A SPELLING CHECKER FOR DYSLEXIC USERS: USER MODELLING FOR ERROR RECOVERY

Submitted for the degree of
Doctor of Philosophy

Roger Ian William Spooner

Human Computer Interaction Group, Department of Computer Science,
University of York, Heslington, York, YO10 5DD

September 1998

Abstract

Although spelling correction has been a domain of interest amongst computer scientists since the subject's earliest days, it has still not achieved complete success. This thesis describes research into the correction of spelling errors by dyslexic people. To pursue higher accuracy in this environment of more severe errors, a user model was employed to capture patterns and direct the search for corrections. The resulting system, 'Babel', was tested with and without the user modelling components and by comparison to previous systems. It was found to perform significantly better than some systems in some cases, and not to be worse, but failed to convincingly prove the merit of user models. This is most likely to be due to there being more variance in errors within work by an author than between authors. The errors used in testing were from a corpus collected and tabulated specifically for the work, and which is available for future research.

An algorithm for approximate string matching, ALGORITHM T is presented which performs the kind of search required in a spelling checker; when given a misspelt word it finds the entries in a dictionary which are most similar to it as measured by an edit cost. The algorithm is faster than all comparable methods and has fewer constraints than many, achieving a speed increase of 150 times over the most similar method.

Contents

Abstract 1
Acknowledgements 7
Declaration 8

1 Introduction **10**
1.1 Rationale 10
1.2 Thesis Overview 12

2 Spelling Correction **16**
2.1 Overview 16
2.2 Name Spellings: Soundex 18
2.3 Computerised Correction 20
2.3.1 Human errors 20
2.3.2 Mechanical Errors 23
2.4 Scientific Texts: SPEEDCOP 25
2.5 A-Star Search 29
2.6 Approximate String Matching 31
2.7 Approximate String Matching for Biology 36
2.7.1 Efficient Dictionary Short-Listing 38
2.8 Phonetic Error detection 41
2.8.1 Phonetic Coding for poor spellers 44
2.9 N-gram methods 49
2.10 The combination of methods 53
2.11 Associative Memory 57
2.12 Other Work 58
2.13 Conclusions 58

3 Language and Dyslexia **60**
3.1 Written Language 61
3.2 Dyslexia 63
3.3 Diagnostic Methods 67
3.4 Specific Error Types 69
3.5 Cognitive Language models 72
3.6 Current technology for helping dyslexic people 76
3.7 Summary 79

4 User Modelling **81**
4.1 Overview 81

4.2 Stereotypes 83
4.3 Production Rules 86
4.3.1 Mal-Rules in Arithmetic 86
4.3.2 Mal-rules in Language Production 92
4.4 Feature Based Modelling 93
4.5 Bayesian Networks 97
4.6 Dempster-Shafer and Fuzzy Logic 100
4.7 Fuzzy Logic 102
4.8 Summary 103

5 System Design of BABEL **107**
5.1 Introduction 107
5.1.1 Input and Output 109
5.2 Rules 110
5.2.1 Generality score 119
5.3 Cost adjustment 121
5.4 Path choice heuristic 124
5.4.1 Pseudo Code 125
5.4.2 Run Time 130
5.5 Training search 131
5.6 Ordinary use 132
5.7 Worked Example 133
5.8 Conclusions 137

6 Sample Text Collection 138
6.1 Transcription 140
6.2 Java Experiment 144
6.2.1 Objectives 145
6.2.2 Presentation 146
6.3 Task choice 150
6.4 Participation 151
6.5 Results 152

7 Results from BABEL **155**
7.1 Sample Text 155
7.2 Error Severity 158
7.3 Comparison With Other Software 160
7.4 Correction Improvement 166
7.5 Rule Usage 169
7.6 Distinctive User Models 176
7.7 Rule Use Position 180
7.8 Discussion 184

8 Approximate String Matcher ALGORITHM T **187**
8.1 Background 187
8.1.1 Dynamic Programming 189
8.1.2 Dictionary Filtering 192
8.1.3 On-Line Searching 195

8.1.4 Suffix Trees 196
8.2 Advantages of a Trie 199
8.3 Pseudo-code for ALGORITHM T 201
8.4 Operation of ALGORITHM T 203
8.4.1 Initialisation 203
8.4.2 Search 205
8.5 Enhancements to ALGORITHM T 208
8.5.1 Edge Compression 209
8.5.2 Dictionary Array 211

9 Results from ALGORITHM T **213**
9.1 Data Structure Size 214
9.2 Run Time 216
9.3 Further Improvements 233
9.3.1 Partial Computation of $H[i, *]$ 233
9.3.2 Further Editing Operations 234
9.3.3 Parallel Execution 235
9.4 Conclusions 236

10 Conclusions 239
10.1 Algorithm T 239
10.2 Babel 240
10.3 Sample Text 244
10.4 Original Contribution 246
10.5 Future Work 248

Bibliography 251

List of Figures

2.1 Desirability of deleting letters 20
2.2 Skeleton and Ommision keys for SPEEDCOP 26
2.3 Calculation of a minimal edit cost 32
2.4 A suffix tree for the word 'bananas' 37
2.5 Some edit cost weightings from `editcost` 46

3.1 The 'universal' model of language production 72

4.1 The IPSOMETER Bayesian Network 98

5.1 An overview of the modular interconnection of 'Babel' units. 108

6.1 Part of a transcribed writing file 141
6.2 The Java experiment requesting personal details. 147
6.3 Trial 2 of the on-line Java experiment 148

7.1 The edit cost of sample errors 158
7.2 Proportion of test in with the correct word was suggested 162
7.3 Average position of the correctly spelt word 163
7.4 Suggestion position improvement by Babel 168
7.5 Number of alternative corrections by Babel 170
7.6 Solution path lengths in Babel 172
7.7 Rule usage differences 175
7.8 Differences between User Models 178
7.9 Rule use position within words for handwritten documents 182

8.1 A suffix tree of the word 'bananas' 192
8.2 ALGORITHM T's search order for a small dictionary 200
8.3 Adding a word to a compressed suffix tree 211

9.1 Data structure size for ALGORITHM T 215
9.2 Run-Time for various edit costs in ALGORITHM T 217
9.3 Run-Time for various dictionary sizes in ALGORITHM T 218
9.4 Run-Time for random and English words 220
9.5 The proportion of trie nodes searched 222
9.6 Node count and compression ratio for the trie 223
9.7 ALGORITHM T compared to the work of Du and Chang 225
9.8 ALGORITHM T compared to EDP and Shang and Merrettal 228
9.9 ALGORITHM T compared to EDP and ABM 230
9.10 Run time with and without calculation of unnecessary values 233

List of Tables

2.1 Letter groups for SOUNDEX 18
2.2 Accuracy of various spelling correction algorithms 51

4.1 Two example errors in arithmetic subtraction. 88
4.2 Young and O'Shea's production-rule system 90
4.3 Additional rules for the production system 91

5.1 Letter substitution rules in Babel 112
5.2 Bigraph letter rules in Babel 113
5.3 Wildcard rules in Babel 114
5.4 General letter rules in Babel 115
5.5 General Phonetic rules in Babel 116

7.1 The number of people in each sample population 156
7.2 Summary of significance of improvement by Babel 165
7.3 Number of solution paths for various sample populations 171
7.4 Frequency of use of error pattern rules 173
7.5 Significance of differences between user models 179
7.6 Significance of differences in rule-use position 183

Acknowledgements

The work in this thesis would not have happened were it not for a number of influences both before and during the research. The motivation was born of work with disabled people inspired (and demanded) largely by Philip Whittaker.

During the work at York, my deepest gratitude goes to Alistair Edwards who supervised the work well despite its twists and turns across so many fields of study. I hope that I have done justice to him. My thanks also to other staff and students too numerous to name, who have supported the work in various ways, or who have asked pertinent questions which have made all the difference. In particular I must thank Eva Chen and Maryvonne Lumley.

Anthony Jameson, Helen Pain and Roger Mitton have contributed from afar, for which I give thanks. Paul Nisbet and the staff of the CALL Centre in Edinburgh provided most timely support by inviting me to participate in real-world research which was the most relevant possible to this; I enjoyed it completely.

Collecting sample data could not be done without the cooperation of experts elsewhere. I am indebted to Jill Wringe, Jane Smedley, Emma Smart, the staff of York Support Services, people at the British Dyslexia Association and myriad other indviduals who contributed material such as the participants in the on-line experiment.

My thanks of course to my parents who made it possible for me to do anything, and supported me before and throughout the research.

The work was funded by a research studentship from the Engineering and Physical Sciences Research Council; I thank the nation's taxpayers for the funds and the Department of Computer Science for choosing to give them to me.

Declaration

The work submitted in this thesis was done entirely by the author and has not been submitted for any other degree or publication, with the following exceptions.

- A paper on the design of Babel and in particular the user modelling, as well as some of the results, was published at the conference UM97 (Spooner and Edwards 1997b).
- A presentation on Babel and in particular the aspects relating to dyslexia was given at Dyslexia 1997 with an abstract published in the proceedings (Spooner and Edwards 1997a).
- The speech synthesis package used for phonetic work in Babel was taken as C source code from McIlroy (1974).
- The sample text population HullP⋆ was provided by a student in Hull University, in exchange for an analysis of the same data (Smart 1997).

TO MY PARENTS

Chapter 1

Introduction

> *"After all," said Rabbit to himself, "Christopher Robin depends on Me. He's fond of Pooh and Piglet and Eeyore, and so am I, but they haven't any Brain. Not to notice. And he respects Owl, because you can't help respecting anybody who can spell TUESDAY, even if he doesn't spell it right; but spelling isn't everything. There are days when spelling Tuesday simply doesn't count."*
>
> – The House At Pooh Corner

1.1 Rationale

The work described in this thesis spans a number of areas of interest, drawing knowledge from some and making a contribution to others. It began with an interest in dyslexia (a condition involving difficulty with literacy) and moved through spelling correction to user modelling, taking in cognitive modelling, string matching and the World Wide Web on the way. Finally it returned to a fundamental Computer Scientific algorithm at the heart of spelling correction.

I was particularly interested in dyslexia because of my previous work in educational software with disabled children; the issue of spelling correction had arisen as one not satisfactorily addressed by existing systems in the real world, and it

seemed to present interesting challenges for DPhil research.

There are three main products of this work. The first is a spelling correction system entitled ‘Babel’ which can accept misspelt words and produce a list of proposed corrections, in much the same style as a conventional spelling checker on a word processing system.

The second major result is not an algorithmic one, but a corpus of spelling errors. The errors have been collected from dyslexic and non-dyslexic people in a number of ways including school work, creative writing and typing to dictation. This corpus has been used to support a fair evaluation of the software, and is also available for future work.

The third result of the work is a method for searching for a string within a large set of strings, finding the closest matching string according to a weighted similarity measure called an ‘edit cost’. This method performs a task similar to existing algorithms, but is faster than any existing method.

The spelling error corpus was collected because a system for dyslexic errors would be useless without a genuine demonstration of its functionality on typical errors. Journals are littered with papers which propose the suitability of an algorithm for a task and then fail to demonstrate that it works well for that particular task. For example, many approximate string matching algorithms are proposed for spelling correction tasks and then tested on randomly permuted strings instead of actual errors. The difference can be substantial.

By examining the user models created for use in Babel, the nature of spelling errors can be observed. This work is, as far as the author is aware, the first example of a quantitative study of a substantial corpus of spelling errors by dyslexic people, and is something that was proposed by Pain (1985). In the past, much work has been based on anecdotal studies of individual cases. For example Boder (1973), although

making many valuable insights and being widely cited ever since publication, bases her work on observations of one patient. Occasionally work is based on simple boolean correct / wrong measures of spelling performance in tests; quantitative studies of errors are not generally published.

1.2 Thesis Overview

The problem for individuals with dyslexia when using computers is that their spelling is too poor to be corrected by normal spelling correction software. Such systems were developed to correct simple typing mistakes such as the ommission of a key stroke. Dyslexic people and other poor spellers often make more severe errors which cannot be detected so easily. Dyslexia, and poor spelling generally, is introduced in Chapter 3 (although this work should not be taken as an authority on dyslexia).

Naturally, a system which could be of help to real people would be an ideal end product for the research, although a quantitative study of spelling errors, and of the application of user models to this domain, would also make valuable contributions to other research.

Chapter 2 contains a comprehensive review of spelling correction literature, and Chapter 8 reviews more literature specifically related to approximate string matching. Current spelling checkers tend to take a word written by the user and apply a number of transformations to it in turn, checking the result of each in a dictionary of correct words. Any entries found are suggested, probably in the order in which they are found. This method is quite successful for simple errors but cannot easily be extended to cover a wider range of errors because the number of suggestions would increase exponentially, and the ordering of the list would become a critical problem. The set of errors which has been found to best combine

descriptive power with computational simplicity for human errors in a number of symbol-processing domains of work includes insertion, deletion, transposition and substitution.

It was with this complexity in mind that User Modelling was brought in. If people make patterns of errors which are not random, then perhaps they could be modelled by an adaptive spelling checker which could suggest words more suitable to that person. With such a system, more serious errors (typically errors compounded of a number of more simple ones) could be accommodated by searching the permutation space more deeply instead of more broadly. Chapter 4 surveys User Modelling literature.

For such a user model to be useful, it would be necessary both that spelling errors made by the target population are in some way regular through time, and that the system could capture that regularity usefully. A visual inspection of some material does suggest a consistency, but such formal measurements have not yet been published. Whether it holds for a majority of poor spellers, and whether the consistencies can be captured by a computer is discussed in this thesis.

As the basis of the consistencies to be sought, a cognitive model of language production was studied. Psychologists have spent some decades or even centuries discussing how the human brain produces language. Probably the most popular proposal at the level of detail of interest here is the *dual route* model. It has been used to help define a number of errors which might be caused by incorrectly functioning brains. This is shown in Chapter 3 along with the material on dyslexia, because much of the interest in language processing is derived from the experience of people who suffer specific difficulties.

With information from the various domains in hand, a spelling checker program was written to test the principles proposed. It requires as input a list of word pairs;

the first of each pair is a word written by a poor speller, and the second is the correct word that was intended. The system, Babel, is written in C so as to be run on any reasonable computer, and has been adapted to record its performance for analysis. The system design is described in Chapter 5.

To test the system, a corpus of text samples were collected from dyslexic and other individuals. The material came from a number of sources including primary and secondary schools, as well as college students and Internet users. A significant difficulty in collecting text from poor spellers is that they generally do not like writing, and so little material is available from any given individual or from a particular group.

A program was written in Java to invite people to type sentences to dictation, and to answer some simple questions about themselves. The keystrokes recorded by this program were used as samples of spelling, and might in future be used for more complex analyses of keyboard usage such as typing speed variations.

The sample texts were all corrected by hand to record the correct spelling of the word intended for each given misspelt word. A procedure was also developed for unreadable words and errors involving spaces. The intention with all aspects of the collection of data was to allow flexibility for future analyses as well as providing the spelling errors for the current project. Chapter 6 describes the collection of sample text in more detail.

The Babel system has been tested with errors from real people by measuring the frequency of suggestion of the correct answer, and the position of the correct answer on the suggestion list when suggested. This information is comparable with most other spelling checkers since they generally produce lists of suggestions in the same manner. The results of running Babel, presented in Chapter 7, make such comparisons with two other systems. The system is tested not only in its normal

mode of operation but also in several degenerate ones which reveal information about the User Modelling techniques. As another test of spelling consistency, separate from suggestion accuracy, the user models built by the system have been compared to each other. This is not comparable with other existing systems because user models are unique to this work. Chapter 7 reports the results of these comparisons.

Another line of research, largely distinct from Babel, is presented in Chapter 8. It considers Approximate String Matching and presents a new method, ALGORITHM T, which is capable of identifying the most similar string in a dictionary to one presented up to 150 times faster than the next best algorithms. It is better suited to dictionary based string matching, unlike some of the other recent methods which operate best on linear input streams. The chapter begins with a detailed survey of relevant literature.

Chapter 9 presents a detailed analysis of ALGORITHM T by comparison to other relevant algorithms, and by comparison to itself under various circumstances such as with different dictionary sizes and edit costs. This analysis is valuable not only for its demonstration of ALGORITHM T but as an insight into the computational complexity of such an algorithm under realistic circumstances which are often omitted from presentations of theoretically good algorithms in publications.

Finally, Chapter 10 draws together the work in this thesis to make conclusions, noting those parts of this work which were particularly interesting or which could be improved and pointing the way towards future work.

Chapter 2

Spelling Correction

This chapter contains a review of the literature on spelling correction, particularly from an algorithmic point of view. It begins with an overview of significant methods, and then continues with a more detailed examination of the work in a number of important cases such as Soundex, SPEEDCOP and Approximate String Matching methods.

2.1 Overview

Spelling correction has been in the minds of computer experts for many decades. It has passed through a number of different generations of requirements, each of which has brought about its own specialist solutions. The changing sources of errors and the changing computing resources available have rendered some methods obsolete while others have stood the test of time and continue to be incorporated in current work. The efficiency of some has become unnecessary, and the constraining assumptions of others have become unacceptable.

In early computer work, spelling errors often came about by mechanical errors in

the reading of paper tape which had been punched with patterns of holes which indicated digital information; fragments of card might fill in holes, tears might create new ones or variations in the motor winding speeds might accidentally create or delete the impression of extra holes in the tapes. The software to identify these errors used simple methods which are still used today, and form the basis of approximate string matching.

As more people typed more material the problem of keyboard typing errors became more serious; instead of giving simple commands to the computer, they were writing letters and reports. The wider range of people involved also meant that non-expert typists were using computers. Typical errors here included letter insertions, deletions, transpositions and substitutions but with some keyboard-specific faults like substituting one letter for another on an adjacent key, or inserting another copy of a letter after it has already been typed.

Dyslexic people and other poor spellers have difficulty spelling words correctly, but have a number of particular error types which are described in more detail in Chapter 3. The errors are often cognitive ones rather than performance errors although such people will make random mistakes at least as often as any other population. Typical errors include phonetic spellings, vowel substitutions, missing or wrong orthographic irregularities (failing to follow exceptions) and, interestingly, real-word errors (writing the wrong word which is nevertheless one listed in the dictionary).

The appropriateness of the various notions of similarity assumed by the algorithms below is of great interest, quite apart from the effectiveness of the algorithm at correcting errors it considers ‘similar’ to the target word. The simplest common notion of similarity is that of minimising the total number of insertions, deletions and substitutions. This is used by most of the Approximate String Matching algorithms. For dyslexic people, similarity must be measured in terms of phonetics,

Group	Letters	Category
1	a e i o u y	Oral resonants
2	b f p v	Labials and labio-dentals
3	c g k q x s z (discard gh, –s., –z.)	gutterals and sibilants
4	d t	dental-mutes
5	l	palatal-fricative
6	m	labio-nasal
7	n	dento or lingua-nasal
8	r	dental-fricative

Table 2.1: The letter groups of the name encoding scheme which developed into Soundex. Each word is encoded by keeping the first letter and then converting the remaining letters to group numbers, removing any repetition.

parts of speech and visual appearance of letters as well as other conventional methods. This is discussed further throughout this thesis.

2.2 Name Spellings: Soundex

One of the earliest methods in the field of spelling correction was intended for surname indexing (Russell 1918). The principle, expounded in the language of patents instead of scientific papers, was to reduce the letters of a surname to members of groups and to list no more than one member of each group. The first letter of a name would be retained exactly as given. The groups are shown in Table 2.1.

The system was intended specifically for surnames, and was designed to solve the problem of speeding up searches for similar names in an index, for example when a secretarial worker had to find the file for a customer who was calling by telephone. For example it converted both ‘Smith’ and ‘Smythe’ to the code S614, so removing

the problem of identifying exactly which name was correct.

The method is reasonably tolerant of spelling errors, pronunciation errors and family name variations, but makes predictably glaring errors in filing very different names together and failing to match names with certain subtle differences which should be together. Also, the more information that a system omits when constructing a key, the more entries there will be under each individual key.

It was updated by Odell and Russell (1922) who removed the vowels altogether, restructured some of the letter groups and enforcing a maximum of 4 letter groups in any one key-code. It is this version that is more widely called *Soundex*. Some other people have devised further developments on the same idea, but Soundex is still widely used today. The American Census department encoded the entire records for 1880, 1900, 1920 and partially for 1910 using Soundex and genealogy enthusiasts (for whom minor changes in spelling are common) continue to do so. Indeed, a commercial product which claims to be the best thing for dyslexic students, 'TextHelp', is based on Soundex.

Soundex is important for a number of reasons. It was the first major algorithm which made an attempt to consider spelling errors in a mechanical sense. It emphasised phonetic errors of the sort that might be made when transcribing speech, and was simple to implement (because it was designed to be executed by people). It has influenced the research work presented here, particularly in its disregard for vowels, but has not been taken as seriously here as it was by Mitton (1996).

Letter	Score	Letter	Score	Letter	Score
A	5	J	0	S	5
B	1	K	1	T	3
C	5	L	5	U	4
D	0	M	1	V	1
E	7	N	3	W	1
F	1	O	4	X	0
G	2	P	3	Y	2
H	5	Q	0	Z	1
I	6	R	4		

Figure 2.1: The logarithm of the desirability of deleting a letter, based on an empirically constructed approximation by Mr Blair.

2.3 Computerised Correction

2.3.1 Human errors

The first widely read paper on spelling *correction* by computers was that of Blair (1960), which was perceptive and forward thinking, although many later works have failed to recognise its contribution. Firstly, it specifically addressed the issue of human spelling correction rather than error detection, mechanical error handling or variant spelling correlation. In fact, as stated in the paper, it would have quite poor performance on misfed paper tapes.

The work begins with a review of the methods that one might try to use in correcting errors. The first is to record a long list of string pairs, where each possible misspelling is linked to the correct spelling of that word. This is in fact used in some circumstances, for example the much-vaunted AutoCorrect feature of some current word-processors. It is extremely limited, however, because it consumes immense quantities of memory and requires observation of all possible errors in advance;

something which is not realistic. Blair then considered a system of orthographic 'rules' which can be used to construct correct spellings for any word, for example the classic "i before e, except after c". These rules have since been found to be incomplete, and Blair predicted this by observing exceptions such as *weight* and *neighbor*. Blair also observed that only 'close' errors would be correctable without reference to the context, and he did not intend to address context-based spelling correction himself.

The *Soundex* system was cited as a way of establishing similarity between two spellings, but that relies on some other measures of correctness for a human operator to compare which a machine would not be in a position to do such as matching an address or customer's first name. Certainly, Soundex puts spellings into groups rather that handling them individually.

Instead, Mr Blair proposed carefully constructed abbreviations of the words in the dictionary and of each word to be tested. It was based on two metrics; the probability that any particular letter would be misused, and the probability that any letter in a particular position would be misused (these two values were multiplied). The letter undesirability was derived from the table in figure 2.1, and the position within the word from another table built up more methodically, assuming that letters near the ends of the word were more important. Thus the first letter was the most important, the last letter next, then the second and the second last, and so on. The middle letter was the most likely to be removed.

For both the dictionary of correct words and the written word being checked, an abbreviation was created by deleting letters in order of their undesirability until only n were left, where $n = 4$ was recommended but left to the judgement of the reader. Where multiple words from a dictionary matched the word being tested, the word was re-tested with a longer abbreviation.

The system was implemented and tested on 117 words listed as typical misspellings in the Standard Handbook for Secretaries (Hutchinson 1956). Of the 117 words, 89 were matched using the abbreviations. The dictionary of correct words contained only the 117 correct spellings; with a more realistic system the abbreviations would have to have been longer, so there would be less flexibility.

As an additional correction for commonly misspelt words, Blair suggests adding a table of alternative abbreviations. This is similar to the huge table of all possible misspellings but would only be used in a relatively small number of cases, and would list only a very small number of abbreviations which the misspellings might reduce to instead of each misspelling in its entirety.

Blair's work was undoubtedly impressive. The fact that many letters are removed from both versions of a word means that insertions and deletions are not considered as such, which may be a good thing. The combination of letter position within the word, and the letter identity, gives the system a higher degree of domain knowledge which may improve its performance, assuming that the construction of the relevant probability tables was done well. Blair also pointed out that the combination of the two values could be done in advance, providing a matrix of probabilities for each letter in each position.

There are, however, a number of limitations to the work. Firstly, the test it applies is boolean; whether for a given value of n, the abbreviations of two words are identical. This is similar to Soundex, and not ideal for a large dictionary where there may be many possible matches.

Secondly, the construction of the probability table seems to have been arbitrary, without a large sample of errors to help in its design.

Thirdly, the system of reducing a word to n letters has a varying loss, depending on the number of letters present initially. If there were 5 letters and one was deleted,

many misspellings may still exist with different abbreviations. On the other hand a 15 letter word reduced to an abbreviation of 4 would allow up to 11 of the missing letters to be in error.

Blair was probably the first researcher to consider spelling correction on computer, and took an approach to individual letters which, although somewhat arbitrary, was much more robust than later workers. In current work on approximate string matching it is customary to assume that each letter has the same probability of being mistaken, and even that each error type is of equal probability. The ideas from Blair's paper have been prominent in the design of the *Babel* software in this thesis particularly in the individual letter rules.

2.3.2 Mechanical Errors

An early and influential paper on spelling correction (Damerau 1964) described a simple method for correcting simple spelling errors in text found either from card punching faults, and is compared to a system intended for spelling errors by people. Naturally the system ran on computers of the day and so efficiency was important, as will be discussed below. The application was to check each word as it was loaded from tape into working memory against a dictionary of correct words; if any was not listed but a near neighbour existed, it would be substituted automatically. The author found that over 80% of the erroneous words (not found in the dictionary) were cases with exactly one insertion, deletion, substitution or transposition. User intervention was not required at any stage in the algorithm, largely because computers of the time were not capable of adequate interaction.

The system first constructed a bit pattern representing the presence of each letter in the word in question; repeats were ignored because the bit for the letter was already set, and letter order was ignored completely. The system also recorded the number

of letters in the word. The same kind of record was constructed in advance for each of the words in the dictionary.

When a word was loaded from the input tape, a bit pattern was first built for it. If there were three letters or fewer the word was ignored (accepted). If there were between 4 and 6 letters, a special dictionary of common words was checked first before moving on to the main dictionary, for speed. When words were checked in the main dictionary, Damerau seemed to use a linear search because he checked the string length first; the letters were compared only if the lengths were the same (this operation is not compatible with a binary search). If a match was found with a dictionary entry, it was accepted.

Assuming that the input word was not in the dictionary entry, it was compared to each of the dictionary entries as follows. First, the word length was compared. If the input and dictionary words varied in length by more than one letter then the system abandoned the comparison and moved on to the next dictionary entry. Next, the 'character registers' (bit patterns) were compared. If they differ in more than two positions, the comparison was abandoned because none of the simple edit operations can account for such errors.

Finally, the positions of the character differences were recorded. If the two strings differed in only one position then a substitution was assumed and the words were considered equal (Damerau found that only 3% of corrections were wrong because of this assumption). If the words differed in two adjacent positions, the letters in one were transposed and compared with the other word; if they matched, a transposition was assumed and the word was corrected. If the input word was one letter longer than the dictionary word, the first differing character was deleted and the remaining letters shifted left in the input word. If the words became equal, an insertion was assumed and the word corrected. Finally, if the dictionary word was one character longer than the input word, the first differing character of that was

deleted, the remainder shifted left and the results compared in the hope of finding a deletion.

The system, when tested on human generated errors, achieved success rates of 74% and 82% on two samples, and a success rate of 84% on a set of 964 mechanical errors (with 3% incorrect changes, and the remaining words with no alternative words identified). By comparison, the system of deletion-probability abbreviations achieved success rates of 76% and 73% on the human errors for which it was designed and 25% on the mechanical errors.

Damerau was the first Computer Scientist to consider the classical error types of insertion, deletion, transposition and substitution. These have been fundamental to the operation of spelling checkers ever since, and have certainly been used in the work here. His work has been widely cited since publication.

2.4 Scientific Texts: SPEEDCOP

A later work (Pollock and Zamora 1984) followed on from the work by Blair (on page 20) although they do not mention his work at all, and Damerau, with a system to correct errors in 'scientific and scholarly text'. The key feature of the kind of text in question is that the words are often long and the writers reasonably competent. By this era, scientific papers were being typed on a regular basis whereas Damerau had been concerned with errors by expert typists who worked with proof-readers.

As with the previous work, SPEEDCOP, the system of Pollock and Zamora, considered only simple single error corrections, but to improve the search for the word within the dictionary of correct words they sorted it in an interesting order which also lends itself to more complex error pattern analysis. The central idea was a *skeleton key*, similar to the abbreviations used by Blair in which the letters were

Word	Skeleton Key	Omission Key
chemogenic	chmgneoi	mghcneoi
chemomagnetic	chmgnteoai	mghcnteoai
chemcal	chmlea	mhclea
chemcial	chmleia	mhcleia
chemical	chmleia	mhcleia
microelectronics	mclntsrioe	mclntsrioe
molecule	mlcoeu	mcloeu
cameral	cmrlae	mclrae
caramel	crmlae	mclrae
maceral	mcrlae	mclrae

Figure 2.2: Examples of SPEEDCOP skeleton and omission keys for misspellings and real words. The intention was to bring incorrect spellings near to their most appropriate corrections by encoding the most common spelling errors.

re-ordered and some removed.

SPEEDCOP built the skeleton keys from the dictionary of known words by taking the first letter of each word, followed by the unique consonants (thus ignoring repeated letters) followed by the unique vowels (see Figure 2.2). The intention was to create a letter string that had the letters most likely to be correct (in a word mistyped by the typical author) at the beginning of the string, and those more likely to be wrong towards the end of the string. The list of keys was then sorted alphabetically. The dictionary of correctly spelt words was also held in a sorted list in their normal form, with pointers from the skeleton keys to the corresponding actual words.

The system also included a simple algorithm for checking that a word contained one simple error transformation. For a given input word and an entry from the dictionary being tested, the lengths of the two strings were first compared. If they differed by more than one, the words were definitely not similar. If they differed by

one or no characters, the position of the first difference was found using a character-by-character comparison. This is called p for the following algorithm (If there were no errors, the position of the first difference was set to the end of the word, and the problem was trivial)

If the dictionary word was longer than the input word, and the dictionary word from position $p + 1$ onwards was the same as the input word from position p onwards, then the words were matched as an omission error. Conversely, if the dictionary word was shorter and matches from position $p + 1$ against the input word from position p then the typist must have inserted a letter. If the two strings were the same length and are both identical from position $p + 1$ onwards then a single substitution must have occurred. If the two strings were the same length and character p from each equals character $p + 1$ from the other, and the strings were identical from position $p + 2$ onwards, then a transposition was reported. If all of the preceding tests failed, then the words were not acceptably similar, and another dictionary word should be tried.

When a word was to be checked, it could first be searched for in the normal dictionary using a binary search. If it was found, no further work need be done.

Assuming that a word was not in the dictionary, a skeleton key was constructed for it as described above, and the list of skeleton keys searched for the nearest entry. The system then searched both up and down from the nearest skeleton key, checking for words which match to within one simple error transformation.

If the system searched more than a set number of entries away from the starting key (initially set to be 50), it might be considered futile to search any further. There could easily be more acceptable matches, for example if the first letter had been deleted, but the system did not search the entire dictionary.

Another stage of key indexing was suggested to further improve performance

of this algorithm. The consonants were re-ordered according to the probability that they might have been deleted (based on general results from the data used in the experiment). Thus consonants, if present, were sorted into the order 'jkqxzvwybfmgpdhclntsr' to produce what was called the 'omission key'. These keys were sorted like the skeleton keys.

If no match was found for an input word by comparing it to skeleton keys, the system would create an omission key for the word, find the nearest entry in the sorted list of omission keys from the dictionary, and search that for a while in the same manner as for the skeleton keys.

The authors found that, for the scientific work (mostly papers in Chemistry) in question, the vast majority (1902 words) of correct answers were found on the first skeleton key searched. A quarter as many were one key away (although this does not mean that the first key was acceptable using the simple edit distance system), and about quarter of that number (134) were one step further.

The two most important parts of the work were the idea of the key design, and the inclusion of the secondary check for the simple error transformation where other published algorithms tended to accept words based on their first impressions. As well as these, the authors included a list of common misspellings that described exact letter sequences that were frequently mistaken and the words intended.

This absolute mapping from errors to corrections is something which is acceptable for small lists of common words but which cannot be done for large dictionaries, not only because of the exponential increase in file size but because of the limit on available spelling error data with which to build such lists correctly. The creation of artificial error data, as done by some experimenters, is not a valid way of building up such a list.

The system, when tested on abstracts from bibliographical databases, achieved

an accuracy of between 56% and 77% of single-error words, where the number of errors which can be described as a single simple error vary from 77% to 96%. In one case studied more carefully, 71% of the single-error words were corrected, 11.5% miscorrected to the wrong word and 17.5% unchanged. It is probably the set of miscorrected ones that should be minimised, especially for an automated algorithm such as this, so as to reduce the need for human proof reading.

If multiple error correction were desired, the Approximate String Matching techniques explained on pages 31 onwards would be ideal.

The work by Pollock and Zamora was intended to correct the errors of a very specific group of writers; competent scientists making simple errors in generally long words. This is significant because it allowed the project to make some quite specific observations about typical errors of that population. It is evaluated in Chapter 7 with the dyslexic error corpus of this thesis.

2.5 A-Star Search

An issue which arises in part in the work of this thesis is that of searching. It might be said that the whole task of spelling correction is one of search, although such a claim would be naïve. Edit cost algorithms generally attempt to search a dictionary quickly for candidate entries in a sophisticated way, often in two passes; the first to exclude entries which cannot achieve the criterion and the second to identify the best match.

The ‘A$\star$’ algorithm (A-Star) is a general purpose search algorithm which uses unspecified operators to generate new permutations of solutions from old ones. This is similar to the search used in the ‘Babel’ system of this thesis in some respects. A solution is developed gradually as a ‘path’ by the repeated application of the

operators to the permutations.

Lirov (1991) amongst others offers an extension to A$\star$ which supports multi-objective searches which are not of particular interest here, but usefully defines an A$\star$ algorithm as follows:

For a queue of partial paths: The initial queue consists of a 0-length path. Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal. If the goal is reached then do nothing. Otherwise...

- Remove the first path from the queue.
- Form new paths from the removed path by extending it by one step using any of the operators available. (This would typically employ all allowable operators to generate a large number of new paths.)
- Add the newly formed paths to the queue.
- Sort the queue by the (cost accumulated + estimate), with least cost paths in front.
- If two or more paths reach a common node, leave only the one with the minimum cost. (For spelling correction, a node would be a certain spelling permutation of a word.)

If the goal is found then exit with success, otherwise exit with failure.

The A$\star$ algorithm is also described as an ordered search procedure which is heuristic and incorporates dynamic programming. Dynamic programming algorithms are defined as those which search for solutions to problems and discard higher cost paths if several paths lead to the same node. The original string edit cost algorithm (Wagner and Fischer 1974) uses dynamic programming to find the lowest cost path to process symbols from the beginning to a certain position in a string. Heuristics are those strategies which use easily accessible but only loosely applicable information to efficiently control search processes.

The user modelling component of the Babel software in this thesis, however, depends on knowing which operators were used in traversing the paths and

sometimes on the different paths which can be used to reach the same node. For this reason, the method used below is not a dynamic programming one. Instead the *Path Choice Heuristic* is used to select from a number of possible paths after solutions have been found.

The perspective on the problem is in some respects reversed; finding the solution is easy for Babel in some cases (for example when both the written and intended words are known), but deriving the decision heuristics can be difficult and is more important. By contrast traditional search tasks have a well defined set of decisions heuristics and need only to establish the solution which maximises their criteria.

2.6 Approximate String Matching

For the purpose of a user-model based spelling checker, it is important to know which operations were used in a correction (for future refinement) and what cost each operation carries (for ordering of suggestions). It is also important to be able to perform multiple transformations on a single string.

If the single-error method in Pollock and Zamora (1984) were converted to recursive operation, it could handle any number of errors in the transformation of a string. However, it would require a run time of $O(4^n)$ for strings with n letters, and would also need some other changes to avoid infinite recursion.

Alternatively, there is the more sophisticated string-to-string correction method, described as ALGORITHM X in Wagner and Fischer (1974) which finds the lowest edit-cost combination of insertions, deletions and substitutions required to convert one string to another, where each operation on each letter can have a different cost.

In the pseudo-code listing of ALGORITHM X below, the input strings are stored in A and B with individual characters in $A\langle i\rangle$ etc. The length of string A is $|A|$.

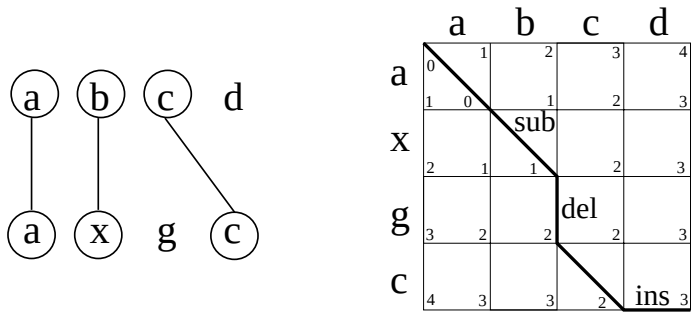

Figure 2.3: An example of editing operations required to transform one string into another. The table on the right shows the edit cost of converting the first part of one string (from the beginning to a given position) into the first part of the other. The thick line shows the lowest cost path, and the operations required to follow it.

The function $\gamma(c_1 \rightarrow c_2)$ returns the cost of converting character c_1 into c_2, and Λ is the null character. The function min(a,b,c) returns the lowest of its parameters. Later examples of approximate string matching algorithms require a differentiation between the pattern string P and the main text T against which it is matched. For ALGORITHM X, however, the two strings being compared are interchangeably called A and B.

$D[0,0] := 0;$
for $i := 1$ **to** $|A|$ **do** $D[i,0] := D[i-1,0 + \gamma(A\langle i\rangle \rightarrow \Lambda);$
for $j := 1$ **to** $|B|$ **do** $D[0,j] := D[0,j-1] + \gamma(\Lambda \rightarrow B\langle j\rangle);$
for $i := 1$ **to** $|A|$
$\{$ **for** $j := 1$ to $|B|$
$\{$ $m_1 := D[i-1,j-1] + \gamma(A\langle i\rangle \rightarrow B\langle j\rangle);$
$m_2 := D[i-1,j] + \gamma(A\langle i\rangle \rightarrow \Lambda);$
$m_3 := D[i,j-1] + \gamma(\Lambda \rightarrow B\langle j\rangle);$
$D[i,j] :=$min$(m_1, m_2, m_3);$
$\}$;
$\}$;

In considering how ALGORITHM X works, it is important to realise that any string can be transformed into any other through the use of the operations insert and

delete. It builds a directed acyclic graph with nodes representing the quantity of each of the two strings that can have been processed, and edges indicating the costs of the edit operations allowable from each node to others. Computing the edit cost becomes a case of calculating the lowest cost path traversing the graph from position $0, 0$ to position i, j.

Figure 2.3 shows two strings and the editing operations required to convert one to the other. On the left are lines linking symbols which are kept or substituted. Those symbols not touching a line are not related to the other string; one could say that d was deleted and g was inserted, if the changes were from the upper to lower strings.

The algorithm operates by passing through each possible letter pair in both strings using two nested loops. Each loop variable represented an index into one of the strings, where the substrings to the left of these indices have already been transformed. A two-dimensional array was built up where the value indexed by the loop variables recorded the lowest cost to transform letters to the left of the loop indices (the ubiquitous *insert* and *delete* operations allowed any sub-optimal solution to be tied up neatly in the end, although a lowest-cost edit sequence would not normally involve deleting the whole of one string and inserting the whole of the other).

At each stage of the array construction, one letter was added to the transformation sequence with the cost increasing by which ever was the cheapest edit operation that described it (either an insertion, deletion or substitution, where the null operation was a special case of a substitution). Thus each array element essentially contained the next operation to apply assuming that the two substrings referred to by its location in the array were on the lowest cost path. If this assumption turned out to be false, and those particular substrings were not used, the array element would have no effect on the final outcome and would not be used afterwards.

By the time all possible substrings which began at the beginning of the strings had been considered (*ie* both loops had completed), the last element of the two-dimensional array would contain the lowest edit cost. To find out which operations were necessary to achieve that cost one could replay a limited version of the previous loops in reverse order (assuming that the array of costs was available for inspection), working out at each stage which operation had been applied, and hence which element of the array to move to next.

The right hand portion of Figure 2.3 shows the array D built by ALGORITHM X as it processes the strings. The top left corner represents the edit cost after consuming none of one string and generating none of the other. The bottom right corner is the final edit cost of consuming all of one and generating all of the other. The intermediate vertices represent intermediate edit costs of processing part of each string. The thick line represents the lowest cost path between the corners, and by implication the editing operations required to complete the task. The example given can be edited using a substitution, a deletion and an insertion leading to an edit cost of 3.

ALGORITHM X relied on the fact that the strings could be cut at any point, with any increase in length of the strings definitely not resulting in a decrease in cost over the lowest cost edit sequence for the substrings from the start to the cuts.

Later, Lowrance and Wagner (1975) proposed ALGORITHM S which allowed transpositions[1]. This was more difficult because it could not be assumed that adding a character to an arbitrarily cut pair of substrings would always result in an equal or higher cost; a transposition may become possible which would have a lower cost than the insertion or deletion previously required without the new character.

[1]In fact there was a misspelling in the original publication of AGLORITHM X which was perhaps overlooked because it could not correct transpositions.

For convenience of notation, the editing costs were reduced to W_C for a change (substitution), W_S for a transposition, W_I for an insertion and W_D for a deletion. The operation of the algorithm is not actually more restricted than the earlier one in this respect.

However, their second algorithm was reported to work correctly where $2W_S \geq W_I + W_D$. That is, the cost of a transposition (W_S) is at least the average of the costs of an insertion and a deletion. If the starting and ending strings are written one above another as in Figure 2.3 and lines drawn to connect letters which are carried forward from A to B (including substitutions), then W_S is actually the cost added when two such lines cross. ALGORITHM X, presented above, does not work where one line crosses more than one other, and will instead delete one character and insert it again in another place.

ALGORITHM S stored, in two simple arrays, the index into each of the two strings the position of the last occurrence of each character. Each array held as many entries as there were characters in the alphabet. Using this and the assumption that no more than two editing lines should cross, Lowrance and Wagner were quickly able to establish whether a transposition of a current character with a previously observed one would be preferable.

The difficulty, of course, is in coming up with a method which always produces the optimal answer while consuming a realistic run-time. Both ALGORITHM X and ALGORITHM S used $O(n^2)$ time assuming each letter string had n characters. An exhaustive search of the possibilities would require a time of the order of $O(4^n)$.

Naturally, the approximate string matching algorithms require costs for each of the operations. Although these might be adjusted, they cannot be modified post-hoc for an edit sequence already chosen. Also, a comprehensive spelling correction application would require an edit cost for each of the words in the dictionary

against the word written; this would take a considerable time, although in some cases such as Pain (1985) it has been found to be preferable to other economies.

The works of Wagner, Fisher and Lowrance are perhaps the most important in a traditional 'Computer Science' sense to spelling correction. They have been developed more by a number of other scientists, and Chapters 8 and 9 of this thesis takes the issue even further. Babel, presented in Chapter 5, did not use conventional edit cost methods alone because some spelling errors cannot easily be assigned to particular positions in strings, for example the general concept of a phonetic error or the substitution of graphemes.

More recent literature on approximate string matching is discussed at the beginning of Chapter 8.

2.7 Approximate String Matching for Biology

Biological research has recently derived large samples of sequence information from DNA and protein samples. These are long strings (thousands of symbols) from an alphabet of 4 DNA bases or 20 proteins. With large databases already recorded, the new samples found are compared to establish where they come in the existing body of knowledge, if anywhere. Samples are normally incomplete, being substrings which begin and end at random points in the known database. They also often contain inaccuracies. Errors may occur in the sampling machinery, or more often in genetic differences in the organisms from which the new and canonical samples were taken which lead to some mismatching sequences or individual symbols.

The computational algorithms for biological sequence matching are very similar to the approximate string matching methods used in spelling correction, although

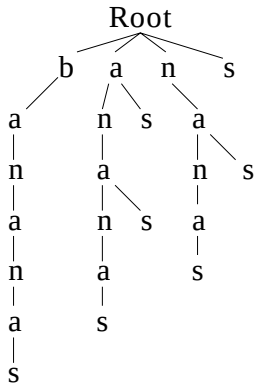

Figure 2.4: A simple suffix tree for the word ‘bananas’. Any sub-string of the whole can be found quickly by searching from the root node.

---

the scale of the problem is larger. A number of researchers have studied the field of approximate string matching recently and have come up with a number of improved methods.

