An Assessment of Name Matching Algorithms

A. J. Lait[1] and B. Randell[2]

Department of Computing Science

University of Newcastle upon Tyne

Abstract

In many computer applications involving the recording and processing of personal data there is a need to allow for variations in surname spelling, caused for example by transcription errors. A number of algorithms have been developed for name matching, i.e. which attempt to identify name spelling variations, one of the best known of which is the Soundex algorithm. This paper describes a comparative analysis of a number of these algorithms and, based on an analysis of their comparative strengths and weaknesses, proposes a new and improved name matching algorithm, which we call the Phonex algorithm. The analysis takes advantage of the recent creation of a large list of "equivalent surnames", published in the book Family History Knowledge UK [Park1992]. This list is based on data supplied by some thousands of individual genealogists, and can be presumed to be representative of British surnames and their variations over the last two or three centuries. It thus made it possible to perform what we would argue were objective tests of name matching, the results of which provide a solid basis for the analysis that we have performed, and for our claims for the merits of the new algorithm, though these are unlikely to hold fully for surnames emanating largely from other countries.

1. Name Variations

There are many applications of computer-based name-matching algorithms including record linkage and database searching where variations in spelling, caused for example by transcription errors, need to be allowed for. The success of such algorithms is measured by the degree to which they can overcome discrepancies in the spelling of surnames; evidently, in some cases it is not easy to determine whether a name variation is a different spelling of the same name or a different name altogether. Most of these variations can be categorised as follows [Bouchard&Pouyez1980]:

- *Spelling variations:* These can include interchanged or misplaced letters due to typographical errors, substituted letters (as in Smyth and Smith), additional letters (such as Smythe), or omissions (as with Collins and Colins). Generally such variations do not affect the phonetic structure of the name but still cause problems in matching names. These variations mainly arise from mis-reading or mis-hearing, by either a human or an automated device.
- *Phonetic variations:* Where the phonemes of the name are modified, e.g. through mis-hearing, the structure of the name is substantially altered. Sinclair and St. Clair are related names but their phonetic structure is very different. Indeed, phonetic variations in first names can be very large as illustrated by Christina and its shortened form Tina.
- *Double names:* There are some cases where surnames are composed of two elements but both are not always shown. For example, a double surname such as Philips-Martin may be given in full, as Philips or as Martin.

---

[1]Present affiliation: TeleWare Ltd., Thirsk, Yorks.

[2]Contact author: Email = Brian.Randell@newcastle.ac.uk PHONE = +44 191 222 7923, FAX = +44 191 222 8232

- *Double first names:* Although not common in the English language, when considering French, for example, names such as Jean-Claude may be given in full, or as Jean and/or Claude.
- *Alternate first names:* Where an individual changes their name during the course of their life or is called by one of their first names during a period of their life and another later on, this causes a major problem in name-matching. In these situations an algorithm that recognises simple variations in spelling or phonetics would not be able to identify two such names as referring to the same person.

Evidently, in differing circumstances, the type of variations as well as the types of names encountered, and hence the name matching algorithm needed, will vary. Even taking this into account, the assessment of the relative merits of different name matching algorithms is normally hampered by the absence of any objective means of deciding whether any two different names should be regarded as equivalent.

Needless to say, any name matching scheme can be but a contribution to the frequently encountered task of trying, in the absence of any scheme of personal identity numbers, or the like, to determine whether two sets of personal information in fact relate to the same individual. Thus further checks, such as of address, occupation, etc., will often be needed. Nevertheless, a fast and relatively accurate name matching algorithm is of great utility in many applications involving the use of nominal records.

The work described in this paper was prompted by the realisation that, for one major arena in which much use is made of name matching, that of historical record linking of the sort practised by genealogists (and some geneticists), a source of data had become available which could serve as the basis for what could be argued were objective tests of name matching. This data source is a list 35,000 "equivalent" surnames, taken from the book Family History Knowledge UK [Park1992], a directory whose purpose is to help genealogists who are researching the same family get in contact with each other.

This list is in fact based on data supplied by several thousand individual genealogists, based presumably on their own experience and judgement as to what spelling variations are encountered in practice. (It should be noted that a century and a half or more ago surname spellings were much more varied than during the present century, as people have become much more careful about how their names are spelled.) The list can therefore be presumed to be representative of British surnames over the last few centuries - no other such publically available list is known to us.

In the work reported on in this paper we use equivalence lists generated either directly from the Family History Knowledge data, or by making statistical perturbations of such data that we claim are representative of some of the other variations, due to transcription and typographical errors, that name matching should be able to cope with. These lists were used as a means for what we would argue are well-justified comparisons of the performance of a number of existing name matching algorithms, an analysis that revealed the misleading nature of the accuracy claims that have been sometimes been made for these algorithms. We have also been able to use this data to guide the design of a new algorithm which provides a significant increase in performance over that achieved by the existing algorithms. (Our analysis, and claims for the merits of the new algorithm, are restricted to British surnames, and unlikely to hold fully for surnames emanating largely from other countries.)

2. Current Name-Matching Methods

At present there are a number of name-matching algorithms employing different degrees of complexity to overcome name variations. These range from simplistic algorithms that just consider variations in the characters making up the name to those taking account the variations in phonetics as well. The following selection of techniques illustrates the current range of approaches to the problem in their more general form, since many of these algorithms can be, and have in fact been, modified to produce more accurate matches for more specialised applications.

2.1. Russell Soundex Name-Matching

The Russell Soundex Code algorithm is designed primarily for use with English names and is a phonetically based name matching method. The algorithm converts each name to a four-character code, which can be used to identify equivalent names, and is structured as follows [Knuth]:

1. Retain the first letter of the name, and drop all occurrences of a, e, h, i, o, u, w, y in other positions.
2. Assign the following numbers to the remaining letters after the first:

   b, f, p, v → 1 l → 4

c, g, j, k, q, s, x, z → 2 m, n → 5

d, t → 3 r → 6

3. If two or more letters with the same code were adjacent in the original name (before step 1), omit all but the first.
4. Convert to the form 'letter, digit, digit, digit' by adding trailing zeros (if there are less than three digits), or by dropping rightmost digits it there are more than three.

For example, the names Euler, Gauss, Hilbert, Knuth and Lloyd are given the respective codes E460, G200, H416, K530, L300. However, the algorithm also gives the same codes for Ellery, Ghosh, Heilbronn, Kant and Ladd [Knuth] which are not related in reality.

Although the Russell Soundex method is more accurate than just relying on character similarities between the names, the algorithm is not ideal. Indeed, when comparing first names, different codes could be given for abbreviated forms of a name so Tom, Thos. and Thomas would be classed as different names [Floyd1993].

2.2. Henry Name-Matching

The Henry Code, devised by Louis Henry, is based on the Russell Soundex method but is adapted for the French language and classifies each name as a three-letter code. Like the Russell Coding Technique, the Henry method can also result in completely different names being brought together as with the French names Mireille, Marielle and Merilda which are all given the code MRL.

A second form of the Henry algorithm returns codes that are not limited to three letters; however, it is still prone to mismatching as it often modifies the phonetic structure of the names, resulting in it sometimes missing good links or linking completely different names. In its present form, the Henry Code cannot be programmed and the formulation of the rules is such that they cannot be 'translated' into computer instructions [Bouchard&Pouyez1980].

2.3. Daitch-Mokotoff Coding Method

Devised by Gary Mokotoff and Randy Daitch, this method is another version of the Russell Soundex adapted for Slavic and German spellings of Jewish names [Mavrogeorge]. The basic enhancements to the original method are: the Soundex code is expanded from four to six digits, the first letter of the name is coded, single codes are assigned to some double letter combinations, letter combinations pronounced different ways coded more than once, Romance language pronunciations are changed to the Slavic/German tongue, and the hard and soft sounds of the letter C are separated.

The basic coding method is as follows [Mavrogeorge]:

1. Names are coded to six digits, each representing a sound listed in the coding chart.
2. When a name lacks enough coded sounds for six digits, use zeros to fill to six digits.
3. The letters A, E, I, O, U, J, and Y are always coded at the beginning of a name. In any other situation, they are ignored except when two of them form a pair and the pair comes before a vowel as in Breuer (791900) but not Freud.
4. The letter H is coded at the beginning of a name as in Haber (579000) or preceding a vowel as in Manheim (665600), otherwise it is not coded.
5. When adjacent sounds can combine to form a larger sound, they are given the code number of the larger sound. Mintz is coded as MIN-TZ, not MIN-T-Z.
6. When adjacent letters have the same code number, they are coded as one sound, as in TOPF, which is not coded TO-P-F but TO-PF. Exceptions to this rule are the letter combinations MN and NM whose letters are coded separately, as in Kleinman, which is coded 586660 not 586600.
7. When a surname consists of more than one word, it is coded as if one word, such as Ben Aron which is treated as Benaron.
8. Several 'letter and letter' combinations pose the problem that they may sound in one of two ways. The 'letter and letter' combinations CH, CK, C, J, and RS are assigned two possible code numbers.

Letter	Alternate Spelling	Initial Letter	Before a Vowel	Any other situation
AI	AJ, AY	0	1	Not coded (NC)
AU		0	7	NC
A		0		NC
B				7
CHS		5		54
CH	Try KH (5) and TCH (4)			
CK	Try K (5) and TSK (45)			
CZ	CS, CSZ, CZS			4
C	Try K (5) and TZ (4)			
DRZ	DRS			4
DS	DSH, DSZ			4
DZ	DZH, DZS			4
D	DT			3
EI	EJ, EY	0	1	NC
EU		1	1	NC
E		0		NC
FB				7
F				7
G				5
H		5	5	NC
IA	IE, IO, IU	1		NC
I		0		NC
J	Try Y (1) and DZH (4)			
KS		5		54
KH				5
K				5
L				8
MN				66
M				6
NM				66
N				6
OI	OJ, OY	0	1	NC
O		0		NC
P	PF, PH			7
Q				5
RZ, RS	Try RTZ (94) and ZH (4)			
R				
SCHTSCH	SCHTSH, SCHTCH	2		9
SCH				4
SHTCH	SHCH, SHTSH	2		4
SHT	SCHT, SCHD	2		4

SH				43
STCH	STSCH, SC	2		4
STRZ	STRS, STSH	2		4
ST		2		4
SZCZ	SZCS	2		43
SZT	SHD, SZD, SD	2		4
SZ				43
S				4
TCH	TTCH, TTSCH			4
TH				4
TRZ	TRS			3
TSCH	TSH			4
TS	TTS, TTSZ, TC			4
TZ	TTZ, TZS, TSZ			4
T				4
UI	UJ, UY	0	1	3
U	UE	0		NC
V				NC
W				7
X		5		7
Y		1		54
ZDZ	ZDZH, ZHDZH	2		NC
ZD	ZHD	2		4
ZH	ZS, ZSCH, ZSH			43
Z				4

Figure 1: Daitch Mokotoff Coding Method

2.4. Metaphone Coding Method

This technique was developed by Lawrence Philips for matching words that sound alike and is based on commonplace rules of English pronunciation. Metaphone ignores vowels after the first letter and reduces the remaining alphabet to sixteen consonant sounds, although vowels *are* retained when they are the first letter. Duplicate letters are not added to the code. Zero is used to represent the ‘th’ sound since it resembles the Greek theta when it has a line through it, and ‘X’ is used for the ‘sh’ sound.

