Approximate String Matching and the Automation of Word Games

†Hal Berghel
†David Roach
‡John Talburt

†The University of Arkansas
‡The University of Arkansas at Little Rock

Abstract

In this paper we discuss the utility of approximate string matching procedures in the automation of various aspects of word game construction and solution. These procedures are then related to the underlying issues in computational linguistics.

Introduction

Word games may be enjoyed at a number of different levels. The neophyte may derive satisfaction from an incomplete attempt at solving a crossword puzzle. The serious practitioner may find pleasure at a loftier plateau, perhaps quickly solving a very difficult cryptogram or a lengthy anagram. Possibly some are attracted by the attendant silent soliloquy or the individuality of the effort. For whatever reasons, word games in the twentieth century are enormously popular and, for some, have come to be associated with intelligence and erudition.

However, word games are also interesting because they illustrate classes of problems which are of pressing concern to the computational linguist. In the automation of various aspects of word games, one deals with such issues as efficient lexical organization and processing, exact and approximate string matching techniques, search strategies and heuristics, problem representation, knowledge representation, rule based expert systems, and so forth. In this paper, we describe the interrelationships between these issues and illustrate how they impact current approaches toward automating word games.

The Crux of Cruciverbalism

Modern crossword puzzles are as old as this century. They are typically defined upon an mxn matrix where most, if not all, of the cells are place holders for characters. Consecutive characters make up words along the horizontal and vertical axes. These words are semantically related to a 'clue' which is usually associated with the ID of the first cell in the word slot.

TH0307-9/90/0000/0209$01.00 © 1990 IEEE

Of course there are numerous variations on this theme. Some puzzles, like those commonly found in British newspapers, are more sparse (i.e., have a lower density or percentage of unfilled or open cells) than others. Some theme puzzles are defined over non-rectangular shapes (perhaps a heart shape for Valentine's Day, or a tree shape for Christmas). Others may require that all open cells interlock horizontal and vertical word slots which include them (as in the typical American puzzle). Though the format of crossword puzzles may differ, they all can be described in terms of these three characteristics: the geometry of the puzzle, the density of the puzzle and the degree of interlocking.

Crossword puzzles are created in stages. To use the terminology of Smith and Steen [24], we refer to the creation as 'crossword compilation'. The following operations are involved in crossword compilation: 1) creation of host matrix, 2) determination of overall design, 3) specification of word slots, 4) identification of occurrences of cell sharing, 5) construction of solution set(s), and 6) composition of clue set(s) for the solution set(s).

We note that the typical solution of the puzzle involves stages 5) and 6) in reverse order. Stages 1) through 4) are given consideration in the construction of the puzzle. Although the end-user normally does not give a great deal of thought to these initial stages, they very much affect the aesthetics and recreational value of the puzzle.

Each of these stages have some interest to the computational linguist as well. Without question, most of the literature involves stage 5). Although the algorithms are of only historical interest now, Mazlack's pioneering work on the generation of solution sets [15,16,17] remain useful references for they first defined the problem and first called attention to the fact that crossword compilation characterized several interesting issues in computational linguistics. It was Smith and Steen [24] who first came to understand and address the computationally problematic aspects of compilation. Further, they set many of the standards against which current work is measured. In addition, much of the current nomenclature derives from their work.

In recent years, several efficient solution algorithms have been developed [3,14,24,28], based upon a wide variety of computational paradigms. In addition, significant work has been done on the automation of clue set construction [23], the technical aspects of automating the early stages of crossword compilation [9], the human factors aspect [25,27], search heuristics [9,15,16], performance analysis [5] and estimation of the size of the solution set [12,13].

However, it is stage 5) which continues to receive the greatest attention. Specifically, most investigators have been concerned with 1) the development of efficient search strategies for lexical look-up and 2) with heuristics which narrow the search space and/or reduce the complexity of the problem (see, e.g., [3,14,24] for recent illustrations). It is easy to see why this is the case by referring to Figure 1.

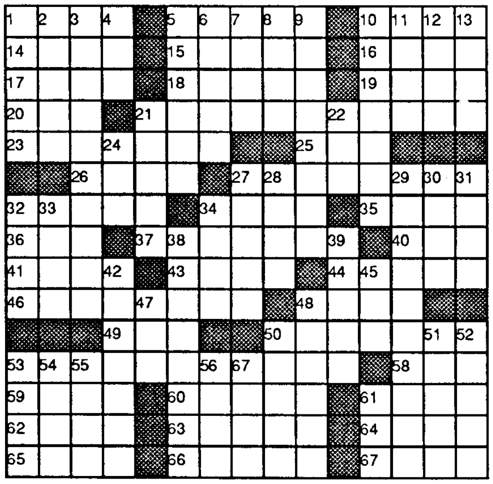

Figure 1: Typical American Crossword Puzzle

Suppose that we wish to compile this puzzle. This entails the construction of a solution set where all of the words interlock appropriately (we note that there are different conventions which are employed in different 'crossword cultures') according to the geometry and density of the particular puzzle. Lexical insertion becomes more complicated as we move progressively through the sub-regions of the puzzle. The reason for this is that there are increasingly many constraints placed on inserted words. To illustrate, suppose that we have completely filled out the top-left region. We note that 23-across carries into the top-middle region. Thus, the insertion of 21-down is constrained from the start by an 'inherited' character in position 2. Denoting this character by the variable, X, we're looking for a word of the form '_ X _ _ _'. Of course, this compounds our lexical look-up procedures: sublinear algorithms are no longer appropriate and we lose our entry point into the lexicon without the first character.