Foremost amongst the recent work is the Suffix Tree which was first proposed by Weiner (1973). An example is shown in Figure 2.4, in which the word ‘bananas’ has been stored as a linked list of characters and is shown on the left of Figure 2.4. The first ‘suffix’ of the word consisting of everything except the first character has then also been stored, as indeed have all suffices (substrings beginning at any point in the text and ending at the end). Where suffices begin with the same character(s), they share nodes in the tree until their divergence.

Such a structure is not small, but has the advantage that it can establish, in linear time, whether any substring of the stored string is valid. Approximate matching using dynamic programming can also be performed, achieving a highly efficient search.

A further development of the suffix tree is the suffix automaton, a finite state machine based upon the pattern string P and which takes the main text T as input.

It accepts the input when an approximation to the pattern has been found. Jokinen, Tarhio, and Ukkonen (1996) present one example of a suffix automaton system which was later found to run more slowly than other methods.

Many recent approximate string matching methods are designed to match a short pattern string P against a longer text T. They often record substrings of T so that they can avoid recalculation if the same substring reappears later. This is often done using a suffix tree, and thus is similar to the work in Chapter 8.

The work most similar to the new algorithm in Chapter 8 of this thesis is that of Shang and Merrettal (1996) which uses a short suffix tree, illustrated with a path from the root node to a leaf node if a dictionary word exists with the spelling defined by that path. This is discussed, along with other relevant string matching methods, in Chapter 8.

2.7.1 Efficient Dictionary Short-Listing

The work of Wagner and Fischer (1974) and Lowrance and Wagner (1975) has been very influential in spelling correction work, although people continue to try to devise improved algorithms. Du and Chang (1992) have suggested a method by which unnecessary computation can be avoided.

Beginning with the edit cost algorithm, Du and Chang (1992) applied a test after each iteration layer to establish whether the two strings being compared could possibly achieve a close match. The system iterates through a dictionary of words to compare several times, relaxing the required closeness of match each time. A written word is then compared with every entry from the dictionary but the cost matrix (the array D in ALGORITHM X shown on page 32) may be not be fully calculated. This concept of heuristically driven abandonment of the standard dynamic programming edit-cost method is common in modern algorithms.

A *radius* value is specified, being the required closeness of match for the current pass through the dictionary. If the last calculated cost matrix entry diagonally in line with the final corner is more than this radius value, the calculation is aborted on the grounds that an adequate match has already been found impossible with this string pair.

The algorithm as published (which has been left out, since it would not clarify the issue) does not retain any information about previous cost matrices for string comparisons which have been abandoned. The authors point out, however, that storing the information required would consume only $2 \times len_{max} \times n_{dict}$ (because only values from the previous radius layer need be recalled for each subsequent layer to be computed). Strangely, this second algorithm was not implemented.

For experimentation, words were used from databases of scientific citations, similar to those used by Pollock and Zamora (1984) which contain mostly technical terms and proper names. The average word length was just over 8 letters. The spelling errors tested on the system were mechanically constructed permutations of the known words, with pre-set numbers of changes. This is unrealistic.

The system was compared to the Lowrance and Wagner algorithm which fully explored the edit cost of the test word against each dictionary entry. The latter performed about 2^{20} calculations of cost matrix entries for average-length words whereas the new system calculated only around 2^{15} matrix entries. Similar tests were run with $r \in \{2, 3, 4\}$. The new algorithm's run time increased to approximately $2^{18.5}$ whereas the old one of course remained at a constant regardless of the number of errors in the sample word. The constants of proportionality in the run time are presumably similar; the authors assumed that the run time was proportional to the number of matrix entry calculations in both cases.

This method highlights the fact that heuristics can be used to improve search

performance, but the change is minor and unremarkable. The run-time savings are useful although the performance of the system is not shown in the more realistic circumstances where the number of errors in a word is not known, and so the entire dictionary must be searched repeatedly. There is also no new commentary on the likelihood that the principle of calculating the nearest neighbour in the dictionary is actually similar to the distribution of errors made by real writers.

In a follow-up paper, Du and Chang (1994) proposed a further improved algorithm that could match a word with the nearest entries from a dictionary of words. It worked by pre-computing lists of dictionary entries in which an appropriate number of wildcards had been used; each wildcard could represent an insertion or substitution, and deletions or transpositions did not require wildcards.

The new algorithm then took a word from the user, derived all the possible wildcarded strings using the four famous editing operations, and used a hashing function to short-list all valid words from the pre-computed tables, and executed the edit-cost algorithm on each (actually the slightly improved edit-cost algorithm presented in the previous paper). The important feature was that a selection of pre-computed tables would contain all possible candidates for the nearest neighbour to the given word from the dictionary, so long as it was within the pre-specified edit cost r, and yet would be only a small subset of the total dictionary size.

In fact, the pseudo-code algorithms presented by Du and Chang assign an identical unit cost (of 1) to each edit operation whereas Wagner and Fischer (1974) gave a different cost to each operation on each letter (or letter pair). Changing the costs in Du and Chang's algorithm would utterly destroy the benefits of their dictionary short-listing, whereas the run time of ALGORITHM X would remain unchanged with varying costs.

The final algorithm presented in Du and Chang (1994) is faster, at the expense of

using more memory. However, it suffers from two constraints which were merely assumptions in previous work. The first is that each edit operation must have the same cost, $W_I = W_D = W_C = W_S = 1$. The second is that the maximum acceptable edit cost, r, must be specified in advance when the tables of words in the 'neighbourhood' of a given wildcarded string is being computed; although the lowest cost of the neighbours will be found, any word outside the neighbourhood will not be considered.

The improvements suggested are worthwhile, however, as they do reduce the run-time of such a dictionary search algorithm by about an order of magnitude. In the same spirit, another is proposed here: when measuring the edit cost between two strings, the search should begin at the end of the strings, because most such comparisons will be done on alphabetically sorted words for which the similarity near the beginning is already established, and because most human spelling errors tend towards the end of the words.

2.8 Phonetic Error detection

The majority of work in the field of spelling correction has been done on English language texts, presumably because that is the language in which most scientific research is conducted, and perhaps partly because most studies in other languages would be published only in local journals which are not easily available to global literature searches. A well written exception is Veronis (1988) which gives an overview of work on phonetic spelling errors in French, with a view to foreign language learning with computers.

The work is based on the concern that most spelling checkers consider only letter-based errors, and many errors made are phonetic. A typical error which the project set out to correct was the word 'hypoténuse' being written as 'ippeauttainnuze'.

The knowledge of the phonetic structure of languages with regard to this kind of error is limited, so Veronis began by defining a term: *graphoneme* which describes a tuple of letters with a unique pronunciation which cannot be subdivided. The sound[2] /o/ can be written in French a number of ways including 'o', 'eau', 'au', and each possible pair of letters with a sound would be one graphoneme.

A lexicon of 3724 words was collected and analysed. It was found to contain 22317 graphonemes in total, or 141 unique graphonemes. The single graphoneme (*r*, /R/) (that is, the letter *r* being pronounced /R/) occurred 2153 times, or as nearly 10% of the total number of graphonemes. The author found that having just 11 graphonemes allowed one to write 50% of the lexicon, 42 allowed one to write 90% and 90 allowed one to write 99%. The remaining 51 graphonemes were used in less than 1% of the total graphonemes of the lexicon.

After considering various methods including Soundex and the string-to-string correction problem, Veronis proposed an adaption of the string-to-string method working on graphonemes. It was considered important not to rely heavily on the correct pronunciation of a misspelt word precisely because it was misspelt. However, component parts of it could be used to reconstruct the sound of a word which might then be compared to a dictionary.

The string-to-string algorithm was slightly adapted by the use of two costs for substitution; the substitution of a similar sounding graphoneme and that of a different one. For simplicity the edit cost of a similar substitution was deemed to be 0. The similarity of graphonemes is recorded in a table of binary values which is not symmetrical; although certain letters are used with certain sounds in correctly spelt words, the same letter sequences should not be assumed in mistakes, for example

[2]The notation /s/ for a given phoneme 's' is a tradition from phonology. There are clearly defined phonetic sounds which can be written, described for example in Catford (1988) but for its limited use here, the basic sound of the letter should be pronounced.

the letter *g* being pronounced like a *j* only when it precedes *e, i, y*. This method, although elegant, requires an $O(n^2)$ subroutine to be tried with each dictionary entry which takes a long time.

As an alternative and faster method, the single error approach used by Pollock and Zamora was chosen. As stated above, it begins by finding the length of the initial substrings of two words which match, then allows one error and then tries to match the remaining letters. It was adapted to encode the words phonetically by finding the longest graphonemes in the dictionary words; the dictionary was then stored in this phonetic form. When a word was presented for correction it would be converted to graphonemes in the same way and compared. This allowed for any number of 'similar' graphoneme substitutions because the actual letters were no longer being compared. It then allowed for one insertion, deletion, substitution or transposition outside the bounds of 'similarity'.

The algorithm was implemented using a small dictionary of 500 words, and found to run in around 50ms on a computer of the time, but the results of its accuracy have not been presented in the paper. The method is an interesting one nevertheless because it suggests a way of handling phonetic errors which are otherwise barely considered in the spelling correction literature. The phonetic component of Babel was a relatively small part and employed a standard module from a speech synthesiser.

A substantial paper on spelling correction was that of Yannakoudakis and Fawthrop (1983a) which, along with another paper in the same volume (Yannakoudakis and Fawthrop 1983b) presents a heuristic system for correcting actual spelling errors. It is based on a sample from 1534 errors, mostly from adult typing work. The system has a number of rules with probabilities which combine with word frequency probabilities to generate a list of suggestions.

Their initial assumption is that all errors begin with the correct first letter, and differ in length by no more than two letters. They also consider it very unlikely that two error rules need to be applied in a single word, and impossible for more.

The words were divided into pseudo-graphemes according to particular rules which were then used with the rules. They recognised 299 graphemes and had a table mapping the probability of transforming each into each other with a number of variant rules.

In tests on the same easy text as was used for training, the program described by Yannakoudakis and Fawthrop (1983a) apparently achieved a success rate between 82% and 94% although the authors excluded inconvenient failures such as occasions when the first part of the algorithm selected the 'wrong part' of the dictionary, probably making the true performance between 61% and 70% depending on source population.

The method of weighting rules is a useful one which has been taken further in the work here. The failures of the work by Yannakoudakis and Fawthrop (1983a), such as there were any, were mostly due to a strict interpretation of the range of errors allowed in pre-processing, although it is not entirely clear because the error samples being used seemed to be rather tame (they were mostly from adults or from children aged around 17).

2.8.1 Phonetic Coding for poor spellers

Many previous researchers have considered spelling errors in the context of the four classic error types; insertion, deletion, substitution and transposition of individual letters, but this is not ideal for dyslexic writers or other poor spellers, who are believed to have some sort of systematic errors.

Pain (1985) studied the errors of children with spelling difficulties and developed two computer programs which correct their errors with a very high degree of success. The study was based on work by 15 children (mostly boys) aged approximately 11. They all attended the local council Reading Unit in Edinburgh, having been diagnosed as dyslexic. The children were asked to write a number of stories using a word processor which included an interactive spelling checker in two studies. The research work was interested not only in the spelling correction system but in the idea of interactive spelling correction inside a word processor.

In the first study, the word processor was introduced and the children were invited to write some stories. In the second study, a different group of children were invited to use the word processor and to check (and correct) words which they thought they had misspelt, using the `editcost` spelling checker described below.

The first of the two programs written by Pain (1985) was called `editcost`. It takes a word from the user and checks it in a dictionary of about 1000 carefully chosen words, suggesting the 4 words with the lowest edit cost (The algorithm used, from Backhouse (1979), is very similar to that of Wagner and Fischer (1974) with simple transpositions included, although Backhouse was more interested in parsing program source code with its more varied structure and complication than simple spellings). The costs of the edit operations were hand-crafted to include common letter errors observed amongst the children, being both orthographic and partially phonetic errors.

Certain letter combinations were also converted to special codes, as shown in Figure 2.5, so that they could be treated as single operations. The weights shown are on a scale from 2.5 to 10.0, and are transformed into edit costs using the function $c = 0.05 + (2.5/w)$ so a higher weight in Figure 2.5 relates to a more likely error type. The terms ‘insertion’ and ‘deletion’ refer to the operation required to correct the error, whereas in the Babel description the same terms refer to the error made

Letter(+next)	Code	Insert	Delete	Change	Transpose
final e	A	10.0	6.0	default = 3.0	default = 7.0
initial k(n)	B	8.0	3.5	default	default
w(r or h)	C	8.0	2.5	default	default
h as second letter	D	8.0	4.0	default	default
t(t)	E	7.0	3.5	default	default
r(r)	F	9.0	4.0	default	default
o(o)	G	7.0	5.0	default	default
e(e)	H	10.0	5.0	default	default
p(p)	I	6.0	3.5	default	default
c(k)	S	7.0	3.0	default	default
(c)k	T	6.0	3.0	default	default
g(h)	U	4.0	4.0	default	default
r	r	9.0	4.0	default	default
s	s	5.0	3.5	s→c 6.0	se→es 9.0
k	k	4.0	2.5	k→ c 4.0	default
j	j	4.0	2.5	default	default
a	a	9.0	6.0	a→ e 7.0	ae→ ea 9.0
				a→ o 7.0	au→ ua 8.0

Figure 2.5: A few letter sequences and their weightings for `editcost` developed by Helen Pain. There were also special weightings for other characters; this selection is shown as an example only.

by the writer.

The second program, `phoncode`, worked on a different principle. It took a word as written and converted it to all possible phonetic forms, then looked them up in a phonetic dictionary. It worked on the basis of graphemes, and parsed the input word, looking for the longest available substrings from the table of graphemes it recognised. The program constructed a directed acyclic graph with one node per letter in the original word, initially.

On finding a grapheme (a sequence of one or more letters which can be spoken as a single phoneme) it added an edge to the graph connecting the preceding and

following letter nodes. There could easily be more than one edge per grapheme. After such a match the letters of the grapheme were either marked as being fully explored or not, according to an internal table. If not, further matches would be considered on smaller substrings within it.

The edges between nodes did not necessarily cover exactly one letter each, although there were no cases where one grapheme (one edge) had more than one phoneme. For final 'e' vowels, a special syntax was used which changed the preceding vowel. The edge containing that vowel used a special marker which indicated that the next edge (a consonant) should be rendered, and then the word ended (without the final e being pronounced).

The grapheme to phoneme mappings were devised with the help of a linguist, and offered all reasonable pronunciations of each grapheme regardless of the surrounding context (which could not be assumed to be correct because the application related to spelling errors). Eventually, a graph would exist with the correct pronunciation for the word intended, assuming that it contained only phonetic errors.

Using the phonetic graph, the phonetic dictionary was searched. This was stored in a tree form which enhanced search performance; at each stage where several possible phonemes could be used, a recursive algorithm explored the relevant child nodes from the current position in the tree. The actual words were stored at the appropriate final phonetic node in the tree. This allows for fast wildcard searching, and is discussed more in chapter 5.

`editcost` was used interactively, and the four most likely suggestions from a dictionary of about 1000 words (including a 750 base dictionary and a few hundred more from topic dictionaries chosen for the session). It was found to present the correct word on the list 80.6% of the time, where many of the failures were because

the correct word was not in the dictionary. If the dictionary omissions are ignored, the system succeeded 91.1% of the time for one of the subject groups used. When suggested, the word was at the top of the list 81.7% of the time, and in positions 2, 3 or 4 on 18.3% of occasions.

`phoncode` was not interactive. It was tested after the two main studies had been conducted, using the data gathered at the time. It is not clear how long the suggestion lists were, and there was no prioritisation of the lists (nor were any non-phonetic errors tolerated, although many of those considered to be phonetic and thus corrected would not be by a human). The correct word was included on the list 61.3% of the time for the first study (235 errors amongst 7 children) and 55.9% of the time for the second study (600 errors by 8 children).

The work by Pain (1985) is interesting for a number of reasons. It is the first seriously to consider a mechanical aid for the spelling errors of dyslexic children, and in this respect is the only work to properly precede this thesis. It accommodated a wide range of phonetic mistakes directly using the `phoncode` program, and was possibly the first spelling checker to do so. It also accommodated a number of common phonetic errors in the mainly orthographic analysis of `editcost`. The weights in `editcost` were carefully set to reflect the expected error frequency in writing amongst the users.

The project worked with a small number of users (15), a small number of errors (around 800) and a small dictionary (700 basic words plus 50–300 topic-related words), as would be dictated by the computing resources of the time and the difficulty of obtaining text from suitable people. However, the results are impressive. Although the work was done around 1978–1985, the software appeared to be rather old fashioned, for example requiring the children to retype both their errors and the chosen correction.

Helen Pain's work is the most relevant to Babel because it deals directly with the spelling errors of dyslexic children and the computerised correction of their errors. However, it does not overlap the new work because it was not concerned with user modelling. It suggested the possibility of building a 'profile' of a child in the conclusions, which is one of the ideas to have been followed up. The difficulty of obtaining adequate sample text was the similar to that experienced in this work.

2.9 N-gram methods

In a recent work, Zobel and Dart (1995) present a comparison of approximate string matching methods, and propose a new one. Their interest is in large database text searches where search terms or personal names are mistyped or misheard, and for which a list of corrections should be displayed containing all of the plausible corrections.

The tests were performed on a simplified database of surnames, a dictionary of 113,212 words from the `ispell` spelling checker and a lexicon of all the words used in the TREC corpus of newspaper and journal articles, and regulations and patents, all from the USA.

Effectiveness of a method was a rather mysterious combination of two measures; the *precision* with which suggestions would be considered relevant by a human, and the *recall* of all relevant words from the lexicon. The surnames were used in their original form, in a search for themselves (so precision would be the main contributor to the effectiveness) and in the dictionary the top 200 suggestions were examined by hand and assumed to contain all the words which could reasonably be recalled.

In the dock were a number of algorithms. Firstly and inevitably was the edit cost

measure (here called *edit*) from Wagner and Fischer (1974). Next came a slightly modified edit cost method which allowed simple transpositions.

N-grams were used to measure similarity in a completely different way, with n being varied experimentally. The first measure was a simple count of the number of matching n-grams, specified as

$$\textit{gram-count}(s,t) = |G_s \cap G_t|$$

where s, t are the source and target strings to be compared and G_s is the set of n-grams in s. This has the disadvantage, however, that words which are substrings of others will match the longer word as well as they do themselves. To counter this, Ukkonen (1992) later proposed another n-gram distance measure as:

$$\textit{gram-dist}(s,t) = \sum_{g \in G_s \cup G_t}$$

In the above expression, all valid values of g are added. However, if one assumes that any n-gram appears only either 0 or 1 times in a word (and this is true for 98% of bigrams in English, and almost all trigrams), the above equation simplifies to

$$\textit{gram-dist'}(s,t) = |G_s| + |G_t| - 2|G_s \cap G_t|$$

where $|G_s|$ and $|G_t|$ are trivial functions of the string lengths and $2|G_s \cap G_t|$ can be computed from a precompiled inverted index.

Zobel and Dart (1995) compare the above methods to two that they claim are phonetic. The first is Soundex (which is only an early approximation to being phonetic), and the second a development of that called *Phonix* (Gadd 1990). Phonix performs about 160 transformations on multiple letters, for example mapping all of `gn, ghn, gne` $\rightarrow$ `n` and `^tjV` (that is, a string beginning tj followed by any vowel) into `chV`. A second stage then groups letters in much the same way to Soundex. The first part on its own is also tested under the name 'partial Phonix'.

PRECISION %		
Method	**Names**	**Dictionary**
edit	63.7	39.9
modified edit	61.2	45.5
gram-dist	61.5	45.5
gram-count	55.9	21.7
Soundex	27.4	4.5
Phonix	25.0	3.3
Mod.Soundex+edit	32.9	9.1
Part.Phonix+edit	60.7	36.1

Table 2.2: The accuracy of various algorithms tested by Zobel and Dart in correcting spelling errors in names or general words.

The results of tests on each algorithm with names and other words are shown in Table 2.2.

The authors then go on to consider n-grams as a coarse filtering function before another method such as the edit-cost algorithm. They used *gram-count* to find the 50 words with most similar n-grams (where n was best set at 2), and achieved an effectiveness of between 47% and 55% for the surname index, and between 16% and 22% for the English dictionary. They found that for a dictionary of 10,000 words a Sun Sparc 10 computer took nearly a second to measure *gram-count* for all entries.

Perhaps the most interesting idea, but not a successful one, was to permute a dictionary by rotating the letters and creating extra entries. For the word `wine`, for example, the entries `wine|, ine|w, ne|wi, e|win, |wine` were added to the lexicon. When given a word to match, it was rotated similarly and the near neighbours for each of the rotations were considered further for more careful examination. The idea was that, wherever the error in a word was, it would be moved in one of the rotations to the end of the string where it would be least

significant. Unfortunately, this method was only 44% effective on the name list and 8.1% effective for the English dictionary.

The main result that the authors present is that phonetic encoding of spelling errors is a poor method for correcting them, but this is based solely on the Soundex and Phonix methods. Although Phonix is a very limited method, the work of Pain (1985) and that in this thesis also found phonetic errors to be difficult to correct properly. The rotation of entries is an interesting one but has been demonstrated to be poor in use. The measure of effectiveness used is rather poorly explained, so it may also be that the results are not quite what they seem.

Owolabi (1996) also considered the use of n-grams in a spelling correction context as well as a number of other methods. The dictionary was partitioned according to the n-grams in each word in advance. For a given query word, the n-grams were used to create a short-list by finding all relevant dictionary partitions. The short-list was then searched with a more comprehensive edit-cost method. With a simple n-gram method the partitioned dictionary would occupy 45Mbytes, but several could be overlapped into the same partition. If one partition is used for 500 n-grams, the run time and size became more acceptable, at the cost of searching a larger number of actual words in the dictionary. In the end 3.5% of the whole dictionary was typically searched with the n-gram method, and the final accuracy depended on the secondary method.

Owolabi also tried indexing words on word halves on the principle that it was likely that either the first or second half of each word would be correct, (definitely so if only one error is expected) and found that only 0.6% of the dictionary needed to be searched.

The n-gram work is interesting as an alternative method, and is of moderate efficiency, but does not compare well with the best algorithms otherwise available.

Babel used n-grams as an alternative measure of similarity in its training search mode because they are so different from the metrics normally used in Babel.

2.10 The combination of methods

Various methods have been suggested which purport to find an optimal spelling correction for a misspelt word, and each has its strengths and weaknesses. Mitton (1996) wrote a substantial program which incorporated several features from other research as well as a number of original ones to produce a system which works surprisingly well on many types of text. He also tested his system most thoroughly on real spelling errors from a substantial number of sources.

At the core of the work is the trusty edit-cost algorithm. It has been modified like Helen Pain's to give certain letters different weights in their edit operations, and also to have different weights in different words.

Instead of attempting to derive global rules for spelling patterns and errors, Mitton works on the principle that "The main reason why people make misspellings in English is that English spelling is quirky; a corrector has to know about the quirks" (page 122).

Each word in the dictionary contains a number of flags indicating which letters might be inserted or deleted; these flags were derived earlier by experimentation and observation of actual errors. These include information like the increased probability of a person adding the letter *t* to the word *pouch*, or of removing the letter *c* from *scissors*. Some of these expected errors were derived from a phonetic encoding of the words. Although the software does not encode words phonetically at run time, the words in the dictionary were recorded with their correct pronunciations, and then converted back to the naïve spelling. The

differences between the real and naïve spellings were recorded in terms of letters likely to be missed or inserted.

Mitton (1996) also used a derivative of the Soundex system to restrict his edit-cost search to a smaller set of words from the dictionary. It is important, of course, to include the correct word in such a restricted set and yet the larger the set is, the slower the system will run. The Soundex system was modified, firstly by using the more modern version which removes all vowels and clips the code at one letter and 3 digits. Secondly, the consonant letter *r* was ignored along with vowels and *y,h,w*.

To overcome the difficulty in the importance of the first letter, the system equates a number of them such as *g* and *j*, which may be used wrongly by a writer. It also considers the naïve spelling based on the pronunciation of the word and uses that to generate another Soundex code. If any homophones are known (as stored in a table prepared earlier), the Soundex codes for them are also thrown in. One final set of Soundex codes to be added is that from another stored table of common misspellings of function words. The result is a longer list of Soundex codes which is more likely to contain the correct solution. Naturally the dictionary entries matching the various codes are retrieved, but before being checked in the edit-cost routine their lengths are measured.

Mitton found that the lengths of most misspellings were within ± 2 of the length of the target word. In fact, spelling tests led to a wider distribution (14% of long words were 2 letters shorter and 11% were shorter than that), but in free writing people tended not to use words they could not spell and so the variation in errors was smaller. Mitton decided to write his program to correct the errors in free writing, and so set the limit at ± 2.

After generating a list of suggestions, it is modified by a number of post-processing routines before being presented to the user. The first is to consider the part-of-

speech (POS) word types. Using a corpus of POS-tagged text, a table of POS bigrams (and also trigrams, although that was less productive) was built. If the preceding word type was never observed before the type of a suggestion on the list, that suggestion is moved down the list. This was applied to the actual word written as well, and so might stimulate the machine to suggest that a real-word error has been made even when the spelling was found in the dictionary.

Words written with capital letters but which are not at the beginning of sentences are considered intentional, and are compared initially with words known always to have capital letters, although a failure in that dictionary will fall through to the normal one. Words with hyphens are accepted only if the whole word is in the dictionary, or if both halves are recorded as real words. It does not contain a list of all hyphenated words because so many can be created, such as *radio-controlled, capital-intensive*, and so on.

Suggestions are also moved by their word frequencies. If a suggestion is rarely used, it is moved down the list, and *vice versa*. Also, if a suggestion has been observed within the last 100 or so words of actual text, it is raised up the list as it may be contextually correct.

Another substantial part of the system is one that counts syllables in words. Mitton found that most misspellings contain the same number of syllables as the target word. To this end he counts the number of both the given word and each of the suggestions, and reorders them appropriately.

The work by Mitton is good. It gave the correct answer at the top of the list more often than the commercial packages it was compared to (40% for young people's writing compared to between 11% and 26%), and on the list more often (it was on the list 94% of the time for the same group, compared to between 77% and 87%). For office documents it also beat the competition in all respects except the number

of false alarms, which was for only one extra word.

The system is characterised by its ad-hoc nature, as stated earlier. It contains a wide variety of techniques which are implemented one after another, each modifying the weights of suggestions or filtering lists. None on its own achieves any great success, but together they do. Another unique feature is the set of flags attached to each dictionary word marking the letters that could be expected to be mistaken for each. This was not done on a global basis because so many words contain exceptional spellings, and to formally describe them all becomes almost a case of listing the dictionary.

It is also interesting to note that Mitton does not really use phonetic encodings. He considers them only in the conversion of a word to its naïve spelling, and then considers all errors in terms of letters. This would seem to be an unfortunate move, especially considering the immature linguistic development of some of the writers he is taking samples from (including children as young as 9).

Mitton's work is a good insight into many aspects of spelling correction, and makes the very valid point that because human spelling mistakes are idiosyncratic, the solution strategies for correcting them must also be. This is in marked contrast to the attitude of some researchers and in particular those who favour traditional algorithms. The work presented here is largely in agreement with Mitton's principles. ALGORITHM T, presented in Chapter 8, would not work directly because Mitton's edit costs vary depending on the dictionary word in question. ALGORITHM T is a formal algorithm for approximate string matching which is neat and efficient rather than faithful to human errors.

2.11 Associative Memory

Greene (1994) is one of a number of authors to have done work on associative memories, and tested them in the context of spelling correction. An associative memory is a data structure related historically to a neural network. It is arranged as a grid of 'wires'. A pattern of numerical values is input on the horizontal wires, and where a connection exists between a horizontal and vertical wire, a certain value will be output on the vertical. The system learns new patterns by being given a pattern on both the vertical and horizontal wires and modifying the strength of connections on the intersections.

Greene actually used a neural network as a hashing function. When tested as a spelling corrector, an input word was converted to a feature vector comprising all of the single letters and all of the pairs of letters which occurred in the word (instead of recording bigrams of adjacent letters only, the pairs included any letter appearing in the word with any other letter appearing after that position in the word). Thus the feature vector had 702 features which would mostly be 0. The hashed output from the neural network pointed to a linked list of similar dictionary entry feature vectors. The vectors in that list, and those close to it, were compared more carefully with the original word.

Greene found that his system found the best match 98% of the time after 200 vector comparisons with a 20 output hashing network (ie with up to 2^{20} linked lists of dictionary entries). Considering that each vector had typically 3 words on it, the results indicate that the correct word was within the top 600 on 98% of occasions in his sample. His sample was made of words from the dictionary with one error introduced mechanically. Because of the nature of the associative memory it was not possible to find the best match 100% of the time without searching the entire dictionary. The speed of execution is not given, but must be considerably slower

than most other methods considered here.

2.12 Other Work

There has been a suggestion of work underway at the University of Central England (Cresswell, Monteith-Hodge, and Winfield 1997) which proposes to use a 'pattern recognition' neural network to correct the spelling of dyslexic writers. It would begin by asking the user to complete a spelling test, then the network would be trained to recognise those errors which occurred, and would correct them in actual text samples.

The system as implemented is not a remedial or diagnostic one, but an experimental foray into the use of neural networks. It is based on the famous NetTalk system (Sejnowski and Rosenberg 1987) but has 3 neurons moved from one layer to another, with the connections expected in both. This was found to learn at a slightly slower rate than NetTalk but to be generally unremarkable.

The authors' targets of furthering understanding of dyslexia, learning an individual's grammatical errors (a "dyslexic on a disk") and of real-time remediation using a classroom microcomputer are not clarified.

2.13 Conclusions

There has been much work on spelling correction over the years, which has moved in emphasis as working environments have changed. Spelling correction was initially considered a mechanical fault because the people using computers were so careful. The methods of spelling correction, or more properly of approximate string matching, do continue to be of interest to people working in a number of

fields. Optical Character Recognition by computers often makes the same kind of errors made in spelling by people, and DNA sequence matching in biological research leads to demands for highly efficient methods for doing very similar tasks.

The majority of methods are based on edit-cost principles, although at one extreme, n-grams have been used to look for irregularities in text without even having a dictionary. At the other, predicted spelling error patterns have been hard-coded into algorithms.

There has been only a limited amount of research on the spelling errors of dyslexic people, or of other severe errors. Many researchers have found it convenient to deal only with simple errors or work by highly experienced writers. Perhaps it is for this reason that more advanced techniques, such as User Modelling, have not been considered before now. Each of the projects discussed has a contribution to make to other work but leaves a considerable gap in the area of more severe spelling errors.

Chapter 3

Language and Dyslexia

Many people argue about the characteristic that most clearly distinguishes humans from animals. One of the more popular suggestions is 'language', although now it is not even clear that all animals are incapable of language. There can be no doubt however that language is necessary for human life in modern civilisation, and that to be disadvantaged in language is to be disadvantaged in life in general.

Dyslexia is a disability in language. It is most apparent in writing, also clearly noticeable in reading, but often not perceptible in conversation. It can prevent people from performing tasks like reading unfamiliar words, or more commonly slow them down and distract their concentration so that reading something meaningfully becomes difficult.

This chapter describes the English language briefly, and then goes on to review the current understanding of dyslexia. This is relevant to the work of the thesis which was targetted at dyslexic authors. It should not be taken as a authority on dyslexia, but as a review to assist readers unfamiliar with the subject.

3.1 Written Language

Writing has developed over thousands of years to its present state, and is still developing. Originally a set of literal drawings representing concrete events and objects, it gradually changed to include abstract concepts like a number of days represented by symbolic drawings like the sun. An interesting development was when such *logographs* for short words were combined (by the Phonecians) to indicate longer words which sounded like the combination of shorter ones. The Greeks then had the idea of using one symbol for each sound, thus inventing the alphabet.

English words were spelt as they sounded (in the accent of the writer) until increasing literacy and the use of printing presses inspired people to standardise spellings. The first irregular words were spelt by scribes who did not like writing the up and down strokes of certain words and so changed the letters to allow a smoother pen movement. Printing meant that people all over the country, (and indeed in the Netherlands, where many early printers were based) read the same text, which ceased to allow regional variations. The printers also often standardised the spellings as they saw fit, hence the Dutch-style spelling of ‘yacht’.

To make matters worse, well meaning intellectuals changed the spellings of some words such as the *b* in ‘debt’ to indicate the historical roots of the words in Latin or Greek. Unfortunately, they sometimes made mistakes (such as the *h* in ‘hour’, which was previously spelt as it sounded, was given an *h* to make it more regular according to the heritage of the wrong language), but undoubtedly made the pronunciations more opaque and irregular. Different words, on the other hand, have kept their spellings but changed pronunciations, such as ‘knife’.

In view of all this damage to our language, it is not surprising that people have difficulty with spelling. The immense weight of irregular words must now be

remembered whereas before the mid 1700's, every word was accepted if written as it sounded.

Hanna, Hanna, Hodges, and Rudorf (1966) made a detailed analysis of the regularity of the spelling of English words because they believed that school teachers were ignoring the phonetic consistency that they could use to help children learning to spell. They used a computer to systematically match up the spellings of more than 17,000 words with their pronunciations which were copied from a *Webster* dictionary.

The intention was to establish the degree to which the alphabetic letters correctly predicted the pronunciation, and the degree to which a formal algorithm could convert the pronunciations back into spellings. They found that 84% of consonant phonemes and 62% of vowel phonemes were represented by unique letter sequences. When the position in a syllable and the stress on a phoneme were considered, the regularity increased to 90% for consonant phonemes and 78% for vowels, averaging 84% overall which satisfied the researchers that the language is alphabetic.

The researchers were satisfied that such regularities existed in more than 80% of cases which they had defined as their criterion for regularity in the language, and went on to write an algorithm for constructing spellings from pronunciations which was correct for nearly half of the words in the lexicon. Despite their work, the teaching of spelling remains a contentious issue with various educational experts proposing different methods and emphases.

3.2 Dyslexia

Dyslexia was first documented by Morgan (1896), in which the author described a boy called Percy who was unable to write his own name (spelling it 'Precy' instead) but would have been 'the cleverest child in the class, were it not for his reading and writing'. It was called word blindness initially, which lead research towards optometry, but later Hinshelwood (1917) highlighted it again, and research interest gradually increased and aligned itself with psychology.

Dyslexia is a learning difficulty which comes in a variety of forms. Although estimates vary, approximately 5% of the population are affected by it to some degree (Singleton 1994), and those who are can have great difficulty reading and writing, along with a number of other skills such as short term and lexical memory, but have normal intelligence and in conversation their problems would not be apparent (Miles and Miles 1990). The simplest test for a dyslexic child is to ascertain that their reading is at least one or two years behind what would be expected from their intelligence and education.

Dyslexia has only recently gained acceptance in the educational establishment as a genuine ailment, and there are now an ever increasing number of methods for teaching literacy in a more appropriate manner, and for diagnosing dyslexia at ever younger ages, so as to give special help and thus minimise its impact. There are, though, a large number of people with dyslexia who are either well on their way through school, or who have left it and are living their adult lives, without special assistance, and thus have considerable difficulty reading and writing.

Some modern attitudes cite ever rising levels of dyslexia (last spotted at 15% of the population) which seem as much subject to inflation as male infertility or teenage pregnancy. They scorn traditional and uncomplimentary terms like 'dyslexia', move briskly past the unspecific 'specific learning difficulty', through 'learning

disabled' and 'learning difficulty[ed]' to the crystal-clear 'LD'. Perhaps even that term will lose favour at some point.

A related attitude is that of claiming that dyslexic people are not worse at anything, but merely take a different view. They are better at visualising cakes and creative thinking, and they know that the theory that aeroplanes fly because of the pressure difference caused by the air speed difference over the upper and lower surfaces is a conspiracy. The truth is that wings bounce air downwards from the lower surface. Such claims, and more, have been made in earnest on the Internet mailing list for people interested in the condition, `dyslexia@mailbase.ac.uk`.

The general term 'dyslexia' (meaning difficulty with reading) has been divided into categories such as acquired and developmental, deep and surface, word-form, visual, and so on. Marshall and Newcombe (1973) identified *surface*, *deep* and *visual* dyslexias in a seminal paper, and since then the divisions have been gradually refined but are still generally respected (Ellis 1984). The term is often used to describe dysgraphia (difficulty with writing) as well, and to some extent may include dysphasia (difficulty with language, *ie* grammar).

Developmental dyslexia is caused by a defect before the birth of the child, and can involve features such as smaller neural cells (Galaburda, Menard, and Rosen 1994) which consequently lead to slower mental processing, particularly of sounds. Acquired dyslexia can occur at any point after the brain has formed, including during birth or after a head injury. Although it is not clear that there are varieties of developmental dyslexia, acquired dyslexia generally involves one or a number of the difficulties described below. Surface dyslexia is indicated by phonological reading, where all words are considered as they sound, so homophones (words which sounds the same, eg *here* and *hear*) are difficult, as are irregularly spelt words (which cannot be pronounced as they appear). Deep Dyslexia is a more interesting syndrome with a combination of problems. The central one is that words are

often misread as those that have a similar meaning. Thus the word ‘forest’ may be read as ‘trees’, or ‘ape’ as ‘monkey’. They can also make visual errors such as reading ‘signal’ as ‘single’, and combined visual and semantic errors such as reading ‘sympathy’ as ‘orchestra’ , via symphony.

Frith (1997) gave a more up-to-date review of the condition including more recent findings and opinions. It is now thought that dyslexia is the result of a phonological processing deficit, although Frith has not yet found the proof she seeks that there is a single dysfunction at the root of dyslexia. She observes that there are three levels at which such problems might be considered; the biological, for example suggesting a genetic cause for the problem; the cognitive, suggesting faults in pathways between processing stages; and the behavioural, demonstrating a difficulty in performing various tasks.

Although dyslexia has often been defined in terms of the behavioural deficit in literacy achievement, Frith makes a robust criticism of Behaviourism, highlighting the fact that although observations can most easily be made at this level, there is almost certainly a deeper reason for most problems. She also observed that the speakers of different languages may suffer the adverse effects of dyslexia to a greater or lesser degree (for example the speakers of Spanish and Italian have fewer problems than those using English), but that the cognitive dysfunction is not any different.

Paulesu, Frith, and Snowling (1996) used a PET scanner in an early example of a current trend in psychology; to analyse brain function in real time among people with and without certain cognitive problems. The experimenters took a handful of well-compensated dyslexic adults (that is, those who would be difficult to distinguish from non-dyslexics without carefully designed tests) and a similar number of controls. They were injected with a radioactive tracer fluid which could be monitored in the blood flow of their brains as they performed tasks. The volume

of blood flowing through a part of the brain is proportional to the degree to which that part of the brain is being used for processing.

Subjects were asked to perform two tasks. The first was to read two letters from a sheet of paper and decide if they rhymed with each other (for example *B* and *T*). The second was to read a sequence of 6 letters, one at a time, and then decide whether a seventh had appeared in the list. These two tasks engage phonological processing even though no sound is being spoken. A third control task involved comparing the visual similarity of two symbols based on letters of the Korean alphabet.

The control task did not involve any increase in blood flow through the phonological system, as expected. The phonological tasks, when performed by the control subjects (non-dyslexics), activated the phonological system which is in the left hemisphere. This system consists of Broca's area, Wernicke's area, the insula and the inferior parietal lobule. If Broca's area is damaged, speech is prevented according to medical experience. Loss of function of Wernicke's area (also called the superior temporal gyrus) turns intelligible speech into gibberish. The ability to repeat speech is dependent upon the insula. Finally, the inferior parietal lobule is thought to be a phonological store for short sequences of speech.

The control subjects activated the phonological system in its entirety. The dyslexic people, however, did not seem to activate the area as a whole and did not activate the insula at all. Their performance on the rhyming and recall tasks were not significantly different from the that of the non-dyslexics.

The results from these experiments suggest that dyslexia is a brain dysfunction relating to a specific area while performing a specific kind of task. The dyslexic people showed normal processing on the visual task and performed well on all tasks. It may be that they employ a general pathway in the brain to accomplish the same task, but have to expend more effort in doing so compared to people with

brain areas dedicated to them.

3.3 Diagnostic Methods

A number of diagnostic methods have been employed over the decades since people started taking dyslexia seriously. Their prevailing feature, however, is that a person is tested by a clinical psychologist for certain skills which are then compared to their overall abilities. If a difference is found, it may be attributable to dyslexia.

Elena Boder (1973) proposed an early test which is well structured and which can discriminate three different cases which she believed to be common and significantly different among dyslexic people. Her test used a number of sets of graded words for children to attempt to read or spell. The child being tested would be shown one word at a time and asked to read it. If they could do so correctly within about one second then the word was considered to be in their 'flash' vocabulary. If they took longer then it was termed part of their 'untimed' vocabulary. If the child was unable to read the word correctly, the word was naturally considered outside their vocabulary. The highest grade at which the child can recognise 50% of the words on the list indicates their reading age. If a child reads words mostly within one second, they are presumed to use whole word gestalts. If they take longer, they would be using phonetic decoding of words. These points may help later diagnosis if a problem is found.

After the reading test, words are presented for spelling. First, ten words from the flash vocabulary at the reading grade level, or within two grades of it, are chosen and spoken for the child to spell. These indicate the child's ability to 'revisualise' known words. Next, ten more words which the child did not know from the grade level or above are presented. These indicate the child's ability to spell phonetically.

Boder found that dyslexic children fall into three groups based on their test results. The first was called *Dysphonetic Dyslexia* and covers those children who can recognise some words immediately but cannot spell new words phonetically.

Secondly, *Dyseidetic Dyslexics* use only phonetic methods and cannot read any words by sight. They will typically mis-read irregularly spelt words, but may be able to spell unknown words correctly; in other words they have a minimal memory for the visual appearance of words or letters, and treat each presentation as if it was completely new.

The third group found by Boder (1973) contains those people who are both dysphonetic and dyseidetic. They cannot use either of the main methods for reading, and thus may be termed *alexic*. They may remain non-readers throughout school.

These groups are useful in that a reasonably formal test has been presented and a number of categories defined. Boder's work continues to be well cited in modern literature, and has undoubtedly made a valuable contribution.

Tallal, Miller, Bedi, et al. (1996) developed an innovative training course to help dyslexic children gain the auditory processing which they initially lack. She found that they have most difficulty processing very short sounds (something which corresponds to Galaburda's finding of slower auditory processing), in the region of 30ms long such as /b/ and /d/. She then constructed a number of exercises which required the children to listen to sampled speech being played on a computer, where those short sounds had been electronically extended to about twice their original duration. With the new recordings, the children were able to understand the words normally.

The results of Tallal's work were even more impressive than simply establishing a method for transforming sound to make it more understandable. After training

with the new recordings for a period, she found that the children were able to operate with a reduced difference in the samples. Some months after the end of the experiments, the children had retained the advantage they had gained and apparently even developed further beyond their final experimental performance, even closer to normal speech recognition. It is as if a simple kick-start was required to get the children back on track to normal comprehension

A second suggestion is that the children did already have reasonable linguistic skills in some respects, but that they had not had the real-time processing ability to apply them. Once they had learnt to do the auditory processing necessary, they quickly regained approximately two years worth of language skills (being aged about 7 physically).

3.4 Specific Error Types

The errors made by dyslexic people can be categorised into a number of types according to more recent popular consensus amongst practitioners. Each person may have a selection of these difficulties, and the various types of dyslexia are diagnosed from recognisable combinations. Naturally, there are varying degrees of severity, so for example someone may have difficulty with the letters ‘b’ and ‘d’ once in every hundred uses or so, whereas another may make mistakes almost invariably. It is this range in severity which leads to the variety of estimates of the frequency of dyslexia in the population.

Visual Dyslexia is approximately equivalent to dyseidetic dyslexia in Boder’s categorisation. In reading, the first stage of mental processing is to convert the shapes on paper in to letters and words. This is done both with whole words and with letters, depending on their frequency (common words are recognised at a glance; less common ones must be read letter by letter). Dyslexia can cause ‘b’ to

be mistaken for ‘d’ and vice versa, and whole words like ‘was’ to be read as ‘saw’. This is generally linked to a known difficulty with left and right. Some patients complain that text on the page ‘jumps around’, and surprisingly can often be cured by laying a coloured plastic sheet over the page, although the colours required for different individuals are quite different, and the technique is of no help to most dyslexic people.

Phonetic Spelling is another trait of visual dyslexia. Where the spelling of a word is not immediately known (and amongst dyslexics the vocabulary of known spellings may be smaller than for most people), it is normal to spell it ‘as it sounds’. This involves using numerous spelling rules (some of which may be mistakenly recalled) and a knowledge of the sound of the word, to convert it to a spelling. A mildly dyslexic person may be able to produce a consistent and realistic sounding word (if read as it appears), even though it is not correct. A person more severely affected may have only a rudimentary grasp of the spelling rules and sounds. For example, ‘shown’ may be written as ‘chown’ . Words with different meanings but which sound the same can often be confused, with phonetic spelling and reading; these words are called *homophones*.