The sixteen consonant sounds are: B X S K J T F H L M N P R Ø W Y

Letter	**Code**	**Comments**
B	B	unless at the end of a word after “m” as in “dumb”
C	X	(sh) if “-cia-” or “-ch-”
	S	is “-ci-”, “-ce-”, or “-cy-”
		silent if “-sci-”, “-sce-”, or “-scy-”
	K	otherwise, including “-sch-”
D	J	if in “-dge-”, “-dgy-”, “-dgi”
	T	otherwise
F	F	

G		silent if in “-gh-” and not at end or before a vowel
		in “-gn” or “-gned”
		in “-dge-”, etc., as in above rule
	J	if before “i”, or “e”, or “y”, if not double “gg”
	K	otherwise
H		silent if after vowel and no vowel follows
	H	otherwise
J	J	
K		silent if after “c”
	K	otherwise
L	L	
M	M	
N	N	
P	F	if before “h”
	P	otherwise
Q	K	
R	R	
S	X	(sh) if before “h” or in “-sio-” or “-sia-”
	S	otherwise
T	X	(sh) if “-tia-” or “-tio-”
	Ø	(th) if before “h”
		silent if in “tch-”
	T	otherwise
V	F	
W		silent if not followed by a vowel
	W	if followed by a vowel
X	KS	
Y		silent if not followed by a vowel
	Y	if followed by a vowel
Z	S	

Exceptions:

Initial “kn-”, “gn-”, “pm-”, “ae-”, “wr-”	drop the first letter
Initial “x”	change it to s
Initial “wh-”	change to w

Figure 2 Metaphone Coding Method.c1.Figure 2: Metaphone Coding Method;

The first four letters of the phonetic spelling (if there are that many) are used for comparisons, giving, for example SKL for school and XBRT for Shubert. Modifications to the Metaphone technique can be made to account for the ‘k’ sound in ‘chemical’ and ‘technical’, or the ‘SH’ sound in ‘casual’, for example, but “after a certain point you run into the contradictory cases (‘-sua’ in ‘casual’ is ‘SH’ but in ‘persuade’ the ‘s’ sounds like an ‘S’)” [Mavrogeorge].

In comparison to the Russell Soundex code, the Metaphone technique is able to distinguish names such as Bonner and Baymore (BNR and BMR), Smith and Saneed (SMØ and SNT), or Van Hoesen and Vincenzo (VNHSN and VNSNS), which the original method gave the same code to.

2.5. FONEM, I.S.G, INC and DIC Name-Matching

These algorithms were designed for use with French names only, for the Saguenay Research Programme, and are written for use with pairs of individuals rather than individuals themselves [Bouchard&Pouyez1980]. The FONEM Programme is designed to overcome spelling variations and contains 64 rules. Of these, 53 are transcription rules, and 11 are auxiliary rules that establish an order of priority among the transcription rules when more that one rule can be applied to the same name. It converts names into generic forms, such as SAINT-GELAIS and CINGELET into StJELAIS, or GADRAU and GODREAULT into GODRO. The algorithm is claimed to reduce the number of variant spellings by 18.6 percent (nearly the same for first names and surnames).

Two programs were developed to overcome phonetic variations between names: the I.S.G and the INC Programmes. The first of these, the I.S.G. Programme, computes an index of similarity, combining work by the P.R.D.H. group in Montreal and Gloria Guth (see Guth Name Matching, below). The index is computed according to the formula

$$i = \frac{I}{I + D}$$

where i is the index of similarity, I is the number of identical letters in the two names, and D is the number of different letters in the two names. This gives a measure of the degree of similarity in the range 0 to 1. The I.S.G. Programme is applied only when two names have not been accepted as ‘included’ through the INC Programme (see below).

The INClusion Programme overcomes situations where large differences in truly linked names give a very low I.S.G. score (such as Alex for Alexander, 0.44, or Bart for Bartholomew, 0.36). Two names are determined to be ‘included’ if one is necessarily shorter than the other and bears a close resemblance to some part of the longer one.

2.6. K-Approximate String-Matching

Another method of tackling the problem is by utilising algorithms employed in some spelling checking systems where *k-approximate string-matching* is performed. A *k-approximate match* is a match of P (a pattern to be searched for) in T (a section of a text). The differences may be any of the following:

1. The corresponding characters in P and T are different.
2. P is missing a character that appears in T.
3. T is missing a character that appears in P.

This is a generic method for searching for character variations in strings and takes no account of phonetics or language. This may be seen to be an advantage but the algorithm only works reliably if P is shorter than T. For example, when searching for St. Clair the algorithm matches Saint Clair with k=2 but when searching for Saint Clair, the algorithm requires k=4 before it matches with St. Clair in the text. The accuracy of this method is also dependent on whether the case of letters is considered or whether spaces between letters are ignored.

2.7. Guth Name-Matching

Whereas the Russell Soundex Code, Henry Code and FONEM method are all language-specific systems, Gloria Guth's algorithm, using letter-by-letter comparison, is not and has obvious advantages when applied to multi-ethnic populations.

The algorithm first checks for the case that two names are identical, considering each name as a single character string. If this fails, the algorithm proceeds to compare the surnames letter-by-letter and when the program encounters different letters in the same position it then searches for matching letters in other positions. The comparison pattern is illustrated in Figure 3 [Guth1976].

Tests for Position of Letters in Matching

Alternate Surname Spelling

	Position in Name 1	Position in Name 2
Test 1	X	X
Test 2	X	X + 1
Test 3	X	X + 2
Test 4	X	X – 1
Test 5	X – 1	X
Test 6	X + 1	X
Test 7	X + 2	X
Test 8	X + 1	X + 1
Test 9	X + 2	X + 2

Figure 3: Guth Matching Comparisons

If a search fails to find any of these patterns in the surnames, they are declared to be different. Figure 4 [Guth] shows how the algorithm processes the two (presumably equivalent) names Glawyn and Glavin. (Letters being compared are shown in bold-italic face.)

Example of Name Recognition Algorithm Used to Identify Alternate Spellings of the Same Surname		
I	II	III
G L A V I N	G L A ***V*** I N	G L A ***V*** I N
G L A W Y N	G L A W ***Y*** N	G L A W Y ***N***
IV	V	VI
G L A ***V*** I N	G L ***A*** V I N	G L A V ***I*** N
G L ***A*** W Y N	G L A ***W*** Y N	G L A ***W*** Y N
VII	VIII	IX
G L A V I ***N***	G L A V ***I*** N	G L A V I ***N***
G L A ***W*** Y N	G L A W ***Y*** N	G L A W Y ***N***

Figure 4: Guth Matching Example

The algorithm is claimed to have four principal advantages over other methods of surname recognition [Guth]:

1. It is not dependent on prior generation of a sorting key, such as the Russell Soundex code, based on the phonetics of the surname.
2. It is data independent and could be adapted easily to different types of data or linkage requirements.
3. When used in record linkage applications it can be incorporated into the actual record-linkage procedures, permitting the degree of match in the surnames to be included as one of the criteria for record linkage, but it does not alter the input records in any way.
4. Alternative spellings of the same name are identified by the position of the letters in the surname and not by phonetic equivalency.

It does not, however, allow for name variations such as double names but neither do any of the other algorithms listed above. It can also produce incorrect matches between surnames that bear little visual resemblance to one another; a problem that becomes particularly acute when comparing short names where one or two common vowels can produce a mismatch [DeBrou&Olson1986]. DeBrou and Olson suggest that including a function to assign a level of 'confidence' in the link, according to the number of letters that occur in the same positions of the names being compared, would overcome this weakness.

Guth's own experiments, using data from Eighteenth-Century Norwich pollbooks, showed the algorithm to correctly link 98.32 percent of the names (failing on 46 out of 2740 truly linked pairs), compared to the Russell Soundex Technique which overcame 80 percent of spelling variation in a study in Hamilton, Ontario. Experiments by DeBrou and Olson suggest that, on an ethnically mixed population, the Guth algorithm can achieve an overall degree of accuracy of 86.9 percent.

3. Algorithm Comparison

It was decided that of the techniques described above the following four were most suitable for implementation and subsequent performance comparison, since they provided a range of different approaches and complexities:

- Russell Soundex Name-Matching method
- Guth Name-Matching method
- K-Approximate String-Matching method
- Metaphone Coding method

The Russell Soundex name-matching method is a commonly used technique and has been modified for use with languages other than English. For the purpose of this investigation it was considered most appropriate to implement the original Russell Soundex coding algorithm instead of a more specialised form.

The Guth name-matching method compares names independent of the language or ethnic origin so is therefore very portable. It is fairly simple to code and appears to give reliable results, although is supposedly less effective when comparing short names.

The K-Approximate string-matching method offers an approach that is not language-specific and can be applied to comparison of first names and surnames. It is also simple to implement and provides a distinct alternative approach to those offered by the other methods selected.

The Metaphone coding method, although not strictly a name-matching technique, was chosen since it also can be applied to both first names and surnames. Like the Russell method, Metaphone also considers the phonetics of the names being compared and hence is specialised for use with the English language.

3.1. The Comparison Program

The language chosen for implementation of the name-matching techniques was C++. The different methods were first implemented as separate programs that took two names as input, converted all characters to upper case, and determined whether or not the names match.

The main body of the Russell Soundex method code was based on the C code of a similar program, SDX v1.1, by Michael Cooley. The Metaphone, Guth and K-Approximate matching methods were implemented to correspond to descriptions of the algorithms [Mavrogeorge1993, Guth, and Baase1989, respectively].

In order to allow direct comparison of the performances of the various name-matching algorithms, it was found necessary to develop a special Comparison Program. This program executes a selection of name-matching algorithms for a set of input data and produces results indicating the accuracy and performance of the methods being compared.

Input to the program is in the form of the external data file containing a formatted list of names; each name is compared to all other names in the file for each name-matching algorithm to be tested. The lists of sets of equivalent names in Family History Knowledge UK relate only to genealogical equivalents due to vagaries of spelling and do not in general include variant names that differ due to basic typographical errors, for example character omissions, additions or transpositions. Lists of names that incorporate such typographical variations were therefore generated by corrupting characters in the original list of sets of

equivalent names in a controlled manner. Since some name-matching algorithms rely on the first letters of names being the same to determine equivalency (such as the Russell Soundex Code method) it was necessary to consider corruption of the first letter of names separately.

Every name in the external data file is compared to every other name in the file (including the comparison of a name to itself) This is important, since some methods (such as the K-Approximate string-matching algorithm) treat the comparison name1/name2 and name2/name1 separately.

The accuracy of each name-matching method is determined by the number of so-called 'mu' ('false positive') and 'lambda' ('false negative') errors resulting when applied to a set of names to be compared. The former category of matching-error occurs when names determined to match are not truly equivalent, and the latter when truly equivalent names are determined not to match by the algorithm. Given a set of names that may or may not be equivalent, for a specific name, the comparison algorithm applies the separate name-matching algorithms to this data in turn and determines the accuracy of the results obtained, in terms of these "mu" and "lambda" error rates.

