance[V] ← length[V] − last;
for i ← first to last do
    if V[i] > 0 and types[i] = 's' then
        stopcount[V] ← stopcount[V] + 1;
    else if V[i] = 0 and types[i] ≠ 's' and types[i] ≠ 'h' then
        misses[V] ← misses[V] + 1;
```

```
compare-vectors(A, B)

vector-values(A);
vector-values(B);

if misses[A] > misses[B] then
    return (B);
else if misses[A] < misses[B] then
    return (A);
if stopcount[A] > stopcount[B] then
    return (B);
else if stopcount[A] < stopcount[B] then
    return (A);
if distance[A] > distance[B] then
    return (B);
else if distance[A] < distance[B] then
    return (A);
if size[A] > size[B] then
    return (B);
else if size[A] < size[B] then
    return (A);
return (A);
```

Figure 5: The `vector-values` and `compare-vectors` routines.

where the error percentage is configurable at runtime (20% by default). If the confidence level is greater than or equal to one, the algorithm continues with `parse-LCS-matrix`. If the confidence level is less than one, the search is abandoned since there is not an adequate correlation between the text window and the letters in the acronym (i.e. there probably isn't a definition to be found).

An exact matching algorithm is less error-prone, but allowing limited misses in definitions compensates for some of the more creative and unusual acronym definitions:

```
Northeast Utilities Service Company (NUSCO)
Intergranular stress-corrosion cracking (IGSCC)
Superconduction Quantum Interference Device (SQUID)
independent interim plutonium oxide storage facility (IIPSF)
```

5 More Examples

Example 4

Given the following text:

```
These costs also include the effect of additions
to utility supplies such as electrical substation;
heating, ventilating, and air conditioning (HVAC);
compressed air; and similar auxiliaries at the
FRP; as well as the cable, piping, and other bulk
materials incorporated directly into the FRVSF.
```

the pre-window for the acronym `HVAC` is:

```
[as electrical substation; heating, ventilating, and
air conditioning]
```

the leader and type arrays are:

`[a e s h v a a c]` leaders
`[s w w w w s w w]` types

producing two LCS's with the following vector representations:

`[0 0 0 1 2 3 0 4]` vector A
`[0 0 0 1 2 0 3 4]` vector B

Calculating the vector values, we get:

	A	B
misses	1	0
stopcount	1	0
distance	0	0
size	5	5

Vector B will be chosen, since $misses[A]$ ¿ $misses[B]$.

Example 5

Given the following text:

```
Threat scores produced by NMC's operational regional
model (the Limited area Fine-mesh Model, or LFM)
for 0.25 mm of precipitation in the 12-24h forecast
period are considerably higher (averaging 0̃.40) and
have shown a slight increase since 1976 (Fig.  4).
```

the pre-window for the acronym `LFM` is:

```
[(the Limited area Fine-mesh Model, or]
```

the leader and type arrays are:

`[t l a f m m o]` leaders
`[s w w H h w s]` types

LCS vectors:

`[0 1 0 2 3 0 0]` vector A
`[0 1 0 2 0 3 0]` vector B

Calculating the vector values, we get:

	A	B
misses	1	1
stopcount	0	0
distance	2	1
size	4	5

Vector B will be chosen since $\mathit{distance}[A] > \mathit{distance}[A]$, producing

"Limited area Fine-mesh Model"

as the definition, rather than

"Limited area Fine-mesh".

6 Training and Test Sets

`AFP` was tested on a collection of documents provided to our institute by the Department of Energy (DOE). This collection is almost entirely made up of government studies relevant to the Yucca Mountain Waste Disposal Project. The ASCII text of the collection is considered to be 99.8% correct. This collection consists of 1328 documents in a variety of formats. The documents have a wide content range and represent the work of many different organizations and authors. Since the government seems to be the source of most acronyms, we felt this collection was appropriate for our testing.

The training and test sets, while mutually exclusive, involved only a fraction of the documents in the collection. To select these sets, the full collection was automatically analyzed and sequenced according to the approximate ratio of acronyms to document length. A small set of these were selected for training and 17 documents were randomly selected from the remaining top 10% of the sequenced list for the test set. The training set was used to tune the acronym finding algorithm, develop new strategies, eliminate bugs, and adjust parameters. For example, the appropriate window size, word categories (e.g., stopwords, hyphenated words), and the default error percentage were tuned using the training set. No changes were made to the algorithm at evaluation time; and except for the information about the high incidence of acronyms in the test documents, no other information about their content was known prior to our evaluation.

7 Evaluation and Results

Our evaluation method for `AFP` mirrors the standard methods applied in most text retrieval experiments. We use:

$$recall = \frac{\text{\# of correct acronym definitions found by } \texttt{AFP}}{\text{total \# of acronym definitions in the document}}$$

$$precision = \frac{\text{\# of correct acronym definitions found by } \texttt{AFP}}{\text{total \# of acronym definitions found by } \texttt{AFP}}$$

An independent evaluator tallied the number of acronym definitions in the text, as well as manually examined the algorithms performance on the test set. The results did not include what the evaluator classified as *abbreviations*. Abbreviations encompass acronyms, so the evaluator distinguished between them by applying the following rules:

- Abbreviations shorten single words, acronyms do not.
- Abbreviations can include break characters, acronyms do not (e.g. ".").
- Abbreviations are used for unit measures, acronyms are not.
- All other shortened word forms were counted as acronyms.

Excluded words:

DOP	dioctyphthalate
MFBM	thousand board feet measure
TRU	transuranic
MW-hr	megawatt-hour

Included words:

EDBH	Engineered design borehole
D&E	Development and evaluation
CHEMTREC	Chemical Transportation Emergency Center

Following this definition, there were 463 acronym definitions in the 17 documents used for the evaluation. Of these, 398 were correctly identified by `AFP`, yielding:

$$\begin{aligned} \textit{recall} &= 86\% \\ \textit{precision} &= 98\% \end{aligned}$$

We made a conscious decision to exclude acronyms of two or fewer characters. If we exclude these from our evaluation, the recall results improve:

$$\begin{aligned} \textit{recall} &= 93\% \\ \textit{precision} &= 98\% \end{aligned}$$

Acronyms missed by `AFP`, and the reasons they were missed include:

MSRE: molten salt reactor— Falls below the default 80% threshold.

R&D: research and development— Was not considered an acronym candidate due to the '&' symbol.

GBL: grain boundary sliding controlled by lattice diffusion— Filtered out due to too many misses.

TWCA: Teledyne Wahchang Albany— Falls below the default 80% threshold.

USGS: U.S. Geological Survey— "U.S." was considered a single word when parsed, and therefore falls below the default 80% threshold.

8 Conclusion

`AFP` did quite well on a difficult document collection. Of course, with hindsight, it is easy to see how the program could be improved; most notably, the inclusion of '&' as an acronym character would increase recall. Some adjustments like special acronym characters or acronym length could be provided as options to `AFP` so the program could be tailored to a document's or collection's content. But in its current form, the program's framework is quite solid for its dedicated task.

Previous work involving the automatic recognition of special terms[2, 4, 5, 6, 10] implicitly assumes "clean" text, not the error-prone output of OCR devices. As a result of allowing misses in `AFP`, this algorithm is naturally suited for use with OCR data without any further modifications except possibly tuning the allowable error percentage. Further analysis is needed to determine the algorithm's precision and recall on OCR text.

9 Acknowledgments

We thank Julie Borsack and the anonymous referees for their helpful comments and suggestions on an earlier draft of this paper.

References

[1] C. L. Borgman and S. L. Siegfried. Getty's Synoname and its cousins: a survey of applications of personal name-matching algorithms. *JASIS*, 43(7):459–476, 1992.

[2] Jack G. Conrad and Mary Hunter Utt. A system for discovering relationships by feature extraction from text databases. In W. Bruce Croft and C. J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 260–270. SIGIR, Springer-Verlag, July 1994.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, tenth edition, 1993.

[4] W. Bruce Croft, Howard R. Turtle, and David D. Lewis. The use of phrases and structured queries in information retrieval. In *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 32–45, Chicago, IL, October 1991. ACM Press.

[5] Scott C. Deerwester, Keith Waclena, and Michelle LaMar. A textual object management system. In *Proceedings of the Fifteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 126–139, Denmark, June 1992. ACM Press.

[6] J. Fagan. *Experiments in Automatic Phrase Indexing for Document Retrieval: A comparison of Syntactic and Non-Syntactic Methods*. Ph.D. dissertation, Cornell University, 1987.

[7] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[8] M. E. Lesk. Lex—a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[9] Michael L. Mauldin. *Information Retrieval by Text Skimming*. Ph.D. dissertation, Carnegie Mellon University, 1989.

[10] Lisa F. Rau and Paul S. Jacobs. Creating segmented databases from free text for text retrieval. In *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 337–346, Chicago, IL, October 1991. ACM Press.

[11] Stephen V. Rice, Junichi Kanai, and Thomas A. Nartker. An evaluation of OCR accuracy. Technical Report 93-01, Information Science Research Institute, University of Nevada, Las Vegas, April 1993.

[12] Kazem Taghva, Julie Borsack, Bryan Bullard, and Allen Condit. Post-editing through approximation and global correction. Technical Report 93-05, Information Science Research Institute, University of Nevada, Las Vegas, March 1993.

[13] Kazem Taghva, Julie Borsack, and Allen Condit. Results of applying probabilistic IR to OCR text. In *Proceedings of the Seventeenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 202–211, Dublin, Ireland, July 1994.

[14] Kazem Taghva, Julie Borsack, Allen Condit, and Srinivas Erva. The effects of noisy data on text retrieval. *Journal of the American Society for Information Science*, 45(1):50–58, January 1994.

[15] Kazem Taghva, Julie Borsack, Allen Condit, and Jeff Gilbreth. Results and implications of the noisy data projects. Technical Report 94-01, Information Science Research Institute, University of Nevada, Las Vegas, March 1994.