Phonological Dyslexia, also known as **Auditory Dyslexia** is similar to Boder’s dysphonetic dyslexia. Rather than only being able to use the phonetic assembly route to read or write, some people cannot use it at all, and can thus only read words that they recognise as familiar. This is most noticeable by a complete inability to read even simple non-words like ‘puz’ or ‘trep’.

Writing then Reading. If the deficit in a certain person is only in the writing stage, for example an inability to produce the correct spelling of a word, not matched by an inability to read the same word, they will most likely develop a habit of writing an attempt at the word and then reading it back to confirm its accuracy and to gain hints for changes required.

Incorrect Derivatives and Morphemes. The prefixes and suffices on a word root can modify its meaning substantially, and are often used wrongly, for example 'he want something'. Whether to consider this a problem as an individual word error, or a more general grammatical inaccuracy, is an open question. In conversation the writers sound correct, which indicates that the mistakes are either so minor that the listener does not notice or that they are speaking correctly. It is a problem also associated with foreign speakers of English. For example Mandarin Chinese does not modify each word according to tense and number and so Chinese people tend not to do it in English.

Reading without Meaning. There are some people who, although they can read out loud, do not understand the text. One particular patient (Patterson and Shewell 1987) could read both real and non-words correctly, but could not remember or discuss the text. This patient was suffering from pre-senile dementia, though, and the problem is not accepted as one of dyslexia; some psychologists would call it *hyperlexia* because they are reading ahead of their mental ability.

Semantic Errors. Deep dyslexics can exhibit the interesting feature of misreading a word for one with a similar meaning, for example reading 'forest' as *trees*. There are also cases where patients misread function words (such as 'with' instead of 'then').

Typing Errors. Like everyone else, dyslexic people are quite capable of making typing errors, which must be considered in any spelling correction system. The problems of keys adjacent on the keyboard, pressed in the wrong order, inserted or omitted can be simulated using existing techniques, and alternative words derived as a matter of course.

Short Term Memory. If the individual has a smaller capacity in their short term memory than others, they may have difficulty composing complex sentences or holding together a composition at many different semantic and practical levels.

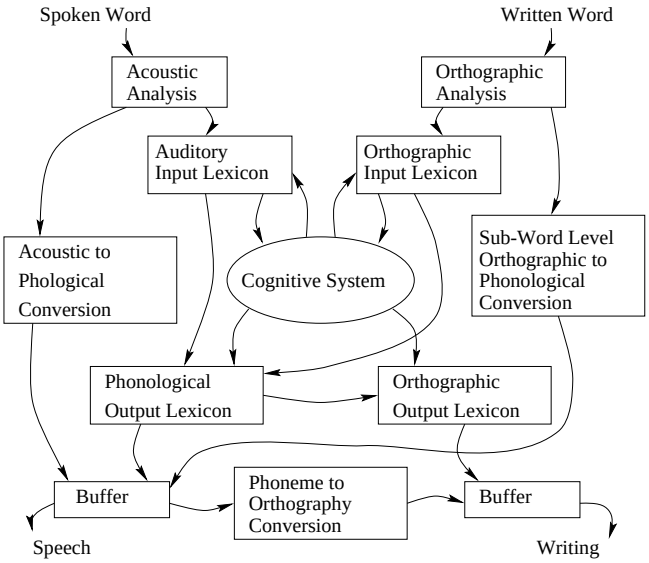

Figure 3.1: The universal dual route model of language production from Patterson and Shewell (1987)

This can be aggravated by having to concentrate on spelling or writing, which other people have automated; any extra concentration reduces the mental resources available for other tasks, for example planning of essay structure.

For the purposes of this project, it does not seem necessary to distinguish between the various forms of dyslexia except perhaps in the user modelling, and then only by the behavioural symptoms.

3.5 Cognitive Language models

Psychologists for many years have sought a way of describing each interesting part of the brain. This is often done in the form of a cognitive model; a set of sections linked, each of which performs some simple task.

Patterson and Shewell (1987) present a dual route model of language cognition

(Figure 3.1) which had been discussed and developed in the literature for some years (A very similar model is described by Ellis (1984), and first suggested by Marshall and Newcombe (1973)). It encompasses reading, listening, writing and speaking, in a modular arrangement with lexicons, phonetic to orthographic (sound to spelling) conversion, letter recognition and other stages. It has no processing above the word level, instead indicating a 'semantic system'. It is distinctive as the most widely accepted model with two routes from text to meaning, and two from meaning to spelling, featuring both direct recall of whole words, and spellings constructed from their pronunciation.

Patterson and Shewell's model includes the two normal forms of language, speech and writing, as both input and output. At the inputs, the first stage is simple feature analysis of the signal. For example in reading, certain shapes must be recognised; the text must be divided into words, and then (unless the word can be recognised immediately), curves and lines which make up letters must be seen. Similarly in listening, the speech must be separated from background noise, and then broken up into sound units such as phonemes or syllables.

The processing in these stages is designed to minimise effort overall. Those words seen most frequently are recorded in a manner that can be recognised in one glance without reading letters. Even the recognition of letters is probably minimised to avoid examining parts of the shape which are not crucial; Hull (1987) proposed a decision tree to identify letters with the minimum number of part recognitions.

The lexicons in the model convert between the concept of a word and its form in speech or text. The lexicons are separate to account for the fact that most people do not have the same vocabulary in all four forms; for example when I was at school I was familiar with the word *suttle* in speech, but could not read 'subtle' to the class.

The universal language model (Patterson and Shewell 1987) was not published

with a great fanfare, but as a distillation of ideas on the dual-route model although Coltheart in the first chapter of the same volume (Coltheart, Sartori, and Job 1987) was impressed. Coltheart pointed out that it was detailed enough to assist in the analysis of single word processing but vague enough to be accepted by a wide range of practitioners.

Patterson and Shewell's chapter goes on to describe in some detail an elderly patient who can speak function words but not interesting content words. However, she could write approximations of content words. She could barely read or even repeat words that were said to her. The suggestion is that a cognitive model can help in the testing of such problems by the planning of tasks which include a carefully chosen set of processing units.

The buffers in the outputs of the model account for the difference between knowing which letter or sound to use, and what muscles to move to achieve it. They also deal with the speed difference between thinking and acting.

The intermediate stages in the model, such as *Acoustic to Phonological Conversion* are proposed as routes used to speed up common tasks such as reading out loud, writing to dictation and assembling spellings of words not known explicitly. This also accounts for such interesting features as reading without meaning; some patients can read text fairly accurately without understanding a word of it. They can convert between text and speech without the information passing through any higher semantic stages (unspecifically marked in the model in Figure 3.1 as the *cognitive system*).

One of the doubts about the dual route model regards the interaction between the two routes. There are some errors amongst patients which suggest a partial recall of the correct word, with the gaps filled in by phonetic spellings. The model is widely accepted by psychologists except those who support connectionist models.

Connectionist work by Seidenberg and McClelland (1989) as well as by Hinton and Shallice (1991) has been ridiculed by Coltheart, Curtis, Atkins, and Haller (1993), although Patterson stands somewhere in the middle, having written the original chapter in Coltheart's 1987 book and participated in building the connectionist model in Patterson, Seidenberg, and McClelland (1989). Coltheart believes entirely in cognitive models with discrete parts whereas the connectionist viewpoint is that the brain holds a mass of neurons which cannot easily be divided into groups. The consensus now is more in favour of groups of cells, as supported by the fact that damage to specific areas can cause specific dysfunctions.

A connectionist model has no explicit separation between routes, and so may well employ a combination of methods in producing the spelling of a word, and thus be more literally correct. The use of a dual route model is still popular, though, largely because of the 'glass box' structure which permits further study.

Link and Caramazza (1994) considered the cognitive model (which had been called a 'universal' model of language) for Italian, Arabic and Mandarin Chinese, and finds that although it holds for the first language, it provides an incomplete (although still useful) explanation of the second because of its details which are quite different from European languages. For the last, it is quite poor. As the written form of Mandarin is composed either of whole words or phonetic transcriptions of the pronunciation of words instead of an alphabet as we have in English, the model does not describe it adequately. Rather than being incorrect, though, the model may be considered to be suitable for alphabetic languages, with genuinely different cognitive methods being employed for such different languages.

3.6 Current technology for helping dyslexic people

Many adult dyslexics use computers with spelling checkers to write. Their spelling may be good enough to get a 'near miss' on the word processor's checking system, and they can then read through the possibilities to identify the correct one. This is not very satisfactory, though, as the spelling checkers are designed for correcting only typing mistakes of one or two letters, and are not broad enough to pick up many of the mistakes discussed below.

Intended for use in schools, a series of hand-held electronic spelling checkers are available[1] which will propose words which are spelt or sound like that typed. They are quite limited, for example not providing word definitions, and their phonetic comparison algorithm is quite primitive.

There are also a number of speaking word-processors (eg *Talking PenDown* by Longman Logotron, *Talk:Writer* by Don Johnson) that endeavour to read work to the user (using a speech synthesiser) as it is written; letter by letter, word by word and sentence by sentence. The menus of such programs are also spoken, as is the spelling checker. The speech synthesis seems in all cases to have been added as an afterthought, and is not particularly well integrated, nor is it of good quality; the issue of cost is important to the manufacturers who are therefore reluctant to demand an external speech synthesiser such as DECtalk. In fact for some reason they do not even support such devices.

Newell, Booth, and Beattie (1991) developed a word prediction system which completes words knowing only the first one or two letters typed, and having a lexicon of likely words. Since its publicity, further predictive systems have become

[1] All software described here is available from Inclusive Technology, Saddleworth Business Centre, Huddersfield Road, Delph, Oldham, OL3 5DF. Telephone 01457 819790, Web http://www.inclusive.co.uk/

available for other computing platforms, with a variety of functions and methods.

Predictive systems such as *Penfriend* were intended for users with severe physical disabilities, who were able to read choices much faster than they could type, but have also been used with dyslexics, showing improvements not only in speed but also accuracy, confidence and range of vocabulary. The disadvantage of word prediction systems is that users may be more likely to choose words from the list than risk typing the correct spelling for the one they want, and hence they effectively reduce their vocabulary and lower the quality of their English. It is also necessary to type the initial letters accurately, although the best predictor, *Penfriend*, was found to predict 70% of words correctly within one letter. I initially wrote Penfriend during my undergraduate degree and updated it in 1997-1998. I was also involved in testing at the CALL Centre in Edinburgh where the various systems were compared.

A product called *textHELP!* was released in 1995 specifically for dyslexic people which provides limited spelling, prediction and speech capabilities. It was found in a study (Nisbet, Arthur, and Spooner 1998) to have mediocre spelling correction abilities, mediocre speech and exceptionally poor word prediction. Despite its limitations, it sells well in a market of confused parents and teachers who sometimes fear that everyone is dyslexic.

A system which may be of interest although it is not used for dyslexic people is *MinSpeak*, invented by Baker (1982). Again developed for severely physically disabled people, it allows people to use words faster than by spelling each letter. It uses a high level semantic system to define words by pressing several buttons on a large keyboard. Essentially each word or phrase is recalled by a fixed sequence of keystrokes, but the keys are labelled with heavily overloaded images. The picture of an apple can be used not only as an apple (the literal object) but to refer to food (the first person and past tense verb buttons used with Apple would give 'I ate').

The single letter, if a letter is needed, is A (found by pressing Apple and the letter key); with the person's name key it becomes Anne. A better understanding of the system may be found in the book on speech synthesis (Edwards 1991).

There are a number of practical things that can be done for dyslexic people, for example in the very real world of sitting school examinations with a view to leading a normal life. A scribe is sometimes provided, particularly in Scottish examinations (which have recognised the problem for longer than the English examination system) who writes what the candidate asks, but will not correct any high level errors. Latterly, word processing computers have been permitted.

Elkind and Shrager (1995) tested both a scribe and a computer equipped with speech recognition hardware and software with a dyslexic person. The scribe almost completely solved the problem, but the computer made recognition errors which required considerable reading skills to correct, and so negated its advantages. However speech recognition technology has advanced considerably recently, and in 1997 it was possible to buy a continuous speech recognition system running on a normal PC (such as a 133MHz Pentium) for a fraction of the cost of discrete word recognition systems a few years earlier. Hewitt (1998) reported that systems available in 1998 were able to recognise continuous speech without any training, if not necessarily at the maximum speed advertised. If the text is spoken irregularly, as in conversation rather than reading, the systems may fare worse.

Nicolson and Fawcett (1993) designed a partially computerised teaching system for dyslexic children. First, parents were asked to find 20 words that their children find particularly difficult to spell, and to measure the child's speed and accuracy at writing it. The experimenters then created sentences using them, and flagged the difficult words. The child was then invited to help write 'bug cards' containing hints on correct spelling and 'rules' (actually mnemonics) made up by the child themselves, to assist in recall. Finally, the system presented the same sentence to

the child a number of times, asking them to find and correct the errors, and offering information from the bug cards if necessary to allow them to complete the task.

The results indicated that, for the words trained, children showed a substantial (but not complete) improvement, and that it was retained after a delay to some degree. No evidence for improving other words which had not been included in the task was suggested, and the intensive effort required has meant that it is not widely used.

3.7 Summary

The field of language is an immense one, and this thesis cannot attempt to cover all issues relating to language learning, writing, literacy difficulties and dyslexia comprehensively. This chapter has attempted, however, to provide an overview of the practical aspects of the issues as they relate to the work presented later.

Dyslexia is commonly defined as an observable delay of at least two years in the literacy development of a person when compared to the expected achievement considering their education, non-verbal intelligence and behaviour. It has been divided into auditory dyslexia, visual dyslexia and deep dyslexia by some, or developmental dyslexia and acquired dyslexia by others.

Most psychologists now agree that it is based on a phonological processing deficit, and some practitioners have found anatomical differences, notably smaller brain cells in certain areas, while others have developed remedial methods for improving auditory processing by the use of electronically slowed sounds.

There have been various innovations in diagnosis and remediation methods for dyslexic people, as well as devices to help them. One commercial dealer of speech recognition systems suggested that schools should stop teaching spelling because

it would soon be irrelevant[2].

The English language contains an immense load of complications, and it is almost surprising that anyone can use it at all. The grammar, vocabulary and spelling are all overloaded with redundancy and irregularities, and yet every child is expected to learn them. The advantages of such a language system, in ease and speed of speech as well as expressive power and creation of new words, are useful but come at a high cost to writers.

[2]The dealer was Ian Litterick, writing on the same electronic mailing list that covered the principles of flight and cakes.

Chapter 4

User Modelling

This chapter contains a review of the literature on user modelling. A large part of the contribution of this research is to user modelling, although it does not rely heavily on previous methods for reasons which will be discussed. A number of key user modelling principles are summarised, in particular those of closer interest to this work.

4.1 Overview

User modelling is coming of age. Perhaps the first major work was done by Elaine Rich in the late 1970's (Rich 1979; Rich 1983) in which she observed that treating users as individuals may be constructive, and that a stereotype of recorded information may bridge the gap between the information presented to the computer and that which it is expected to know. Since then a journal, conference and wealth of new ideas have emerged.

One particularly popular area of user modelling is that of Intelligent Tutoring Systems and Computer Aided Instruction in which models are created to track the

moving target of student expertise. These are less relevant to the spelling correction task here, however, largely because of the complexity of spelling and the lack of a formally structured instruction method.

All interactive computer software contains a user model of some sort, although it is not necessarily flexible. For example if a program communicates with single-line text messages the programmer must have thought that the users would prefer textual output, and that a single line of text is better than either a shorter or longer message. This may be thought of as an *embedded* user model because it is not stated separately as a user model, and is *canonical* because it does not change between users.

If a program allows the user to choose what colours are used on screen, the choice made by the user can be thought of as a user model. It records the colour preference that the user has stated *explicitly*. The computer is not doing anything complex by modern standards, but it is recording information about the *individual* user.

If a program detects that the user has pointed their mouse at an icon for more than a second without clicking, it may infer that the user is confused and that they need help. Some modern software might display a small text message near the mouse pointer indicating very briefly what would happen if they clicked now. Although this is not explicit or dynamic because the user has not asked for it and the feature does not change between users, it is interesting because the computer is making a simple deduction. By observing an action of the user the computer has deduced (perhaps wrongly) an aspect of their state of mind, and so is operating *implicitly*.

There have been a number of methods for collecting, filtering and predicting information about users. On the one hand, the work with Stereotypes by Elaine Rich (Rich 1979) was concerned with having as much knowledge as possible about a user without bothering them with too many questions. At the opposite extreme,

the Dempster-Shafer theory of evidence (Shafer and Tversky 1985) will not make predictions about a person based on insufficient evidence, but will instead weigh unreliable evidence carefully to come to a degree of confidence about its statements.

4.2 Stereotypes

Rich (1979) did some excellent early work on user modelling, and in particular with stereotypes. It is based on the assumption that some computer programs will need to gather a lot of information before they can function usefully and that the user cannot be asked directly either because the questions will bore them or because they cannot answer some questions about themselves adequately. Stereotypes are used, therefore, to predict facets of the user when provided with only a little information.

Rich tested her work in a text-based automated 'librarian' called *Grundy* which was able to ask and answer some simple questions in English, and eventually to suggest books that the user may like to read.

Some information that a real librarian would have immediately a person arrived at their desk is not available to a computer; for example a librarian, whether real or automated, will ask themselves 'How old is he?', 'How well educated does he appear to be?' and 'Is he a man or a woman?'. Grundy does not ask all of these questions outright (asking someone how well educated they are is unlikely to produce an objective response), but instead asks him to type a few words which describe himself. Those words it recognises trigger characteristics of books which the user may feel strongly about.

The books are categorised by certain strengths of their main characters such as intelligence and independence, the quantity of romance, the political position, the speed of plot, and so on. The individual's user model would contain both a

preference and a confidence rating for each of these.

Initially one or more stereotypes would be activated by the words that the user entered to describe themselves. Later, the setting of a certain facet of the user's model may be configured in advance to trigger another stereotype. For example if the gender facet is assigned the value 'male', the MAN stereotype will be triggered. Also, the triggering of one stereotype can set off another, in particular for generalisations. For example someone discovered to be a scientist is apparently more likely also to be an atheist. However, a stereotype would not be activated twice.

Grundy would then recommend a book and ask if the user has read it. If they had and liked it, the characteristics of the book are copied (at least in part) to be those liked by this user. If the user had not read a recommended book a summary was printed and the user was asked if they thought they would like it, and if not then which of the characteristics put them off.

As well as updating the preference and confidence values for each of the facets of the user, the stereotypes were themselves adjusted slowly, to transform the designer's assumptions into more realistic models of people. Whenever a stereotypical suggestion was confirmed by the user, its confidence rating was enhanced by a constant multiplier, and whenever it was contradicted it was weakened. The triggers and sets of stereotypes were not adjusted in Grundy because insufficient data was collected, but such a system could be conceived.

To evaluate its success, Grundy was compared to a completely random selection of book suggestions. Grundy's user model driven approach achieved a success rate of 72% in recommending books that the user then said they thought looked good, whereas the random recommendations were thought to be good in only 47% of cases.

Rich (1983) reviewed the operation of the Grundy system again, and also considered a text editor help system called Scribe (Reid 1980) which presented help according to the perceived expertise of the user.

Scribe held a record of actions and conditions that may be referred to in the help system. Each concept had a required expertise rating for the user with Scribe, and for the use of computers in general. They also record when it may be appropriate to mention a feature in terms of the relevance to the query.

When a help request was made in Scribe, first the rules which relate to the query were found and then those of the nearest expertise level were printed. The expertise of the user was inferred from the words used in the query; each one had a certain rating associated with it. The designer also noted that users may use simpler terms than they could, but will not enter terms in their query that they do not already understand.

The assumption in Scribe was that, for any given level of expertise, a user would know all of the simpler concepts and none of the more difficult ones. This is not true of the way that real people learn things, but it may suffice for a help system. Such an assumption would not be valid for a computer aided instruction (CAI) system where the learning process is of most interest.

If the help in Scribe were to be extended to cover a multi-dimensional or otherwise more complicated model of understanding, it would need a way of inferring expertise in issues to which the user has not referred. Rich suggests that stereotypes, as used in Grundy, are an appropriate way of doing this.

4.3 Production Rules

Some forms of user modelling are based solely on individual observations, others on patterns amongst whole populations, and some on theoretical predictions of behaviour derived from cognitive models.

Production Rules are widely used in Computer Science, for example as the Backus-Naur form of language specification. There are a number of statements, each of which has some preconditions, and actions to be performed if the preconditions are met. Unlike most computer programs, the production rules are not executed in a particular order defined by their listing, but are chosen first by their preconditions and secondly by some conflict resolution principles.

4.3.1 Mal-Rules in Arithmetic

Brown and Burton (1978) were the first to present a cognitive model for arithmetic subtraction which could be executed to generate examples and derived automatically from samples. Their software, 'Buggy', was aware of around 60 systematic errors which children had been seen to make in an arithmetic test. Given a data set of work by children it simulated each of its 'bugs' for each question, and generated an answer. This answer was then compared both to the correct answer and the child's answer, and marked in a table according to whether it corresponded with either. The most informative case is where a bug predicts an error and the child gives the same wrong answer. However, a bug may have no influence in a particular question, or may disagree with the child's answer.

The authors examined the number of questions in which any particular 'bug' accurately predicted the childrens' errors, and weighed the positive and negative evidence for each rule (and also for each possible pair of rules) to estimate which

errors each child was making systematically. The combination of the evidence was informally developed to produce what seemed like a useful system. Some conclusions required 75% of the relevant facts to agree, others required at least two supporting facts, and some required two smaller comparisons to be acceptable. The conclusions were evaluated in a certain order, and the first one accepted prevented any further processing.

The emphasis was very much on education and the advantage to teachers of being able to see not only when a student is wrong but why they are wrong (which good teachers do all the time). The software was even changed slightly to test teachers instead of children by showing arithmetic errors and asking them what was wrong. The paper has been widely considered by other researchers, including being a formative influence on Feature Based Modelling and a number of other studies of user modelling, cognitive modelling and education.

Subtraction was chosen because it is logical and has a number of likely faults which can be diagnosed relatively easily. Although the work is undoubtedly interesting as an early exercise in User Modelling, it is of limited relevance to spelling correction because of its illogical nature.

Young and O'Shea (1981) argue that when children make mistakes in arithmetic subtraction, they are doing so not necessarily because they simply make mistakes but perhaps because they have forgotten to apply a procedure or because they have wrongly applied one when they should not. The authors contrast their work with that of Brown and Burton (1978) in their development of a superior model of the subtraction process, both for correct calculations and for simulation of the errors made by particular children. In this sense the work is one of user modelling, but it was presented to the world as one of cognitive modelling with benefits to school teaching.

	6	3			7	1
-	4	4		-	5	2
	2	1			1	8

Table 4.1: Two example errors in arithmetic subtraction.

Table 4.1 shows two simple examples of mistakes in subtraction made by 10 year old children. In the right hand column of the first, the child has realised that the 4 is larger than the 3, but instead of 'borrowing' a ten from the column to the left, chose to subtract the smaller digit from the larger. In the second example, the child realised they needed to borrow to handle the subtraction of 2 from 1, but mistakenly turned the 1 into 10 instead of 11. Neither of these errors is coincidental; they can both be observed clearly against a background of 'number fact' errors such as 7-5=3.

Young and O'Shea analysed 1570 subtraction questions from experiments done on 33 children aged 10 by Bennett (1976) looking carefully not only at the answers given, but also at the notes made on the paper such as the borrowing of digits. The intention was to understand the fundamental mistake being made by the child rather than just the external symptoms of it.

In 51 cases between just 3 children, the child had borrowed a digit when inappropriate and not borrowed when they should have. In 50 cases between 6 children the child never borrowed, but instead subtracted the smaller digit from the larger, as in the first example above. In 41 cases the child wrote that $0 - N = N$ for some value of N; in fact the digit 0 caused considerable confusion. For comparison, there were 127 'number fact' errors spread across 25 children. These figures suggest systematic errors rather than simple slips.

The authors then set about devising a production system model for subtraction

to explain the errors as well as the correct performance in a concise and accurate way. They used the OPS2 architecture (Forgy and McDermott 1977) which contains a working memory of information, a production memory of production rules that can be applied, and a conflict resolution method to ensure that only one production rule would be applied on any one occasion.

The working memory would contain observations or internal processing information such as (S EQ M) which would be taken to mean that the subtrahend (lower digit of a column) is equal to the minuend (upper digit). The production system for correct performance is listed below in Table 4.2. This may have rules added or removed to model incorrect performance.

In Table 4.2, the conditions are entries from the working memory. The value $= m$ is a wildcard which will set the temporary variable m to whatever value occupies the next single position in the WM. The actions with stars represent primitive actions which are not represented further here; those without are values to add to the Working Memory for future condition matching, typically representing tasks which need to be performed.

The conflict resolution method is a key part of the distinction between this system and a normal computer program. Instead of demanding that the rules be executed in the order listed, the rule to execute next is chosen according to the contents of the Working Memory. If several rules are eligible, the following method is used in the order listed to choose one.

- No rule is ever activated more than once with its conditions matched to the same set of elements in the working memory.
- Rules whose conditions match more recent elements in the WM override rules which match only older items.
- If rule R1 matches all of the elements that R2 does and also other elements, R1 is a general case of R2 and is preferred.

Tag	Conditions	Actions
FD	$M = m, S = s$	FindDiff, NextColumn
B2A	$S > M$	Borrow
BS1	Borrow	*AddTenToM
BS2	Borrow	*Decrement
CM	$M = m, S = s$	*Compare
IN	ProcessColumn	*ReadMAndS
TS	FindDiff	*TakeAbsDiff
NXT	NextColumn	*ShiftLeft, ProcessColumn
WA	$Result = x$	*Write =x
DONE	NoMore	*Halt
B2C	$S = M$	Result 0, NextColumn
AC	$Resut1 = x$	*Carry, Result =x

Table 4.2: Young and O'Shea's production system for correct subtraction by decomposition (above). The actions with stars are primitive operations which are not explained in more detail here.

- If there is still more than one rule in contention, only the one(s) added most recently are considered.
- If there is still more than one rule to choose from, a random choice is made.

Having established the correct way to perform subtraction, the investigators then went on to examine the errors made by the children to find whether they could describe them systematically and briefly. They came up with a set of 8 more rules, shown in Table 4.3 which notably include copying a number instead of processing it, or writing 0 instead of the correct calculation. They then manually considered each child's work and removed correct production rules and / or added faulty ones to model more closely the mistakes made. Thus one revised production system was produced for each child.

The authors found it difficult to measure the effectiveness of their methods using traditional statistics, and eventually chose to count the number of occasions when

Tag	Conditions	Actions
B2B	$S < M$	Borrow
B1	$M = m, S = s$	Borrow
NZN	$M = m, S0$	Result =m, NextColumn
ZNN	$M0, S = s$	Result =s, NextColumn
NZZ	$M = m, S0$	Result 0, NextColumn
ZNZ	$M0, S = s$	Result 0, NextColumn
SM0	$S > M$	Result 0, NextColumn
NNN	$M = n, S = n$	Result =n, NextColumn

Table 4.3: The modified and additional faulty rules which can used with the main production system.

the production system correctly modelled errors, and then to subtract the number of times when it modelled an error but where the child was correct. The result was a score of 128 for the total of 178 errors made by the children. That is to say, about two thirds of the errors observed were part of an identifiably consistent pattern of behaviour, and not because of slips.

The production rules were devised by hand and applied to each individual child by hand; indeed computers played no part in the work at all, so one might think that it was not user modelling in the normal sense. However, the model built is surprisingly similar to those working on the issue from other directions. It is also a cognitive model in that the authors do claim to understand what is going on in the head of the child while performing subtraction. This is most obvious in the fact that the investigators examined the problem sheets carefully so that where a number of different production mal-rules could be applied, they worked out which one was being used from observation of the crossing-out, carrying, borrowing and so on that was written on the page.

4.3.2 Mal-rules in Language Production

McCoy, Pennington, and Suri (1996) offer a user model based system for the detection of grammatical errors; something which is quite similar to the work presented in this thesis. The research was primarily concerned with the mistakes made by deaf people who were already experts in American Sign Language (ASL), learning to write English. Because ASL has a substantially different grammar from English, such people make a number of mistakes. Furthermore, the mistakes are characteristic of the combination of previous and new language.

For example the error taxonomy developed by studying samples of writing included 'He taught _ directed, for almost 30 years' which omits the word 'and' because it would not be used in ASL, and 'Somehow, I am interesting in ASL' which uses the wrong morpheme of *interest* because 'interested' and 'interesting' are the same word in ASL. Such errors of 'language transfer' were described in a language model using mal-rules which combine with a parser to detect specific errors in the user's writing.

The errors were then combined into a number of lists of features which would, according to linguistic theory, normally be learnt in a certain order. Each of the lists can be thought of as a different dimension, and the user would be at a certain stage in each one.

The system would, in parsing text, look for errors around the user's current level of acquisition of the new language. This offers two advantages. First, if several mal-rules fire for a given error, the one nearest the current threshold of expertise for the user would be chosen. Secondly, errors which are slightly above the user's expertise might be selected for tutorial explanation.

Where insufficient information is available on the expertise of the user, it would

be assumed that they are at a comparable level of expertise in each dimension. However, the details of the model are not explained clearly in the paper because the work was still underway. Issues such as the aging of outdated information, and the danger of interpreting accidental errors as systematic ones in the initial samples (*ie* signal noise) are not addressed.

4.4 Feature Based Modelling

Webb and Kuzmycz (1996) summarise a number of earlier papers, mostly by themselves, on Feature Based Modelling (FBM) with particular regard to student modelling. The ideas are based at least in part on the influential work of Brown and Burton (1978) on arithmetic subtraction, with links to Intelligent Tutoring Systems and reports for human teachers.

Instead of contriving a cognitive model which presumes to know what goes on in the mind of the user, a Feature Based Model considers only directly observable features and consequences, and creates a model of associations between these which can be used to predict behaviour. The features of an FBM are similar to those of production rules, but the meaning and interpretation are rather different.

The features to be observed by the system are aspects of the real world which the system designer thought would be of interest to the task. If too few are observed, the system will be unable to draw useful conclusions but if too many are defined, the software becomes rather slow. Features are divided into two sets: context features and action features, with the intention of producing a limited number of associations $X \rightarrow a$, each of which links a number of context features X to one action a. An example of an association in mathematical subtraction might be $\{$*the subtrahend is zero, the subtrahend to the immediate left is less than the minuend to the immediate left*$\}$ $\rightarrow$ *the result equals the minuend.*

The features are grouped in Feature Choices, where no two members of the same Feature Choice may apply to the same context or action. Also, some features can be generalisations of others, so for example the feature *the result is greater than or equal to the minuend* is a generalisation of the more specific feature *the result equals the minuend.*

In constraining associations that need to be considered, generalisations are used to exclude context features which have already been described by more general ones in the same set. They are used again to ignore the illegal case of a set of context features which includes a feature f and a feature x where f is a sibling of a generalisation of x. These constraints reduce the number of associations that need to be examined when updating the model, and thus reduce the run-time of the algorithm.

Some observations from the real world may be caused by noise instead of being meaningful and correct, so FBM has some constraints on the evidence required before an association is made. For a set of context features C, and an action feature a, a count is made of the number of observations supporting an association, $\#(C \rightarrow a)$ and the number which contradict it, $\#(C \rightarrow\sim a)$. It is required that:

- $\#(C \rightarrow a) \geq$ min_evidence
- $\frac{\#(C \rightarrow a)}{\#(C \rightarrow a)+\#(C \rightarrow \sim a)} \geq$ min_accuracy
- There is no association between a specialisation of C and a sibling of a.

In practice, the authors used min_evidence = 3 and min_accuracy = 0.8 although these values are of course adjustable. The execution of these rules is simple and easy to apply (unlike some other methods), with the outcome of generating a list of associations which are supported by the observations made so far.

To accommodate the change in concepts which a person may undergo, particularly

if they are a student and the FBM relates to the subject they are being taught, the data are aged as more observations are made. Each time an action is observed which can be described by the context C, all tallies for observations relating to $\#C$ are multiplied by 0.9.

To represent the FBM to a human, a selection of interesting associations can be derived. They are found by a series of similarly simple conditions which establish associations which are as general as the current evidence supports. If a situation has not been observed, the generalised associations which cover it may still have been accepted, but should not be presented in to a human because such generalisations may be premature. Whether an association is appropriate or not is a matter for the humans (particularly teachers) to decide.

As well as presenting results to people, the model can be used in a tutoring environment to suggest tasks for tuition or testing. Again using a simple series of conditions, associations are found which can be described as the *most general with insufficient evidence*, or the *most specific with inconsistent evidence*.

A number of FBM systems have been built and tested in a number of domains, particularly arithmetic subtraction and the Prolog programming language. In the Subtraction Modeller experiment, 73 children aged 9 or 10 were given a series of 2 tests, each of which contained 40 subtraction questions. The tests were administered with a gap of one or two weeks. The FBM system operated on individual digits in the calculations rather than entire questions because each digit could effectively be treated as a separate task with different opportunities for errors.

The system was trained on data from the first test for each child, and then was measured on its ability to predict behaviour in the second test. It was able to predict a solution in 80% of cases and was correct in 92% of those. The cases for which no solution was predicted occurred because the system had not established a model

with any confidence. However for the digits in which the children actually made errors in the second test, the system predicted a value in 62% of cases and was correct in only 30% (being 19% of the total). This is clearly well above chance however, as the decimal counting system would suggest an accuracy of 10% by random guessing of solutions for each digit.

The investigators then considered the change in behaviour between tests. Of the errors made in the first test, 49% were corrected on the second test, 26% were changed to a different error and only 25% were repeated. The best that could be expected would be for the 25% to be predicted correctly, and the actual results are remarkably close to that.

To avoid the problem of the children actually becoming better at subtraction between tests, a smaller group was tested twice with a gap of 30 minutes. The system predicted a value for 57% of the digits in which the children made errors in the second test, and was correct in 44% of those cases. These values are noticeably better than the previous test, which must be due to changes in the test conditions rather than the software (which did not change). Overall it predicted values for 84% of digits and was correct in 93% of those.

Feature Based Modelling is of interest generally because it seems to offer reasonably high accuracy in its modelling, and its models can be executed to predict solutions to problems. It is independent of the rather hypothetical nature of cognitive models, and thus is freer than most from the viewpoint that the user takes in considering a problem. It is still necessary for the system designer to choose features to observe, and to find a balance so that the run-time of the system is acceptable; the Unification Modeller which taught the Prolog language took several seconds to update its model after each observation.

An advantage of this kind of model is that it does not require the user to answer

any questions, unlike the Grundy system (Rich 1979), although it can recommend interesting tasks for the user to perform if that is seen as useful.

4.5 Bayesian Networks

One of the most popular user modelling techniques currently being studied is that of Bayesian Networks (BN). Jameson (1996) offers a good review of the field of Bayesian networks as well as some other uncertainty-management techniques.

A Bayesian network operates in a manner somewhat similar to a hand-crafted neural network with a carefully arranged pattern of nodes and connections. Some nodes will relate to observations made of the outside world. Others may relate to actions that the network can stimulate. Finally, the remaining nodes relate to intermediate concepts such as cognitive states or intermediate stages of logical reasoning that would be necessary in a student.

Each node in a Bayesian network has a value associated with it indicating the degree to which the system 'believes'[1] the proposition represented by that node to be true. For the nodes relating to observations this is simply the truth of the observation, for example that the user has performed a certain action. In fact a number of different values can be stored in each node without great complexity. The values in nodes are updated by both upward and downward propagation using Bayes' rule and a table of weights.

In an example which measures the expertise of a user in handling the Unix operating system, there are 4 possible degrees of expertise for the user, three possible degrees of complexity for each command about which the system knows,

[1]The term 'belief' is used here to represent a higher probability (without certainty) that a proposition within a system, about a user, is true. It is not suggested that the system is truly conscious.

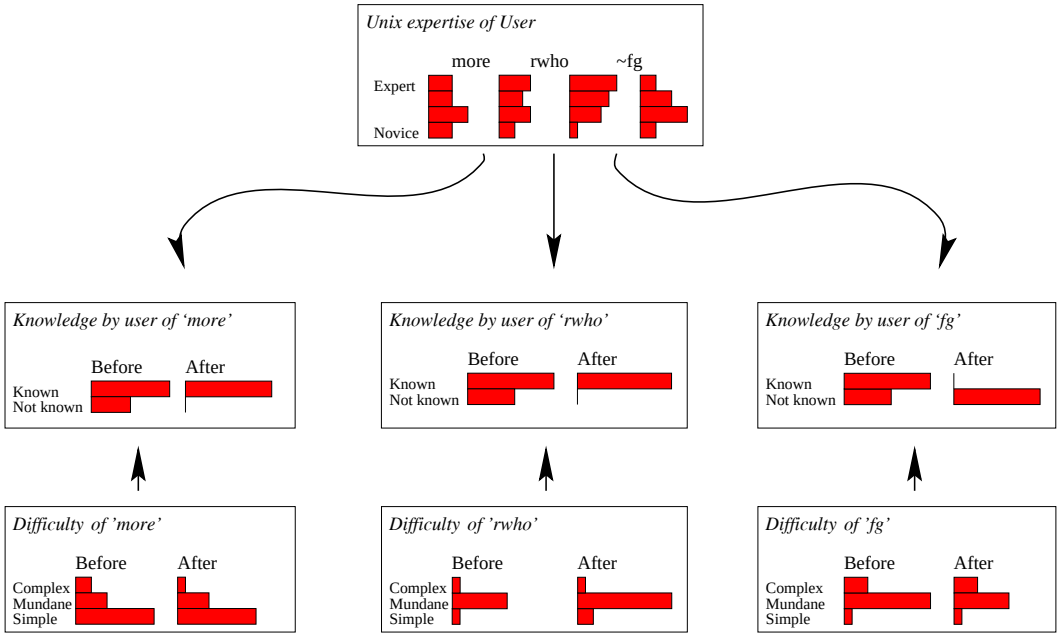

Figure 4.1: A Bayesian Network representing belief in a user's expertise in Unix before and after observing three facts about their knowledge. As the commands `more` and then `rwho` are observed the model's rating of the expertise rises, but later falls when the user is seen not to know about `fg`.

---

and a degree of belief for both whether the user knows a certain command or not. Thus there are 24 values which are combined in the lookup table relating the general user expertise and command complexity to the knowledge of the command.

Figure 4.1 is based on one from Jameson (1992). It shows a simple Bayesian network called IPSOMETER with weights adjusted after the system has observed three commands. In the leftmost of the four expertise models, the system starts by assuming that the user is slightly more likely to be a 2nd level novice. After observing that they do know the command 'more' the expertise belief remains largely unchanged but the believed difficulty of the command is reduced. The system is later impressed by the user's knowledge of 'rwho' which it thought was mundane; the user's believed expertise is raised, and the difficult of 'rwho' reduced.

Finally, the user is found not to know the command ‘fg’ and so their expertise is reduced and the difficulty of that command increased somewhat. There is no doubt that the user does know the command ‘more’, so the belief for that node is entirely within the ‘knows’ part after the fact has been observed.

Bayesian networks have been used with some effect in a number of user modelling environments. They are capable both of evaluating the uncertainty in information which contradicts other evidence, and of making predictions where no information is available (for example whether the user knows the command ‘fg’, when the system has only observed them using ‘more’).

However, the time taken to propagate values in a Bayesian network can be substantial, depending on the design of the network. At worst it is NP-complete. In practice, substantial effort in optimising performance will be rewarded with acceptable run-times. On a vaguely related note, the effort involved in developing a BN can be substantial, although some programming libraries exist which may make the task easier.

A serious concern relating to the use of Bayesian networks for the work in this thesis is the initialisation. The design of the network topology normally derives from a cognitive or process model, typically being one which encompasses all imaginable approaches to the user’s task. The initial node values often arise from the designer’s preconceptions of the task, and the tables for adjustment weighting may be derived from psychological theories or hand-tuned to improve performance of the modelling task in hand. Each of these would require justification for the design, especially where only small amounts of data will be available to update the beliefs (if thousands of observations were available, the initial assumptions would become much less important). The network connections used are also not adjustable at run-time, and so are sensitive to the preconceptions of the designer.

There is no record in a Bayesian network for information which is unknown as compared to information of a known degree of uncertainty. If, for example, a network were to represent the probability with which a person could successfully perform a task, that might indicate that they have a 50% probability of succeeding, or the same data might ambiguously indicate that the system has no idea how likely the person is to succeed.

Confidence in Bayesian systems is also diminished by the lack of psychological support for their reasoning process in humans. When given tasks which require Bayesian reasoning, people do badly compared to using other forms of reasoning. This is somewhat contradicted by the Vista project (Horvitz and Barry 1995) in which a Bayesian Network was used to second-guess humans in the control of a rocket engine. The removal of connections in the network caused the same kind of errors that novice NASA staff made, suggesting some cognitive similarity.

4.6 Dempster-Shafer and Fuzzy Logic

The Dempster-Shafer Theory of Evidence (DST) was initially presented by Shafer and Tversky (1985). It is much better at handling degrees of confidence in beliefs as well as degrees of belief themselves. It is intended mainly as a record of evidence which has been obtained, and is not good for making predictions about concepts which have not been observed.

The designer constructs a tree of concepts, with the root being the most general possible belief. In the example of Unix expertise, the issue being addressed is what degree of expertise the user has with Unix in general. The possibilities are levels 1,2,3 or 4 (from Novice to Expert). A subset of those is the set of consecutive expertise levels 2,3,4 and a subset of that is 2,3, etc. The system may believe, then, that a user has an expertise of 2 or 3 but it may have no information as to which of

those levels the user is at. DST functions well under such circumstances.

For each node in a DST tree, there are three numerical variables. The first is the *belief mass*. That is a fairly empirical measure of the information known about that node; it is not inherited up or down the tree automatically. All belief masses in the system add to 1.0. The second is the *total belief*. This is the 'output' measure of the degree to which the system believes the concept is true, and is the sum of the belief masses for the current node and all its child (more specific) nodes. The final one is the *plausibility*, which indicates the degree to which the node's concept is compatible with the general beliefs.

Because in DST, evidence is measured as a proportion of the whole body of information collected, that collected first will affect the belief of the system entirely. If it is noisy, random or short term information, this may be inappropriate. For example Bauer (1995) considers the actions of users of an email system, and considers users to have both high-level and low-level goals in their actions. To prevent the first few from biasing these beliefs, the system forges a log of one 'null' session in which the system made no inferences about the user's actions. When the first actual session is recorded, it will have only half the weight it would otherwise have done. After 100 real sessions have been observed, the null session will account for only 1% of doubt.

The Dempster-Shafer theory does not make any prior assumptions about the user, unlike Bayesian networks. By the same token, it cannot make any statements about them until data has been collected on the subject in question. It cannot be used to predict behaviour of the user in the future, except in terms of repeating what has already been done.

The great strength of DST, then, is that it can evaluate possibly conflicting evidence and state both a degree of belief for the truth of a concept, and a degree of

confidence in that belief. Shafer and Tversky (1985) used it for the evaluation of evidence from unreliable witnesses.

Constructing the error patterns of spelling errors into a strict hierarchy as DST requires would be difficult in some cases. Also, there is not necessarily just one error pattern being made; an individual may have several faults in their spelling, or may have only one. These problems may well be surmountable in future work, but DST was not used in Babel.

4.7 Fuzzy Logic

Derived from the world of Machine Learning, Fuzzy Logic has found a small but comfortable home in User Modelling where degrees of confidence and knowledge can be stated and used in vague terms. People typically speak and indeed think in indefinite terms like “This student is quite advanced, so he ought to be able to handle this task fairly well.” The same kind of ambiguity can be represented in Fuzzy Logic, with a sort of boolean IF - THEN statement where it is not necessary for all of the conditions to be met, but merely requires most of them.

KNOME (Chin 1989) uses nine different degrees of truth where simple boolean logic has two. If at any point an extreme degree is reached, then the concept is accepted and all contradictory ones are rejected. However, while the values are intermediate, a number of different ones can be held. The logical statements require vague preconditions for the IF part, and instead of setting exact values as output, they increase or decrease the confidence in them. Similarly the operators like AND and OR have new meanings. These vary somewhat between implementations, for example OR has been interpreted as the most common value from the inputs and AND as the arithmetic mean of the inputs.

An interesting advantage of Fuzzy Logic systems is that they can reduce the number of rules in the updating tables by exploiting the ambiguity (Zadeh 1994). For example an intermediate user is to some degree an expert, so propositions which refer to experts will to some degree affect intermediate users.

Unlike DST, Fuzzy Logic has no representation of a degree of confidence in a value, and default values are needed to initialise the system. Even more important than the default values are the rules which manipulate them. These are typically decided by the designer and are not adjusted automatically; they may be fine-tuned by the designer to improve performance but this begs the same question as Bayesian Networks: How can the designer demonstrate that the system design is sound rather than merely convenient? One answer is that Fuzzy Logic systems were originally designed to allow vague definitions of the rules, and a domain expert would find it easy to provide the rules in such terms.