The execution times of the algorithms are measured as the time elapsed between commencing execution of the algorithm and outputting the results of names determined to match. (To avoid the execution times being affected by time taken to access these files. Care was taken that all algorithms use the same code to read data from any external files.)

Since some name-matching techniques may be more processor-intensive than others they may perform better on faster systems. The number of character comparisons performed by an algorithm on a specific set of data was chosen as giving a good additional indication of the processing demands of each method. To count the number of character comparisons during execution of an algorithm, an integer variable is incremented each time a comparison occurs. Although this required minor alterations to the original name-matching algorithm code, it was applied to all of the algorithms, so as not to affect the relative execution times of the different methods.

Measurement of total elapsed time is implemented by calculating the difference between the clock time before and after execution of each algorithm for all of the name-pairs. Following this, the statistics for the algorithm are computed and output before commencing execution of the next name-matching algorithm.

In summary, the comparison program calculates the following for each name-matching method:

- Total number of name-pairs determined to match, and the number of those that were correct.
- Number of true matches and the percentage of those identified.
- Total number of name-pairs determined not to match, and the number of those that were correct.
- Number of true mismatches and the percentage of those identified.
- Total number of correct determinations.
- Total number of incorrect determinations.
- Overall accuracy as a percentage (= 100 * total correct / total name-pairs).
- Total number of character comparisons performed, and average number of comparisons per name-pair.
- Time elapsed during execution.

This allows a comparison of algorithms' relative performances and includes an indication of those which determine more names to match (more false positives) and those which determine more names not to match (more false negatives).

4. Selection of Data

In order to perform a valid analysis of the performance of the name-matching algorithms, it was crucial to select a representative sample of data. Based on an understanding of the basic principles of the algorithms selected for the comparison, it was apparent that four main aspects of the data sample were of particular importance:

1. *Character content* of the names, since those name-matching methods that rely heavily on the phonetic structure of the names would be influenced by the particular letters that comprise them. Hence if all the names in the sample begin with the same letter, it may favour one particular method of name-matching which performs better for that letter than other letters. It was therefore necessary to obtain a sample of

data that contained names beginning with each letter of the source alphabet (English). For each sample beginning with a particular letter, it was also important to ensure that the subsequent letters of the names were also different (as far as possible).

2. *Frequency of occurrence* of the names had also to be considered since it was important to obtain a sample that reflected the alphabetic distribution of names in reality. For example, it would have been unrealistic to have more names in the sample beginning with 'Z' than with 'S' since these are far less common in reality.
3. *Size* of individual names can also influence the performance of specific algorithms so the chosen sample had to contain a representative distribution of name-lengths.
4. *Volume* of data in the sample must be considered since this directly affects the execution-time of the name-matching algorithms. If the sample is too small, the time each method takes to process the data may be influenced more by physical aspects (such as disk access time) than by the algorithm itself. Additionally, the smaller the data sample, the smaller the difference in execution-times between algorithms is likely to be. Conversely, a data sample that is too large will result in the execution-times being unnecessarily long and could result in overflow of statistical variables if their values exceed the maximum supported.

As indicated earlier, the data source chosen was the book Family History Knowledge UK (1992/3 edition), since Section B (pages 11-155) consists of lists of equivalent surnames. Using an 'Equivalence Coding' program, this data could be converted into a list of names and associated 'equivalence codes' suitable for use with the comparison program.

Since the book contains in the region of 35,000 surnames, it was necessary to select a representative sample from the list. This involved the use of a scanner and Optical Character Recognition (OCR) software to obtain the data into a 'machine-readable' form, then manual verification and correction to remove any incorrectly scanned letters and errors from the data. For each letter of the alphabet (for which names existed in the list) a randomly chosen but representative subset of the list was obtained to conform to the first three selection criteria above. This resulted in lists for each letter of average size approximately 2 KB (depending on the volume of names available for that letter), where 1 KB = 1 kilobyte = 1024 characters.

The total sample, after processing with the 'Equivalence Coding' Program, contained approximately 67 KB of data (including 'equivalence codes'), composed of 5600 distinct surnames and almost 800 distinct equivalence classes of names.

For more specific analysis, further samples of this data were made to analyse the name-matching algorithms' performances on short and long names separately. This was achieved by selecting those names of 5 characters or less, and those of 9 characters or more, respectively, and comparing the algorithms using those samples. The values of 5 and 9 were used, since the average length of name in the total sample was found to be 7.6 letters, and these two values are of approximately equal deviation, each side of this mean value.

Due to the nature of the unique equivalence codes assigned to the names, removing names from the data sample does not affect the integrity of the coding (the comparison program depends on uniqueness of the values, not sequential continuity so it does not matter if, say, codes 123 and 125 are present in the sample, and 124 is not).

5. The 'Equivalence Coding' Program

To automate the translation of data from the book Family History Knowledge UK into a form suitable for input to the Comparison Program, it was necessary to develop a program which produces, as output, a list of names and associated 'equivalence codes'. An 'equivalence code' is a unique integer value assigned to all names in a particular equivalence class. It is important that the output file does not contain any duplicated names, since these would detrimentally affect the results obtained from the Comparison Program.

The resulting 'Equivalence Coding' Program allows automatic equivalence code generation without the need for manual cross-referencing of the names; however, it does rely on the input data being accurate (with no misspelled or incorrectly formatted names) in order to associate names that are truly equivalent. In simplistic terms, the program reads lines of names from the scanned data and uses the indented formatting

to determine names that are equivalent (i.e. all those following a name on the same line, or those on the line below if indented).

5.1. Input File Format

```
NAME 6 FIRST_EQUIV_NAME SECOND_EQUIV_NAME√
  THIRD_EQUIV_NAME FOURTH_EQUIV_NAME√
  FIFTH_EQUIV_NAME√
NAME FIRST_EQUIV_NAME SECOND_EQUIV_NAME√
:
NAME FIRST_EQUIV_NAME SECOND_EQUIV_NAME√
:
<EOF>
```

(√ denotes a carriage return, <EOF> is the end-of-file character)

Figure 5: Input File Format

The format of the input data is a formatted list of names (as an ASCII file) closely matches that printed in Family History Knowledge UK, where each line contains one or more equivalent names, separated by white-space, with further names in the same equivalence class on subsequent lines, indented with white-space characters. Where more than 4 equivalent names are listed in a class, the total number of such names is given as an integer value following the first name, surrounded by white-space.

5.2. Output File Format

```
1 FIRST_EQUIV_NAME√
1 SECOND_EQUIV_NAME√
:
1 LAST_EQUIV_NAME√
2 FIRST_EQUIV_NAME√
2 SECOND_EQUIV_NAME√
:
2 LAST_EQUIV_NAME√
:
<EOF>
```

(√ denotes a carriage return,
<EOF> is the end-of-file character)

Figure 6: Output File Format

The format of the output file produced is a plain ASCII text list of ‘equivalence codes’ and names in upper case. Each line of the file starts with an integer ‘equivalence code’ followed by white-space, then the name in upper case. The end of the line is denoted by a carriage return character.

The ordering of names within the file is not important, although it should not contain duplicated names - something that the Equivalence coding program ensures by merging sets of equivalent names that it finds are overlapping.

6. The ‘Data Corruption’ Program

Since the sample of names taken from the book Family History Knowledge UK represented only surnames that were genealogically equivalent, it was also necessary to compare the name-matching algorithms' performances on data that differed due to corruption of its content. For example, names that differ due to

missing, additional, interchanged, or substituted letters (such as generated through typographical errors or misinterpretation). Such lists of names were generated automatically using a Data Corruption Program.

In order to simulate data which is 'corrupted' in some manner such as that generated due to transcription errors, the Data Corruption program was designed to randomly corrupt characters of names in a file of 'equivalence coded' names in the following manners (an 'equivalence code' is a unique integer value assigned to all names in an equivalence class):

1. Add n random extra letters to the name, or
2. Delete n letters from the name, or
3. Interchange n pairs of adjacent letters in the name, or
4. Substitute n letters in the name with random replacement letters

In all cases, the value n indicates the number of corruptions to be made to the name and should be user-definable. The corrupted names are output to 4 separate files, one file for each category of corrupted names, formatted identically to the original file of 'equivalence coded' names. All four of these categories may arise from mis-hearing, mis-typing or mis-scanning names, although it is less likely that scanning would result in interchanges of letters within a name. (For this investigation, only one corruption (one additional letter, missing letter, interchanged pair of letters, or substituted letter) was made to each name.)

7. Comparison Results

The comparison program was executed for a variety of data samples in addition to the original 'full' data set. These included sub-samples of names with 5 letters or less and names with 9 letters or more. Tests were also conducted with reduced sub-samples of the original data, and with corrupted forms produced using the Data Corruption Program.

7.1. Original Full Data Sample

From the results obtained during testing of the name-matching algorithms, the execution-time, accuracy, number of character comparisons per name-pair, and the number of correct matches attained by each algorithm proved the most applicable comparative statistics. The following charts illustrate these results in a graphical form which allows direct comparison of the relative performances of the algorithms:

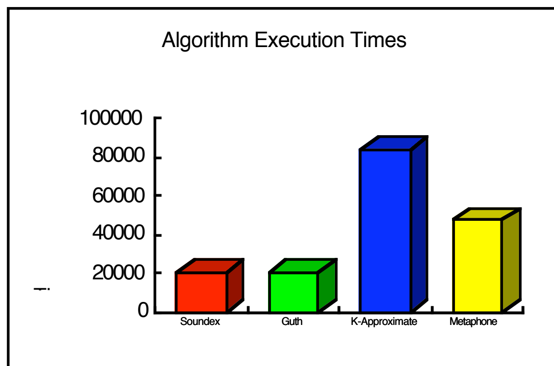


Figure 9 (a)

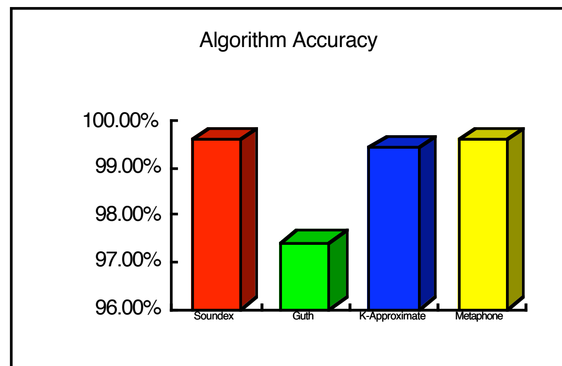


Figure 9 (b)

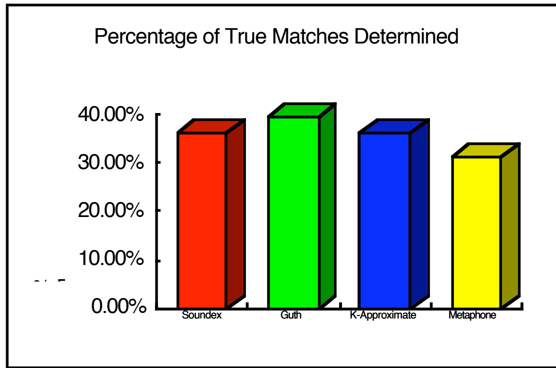


Figure 9 (c)