Of course, we could organize the lexicon by word length, but this would only solve part of the problem for it alone would require sequential searches of the partitions. In general, most such searches will fail because of the non-uniform distribution of characters within word positions. A better approach would be to test within the constraints imposed by the very insertion, itself. In other words, we will filter our database look-ups in advance. One common filtration technique is n-gram analysis (cf., [4,11,19]).

An n-gram of a string of characters is any segment of length n within the string. Thus, the digrams of 'ABC' are 'AB' and 'BC'. Typically, for computational linguistics purposes, strings are padded with n-1 spaces for n-gram analysis, so that each character appears in exactly n n-grams, but we'll ignore this variation for the sake of simplicity.

N-gram analysis is useful precisely because of the non-uniform distribution of characters within the word positions, the phenomena which created the problems for us. A list of legitimate n-grams is extracted from the lexicon (we can even increase the accuracy by relating n-grams to positions within words). We note that in the case of our current example, the failure to produce trigrams of the form '_ X _' and 'X _ _' means that there is no way for the string '_ X _ _ _' to be matched with any lexical entry. Since this happens more often than not, efficiency is achieved through reduced lexical look-ups (see refs. [4,19] for details).

Thus, n-gram analysis is one of a number of filtration techniques which increase the efficiency of the lexical processing. Equally important, however, are the heuristics which are used to reduce the complexity of the puzzle. These heuristics frequently arise at several different levels in the compilers design. To illustrate, let's return again to Figure 1. Were we to insert words in the sequence 1-across, 14-across, 17-across, 20-across, 23-across, 1-down, 2-down, 3-down, 4-down, we would have no 'fail points' prior to the sixth insertion. However, efficient compilation will rule out impossible combinations of the interlocking words as soon as possible so as to narrow the search space. Were we to proceed 1-across, 1-down, 14-across, 2-down, etc., we would have one fail point after the second insertion, two after the third, etc. It is plain to see that since most of the possible word combinations will not conform to the rules of the puzzle, it is to our advantage to proliferate fail points as high in the search tree as possible.

A refinement of the strategy of failing as early as possible applies to migration between sub-regions of the puzzle. Again, in the top-left region of our puzzle there are 2 words of length 3, 3 of length 4, 2 of length 5, 1 of 7 and 1 of 10. Since there are considerably fewer words of length 3 and 10 than there are of lengths 4, 5 and 7, these slots will be harder to fill. Thus, we may adapt our strategy to the more general case by beginning with the toughest word slots prior to alternating between horizontal and vertical insertions. In the literature, this is called 'neighborhood prioritization'. (For additional details on compilation/ solution strategies, see references [3,9]).

As we shall see below, the quest for efficient lexical searching (usually involving some sort of filtration) and the development of sophisticated heuristics are the dominant computational linguistic themes in the automation of word games.

Anagrams

Anagrams are transpositions of words. 'OGD' is anagrammatic for 'GOD' and 'DOG'; 'DLSE' and 'EDSL' for 'SLED', and so forth. Faulk [10] classifies anagrams as strings with material identity, meaning that both the anagram and the related word have the same character composition.

Anagrams are a pleasantly different type of word game. Unlike crossword puzzles, the character content of an anagram is given in advance; it is simply a matter of finding the transposition(s) of the characters which are legitimate words. Like crossword compilation, filtration has been found to be an important component of efficient anagrammatic processing. And once again, n-gram analysis has been found to be useful. The procedure is similar to those described in the previous section, but the tested n-grams are variable-free.

For an anagram of length n, the worst case obviously involves n! transpositions to check. A straightforward algorithm might work like this. First, one transposes the anagram into one of the n! materially similar tokens. For each one, we check the constituent n-grams within the token, left to right, to determine whether they are legitimate with respect to the lexicon in use. If an n-gram is found to be illegitimate, the entire transposition is rejected. When all n-grams are legitimate, the transposition is compared with the lexicon. Upon failure, the next transposition is tried, etc.

Unlike the case with crossword compilation, n-gram analysis has been found to be sub-optimal for the filtration of anagram transpositions [6]. The reason is that n-gram analysis ignores the additional information which a complete transposition provides. Since there is certainty concerning the character composition of the actual word(s), this information can be advantageous. One way of doing this is by basing the filtration upon the distribution of vowel and consonant patterns found in words of that length.

For present purposes, we will assume that a,e,i,o,u and y are vowels and the remaining characters of the alphabet are consonants. The theoretical distribution of vowel to consonant ratios for words of length 6 would be:

V-C Ratio	N
6-0	1
5-1	6
4-2	15
3-3	20
2-4	15
1-5	6
0-6	1

while the actual distribution for a particular lexicon might be

V-C Ratio	N
6-0	0
5-1	0
4-2	10
3-3	17
2-4	15
1-5	5
0-6	0.