The user modelling system in Babel is more similar to Fuzzy Logic than to any of the other methods described here. Babel employs a number of propositions representing error types which may be made, and considers them in combination or separately as far as is necessary to reach a number of correctly spelt words in its dictionary. The preference given to the rules is most closely aligned with that of continuously variable probabilities. The general operation depends very much on the details of the spelling error, the number of similar words in the dictionary, and the similarity of rule probabilities.

4.8 Summary

Rich's work is relevant to that presented here in a number of respects. Firstly, it employs a relatively simple way of updating its model based on new evidence; a similar thing has been done in *Babel* which will be discussed later. It also attempts to

predict the nature of the user based on its user model, and then gathers information on its success after it has made a recommendation.

It is different from the work here, however, in that it has the liberty of asking questions to clarify inconsistencies in the model. The *Babel* spelling checker cannot propose things to the user because it is intended only to correct errors in words that the user intended to type. Perhaps an extension of Babel would be to set some sort of spelling test but that would make it a very different beast from the prosthetic approach taken so far.

The shortcomings of Babel could perhaps be accommodated by further use of stereotypes in predicting spelling error patterns which have not been attempted yet. The first major version of it did not contain such a method.

Fuzzy Logic contains some intrinsic appeal to this work because of its imprecision. Given a number of weightings for its propositions, it evaluates them rather vaguely relative to each other. This is somewhat similar to the method in Babel in which the choice of error paths considered depends very much on the differences between rules, the number of eligible rules and the variety of words from the dictionary which are similar.

The idea of implementing mal-rules, as incorrect behaviour patterns which have been learnt and are applied systematically is one which has been used further in the research in this thesis, although no such strong claim on cognitive accuracy can be claimed here, partly because the process of constructing the spelling of a word is not generally made available to a reader, and is far more complicated than the process of arithmetic subtraction.

The user modelling of spelling correction is very different from that of more popular domains such as subtraction because of the complexity of correct performance in the domain, and because of the diversity of error sources. Although it is possible to

state the correct rules of subtraction in a few lines (Table 4.2), a model of comparable accuracy for spelling would take a document the size of a dictionary at least.

The first version of Babel has not used a system such as Feature Based Modelling to help derive generalisations of errors or to capture the consistencies compared to the noise. It is difficult to convert the language of context and action features into the behaviour of a spelling corrector, although this may be possible. There are few operations which are true generalisations, for example. It is true that substituting one letter for another is a generalisation of substituting one vowel for another, but this has already been accommodated. The most obvious FBM architectures would not naturally assume that there was any special link between $b \leftarrow d$ and $p \leftarrow b$ without also assuming a relationship between $m \leftarrow w$ or $n \leftarrow u$ although for many dyslexic writers there is.

Although the Dempster Shafer theory of evidence has also not been used in the spelling correction work here, it may be relevant to future implementations. Where error types can be grouped, for example the mirroring of letters like b-d, p-q, p-b, a hierarchy might be constructed and applied usefully. Interestingly, the results of running Babel have shown that even such apparently similar rules do not co-occur as much as would be expected, so producing a reliable hierarchy would be very difficult if not impossible. Each observation for a DST tree would be only at the level of a terminal node and so the problem of predicting related actions would remain; DST does not propagate knowledge to other nodes as Bayesian Networks do.

Because *Babel* needs to kick-start itself with a minimum of data, a system of stereotypes would seem a good choice to begin with. The direct questioning of Grundy is not possible in a spelling correction environment, and the nature of its inferences are also likely to be unsuitable in this case.

In conclusion, there are many differences between the methods proposed in other user modelling research and the requirements of a task for spelling correction. None of the existing methods was appropriate, nor have they been used directly.

The method used for Babel was a combination of existing ones. It is similar to the early methods of probability of belief in each concept, and also arranges its concepts according to their generality (scope) so as to identify the most specific relevant concept in each case. This is most similar to Feature Based Modelling. The Path Choice Heuristic (on page 124), in which a minimal rule-set to describe the observations is found, is unusual in User Modelling.

Chapter 5

System Design of BABEL

5.1 Introduction

The main research in this thesis is based on a spelling checker called Babel which was designed with the intention of working better for dyslexic users than traditional spelling checkers. Misspelt words from dyslexic and non-dyslexic writers were fed in as input and the suggested corrections compared to the hand-made corrections for the text samples.

Babel consists of a number of modular units, an overview of which is shown in Figure 5.1. The user provides text as a single word permutation which is then modified by the error rules to produce a number of other permutations, with the error rules being applied in an order determined by the user model. Those permutations which are real words in a dictionary are presented to the user, and the user's choice is used to update the user model.

The user model of the system is a key feature. It holds a list of costs for each of a number of error classes that the system can detect, and is driven primarily by the choices made by the user from previous lists of suggestions. After each input-

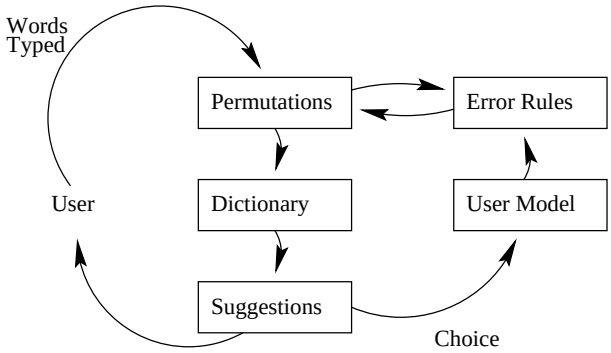

Figure 5.1: An overview of the modular interconnection of 'Babel' units.

---

suggestion-choice cycle the user model becomes slightly more finely attuned to the errors of the user in question.

The spelling correction engine works in a novel way, because of the nature of the user modelling system. It operates by taking a string of letters, either directly from the user or from earlier instances of the operation, and applying one of the mal-rules to create a new letter string. If the new string is in a dictionary of correct words then the word is added to a suggestion list. If not, the permutation is kept for possible further changes. After each change, a cost is added to a running total for the permutation according to the frequency with which the mal-rules applied have been found to be useful.

After a list of suggestions has been presented to the user, one is selected. The rules used in the construction of the permutation are updated to recognise their usefulness, and the remaining rules are weakened slightly. This chapter describes the design of Babel, which is then combined with the sample text in Chapter 6 and evaluated in Chapter 7.

5.1.1 Input and Output

The software can run in two modes. In the first, called a directed search, a pair of words are given to the system which will attempt to find a sequence of error rule transformations to change the first spelling (the error) into the second (the correction). The second mode, that of normal operation, involves a probabilistic search from a given error word to a list of suggestions using the user model. This list of correct dictionary words is then presented to the user for them to choose from.

In an experimental research project it is necessary to collect performance statistics, although they might not be used in a real-life spelling correction task. The list of suggestions is compared to the correct spelling, and a counter is incremented according to where on the list the correct answer stood.

The input to the program in both cases is in the form of a list of pairs of words. The first is the word as written and the second is the correction for it. If running in the 'normal' mode the second word will not be used except for the statistics at the end of each search. This means that Babel operates in a 'batch processing' mode rather than interactively. It was done in this way to aid development, and could easily be adjusted to work interactively if necessary.

The output is a sequence of lists of suggested words, and in the file saved at the end containing the updated user model information. This can then be analysed by other smaller programs which compare models between users, count frequency of letter positions within words and so on. The user model information is stored persistently between sessions, and developed gradually for each user throughout their use of the system.

The input is filtered through another program which takes a written sample

transcription. It checks it for some prerequisites such as not having been corrected already and then normalises the format to the list of error word pairs.

The description below explains how the user model of the Babel spelling checker works. The model could be used in a number of application areas by changing the rules used but without changing the user modelling principles. It simply requires an input permutation string to be presented which may be incorrect, and a dictionary of correct strings or some other method of assessing the correctness of a string. It will apply a set of rules to transform the permutation, producing an ordered list of alternatives from the dictionary, one of which should be chosen by the user.

5.2 Rules

Production rules are a familiar concept (see for example Young and O'Shea (1981) and Brown and Burton (1978)). However, the rules for English spelling are complex. The approach taken here has been to model only the mal-rules (production rules which give the wrong answer) in a manner somewhat similar to Young and O'Shea, except that their system included correct rules as well as mal-rules.

McCoy, Pennington, and Suri (1996) suggested a set of linguistic mal-rules for detecting grammatical errors in English written by people whose first language was American Sign Language. In similar style here, the rules are based on simple letter errors; widely agreed spelling construction rules; and rewriting rules which build long words from substrings or root words.

The transition rules are shown in Tables 5.1 to 5.5. They are a set of routines which describe every possible error made by the user. They include very general ones such as deletion and insertion (in Table 5.4) which together can describe all possible

errors, but also include more specific ones such as the substitution of a 'B' for a 'D' (in Table 5.1).

The frequency terms used in the tables are intentionally vague but are based on observations from the sample text in Chapter 6. In order they are; Very Common > Common > Less Common > Fairly Rare > Rare > Very Rare > Extremely Rare > Not observed.

The rules are largely derived from a cognitive model. In the case of this spelling checker, the model is that of Patterson and Shewell (1987) which is illustrated in Figure 3.1 on page 72. As far as possible the rules should represent fundamental errors which are expected rather than mere string transformations which have been observed, although a comprehensive set of rules for spelling is practically impossible to create. The derivation of the rules is not important to the user model, however. If, once executed, a rule is found to be used many times it may be possible for the developer to sub-divide it to allow the user model to gain a more detailed image of the user. This evolutionary procedure is not haphazard but systematically experimental[1].

The Letter Substitution rules in Table 5.1 generally fall into two categories. Rules such as CforK, SforC and ZforS are phonetic errors which map single letters to other single letters. The majority of the rules in that table are visual correspondences of letter pairs which look similar and so are often confused. These correspond to the Phoneme to Orthography Conversion stage or the Writing Buffer stage of the dual route language model in Figure 3.1.

The Bigraph errors in Table 5.2 are all logical errors in the generation of correct spellings where a person forgets that the two letters are required as a pair. They might also be seen as a deletion (or an insertion in the case of WHforW) but such

[1]Mitchell and Welty (1988) observed that Computer Science has very little experimental work, and suggested that more should be conducted with a view to assessing the quality of new ideas.

Letter Substitutions		
Preconditions: No previous letter substitutions at this character position		
Name	**Description**	**Comments**
CforK	Wrote C; meant K	A fairly rare substitution. Phonetic letter substitution in Phon→Orth conversion, or mistake in orthographic output lexicon word construction.
KforC	Wrote K for C	Slightly more common than CforK. In Phon→Orth conversion
CforS	Wrote C for S	Fairly rare. In Phon→Orth conversion
SforC	Wrote S for C	More common. In Phon→Orth conversion
BforD	Wrote b for d	Typical dyslexic error. In Phon→Orth conversion, Output Buffer reflection, or Orthographic Analysis.
DforB	Wrote d for b	Typical dyslexic error but rarer than BforD. Similar to BforD.
PforQ	Wrote p for q	Not observed in sample. Output Buffer reflection or Orthographic Analysis reflection. Similar to BforD.
QforP	Wrote q for p	Extremely rare. Similar to PforQ.
MforW	Wrote m for w	Not observed in sample. Vertical reflection, similar to BforD.
WforM	Wrote w for m	Rare. Vertical confusion, similar to MforW.
NforU	Wrote n for u	Fairly rare. Similar to MforW.
UforN	Wrote u for n	Rare. Similar to NforU.
GforC	Wrote G for C	Fairly rare. Output Buffer or Orthographic Analysis slip. Possibly somewhat phonetic.
CforG	Wrote C for G	Rarer than but similar to GforC.
GforQ	Wrote G for Q	Never observed in sample. Orthographic Analysis slip.
QforG	Wrote Q for G	Never observed in sample. Similar to GforQ.
SforZ	Wrote s for z	Never observed in sample. Phonetic conversion or Output Buffer error.
ZforS	Wrote z for s	Fairly rare. Similar to SforZ.

Table 5.1: Letter substitution rules in Babel

Bigraph Transformations		
Preconditions: No bigraph transformations or letter substitutions at this character position		
Name	**Error**	**Frequency / Cause**
PforPH	Bigraph H deletion	Very rare. Output buffer omission.
CforCH	Bigraph H deletion	CH occurs more often in English. Output buffer omission.
SforSH	Bigraph H deletion.	Omission or phonetic→orthographic substitution.
GforGH	Bigraph H deletion	More common than expected. Non-phonetic omission.
TforTH	Bigraph H deletion	Omission, phonetic→orthographic substitution, or rewriting error.
WforWH	Bigraph H deletion	More common. Omission, phonetic→orthographic substitution, or rewriting error.
WH-forW	Bigraph H inserted	Even more common. Insertion, phonetic substitution, or rewriting error.
CforCK	Bigraph K deletion	Rare. C and K sound the same. Omission, phonetic error, or rewriting error.
KforCK	Bigraph C deletion	Fairly rare. As above.
CKforK	Bigraph C inserted	Insertion or phonetic error. As above.
QforQU	Bigraph U deletion	Not observed. Q should always be followed by U. Output buffer omission or severe orthographic lexicon fault.
SforSC	Bigraph C deletion	Fairly rare. Omission or rewriting error.

Table 5.2: Bigraph letter rules in Babel

Miscellaneous Insertions/Substitutions/Deletions			
Name	**Error**	**Precondition**	**Frequency / Cause**
Cons2	Deleted one of a doubled consonant	Position at least 2 characters from word end. Neither 2Cons nor Cons2 applied before.	Commonly used for many letters. Rewriting error in suffix addition.
2Cons	Doubled consonant	As above	Common. Rewriting error in suffix removal.
-finE	Omitted final E	+finE not applied	Common. Rewriting error sometimes involving suffices.
+finE	Added a final E	-finE not applied	Not as common. As above.
2Vow	Added unnecessary vowel	No bigraph or letter substitutions at this position	Common. Phonetic→Orthographic conversion error.
Vowel2	Deleted vowel adjacent to another vowel	No bigraph or letter substitutions at this position	More common. Phonetic→Orthographic conversion error.

Table 5.3: Wildcard rules in Babel

General letter operations		
Preconditions: Same rule not previously applied anywhere. This application must not affect a Cons2, 2Cons, +finE or -finE operation.		
Name	**Error**	**Comments**
Ins	Inserted a letter	Common. Buffer error.
Del	Deleted letter	Very common. Buffer error.
Subst	Substituted letter	Very common. Buffer error.
SubVow	Substituted vowel	Common. Buffer or phonetic conversion error.
Trans2	Transposed 2 letters	Common. Buffer error.

Table 5.4: General letter rules in Babel

errors are rarely simple slips. These would typically be caused in the Orthographic Output Lexicon of Figure 3.1.

The Miscellaneous errors in Table 5.3 are perhaps the most interesting rules. Most of them represent a number of logical errors in word production which are neither truly phonetic nor orthographic, but are related to the assembly of word parts. They can be attributed to various parts of the language model including the Phoneme to Orthography Conversion for Vowel2, 2Vow, +finE and -finE, and to the Writing Buffer for Cons2 and 2Cons.

The General letter operations in Table 5.4 are common wildcard operations which normally occur in the Writing Buffer of the language model.

Finally, the Phonetic errors of Table 5.5 occur in the Phoneme to Orthography Conversion stage of the language model, as well as in the other phonetic stages such as Sub-Word Level Orthographic to Phonological Conversion if reading or listening is involved such.

The phonetic rules are applied by converting each new word submitted for

Phonetic operations			
Name	**Error**	**Preconditions**	**Comments**
Phon	Phonetic pre-processing	No rules already applied	Uses speech synthesiser module with many vowels
phSubV	Substituted phonetic vowel	Phon applied	Common. Sometimes merges vowel letters usefully. Phonetic conversion error or possibly buffer error.
phSubst	Substituted any phoneme	Phon. Not phSubst	For phonemes which are not single letters. Occasional. Phonetic conversion sequence error or phonetic output lexicon error.
phTrans	Transposed phonemes	Phon. No transposi-tions.	For phonemes which are not single letters. Very rare. As above.
phonIns	Inserted phoneme	Phon. Not phonIns	Fairly rare. Phonetic conversion or lexicon error.
phonDel	Deleted phoneme	Phon. Not phonDel	Occasional. As above.

Table 5.5: General Phonetic rules in Babel

checking into a phonetic permutation, and then considering that along-side the other permutations in the system. Only the phonetic rules can be applied to the phonetic permutations because the character set they use is not the normal alphabet, but is instead one of phonemes. Each correctly spelt word in the dictionary is stored both othographically and phonetically, and each new phonetic permutation generated is compared to the phonetic entries in the dictionary. The phonetic conversion is performed according to the method described in McIlroy (1974) and works only on whole words.

Each rule is implemented in a routine that takes a spelling permutation, performs a number of preliminary tests and produces 0 or more new permutations in addition to the one presented as input. The routine will be called by the user model if required.

Each permutation is a data structure which contains the current letter sequence of the word it describes. This may or may not be a valid word from the dictionary as it may yet require further rule applications. It also includes administrative information about which rules have been used so far and the positions of letters in the sequence compared to their original positions. A set of flags indicates if the word is known to be in the dictionary and other such information. Real number variables hold the total cost (edit distance) of the transformation rules which have been applied so far, *i.e.* the sum of the costs of those rules. A bit pattern indicates the classes of rules which are eligible to be applied next. This bit pattern is modified by the rules when they are applied in accordance with the cognitive language model.

Rules will typically exist in pairs, a positive and negative form, although there is no obligation for this. This is because, when a person is composing the spelling of a word, they will apply spelling rules which may or may not be appropriate; they can apply the actual rules when not appropriate or not apply them when they would have been correct (such as Cons2 and 2Cons), as well as the two cases of correct

usage. There may also be some non-rules; patterns which transform spellings but should never be used as they never construct valid words, such as Ins and Del.

Rules can be inhibited by previous rule applications according to paths in the cognitive model. In the composition of the spelling within the writer's mind, the cognitive model describes certain stages which are passed in sequence. If an error transformation rule from one stage is applied to a certain permutation by the software, the location of the permutation in the cognitive model is defined (although not always precisely), limiting the variety of other transformation rules which can then be applied.

Rules can be inhibited by previous rule applications to avoid cycles. In particular, those rules which exist in both positive and negative forms could be applied alternately forever, wasting computer resources without producing worthwhile or even valid results. Thus each rule routine performs a number of preliminary tests both to the form of the permutation it is to act on (the consonant doubler will not work at the start or end of words; BforD can only be applied if there is a 'b' in the permutation) and to the record of rules which have previously been applied to the string; if there is a 'c' but it was produced by a KforC operation, it cannot subsequently be converted to 's'.

The search for a correction will be stopped in any one of the following cases: If so many permutations have been tried that the system has reached a limit on memory consumption or time spent, if several real words have been found in the dictionary which are plausible corrections for the written word, if all permutations have more than a certain number of rules in their correction path or if all the rules have been applied in all possible combinations, leaving no possible transformation paths. The limit on the path length was set at 4, based on observed solutions which are shown in Figure 7.6 on Page 172.

Where characters are inserted into the permutation, for example by Del, Babel will insert a wildcard symbol instead of any particular character or set of characters. In searching the dictionary for this, the original method used was to find upper and lower bounds in the alphabetically sorted dictionary at which matches for the characters preceding the first wildcard existed, and then to search linearly through the remainder.

Later, it became clear that Pain (1985) had used a tree to search for letter substitutions more efficiently and in parallel. The same method was introduced into Babel resulting in a substantial increase in speed (two orders of magnitude) and ultimately leading to the development of ALGORITHM T which is presented in Chapter 8.

5.2.1 Generality score

Some of the rules which can be applied to transform words are subsets or supersets of others. The more specific a rule is the more information can be learnt from its application, so it is important to use the least general rule possible on any given occasion. To enforce this, a system of 'generality' is given precedence over the normal path edit costs.

Features to be captured by the user model which are general cases of others are not a new concept, although most user models based on hand-designed cognitive architectures generally do not allow overlap between cases. Feature Based Modelling (Webb and Kuzmycz 1996) notes which features are general cases of others and has prepared rules preventing their combination at run-time. Finlay (1990) uses automatic classification processes which do not lead to general rules. The A$\star$ search method (Lirov 1991) deletes paths which reach the same node, and so never observes more than one path describing the same error.

The system proposed here, unlike existing systems, allows the summation and comparison of a sequence of transformations on a string; it will select the transformation path with the lowest total number of general operations rather than operating on individual rules. It may be that one general operation better describes an error than two specific ones, in which case the general operation should be used; on this basis the generality of whole paths is considered more important that that of individual steps.

Babel produces as output a list of transformation paths which use the transformation rules to convert written to correct word spellings. In some cases several paths will exist because several different combinations of rules can achieve the same final spelling. In those cases which employ specific rules, other permutations will exist which employ more general rules (for example where a *b* has been changed to a *d* in one permutation, another will exist which uses a generic substitution).

Although both may be correct, the path employing the more specific rules is of more interest because it more usefully characterises the errors made by the writer using the system. A method is needed to exclude the general path if possible, but to retain it if there is no more specific case.

To achieve this, the measure of generality is applied as a pre-filter to the possible choices before a set of rules are used. That is to say, at the stage in the system where a correction has been selected and more than one transformation path exists which changes the written word to the chosen correction, some of those paths may be removed because they are too general before the user model is updated.

Those rules which are general cases of others have higher generality scores. Those rules which are unrelated to each other have no predetermined relationship in their generality scores.

The generality values for all rules in each correction route will be added and

the minimum sum found. For any given correction (a certain written/correct word pair) all of the transformation paths will be deleted except those with the minimum generality sum. The remaining paths are those employing the most specific applicable rules.

It is important to take care when assigning generality values (this process is done at the time of creating the rule set). As few different values are used as possible because only correction paths with exactly the same summed score are comparable. Unlike the sum of the rule costs which exist on a continuous scale, the only transformation paths used are those with exactly the minimum generality sums. It is of course also important to ensure that rules which cover the same transformation have different values according to the variety of cases which the rules cover.

Table 7.4 on page 173 is listed according to the Generality values which were assigned to each rule. The rules on the left have a value of 0, those in the middle a value of 2 and those on the right, a value of 3.

5.3 Cost adjustment

The key to the user model is the cost of applying each rule. When a rule is applied to a spelling permutation its cost is added to those already used; the final permutation therefore has a cost equal to the sum of the rules used in constructing it. The costs are adjusted for each person to give lower costs for the errors they have been found to make.

In the very first case, all rules have equal costs. This is unrealistic as a spelling correction model though, suggesting for instance that *G* is mistaken for *Q* as often as a letter is missed out. Users will not see the system until the model has been given some initial training on samples of errors.

Once the system has been offered to a new user it is important to model their errors as quickly as possible. This is proposed in two stages; a short period of training followed by ongoing development. In the training stage a proficient user helps the new person to learn how to use the system and ensures that the correct transformations are chosen from the list of suggestions (*ie,* that the user does not select the wrong word from the list of suggestions which would reinforce the wrong set of rules). In the remaining time the suggestions chosen by the user refine the user model gradually.

Words are offered to the system as pairs of written and intended words. In most cases the intended word can be chosen from the list of suggestions but the expert user should be sure which word is required without the help of the system, initially. In real-world use, a log might be kept of all actions so that the expert could adjust the model 'off-line' rather than sitting with the user continuously. Because Babel is not used in the real world, such issues need not be taken further.

Babel will attempt to find corrections for the written spelling using the transformation rules. These will initially be weighted towards the rules used by the population in general (as seen by the system before personalisation for this user). The correct spelling is not used at this point.

The spelling permutations are compared with a dictionary of known words; those which are not real words are excluded from the list of suggestions (most permutations will not be real words). The set of known words are sorted according to the sum of their rule costs; the cheapest (most likely) first. Those words which have been found by more than one transformation path are kept intact but only presented on the list for the user once, positioned according to the lowest cost instance.

The user (or expert) will then be offered the list of real word suggestions and

asked to select one or type the correct spelling. The implementation may store all suggestions and permutations on disk for later correction by the expert instead of requiring the expert to be present whenever the new user interacts with the system.

If the correct spelling was not on the list of suggestions, the special genetic search (described below) is used. If the correct word was on the list of suggestions, it is pre-filtered using the generality scores and recorded for future analysis. For each rule in each permutation the generality values are added. The lowest sum of values is found and all permutations with higher sum path costs are excluded. This is described in more detail in Section 5.2.1 above.

The process of correcting words is repeated for a substantial number of word errors from the same user, after which the accumulated transformation paths are analysed.

For each correction word pair (written/correct) a number of paths may exist. However, only one sequence of rules is required to account for an error; the other paths are not of interest. A heuristic described in Section 5.4 below will be used to find the smallest set of rules used in the largest number of correction pairs.

The final rule set and costs should be proportional to the usage of the rules; the rules used most commonly will have the lowest costs and will therefore be used in preference when generating suggestions.

For those rules which have been used very rarely, the number of occurrences may not be representative of future errors. The number required to trust the rule costings could be a constant, and the value 3 was suggested by Webb and Kuzmycz (1996) but there is ongoing debate about how to appropriately ignore background noise in such data.

5.4 Path choice heuristic

The algorithm presented in this section addresses the issue of choosing a path that best describes the errors probably made, by comparison to the task of most search algorithms which ignore the path and are concerned with the solution only, or continuously prune the set of solutions while searching. This algorithm is small and simple, and is novel as far as the author is aware.

After a number of words have been corrected, there will be a set of string pairs (written, correct) and one or more different transformation paths employing one or more rules to achieve each correction. The task here is to select the most appropriate single transformation path for each correction, and to lower the rule costs used in that path. The other possible paths not selected are of no further interest. On input to the algorithm the list of pairs should be sorted so that all paths with the same pair values are consecutive. Also provided is an array of the transformation rules and a count of the number of valid paths.

For each word pair, the number of rules used in the transformation paths are counted. Any path with more than the minimum number of rule applications is discarded on the grounds that spelling errors are most likely to be simple (and it is the responsibility of the rule set designer to capture the errors with sufficient descriptive power).

This will probably leave more than one transformation path for many word pairs; these must be reduced to exactly one in all cases. The following refinement procedure should be applied repeatedly until only one path remains for each word pair.

An integer array should be initialised to 0, with one entry for each valid transformation rule. (A valid rule is one which has not been selected as the most common

in a previous pass of this loop).

For each word pair the following should be done: an empty set should be created. For each valid rule used in any transformation path for the given word pair, the rule should be united with the set. Naturally, any rule used in several paths will exist in the set only once. Once each path for a given word pair has been considered, the integer array should be incremented for those rules which appear in the set.

After considering every word pair, the integer array will contain the number of word corrections in which each rule plays any part.

The most used rule should be selected. If several rules are equally common, the least general one should be selected (*ie*, considering its generality value). If several still remain, one should be chosen at random.

Each word pair is again scanned. If the selected rule exists in any transformation path for that word pair, that path is kept and other paths (not using the rule) are deleted from the path set. This may still retain more than one transformation path per word pair. The rule selected as the most common should be invalidated for further passes, and the above procedure repeated until only one path remains for each pair.

After each word pair has been reduced to having only one solution path, each rule used should be strengthened in proportion to the number of corrections it appears in by lowering its cost. This can be done in a number of ways, and in Babel is done by setting the rule cost to an exponential function of the number of uses.

5.4.1 Pseudo Code

The algorithm for the Path Choice Heuristic, which has been described above, is presented below in Pascal. The listing contains only the crucial code and is not a

complete program.

```
CONST maxrules = 50;
      maxpaths = 1000;

TYPE
  ruleflagtype = SET OF (processed,used);
  pathflagtype = SET OF (deleted);

  string =
    packed array[1..64] of char;

  correction_path =
    RECORD
      source : string;
      target : string;
      nrules : integer;
      cost : integer;
      flags : pathflagtype;
      rule : array[1..maxrules] of integer;
    END;

  rule =
    RECORD
      name : string;
      flags : ruleflagtype;
      cost : integer;
    END;

  correction_array =
    array[1..maxpaths] of correction_path;
  rule_array =
    array[1..maxrules] of rule;

PROCEDURE path_choice(var rules: rule_array ;
  var corrections: correction_array ;
  npaths: integer) ;
  { Parameters:
      rules is an array of rules to be updated and
        returned.
      corrections is an array of spelling correction
        paths, sorted in order of the source and then
        target strings so all possible paths for the
        same strings are adjacent.
      npaths is a count of the number of valid items
      in 'corrections'.
```

```
  }

VAR
  source,target : string;
  p1,p2,r,minref,maxuses,mostused : integer;
  anyrulesleft,usedhere,usedthisgroup : boolean;
  rulecount : array [1..maxrules] of integer;

BEGIN
  { first mark all paths as undeleted and
    add dummy trailing record to allow loops with
    conditions inside to run their finishing condition }
  corrections[npaths+1].source := '';
  corrections[npaths+1].target := '';
  FOR p1 := 1 TO (npaths+1) DO
    corrections[p1].flags := [] ;
  FOR r := 1 TO maxrules DO
    BEGIN
      rules[r].flags := [];
    END;
  source := '';
  target := '';
  minref := 1;  {index of the first path for this pair }
  minused := 0; {minimal path length }
  {remove paths longer than shortest for this pair }
  FOR p1 := 1 TO (npaths+1) DO
  BEGIN
    IF NOT ((source = corrections[p1].source)
      AND (target = corrections[p1].target)) THEN
    BEGIN
      minused := corrections[minref].nrules;
      FOR p2 := minref TO (p1-1) DO
        IF minused > corrections[p2].nrules
        THEN minused := corrections[p2].nrules;
      FOR p2 := minref TO (p1-1) DO
        IF corrections[p2].nrules > minused THEN
          corrections[p2].flags :=
            corrections[p2].flags + [deleted];
      source := corrections[p1].source;
      target := corrections[p1].target;
      minref := p1;
    END
  END;

  anyrulesleft := true;
  WHILE anyrulesleft DO
  BEGIN
    source := '';
```

```
target := '';
FOR r:= 1 TO maxrules DO
BEGIN
  rules[r].flags := rules[r].flags - [processed,used];
  rulecount[r] := 0;
END;

FOR p1:= 1 TO (npaths+1) DO
BEGIN
  IF NOT ( deleted IN corrections[p1].flags ) THEN
  BEGIN
    IF NOT ((source = corrections[p1].source)
        AND (target = corrections[p1].target)) THEN
    { finishing code concerning all paths for previous pair }
    BEGIN
      FOR r := 1 TO maxrules DO
        IF used IN rules[r].flags THEN
          rulecount[r] := rulecount[r]+1;
      source := corrections[p1].source;
      target := corrections[p1].target;
      FOR r := 1 TO maxrules DO
        rules[r].flags := rules[r].flags - [used];
    END
    FOR r := 1 TO corrections[p1].nrules DO
      rules[corrections[p1].rule[r]].flags :=
      rules[corrections[p1].rule[r]].flags + [used];
  END
END; { for p1 }
{ rulecount now records the most used rules in valid paths }

maxuses := -1;
mostused:= 0;
FOR r := 1 TO maxrules DO
BEGIN
  IF (rules[r].flags * [processed,used] = [used] )
    AND (rulecount[r] >= maxuses) THEN
    BEGIN
      IF ((rulecount[r] = maxuses)
        AND (rules[r].generality < rules[mostused].generality))
        OR (rulecount[r] > maxuses) THEN
      BEGIN
        maxuses := rulecount[r];
        mostused := r;
      END
    END
END; { for r }

{now remove paths that failed to use the rule }
```

```
    IF maxuses < 0 THEN
      anyrulesleft := false
    ELSE
    BEGIN
      source := '';
      target := '';
      minref := 1;
      usedthisgroup := false;
      FOR p1 := 1 TO (npaths+1) DO
      BEGIN
        IF NOT ((source = corrections[p1].source)
          AND (target = corrections[p1].target))
        THEN
        { run finishing code because we are about to meet a new
          word pair }
        BEGIN
          IF usedthisgroup AND (minref < p1) THEN
          BEGIN
            FOR p2 := minref TO (p1-1) DO
            BEGIN
              usedhere := false;
              FOR r := 1 TO corrections[p2].nrules DO
                IF corrections[p2].rule[r] = mostused THEN
                  usedhere := true;
              IF NOT usedhere THEN
                corrections[p2].flags :=
                  corrections[p2].flags + [deleted];
            END
          END;
          source := corrections[p1].source;
          target := corrections[p1].target;
          usedthisgroup := false;
          minref := p1;
        END
        FOR r := 1 TO corrections[p1].nrules DO
          IF corrections[p1].rule[r] = mostused
            THEN usedthisgroup := TRUE;
      END
      rules[mostused].flags := rules[mostused].flags
        + [processed];
    END
  END { while anyrulesleft }
END; { procedure pathchoice }
```

5.4.2 Run Time

This heuristic routine is not greedy for computer resources. By inspection, it can be seen that it operates using `FOR` loops which iterate either over the transformation paths provided as input or over the number of rules. The loops whose control variable is `p1` consider each of the input paths, and search for clusters of adjacent paths with the same source and target spellings as each other. On each such cluster, further processing is performed. The smallest such cluster is of one path, and the largest is of `npaths`.

The loop `WHILE anyrulesleft DO` processes one rule per iteration, and so will iterate at most as many times as there are transformation rules. It contains a loop which considers each cluster of transformation paths.

As the input size increases, the number of path clusters will increase sub-linearly under typical conditions. This is because some errors are likely to be repeated, leading to slightly larger clusters. The overall worst case run-time could be stated as $O(n \times r)$ where n is the number of input paths and r is the number of transformation rules.

The maximum run-time of this procedure will be as many iterations of the main loop as there are rules, and will produce a set of single paths as desired. The same path cannot exist twice for a given word pair as duplicate paths are not explored, so a process of excluding paths based on their member rules will certainly lead to a distinction between paths.

5.5 Training search

In the event of the target (correct) word being known and no transformation path having been found using the ordinary search method, this routine will be used. It is also used on new data before the rules have gained meaningful values. It disregards the rule costs which are known to be invalid, and uses a simple evolutionary search based on the A$\star$ method.

Starting with the written word, all rules are applied once to produce new permutations. The similarity between each permutation and the target word are found as described below, and all but the most similar ones are deleted. The process then repeats with the remaining permutations.

The search finishes when a match has been found, when an imposed limit to the number of permutations has been reached or when no further permutations are possible given their prerequisites.

The similarity between each permutation and the target word is measured using trigrams (sequences of 3 consecutive letters). A spelling permutation is scanned, and each occurrence of any trigram is recorded. This method has been chosen because it is quite different from the weightings associated with rule uses, and so provides a fresh perspective on the similarity of words. If only single letters were considered then their position in each word would be ignored, and similarly bigrams might fail to distinguish between patterns in different parts of some words. On the other hand, a long *n*-gram sequence would reject small improvements in the permuted spelling; for example an *n*-gram covering the whole word would only match if the permutations were identical.

The list of these trigrams is then sorted for further analysis. The fitness function which evaluates useful permutations compares the number of identical trigrams

in each permutation with those in the target word, returning a one-dimensional value for comparison or sorting. If several permutations have the same number of matching trigrams (compared to the target word) then they are all pursued further. Typically only one or two will have the maximal number of matching trigrams.

Once one or more matches have been found, the generality values of the rules used are applied as a pre-filter before accepting a transformation path. As with the ordinary search, more general rules such as wildcarded ones will be excluded at this point, leaving the specific cases, if applicable, to be reinforced for the user model.

5.6 Ordinary use

The ordinary case of spelling correction, in which the written word is known and a list must be produced for presentation to the user is similar to the preferred case during training.

This is a heuristic search unlike most existing algorithms. It applies transformations to permutations and tests them with a simple boolean fitness function; whether the new permutation is in the dictionary of known words (for a grammar based environment, the fitness function might return a boolean value indicating whether the permutation was valid, but such an extension has yet to be implemented).

The list of permutations produced so far is searched for the one with the lowest summed costs (of the rules applied to it so far). Initially the written word is the only permutation present with no rules applied and a cost of 0.

The permutation with the lowest cost (or one of them if several have the same cost) then has all appropriate rules applied to it. This produces new permutations and increments their costs according to the new rules applied.

The search is stopped when more than a certain number of valid words has been found in the dictionary, when all the permutations have a cost more than a set maximum 'budget', when all permutations have more than a set maximum number of rules applied, or when there are more than a maximum number of permutations stored.

The list of valid suggestions is sorted in order of the summed rule costs and presented to the user who chooses a word from the list. In fact, the list will already be sorted in this order, potentially allowing presentation of early results to the user before the search has finished.

If other information were available such as the grammatical correctness of each word or the appropriateness of a word in the context of discourse, such information could be used to exclude some words from the list of suggestions and then re-order it.

The permutations for the correct spelling are selected and the generality sum is used to exclude any containing unnecessary wildcards as before. The rules used in the remaining transformation paths of the permutations are all strengthened (*ie* their costs lowered) and all other rules weakened by multiplying each rule cost by a constant. Thus on the next word correction the response will be slightly different.

5.7 Worked Example

To review the preceding description of the parts of Babel, the example below explains what functions might be performed at which stage in its operation. It describes what would be done for a system trying to correct the misspelling '`pece`' in the Ordinary Search mode while using a completely blind user model in which all rule weights are equal.

When a word is submitted for correction, it is constructed as a permutation (a possible solution to the word correction problem), and becomes the only member of the list of permutations. As the search proceeds, this list grows and shrinks, and hopefully contains at some time the correct spelling of the word. In the initial example, this permutation is the misspelling `'pece'`.

The word is also converted to its phonetic form at the beginning of the search, which is then stored as a second permutation in the list with a special marker flag attached indicating that it is phonetic. In the example, the phonetic form would be */p/ /ee/ /s/*. In converting the word to this form, the rule `Phon` is applied and so is stored in the record of the transformation rule path for that permutation. The total cost of this permutation is also incremented by the cost associated with the rule `Phon` which in this case is 1.0 as there have been no cost adjustments yet. The main phonetic rule `Phon` cannot be re-applied to any permutations in this search. Only phonetic rules like `PhonIns` can be applied to permutations with the phonetic flag set, and only letter-based rules can be applied to non-phonetic permutations.

Next, the main search loop begins. Babel considers all (both, in the first case) permutations in the current list and finds the subset with the minimal edit cost. In this case it will be the initial permutation without any rules applied, which has a cost of 0. On some occasions there may be several permutations with equally low edit costs, in which case all are considered for the following stage.

Then, all allowable rules are applied to the low cost permutations. The preconditions which define 'allowable' rules are summarised at the top of Tables 5.1, 5.2 and 5.4, and in each row of Tables 5.3 and 5.5. As each rule is applied, zero or more new permutations are generated. Each is updated to contain a transformation path noting each of the previously applied rules, and the new one. The position in the word at which the rule is applied is also noted, although some rules do not have a meaningful and precise sense of position (such as `Phon` and `-finE`). The

accumulated edit cost of the permutation is incremented according to the rules being applied which in this initial example will be 1.0. Also, the Generality score of each rule is addded to the appropriate value for each permutation.

One rule which would be applied to 'pece' in the example is CforS at position 2 (the third letter), generating the permutation 'pese'. Similarly CforK will be used to generate 'peke', CforCH will be used to produce 'peche' and PforQ will be used to generate 'qece'. The wildcard rule Del will be used to make the permutations '#pece, p#ece, pe#ce, pec#e, pece#' where the symbol # represents any single character. The rules MforW, NforU, DforB and many others are not applied because their implicit precondition requiring a certain letter in the permutation is not met. Also, a range of further rules which are valid are applied to produce other permutations.

As each new permutation is generated, it is checked against the dictionary of known correct words. If it is found, an entry is made in a separate list of 'suggestions' indicating that a word is correctly spelt. When several different such entries exist, the search can be halted. If several permutations each lead to the same spelling, all of them are noted to be filtered later by the Generality system and subsequently by the Path Choice Heuristic.

After all valid rules have been applied to the lowest cost permutations, and the list of permutations has been extended, the process is repeated. The permutations which have just been processed to generate the new ones are deleted from the working list. Again a subset is found, being the minimal cost permutations from the working list. All valid rules are then applied to all of the permutations in that subset. On this second iteration, the phonetic permutation will be included as all permutations at this point have a cost of 1.0.

When any of the termination conditions of the search are met, for example

when more than a certain number of real words have been found, or when all permutations have been produced by the successive application of at least four rules, the search is stopped.

After generation of the permutations has been terminated, the list of suggestions is prepared for the user. This is ordered by lowest cost, and does not display repetitions of the same word produced by different rule paths. The system then waits for the user to select an entry, or in batch mode it consults the buffer containing the correct spelling.

If the correct word is on the list of suggestions, a subset of the list is found for all permutations leading to the same spelling. From that, the Generality scores of each path are found, and any with non-minimal scores are ignored. This is because some rules, such as `Subst` are more general than others, such as `BforD`. If paths employing both have been found for the same correction, the less general one should be recorded as it provides more information about the nature of the error.

The remaining set of permutations are considered correct solutions. The cost of rules used in each are reduced by a constant multiplier, and the cost of other rules which were not used in the permutations are increased slightly. After repeated word corrections, the costs of rules are therefore adjusted to account for the popularity of rules in correct suggestions, and rules which could be applied to lead to real words but which are incorrect are weakened by having their costs increased. Babel is then ready to check another word using its improved rule costs.

After some time of use, an expert may choose to run the Path Choice Heuristic program on the User Model. This consults the log of corrections found, and re-generates the rule costs according to the total number of preferred rule uses, discarding the incremental changes made during ordinary processing because they are less significant when a larger set of corrections is available as a whole. Without

it, the rule costs are reasonably functional in themselves.

5.8 Conclusions

A system architecture has been described for the user modelling and spelling error rule components of Babel. The complete system has been implemented and tested with sample data which is described in the next chapter. The results of the tests are presented in Chapter 7 using statistics gathered during the running of the program.

Much of the work described in this chapter was published in Spooner and Edwards (1997b) and Spooner and Edwards (1997a). The former also includes some of the results in Chapter 7.

Babel was written in C and run on a Silicon Graphics Indy computer, normally in a batch processing mode. Despite this, the methods and design would be just as valid in an interactive system where a user makes choices after each suggestion list, and where interaction is through a word processor with a mouse and keyboard instead of through a file containing a list of words.

Chapter 6

Sample Text Collection

If a spelling correction prototype system is to be of any use in research, it must be shown to perform well in realistic tests, by comparison with other systems. Many spelling correction algorithms presented in journals use methods such as dictionaries of English words, or even random strings, with exactly one transformation of one of the types understood by the system. The systems are often presented more as exercises in algorithm design than human error correction, but such a bias is not always reasonable. The work in this thesis has been assessed by the use of real samples of spelling errors from a variety of people.

Each sample is from one person, and where possible several sample documents have been collected into sets for individual people which can be compared with each other for patterns of errors, over a period of months. They are also divided into populations of similar people or of people providing samples in similar ways.

The samples used have come from two main sources. The first is a collection of essays written by school children in a number of schools around Britain. These are typically between one and three pages of handwriting which have been attributed to individuals about whom a small amount of extra information is available. Many

of the samples were from boys aged around 15 in one school in Surrey.

The second major source of data is an on-line experiment. This was a program written in Java which was made available on the World Wide Web. People from various walks of life were invited to view the related web page and type some text, both copying from an audio recording and composing original text. Their individual key strokes were then recorded for further analysis in York.

Additionally, some psychological assessments were copied from a clinical psychologist in Hull University who tested children and university students for dyslexia. Also a number of other individuals provided single samples of work.

The sample texts were collected as described and then transcribed into a computer, including corrections for the errors. The pairs of written and correct words could then be fed in a controlled manner to Babel to assess its performance. Also, the keystroke data from the Java experiment could be used for other measures of writing proficiency not directly related to spelling.

Obtaining this sample text was not trivial. Whereas in some computing domains it is feasible to obtain many thousands of samples of the required data type, and where user modelling tasks perform best with large numbers of samples, it is difficult to do the same with dyslexic writing. Synthesising samples is unfair to the algorithms under test in spelling correction, and so samples must be obtained from real people.

Contacting people who are willing to provide samples is often difficult, and obtaining those of a suitable type is also often challenging. Issues of confidentiality and self-confidence are prominent. The most difficult factor, however, is that dyslexic people by their very nature (and nurture!) do not produce much written work simply because it is so difficult for them.

6.1 Transcription

Each document received from each user is entered into a file in a standard format so that it can be analysed. This process involves hand-correcting each error made, which for the vast majority of cases is easy using human expertise such as context (what words are possible at this point), semantics (what words are meaningful at this point) and from whatever information can be seen in the written word.

A ‘document’ in this case is the text produced by one session of writing, typically one essay or one set of responses to the typing experiment. Where possible, many documents have been collected for each user, but this has most notably not been possible with the typing experiment where each participant has attempted it only once. Documents from school children were written over a period of months, providing a useful distribution on which to test the hypothesis of consistency proposed on page 159.