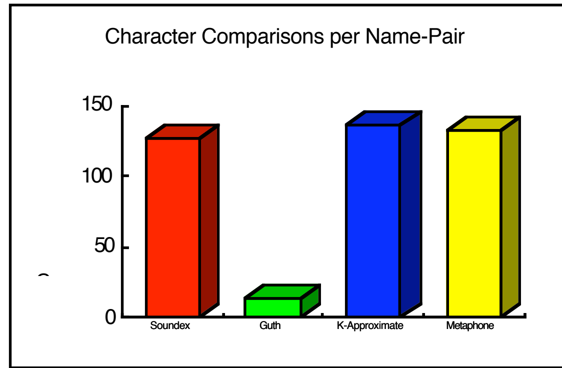


Figure 9 (d)

7.2. Original and Corrupted Data Samples

The following charts illustrate the results for the original 'full' data sample and the four corrupted forms generated with the Data Corruption Program (additional letters, missing letters, interchanged letter pairs, and substituted letters):

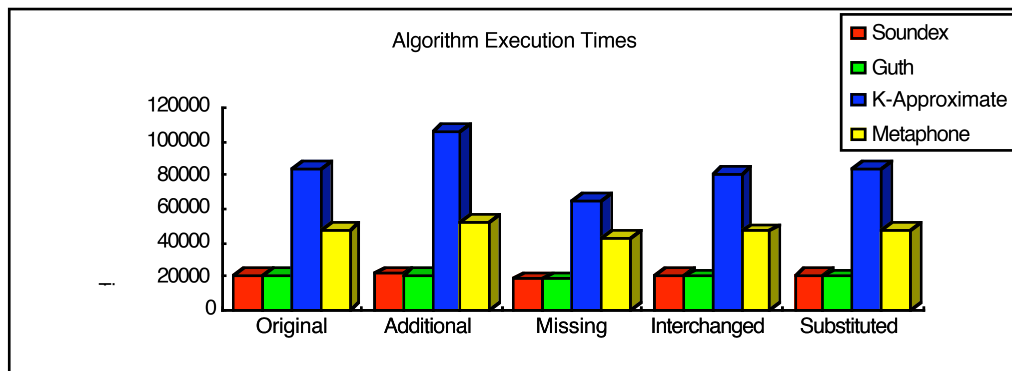


Figure 10 (a)

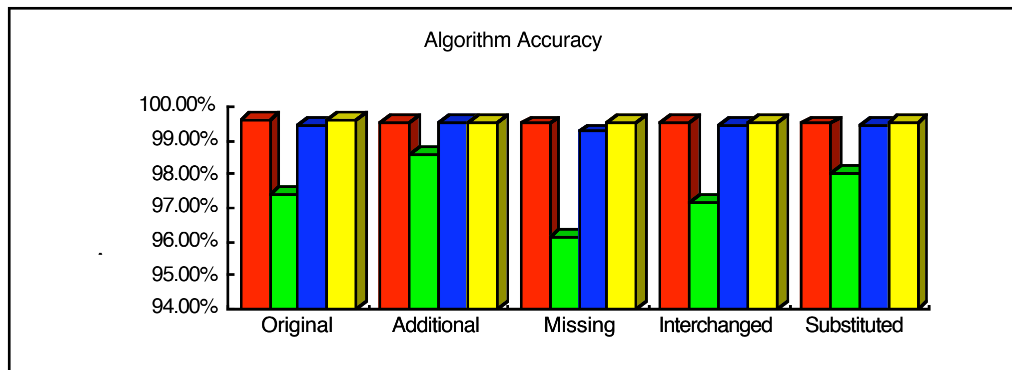


Figure 10 (b)

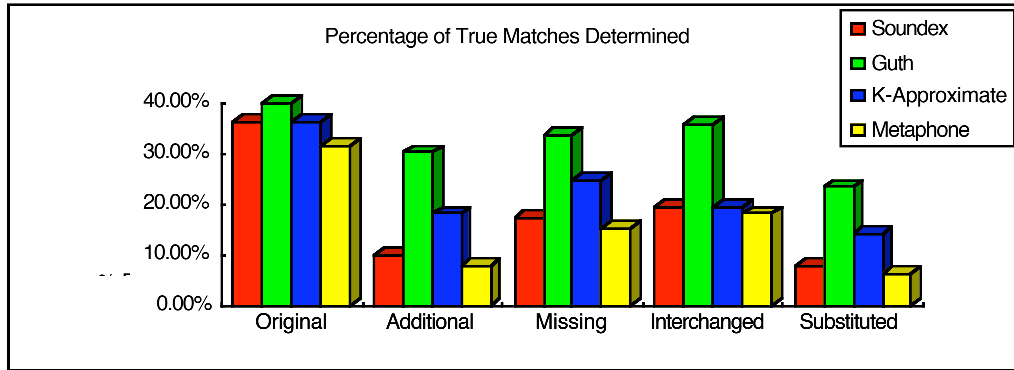


Figure 10 (c)

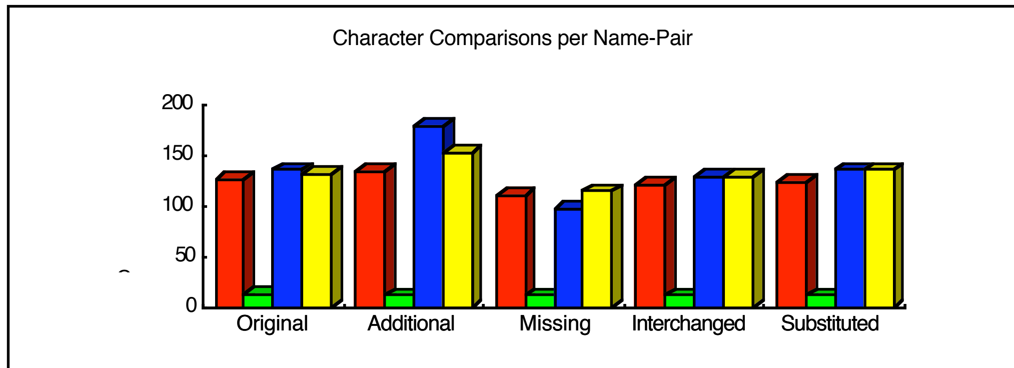


Figure 10 (d)

7.3. Full, Small Name and Large Name Samples

To determine whether certain name-matching methods performed better for small or large names, two sub-sample data sets were produced by extracting names of 5 letters or less and names of 9 letters or more, to two separate files. The were then used as input to the comparison program and the results are illustrated by the following charts (allowing comparison to those of the original full data sample). The execution-times for these data sets were lower than those for the original 'full' sample, as expected, due to the reduced volume of data, and hence a comparison of the times would be meaningless. Consequently a chart comparing these statistics is not given below:

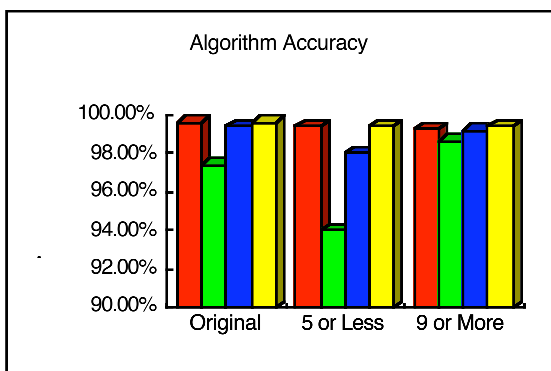


Figure 11 (a)

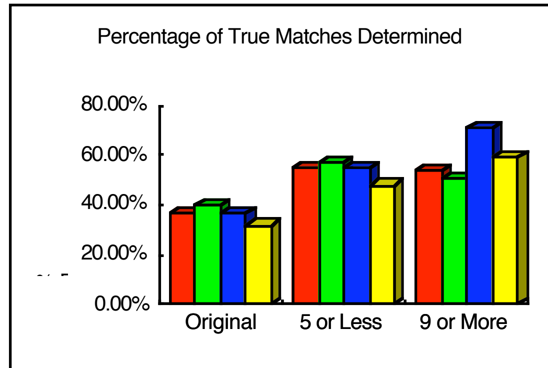
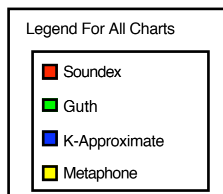


Figure 11 (b)



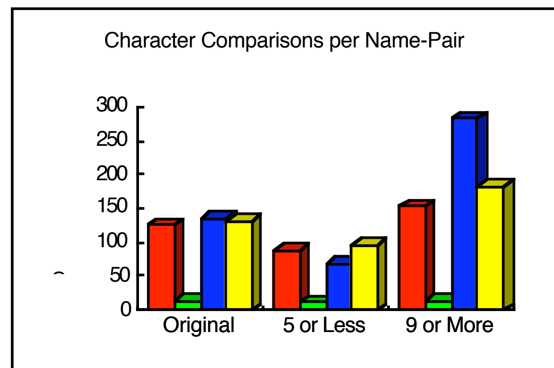


Figure 11 (c)

7.4. Full and Reduced Samples

In order to justify the validity of the original 'full' data sample, the algorithms were compared on 'reduced' samples taken from this. These were formed by extracting every other 'equivalence coded' name from the full sample (to produce a half-sample data set), then repeating the procedure to produce data sets that contained a quarter and an eighth of the original data sample. The results from the original and the one-eighth sample were very comparable, thus providing good evidence that the original data constituted a realistic sample.

Obviously, the execution times for the full sample were significantly longer than those for the eighth-sample. The only other discrepancy between the two sets of results was in the percentage of true matches obtained by the algorithms which was approximately 10% higher for the eighth-sample data (for all four methods). The cause of this was not initially obvious, but on inspection of further sub-samples of the original (sixteenth, thirty-second and sixty-fourth samples) it became apparent. Since all names in the sample were compared to every other name (including the comparison of a name to itself), the fewer names there were in the data, the higher the proportion of possible matches which were of a name with itself. All of the four name-matching algorithms compared will always determine a name to match with itself (naturally), and hence if a data sample contains no equivalent names (an extreme case), the only matches will be of names with themselves, resulting in the algorithms obtaining 100% of possible valid matches. (The percentage of true mismatches correctly determined is not affected in this manner.)

8. Analysis of Results

It is important to bear in mind that the results and statistics returned from the comparison program are largely specific to this implementation. The algorithm run-times are highly dependent on the processor speed and specific configuration of the machine the program is executed on. (For this investigation an IBM-PC compatible 386SX, running at 25MHz, was used and the comparison program executed as a single task, since multi-tasking is likely to affect algorithm execution-times when other processes use the CPU.) The timing results are also dependent on the volume of data and hardware speed, such as disk access times (in this case a Conner 42MB hard-disk with 18ms access-time was used).