The transcription process was not always simple; some words are written ambiguously, and sometimes done so intentionally by the author. Where a letter was not clear it was flagged as such in the transcription and excluded from the analysis.

Only a relatively small number of words was so excluded; out of 35066 words transcribed in the corpus as it stood at the time of writing, there were 6171 misspelt words. There were 54 words where the written word was unclear (type *s*) and 132 where the intended word was unclear (type *t*), totalling 165 words where either the source or the target words were unclear, representing 2.7% of the number of errors.

Sometimes a word was misspelt as another real word, or a misspelling of the wrong real word such as a morpheme of the correct root. If the error was small and simple, for example missing the ‘s’ off the end of a plural, the correct word was entered as the target. If it was a more subtle case of grammar, for example including the wrong

```
%Author: Xxxx Haxxxx
%Title: Murder Story
%CompositionDate: 19950000 ?
%AuthorBirthDate: 19810000 ?
%AuthorLocation: Sunnyxxxx School, Cxxxx, Surrey
%DocumentType: creative prose
%AuthorCondition: dyslexic male
%TranscriptionDate: 19960507
%Transcriber: riws
%TranscriptionFormat: riws1
%Medium: handwritten
%Correction: none
%TranscriptionNotes: not double checked

He
had
din been
briving driving
of for
3
hawos | hours
```

Figure 6.1: Part of a transcribed writing file, showing the 'header' information followed by written words and corrections.

tense, then the nearest correct word was entered as the target, so the transcribed target text would remain grammatically incorrect but the spelling error would be recorded as a shorter transformation sequence.

Material transcribed for Babel may also be used at some point in the future for another analysis, for example of grammar. Words are transcribed by hand into a file with one original written word per line. If the word is misspelt then the correct spelling follows it. If the error involves a space then a vertical bar is used to separate what was written from what was intended. However, long phrase errors (for example 'month of the end') are not corrected at this point. A comment may be made in the file indicating such errors in a form which will not be processed by the spelling checker.

Each transcription file contains a number of fields of 'header' information at the top which indicates the name and age of the author, the type of material being written, the original form of the work, and any other relevant comments. Figure 6.1 shows an example of part of a transcribed file, with a typical set of headers but with personal details obscured.

The **%Author** field is the name of the person who wrote the text if known, or a unique identifier if not. In the example it has been obscured.

The **%Title** field is the name of the document given by the author. This is not used by the software, but may help administration.

%CompositionDate is an eight digit code representing the year (including century), month (as two digits) and day (as two digits) on which the document was written. If this is not known exactly it may be estimated with zeroes for the month and day. This information is necessary for calculating the age of the author, which may be needed.

The **%AuthorBirthDate** field is another eight digit code in the same format representing the birth date of the author. This is also needed for calculating the age of the author, and perhaps for correlating with other information such as the environment in which the author was educated.

%AuthorLocation is a description in text of the location of the author. This may be matched with that of other authors, but is unlikely to be used automatically.

The **%DocumentType** field is an important field representing the form of text in the document. This includes one or more words from the set { `creative, report, response, essay, prose, poem, report, experiment` }. Such words can be used to select the types of document used in further analysis, because the words used are quite different.

%AuthorCondition indicates the degree of dyslexia or other conditions that the author has. These include various terms, such as `'dyslexic male'`, `'dyslexic female'`, `'severely dyslexic'`, `'mildly dyslexic'`, `'moderately dyslexic'`, `'not dyslexic'`, `'auditory dyslexic'`.

%TranscriptionDate defines the time at which the material was entered into the computer. This may be used for backtracking in the event of a problem, or for working out the order in which documents were entered (if that were thought to indicate reliability or transcription style). Similarly **%Transcriber** indicates the name (or initials) of the person who did the transcription work, in case any systematic difficulty should be found. **TranscriptionFormat** is always `riws1`. **%TranscriptionNotes** indicate other information about the transcription process.

The **%Medium** field indicates the form which the source document took. This is one of {`handwritten; word processed`, or `typed into Java applet`}. The difference between handwriting and typing is of some interest but there were, at the time of testing, inadequate samples of both media from the same authors to

perform fair comparisons.

%Correction is a vital header for establishing that a document contains genuine spelling errors. The preferred value is `none`, which may be replaced by various alternatives such as `checked?` where it is unclear what method has been used but the errors have obviously been through some pre-processing (in particular where the words would all appear in a dictionary but are not always suitable for the context) or `franklin` for hand-written documents with the same real-word trait which appear to have been corrected by a Franklin Spellmaster device.

The transcribed files are filtered by a program, `parsesample`, that can select words of interest to the application. Firstly it excludes files which have been corrected or fail other criteria according to the headers. Then the format of the file is normalised into pairs of words (written and intended) in two columns. For normal use, only the incorrect spellings are used; words which were written correctly are not passed to the main spelling program. Errors involving spaces, *ie* phrase or segmentation errors, are not used at present.

A further problem with the transcription of spelling mistakes is the correctness of the transcription. After hours of work, checking words for letter-by-letter individual correctness, it is difficult to guarantee that there are no errors in transcription. To this end, samples have been double checked where time permitted, and a small proportion of errors were found (less than 5%). It is believed that those errors which remain undetected are not serious enough to jeopardise the results of this work.

6.2 Java Experiment

In a search for more sample data, it was decided to write another program to administer an experiment which plays recordings of sentences and asks people

to type them at the keyboard. The program would be offered to a wide range of people over a long period of time, bringing in substantial samples of data for less effort than that required for other methods.

6.2.1 Objectives

The prime objective of the Java experiment was to collect error samples to use in testing Babel. There were a number of further objectives from this procedure. Firstly, detailed timing and behaviour information was to be gathered in a way not possible by reading a written manuscript. The keystrokes would show general typing proficiency and specific faults of interest.

Each intermediate spelling of a word would be visible; *ie* it was believed that some people write by 'trying out' a spelling on screen and then editing it if it does not look right. All of these spellings would be recorded.

It was also believed that typing speeds vary according to the confidence of the writer. Thus when writing a doubtful word the user would slow down. This might be used in a real-time spelling checker to initiate word correction without a specific command from the user. Measuring such variations in speed would be desirable if research time allowed.

As with typing speeds, so the use of the *Delete* key may reveal information about the user. It is probably possible to identify someone trying out various spellings by examining a log of keystrokes. Each of the attempts might hypothetically be used in a spelling checker to boost the confidence of the result.

It was hoped that samples could be gathered from a larger population, hence bringing more statistical confidence to the results. Activity on an electronic mailing list (`dyslexia@mailbase.ac.uk`) seemed to suggest that a reasonable number

of appropriate people would be willing to participate.

The text written in the experiment would have been directly typed by the original author, hence no new errors would be introduced at the transcription stage. There have been some doubts about the transcription procedure for handwriting in that some aggressive interpretation may lead to apparent letter substitutions where in fact letters were being written poorly but not wrongly.

It would also be possible to see the attempts by each different person at writing the same standard words, hence some accurate measure of variation may be made in a way not easy with different spellings of different words which are found in creative writing.

6.2.2 Presentation

The experiment program was written as an 'applet' in the Java programming language which can be run on many computers across the World Wide Web. Java was chosen firstly because it is widely available across platforms. It also contains all the required graphics and sound facilities in its standard distribution, and would carry a certain kudos amongst users wondering whether to try using it.

The experiment begins with an introductory web page explaining the principle of the work, the reasons for the experiment (including what will be recorded) and advice on behaviour. At the bottom of the introductory page is a link to the main experiment; a page with very little text. There is a link from the main experiment page to the previous page, and a large space in which the experiment applet itself runs. If the user cannot run Java applets then a message appears in its place explaining this and pointing to links for Netscape software.

The applet begins with a form asking for some personal details (Figure 6.2). Note

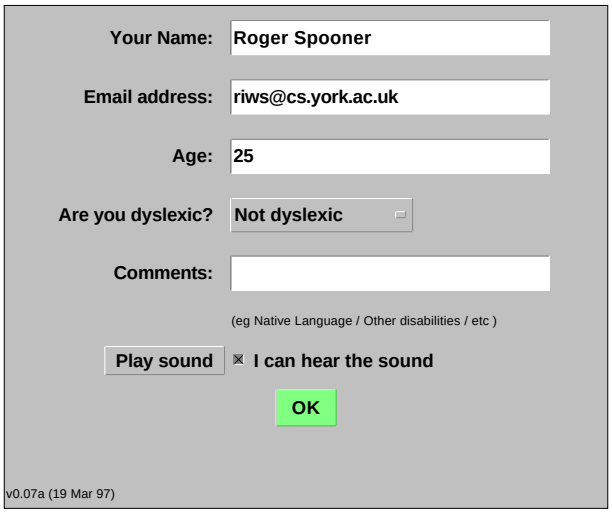

Figure 6.2: The Java experiment requesting personal details.

---

that in addition to these questions, some tasks in the experiment invite more details about the user. The form requests the user's name and age, their opinion of their dyslexia severity and other descriptive comments they see fit to add, such as their native language. They must confirm that they can hear the sound sample played to them which also serves as an opportunity to improve their listening environment (for example by wearing earphones) and to begin to understand the voice.

During this period of filling in the form the applet also communicates with the server computer in York to establish a TCP network connection on which to record key presses. The experiment will not proceed unless the key presses can be recorded. The software was designed to avoid the delays of synchronous communications as far as possible; once the applet has established that the server is listening, it sends keystroke data to it and does not request any further acknowledgement at the application level until the end (the TCP protocol ensures reliability of the stream so long as bi-directional communication remains possible).

The time is recorded in three ways; the time on the server at which the recording

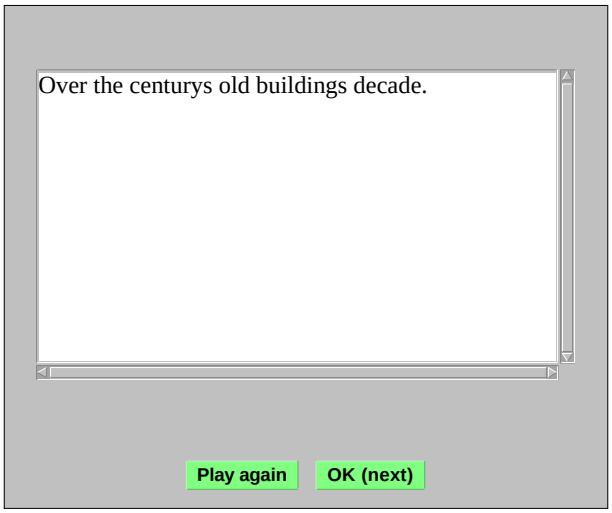

Figure 6.3: Trial 2 of the on-line experiment. Some window tools would normally also be visible to the user.

---

connection was opened, the “real time” in the applet on the user’s machine at which each action was noted, and the time reported by the window environment of the user’s machine at which user input events occurred. In the present analysis only the applet’s real time clock was used, but the three methods were used because each had unique merits. The network delay in transmitting data meant that time could not be measured accurately on the server. However, application and operating system delays also meant that time in the Java applet was not always reliable. Some window environments did not properly report the time of an event such as a key stroke, and so this measure is recorded but not normally used, but where it is correct it provides the most accurate measure of time, being before much processing has been performed.

After the initial form has been completed the display changes to a large text area into which the user can type with two buttons at the bottom. One button replays the sound sample whenever requested, and the second moves the experiment on

to the next task. This is illustrated in Figure 6.3. The user can click on either at any time (although they must enter some text for each task before proceeding to the next).

The text area uses a large and easy to read font; 18 point Times. This is to avoid confusion due to the size of text as far as possible, which is known to affect some dyslexic people. There are scroll bars on the area, and its operation in general is the standard for the computing platform being used to the extent permitted by Java[1]. It is possible, therefore, to use the mouse and cursor keys to move around the typed text and edit what had been written before. The exact appearance and functionality varied between computers.

After each task sentence the text area is cleared, and after the sixth it gets slightly larger to indicate the change of context. At the end of the experiment the display changes to a closing message thanking the user for their participation.

The choice of sound quality in the experiment design was a rather difficult issue. The sound was recorded in an office by the experimenter using the microphone provided with a Silicon Graphics computer. It was recorded with as little background noise as was practically possible. To deliver the sound to users across the world wide web, some of whom were using modems which could transfer as little as 2 Kbytes per second, the quality had to be reduced. Another practical problem was that Java applets seem only able to guarantee the playback of sounds in the Sun 'mulaw' encoding format.

It was decided to use a sample rate of 8Khz with monophonic 8-bit sampling. This lead to sound file sizes of between 16Kb and 56Kb, thus having a typical downloading time through modems of around 15 seconds. This time was spent

[1]Java provides a number of 'window toolkit' features such as graphical buttons and text input fields. These are rendered on the user's display in a platform dependent manner, with the intention of making the applet fit more comfortably into the user's computing environment.

at the start of each trial before the sound was heard, and hence at a time where downloading rates are not critical to the results. Once it had been played once, a replaying at the user's request would be immediate.

6.3 Task choice

People performing the experiment would be doing so by choice, and would presumably stop as soon as they saw no benefit in continuing. Because there was no payment for the experiment, the threshold of interest was low. It was decided to ask users to perform a handful of simple tasks and a smaller number of more complex ones, with the whole task taking between five and ten minutes.

The tasks for the experiment consist of a series of 10 sentences. The first six of these are short and fairly simple, with a range of complexities of words. They are to be typed in exactly as heard. The sentences use a variety of words including some from psychological spelling tests such as the Wechsler Objective Reading Dimensions test and the Wide Range Achievement Test from Jastak Assessment Systems. The author of this work is not a psychologist, and so cannot justify the choice of words precisely. They were chosen to include words of varying familiarity and orthographic regularity while being presentable in brief and meaningful sentences. The sentences were:

- I saw an exciting drama yesterday.
- Over the centuries all the buildings decayed.
- The sovereign of a country is a prestigious person.
- Apparently, pharmaceutical companies make lots of drugs.
- The beginning of the movie was set at night.
- Lots of actors went to the audition.

The remaining four tasks required a more personal response to a question. These were intended to stimulate creative writing, longer answers, a demonstration of the user's vocabulary and grammar. They were also intended to be more relaxing than the formality of the first part. The task requests were:

- OK, that's the end of the sentences. Now I'd like you to tell me something about your work.
- Would you like to tell me a bit about some interesting people you've met recently?
- Could you tell me how much experience you have of typing and writing, and what you think of your own spelling, please?
- Here's your chance to get your own back on me! What did you think of doing this experiment?

There was only one experimental 'group' of this experiment, that of full participation. Each participant received the same tasks, but a division might be made later based on any of the information known about each user such as their age, gender or diagnosis.

Text typed by the experiment participants was converted to the standard transcription format used for the main spelling checker program. Only those people who made a full and proper attempt at the experiment were included; some people stopped part of the way through or did not complete answers to the tasks.

6.4 Participation

The experiment was advertised on a number of mailing lists and news groups, to a number of specific individuals, and in the newsletter of the British Dyslexia Association. The statements suggested that it was open not just to dyslexic individuals but to anyone; this was because the user model on which the spelling

correction system is based may not require dyslexic errors to function, and so there is no point restricting participation in this respect. It was also linked to a number of other Web pages which later turned out to draw more participants than the advertisements.

Each person was invited to gauge their own level of dyslexia which is difficult for many people, but not essential for the analysis of results. The only human requirement is that the experiment be undertaken seriously so that typing errors made are real errors. If that were not the case, an analysis of typing speed might reveal frauds. At the time of writing, the only such invalid responses were experimental tests conducted by the author.

6.5 Results

The experiment has not been used by as many people as was hoped, probably partly because it was not adequately advertised and partly because people did not see much point in participating (the author may have overestimated the enthusiasm of Internet users for novel opportunities to help other people). The technical requirements (of a web browser with Java execution capability, an Internet connection, and a sound output device), necessary to make participation possible at all reduce the number of eligible people but not in any problematic pattern. There were more computer technicians and other enthusiasts.

The set of people with equipment capable of participating in the experiment is limited, but not impossibly so. Because it requires people to choose to participate, it is self selecting. This may affect the results of the data; more severely dyslexic people seemed reluctant to reveal their typing habits to someone else. This is understandable although unfortunate. Apart from the generally high standards of spelling, none of the patterns of behaviour discovered undermine the results.

Good spelling is not a bad thing in itself; however it may render a user model insignificant with too few occurrences of errors.

The text area widget for typed responses did not automatically wrap the text at the end of each line and some users did not press Return. One complained that the inconvenient scrolling of this box made their results worse than they should have been. It is possible that in the long answers they were more confused.

Java, although it is intended to be a standard which runs on many platforms, is not entirely consistent. On some machines it was not possible to record individual keypresses while on others typed text was not displayed until the mouse was moved out of and back into the window. This is unfortunate. The appearance of the display was not identical on each machine because of the platform dependent window style. However, this is unlikely to have affected the users' performances.

Many of the participants were not dyslexic. Thirty eight stated that they were not while 1 was Severely dyslexic and 5 moderately or mildly so. Ten were 'not officially diagnosed' which probably includes not being dyslexic in some cases. It was perhaps a design fault to include that choice as an option.

Not every participant disclosed their age. The youngest who did was 13 and the oldest 53. The mean age was 28 with a peak around 24.

The design was intended for the experiment to take around five or ten minutes, so it is interesting to see that the most common time was about eleven minutes, with three quarters of the participants finishing within 19 minutes and one taking over an hour to write an in-depth discussion of his work. The mean number of keystrokes was just over 900, with three quarters of the participants writing less than 1800 keystrokes, and the marathon man achieving nearly 4000.

Comments entered in the form at the beginning included one statement that the

participant's native language was Dutch, several references to dyslexic relations, one complaint about the quiet sounds and one offer of a curry when I'm next back in Edinburgh.

To examine the records of participation, another Java applet was written to read the log file created by the experiment system and replays it at the same speed. It revealed that many users write at varying speeds, edit their sentences a lot, and even change their statements completely, apparently because of a repeated inability to spell a single word. Most people almost never use the cursor keys to edit earlier text in place, preferring to delete from the end to the point in question and retype the remainder.

Typing speeds varied substantially. The most common typing rate was about 4 keystrokes per second (a fast typist would achieve perhaps 12). A number of people type in clusters of keys, seemingly preparing themselves by thinking of spellings and key positions and then launching into a burst of letters.

Chapter 7 contains a review of all of the sample text obtained and the size of each population. It is summarised in Table 7.1.

Chapter 7

Results from BABEL

This chapter presents the results of running the sample text errors through Babel. It considers the degree to which it is possible to obtain a result to establish the performance of Babel compared to other programs, the performance of Babel in various modes, and a number of other less quantitative measures. The results are less impressive than would be liked and so some consideration is given to the possible reasons for this.

7.1 Sample Text

The samples of text, collected from various different sources, are of quite different sizes and natures. To some degree this affects the analyses that they can be used for.

Table 7.1 summarises the sample populations used. The column ‘group’ indicates the name used to refer to the population throughout this thesis. ‘People’ gives the number of people from whom the samples were taken, and ‘Docs’ shows the number of documents sampled, with ‘Words’ being the total word count.

Group	People	Docs	Words	Rate	Quality	Description
edin	1	4	385	30.4%	Dyslexic	Dyslexic boy
yorksch	4	4	614	17.9%	Dyslexic	Dyslexic children
hullp1	52	104	7592	35.2%	Young	Children, aged $\cong$ 7.4
hullp2	84	168	12264	41.6%	Young	Children, aged $\cong$ 7.8
HullP$\star$	344	688	50224	33.8%	Young	All Hull primary
sunny	10	60	19301	14.0%	Dyslexic	Boys aged 13-15
java	6	6	760	22.8%	Dyslexic	Adult students
inet	25	25	4589	10.5%	Mixed	Mostly adults
hullstud	17	17	3652	11.6%	Mixed	Students
yorkcoll	3	6	1309	10.8%	Dyslexic	Students

Table 7.1: The number of people, words and errors in the various sample populations used in this research. Not all of them were dyslexic.

'Rate' shows the percentage of the total number of words which are mistaken and 'Quality' indicates whether members of the population were known to be dyslexic or not. The 'Description' is a very brief summary of people making up the population.

By far the largest, HullP$\star$ is that from ordinary primary school children in Hull being tested as part of a student project. There were more than 300 children in four schools, aged 6-9. They were given two identical tests of 73 words for spelling. Some of the children got almost all of the words wrong, whereas a few got the majority correct. Because the children were so young and because there were no special reasons to believe they were dyslexic, it would be unwise to assume that they had even a basic knowledge of correct spelling from which they were deviating or that there were unusual patterns in their deviations. The `HullP`$\star$ samples were therefore excluded from most of the analyses because the immaturity of the authors posed serious questions about their errors. Samples from the first two schools, `hullp1` and `hullp2` were used in some tests.

The `inet` group was much smaller but more diverse. It included many adults in jobs who thought they were dyslexic, as well as a number of children and people who were not dyslexic. Both the `inet` and `java` groups used the Java typing experiment program described in Section 6.2 and so provided detailed writing logs beyond just the final spellings of words. The remaining samples were handwritten.

The `java` group was tested using the same software as the `inet` group, but were students at a college. They had all been diagnosed as dyslexic adults, and were attending literacy classes. They were mostly very poor spellers and wrote slowly. Their work was all done carefully, and under more controlled environmental circumstances than the `inet` samples. They also contributed some handwritten material, labelled `yorkcoll`.

There were a number of students at Hull University who were tested for dyslexia, but not all were confirmed as having it. The sample, where used, is labelled `hullstud`. These samples were obtained with the help of staff at Hull university but have no other connection with the `HullP`$\star$ samples.

The `sunny` group included substantial samples of text from dyslexic children in a secondary school in Surrey. There were about a dozen participants, each submitting six documents on average, each of which was of substantial length. They were handwritten, like the Hull samples but unlike the Internet and York samples. This group is the most reliable as there were substantial samples from an adequate number of people, all of whom were known to be dyslexic.

The remaining samples; `edin` and `yorksch` were from dyslexic school children aged 11-13 years. The samples were handwritten in normal classes by children with recognised literacy difficulties and transcribed later.

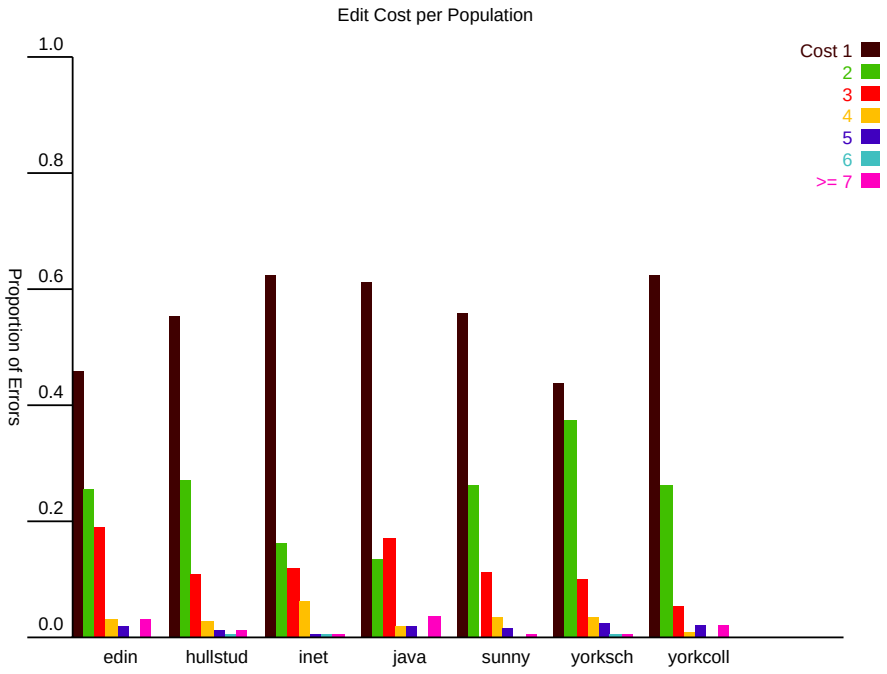

Figure 7.1: The number of unit-cost editing operations for various sample populations. This can be compared to the number of rules per path in Babel, shown in Figure 7.6 on page 172.

7.2 Error Severity

Figure 7.1 shows the number of errors which were found during a training search with Babel. The number of error transformations equates reasonably to the severity of the error. Words without any errors are of no particular interest, and those with just one are probably simple mistakes that many spelling checkers could identify.

The row labelled ‘Unknown’ in Figure 7.1 indicates those words for which the training search terminated before finding a match. This occurred typically where the precondition rules prevented a correct solution from being found, for example because several letter deletions or insertions were required to correct the error in reality. The preconditions were included to limit the run-time of the system with

difficult words.

The graph in Figure 7.1 shows a visual representation of the error rate when measured as a unit edit cost. The editing operations are simpler and thus often require several to correct the same error as can be corrected with one rule in Babel. It is fairly clear that the populations are different; the school children in `edin,sunny,yorksch` show relatively more errors with higher costs ($k > 1$) than the other populations and only a few populations have words with more than four unit-cost editing operations.

Documents from each user, whether from handwriting or typing, were processed by the main spelling program. The format generally adopted was to begin with a 'blank' user model and to run one document from a particular user through it with the program in its heuristic search mode, which tries to find a path connecting the two spellings of the word (that written and that intended) using the rules available.

Once the user model has been built for a particular document it is then copied for future reference, and then re-run with the program in its normal search mode. Here the number of corrections is counted to assess the usefulness of the system. The program can also be run on a different document by the same author.

The consistency hypothesis was devised to test whether a user model makes any useful contribution to the function of a spelling checker. When given a number of documents written by one person, and a number of documents written by other people, it proposes that there should be significantly more consistency between the user models for documents by the same person than with documents by other people. If the user models are not significantly different from each other then there is little point having a dynamic user model and one might as well use an ordinary spelling checker.

To test this hypothesis, two major experiments were conducted (Section 7.5) with

the results of using the 'Babel' spelling checker prototype. The first was to review the frequency of use of each of the error pattern rules. The second (in Section 7.7) was to compare the letter-position in the words of each application.

A second hypothesis would claim that using Babel with a user model gives an improvement over that without. This can be divided into two forms; that an improvement occurs between the 'blind' and 'canonical' user models, and that an improvement occurs between the 'canonical' and 'individual' models. This is tested in Section 7.4.

7.3 Comparison With Other Software

In addition to testing the hypotheses for user model performance, Babel has been run to compare its overall performance with two existing spelling correction systems. The first, `ispell`, is a standard package used throughout the academic community and has a high success rate at correcting simple errors. It is available by anonymous FTP from ftp.cs.ucla.edu in the directory /pub/ispell-3.1. The second system, SPEEDCOP, was presented by Pollock and Zamora (1984) and re-implemented for this work.

Sample text was collated by author (where multiple documents by a single author existed) and randomly divided into two parts. The 'part 1' section contained approximately 75% of words from the text. 'Part 2' of the text contained the remaining random sample of the words by that author; typically around 25%.

Where Babel was used, it was first trained on the Part 1 section using the directed search described elsewhere, and then the user model pruned using the path choice heuristic algorithm and the system re-tested for correcting errors from the remainder (part 2) of the sample data in the normal fashion.

`ispell` was tested by running it on Part 2 of the sample text, and recording the position of the correct word if present. The spelling dictionary used for `ispell` is the same as that used for Babel and contains all of the correct spellings. The option to sort results by probable correctness was enabled. In this way, the results of Babel and `ispell` are comparable.

`ispell` operates using the conventional single edit principle, in which one insertion, deletion, transposition or substitution is allowed on any letter other than the first, in a given word. Each permutation generated is searched for in a dictionary which contains many words including derivative morphemes in a highly compressed format. The order in which suggestions are sorted is not clear.

SPEEDCOP re-orders the letters in a word by putting the consonants first in a specific order followed by the vowels in the order in which they appeared. It then searches from the nearest sorted position in its dictionary for words up to a fixed radius of 50 words. The original algorithm only accepted words with one editing operation but that has been removed for the benefit of the dyslexic samples considered here. That is, SPEEDCOP as published would only accept words with an edit cost of one and less than fifty words away from the key generated for the error. In this case, the correct word was known and searched for amongst the words near the error word's key regardless of the degree of difference. SPEEDCOP was designed for scientific and scholarly text; something quite different from dyslexic work, but it has been included in this comparison because it was feasible to test, and because it was the result of published research and as such could contribute to assessment of this work.

The performance of the various systems have been assessed in terms of the frequency with which the correct answer appears on the list of suggested corrections, and the position of that suggestion on the list when it is present. Although it would be more agreeable to combine these two figures into one overall measure

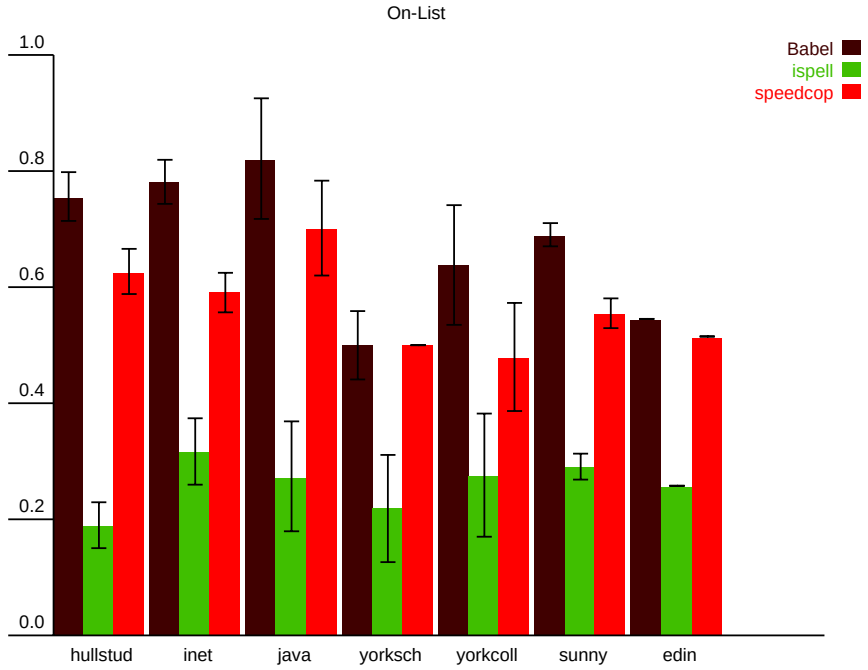

Figure 7.2: Proportion of words tested in which the correct appeared on the suggestion list when run through Babel, `ispell` and SPEEDCOP.

---

of performance, it is not possible to do so without introducing subjective measures.

Figure 7.2 shows the proportion of occasions on which the correctly spelt word was suggested on the list, for a number of different author populations being run through Babel, `ispell` and SPEEDCOP. The significance of the differences are stated in Table 7.2. Standard Error bars are also shown on the graph, being $\frac{\sigma}{\sqrt{n}}$ for a population with n people and a standard deviation of σ between their results. When tested using an Analysis of Variance[1], Babel suggested the correct answer significantly more often than `ispell` in all cases (with 95% [$p < 0.008$] confidence for `yorkcoll` and `yorksch`, and at 99% [$p < 0.0016$] for the remainder). Babel also out-performed SPEEDCOP in two populations (`inet`, `sunny`) at 99% significance

[1]The Analysis of Variance test is discussed in more detail on page 177 in this chapter.

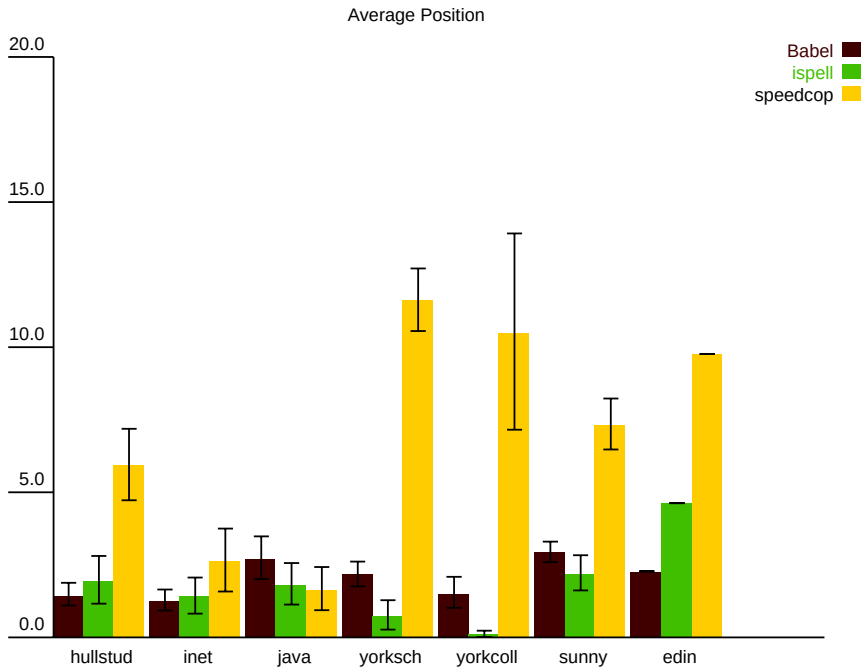

Figure 7.3: Average position of the correctly spelt word (if present) when run through Babel, `ispell` and SPEEDCOP. Lower scores indicate better performance.

and the `hull` population at 95%.

The results in Figure 7.2 concerning `ispell` are almost all very significant for Babel; that is to say, Babel performs significantly better than `ispell` at including the correct word on its list. In a series of one-way Analysis of Variance tests, all but two populations achieved a probability of an accidental difference (the probability of an insignificant result) of $p < 0.008$ (95%), and indeed for the `hull, inet` and `sunny` samples the probability of insignificance was reported by the statistical software package SPSS to be 0.0000. The difference was not significant for the `yorksch` and `yorkcoll` samples.

Recall that SPEEDCOP always includes a list of 100 words, being those closest to the original word when sorted in SPEEDCOP's particular manner. Thus it is not so surprising that the correct word was often on the list albeit far from the head position.

Figure 7.3 shows the mean position of the correct word on the list, when suggested. The head of the list has positional index 0, so a lower score is preferable. Unfortunately in this case Babel performs similarly to and often worse than `ispell`. It is clear from inspection that the difference is not significant, and this has been confirmed by an ANOVA using SPSS, in which the probability of insignificance varied from 0.09 to 0.83; nowhere near the threshold of 0.008 (equivalent to 95% confidence).

Compared with SPEEDCOP, however, most of the results for Babel are significant in Figure 7.3. Babel suggested the word significantly nearer the top of the list with a confidence of 95% for the `yorkcoll` population and with a confidence of 99% for the `hull, sunny` and `yorksch` populations.

Table 7.2 summarises the degree of confidence with which Babel can be said to perform significantly better than `ispell` and SPEEDCOP both in terms of average

	On-List Rate		**Average Position**	
Population	`ispell`	SPEEDCOP	`ispell`	SPEEDCOP
`hullstud`	99%	95%	-	99%
`java`	99%	-	-	-
`inet`	99%	99%	-	-
`sunny`	99%	99%	-	99%
`yorksch`	95%	-	-	99%
`yorkcoll`	95%	-	-	95%

Table 7.2: The confidence that there is a significant improvement in Babel over the other systems `ispell` and SPEEDCOP. Neither of the other systems was ever significantly better than Babel in any way tested. The results are shown graphically in Figure 7.2 and Figure 7.3. The `edin` population has no variance.

position on the suggestion list and the rate of appearance on the list.

Deriving a relationship between the rate at which the correct answer appears on the list at all, and the position that it appears in when it does, has proven difficult. If an arbitrary position were to be assigned to the concept ‘not on list’ then this would weight the resultant average severely. For those cases where the answer is on the list more than half the time, the median position might be used as this disregards the details of the extremes. However, in some cases the median would be off the suggestion list, and hence would again need a value to be assigned arbitrarily.

Even combining the issue of the correct word being on the list and that of the position on the list, if present, is a difficult one and has been avoided to retain clarity and avoid subjective judgements.

Nisbet, Arthur, and Spooner (1998) presents a method for combining the performance of spelling checkers into a single value but does assign pre-defined constant values to situations such as the absence of the correct word from the list, the acceptance of an incorrect word and the presence of the correct word lower down

the list. In that case, a penalty was applied where the word was not initially visible and the user was required to scroll down a long list to find it. It also weights the relative importance of the position of the correct word and the total length of the list, based on informal practical experience.

7.4 Correction Improvement

To evaluate the User Modelling system, an experiment with three stages was constructed. The aim was to find out if having a user model improved performance. A positive result for this test is a fundamental part of a positive result for the whole work on Babel.

The sample text was divided into two parts, for “training” and “testing”, as described on page 160. As before, all training of the models was done on words from Part 1, and all testing on Part 2. This ensured that the User Model could not rely on learning individual word errors.

The first stage of the experiment took a ‘blind’ user model in which all rule weights were set to 1.0. This is equivalent to the *unit cost* systems described in Chapter 8. The Part 2 samples of text were run through Babel, author by author in its normal mode but with rule cost adjustment disabled. For each error word a list of suggestions was prepared, and the position on the list of the correct word (when included) was recorded.

The second stage of the experiment involved a ‘canonical’ user model. This was built by running all Part 1 sample text through Babel in its training mode. After this, the Path Choice Heuristic analysis described in Section 5.4 was applied to generate globally preferable rule costs. Figure 7.4 shows the rule priorities after this analysis.

Once the canonical user model had been built, each 'part 2' document was tested in Babel's normal mode but with rule cost adjustment disabled (to ensure that the user model remained canonical). The position of the correct word on each suggestion list was again recorded.

The third and final stage of the experiment was to consider individual user models. If only one document or the documents by one author were considered then an individual model would have a disadvantage compared to the canonical model which is built from thousands of words. Thus building an individual user model directly from the blind one would lead to a poor performance.

To remedy this, individual user models were built by taking the canonical model from the second stage and training it further by running the Part 1 text by one author through it in the normal search mode. They were then tested by disabling further changing of the user model and considering the position on the correction list of the correct word, when tested on each of the Part 2 words in the document.

Throughout this experiment, the rule use position feature, which records the location within words at which errors more commonly occur, was disabled. This was to simplify the operation of Babel and make the results more reliable and meaningful.

Without any preconception of the accuracy of Babel or its performance by comparison to other spelling checkers, the user model should improve performance by bringing the correct word closer to the top of the list of suggestions.

Figure 7.4 shows the improvement in suggestion list position between the three stages of the test. Standard errors are again shown on the graph, calculated as $\frac{\sigma}{\sqrt{n}}$ where σ is the standard deviation. The standard deviations were also used to find the probability that the results were significantly positive. Although in most cases Babel achieved an improvement through the use of its user models, results

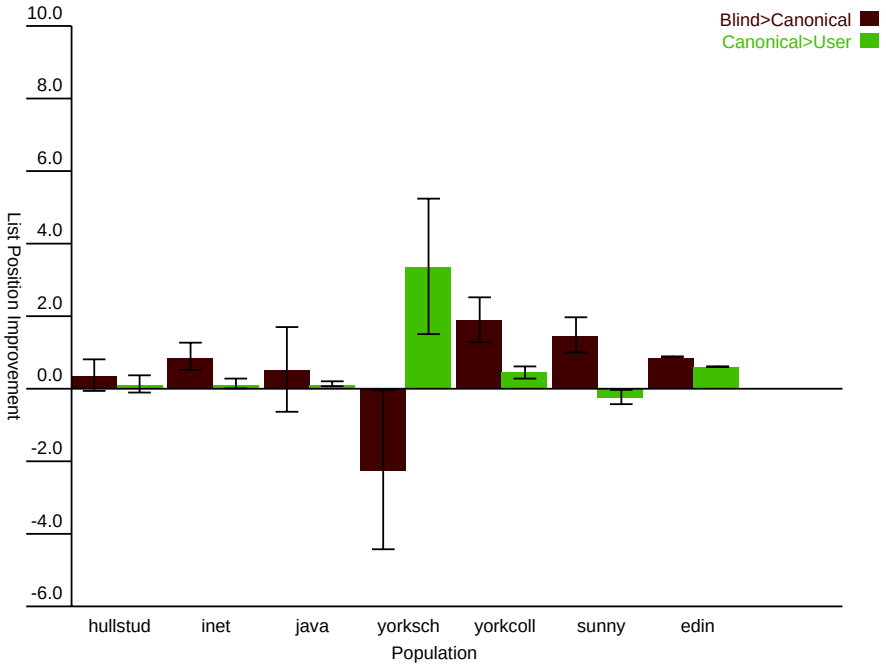

Figure 7.4: Suggestion position improvement between Blind, Canonical and User models of spelling errors for various populations.

for each population group were found to be insignificant; some of the changes in list position were even negative, unfortunately.

The proportion of words in which the correct word was suggested remained approximately constant. In the vast majority of cases no change was observed and in a few, one or two words were either included or excluded from the list where they had not previously been; the average was a reduction in the number of correct words on the list of 0.2%. This is not shown in a graph.

Figure 7.4 also shows that the improvement in list position varied between populations. Recall that not all participants were dyslexic; in fact only `yorkcoll`, `sunnydown` and `edin` had been diagnosed as such. Those groups had a more substantial improvement in Babel performance, while still failing to be a significant improvement (except in the case of `edin` which contained work from only one author and so has no statistical variance).

7.5 Rule Usage

Not all of the rules were equally popular, and some varied noticeably in usage across the different populations; something which might indicate the diversity of styles of the various author groups.

The Path Choice Heuristic is only useful where several paths exist for the same word pair. With such a choice it can select the paths with the most likely solutions to enhance future operation. Because of the generality scores and rule counts, the majority of cases featured only one valid path. That is to say, only those solutions with the correct spellings and with a minimal number of rules, and a minimal generality score, would be considered. Only about 5% of the words attempted had more than one such viable path.

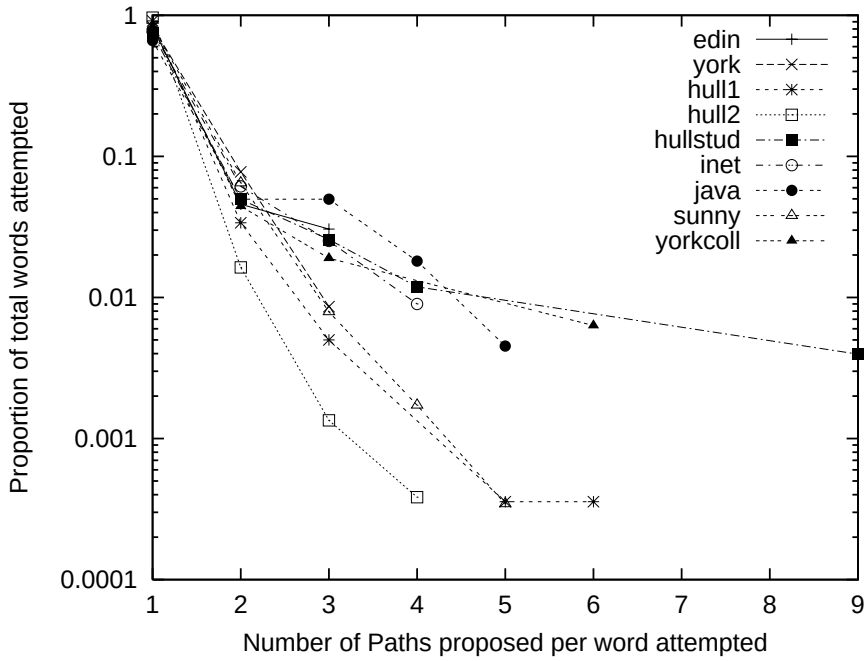

Figure 7.5: The number of solution paths (combinations of rules) selected for each of the words considered for various populations of authors.

Group	Paths	Words	Unique
edin	131	118	105
yorksch	116	106	90
hullp1	2804	2673	1503
hullp2	5210	5106	3088
sunny	2905	2653	1918
java	221	173	155
inet	556	480	387
hullstud	505	421	387
yorkcoll	158	141	125

Table 7.3: The number of solution paths, and the number of distinct words tested, for the various sample populations.

Figure 7.5 shows the proportion of words for which several paths exist. The lines shown represent several different sample populations. Table 7.3 lists the number of solution paths for each population, and also the number of words, and the number of uniquely different words. The same word with the same misspelling may occur more than once in a sample. This is counted as several words and will have several solution paths, probably including identical ones.

Before going on to consider the use of each type of rule, it may help to see the number of errors found in each word. Figure 7.6 shows the number of error pattern rules found in each of the correction paths (from each of the population sources shown above).

It can be seen from Figure 7.6 that around half of the words can be corrected using only one error rule, with the proportion falling almost logarithmically, up to a maximum of four rules per word. This maximum is imposed by the system to prevent sets of permutations ‘running away’ and consuming too much computer time.