In an investigation of this form, it is also important to consider the implications of using a disk-cache since this will affect the time required for subsequent reads of the data. The cache should either be disabled so each algorithm will have to read the data directly from the disk, or it must otherwise be ensured that all of the data is cached prior to execution of the first comparison algorithm. The latter is harder to achieve with large volumes of data since the cache must be large enough to hold all of the data and not have to reload any part of it during execution of the comparison algorithms, which would affect the execution times. For this investigation a 1MB disk-cache was used (using the MS-DOS 6.2 SMARTDRV.EXE program to provide read and write caching) which was sufficiently large to hold the data files and executable code of the comparison program.

The number of character comparisons and the algorithm accuracies are dependent on data content and volume since different names will cause algorithms to perform different execution paths through code. It is therefore only valid to consider average results when the comparison program is applied to large and small data volumes, with different name lengths.

The first impression of the performance of the name-matching algorithms is that the overall accuracy is extremely high (between 94% and almost 100%). However, this is misleading since the figure is influenced by the number of correct mismatches determined, rather than the number of correct matches made. Since the data will undoubtedly contain more name-pairs that are not truly equivalent than those that are, in the extreme case, if an algorithm determined all name-pairs to mismatch, the overall accuracy could still be in the region of 80% to 90%. Hence the number of correctly determined matches does not have a great effect on the overall accuracy of the results. This could explain the surprisingly high accuracy statistics quoted for other experiments, such as those by Gloria Guth and the Saguenay Research Programme.

However, although the overall accuracy is generally very high, the small variations in this statistic indicate differing performance when algorithms are applied to different data samples.

8.1. Original Full Data Sample

The results for the original data set give a good indication of the overall performances of the algorithms and can be used for direct comparison with results for the reduced and corrupted data samples. The execution times of the algorithms (Figure 9(a)) are quite crucial in determining their overall performance and it is clear to see that the K-Approximate string-matching algorithm takes significantly longer than the other three methods. In general the execution-time for the K-Approximate algorithm was about twice as long as that for the Metaphone method and between four and five times as long as the times for the Soundex and Guth algorithms. The execution-times for the latter two methods were virtually identical for all experiments and were generally half as long as those for the Metaphone algorithm.

The impressive speed of the Soundex method is probably due to the specific implementation used in this investigation where C++ pointers are largely used to reference the arrays of characters that represent the names under comparison. Similarly the Guth algorithm uses only references to characters (i.e. character comparisons) to determine if a pair of names matches and requires little additional computation.

Although the Metaphone method calculates a coded form of the name (in a similar manner to the Soundex technique), the additional complexity of this algorithm costs time and results in the method taking approximately twice as long. The K-Approximate string-matching algorithm also involves a large degree of additional computation in maintaining and evaluating the 'difference table' and this is the most likely reason for the significantly longer execution times. An interesting point is that neither the Guth or K-Approximate matching algorithms perform coding of the name before comparison (which could result in a longer execution- time) but this apparently does not give either of them a speed advantage.

Considering the accuracy of the name-matching algorithms (Figure 9(b)), the Soundex and Metaphone methods appear to be marginally more accurate; the K-Approximate method being just slightly less accurate on average (approximately 0.2%). The Guth method is surprisingly less accurate overall than the other three methods (despite the promising figures quoted by Gloria Guth herself) resulting in a figure of nearer 97.5% than the 99+% recorded for the other methods. As described previously, this figure is largely influenced by the number of mismatches determined correctly and hence methods that determine more names to match (regardless of whether these determinations are accurate) will achieve less overall accuracy.

Indeed this is the case with the Guth algorithm, since when analysing the percentage of true matches determined (Figure 9(c)), it is apparent that this method correctly identifies more matches than the other three (almost 40%). On analysis of the results the Guth method appears to match names more liberally than the other methods resulting in a higher total of name-pairs determined to match. Of these it is likely that

more will be correct than the number of correctly identified matches by the other methods, but also there will be more 'false positive' errors where the names are not truly equivalent. The Soundex and K-Approximate methods performed equally well in this measure (approximately 36% of true matches identified correctly) with the Metaphone algorithm achieving only 30%.

The number of character comparisons per name-pair required by each method (Figure 9(d)) appears to roughly correlate with the execution-times of the algorithms (with exception of the Soundex method) since it is likely that the more character comparisons are made, the more processing time is required. Since the implementation of the Soundex method uses C++ pointer variables more than the others (resulting in faster memory accesses), it can perform more access in less time than the other algorithms. It is important to note that this is largely due to the particular implementation of the Soundex algorithm and it is possible that this method had an unfair advantage over the others due to its use of pointers.

Contrary to what may have been expected, there is no general relationship between the accuracy or percentage of true matches determined, and the execution times of the algorithms. One might have guessed that algorithms which were more accurate would take longer, but the Soundex and Metaphone methods both have very high overall accuracy but take less time than the K-Approximate method which is less accurate.

In comparison to the other three methods, the Guth algorithm requires far fewer character comparisons per name-pair (approximately 10 times less). This implies a high level of efficiency in comparing a pair of names resulting in a relatively fast execution time, however, this seems to be traded for a lower accuracy as a consequence.

8.2. Corrupted Data Samples

The trends in execution-times (Figure 10(a)) shown in the results for the original 'full' sample are reflected in those for the four corrupted forms of the data. As may be expected, the execution-times for all methods are longer for the data containing random additional letters (since the volume of data is larger), and generally shorter for the data with letters missing (the times for the 'interchanged' and 'substituted' data sets are approximately the same as for the original). This indicates that on average, the content of the data does not directly affect the execution-time, and the volume of data is more responsible.

The accuracy (Figure 10(b)) of the algorithms was not greatly affected by the corruptions to the data, although it produced a greater variance in the accuracy of the Guth algorithm than the other three. For all four 'corrupted' data sets, the accuracy of the Soundex and Metaphone algorithms was not greatly affected, resulting in at most a 0.2% reduction. The K-Approximate string-matching method showed a surprisingly higher overall accuracy for the data containing additional letters than for the original untouched data set (a 0.1% increase in accuracy). This increase results from a larger number of mismatches correctly determined in the data with additional characters. It is likely that these extra random characters caused the method to correctly determine names that it had incorrectly determined to match in the original data sample, resulting in a higher number of correctly determined mismatches.

The percentage of true matches determined (Figure 10(c)) is affected more by the corruptions although the trends of the original data sample results are still evident in the results for the corrupted forms. The Guth method was least affected by the corruptions which is most probably due to its technique of comparing names. Since there is at most 1 difference between a given character in the original name and in the corrupted form, the algorithm is able to ignore this to a certain extent since it also compares the next two character positions in each name (see Chapter 3).

The effect of the corrupted data on the other three methods was more significant resulting in a 20-30% drop in the percentage of true matches determined. The Soundex and Metaphone methods were affected greatest since additional, missing, interchanged or substituted consonants in the names will result in different coded forms being computed.

Finally, the number of character comparisons per name-pair (Figure 10(d)) was not greatly altered by the corruption although the data containing additional letters resulted in generally more comparisons and the data with missing letters resulting in less. This is due to a respective net increase and decrease in the lengths of names in the data, which affects the number of comparisons per name (and hence per name-pair). The Guth algorithm is the only method of the four that was not affected significantly by the corruptions since it maintained approximately 12.5 comparisons per name-pair for all data samples.

8.3. Large and Small Name Samples

With the exception of the Guth method, the overall accuracy of the matching algorithms (Figure 11(a)) was not greatly altered when comparing only small (5 letters or less) and large (9 letters or more) names. The K-Approximate method was less effective on the small name sample but the greatest variation is seen in the results for the Guth method. As stated earlier, it has been found that this method is less effective when applied to shorter names (see Chapter 3). This was indeed true – the algorithm was 3.3% less accurate for the small names data, but also 1.2% more accurate for the large names data.

Strangely, though, the percentage of true matches determined (Figure 11(b)) was higher for all methods when applied to the small names data and large names data. This would appear to indicate that the algorithms match more small and large names correctly than average-length names. The K-Approximate method correctly matched 71% of all true matches compared to only 36% for the original data sample. However, it must be remembered that these reduced data samples will contain a higher proportion of truly matching names (see explanation above) which will make these results appear more impressive. The only valid conclusion from these statistics is that all of the methods seem to perform better for longer names (as there are more letters per name to aid in the determination). The Guth method determines more matches correctly when applied to small names, but this is due to it matching names more liberally than the other methods, resulting in more 'false positive' matches or 'lambda' errors.

The number of character comparisons per name-pair (Figure 11(c)) is affected as would be anticipated, with less comparisons required on average for the small names data, and more required for the large names data.

8.4. Conclusions

Considering the overall results of the name-matching algorithms, the comparison has confirmed many prior expectations and revealed a few surprising characteristics of certain methods. For example, it might have been anticipated that the more complex an algorithm, the more accurate the results obtained, however, this seems not to be the case. The two more complicated techniques (in terms of required computation) are the K-Approximate and Metaphone matching algorithms which performed worse in general than the Soundex method which was not only more accurate but faster.

Conversely though, the Guth name-matching algorithm, which required similar execution-time to the Soundex algorithm, was less accurate and tended to determine more names to match which were unrelated in reality. It did, however, achieve more true matches than the other three methods (except when applied to long names) due to its tendency to match names more readily.

The choice of a name-matching method would seem to depend largely on the required application and desired results. If an algorithm was required to find as many possible matches in a data sample with a given name, then the Guth algorithm is fast and is likely to achieve more true matches than the other methods tested, even when applied to 'corrupted' data. In doing so it will almost certainly match more names that are truly unrelated than the other methods that are more accurate overall (when correct mismatches are also considered).

In this investigation, no matching algorithm correctly determined more than 40% of the truly matching names to match so that this is a definite area in which to seek for improvement in overall performance. Obviously any decrease in execution-time for an algorithm will be beneficial, providing this does not result in less accuracy as a consequence. In this respect, it is possible to 'pre-code' the names in the data sample with methods such as Soundex and Metaphone whereby each name is first coded using the algorithm, then the name-matching is determined by comparing the codes for each name. Although this approach was not considered in this investigation (in order to compare the algorithms on a more equal basis), it would undoubtedly speed up the process of comparing names.

An alternative technique, which does not rely on 'pre-coding' of names, would be to code the two names simultaneously and reject the match as soon as a mismatch was encountered in the two generated codes. This would marginally decrease the time to compare names, especially those with different first letters (which are automatically rejected as mismatches by the Soundex method).

This leads to a second possible area of improvement, since the Soundex method in particular, and the Metaphone method to a certain degree, ignore the possibility of two names being equivalent if their first letters are different. This is especially true with names beginning with vowels (where many such equivalents exist) such as EWELL and ULE which are truly equivalent but neither method recognizes as matching. However, it may be the case that concentrating on improvements that handle specific cases will

result in a more complicated algorithm which consequently takes longer to execute (such as is already true of the Metaphone algorithm).

A possible improvement to the Guth algorithm, which has already been suggested, is to include a measure of confidence in a match between a pair of names to reduce the number of 'false positive' matching errors. This could be based on the number of character comparisons necessary to determine a match, i.e. the more character positions compared that did not match, the less likely the names are to really match.