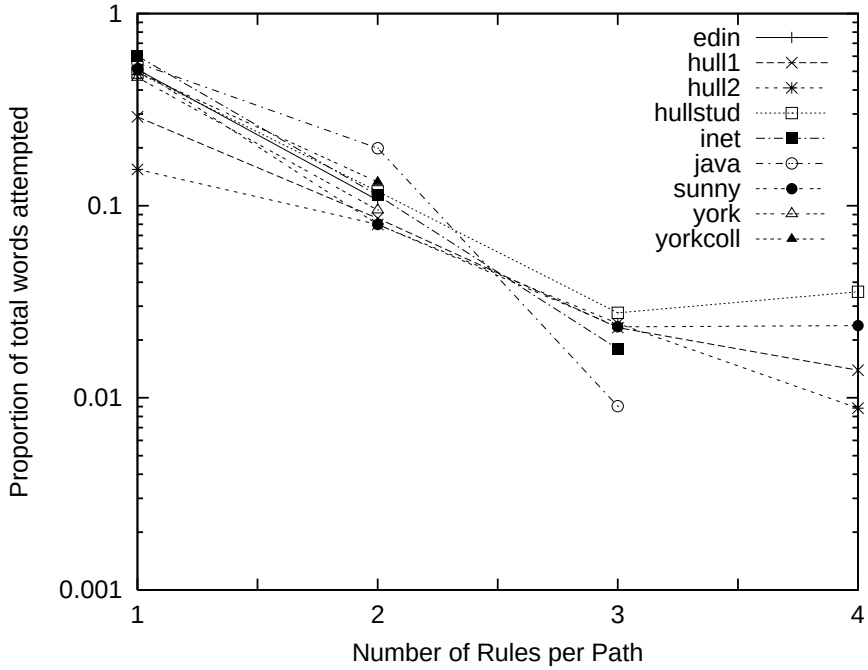

Figure 7.6: The number of error pattern rules needed to correct each of the words checked.

Generality:	0
Rule	**Usage**
Phon	1.00000
Trans2	0.14367
Cons2	0.11616
2Vow	0.09738
2Cons	0.06157
+finE	0.03886
WHforW	0.02882
SforC	0.02751
-finE	0.02533
WforWH	0.01965
DforB	0.01659
BforD	0.01397
TforTH	0.01004
CKforK	0.00786
CforCH	0.00742
KforCK	0.00699
ZforS	0.00611
GforGH	0.00568
CforK	0.00524
phTrans	0.00480
KforC	0.00480
SforSH	0.00437
CforS	0.00437
UforN	0.00306
NforU	0.00306
GforC	0.00262
CforG	0.00262
SforSC	0.00131
WforM	0.00087
QforP	0.00087
QforQU	0.00044
CforCK	0.00044

Generality:	2
Rule	**Usage**
phSubV	0.4507
SubVow	0.4319
Vowel2	0.2410

Generality:	3
Rule	**Usage**
Del	0.5341
Subst	0.3843
Ins	0.2196
phonDel	0.1192
phonIns	0.0485

Table 7.4: Relative usage of Babel's pattern rules after training and testing on most samples. The proportion 1.000 has been scaled to represent the most commonly used rule. See tables 5.1 to 5.5 (from page 112) for an explanation of the rules. Rules in the left hand column have a generality value of 0. Those in the middle have a value of 2, and those on the right use 3.

Not all of the rules were used with equivalent frequency. In a training search session of the samples `edin, hullstud, inet, java, sunny, york, yorkcoll`, rules were observed as shown in Table 7.4 (numerical values are as a proportion of the most used rule). As the data above were gathered from a training search, the rule use patterns are not necessarily the same as for normal searches. However, these will substantially influence future behaviour through their adjustment of rule weights. The low priority of rules like 'DforB' is disappointing although somewhat offset by the generality score system which will always choose such rules in preference to wild-card rules like 'Subst'.

To begin to assess the difference between users, graphs were plotted showing the difference between the rule usage for different users and different populations. A small selection of these are shown in Figure 7.7. The rules were scaled as in Table 7.4 so that the most used rule measured 1.000, for each of two sample sets. One set was then subtracted from the other.

Figure 7.7 shows some interesting differences for a few cases which are illustrated, but in the majority there were no significant patterns of consistency within populations or even authors, and no significant differences between them. Where the differences are shown as less than 10% it is most likely to be due to noise rather than systematic influences. Author MI from `sunny` includes a distinct preference for rules Cons2 and 2Cons as well as Vowel2. 'BD' from population `inet` has a preference for Del rather than Ins and for 2Vow rather than Vowel2.

The lower two graphs in Figure 7.7 represent the difference between one population and all the others. The `sunny` sample has negligible differences whereas `hullstud` shows some preference for Subst, phSubV, Ins and phonIns but not SubVow.

Overall, there is only limited consistency between users within a population or even within work by a user. The majority of comparisons showed negligible differences,

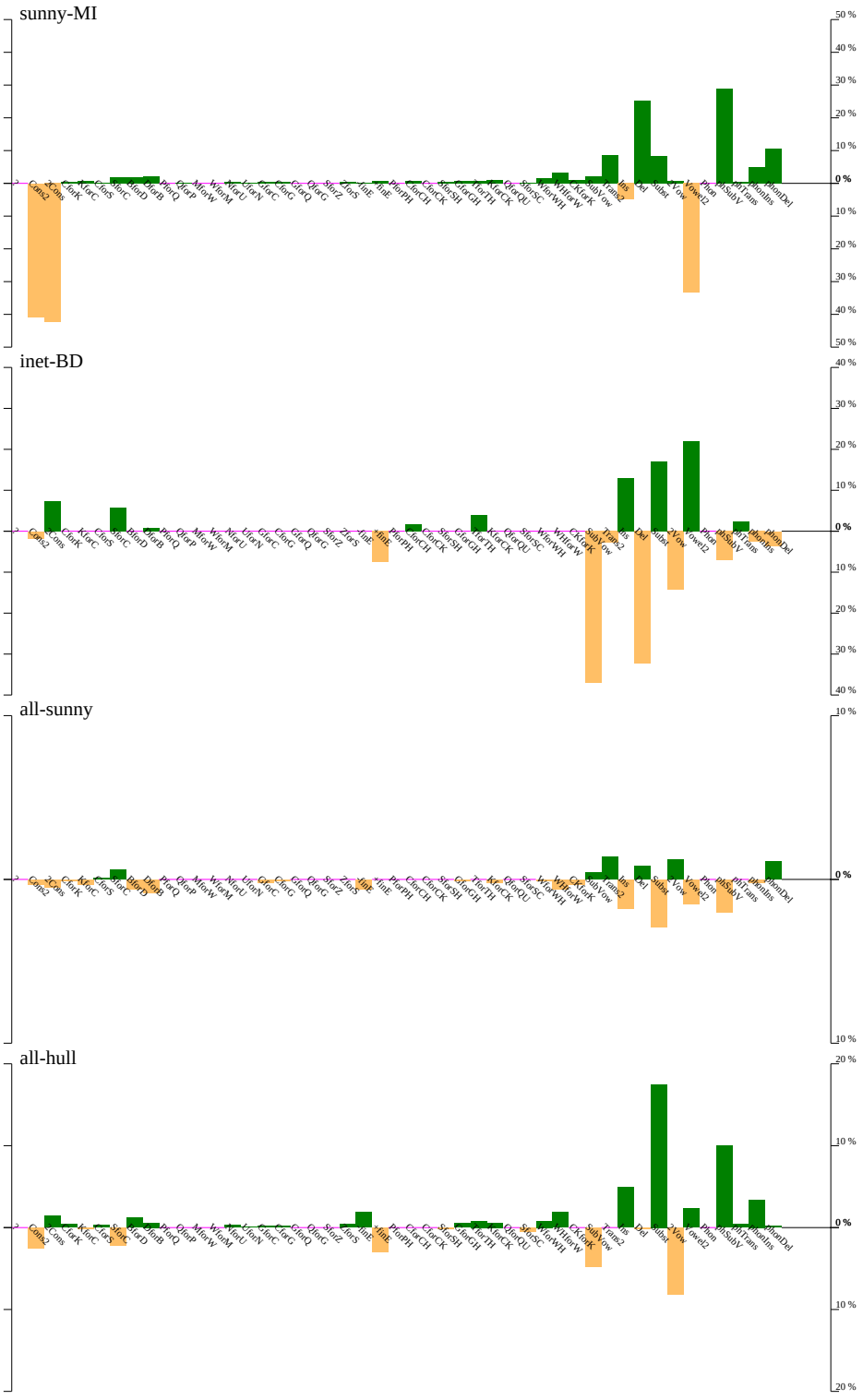

Figure 7.7: Rule usage differences between users and the populations from which they came. The top graph shows author 'MI' compared to the whole of population `sunny`. The second shows author 'BD' from population `inet`. The third and fourth graphs are for populations `sunny` and `hullstud` respectively when compared to all populations.

and those which should have been similar were not. This indicates that the authors were not making systematic errors which could be identified by Babel, and is discussed more in the following section.

7.6 Distinctive User Models

The user model is in the form of a list of weightings on the transformation rules. The consistency hypothesis (that any individual person consistently makes errors with recognisable patters) would suggest that the user model for one document by one user is significantly more similar to the user model for another document by the same user than to any user model for another user.

It was decided to compare each user model with each other; the result is shown in Figure 7.8 for members of the `sunny` population. The size of each square represents the difference between the two models in question, so a very small square indicates a good correlation, and a large one indicates a large difference (each square has been enlarged slightly to make the zero-size ones on the main diagonal visible). The darker the shading of the box, the more words have been used in building each model and hence the more reliable is the result.

The rows and columns of Figure 7.8 both indicate individual documents written by various people. The first two letters of the label of each row or column indicate the author, and the following number indicates which document is being referred to. At the intersection of a row and a column is a block indicating the correlation between the document defined by the row and the document defined by the column.

The correlation is measured as a distance between two points in a multidimensional space. If each error pattern rule is assigned a dimension then any particular user model can be represented as a point in that space. The distance between two points

is the difference between the user models. Figure 7.8 is based only on the frequency of application of rules, not on the position in words of errors or any other pattern.

These dimensions are first scaled so that each has the same maximum. Thus instead of one unit being one application of the transformation rule, it is proportional to the total number of applications of that rule. By this scaling, rarely used rules can still form an important part of the characterisation. This is slightly different from the scaling in Figure 7.7, above, where rules were scaled by the usage of the most common one.

An ideal result would show a small cluster of small squares near the diagonal for those documents written by the same user, and large squares elsewhere. This is clearly not the case for all, although some do show signs of it.

The number of documents written by each author varies simply because only the material collected for research could be used. Those with more material were welcomed and those with less were not excluded. Figure 7.8 includes work from the `sunny` population only, as it is the most reliable source of dyslexic text.

The results were compared in an Analysis of Variance (ANOVA) using the SPSS statistical analysis package. One ANOVA test was performed for each user, comparing all works written by them with all works written by others. That is to say, the input file to SPSS contained two columns and many rows. Each row was the vector distance of the comparison between the user models of two documents. One of those documents was by the author in question, and the other was either by the same author or another. The first column indicated whether the two documents were by the same author, and the second was the vector length. SPSS indicated whether the vector lengths were significantly different between the two 'populations' (The *Same User* population and the *Different User* population). This analysis was repeated separately for each user. A sample was never compared

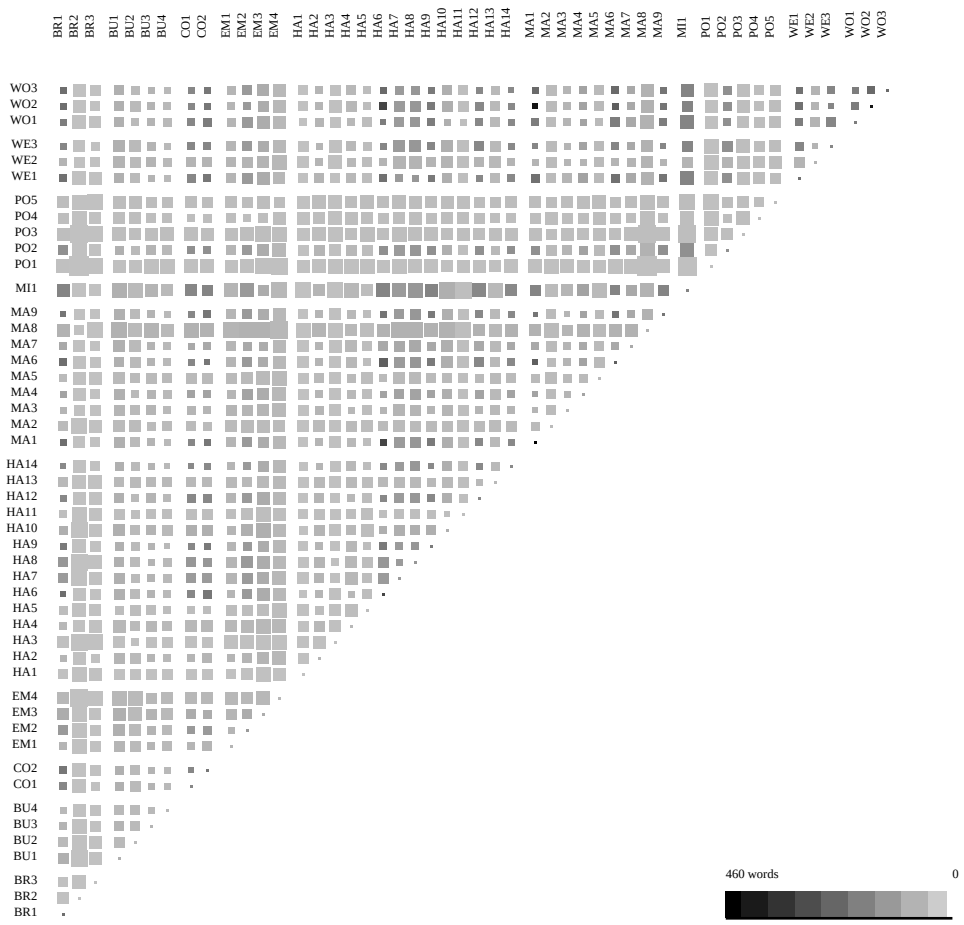

Figure 7.8: Differences between user models for handwritten documents by dyslexic boys in the sunny population

User	**same**	**p**
BR	6	0.68
BU	12	0.32
CO	2	0.047
EM	12	0.45
HA	182	0.0005
MA	72	0.0032
PO	20	0.55
WE	6	0.29
WO	6	0.12

Table 7.5: ANOVA significance of differences between user models of authors in the sunny population.

with itself, thus no zero-length vectors were included in the analysis. This may seem complicated but it was considered the only reasonable way of reducing such a complex data set to an ANOVA test.

In normal ANOVA studies a value of $p < 0.05$ is considered significant (with 95% confidence). In this case, however, one ANOVA calculation has been performed for each user, to establish if the vectors between samples by the same person were shorter than the vectors to documents by other people.

Because a significance level of $p < 0.05$ means that the result given has a 5% chance of being merely a coincidence, one can expect 1 in 20 such calculations to yield a falsely significant result. To remedy this, the significance level here is divided by the number of ANOVA calculations being performed, thus values in the table above can be considered significant if $p < 0.006$. By this measure, only 2 out of the 9 people have user models which are significantly different to each other based on the frequency of use of Babel's transformation rules. The significance measures are shown in Table 7.5 for the same data as that shown in Figure 7.8.

The column ‘p’ gives the probability that the observation that the author’s user models are significantly more similar to each other than to those of other people, is a coincidence.

The fact that two of them are different suggest that for some people’s errors the user modelling performed here is adequate. For the majority of users it is not. Considering the variety of dyslexic people this would seem to be a meaningful result. It also correlates to some degree with the earlier results of comparative performance for different author populations in which the more genuinely dyslexic populations showed a more positive improvement.

It was suggested in a private communication by Anthony Jameson (a prominent figure in User Modelling) that this ANOVA test is too harsh. It may not be necessary for all of the user models to be significantly different from each other for a worthwhile performance improvement to be observed. In this sense, the earlier results of Figure 7.4 are more interesting as they show the change in performance of Babel with and without the User Models.

7.7 Rule Use Position

Another possible pattern is the position within each word at which an error type is made. For example, someone may delete letters only from the ends of words, or mistake *b* for *d* at the beginnings and ends of words.

In fact, the rule use position system was implemented in Babel and from it the following results were obtained. However, it was not enabled in the version used for the majority of tests in this chapter because the sensitivity to small quantities of data made it unreliable.

Figure 7.9 shows a multitude of small graphs. Each one is the position within words

at which the particular error rule was applied, again only for the `sunny` population. This position is normalised to one of five positions. These represent the first letter (regardless of word length), early letters, middle letters, late letters and the last letter. This normalisation removes most effects of individual word length, although rounding errors combined with changing word length confuse some cases such as `-finE`, which should only occur at the end of a word.

Some rules can only be applied at certain places within a word either because of a linguistic constraint or because of the design of the software, also, some rules (such as `Phon`) do not have positions because they apply to the whole word.

The horizontal axis on each small graph represents the proportion of uses of that rule in which it was used at any particular position. Thus one large 'spike' indicates that all uses of that rule were at one particular position. A roughly flat line indicates that uses of the rule were evenly spaced through each word.

As with the overall user model comparison the information in Figure 7.9 was checked in SPSS for significant differences in similarity between documents by each user as compared to documents by other users although the actual analysis was different. In this case a multi-factor ANOVA was prepared with two categories of trial: documents by the same user and documents by different users. For each trial, which represented one of the small graphs in Figure 7.9, the 5 position frequencies were recorded. SPSS was used to ascertain whether patterns for these 5 position frequencies were significantly different between the two categories. This analysis was performed once for each rule for each user.

The results of the ANOVA show, as expected, that in only a few cases the rule positions are interestingly different between users, so would be of use to a spelling checker.

For some rules the applications are so infrequent that a "significant" result is found,

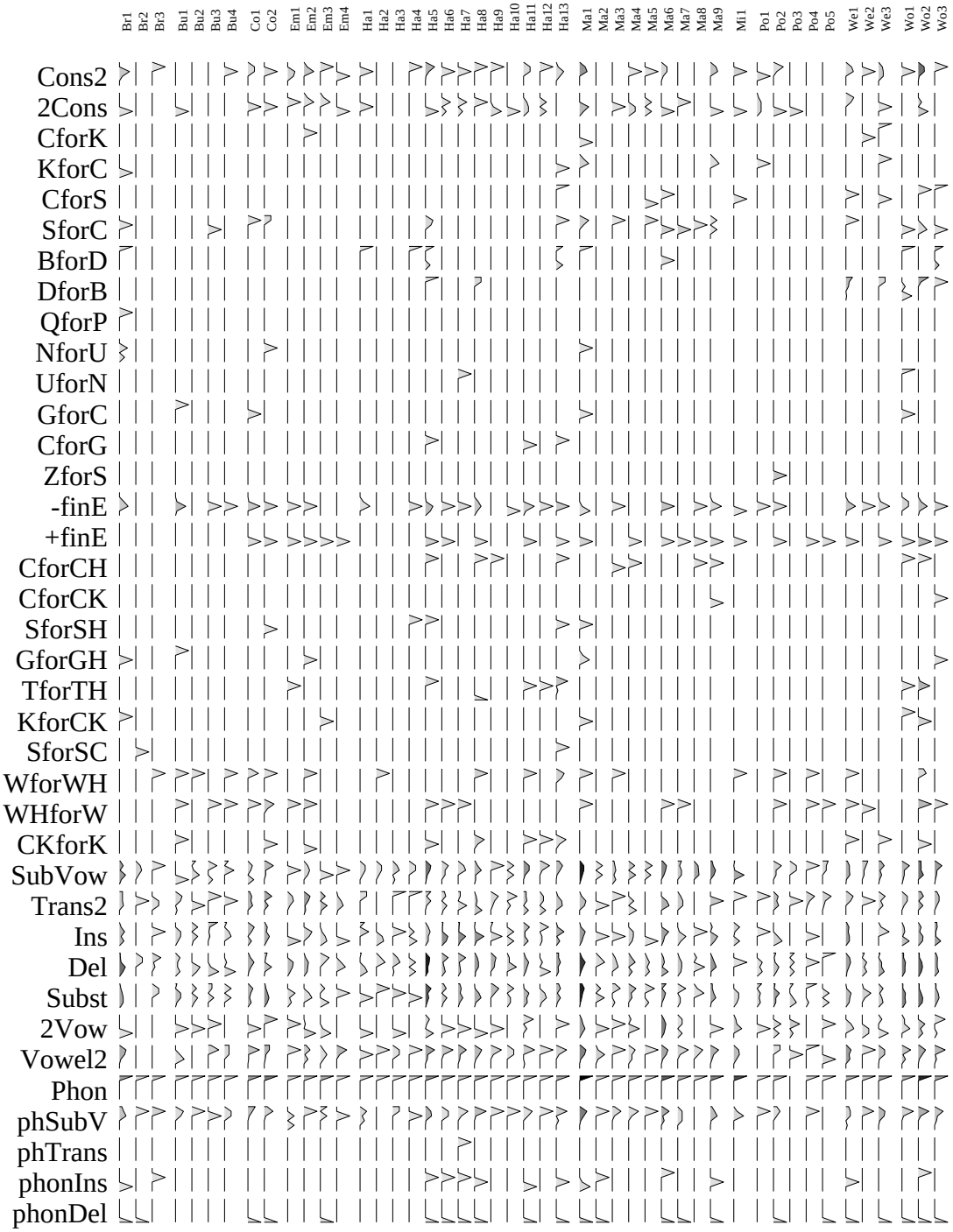

Figure 7.9: Rule use position within words for handwritten documents

User	**Rule**	p	**Comments**
BU	SubVow	0.03	Evenly distributed; others are biased to start
BU	Del	0.001	Towards end; others are more even
EM	2Cons	0.022	Earlier than most (pos=2)
EM	SubVow	0.001	Middle of word; others are more towards start
EM	Ins	0.044	Towards end; others are nearer middle
EM	phSubV	0.006	Towards middle; others are at start
PO	Subst	0.005	Later than most
WO	BforD	0.004	Start and end of words
WO	DforB	0.004	Start and end

Table 7.6: Selected ANOVA results for rule-use positions. Although several are significant according to the tests, the patterns are not widespread.

but is not reliable because of the small number of raw data points used. For example, all applications by user CO of the rule `NforU` were at position 2 (the middle of the word). However, that was only one occasion. (The shade of grey on the diagram is intended to indicate the number of words supplying data to the result, but it is not very clear). On the other hand, all uses of `GforC` by user CO were at position 3, and that was a total of 4 words which is more useful. Table 7.6 shows a number of interesting rule-use position patterns from authors in the `sunny` population.

Many of the results are significant even at the $p < 0.006$ level (as before, the significance level has been set at $0.05/n$ for n people). A first response may be to say that they are not significant because there are so many comparisons being done. In this multi-factor ANOVA, however, there is no overlap between rules or between application positions within words. The only challenge to significance is from the number of people, for each of whom an ANOVA is being performed.

It is also interesting to note that some documents contain widely distributed errors

for the wildcard rules (Ins, Del, etc) rules. For example all documents by the author WO and MA1, BR1, HA5.

There is less data relating to individual positions within words than to rule usage in general. Where many rules are only applied half a dozen times within a document, there may only be 1 or 2 occurrences in a particular letter position. For this reason the rule use position feature was excluded from most other analyses.

7.8 Discussion

The performance of Babel as a simple spelling checker was not impressive. Its purpose was to establish the usefulness of user models which no other current spelling correction system has. The results showed, disappointingly, that there were only weak arguments in favour of such models using the current system. In a simple and fair test it failed to significantly improve performance when the user model was trained on some errors by each author and then tested on other errors by the same person.

Such a result does not in fact disprove the value of user models in all aspects of spelling correction. It shows that the methods used here are not suitable for normal use. Rule Use Position was excluded from the result in Figure 7.4 and might reasonably be tested separately if more data were available. With the current samples, so few occurrences of each rule at each position have been observed that results are unreliable.

When compared to `ispell`, the popular but conventional spelling checker for Unix computers, Babel was found to present the correct word significantly more often (in most cases, which included the important `sunny` population) but there was no significant difference in the position of the word on the list. If anything, `ispell`

presented the correct word nearer the top of its list. Babel's failure to demonstrate clear superiority over such a conventional program is disappointing.

SPEEDCOP performs unremarkably by comparison to Babel. This is largely because it was designed for scientific text rather than dyslexic writing and so is operating outside its specification. That design choice made the task substantially easier for the authors of SPEEDCOP since drafts of scholarly papers contain many long words with few possible alternatives when they need to be corrected, unlike dyslexic writing. Babel was significantly better at presenting the correct word for some key populations, and significantly better at putting the correct word near the top of the list for some. Babel outperformed SPEEDCOP in both respects for the `sunny` population, which is perhaps the most reliable source of dyslexic errors. The significance of these improvements is shown in Table 7.2 on page 165.

The error pattern rules tested here are perhaps inadequate. A more sophisticated set which considers the origins of spellings either through orthography or modern primary education might illuminate the source of complex errors. One example is the word 'charlatan' being misspelt as 'charlotten' by a girl who already knew the girls' name 'Charlotte'. Such complexity is beyond the scope of this work, and is probably intractable.

The rules have very limited interactions with each other and with other parts of the word in question as they are currently defined. This might be improved but again it would be difficult to ensure accurate and comprehensive coverage of the problem.

Roger Mitton, the author of a clear and detailed account of computerised spelling correction (Mitton 1996), suggested in a private communication that the variations within an individual's errors are likely to be greater than those between people. This suggestion is supported by the fact that Babel failed to improve significantly

after training on part of a text sample.

It is also possible that, rather than forming individual patterns, the mistakes of poor spellers can be grouped into categories. Thus one might have phonetic vowel-error spellers, letter transposition people, and suffix deletion people. Again, such an investigation is beyond the scope of this research.

Chapter 8

Approximate String Matcher

ALGORITHM T

Every spelling checker relies on some method to evaluate the similarity of a given text string to the words in its dictionary. The majority of this thesis is concerned with the dimensions of the similarity. A sacrifice of speed has been made in Babel for the intended improvement in accuracy. The work in this chapter however, considers a simpler edit-cost based similarity measure referred to here as ALGORITHM T which is both accurate and fast.

8.1 Background

Edit cost measurements, introduced most usefully by Wagner and Fischer (1974), find a sequence of changes which can be made to one string to convert it into another. Each of the allowable change operations has a cost associated with it, and a number of different combinations of operations might lead to a correct solution, but not all will have minimal costs.

The changes allowed by different algorithms vary, but generally include insertions, deletions and substitutions of individual symbols. The transposition of two symbols is also sometimes considered. In the most versatile applications, each operation has a different cost according to the symbols being operated on, thus $\gamma(c \rightarrow \Lambda)$ might represent the cost of deleting 'c', whereas $\gamma(d \rightarrow b)$ could be an entirely different value, for converting a 'd' into a 'b'.

More simplistic algorithms treat each of the editing operations as having a single cost regardless of the symbols involved, thus W_I might represent the insertion of any symbol, W_C the changing of any one symbol to another, and W_D the cost of deleting any symbol (Lowrance and Wagner 1975). Some researchers have published a number of algorithms which go a step further and consider 'unit-cost' editing operations, in which $W_C = W_I = W_D = 1$ and thus the edit cost, k, is also a count of the number of operations. This has been called the 'k differences problem'.

Finally, some algorithms consider only substitutions; $W_C = 1,\ W_D = W_I = \infty$. Such algorithms are not suitable for spelling correction as insertions and deletions cannot safely be ignored, although the methods are useful to other situations where substitutions are the only possible error.

ALGORITHM T, presented below, does not assume unit cost operations, although it has been tested in such a configuration to make it more comparable with other algorithms.

The task most commonly solved by the ever-faster algorithms published is the k *differences problem*. The algorithms take a long text T and a short text P (the pattern string) as well as a threshold cost k, and return all substrings of symbols $t_i..t_{i'}$ within the text T such that the edit cost $d(P, t_i..t_{i'}) \leq k$. Some of the algorithms fundamentally require that $W_C = W_I = W_D = 1$ while others use it merely as a working assumption to simplify discussion.

The algorithm presented below, ALGORITHM T, solves a slightly different task from the k differences problem which is more relevant to spelling correction tasks. It takes a series of short input texts $T_1..T_n$ (which can be considered informally as n words from a dictionary) and a pattern string P, and returns those input texts T_i at which the edit cost is minimal: $\exists i \, \forall j \quad i, j \in \{1..n\} \ \wedge \ d(P, T_i) \leq d(P, T_j)$

8.1.1 Dynamic Programming

Using dynamic programming (a particular class of fixed-time algorithm), the lowest-cost sequence of changes can be found. The basic algorithm (which is presented on page 32) runs in time $O(nm)$ for two strings of length m and n. For a spelling correction task, this might involve 50,000 string comparisons where $n = m = 6$ or 1,800,000 calculation steps which would normally be too slow.

Given two strings as input, the original algorithm works through each possible pair of character positions (i, j) from the two strings A, B finding the lowest cost way of converting the substring $A_1 \ldots A_i$ into the substring $B_1 \ldots B_j$. At each stage the cost of reaching the positions i, j is stored in array element H[i, j] and is defined as the following recurrence relation:

if $A_i = B_j$ **then** d:=0 **else** d:=W_C ;
H[i, j] := min(H[i-1, j-1] + d,
H[i, j-1] + W_I,
H[i-1, j] + W_D) ;

In the original edit cost method, each edit operation had a different cost depending on the characters involved in the operation. For spelling correction the deletion of the letter *k* might have a higher cost than the deletion of *a*, but for optical character recognition substitutions are affected by the similarity of appearance of letters and insertions and deletions are both rare.

Many hybrid algorithms set a maximum cost, k, and preprocess the dictionary to remove unlikely candidates. Some of these are based on strings which are so different that they cannot produce a sufficiently low cost transformation. Others employ spelling-error heuristics such as the assumption that the first letter of the word will be correct. The work of Du and Chang (1992) is reasonable in that it sets a limit for k and performs the search, then repeats the search with a higher limit if insufficient matches were found after the previous pass.

More recent work in biological sequence matching has led to a number of improved algorithms which perform well under typical conditions. The task, for most biologists, involves identifying a recently collected sample of DNA or protein.

There are databases available of protein and DNA sequences which have been found with millions of 'bases' stored. When people sequence new DNA fragments they naturally want to find if the same sequence occurs elsewhere, and so compare their new data with the database.

DNA sequences are made up from a symbol alphabet of four bases; A, C, G and T. Protein sequences are made up from 20 amino acids; C, S, T, P, A, G, N, D, E, Q, H, R, K, M, I, L, V, F, Y and W by using three bases per amino acid with a number of other three-base codes reserved for sequence start and stop codons. Each protein, which is equivalent to one gene, typically has between 300 and 1000 bases although some, such as the gene for muscular dystrophy, have tens of thousands of bases. Because sequencing techniques become less accurate with longer samples, they are often used iteratively on fragments of a sequence and stitched together later. Typical databases contain millions of bases, and new sequences being compared may require dozens of editing operations (deletions, substitutions, insertions) to find a match.[1]

[1]My thanks to George Russell at the Roslin Institute who provided much of the information in this paragraph.

Sometimes the sequencing machines make mistakes, and record the wrong symbol in their tables. Even more often, the sequence which they are given is incomplete, and so represents a substring of the entire actual sequence. Perhaps more interestingly, some sequences are genetic variations of others and so differ, perhaps substantially, from those stored on the databases.

Most of the recently proposed algorithms consider only the k differences problem and employ substantial data structures, running times, and working memory consumption. A popular data structure used recently is the *Suffix Tree*. A suffix is a substring of a text which begins at any point and continues to the end. The word ‘tree’ in this instance is equivalent to ‘trie’, being a tree with as many children from each node as there are symbols in the alphabet.

A suffix tree is a way of storing a string of symbols so that any valid substring can easily be matched. To build one, the entire text is built into a linked list of symbols, which becomes the first branch of the tree. Then the first symbol is removed and the process repeated. If the current string begins with the same symbols as an existing part of the tree, the same nodes are used and a branch added where the strings differ. Figure 8.1 shows a suffix tree for the word ‘bananas’.

A development of the suffix trie is to compress linked-list chains into single nodes with several symbols. This assumes that your data structure for a single node can store several symbols more cheaply than it can store several nodes in a long chain. Processing of this compression is slightly harder than uncompressed tries, but is still fairly easy. One implementation of a suffix tree (Nelson 1996) kept a copy of the original text string (the dictionary or a DNA database) and stored an index for the start and end symbols represented by each node. This allowed any length of compressed sequence to be stored in the space of two integers, if the entire text is already stored.

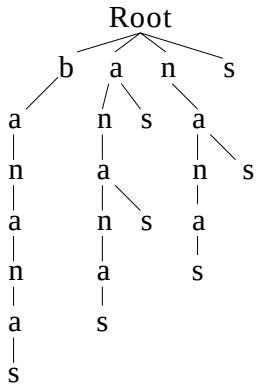

Figure 8.1: A simple suffix tree for the word 'bananas'. It contains a path from the root node for every suffix of the text.

In spelling correction work, Pain (1985) assigned specific costs to certain letters and folded some sequences of letters into single characters so that they could be given special lower costs (some of these sequences involved the beginning or end of a string, so assigning positional costs). This would be easy to encode in a suffix tree. On the other hand, Mitton (1996) assigned different costs to operations on certain letters in certain dictionary words as an approximation to expected spelling slips or phonetic errors (for example silent letters were flagged as more likely to be omitted). Because this treats different dictionary words differently, it would not be possible to encode in a suffix tree which merges words with the same prefixes. On the other hand, Du and Chang (1994) proposed an efficient dictionary search for the *k differences* problem which does not merge the storage of any words.

8.1.2 Dictionary Filtering

The original edit cost algorithm has a run time of $O(|A| \times |B|)$ (where $|A|$ and $|B|$ are the lengths of the input word A and the dictionary candidate word B respectively) which might be rephrased as $O(n^2)$ for words of n letters. To find

the 'nearest neighbour' using this and a dictionary of correctly spelt words would require comparing the user's input word to each of the dictionary words one after another. This is generally seen as unacceptable for dictionaries of many tens of thousands of words.

Some people have tried to guess likely errors. For example Pollock and Zamora (1984) built 'skeleton keys' to improve the first guess in their SPEEDCOP system. Their task was to correct single-error typing mistakes in scientific reports, mainly about chemistry. Their method is based on the creation of keys derived from the input and dictionary words. To build the key the first letter of the word was retained (under the assumption that it was correct), then all vowels were moved to after the consonants and any letter repetitions removed. The keys for correct words were sorted alphabetically, and any test word was converted to its key and then looked for by binary searching the list of keys.

It was found that viable alternative words were mostly found at the binary search target position, and if not there, it was normally within a handful of places; these were identified using a simple method which established if the string could be edited with at most one operation[2]. The first such entry found was proposed as a correction. If one was not found within 50 entries the search was performed again with a modified key which took account of likely consonant deletions, and abandoned if no match was found. Clearly a better match might exist further away in the dictionary (for example if the first letter were changed), but could be missed by this method. The authors found that it worked quite satisfactorily for their task which involved skilled writers making single-error slips in long words.

[2]It is widely believed by computer scientists that 80% of misspellings require just one of the classic editing operations to be corrected, as stated by Damerau (1964). That statement was based on mechanical errors and mistakes by skilled typists. The corpus used in the experiments in Chapter 9 have up to 6 errors each.

Another popular method is to filter out words from the dictionary on the basis of a faster metric. If one is searching for words within one edit operation of a given string, only words with the same number of letters or one more or one less could possibly be acceptable (Du and Chang 1992). Du and Chang (1994) take this further and precompute lists of words matching wildcard masks. For a given word and an assumed maximum number of errors, all possible wildcard positions are found and then the appropriate set of masks selected. They also abandon edit cost calculations if the cumulative cost at any position (i, j) is more than the acceptable maximum. If no match is found, they fetch the precompiled lists for words with a larger number of errors and begin the search again.

Owolabi (1996) tried a number of methods for dictionary partitioning with varying success, but each of which had limitations in accuracy. The most efficient search, covering 0.6% of the dictionary, was an index of word halves. If only one error exists then either the first half or the second must be correct and hence a search of two linked lists, one of each half of the written word, must find the correct match. However, this still requires around 900 calculations of $H[i, j]$ compared to the 300 or so presented here, and cannot guarantee a match involving more than one error.

Mitton (1996) used a variant of *Soundex* (Russell 1918) to select a set of dictionary words that would probably contain the correction. Like SPEEDCOP above, this does not guarantee to find the nearest neighbour but does achieve a high success rate with typical errors.

Greene (1994) used an associative memory to store 'feature vectors' which relate to words from a dictionary. The features included the presence of each letter or each possible pair of letters. A neural network converted such a vector (from the user) to a hash value; each stored vector with that hash value or any within a limited Hamming radius was compared to the input vector. Because of the nature of associative memories and the hashing network, it is not possible to guarantee a

best match without searching the whole dictionary.

8.1.3 On-Line Searching

An on-line search for an approximately matching string is in some sense the opposite of what is done in spelling correction. The pattern string P is known in advance and can be pre-processed, and the main text T is not and so must be processed one symbol at a time and in one pass as it cannot be recorded and re-analysed. This is useful for functions like the Unix `grep` command, and the field includes a number of worthy ideas.

Baeza-Yates and Perleberg (1996) included an undoubtedly fast and simple algorithm, but which worked only on substitutions of symbols and not on insertions and deletions. As such, it is not directly relevant to the work here. It worked by first recording the index into the pattern string P at which each symbol occurred, and then scanning once through the main text T. For each symbol t_i it found the position in P at which that symbol occurred (which we may call P_s) and then incremented a counter for T_{i-s}. If a sequence of correctly matching symbols were found, the same counter would be incremented many times, finally passing a threshold which represented an acceptable match.

One of the same authors went on to write another paper in the same year (Baeza-Yates and Gonzalo 1996) to perform approximate string matching including insertions and deletions as well as substitutions. However, the algorithm limited itself to on-line searching in which the main text T was not known in advance. This means that the minimum run-time for the algorithm is o($|T|$) even if an exact match is to be found; this is much longer than the typical run-time of ALGORITHM T. The operation of the algorithm is interesting, however.

It built a state machine from the pattern P as a rectangular grid of nodes, starting

in the top left corner. For each symbol successfully matched, the node to the right of each active one was activated. For each symbol not matched, the nodes below and below-right were activated. The number of rows can be set to correspond to the maximum allowable edit cost k, with a match being reported when any node in the last column of the automaton is activated, meaning that the last symbol of P has been reached.

The authors then observed that every node beyond a threshold position on each diagonal would be active at all times because insertions or deletions could be used even where they were not necessary. The only information that needed to be stored was the threshold position for each diagonal after which each node was active. This could, for limited cases, be represented completely in a single (32 bit) word of a computer's memory and be operated on quickly. A method for dividing larger problems so that they fitted into the limited case was also presented.

Despite its appeal and efficiency for on-line processing, the work is not relevant to spelling correction because it assumes that P is constant and T is not known whereas we know the dictionary T in advance but change the pattern P regularly.

At the same conference, an improvement of the earlier paper for k mismatches was presented (El-Mabrouk and Crochemore 1996). This again uses carefully constructed bit representations to fit within a computer processor's word size, and operates with as few as $3n$ operations from end to end. However, this again is neither suitable for spelling correction nor faster than ALGORITHM T when a close match is found.

8.1.4 Suffix Trees

Tries and Suffix Trees (introduced on page 191) have also been used for approximate string matching in recent years. Ukkonen (1993) proposed three algorithms for

approximate string matching over Suffix Tries. In essence he executed a full dynamic programming match over the main text one symbol at a time, but stored partial matches from previous symbols and re-used the results where they were identical. The algorithms produced all substrings matching with an edit cost $\leq k$. The first required a working memory space $O(qm)$ where q is the number of viable substrings of the main text which may lead to matches given any current stage of calculation, and m is the size of the pattern string being matched. The run time was $O(mq + n)$ where n was the size of the main text.

Jokinen, Tarhio, and Ukkonen (1996) used suffix automata instead of a suffix tree. These are state machines which recognise the suffices of the text rather than a tree which stores it, but operate similarly. The algorithm searched for all matches with a cost $\leq k$, requiring a working memory sufficient to run a 'modified' Dijkstra's Shortest Path algorithm, presumably $O(m)$. The run time is $O(|P||T|)$ or $O(k|T|)$, with a best case of $O(|P|)$. The method presented below consumes a similar run-time but less working memory.

Shang and Merrettal (1996) also present a tree based approximate match search with a clear description of its application to spelling correction, which is missing from most other work. Their method is very similar to the trie searching here, and mentions many of the same economies to be made such as the omission of repeated calculation of the edit cost array $H[\ ]$ for common prefixes of several dictionary words, and of the opportunity to abandon searches down branches following non-viable prefixes.

Shang and Merrettal also consider, in passing, the application of non-unit-cost editing operations and the refinement of the required edit cost k to find the best match. Their treatment of the transposition of two consecutive symbols seems appealing at first sight, but the authors failed to consider the restriction on the cost of a transposition; unless $\min(W_I, W_D, W_C) \leq W_T$ a search might be abandoned

before the viability of a transposition is considered.

Interestingly, Shang and Merrettal did not make any effort to optimise the search order of child nodes, preferring instead to pursue them in alphabetical order.

Cobbs (1995) uses Suffix Trees to find approximate substring matches in a manner which is, in practice, quite similar to that used here. It refers to triangles in the array $H[\,]$ which need to be calculated, based on viable prefixes. Also, through the use of a suffix tree which stores values of $H[\,]$ the method avoids duplication of calculation. The method requires working memory $O(q)$ where q is the presumably large number of viable prefixes, and puts a greater emphasis on the single-symbol shift through the text T than is appropriate for spelling correction.

The single symbol shift is part of typical substring matching algorithms. They compare the pattern P to a substring of T, $t_1..t_{|P|}$. Following that, the pattern P is shifted one symbol along T and compared to $t_2..t_{|P|+1}$. The second search, starting at t_2, may require less processing as some information will already be available from the previous search. However, for spelling correction applications there is no such overlap between searches. The pattern P is compared to the entirety of the first word, T_1 and next compared to the entirety of T_2. There is not necessarily any similarity between T_1 and T_2.

Yet another paper (Andersson, Larsson, and Swanson 1996) at the conference on Combinatorial Pattern Matching in 1996 presented an apparently new data structure, the *word suffix tree*. Although it looks initially appealing it employs some rather strange methods and finally ends with a suffix tree which stores every sequence of words in the long text T, where a word is a sequence of symbols delimited by some indicator. It stores full copies of the text T, a trie holding each word and a more conventional suffix tree where each node holds a reference number representing a particular word. Although this may be of use in operations

above the word level, it is not relevant to spelling correction.

Much work on approximate substring matching emphasises the work that does not need to be repeated in consecutive steps through the main text. However, dictionary matching is concerned with finding a whole sub-text (word) with the lowest edit cost; each match must begin at the beginning of such a sub-text and not overlap the boundaries. Thus many economies of substring matching algorithms are effectively voided. Because of this, it is important that the original spelling correction application of approximate string matching be considered in its own right rather than as a subsidiary of DNA sequencing. ALGORITHM T does this.

8.2 Advantages of a Trie

Tries are tree structures where each node fans-out to up to $|\Sigma|$ child nodes for an alphabet Σ which contains $|\Sigma|$ symbols. Pain (1985) used one for substitution of sounds in a phonetic dictionary, and a number of recent approximate string matching methods have used them, as described on the previous pages. Trees occupy more working memory than lists but often afford better access times and easier modification.

The dictionary for ALGORITHM T, presented below, is structured as a tree of letters. Each node has one child for each possible subsequent letter. A path in the tree from the root to a leaf represents a correctly spelt word (see Figure 8.2). Unlike other suffix tree methods, a match cannot be reported unless a node has been reached which holds a flag indicating that it represents the last letter of a word.

For comparison with other approximate sub-string matching algorithms, it may be more helpful to consider this spelling-correction task as one of approximate whole-string matching over many texts.

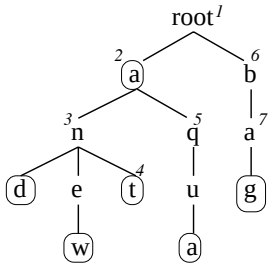

Figure 8.2: An example tree for a small dictionary, numbered with the order in which nodes are visited when searching for the string 'amt' in the dictionary of words: `a, and, anew, ant, aqua, bag` . Each node in a box represents the last letter of a real word.

In this application the use of a tree is apt because the basic edit cost method operates on substrings from the beginning of a word to a certain point; by using a tree, that calculation is done simultaneously for all strings beginning with the same letters. That is the basic premise of this work.

Where a substitution is to be considered, it is immediately clear what alternative letters are available in a tree; each is linked as a child of the node representing the previous letter. The popular `ispell` spelling checker, by comparison, considers substitutions and insertions by searching for each letter separately. Although its hash-table based dictionary lookup is fast, one cannot help believing that improvements are possible.

Exact dictionary look-ups are fast in a tree; they take $O(|P|)$ iterations for a word P of length $|P|$, whereas a binary search takes $O(m\log_2(n))$ for an n word dictionary and for some $m \leq |T_i|$ for each candidate dictionary word T_i.

The memory consumption of a tree is larger, however, than a simple list. This may have dissuaded earlier researchers from their use but with modern hardware, finding several megabytes is no longer a problem. The simple tree used in experiments described in Chapter 9 contained 25,000 words (205Kb) and occupied 1.2Mb

and the (optional) corresponding sorted list needs 0.4Mb. The edge compression discussed later brought this down to 435Kb for the trie with no sorted list. This grows sub-linearly with the size of the dictionary in words.

8.3 Pseudo-code for ALGORITHM T

To illustrate ALGORITHM T in more detail, it is provided here in the form of Pascal-style Pseudo code, and is described in prose below. The actual algorithm is implemented in C and contains a number of enhancements which are discussed later such as edge compression and linked lists of child nodes.

```
program ALGORITHM T

type string = array of char

record tnode :=
{   l : char
    child[|Σ|] : array of ↑tnode
    realword: ↑string
}
global P : string
global bestcost : integer
global root : tnode
global W_I, W_D, W_C : integer
global bestmatches : set of string

procedure SearchTree(
    ↑node : tnode ,
    ↑H_prev[ ] : integer ,
    depth : integer
    )
local H[ ] : array of integer
local mybestpos : integer
local mybestcost : integer
local mybestcharset : set of char
local b,d : char
local i,c : integer
{   if depth = 0 then
        for i := 0 to |P|
```

```
            H[i] := W_D × i
    else
    {   for i := 0 to |P|
        {   if i>0 then
            {   if node↑l = P[i − 1] then c := 0 else c := W_C
                H[i] :=min(H[i − 1] + W_D, H_prev[i − 1] + c, H_prev[i] + W_I)
            }
            else
            H[0] := H_prev[0] + W_I
        }
    }
    if node↑realword ≠ null then
        if H[|P|] ≤ bestcost then
        {   if H[|P|] < bestcost then
            {   bestcost := H[|P|]
                bestmatches := ∅
            }
            bestmatches := bestmatches ⋃ node↑realword
        }
    mybestcost := H[0]
    mybestpos := 0
    mybestcharset := P[1]
    for i := 1 to |P|
    {   if H[i] ≤ mybestcost then
        {   if H[i] < mybestcost then
            {   mybestcharset := ∅
                mybestcost := H[i]
                mybestpos :=i
            }
            if i < depth then mybestpos := i
            mybestcharset := mybestcharset ⋃ P[i+1]
        }
    }
    if mybestcost ≤ bestcost then
    {   b := P[mybestpos]
        if node↑child[b] ≠ null then
            SearchTree(node↑child[b], ↑H, depth+1)
        for ∀d ∈ mybestcharset
            if d ≠ b and node↑child[d] ≠ null then
                SearchTree(node↑child[d], ↑H, depth+1)
        if (mybestcost < bestcost) then
            for ∀d ∈ Σ
                if d ∉ mybestcharset and node↑child[d] ≠ null then
                    SearchTree(node↑child[d], ↑H, depth+1)
    }
}

{   load_dictionary(↑root)
```

```
    W_I := W_C := W_D := 1
    read_input(P)
    SearchTree(↑root, null , 0)
    print_output('minimum cost = '.bestcost)
    for ∀P ∈ bestmatches
        print_output(P)
}
```

8.4 Operation of ALGORITHM T

The algorithm listed above and described below finds the dictionary entry or entries which have the lowest edit costs, compared to a given pattern. The worst-case run time of the algorithm is equivalent to applying the naïve edit-cost to all dictionary words, $O(nm)$, but in practice is typically very efficient. Its working memory requirement depends only on the length of the pattern string: $O(|P|)$ whereas other algorithms require substantially more.

8.4.1 Initialisation

First a tree is constructed from the dictionary. This is not shown in the pseudo-code except as a call to the function load_dictionary(). A file is opened containing the correctly spelt dictionary words which are then read one at a time. Each word is examined one letter at a time, and the tree descended correspondingly from the root node. If a child node is found with the same letter in the given position, that node becomes the current one. If not, a new child node is created for that letter. When the end of the word is reached, a link is made from the current node to another record containing other information about the word. Currently this includes only the spelling as a normal string (which could be inferred from the tree position) but frequency information or phonetic equivalence might be recorded here.

The tree used may also be called a trie. Each node will have one child for each

symbol which can follow that position, up to a maximum of the alphabet size. In some respects it is similar to a Suffix Trie in that there is a route from the root node to leaf nodes representing many parts of the original dictionary. However, rather than representing every possible starting point within the original text, only the beginning of each whole word is represented.

The nodes of the tree are represented (in the prototype code) using a structure containing a pointer to the first child, the next sibling and to the real word record. Any of these can be null. It uses a linked list rather than an array because 95% of nodes have 2 children or fewer and so this saves space for an alphabet of hundreds of symbols. The time to search a node for a certain child is proportional to the length of the number of children it has, but this is only slightly slower than having an array of all possible children.

The tree building process has no particular maximum size, nor need the words be sorted before loading. It is assumed that only one word has a given spelling. If this were not so (for example if the dictionary were of phonetic pronunciations, and the homophones were of concern), a linked list could be used in the real word record.

To save space, edge compression can be used as it is in other Suffix Trees. This involves finding the nodes with only one child, and encoding the symbol for the child within the parent node. Section 8.5.1, below, discusses this. It has been implemented and found roughly to halve the size of the data structure.

Figure 8.2 on page 200 shows the tree that would be built for the dictionary A, AND, ANEW, ANT, AQUA, BAG. The nodes representing real words are boxed. The italic numbers represent steps of processing discussed in the text below while searching for the string AMT.

The size of the data structure has an upper bound of $O(n)$ for a dictionary with n symbols, regardless of the number of words. This is because the tree building part

of ALGORITHM T can construct at most one node per symbol processed. In practice, Figure 9.1 shows that the space used grows sublinearly.

8.4.2 Search

Having been initialised, ALGORITHM T is then given a letter string which may be misspelt. The task is to find the correct words with the lowest edit costs. In the prototype, all words with equally minimal cost are found. With trivial modifications this might be extended to the n correct words with lowest edit costs. This task is rather different in emphasis from the 'k differences problem', where all strings with an edit cost of k or less are found. Also, a match is only considered to have been found where a node has a real-word structure attached whereas other algorithms happily break substrings at any position.

The search employs a recursive routine, `SearchTree`. This is entered with a pointer to a node in the tree, a pointer to the costs array from the previous level, and a counter for the current depth of recursion. On entry, a local array is created to store editing costs. The local variables within the routine are the only additional working-memory requirements, and in total are proportional to the length of the dictionary strings T_i searched. This compares favourably with other search routines, some of which require memory for each 'viable substring' of which there may be many.

The routine first calculates one column of entries equivalent to the array $H[i, j]$ where i loops from 0 to $|P|$ and where j is the current depth of the recursive search. This uses the recurrence relation given on page 189, and may read values from the cost array of the previous depth (for substitutions and deletions) or from the previously calculated value in this array (for insertions). The algorithm does not currently support transpositions, but they could easily be added subject either to a

minimum cost or an increase in processing time for look-ahead calculations.

This array $H[i, j]$ stores the costs of generating the letters of all the parent nodes in the tree (including the current node) while consuming i letters from the input string, for all valid i. Only column j is stored locally to the routine, and column $j - 1$ is available for reading. For the example of finding a match for 'amt', shown in Figure 8.2 at step 2 (the letter 'a') $H[1, 1]$ would record a cost zero to consume the letter 'a' from the input while producing the same on the output, and would record a cost of W_I for inserting a letter (on the output) while not consuming the input 'a' in array element $H[0, 1]$. The remaining elements, $H[2, 1]$ and $H[3, 1]$ would be $0 + W_D$ (for deleting the 'm') and $0 + 2W_D$ (for deleting 'mt') respectively.

If the tree node contains a valid pointer to a real dictionary entry that implies that there is a derivation for a word in the dictionary. In that case the edit cost for converting the input string into that word can be read from $H[|P|, j]$. If that cost is equal to the best previously found, the new word is added to the list of best matches. If it is lower, any previously found matches are discarded and this word is made the only member of that list[3].

In Figure 8.2 at step 2, the node 'a' does represent a real word, and so the edit cost of converting 'amt' into 'a' can be read from $H[3, 1]$. For the simple case of $W_I = W_D = W_C = 1$, this cost is 2 which becomes the best real-word match. Later, at step 4, the word 'ant' is found to have a cost of 1 and so replaces 'a' as the best match.

Whether or not a match has been found with a dictionary word, the algorithm now considers whether to continue its search. That is to say that there may be other children of the current node which may yet yield lower cost paths. The local array of edit costs ($H[0..|P|, j]$) is consulted. If the lowest of these costs is higher than the

[3]Another possible change would be to keep a fixed number of near matches instead of just the lowest cost. In that case the comparison here would be with the worst of them.

best match yet found, the search is abandoned at this level. While examining the array, the characters $P[i+1]$ for the lowest cost values of i are recorded because they represent the characters most likely to lead to an optimal solution. In the example the lowest cost is $H[1,1] = 0$ which is strictly lower than the cost of the best real-word match found so far. The following character $P[2]$ =m is recorded.

ALGORITHM T will now recursively call itself for a deeper level of the tree. The choice of which letter to use is important to the final run time of the system, and is unique to this new algorithm. If the earliest alphabetical letter were always searched first, words like 'aardvark' would be found first even though they are probably not close matches, so a better method must be found. If a single position i has the lowest cost, the child representing the letter $P[i+1]$ is searched first. If several positions have equally low costs, the position nearest to the current depth of search is used. The order of preference is to follow a diagonal from the lowest cost position so far; to follow other lowest-cost positions tending towards the main diagonal, and finally to follow any other child nodes. In the example at step 2, the preferred child node is 'm' but that does not exist (there are no words in the example dictionary beginning 'am...'). At the same algorithmic stage but at step 3, the preferred child is 't'.

The reason for this preference is that most spelling error words have a length similar to the correction, so substitutions are in general preferable to insertions or deletions.

After searching the preferred child node, or on finding that it does not exist, the algorithm searches children representing exact matches for any remaining lowest-cost positions in the costs array. For instance in the example at step 3 the second preferred letter is 'm' but there is no child with that letter whereas at step 6 the favourites from the input string are 'm,a'. On finding that 'm' is not a valid child, the second favourite 'a' is searched.

Any further operations will involve a cost > 0 and so the cost of the best matched word is again compared to the best of the array entries; if they are equal, the search is abandoned. (For slightly better performance and to allow zero-cost operations, it would be possible to modify ALGORITHM T to exit if the second best cost plus the lowest cost editing operation were strictly higher than the best match yet found, but the method implemented is not an inconvenient assumption.)

In the terminology of other approximate string matching papers, the search does not proceed except down branches of the tree representing 'viable prefixes'. A viable prefix is a string of symbols from the beginning of a dictionary entry to any point in that entry where the edit cost of reaching that string is less than a given threshold which in this case is the lowest cost found so far.

If it is conceivable that a good match might still be found (*i.e.* the previous best cost has not yet been exceeded), all of the as yet unexplored child nodes are then entered in turn, and the search begins again at a deeper level. In the example, considering the letter 'n' at step 3, the child 'd' will not be searched because any matches in that direction will have a cost of more than 1, which cannot be as good as the best match already found. Similarly the 'u' of *aqua* is not searched because it cannot achieve a satisfactory cost.

On exiting from the root level of the search, the best possible matches from the dictionary will be known. In the example, the best match is 'ant' with a cost of one.

8.5 Enhancements to ALGORITHM T

Following the initial design of the algorithm, a number of improvements were recommended either by more recent literature, by practical demands of the computing environment, or by an over-active imagination. Those mentioned in

this section have been implemented, whereas those described in Section 9.3 have not.

ALGORITHM T, as presented above, consumes 1.2Mb of memory for a typical 25,000 word dictionary and also uses an extra dictionary data structure of 500Kb. The work was developed in comparison to Du and Chang (1994) which was implemented on a computer with a '386' processor and a very difficult memory arrangement.

To compare the work to that of Du and Chang, ALGORITHM T was tested on a similar machine. Their algorithm required 561Kb of memory for its two main data structures which forced the authors to use 'Extended Memory'. Rather than struggling with the memory contortions imposed by the computer's architecture and operating system, ALGORITHM T was streamlined somewhat so that its data would fit into the 500Kb of main memory available.

8.5.1 Edge Compression

A technique widely used in Suffix Tree work is to combine nodes of the tree where they exist as a simple linked list. This is quite common, accounting for slightly more than half of the nodes used. Rather than storing one symbol per node, 'edge compression' allows each node to contain many symbols if they only occur linearly at that point in the tree.

The reality of computer memory means that space must be allocated for the symbols of each node somehow. In one implementation seen (Nelson 1996), arbitrarily long sequences were stored using two integer pointers to a full copy of the entire text (dictionary) being searched.

The choice made for ALGORITHM T, however, was to reserve space for a particular

number of symbols in each node. Because of the nature of memory use in the 'tnode' record above, this number was set at 3. One further byte was used for flags.

Figure 9.6 shows the number of nodes used for building tries of various sizes of dictionary using both compressed and uncompressed edges. The advantage of compression is much greater for random text than for English words. This is because random text will have longer unique substrings (as part of the even distribution of letters) which can be compressed easily whereas English words fall into groups with many variants, followed by many letter sequences which do not occur at all.

Tries with compressed edges are built on the principle that after a node has been created, it might later be modified. When the first word is learnt, each node will have a single child pointing to the next letter in the word. In this case, each node will be compressed to the maximum allowable extent (combining three letters, in the implementation used here). The final node may not be fully compressed.

When another word is learnt, the nodes are descended as normal. If the new word differs from an existing word at the end of a compressed node, or at an uncompressed node, a new one is added simply. If they differ in the middle of a compressed node then the existing one is split and a new one created. The old node then holds the first part of the compressed sequence up to the point at which the sequences vary. The node created by the split will hold the remainder of the old sequence. Finally, the new node will contain the new letters starting from the point at which the strings differ.

This process is illustrated in Figure 8.3 which shows two branches of a tree, firstly when it holds only the string 'about' and secondly when the word 'above' has been added.

Nodes are not compressed if a real word ends in the middle of the sequence that

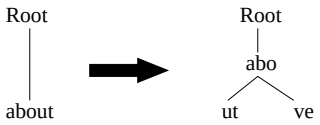

Figure 8.3: A branch of a compressed suffix tree when it contains only the word 'about' and again after the word 'above' has been added.

---

would be compressed. Each node has a single bit flag indicating that it represents the end of a real word. This flag is taken to mean that the word ends at the end of any compressed sequence in that node.

8.5.2 Dictionary Array

In the initial version of ALGORITHM T, a dictionary was also stored as an array. Each tree node which represented a real word contained a pointer to the corresponding dictionary entry. This dictionary entry could contain information on grammar and frequency, but in practice did not.

The spelling of a word is also implicit in the position in the tree at which it is stored. Thus a pointer to a text string is not needed. Not every node represents an actual word, so the only information required in each node is one bit indicating the end of a real word. Removing the dictionary also saved the memory previously used for each pointer to those words, reducing the size of the tree by another 25%.

The bit indicating the existence of a real word is stored in the 'flags' byte which follows the three compressed-symbol bytes. The remaining 7 bits of this byte could be used for other information if required, perhaps encoding some grammatical or frequency information to further assist a spelling correction application.

Chapter 9 contains a detailed analysis of the performance of ALGORITHM T

under various configurations which shows that it outperforms all other algorithms considered under almost all circumstances.

Chapter 9

Results from ALGORITHM T

Chapter 8 described ALGORITHM T, an approximate string matching algorithm which is presented on the grounds that it is elegant, versatile and above all, fast. To demonstrate this, a thorough series of experimental tests are described below which show these claims to be true.

ALGORITHM T was tested on some real spelling error data. It is important to use real data because the degree of variation in performance that is theoretically possible is very different from that actually achieved. The test errors come from a corpus of mistakes from various sources including computer typists copying to dictation, secondary and primary school children and dyslexic adults. The corpus was collected for work on user modelling in spelling correction and is described more fully in Chapter 6.

For simplicity of comparison, the algorithm was run with a 25,000 word dictionary in most cases. The edit cost of insertions, deletions and transpositions were all set to a unit cost of 1.

9.1 Data Structure Size

The size of the dictionary, when stored in the trie as proposed, grows sublinearly with the size of the dictionary. Figure 9.1 shows the number of tree nodes used to store the most common n words from a large sample of English text called the British National Corpus (Leech 1992), where n varies between 10 and 100,000. The horizontal axis represents the number of words, n, in the dictionary and the vertical axis represents the data structure size, in bytes.

The lowest line, `datasize-bytes`, in Figure 9.1 indicates the number of bytes used to store the dictionary. The line `databytes-comp` shows the number of bytes used by a compressed tree, and the line `databytes-imp` shows the number of bytes used by a tree if it does not contain compressed nodes. Each node occupies 12 bytes of data space. The remaining line, `databytes-edp` contains a calculation of the number of bytes used by the EDP algorithm, discussed below.

The data structure is able to grow sub-linearly because of the repetitive patterns at the start of words. As new tree nodes are only created where no existing word has the same prefix (initial sequence of letters), large dictionaries with a denser coverage of possible prefixes are more efficient. This has to be balanced against the length of words; larger dictionaries contain longer words (the average length rises from 2.4 letters for 10 or 20 word dictionaries to 5.5 for 1000 words and 7.8 letters for the most common 100,000 words in English).

As stated earlier, the upper bound on the tree size is $O(n)$ for a dictionary containing n symbols regardless of the number of words. Because each node typically occupies 12 bytes, the tree typically occupies somewhat more space than an uncompressed dictionary.

For comparison, the data structure used by Du and Chang is reported to have

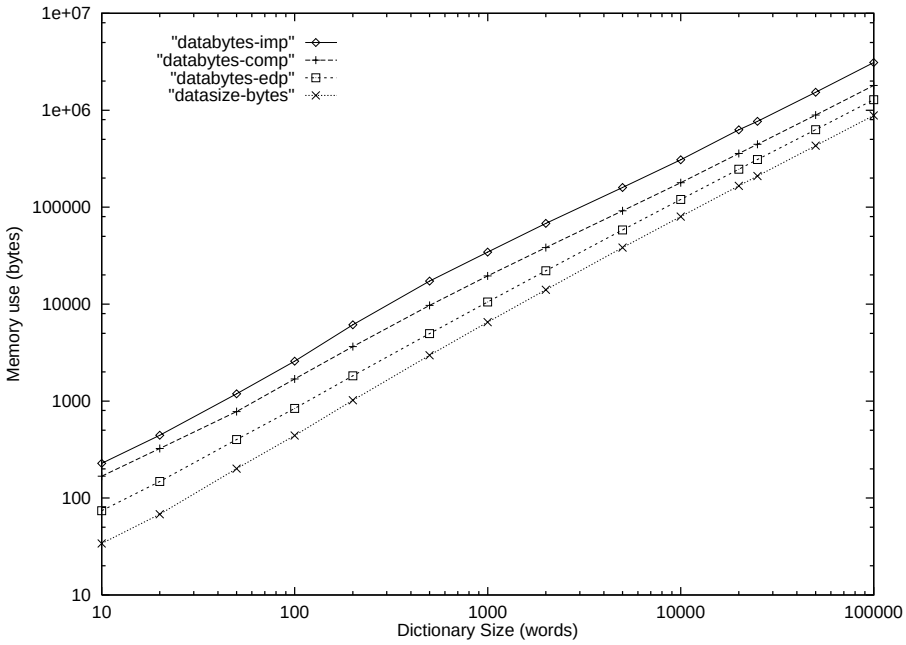

Figure 9.1: The size in bytes of the data structure used to store various sizes of dictionaries of English text in ALGORITHM T, the EDP algorithm and a simple plain-text array. In this and other graphs in this chapter, the key showing line styles is listed in the same vertical order as the lines plotted, to make comprehension easier.

been 561Kb which was unfortunate because the computer they used had around 500Kb of space available to normal programs. The overflow meant that they were forced to use 'extended memory'. The most similar dictionary used by Algorithm T occupies 435Kb which, although not much smaller, is enough to simplify operation substantially on the class of computer used by Du and Chang.

The data space required for algorithms EDP and ABM, presented below, are simply that required for the main text plus (for dictionary work) a pointer to each word. Thus for the 25,000 word dictionary used here, that would require $209866 + (25000 \times 4)$ bytes, or 302Kb. Interestingly, however, they grow at a slightly higher than linear rate, presumably because of the requirement for a copy of the text and an array of pointers.

Working memory used by ALGORITHM T is minimal, just $O(|P|)$ for a pattern string P which is less than any other comparable algorithm. For example Ukkonen (1993) requires working space of $O(m^2 q)$ and Shang and Merrettal (1996) require $O(nm)$ as published.

9.2 Run Time

The speed of running ALGORITHM T, unsurprisingly, depends on the edit cost of the match found. This is because it must exhaustively search the tree until no further operations can lead to a lower cost than those already found.

The experiments conducted here have been performed mainly on a Silicon Graphics 'Indy' workstation with an R4600 processor running at 100MHz. Where comparisons are made with Du and Chang (1994), a comparable '386' machine has been used.

Figure 9.2 shows the time taken to find minimal cost matches for the `sunny` error

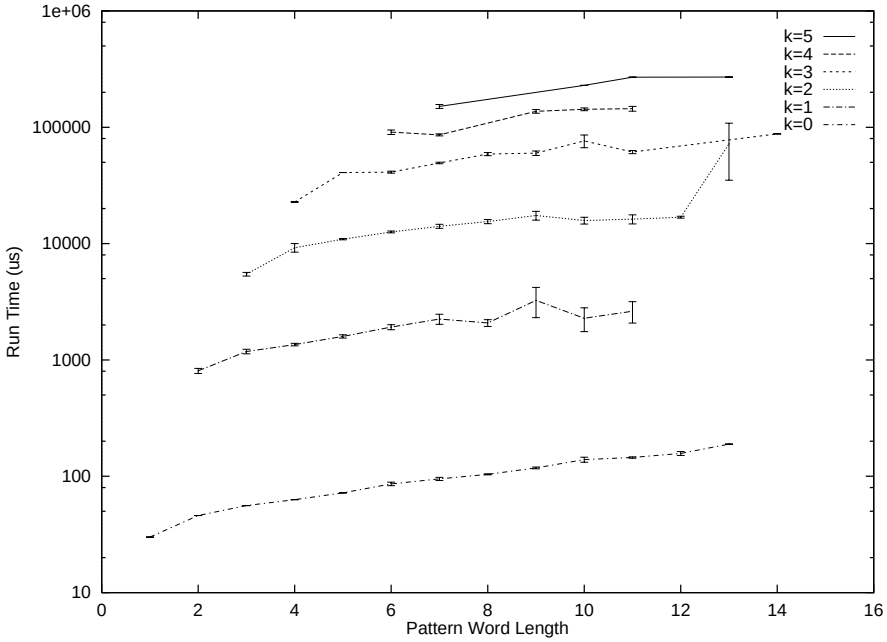

Figure 9.2: The run-time of ALGORITHM T for various edit costs and pattern lengths. Some points on the graph are based on very few data points and so may be unreliable.

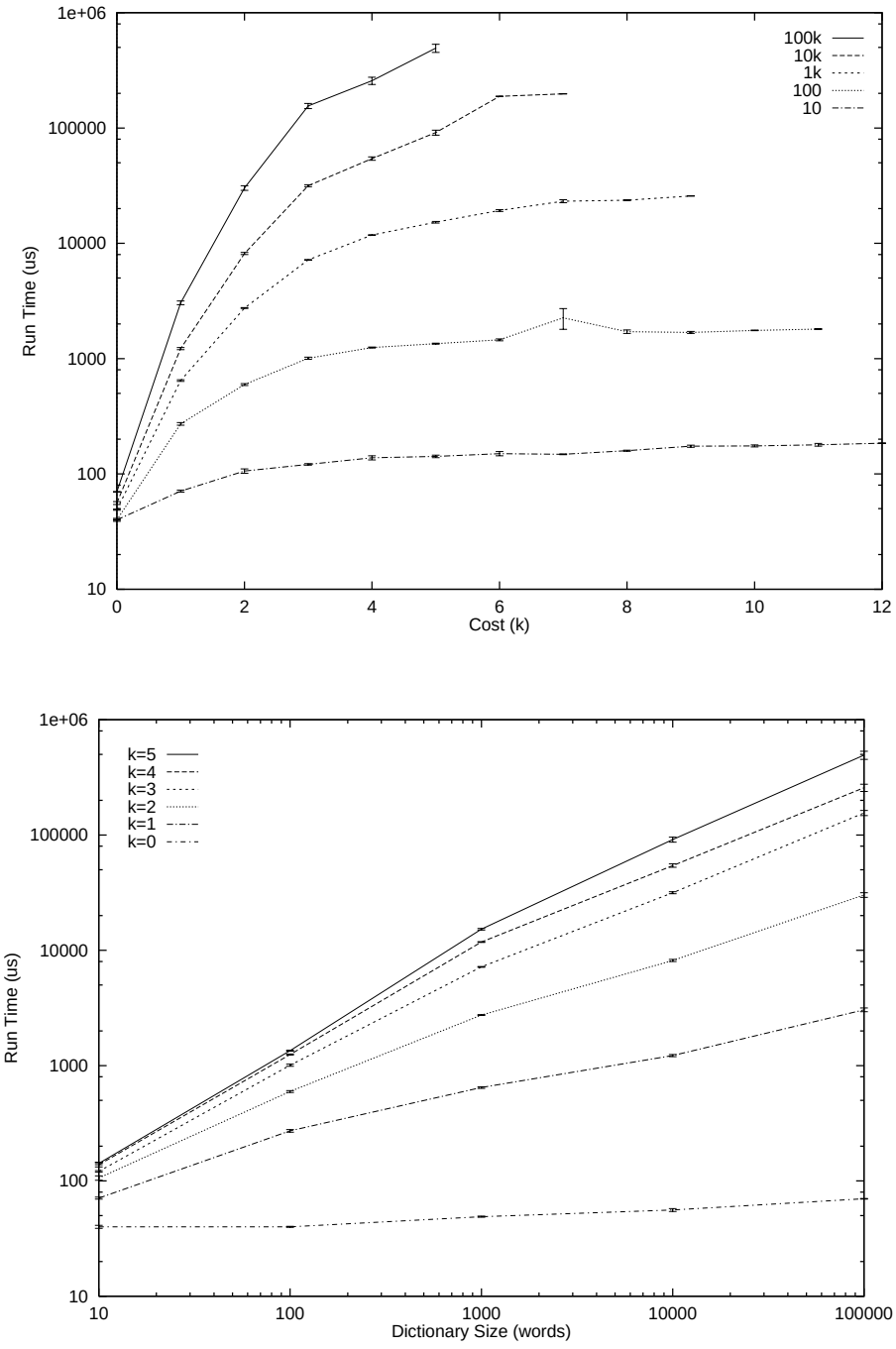

Figure 9.3: The run-time of ALGORITHM T for various dictionary sizes and edit costs, presented both by cost and by dictionary size. The run time increases more slowly with high edit costs as it approaches full coverage of the tree. See also Figure 9.5.

corpus. The horizontal axis indicates the length of the error word being searched for, and the vertical axis shows the time in microseconds.

It was considered important to test the algorithm not on contrived permutations of random strings, but on genuine spelling errors. There may be effects related for example, to the position in the word of the errors, which change the performance of the system. The table below shows the distribution of edit costs in the Surrey data used for these tests.

Cost	N^o **words**
0	16955
1	1681
2	490
3	114
4	45
5	14
≥ 6	2
Total:	19301

The data points shown in Figure 9.2 are averaged from dozens or hundreds of error words, or are in line with the neighbouring values. Those where insufficient samples were tested to be confident of the results have been omitted. Naturally, short words with many errors were not found. For example, a four letter string can normally be transformed into a one letter word by deleting three of the letters, so there are no four letter words shown with an edit cost of 4.

Figure 9.3 shows the run-time of ALGORITHM T with various different dictionary sizes. Each dictionary has the most common n words as observed in the British National Corpus. Other experiments were conducted with a 25,000 word dictionary from the same series because that size is comparable with other algorithms, and with real spelling correction tasks. There are two graphs to show the same information from both the perspective of the edit cost and the dictionary size.

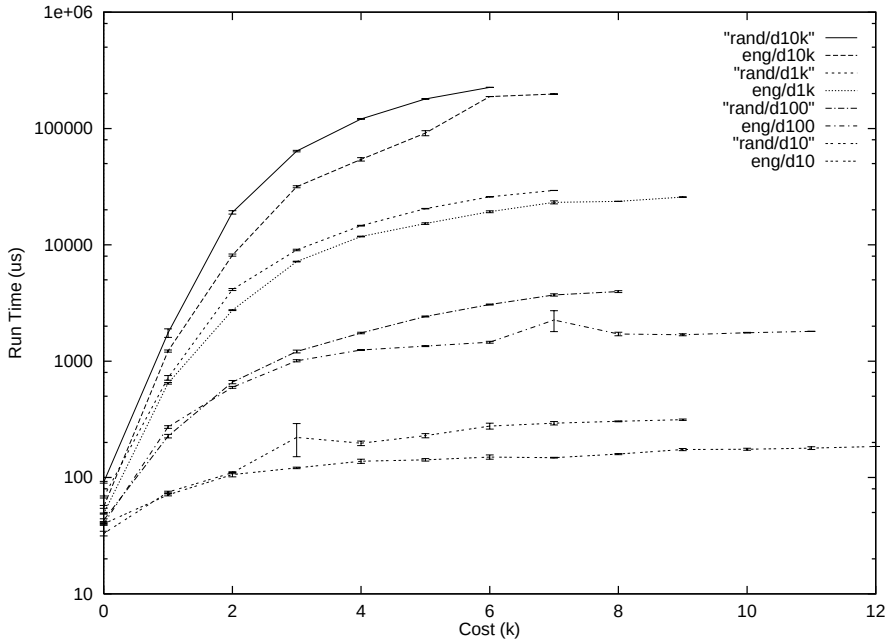

Figure 9.4: The run-time of ALGORITHM T for random letter sequences compared to genuine English spelling errors. Lines in the legend labelled rand/ are for random strings. The second part of each label shows the dictionary size; d10k has 10,000 words and d10 has only ten.

Although the experiments conducted here have used genuine English spelling errors, it is also interesting to compare the performance of ALGORITHM T on English errors to that on random symbol strings. This is because ALGORITHM T was designed to work better on real words and errors; errors near the end of a word are faster to correct than those at the start, and dictionaries of real words compress more efficiently than random strings. To this end, a set of random dictionaries was constructed. The symbol set was of our normal alphabet, $|\Sigma| = 26$. The dictionaries were arranged in 'words' of between 1 and 10 random letters. The lengths of words were fairly evenly distributed. Each dictionary was seeded with a different random number sequence and so would not contain a significant overlap with any other dictionary. An extra dictionary, with 1000 words, was treated as the error word set, and its entries were searched as patterns P in each of the other dictionaries.

The results plotted in Figure 9.4 show that ALGORITHM T performs slightly better with actual spelling errors, by a degree varying from a little more than 2 down to a slight worsening of performance in some cases. The edit cost observed for each word was recorded and analysed appropriately as in the other tests; the edit cost was not 'defined' in advance by any mutation method. In practice, the edit costs observed were split roughly evenly from 0 to the string length, but with a drop-off in numbers of high edit-cost strings, for example having half as many strings $k = 6$ as $k = 4$ for the 10k dictionary.

This can be looked at in two ways. Firstly, the algorithm's performance is not constant for all types of input data. This is unsurprising as the exact paths pursued in the trie depend very much on the input data. Secondly, it can be considered a confirmation that ALGORITHM T performs better (typically by a factor of two) with genuine spelling errors than with random strings, which reinforces the suggestion that it is particularly well suited to spelling tasks.

Figure 9.5 shows the proportion of nodes searched for each word. In the best case of an exact match, only the nodes on the path from the root to the correct leaf node are searched. Roughly speaking, the trie is explored to a depth of k further than would be required with a precognizant algorithm[1]. With small dictionaries, this extra search includes much of the dictionary quickly and so it can be seen to saturate for smaller values of k. In Figure 9.5, some points are based on single data samples and so are unreliable; Points where $k \geq 6$ for the 10k dictionary demonstrate this most clearly.

Even when the entire trie is searched, ALGORITHM T is more efficient than other methods like EDP which, in their worst case, will search every symbol in the

[1]A precognizant algorithm is one which knows in advance where the best matches are. Such algorithms are not currently available in published works! One might also think of non-deterministic algorithms which search directly towards the correct solution at each stage

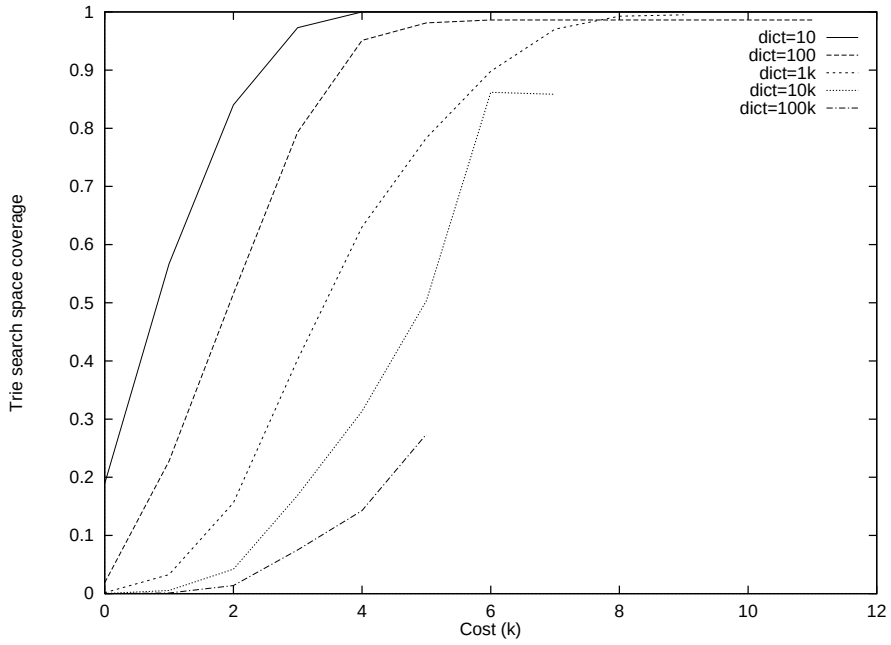

Figure 9.5: The proportion of nodes in the trie searched by ALGORITHM T for various dictionary sizes and edit costs.

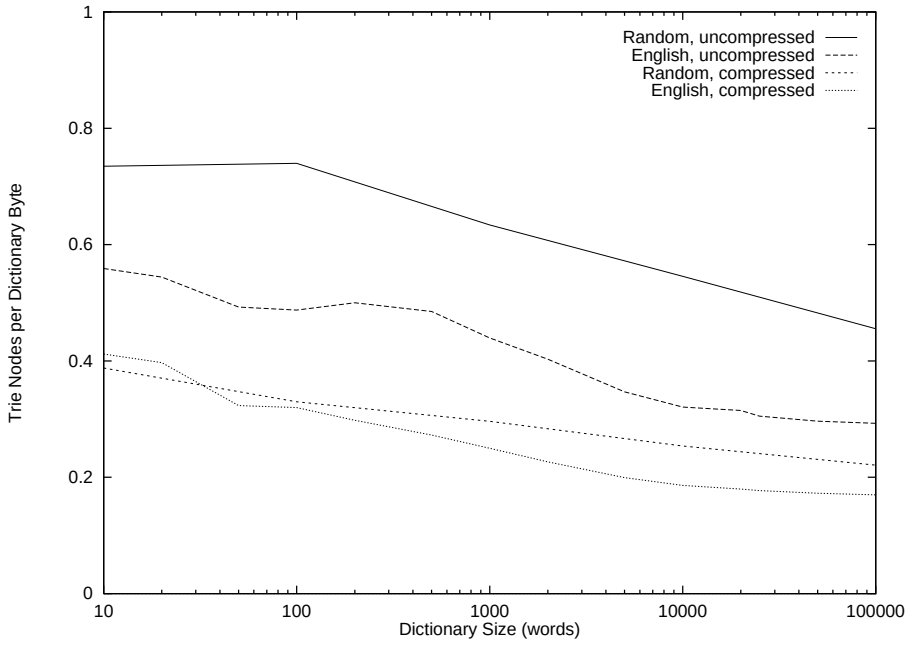

Figure 9.6: The number of trie nodes created by ALGORITHM T as a proportion to the size of the corresponding dictionary in characters. This also shows the savings made with edge compression.

entire dictionary. This is because the trie combines matching prefixes into a single sequence of nodes from the root, and calculates these only once. Thus after following the 'a' node, the first column for all words beginning with 'a' will have been calculated. Figure 2.3 on page 32 shows the computation of a simple edit cost solution for a single pair of strings.

Figure 9.6 shows the number of nodes created for a dictionary compared to the number of character symbols in the dictionary. It can first be noted that in all cases there are fewer nodes than bytes in the dictionary. The compression improves as the dictionary size increases, generally because of the larger number of strings with common prefixes.

Figure 9.6 also indicates the saving made by compressing the edges where only a single sequence of symbols is viable. In the system implemented, the maximum sequence length is 3, and so the maximum improvement is a threefold reduction in data space. Where the reduction is less than threefold (in all cases), an increase in compression capability would not be fully utilised, and to a degree would be unnecessary.

The calls to the main function, `searchtree`, take around $7\mu s$ to complete on the Silicon Graphics computer. Each call represents the calculation of one column of the $H[i,\ j]$ array as well as the processing of any real words found, and the choice and execution of any recursive calls.

Comparison with Du and Chang

For comparison with an existing algorithm, the work here was timed while running on a computer with an 80386 processor. Figure 9.7 shows the time taken by ALGORITHM T compared to that of ALGORITHM_4 (the best) in Du and Chang (1994) for words of 9 letters. It should be noted that the run time of their algorithm

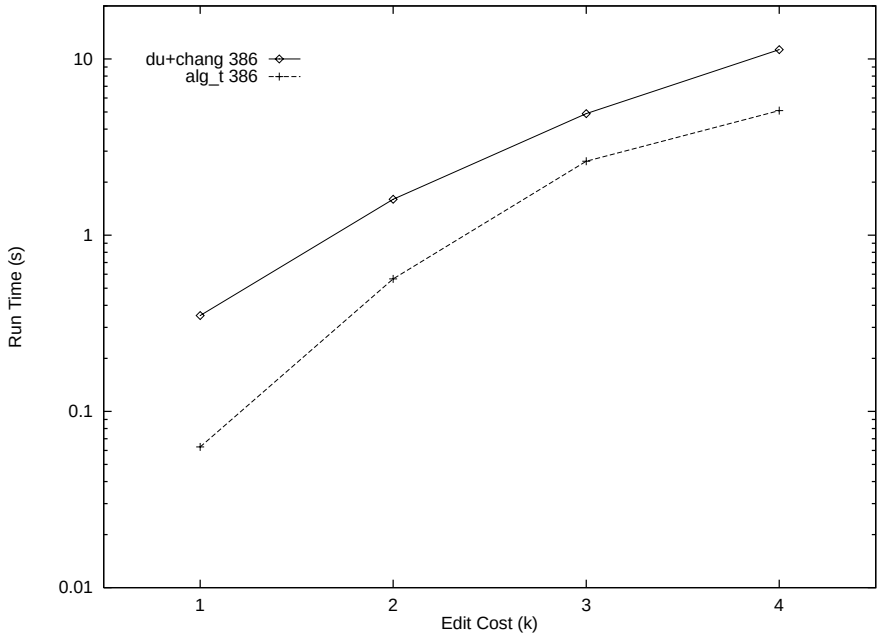

Figure 9.7: The run-time of ALGORITHM T as presented here, compared to that of Du and Chang, when running on a '386' computer.

varies substantially with the number of words in the dictionary with lengths similar to the search pattern. The distribution of word lengths was not published.

In the comparison, both algorithms were measured in real time on machines with with 80386 processors. The one used for ALGORITHM T had a 16MHz clock. However, the clock speed of the machine used by Du and Chang (1994) was not published and may have been as slow as 16MHz or as fast as 33MHz, meaning that ALGORITHM T may be more than two times faster than it appears. It is already substantially faster than ALGORITHM_4.

The tasks being performed were not quite identical. Du and Chang found all words from their dictionary with an edit cost within a predefined limit. ALGORITHM T finds words with minimal edit cost. For spelling correction tasks, the latter is more appropriate so that the user can be offered a short list with the most likely choices. Du and Chang's algorithm would need to be run repeatedly until sufficient words were found, each time increasing the threshold value and duplicating the work of the previous searches.

Both algorithms used a dictionary of 25,000 words. The one used by Du and Chang contained words from a library catalogue. The one used here contained the 25,000 most common words from part of the British National Corpus, a collection of published text.

The error words used by Du and Chang were 100 random mutations of dictionary entries for each test. Although their mutations may have made words more similar to other dictionary entries than to the base from which they were derived, the function of the algorithm would not be substantially changed. The words used for ALGORITHM T were genuine spelling errors from children in the `sunny` group also used elsewhere in this research.

As stated above, the data space required for ALGORITHM T is about 75% of that

required for Du and Chang's ALGORITHM_4. The space required for dictionary sizes other than 25,000 words is not known for ALGORITHM_4, but grows in a slightly sub-linear fashion for ALGORITHM T.

The most striking difference in the accomplishment of ALGORITHM T, presented here, is that it allows any edit cost to be applied to any operation unlike that of Du and Chang which fundamentally assumes that each editing operation is of equal cost, whereas ALGORITHM T demands no such limitations.

Comparison with Shang and Merrettal

Although ALGORITHM T was thought to be novel, having been developed independently of any other recent work, it was subsequently found that similar algorithms had been described, notably under the headings of Suffix Trees and Tries.

Shang and Merrettal (1996) described an algorithm very similar in most respects to that presented here. However, its performance is markedly inferior. The authors claim it to be superior to algorithms available at that time for edit costs of $k \in \{0, 1\}$ but to lose out against existing algorithms for higher values, $k \geq 2$. The principle of pruning the tree when the lowest edit cost in the array $H[\,]$ is too high is discussed, as is the benefit of avoiding recalculation of the array entries for duplicate prefixes of different words. The same could also be said of other more distinct algorithms (Cobbs 1995), the authors of which also pride themselves on these ways of avoiding unnecessary calculation.

A brief implementation of the algorithm of Shang and Merrettal was devised and tested for comparison, by deleting the part of ALGORITHM T which chooses the most probable child node to descend. Such a change is in the spirit of their work, but the code as executed is almost identical to that of ALGORITHM T. The imperfection is that instead of calling their function `EditDist(j)` to calculate

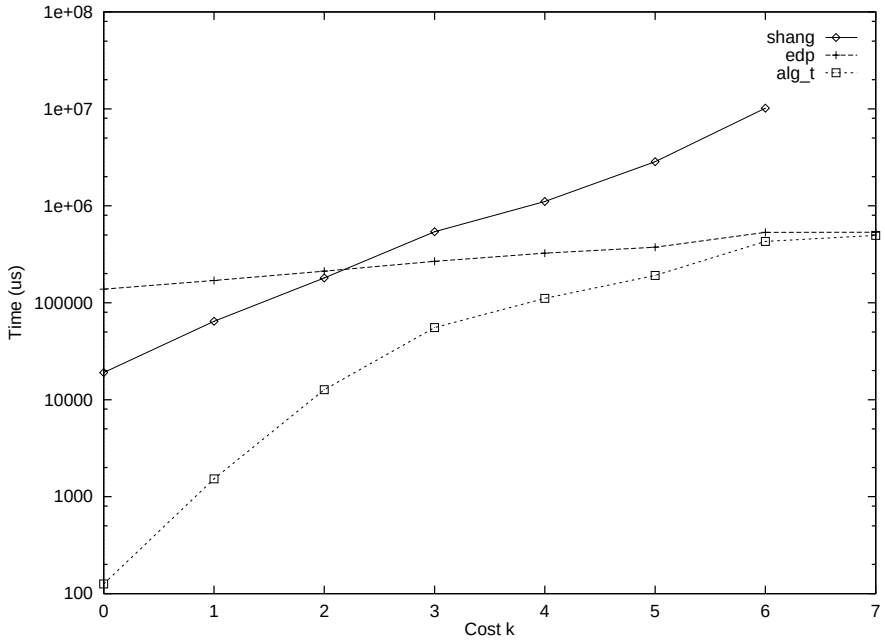

Figure 9.8: The run-time of ALGORITHM T presented here, compared to an approximation to that of Shang and Merrettal and also to the common EDP algorithm.

part of a column of $H[\ ]$, the calculation of a whole column is done within the `SearchTree()`. This is faster for recursive execution but omits the partial computation enhancement which is discussed separately below in Section 9.3.1.