It is apparent that the choice of algorithm to use depends on the nature of the specific application and that there is no 'best' algorithm overall. The K-Approximate method, which is not designed specifically for matching names, performs reasonably but takes much longer to achieve similar results to the Soundex method. Gloria Guth's algorithm, which is also non-specific in application, performs comparably and achieves the highest percentage of true matches. It would seem that the additional complexity of the Metaphone coding method does not really gain any benefits over the Soundex technique and results in a longer execution time.

The Soundex method would appear to have the most merits, especially when considering overall accuracy, execution time and the simplicity of the algorithm. It is let down only by the percentage of true-matches determined since it fails to match over 60% of truly matching pairs; a characteristic that is possibly due to its failure to match many names that start with different letters. For a method that has no intrinsic language or phonetic knowledge, the Guth algorithm also performs well, and determines the highest percentage of true matches.

In order to attempt to devise any form of improved algorithm, it would be necessary to consider the execution time and percentage of true matches determined in addition to any increased overall accuracy. It is possible that a modified form of the Soundex or Guth algorithm could perform better but it is important not to incur a time penalty caused by additional complexity. It is apparent that an algorithm that has a better 'understanding' of the concepts of name equivalence may be more effective (such as a neural computing approach) but this again could result in a much slower technique. Another alternative would be to use a 'knowledge base' of names known to match and develop an expert system to determine if a pair of names matches. Unfortunately this is likely to require a *very* large 'knowledge base' and would be less portable than a simple matching algorithm.

There are advantages and disadvantages of all techniques considered, mainly concerning accuracy and execution-time, but it is also important to consider the application of name-matching methods to multi-ethnic populations (a factor not investigated in this project) where methods such as the K-Approximate and Guth matching techniques may benefit due to their lack of inherent language specificity.

9. An Improved Name-Matching Algorithm

The above analysis and conclusions, from the results of the executions of the comparison program, have identified a number of objectives to be targeted in designing an 'improved' name-matching algorithm. Any improvements can be made in three principal domains:

- Overall accuracy (and percentage of true matches determined)
- Execution-time
- Simplicity of design

Any algorithm that can achieve better results in any one of the above fields, without forfeiting performance in either of the other two, can be considered to be an 'improved' method. What we aimed to do was to produce a simple algorithm which achieved an increase in the percentage of true matches while maintaining a high level of overall accuracy, and without sacrificing performance.

From the findings of the statistical comparison of the existing name-matching methods, it was the Soundex coding method that demonstrated best that a simple algorithm, indeed one whose implementation was fundamentally trivial, could produce reasonably accurate results in a relatively short execution-time (even shorter if 'pre-coding' of the names is employed). Hence, it was decided that the Soundex coding method offered the greatest scope for beneficial modifications.

Areas of possible improvement to the algorithm were identified using the findings from the above comparison experiments. Since the Soundex method relies largely on the phonetic structure of names, an

investigation was made of whether improvements could be made by fine-tuning the algorithm and improving its handling of special cases of phonetic equivalency.

The only drawback with this approach is that the resulting algorithm is likely to be more specialised to the target language (English in this case) which could make it less effective when comparing names from a multi-ethnic population. However, it must be considered that even within one language, the range of name-variations is diverse and it is unrealistic to expect to develop a method that can match names in all target languages.

9.1. Possible improvements

From the results of the comparison of existing name-matching algorithms, a number of observations can be made on the performance of the Soundex coding technique:

The method currently only matches names with the same first letter, but it is apparent that this does not apply to many truly equivalent names (e.g. EWELL and ULE or FILP and PHILP, which would otherwise be considered to match). Therefore, if the method was modified to also consider phonetic equivalents of the first letter when coding names, the overall performance could be improved.

By modifying the comparison program to output those pairs of names which a method should have matched but did not, it was possible to manually identify trends in equivalent names that were not already handled by the algorithm. The following observations were made:

- Many names beginning with a vowel are equivalent to others beginning with a different vowel but the same following phonetic structure. There are also cases where the first letter is a ‘Y’ that also match with other names beginning with a different vowel (such as YULE and EWELL).
- Many names beginning with an ‘H’ followed by a vowel are equivalent to names with the H omitted (e.g. HEAMES and EAMES).
- The Soundex method does not consider phonetic equivalents such as ‘KN’ and ‘N’, ‘PH’ and ‘F’, or ‘WR’ and ‘R’, which are present at the beginning of many names (e.g. KNEVES and NEVES, or PHILP and FILP).
- Certain letters may be present on the end of names that are equivalent to names with these letters omitted, such as ‘G’ or ‘D’ where they are preceded by and ‘N’ (e.g. SLIMIN and SLIMING), or S (e.g. RIVER and RIVERS).
- Within many names the letters ‘R’ and ‘L’, when followed by another consonant, may be omitted from other equivalent names (e.g. CORLEY and COLEY, or CARTON and CARLTON).
- The Soundex method codes ‘TCH’ and ‘CH’ differently when these are generally phonetically equivalent as in CACHPOLE and CATCHPOLE.
- Other single letter phonetic equivalents are not considered by the method such as ‘P’ and ‘B’, ‘V’ and ‘F’, ‘J’ and ‘G’, ‘C’ and ‘K’, ‘Z’ and ‘S’, or ‘Q’ and ‘C’, which are equivalent in many name variations.

Many of the above cases are included in the phonetic processing of the Metaphone coding technique which is largely specialised to matching data in the English language. Although modifying the Soundex method to consider these cases will not improve its handling of corrupted name data or multi-ethnic data, it will allow the method to correctly match many name-pairs that it would otherwise mismatch. In compiling the above list it was important to consider the implications of handling these cases on the number of ‘false positive’ errors when applying the algorithm. Although adding handling for a phonetic equivalence may appear to improve the performance for a certain class of names, in general it may cause the method to match more names which are not related.

9.2. Implementation Technique

For each proposed modification to the existing Soundex method code we observed the effect it had on the overall performance of the algorithm; this was determined by comparing the matching statistics for the revised algorithm with those for the algorithm without the modification. Only those modifications that produced a significant improvement in the number of true matches without reducing the overall accuracy (due to additional ‘false positive’ errors incurred) were considered. Since the original algorithm tested

achieved an overall accuracy of 99.61% and determined 36.37% of true matches, it was considered that if a change resulted in the overall accuracy dropping below 99.0% it was not worthwhile.

An initial experiment, which included the coding of all vowels as a '?' character, found that a figure of 76.31% of true matches determined could be achieved but this resulted in lowering the overall accuracy to only 96.29%. This was due to the revised algorithm matching names more readily and resulting in more 'false positive' matches. This reduction of over 3% in the overall accuracy of the method was regarded as unacceptable despite the high percentage of true matches that were correctly determined.

Another attempt to improve the performance of the algorithm was to categorise the first letter of names beginning with a vowel into two groups; those beginning with 'A', 'O' or 'U', and those beginning with 'E', 'I', or 'Y' (since 'Y' has a phonetic sounding which is similar to that of 'E' and 'I'). This had promising results since this resulted in a 3% increase in the percentage of true matches determined while only reduced the overall accuracy by approximately 0.2%. This choice of two vowel groups was influenced by the distribution of equivalent names that began with a vowel. It was later determined that coding all vowels as the same letter (an 'A') gave even better overall results (an additional 1% increase in overall accuracy without a reduction in the percentage of true matches determined).

Each of the other observations in the above list of trends observed in names not correctly matched by the original method was taken in turn as the basis of a further modification and the resulting effect on the overall performance of the algorithm observed. By this means we arrived at an 'improved' name-matching method (when applied to names in the English language) that implements phonetic processing to handle all of the observed phonetic equivalence trends not considered by the original method. Since some of the coding ideas are based on those employed in the implementation of the Metaphone method, the name ***Phonex*** seemed most appropriate, encapsulating the notion of a combination of the two methods (Soundex and Metaphone).

9.3. The Phonex Algorithm

The algorithm converts each name to a four-character code, which can be used to identify equivalent names, and is structured as follows:

Pre-process the name according to the following rules:

1. Remove all trailing 'S' characters at the end of the name.
2. Convert leading letter-pairs as follows:

   KN → N　　　　WR → R

   PH → F
3. Convert leading single letters as follows:

   H → Remove

   E, I, O, U, Y → A　　　　K, Q → C

   P → B　　　　J → G

   V → F　　　　Z → S

Code the pre-processed name according to the following rules:

1. Retain the first letter of the name, and drop all occurrences of A, E, H, I, O, U, W, Y in other positions.
2. Assign the following numbers to the remaining letters after the first:

   B, F, P, V → 1

   C, G, J, K, Q, S, X, Z → 2

   D, T → 3　　　　If not followed by C.

   L → 4　　　　If followed by vowel or end of name.

   M, N → 5　　　　Ignore next letter if either D or G.

R → 6 If followed by vowel or end of name.

3. Ignore the current letter if it has the same code digit as the last character of the code.
4. Convert to the form ‘letter, digit, digit, digit’ by adding trailing zeros (if there are less than three digits), or by dropping rightmost digits it there are more than three.

Although the resulting four-character code is identical in format to that produced by the Soundex coding algorithm, these two forms are not compatible.

10. An Analysis of Phonex

The comparison program, developed for comparison of the four existing name-matching techniques, was modified to also allow comparison of the improved method, the Phonex algorithm. To maintain accurate and representative results to for later comparison the same test environment and data sets were used as for the original comparison of the four existing methods (see Chapter 8). The modified comparison program was executed and the additional results for the Phonex method observed (the results for the other methods were unchanged since no modification had been made to their implementations).

Since the test environment used for the executions of the Phonex method was identical to that used for the comparison of the four existing methods, a direct comparison of the existing results with those for the improved algorithm was possible.

In comparison to the existing methods, the execution time of the Phonex algorithm is, as expected, marginally longer than that of the Soundex algorithm on which it is based. This is due to the additional complexity of the algorithm enabling it to achieve a better overall performance. The algorithm was seen to take approximately 20% longer than the Soundex algorithm, which is still significantly less than the Metaphone and K-Approximate methods. The final statistic considered, the number of character comparisons per name-pair required by each method, showed a 10% increase in the figure for the improved algorithm on that for the Soundex method.

The following charts illustrate the combined set of results in a graphical form allowing direct comparison of the relative accuracy of the existing and improved algorithms, on the original full data sample:

The accuracy of the improved algorithm (Figure 12(a)) shows a 0.18% decrease (from 99.61% to 99.43%) due to fewer mismatches being correctly determined. As with the Guth method, although not to the same extent, this is due to the algorithm matching names more readily. This reduction is within the bounds determined in developing the algorithm (see Chapter 9) and is only small when considering the difference between the Soundex and Guth algorithms, the latter achieving only 97.42% overall accuracy.

When considering the marginal reduction in overall accuracy and the more significant increase in the percentage of true matches determined (Figure 12(b)), the increase in overall performance of the improved algorithm over the Soundex method is more apparent. The ***Phonex*** algorithm correctly identifies 51.79% of true matches meaning it is the only algorithm to determine over half of the matches correctly. In comparison, the Soundex method achieved only 36.37%, and the Guth (which was the highest of the four existing algorithms) method only 39.89%.

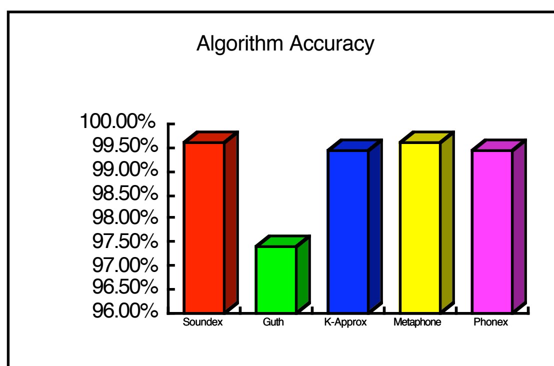


Figure 12 (a)

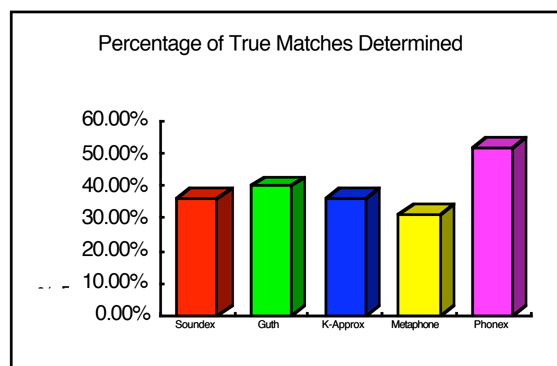


Figure 12 (b)

The results for the improved algorithm followed the same trends as the Soundex algorithm when applied to the corrupted data samples. This was also found to be true for the execution-times, accuracy, percentage of true matches determined, and the number of character comparisons per name-pair.

Where the corrupted data set caused an increase in a statistic for the Soundex algorithm, there was a corresponding increase in the statistic for the Phonex algorithm. This increase was proportional to the difference between the values of that statistic for the two methods. This is most likely due to the two algorithms sharing the same basic structure, with the Phonex method including phonetic pre-processing of names, which accounts for the increased execution-time and number of character comparisons.

The two sub-sample data sets, of names with 5 letters or less and names with 9 letters or more, were used to determine if the improved algorithm performed better on shorter or longer names. Since the data contained fewer names than the full and corrupted samples, the execution-times for the algorithm would be significantly less and hence a comparison of the times would be meaningless. Otherwise, the same correlation between the results for the Soundex and Phonex algorithms was seen in the statistics for the large and small name samples.

To sum up, the trends in performance of the Phonex method were seen to follow those of the Soundex method for the original, corrupted and reduced data sets of short and long names. This is probably to be expected when considering the structural similarity of the two algorithms.

The difference between the two methods is seen in the overall performance, where the following observations can be made:

- The execution time of the Phonex method is approximately 20% longer than for the Soundex method.
- The overall accuracy of the Phonex method is marginally lower (approximately 0.2%) than that of the Soundex method.
- The percentage of true matches determined by the Phonex method is approximately 44% higher than that of the Soundex method.
- The Phonex algorithm required approximately 10% more character comparisons per name-pair than the Soundex algorithm.

These figures show that the improved algorithm forfeits a certain degree of performance in execution-time and overall accuracy, for an impressive increase in the percentage of true matches determined. This improvement certainly justifies the marginal increase in execution-time and the equally small decrease in overall accuracy.

Since the overall accuracy in this investigation is largely weighted by the number of correctly determined mismatches, the number of mismatches that are not identified by the improved algorithm has a higher effect on this figure than the number of matches that are identified. In the majority of applications, the algorithm would be used for finding a useful set of matches for a given name (an equivalence class), and in this case the improvement in the percentage of true matches determined is more important.

In this respect the Phonex algorithm is definitely 'improved' with respect to the overall performances of the four existing algorithms tested. It is possible that further improvements could be made to increase the percentage of true matches determined without a decrease in overall accuracy resulting. However, it must be considered that the algorithm itself employs a high degree of generalisation in comparing names. It is therefore easy to identify particular equivalent name-pairs that the algorithm will fail to match and, conversely, unrelated name-pairs that the algorithm determines to match.

The algorithm is limited by the manner of its approach, which relies on this generalisation to identify classes of equivalent names. Adding additional rules, to handle special cases for particular name-pairs, could result in the algorithm becoming too specific and failing to recognise equivalence in more general cases. Likewise, making the algorithm less specific would cause it to overlook many phonetic equivalents that are particular to certain name equivalence classes.

The Phonex name-matching algorithm is a compromise between generality and specificity, and achieves a comparatively good overall performance when applied to names in the English language. Current name-matching methods, including the Phonex algorithm, fall into one of two categories, those which consider the phonetic structure of names, and those which consider the names on a character-by-character basis. Both of these approaches have advantages and disadvantages that make them better or worse when applied to a specific task. The phonetic-based approaches (such as Phonex) are somewhat more specific to a

particular language but implement a better appreciation of names that sound similar. Methods that rely on analysing the positions of characters in name-pairs (such as the Guth algorithm) perform better when applied to names that contain similar characters but may also contain characters that give the names a different phonetic structure. Such methods tend to be more portable between languages.

There is of course a large degree of compromise within the field of name-matching between speed and complexity, and between generality and specificity. Additionally, it appears that in many cases improving an algorithm to give a higher percentage of correctly determined matches causes the algorithm to match names more readily and so gives a lower percentage of correctly determined mismatches. The Phonex algorithm suffers from this phenomenon to a certain degree, although the increase in the percentage of correctly determined matches far exceeds the decrease in the percentage of correctly determined mismatches.

11. Concluding Remarks

This investigation took advantage of the recent publication of a large, and in effect authoritative, list of British surnames and their variant spellings in order to perform an assessment of existing name-matching techniques and develop an 'improved' matching algorithm based on the results of this assessment.

The name-matching techniques covered by this investigation comprise only a small selection of those existing, but they are representative of many of the current approaches to the problem of name-matching. Due to the diversity of applications for name-matching techniques, choosing a particular algorithm will depend largely on the nature of the data it is to be applied to. Also to be considered is whether the algorithm is in effect employed to provide a comprehensive equivalence class for a particular name, which may include additional incorrect matches, or an accurate list of equivalent names that may be incomplete. Since some name-matching algorithms link pairs of names more readily, these may be more suited to the former of these applications, whereas methods that match names more critically may be more suited to the latter.

Many reports on current name-matching techniques quote specific cases of name-pairs that a particular algorithm fails to determine correctly. Although this may highlight certain classes of names that the algorithm does not perform well on, it must be considered that overall the algorithm may have a very high accuracy. This problem seems to arise from the degree of generalisation employed by those algorithms that consider the phonetic structure of the names being compared; this is necessary to allow the algorithm to identify classes of names that are equivalent. In achieving this, there will inevitably be exceptions to the rules on which the algorithm depends due to the diversity of name variations.

Since name-matching algorithms are employed largely to avoid lengthy and laborious manual analysis of data, provided they can achieve an acceptable level of accuracy, the time saved by their application is an obvious advantage. However, it cannot be expected that such an automated approach to name-matching will achieve 100% accurate results. (Even when performing the task manually it is likely that human-error will cause a certain degree of inaccuracy.) Thus, as mentioned earlier other information such as address, date of birth, and/or occupation, if available, will often need to be taken into account.

From the results we have obtained from the comparison of selected existing name-matching techniques, it was possible to identify areas of improvement to the design of these methods. The Phonex name-matching method, based on the Soundex coding technique, incorporated measures to attempt to overcome some of the limitations of these existing methods, without incurring a penalty of significantly increased complexity or execution-time.

Although it is true that the Phonex algorithm is marginally more complicated, and its execution-time slightly longer than that of the original Soundex algorithm on which it based, we would argue that its increased overall performance justifies both of these characteristics. The algorithm correctly determines an impressively higher percentage of true matches than the Soundex coding method without compromising significantly on performance in other areas.

However, in achieving this increase in performance it was necessary to introduce additional phonetic rules to handle a number of exceptional cases, making the algorithm more specific to the English language. Consequently it is likely it would be less effective than the Soundex algorithm when applied to data in other languages.

References

[Baase1989]: Sara Baase, Computer Algorithms, Introduction to Design and Analysis, Second Edition, Addison Wesley, 1989, pp242-244.

[Bouchard1986]: Gérard Bouchard, The Processing of Ambiguous Links in Computerised Family Reconstruction, Historical Methods, Vol. 19, No. 1, Winter 1986, pp9-19.

[Bouchard&Pouyez1980]: Gérard Bouchard and Christian Pouyez, Name Variations And Computerised Record Linkage, Historical Methods, Vol. 13, No. 2, Spring 1980, pp119-125.

[DeBrou&Olson1986]: David De Brou and Mark Olsen, The Guth Algorithm and the Nominal Record Linkage of Multi-Ethnic Populations, Historical Methods, Vol. 19, No. 1, Winter 1986, pp20-24.

[Floyd1993]: Su Floyd, The Family History Transcription System, Computers in Genealogy, Vol. 4, No. 11, September 1993, pp485-486.

[Guth1976]: Gloria J. A. Guth, Surname Spellings and Computerized Record Linkage, Historical Methods Newsletter, Vol. 10, No. 1, December 1976, pp10-19.

[Knuth]: D. E. Knuth, The Art Of Computer Programming, Vol. 3, Sorting and Searching, Addison Wesley, pp391-392.

[Mavrogeorge1993]: Brian Bonner Mavrogeorge, Coding and Techniques, March 1993

[Park1992]: Keith & Tracy Park. Family History Knowledge UK 1992/93, Family History Club, Mountain Ash, Mid-Glamorgan, UK. 1088p. ISBN 1-873594-04-6

Appendix: Details of the Phonex Algorithm

The modifications to the original code are in two forms; phonetic pre-processing of the name (prior to applying a modified form of the coding algorithm), and revisions to the coding algorithm itself. For convenience it was also found necessary to implement a function that determined whether the character argument passed was a vowel (including the letter 'Y') or not.

This function `isvowelY` is identical to the `vowel` function of the Metaphone coding algorithm with the exception of an additional test to consider 'Y' to also return 'true'. The function returns the integer value 1 if the argument is a vowel or a 'Y', and 0 otherwise.

In most respects the main body of Michael Cooley's original coding algorithm is unchanged, maintaining the same iterative structure to code the letters of the name one by one, replacing them with their coded forms until the code contains exactly four characters. The modifications are largely in the determination of when certain code characters should be applied.

The coding function itself, `phonex`, has the same functional form as the original function, taking a single `char*` argument representing the name to be coded (`name`) and returning an integer value to establish the successful completion of the function (the integer value 1). The coded form of the name is returned implicitly by replacing the letters of the original name with the new four-character code.

Considering the phonetic pre-processing code first, this additional code executed prior to the main coding body of the function, handles specific phonetic equivalents of the first letters of names and deletes redundant first and last letters of the name.

The first pre-processing code segment removes any 'S' characters from the end of a name since these were found to cause many truly equivalent names to be considered not to match by the algorithm. A `while` loop is used to replace the 'S' with a '\0' (string-terminating character) until the last letter of the name is no longer an 'S'.