Figure 9.8 shows the run time of this imperfect (but illustratively adequate) implementation of their algorithm. Also in the graph is the EDP algorithm, discussed below. This highlights the truth of Shang and Merrettal's admission that their algorithm performs poorly for $k > 2$ whereas ALGORITHM T does not.

ALGORITHM T finds the lowest cost row in $H[\ ]$ and descends to the child node representing the next character in that diagonal of the array. Thus an exact match for P takes only $|P|+1$ iterations because at each stage the correct child is found. By comparison, Shang and Merrettal descend every child node in alphabetical order. Although the tree pruning part, which abandons searches when $min(H[*]) > k$, is used, it does not stop the algorithm from being between 150 and 10 times slower than ALGORITHM T. For edit costs $k > 2$ it takes longer than the EDP algorithm, presented below.

Shang and Merrettal also failed to implement edge compression in their Suffix Tree although they were well aware of other authors' efforts regarding compression. It cannot be denied, however, that their work addresses the same issue as that considered here, and that it has been published.

Comparison with Enhanced Dynamic Programming (EDP)

A useful paper on the practicalities of approximate string matching is that by Jokinen, Tarhio, and Ukkonen (1996) which, rather than presenting novel algorithms with ever more unlikely looking upper bounds on their run times, compares seven methods which have already been published. These include dynamic programming, Boyer-Moore string matching, suffix automata, and the distribution

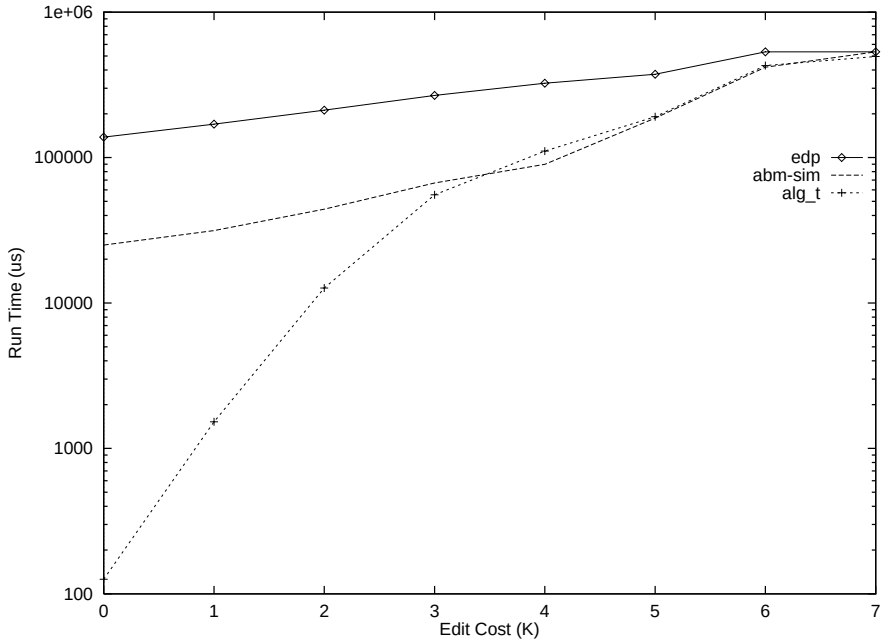

Figure 9.9: The run-times of ALGORITHM T as presented here, compared with EDP and an estimation of ABM which, together, were the best algorithms reviewed by Jokinen.

---

of characters.

Suffix Automata are an improvement on Suffix Trees which have been discussed in other recent literature, so it is helpful to have a practical report of their speed. They perform poorly.

Dynamic Programming is used by the original method of Wagner and Fisher, but is tested here with the 'Enhanced' (EDP) version published in Jokinen, Tarhio, and Ukkonen (1996). EDP processes columns one at a time, and ignores corners of the $H[i, j]$ array which contain no values that can be involved in a satisfactory match.

The EDP algorithm, despite its simplicity and age, was found to be the most efficient for tasks with small patterns (error words) or large allowable numbers of errors. It is compared with ALGORITHM T in Figure 9.9 clearly showing that,

for small edit costs $k = 0$, the new algorithm is more than 1000 times better, and remains better albeit less distinctly for all observed edit costs.

The algorithms discussed in the paper are designed for approximate substring matching, as might be done in biological environments. The case of spelling correction is somewhat different; each word must be considered from its start. The EDP algorithm has been re-interpreted for this environment, with as few changes made as possible. To test its speed against ALGORITHM T, it is run on each word in its dictionary. Initially the maximum allowable edit cost is very large, and is reduced with each match to the smallest cost yet found. All words in the dictionary are compared, typically in alphabetical order. This is rather similar to the approach of Shang and Merrettal (1996) which seems simplistic, and may be responsible for the poor speed. However, any change to the order of comparisons would not have been faithful to the EDP algorithm, and so was not implemented in the tests.

Another algorithm, Approximate Boyer-Moore (ABM) is a development of the exact string matching method devised by Boyer and Moore and was reviewed by Jokinen, Tarhio, and Ukkonen (1996). It was found to be five times faster than EDP for small edit costs $k \in \{0, 1, 2\}$, and falling to about the same speed for edit costs of around $k = 7$. Although it has not been run on a spelling task, timing information from the review paper has been used to simulate its speed as a function of the speed of EDP, and is shown in Figure 9.9. However, the main advantage of ABM over other methods is its efficient shifting through the main text which is not relevant to spelling correction. Algorithm ABM also employed EDP to check those parts of the text which it considered likely to contain matches.

ABM was the fastest algorithm for small edit costs k, and EDP was the best for high edit costs. Various other algorithms occupied niches of other values of the edit cost k or search pattern size m.

It should be noted that points on the graph where $k \geq 6$ are based on very small numbers of sample words, although it is reassuring to see that the algorithms all appear to follow a regular pattern and so the data in that region can be considered satisfactory.

Considering that the majority of words observed have low edit costs, with more than 99% of cases being $k \in \{0, 1, 2\}$, it seems clear that ALGORITHM T is superior. This can be attributed at least in part to the fact that it is designed specifically for spelling correction tasks whereas the other algorithms are intended for DNA sequencing. This is not intended to criticise them, but to highlight the differences in the two tasks which are often considered to have negligible differences.

The data space required for the EDP and ABM algorithms are proportional to the number of characters in the dictionary, being around 302Kb for the 25,000 word dictionary which requires 435Kb in ALGORITHM T; slightly less than 70% of the size. The run-time of EDP grows approximately linearly (slightly faster than linear) with the size of the dictionary.

In a different field of study, Greene (1994) uses an example of spelling correction of single error words. He has to compare around 200 feature vectors, each with 709 features in order to find the best match with a confidence of 98%. One must presume that after such a sequence the words relating to the vectors found to be closest would then need to be compared again with a dynamic programming edit-cost method. Unless the vector comparison is much faster than the calculation of one value for $H[i, j]$ here (which will not occur on any conventional computer), his method would take considerably longer. Exact times were not published.

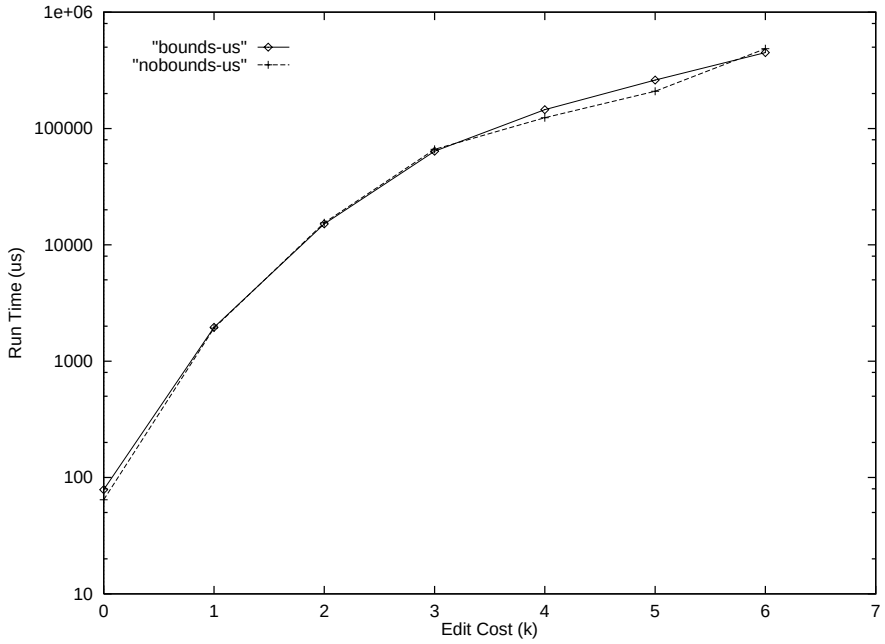

Figure 9.10: The run-times of ALGORITHM T when implemented with and without bounds-checking. The time taken to calculate the bounds appears to consume any savings in time spent not calculating array values.

9.3 Further Improvements

In addition to the basic design and enhancements in Section 8.5, a number of further possibilities might be explored.

9.3.1 Partial Computation of $H[i, *]$

The design of ALGORITHM T makes it simple to avoid unnecessary computation of columns of $H[*, j]$ which relate to high edit costs. The same principle might also be used to avoid calculating unnecessary rows in each recursive call to `searchtree()`. As with the EDP algorithm, the lowest row at which the cost is higher than k is recorded; computation of the cost does not proceed beyond (or

even to) that row.

However, calculation and propagation of the information recording the viable rows takes some time. In practice, Figure 9.10 shows that the bound-checked method actually takes longer in some cases than that in which all rows of $H[i, j]$ are calculated. It is faster on occasion, but the added complexity and the fact that it is generally slower make it less desirable.

Shang and Merrettal (1996) implemented such bound checking and presumably found it to be satisfactory. If the calculation of each entry in $H[i, j]$ took a substantial time, a saving may be made by including bounds checking again.

9.3.2 Further Editing Operations

A number of further editing operations might be considered which would enhance the operation of the algorithm for spelling correction tasks. The first of these is the Transposition, in which two adjacent letters are swapped over. Shang and Merrettal (1996) implemented such an operation, but did so without observing the constraint $\min(W_I, W_D, W_C) \leq W_T < \min(W_I + W_D, 2 \times W_C)$ if the main algorithm is to remain unchanged. This constraint makes transpositions far less useful outside the authors' unit cost environment. For this reason it has not been implemented in ALGORITHM T.

For a more flexible approach to transpositions, one might look ahead of the dynamic programming method on occasions where the search is about to be abandoned, but where a transposition can occur. This is because transpositions can break the basic assumption of the dynamic programming method that the cost of entries in $H[\,]$ will always rise as one moves away from the original corner. The efficiency of ALGORITHM T is derived from its ability to abandon searches when the cost is too high. A situation where the cost might fall if further processing is done

would fundamentally change this; a simple solution is to look ahead one symbol for possible transpositions but this is inelegant.

Possible operations involving multiple symbols also face the same problems as transpositions; that the cost might fall if further processing is done. A potential solution in that case is to convert such symbol sequences to other single symbols which do not occur normally, and then to process them as those single symbols. This was done by Pain (1985) but has not yet been repeated here.

Operations at the beginning and end of each string are also of interest. The popular substring matching problem applies no cost to the deletion of any number of symbols from the beginning of T, nor the end of T. Allowing similar deletions from the start and end of P are desirable but have not yet been implemented here. In each case, exceptions could be made to the values of $H[\,]$ at key depths in the tree to accommodate special costs.

A typical spelling correction method might allow for truncation of the pattern string P by applying a special cost W_{DF} when considering $H[i = m, *]$ while maintaining the standard W_D for other values of $H[i \neq m, *]$.

9.3.3 Parallel Execution

Parallel execution of ALGORITHM T would be relatively simple. Each processor could store a copy of the tree and, whenever several children needed to be searched, a new processor could be used for each. The only communication requirement is in spawning new searches and communicating information about best matches; something which will happen at most *num_matches* + *max_letters* times.

At the present time, spelling correction tasks do not demand such performance, and typical users of such software do not have appropriate computing hardware.

9.4 Conclusions

An algorithm has been presented which performs approximate string matching in a dictionary faster than any other known algorithm for the vast majority of cases. (The algorithm ABM was faster in tests where $k = 4$) It uses a trie to avoid unnecessary duplication of calculations, and prunes its trie search when it will not achieve a satisfactory result. Unlike some other methods for pre-filtering dictionaries, this algorithm will guarantee to find the closest match, and will do so quickly.

Jokinen, Tarhio, and Ukkonen (1996) presented a most helpful review of approximate string matching algorithms in practice which enlightened readers considerably on the practical efficiency (or otherwise) of previously published algorithms. They observed that for low edit costs $k < 3$, Algorithm ABM was the best of those tested and for high edit costs or low pattern lengths, Algorithm EDP was preferable. The practical results were somewhat different to the predicted worst case run times.

ALGORITHM T has been compared directly to EDP and found to be as much as 1000 times faster for typical spelling tasks, and somewhat faster in all cases observed. A simulated run-time for ABM has been derived from information in the review paper. It has shown that ALGORITHM T is again, immensely faster for most cases although marginally slower in one case which accounts for 0.2% of observed spelling errors.

The remaining algorithms in the paper, which include Suffix Automata, are discarded on the grounds that they were found to be worse than ABM or EDP in relevant situations. The paper appears to contain a comprehensive review of such algorithms available at the time of publication. By this reckoning, ALGORITHM T is good.

ALGORITHM T was also compared to Algorithm_4, the best of those published by Du and Chang (1994). It out-performs Algorithm_4 by a factor of between 5.6 and 1.8 times, and might be further optimised if given further attention.

The algorithm was implemented and tested on real spelling error data from a number of sources, and found to work at least as well as previous systems under their constraints, but is able to operate more flexibly with words with unknown numbers of errors; the more realistic case. Although its worst case performance is the same as a full dynamic-programmed search of the dictionary, it typically finds the lowest-cost dictionary entry in around 1000 iterations of the edit-cost recurrence relation, taking around 3mS on the computer used where $k = 1$.

In the prototype system the 25,000 word dictionary tree occupies 435Kb of working memory, falling between EDP and ABM which require about 30% less, and Du and Chang's Algorithm_4 which requires nearly 30% more. The memory requirement grows sub-linearly, albeit only slightly so, with the size of the dictionary. The data size of EDP and ABM grow at a rate slightly higher than linear. This may be due to the increasing word lengths in longer dictionaries (because the English language contains some natural Huffman encoding).

The system accommodates varying costs of editing operations, and could be extended further, for example with character-dependent costs or some multiple-character operations. Similarly, although only the best matches are currently returned, a list of the best n matches could easily be found.

The superiority of the algorithm is twofold. By searching for a minimal edit cost k, it allows for high costs but does not waste time finding all of them if lower cost matches are available. By organising its dictionary as a trie and carefully choosing the suffices to pursue further at each stage, it avoids wasteful duplication of calculations and less probable avenues of search.

Most other algorithms do not take advantage of the special nature of spelling tasks, and so perform poorly by comparison to ALGORITHM T. This is not to say that they are poor in their own environments. A review by a biologist of ALGORITHM T would be interesting to read.

The algorithm is particularly well suited to spelling correction of errors by dyslexic writers as it is more tolerant of words where the errors are towards the end of the string than towards the start. It could also, with some further work, sort the order of node searches to account for common letter substitutions. This is demonstrated by Figure 9.4 which shows that run-times for random letter strings is somewhat worse than for the actual spelling errors.

Whereas some algorithms fundamentally assume that $W_I = W_C = W_D = 1$, ALGORITHM T does not. However, the costs have been made equal in this way for to make the comparisons here fairer. With a weighted cost model, the run time would not be increased in Algorithms T or EDP. Algorithm ABM and Algorithm_4 would consider more potential candidates if the maximum allowed k were higher, leading to a much increased run-time. The exact effect of this has not been studied, although it is likely that the run time of Algorithm T would actually fall with a weighted edit cost model.

ALGORITHM T's speed of execution makes it viable for batch processing jobs where greater speed is required and also for interactive tasks which could not previously be considered such as real-time spelling checking or correction as a user scrolls through a document.

It works better with low editing costs than with high ones, but in almost all cases functions better than all of the algorithms to which it has been compared (Figures 9.7, 9.8 and 9.9), and by implication better than all other published algorithms.

Chapter 10

Conclusions

The work in this thesis has produced mixed results. The work on ALGORITHM T was exceptionally positive, introducing a new algorithm which performed the approximate string matching task against a dictionary faster and with fewer constraints than existing algorithms. The work on Babel was less successful, the reasons for which are discussed below. The sample text collected for Babel is also discussed in its own right as a successful enterprise with further future potential.

10.1 Algorithm T

ALGORITHM T, presented in Chapter 8 was developed independently of methods used by biologists for DNA sequence matching. There are no comparable algorithms properly intended for spelling correction. It was in fact based on observations of work by Pain (1985) in which a similar suffix trie was used for substitution error matching. An almost identical method was incorporated in Babel to improve its performance with wildcarded error strings (*ie* strings with ambiguous symbols) which led to the further development of ALGORITHM T. The

differences between this and other algorithms account, however, for an immense increase in performance; up to 150 times faster than the most similar algorithm and 7 times faster than a limited dictionary-based method. It was marginally slower than one other algorithm in one out of the myriad of cases considered, while being substantially faster in the vast majority of cases.

Not only is the algorithm presented theoretically and in the context of existing literature, but also by empirical comparison to the most relevant parts of that literature. Such an empirical comparison is unusual and interesting itself, particularly as it comprehensively considers the effect of a number of factors such as word length, edit cost, dictionary size and symbol alphabet.

The algorithm is better suited to spelling correction tasks than some other recent works, thus evening up the balance somewhat against the biological emphasis which has been placed on approximate string matching in recent years.

Helen Pain's PhD thesis (Pain 1985) made extensive use of a program, `editcost`, which corrected spellings using the edit cost of the word typed to dictionary words. The program worked well but was found to be slow in interactive use. ALGORITHM T would be capable of performing such a task, on an entire dictionary, well within the time limits of an interactive system.

10.2 Babel

Babel, the spelling checker with a user model, produced less stimulating but still useful results. It demonstrated that such a user model (described in Chapter 5) was not capable of significantly improving the performance of the system either using a canonical model of errors from all authors or an individual one of errors from a single person. It also showed that the whole system, including its error pattern

rules, was not conclusively better than the two existing systems to which it was compared overall.

Babel did achieve the success of presenting the correct word on its suggestion list significantly more often than `ispell` for most user populations including the most important ones, and it also succeeded in presenting the correct word significantly nearer the top of its list than SPEEDCOP for most user populations, again including the most important ones. It presented the correct word on its list in over 80% of cases for some populations.

The Path Choice Heuristic in Babel was an original algorithm for selecting preferred solutions from a range offered by a task. This may find applications in other areas of study and has worked successfully in its task here.

As the experiments on Babel in Chapter 7 produced largely negative results, it is necessary to consider its weaknesses and reasons for failure. These are not all mechanical reasons; in part it may be because human spelling errors are fundamentally unpredictable.

Babel operates in a 'batch' mode and does not present suggestions interactively to dyslexic users sitting at the computer. This was intentional because while the software was under development, it was impractical to ask dyslexic people to return repeatedly to test it. The non-interactive nature may be criticised for ignoring the effect of poor reading on selecting the correct word from a list, but this factor was measured equally between all software tested and can safely be ignored without invalidating the results. This batch processing operation could, however, be considered a shortcoming.

Babel also operated on single words, paying no attention to the context in which an error was made although informal experiments with good spellers showed that errors were almost always correctable in context. It was a design decision of Babel

that it should not consider context, as that would be another large domain of study which should properly be undertaken elsewhere.

The User Modelling system in Babel was based on a novel method incorporating weighted rules and the novel Path Choice Heuristic. There are a number of alternative user modelling methods which may well be useful (described in Chapter 4), but even they were designed for different applications and would not be guaranteed to produce reliable results. After the main work on Babel was completed, it was decided not to return to the design stage with a more sophisticated method. Based on the evidence available in Chapter 7, user models have been found to be unnecessary for spelling correction.

The rule set used for the basis of the whole Babel system is modular and so could be extended or changed without making a design change to the remainder of the system. However, a more carefully tuned rule set might operate more effectively and produce a more accurate system. Although a number of more sophisticated rules can be conceived, implementation of them is particularly difficult.

For example, a model of the standard spelling rules for creating affixes, combined with a list of root words, might help to identify words which were constructed by applying the wrong spelling rule to a particular base, or by applying a rule to the wrong base word. Two examples include ‘maintainance’ ← ‘maintenance’ which is simply an extension of ‘maintain’ and is incorrect, and ‘charlotten’ ← ‘charlatan’ by phonetic analogy to the girl’s name ‘Charlotte’; quite incorrect but logical to a person with limited experience. Accounting for such errors would require an impractically large effort of programming.

Another rule set that might be incorporated could be a better handling of serial errors. If a person omits a single letter, there is a higher chance than before that they might go on to omit the next letter or make other mistakes. Babel takes no account

of this, but perhaps should. Again, implementing this would require much further work and a particularly large sample of source data on which to test the methods and from which to derive the principles. Obtaining such samples is very difficult.

The original working hypothesis (on page 159) for Babel was that spelling errors contained repeatable patterns which could be captured by the computer software. This was been found to be incorrect in Chapter 7. Babel was not able to capture patterns significantly, nor to improve its performance substantially after training. This was shown by the lack of improvement in list position and frequency of presence on the lists, and by the lack of distinction between user models.

As so many of the errors are phonetic, it may have been worthwhile to make a greater effort in the phonetic correction part. At present, the orthographic to phonetic converter of McIlroy (1974) was used to create pronunciations of words written and of dictionary words. It was decided that a major effort on phonetic conversion would constitute another research project and was not justified in this work.

It is probably the cognitive complexity and irregularity of the errors being made which limits the success of Babel. The people making errors are not illiterate, but have faults somewhere in their understanding of language. It is a huge area including pertinent correct information such as basic words as well as incorrect information such as faulty knowledge of when to apply certain spelling rules. When a person is unsure of a spelling, it is clear from observation that they often employ a variety of techniques to generate one, and such techniques can interact in complex ways. It may well be that developing a system on the basis of such rules is an intractable problem.

This work is not based on psychological or educational expertise, and so is not presenting a strong theory of spelling errors or language processing. Instead, it is

concerned with the capture and modelling of errors on a per-user basis.

Roger Mitton, in a private communication, suggested that the variations in errors made by a single person may well exceed the differences between people, thus making a User Model based system unnecessary. His own thesis was based on a wide range of relatively minor spelling correction devices which, when combined, were highly effective on a variety of non-dyslexic errors.

Helen Pain (Pain 1985) proposed in her suggestions for further work that profiles of poor spellers could be constructed using information from the spelling correction software. This was the intention of the work in this thesis. The conclusion is that such a profile can be built but that it is not adequate to be of any likely use to teachers or computers.

10.3 Sample Text

A corpus of spelling errors was collected from schools, universities and other sources. It was analysed and tagged by hand with written and corrected words as well as identification of the authors and information known about their literacy. It could easily be used for further work beyond the application in this thesis. It was used for the first quantitative study of spelling errors from dyslexic authors. The `java` and `inet` parts of the corpus are unusually detailed; the time and duration of each keystroke was recorded, and such information could be used in future research.

Several of the sections of the corpus are from known dyslexic people (`sunny`, `edin`, `yorksch`, `java`, `yorkcoll`) while others contain a wider variety of authors including students being tested for dyslexia, and individuals who believed they were dyslexia.

The quantity of sample text might be thought to be inadequate. If more were available, and particularly if larger quantities were from diagnosed dyslexic people, then more detailed studies might be feasible (such as an investigation into the location in each word at which each error is likely to occur). As it was, obtaining any samples was time consuming and obtaining large samples from single authors was immensely difficult. The Internet experiment described in Section 6.2 produced less than expected during the time available for realistic collection of data, although it continues to generate more as these conclusions are written. The largest difficulty with the sample text was undoubtedly the size of samples from each author; something which would be very difficult to improve. The `sunny` sample includes all written work by a number of boys during a whole year of school work but is still typically less than ten documents each.

The Internet 'Java' experiment was an interesting exercise in the use of scientific experiments remotely. There is little control over the environment in which people work while they participate in the experiment, and those who choose to finish it are entirely self selecting. Many of the people who downloaded the experiment did not complete it (which is to be expected on the Internet), although it is satisfying to know that most of those who began making a serious effort at it did follow it through to the end. There was no payment for participation which may have discouraged some, although the perceived contribution to technology to assist dyslexic people was enough for most.

The data from the serious participants in the experiment has been valuable for this work and could also contribute to future research in a number of fields.

At the beginning of the Java experiment, participants were asked to declare whether they were dyslexic or not. This information is difficult to analyse because of the differing opinions over severity required to answer 'yes', and because of the fundamental unreliability of a person's answer to such a direct question without

other cognitive metrics. In the end, the answer to that question was not included in the data; instead, all those participants who appeared to complete the experiment sincerely and with a substantial number of errors were used. (Samples with very few errors were inadequate for testing in Babel).

The 'Java' experiment requires considerable computer hardware for participation; it needs a system with a live Internet connection, a World Wide Web browser capable of running Java, and audio output. At the time that the experiment was released, such facilities were relatively scarce although a year later, people were participating at a higher rate suggesting better equipped systems. Technical problems with the Java environment also meant that some computers did not send adequate timing information for individual keystrokes, but that excluded only a very few people.

10.4 Original Contribution

The contribution of the work in this thesis to the body of scientific knowledge has been stated throughout the work, where appropriate. There have been two substantial software tasks, and the auxiliary one of text collection, all of which have made useful contributions.

The work on Babel has examined and presented the results of employing a user model in spelling correction. The work is entirely novel, and it is hoped that future researchers considering not only spelling but also other domains of human errors may gain from its experience; that they may consider employing more sophisticated pattern models and a different user modelling technique. Its use of a simple user model is revealing in its failure, and its use of a rule set for error patterns may also assist future researchers. The comparison with `ispell` and SPEEDCOP clarifies the performance of the work in an empirical manner which is often missing from reports of new methods.

The Path Choice Heuristic in Section 5.4 is a new algorithm for the pruning of surplus data from a set of multiple solutions. This could be employed in a wide range of tasks where multiple solutions are found, each of which contains multiple components. Its range of applications is not limited to human error.

The sample text corpus collected was in itself a significant original contribution to knowledge. It represents the work of a substantial number of people who are normally most reluctant to produce text, prepared and formatted in a manner which is easy to process further.

A student in Hull University has already made use of data from Babel (Smart 1997) for evaluation of errors and another research project in Edinburgh University has taken spelling errors from the sample corpus for a study of existing spelling correction systems.

The Java experiment (Section 6.2) was also an innovative trial in using the Internet for scientific data gathering. In an era when the very word 'Internet' arouses a mixed bag of strong emotions, it has been shown to be useful for easy and accurate data collection subject to a number of practical constraints such as working environment and enthusiasm.

ALGORITHM T is a major new work. Although it may appear similar to existing algorithms it is different enough to provide an improvement of 150 times in execution speed while not requiring 'unit costs' and other assumptions which many other algorithms do. It is better suited to spelling correction tasks than most current approximate string matching algorithms, and works better on English text than it does on randomly generated letter strings. All of these things are both original and good.

The empirical comparison of approximate string matching algorithms in Chapter 9 was also productive in its own right. There has been essentially only one

comparison of methods in practice (Jokinen, Tarhio, and Ukkonen 1996), while most algorithms are presented with theoretical worst case performances which do not indicate the real performance. ALGORITHM T was compared to several other algorithms directly and also compared indirectly through results presented in other papers. The performance of the algorithm under circumstances such as varying edit costs, varying symbol alphabets and varying dictionary sizes were all considered carefully and presented in detail. This evaluation constitutes an original contribution to understanding of the performance of such algorithms under these diverse circumstances, and is generalisable in varying degree to many other algorithms.

10.5 Future Work

A number of avenues of study have been suggested. Considerable work on the sample text was performed, and so it would be useful to make it available to other researchers. To this end, a simple package could be prepared and donated to an archive such as the Oxford Text Archive.

The Internet 'Java' experiment continues to draw in more data, and so an effort to analyse and correct the remaining samples (received since the main analysis of Babel was completed) would be sensible and is called for out of respect for the participants.

The collection of more sample text would be a valuable contribution to any further work. As has been stated above, it is a difficult task and would require close association with a number of schools with dyslexic people or other sources of suitable material. Although a thousand words from each of a thousand people would be useful, a more realistic target size should be chosen first, considering the application to which the text is to be put in a particular project.

In experiments, Babel produced mostly negative results. There is no intention to develop it further as it stands, but it is hoped that lessons can be learnt by future researchers before they embark on similar projects. A new system based on a new model of error patterns would be a sensible future step, as would a system employing a standard user modelling system which may achieve more success in this domain. The choice of user modelling system should not be made lightly; the possible relevance of each to a spelling correction environment should be considered carefully with an understanding that none has been used in a very similar manner before.

Before embarking on a repeat of this work, it would be wise to consider carefully the nature of errors being made, and whether a computer is likely to be able to comprehend them. If the errors are based on the huge knowledge of correct language formation combined with a small number of faults in that knowledge, modelling the faults and simulating the techniques for filling in gaps may not be feasible. The evidence from this work is that the task is not possible under the circumstances tested.

Since so many spelling errors are phonetically accurate but orthographically incorrect, it would be interesting to pursue the topic with a more comprehensive approach to phonetic errors. Such a study would need to consider not only the equivalence of different spellings but also the variety of graphemes used by people in error to represent particular sounds. Many previous projects have given limited consideration to phonetic errors, but none yet has gone so far as to consider pronunciation, simplification of spellings and mis-mapping of graphemes across contexts.

Also, a spelling correction system which considers the grammatical context of the error is something which has been talked about much but not done. Even a relatively simply system employing a statistical model of part-of-speech types and

word frequencies might be able to improve correction hugely, and also to pick up homophonic errors which dyslexic people find so difficult to correct. Data from the British National Corpus of English would be ideal for building such statistical models, and indeed has been used for similar applications such as grammatically correct word prediction (Spooner 1997).

Other principles of spelling correction may also be useful, and it may be the case that a hybrid approach such as that taken by Roger Mitton would be most appropriate to combine the diverse sources of information which are available, but none of which has proven reliable on its own.

ALGORITHM T was a pleasure to develop, and has been shown to be very successful in a highly competitive field of research. A publication on the subject should be prepared as soon as time allows. Its incorporation into more substantial products would be welcomed, as would the development of newer and better methods based on its success. A development specifically for spelling correction with varying costs of editing operations would be of interest.

Bibliography

Andersson, A., N. J. Larsson, and K. Swanson (1996). Suffix Trees on Words. In G. Goos, J. Hartmanis, and J. van Leeuwen (Eds.), *Combinatorial Pattern Matching CPM96*, Number 1075 in LNCS, pp. 102–115. New York: Springer.

Backhouse, R. (1979). *The Syntax of Programming Languages: Theory and Practice*. London: Prentice-Hall International.

Baeza-Yates, R. and N. Gonzalo (1996). A Faster Algorithm for Approximate String Matching. In G. Goos, J. Hartmanis, and J. van Leeuwen (Eds.), *Combinatorial Pattern Matching CPM96*, Number 1075 in LNCS, pp. 1–23. New York: Springer.

Baeza-Yates, R. A. and C. H. Perleberg (1996). Fast and practical approximate string matching. *Information Processing Letters 59*, 21–27.

Baker, B. (1982). MinSpeak. *Byte, the small systems journal 9*(September), 186–202.

Bauer, M. (1995). A Dempster-Shafer approach to modelling agent preferences for plan recognition. *User Modelling and User Adapted Interaction 5*, 317–348.

Bennett, M. (1976). SUBSTITUTOR: A teaching program. Technical report, Department of Artificial Intelligence, University of Edinburgh.

Blair, C. R. (1960). A program for correcting spelling errors. *Information and Control 3*, 60–67.

Boder, E. (1973). Developmental Dyslexia: a Diagnostic Approach based on

Three Atypical Reading-spelling Patterns. *Developmental Medicine and Child Neurology 15*, 663–687.

Brown, J. and R. Burton (1978). Diagnostic Models for procedural bugs in basic mathematical skills. *Cognitive Science 2*, 155–192.

Catford, J. (1988). *A Practical Introduction to Phonetics*. Oxford: Oxford University Press.

Chin, D. N. (1989). KNOME: Modelling what the knows in UC. In A. Kobsa and W. Wahlster (Eds.), *User models in dialog systems*. Berlin: Springer.

Cobbs, A. (1995). Fast Approximate String Matching using Suffix Trees. In *CPM '95*, Number 937 in LNCS, Espoo, Finland, pp. 41–54. Springer-Verlag.

Coltheart, M., B. Curtis, P. Atkins, and M. Haller (1993). Models of Reading Aloud: Dual-Route and Parallel-Distributed-Processing Approaches. *Psychological Review 100*(4), 589–608.

Coltheart, M., G. Sartori, and R. Job (1987). *The Cognitive neuropsychology of language*. Lawrence Erlbaum Associates Ltd.

Cresswell, I., D. Monteith-Hodge, and M. Winfield (1997). An Approach to the Diagnosis and Remediation of Developmental Dyslexia Using Microcomputers. In IEE (Ed.), *Computers in the Service of Mankind: Helping the Disabled*. IEE Computing and Control Division Professional Group C14.

Damerau, F. (1964). A Technique for the Computer Detection and Correction of Spelling Errors. *Communications of the ACM 7*(3), 171–176.

Du, M. and S. Chang (1992). A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica 29*, 281–302.

Du, M. and S. Chang (1994). An Approach to Designing Very Fast Approximate String Matching Algorithms. *IEEE Transactions on Knowledge and Data Engineering 6*(4), 620–633.

Edwards, A. D. (1991). *Speech Synthesis - Technology for Disabled People*. London: Paul Chapman Publishing.

El-Mabrouk, N. and M. Crochemore (1996). Boyer-Moore Strategy to Efficient Approximate String Matching. In G. Goos, J. Hartmanis, and J. van Leeuwen (Eds.), *Combinatorial Pattern Matching CPM96*, Number 1075 in LNCS, pp. 24–38. New York: Springer.

Elkind, J. and J. Shrager (1995). Modeling and analysis of dyslexic writing using speech and other modalities. In A. D. Edwards (Ed.), *Extra-Ordinary Human-Computer Interaction: Interfaces for Users with Disabilities*. New York: Cambridge University Press.

Ellis, A. W. (1984). *Reading, Writing and Dyslexia*. London: Lawrence Erlbaum.

Finlay, J. E. (1990). *Modelling Users by Classification: An Example-Based Approach*. Ph. D. thesis, University of York, UK.

Forgy, C. and J. McDermott (1977). OPS, a domain-independent production system language. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pp. 933–939.

Frith, U. (1997). Brain, Mind and Behaviour in Dyslexia. In C. Hulme and M. Snowling (Eds.), *Dyslexia: Biology, Cognition and Intervention*, pp. 1–19. London: Whurr Publishers Ltd.

Gadd, T. (1990). PHONIX: The algorithm. *Program: Automated Library and Information Systems 24*(4), 363–366.

Galaburda, A. M., M. T. Menard, and G. D. Rosen (1994). Evidence for aberrant auditory anatomy in developmental dyslexia. *Proceedings of the National Academy of Sciences 91*, 8010–8013.

Greene, R. L. (1994). Efficient retrieval from sparse associative memory. *Artificial Intelligence 66*, 395–410.

Hanna, P., J. Hanna, R. Hodges, and E. Rudorf (1966). *Phoneme-Grapheme Correspondences as Cues to Spelling Improvement*. Washington DC: US Office of Education.

Hewitt, J. (1998). Current Developments in Low-Budget Speech Recognition Systems. In A. D. Edwards, A. Arato, and W. L. Zagler (Eds.), *Computers and Assistive Technology ICCHP '98*, pp. 474–483. Vienna: Austrian Computer Society.

Hinshelwood, J. (1917). *Congenital Word-Blindness*. London: H.K. Lewis.

Hinton, G. E. and T. Shallice (1991). Lesioning an Attractor Network: Investigations of Acquired Dyslexia. *Psychological Review 98*(1), 74–95.

Horvitz, E. and M. Barry (1995). Display of information for time-critical decision making. In P. Besnard and S. Hanks (Eds.), *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 296–314. San Francisco: Morgan Kaufmann.

Hull, J. J. (1987). Hypothesis Testing in a Computational Theory of Visual Word Recognition. *Proceedings of AAAI87 2*, 718–722.

Hutchinson, L. (1956). *Standard Handbook for Secretaries*. New York: McGraw-Hill.

Jameson, A. (1992). Generalising the double-stereotype approach: a psychological perspective. In E. Andre, R. Cohen, W. Graf, et al. (Eds.), *Proceedings of the Third International Conference on User Modelling*, pp. 69–83. Saarbrucken, Germany: German Research Centre for Artificial Intelligence.

Jameson, A. (1996). Numerical Uncertainty Management in User and Student Modelling: An Overview of Systems and Issues. *User Modeling and User-Adapted Interaction 5*, 193–251.

Jokinen, P., J. Tarhio, and E. Ukkonen (1996). A Comparison of Approximate String Matching Algorithms. *Software - Practice and Experience 26*(12), 1329–

1458.

Leech, G. (1992). 100 million words of English: the British National Corpus. *Language Research 28*(1), 1–13.

Link, K. and A. Caramazza (1994). Orthographic Structure and the Spelling Process: A Comparison of Different Codes. In G. D. Brown and N. C. Ellis (Eds.), *Handbook of Spelling: theory, process and intervention*, pp. 261–294. Chichester: John Wiley & Sons Ltd.

Lirov, Y. (1991). Algorithmic Multiobjective Heuristics Construction in the A-Star Search. *Decision Support Systems 7*(2), 159–167.

Lowrance, R. and R. Wagner (1975). An extension of the string-to-string correction problem. *Journal of the Association for Computing Machinery 5*, 177–183.

Marshall and Newcombe (1973). Patterns of Paralexia: A Psycholinguistic Approach. *Journal of Pyscholinguistic Research 2*(3), 175–199.

McCoy, K. F., C. A. Pennington, and L. Z. Suri (1996). English Error Correction: A Syntactic User Model Basedon on Principled 'Mal-Rule' Scoring. In D. Chin, S. Carberry, and I. Zukerman (Eds.), *Proceedings of UM96, Hawaii*, pp. 59–66. UM, Inc.

McIlroy, M. D. (1974). *Synthetic English Speech by Rule*. Bell Telephone Laboratories, Inc.

Miles, T. and E. Miles (1990). *Dyslexia: a hundred years on*. Buckingham: Open University Press. Description of problem for interested parents and other educated people new to the field.

Mitchell, J. and C. Welty (1988). Experimentation in computer science: An empirical view. *International Journal of Man-Machine Studies 29*, 613–624.

Mitton, R. (1996). *English Spelling and the Computer*. London: Addison-Wesley Longman.

Morgan, W. (1896). A case study of congenital word blindness. *British Medical Journal 2*, 1378.

Nelson, M. (1996). Fast String Searching With Suffix Trees. *Dr. Dobb's Journal 1996*(August).

Newell, A. F., L. Booth, and W. Beattie (1991). Predictive text entry with PAL and children with learning difficulties. *British Journal of Educational Technology 22*(1), 23–40.

Nicolson, R. I. and A. J. Fawcett (1993). Computer Based Spelling Remediation for Dyslexic Children Using the SelfSpell Environment. In S. Wright and R. Groner (Eds.), *Facets of Dyslexia and its Remediation*, pp. 551–565. Elsevier Science Publishers B.V.

Nisbet, P., E. Arthur, and R. Spooner (1998). *Supportive Writing Software*. Edinburgh University: CALL Centre.

Odell and Russell (1922). US. Patent 1,453,663 (Soundex).

Owolabi, O. (1996). Dictionary Organisations for Efficient Similarity Retrieval. *Journal of Systems and Software 34*(2), 127–132.

Pain, H. (1985). *A Computer Tool for Use by Children with Learning Difficulties in Spelling*. Ph. D. thesis, University of Edinburgh.

Patterson, K., M. Seidenberg, and J. McClelland (1989). Connections and disconnections: Acquired Dyslexia in a computational model of reading processes. In R. G. M. Morris (Ed.), *Parallel Distributed Processing: Implications for Psychology and Neurobiology*, pp. 131–181. Oxford: Clarendon Press.

Patterson, K. and C. Shewell (1987). Speak and Spell: Dissociations and Word-Class Effects. In M. Coltheart, G. Sartori, and R. Job (Eds.), *The Cognitive Neuropsuchology of Language*, pp. 273–294. London: Lawrence Erlbaum Associates.

Paulesu, E., U. Frith, and M. e. a. Snowling (1996). Is developmental dyslexia a disconnection syndrome? Evidence from PET scanning. *Brain 119*, 143–157.

Pollock, J. J. and A. Zamora (1984). Automatic Spelling Correction in Scientific and Scholarly Text. *Communications of the ACM 27*(4), 358–367.

Reid, B. (1980). *Scribe: a document specification language and its compiler*. Ph. D. thesis, Carnegie Mellon University.

Rich, E. (1979). User Modeling via Stereotypes. *Cognitive Science 3*, 329–354.

Rich, E. (1983). Users are individuals: individualizing user models. *International Journal of Man-Machine Studies 18*, 199–214.

Russell, R. C. (1918). US. Patent 1,261,167 (Soundex).

Seidenberg, M. and J. McClelland (1989). A distributed, developmental model of word recognition and naming. *Psychological Review 96*, 523–568.

Sejnowski, T. J. and C. R. Rosenberg (1987). Parallel Networks that Learn to Pronounce English Text. *Complex Systems 1*, 145–168.

Shafer, G. and A. Tversky (1985). Languages and designs for probability judgement. *Cognitive Science 9*, 177–210.

Shang, H. and T. Merrettal (1996). Tries for Approximate String Matching. *IEEE Transactions on Knowledge and Data Engineering 8*(4), 540–547.

Singleton, C. (1994). *Computers and Dyslexia*. Hull: The Dyslexia Computer Resource Centre, Hull.

Smart, E. (1997). *An Analysis of Children's Spelling Development*. University of Hull: Department of Psychology.

Spooner, R. (1997). *Penfriend, the Predictive Typer*. Edinburgh, Scotland: Design Concept. User manual for Penfriend for Windows.

Spooner, R. I. and A. D. Edwards (1997a). A Computer-Based Spelling Aid for Dyslexic People. In M. Snowling (Ed.), *Dyslexia: Biological Bases, Identification*

and Intervention. British Dyslexia Association.

Spooner, R. I. and A. D. Edwards (1997b). User Modelling for Error Recovery: A Spelling checker for Dyslexic Users. In A. Jameson, C. Paris, and C. Tasso (Eds.), *User Modelling - Proceedings of the Sixth International Conference UM97*, New York. Springer Wien.

Tallal, P., S. L. Miller, G. Bedi, et al. (1996). Langage Comprehension in Language-Learning Impaired Children Improved with Acoustically Modified Speech. *Science 271*, 81–84.

Ukkonen, E. (1992). Approximate string matching with *q*-grams and maximal matches. *Theoretical Computer Science 92*(1), 191–211.

Ukkonen, E. (1993). Approximate String matching over suffix trees. In *CPM'93*, LNCS 684, pp. 228–242.

Veronis, J. (1988). Computerised correction of phonographic errors. *Computers and Humanities 22*(1), 43–56.

Wagner, R. and M. Fischer (1974). The String to String Correction problem. *Journal of the Asscociation for Computing Machinery 21*(1), 168–178.

Webb, G. I. and M. Kuzmycz (1996). Feature Based Modelling: A methodology for producing coherent, consistent, dynamically changing models of agents' competencies. *User Modeling and User Adapted Interaction 5*, 117–150.

Weiner, P. (1973). Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pp. 1–11. IEEE?

Yannakoudakis, E. and D. Fawthrop (1983a). An Intelligent Spelling Error Corrector. *Information Processing and Management 19*(2), 101–108.

Yannakoudakis, E. and D. Fawthrop (1983b). The Rules of Spelling Errors. *Information Processing and Management 19*(2), 87.

Young, R. M. and T. O'Shea (1981). Errors in Children's Subtraction. *Cognitive*

Science 5(2), 153–177.

Zadeh, L. A. (1994). Fuzzy Logic, neural networks and soft computing. *Communications of the ACM 37*(3), 77–84.

Zobel, J. and P. Dart (1995). Finding Approximate Matches in Large Lexicons. *Software – Practice and Experience 25*(3), 331–345.