Following this, three double-letter phonetic equivalents are processed (identified using the C++ `strcmp` function in the standard String.h file, although direct character comparisons could have been used). The two cases 'KN' and 'WR', being equivalent to 'N' and 'R', are handled by simply replacing the first letter of the name with the second. This creates a duplicate first letter, the second of which will be ignored due to the manner in which the coding algorithm handles duplicated letters.

The case of 'PH' being phonetically equivalent to 'F' is handled slightly differently in that the first letter of the name is replaced with an 'F', leaving the following 'H' unchanged. This 'H' will be subsequently ignored by the coding algorithm since it is considered to have no phonetic significance in the name (as in the original coding algorithm).

The original algorithm did, however, code an 'H' in a name if it was the first letter. This practice is not upheld by the Phonex method since in many cases names starting with an 'H' are found to be equivalent to similar names without the 'H', due to the 'H' having no effect on the phonetic sounding of the name (generally starting with a vowel sound). There were no cases found in the test data where an 'H' starting a name was followed by a consonant that may have been affected by its omission.

The omission of 'H' characters at the start of a name is implemented by replacing the 'H' with the second character of the name, creating a duplicate first character. This duplication is again ignored by the coding algorithm so no change is made to the phonetic structure of the name and resulting code.

The final phonetic pre-processing code segment handles all other singular character phonetic equivalents using a simple C++ `switch` statement. This codes all first letter vowels as an 'A', a first letter 'P' as 'B', 'V' as 'F', 'K' or 'Q' as 'C', 'J' as 'G' and 'Z' as 'S', in accordance with the list of phonetic equivalents not handled by the original method. The 'U' following a 'Q' in most cases will be ignored by the coding routine so will not affect the phonetic sounding of the name that will begin 'CU' after replacement of the 'Q'. Although this is a rather limited special case, since only very few names begin with a 'Q', its addition did not incur a significant time penalty to the execution of the algorithm and improved the percentage of true matches determined.

The main body of the coding algorithm is structured as a `for` loop that increments a `char*`, `p`, starting from the first character in the name. This is initialised to point to the same position as `name` (the `char*` pointing to the first character in the name), then incremented, one character at a time, while the character pointed to is not a space, comma, or the end of string character (ASCII character 0). An additional constraint on the iteration is imposed so that the loop will terminate when 4 characters have been added to the code, maintained by the integer variable, `y`, which is the index of the last character added to the code array (i.e. range from 0 to number of characters in code - 1).

At each iteration of the `for` loop a `switch` statement processes the current character in the name and determines what corresponding digit to add to the name's code (if any), represented by the `char`, `code`. The coded form of the name is created by overwriting the first four characters of the name string and following them with a string terminating character. This is essentially a destructive method since the original name passed to the function will be lost, but it is fast and efficient.

The format of the code is the same as for the method (i.e. letter, digit, digit, digit) but the manner in which these code characters are determined is slightly different. The majority of the original character equivalents are retained, using the following translations of letters in to code digits (the first letter of the name is treated separately and is added directly to the code as a letter not a digit):

b, f, p, v → 1	l → 4
c, g, j, k, q, s, x, z → 2	m, n → 5
d, t → 3	r → 6

The Phonex method does, however, slightly modify this coding technique in that the letters 'D', 'T', 'L', 'M', 'N', and 'R' are subject to further processing before determining what code digit to add:

- If a 'D' or 'T' is followed by a 'C', the 'D' or 'T' is not coded since it is considered that, as in 'TCH' and 'CH', the omission of the 'D' or 'T' will enable more true matches to be identified.
- An 'L' is only coded if it is followed by a vowel, or it is the last character of a name.
- If an 'M' or 'N' is followed by a 'D' or 'G' the following letter is overwritten with a duplicate of the current (which will then be ignored), since these letter combinations are considered to phonetically equivalent.
- An 'R' is only coded if it is followed by a vowel, or it is the last character of a name.

All of these special cases are implemented with direct character comparisons (using `if` statements) and taking appropirate action to determine whether to add a code digit to the code, ignore the current character, or ignore the following character.

The code section that follows the `switch` statement determines whether to add the code character to the code or ignore it. The code character is not added if it is either the same as the last character coded (`last`), an ASCII 0 character (representing a vowel, etc.), or the first letter of the name. The value of the `last` variable is normally the last character added to the code, but an exception is made for the first character, where the value of `last` is the coded numeric form of the character (since the first character is added as a letter not a digit).

Following the `switch` statement a `while` loop is used to add trailing '0' characters to the code if it is less than four characters long. Finally the string terminating character is added to the code, and the `phonex` function returns.

The Implementation of the Phonex Algorithm

```
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//  File name: phonex.H                                                   //
//                                                                        //
//  Purpose: Function declarations for Phonex name-matching method.____  //
//                                                                        //
//  Author : A.J.Lait________   Created: 13/03/95    Modified: 19/03/95   //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

#ifndef PHONEX_H
#define PHONEX_H

int     phonexMatch(char* name1, char* name2);
int     phonex(char* name);
int     isvowelY(char c);

#endif

////////////////////////////////////////////////////////////////////////////
//                                                                        //
//  File name: phonex.CPP                                                 //
//                                                                        //
//  Purpose: Function definitions for Phonex name-matching method._____  //
//                                                                        //
//  Author : A.J.Lait________   Created: 13/03/95    Modified: 20/03/95   //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

#include "defs.h"
#include "phonex.h"
#include <ctype.h>
#include <string.h>
#include <iostream.h>

extern unsigned long comps;


int phonexMatch(char* nameA, char* nameB)
// calculates phonex codes for two names
// and returns 1 if codes match
{
  char  name1[MAXLINE]; strcpy(name1, nameA);
  char  name2[MAXLINE]; strcpy(name2, nameB);
  int   match = 0;

  // turn names into phonex code equivalents
  if (!phonex(name1))
  {
        cerr << "Error coding name1." << endl;
  }
  else if (!phonex(name2))
  {
        cerr << "Error coding name2." << endl;
  }
  else if (strcmp(name1, name2) == 0)
  {
        // phonex codes are the same
        match = 1;
  }

  return match;
}


/* The following C function to create Phonex codes is converted from
   SDX v1.1 coded by Michael Cooley. Released to the public domain, 1994. */
int phonex(char* name)
{
        char last,code,*p;
        int y=1;
```

```
//
// PHONEX PRE-PROCESSING ...
//

// Deletions effected by replacing with next letter which
// will be ignored due to duplicate handling of Soundex code.
// This is faster than 'moving' all subsequent letters.

// Remove any trailing Ss
while (name[strlen(name)-1] == 'S')
{
        name[strlen(name)-1] = '\0';

        comps += 1;
}

// Phonetic equivalents of first 2 characters
// Works since duplicate letters are ignored
if (strncmp(name, "KN", 2) == 0)
{
        *name = 'N';            // KN.. == N..

        // AMENDMENT
        comps += 2;
}
else if (strncmp(name, "PH", 2) == 0)
{
        *name = 'F';            // PH.. == F.. (H ignored anyway)

        // AMENDMENT
        comps += 2;
}
else if (strncmp(name, "WR", 2) == 0)
{
        *name = 'R';            // WR.. == R..

        // AMENDMENT
        comps += 2;
}

// Special case, ignore H first letter (subsequent Hs ignored anyway)
// Works since duplicate letters are ignored
if (*name == 'H')
{
        *name = *(name+1);

        // AMENDMENT
        comps += 1;
}

// Phonetic equivalents of first character
switch(*name)
{
        case 'E':
        case 'I':
        case 'O':
        case 'U':
        case 'Y':
                // vowel equivalence A = E, I, O, U, Y
                *name = 'A';
                        // AMENDMENT
                        comps += 5;
                break;
        case 'P':
                // consonant equivalence B = P
                *name = 'B';
                        // AMENDMENT
                        comps += 6;
                break;
        case 'V':
                // consonant equivalence F = V
                *name = 'F';
                        // AMENDMENT
                        comps += 7;
                break;
        case 'K':
        case 'Q':
                // consonant equivalence C = K
                *name = 'C';
```

```
                        // AMENDMENT
                        comps += 9;
                break;
        case 'J':
                // consonant equivalence G = J
                *name = 'G';
                        // AMENDMENT
                        comps += 10;
                break;
        case 'Z':
                // consonant equivalence S = Z
                *name = 'S';
                        // AMENDMENT
                        comps += 11;
                break;
        default:
                // no equivalence for letter, no change
                break;
}

//
// MODIFIED SOUNDEX CODE
//

p=name;
for ( ;
        *p!='\0'        /* not at end of name */
        && *p!=' '      /* terminate if encountered */
        && *p!=','      /* terminate if encountered */
        && y < 4;       /* code no more than 4 characters */
        p++)
{
        switch(*p)
        {
                case 'B':
                case 'P':
                case 'F':
                case 'V':
                        code='1';
                                // AMENDMENT
                                comps += 4;
                        break;
                case 'C':
                case 'S':
                case 'K':
                case 'G':
                case 'J':
                case 'Q':
                case 'X':
                case 'Z':
                        code='2';
                                // AMENDMENT
                                comps += 12;
                        break;
                case 'D':
                case 'T':
                        if (*(p+1) != 'C')
                        {
                                code='3';
                        }
                        // else do not code: TC.. = DC.. = C..
                                // AMENDMENT
                                comps += 14;
                        break;
                case 'L':
                        if (isvowelY(*(p+1)) || (*(p+1) == '\0'))
                        {
                                // only code if followed by vowel of end of name
                                code='4';
                        }
                                // AMENDMENT
                                comps += 15;
                        break;
                case 'M':
                case 'N':
                        if ((*(p+1) == 'D') || (*(p+1) == 'G'))
                        {
                                // NG.. = ND.. = N..
                                *(p+1) = *p;
```

```
                                }
                                code='5';
                                        // AMENDMENT
                                        comps += 17;
                                break;
                        case 'R':
                                if (isvowelY(*(p+1)) || (*(p+1) == '\0'))
                                {
                                        // only code if followed by vowel of end of name
                                        code='6';
                                }
                                        // AMENDMENT
                                        comps += 18;
                                break;
                        default:
                                code=0;
                                break;
                }

                // AMENDMENT
                comps += 3;

                if(
                   last!=code                   /* not the same as previous position */
                   && code!=0                   /* is not a vowel, etc. */
                   && p!=name)                  /* is not the first letter */
                        name[y++]=code;         /* then code it */

                last=name[y-1];
                if (y==1) last=code;            // special case for 1st letter
        }

        while(y<4) name[y++]='0';               /* fill in with trailing zeros */
        name[4]='\0';                           /* NULL terminate after 4 characters */
        return 1;
}


int isvowelY(char c)
// returns 1 if c is a vowel or Y, 0 otherwise
{
  // convert to upper case
  c = toupper(c);

  // AMENDMENT: average no. of comparisons
  comps += 3;

  return ( (c == 'A') ||
           (c == 'E') ||
           (c == 'I') ||
           (c == 'O') ||
           (c == 'U') ||
           (c == 'Y') );
}
```