We may put this knowledge to work for us in filtering the anagram transpositions. For example, suppose that we have an anagram with a 4-2pattern. We know from the data above that of the 15 possible patterns, only 10 are to be found in the lexicon. Thus, 33% of the possible comparisons may be eliminated. While the a priori advantage of v-c pattern filtration has yet to be determined, there is empirical evidence which suggests that for smaller lexicons at least it offers a better average-case behavior than trigramanalysis [6].

As with crossword compilation, increased efficiency for lexical matching is the dominant theme in anagram unraveling. So far, no one has suggested the need for heuristics.

Palindromes

Palindromes are strings of symbols which are symmetrical about the center. The most basic form of palindromes are orthographical palindromes, meaning that they have no meaning above and beyond their symmetry (e.g, 'ABCBA'). Lexical palindromes are at the next level. A lexical palindrome is an orthographical palindrome for which there exists some partitioning into legitimate words (e.g., 'REFER', 'TEN NE PEN NET', 'REVEST AH THAT SEVER'). Phrasal palindromes are next (e.g., 'DIARY RAID'); and finally, at the pinnacle, sentential palindromes ('ABLE WAS I ERE I SAW ELBA', 'A MAN A PLAN A CANAL PANAMA'). The beauty of his last type is that the sentence is formed with an orthographical 'joint'. The interesting question for the computational linguist is whether an efficient procedure may be found to generate these sentential palindromes. While we know of no success at this writing, lexical palindromes have been assaulted, if not conquered [20].

Just as there are an infinitude of orthographical palindromes, there are an infinitude of lexical palindromes: simply by adding a palindromic word (e.g., 'ANNA') to each end of a lexical palindrome generates another lexical palindrome. However, there are only a finite number of lexical palindromes of a fixed length with respect to any given lexicon. This provides an attract research environment for lexical palindrome study.

As far as we can determine, there has been only one attempt to develop an effective means for the generation of lexical palindromes [20]. The simplest strategy would be to generate palindromes from the Roman Alphabet, and then to test to see if they are also lexical palindromes. The main difficulty with this approach derives from the fact that the generation is at the orthographical level where the testing is at the lexical level. This means that there will be an enormous amount of time spent by the system partitioning and checking mostly unrecognizable output thereby applying a factorial process to an exponential output culminating in a combinatorial explosion.

To ameliorate the problem, one might attempt a word-based approach. This would ensure that the segments inserted are already lexically correct. For example, we might insert words, from left to right, until we reach the middle position, and then append the reversed input to complete the string. The problem here is that the earliest lexical insertions may create reversed patterns which defy lexical partitioning. That is, we will spend too much time working with unworkable lexical combinations.

One way to avoid this difficulty is to restrict lexical insertions to those words whose reversals conform to some acceptable lexical partition. For example, it would be pointless to insert 'SEEK' in the first position of an eight character palindrome for 'KEES' cannot be partitioned in any lexically meaningful way. This approach may be further improved if we continuously check the reversed strings for acceptability rather than deferring the judgment until the entire orthographical palindrome has been generated.

An even more selective strategy is outlined in [20]. In this case, two lists are maintained in processing, the concatenation of which contains a palindrome. After each word is inserted, the resulting reversed string is analyzed and the constraints are determined for the subsequent word. One of three conditions must obtain: 1) either the reversed list can be partitioned in such a way that all parts are already words (even length palindrome), or 2) the reversed list together with the last character in the forward list can be so partitioned (odd length palindrome), or 3) the reversed list can be partitioned so that all parts except the left-most part are words, and that left-most part is the ending of some word or other which fits the remaining slot. This ending is then used to index further search.

Perhaps an example will clarify the procedure. Suppose that our forward list contains 'AS'. The reversed list will contain 'SA'. 'SA' is not in the lexicon, but 'A' is. So the left most part, 'S', is used as an index to select another word for insertion from the backwards dictionary (perhaps, 'POTS'). Since the lexical insertions are bi-directional the fail points are driven higher in the search tree than would be the case with uni-direction insertion. Of course, this technique works best with two lexicons: a standard lexicon and one which has all of the words spelled backwards. However, the resulting efficiencies have at least made the goal of generating lexical palindromes tractable.

Conclusion

The discussion above has outlined how approximate string matching may be involved in the automation of various aspects of word game construction or solution. In the discussion, we have tried to identify and explicate the underlying issues in computational linguistics, and suggest techniques which have been used to address these issues. As we can see, the two main issues which arise in this context have to do with lexical processing and heuristics, issues which arise in a more practical contexts as well [1,2,4,18]. For an excellent introduction to these topics, especially as they relate to text processing, see reference [26].

References

[1] Berghel, H., "Extending the Capabilities of Word Processing Software through Horn Clause Lexical Databases", Proceedings of the 1986 National Computer Conference, pp. 251-257, 1986.

[2] Berghel, H., "A Logical Framework for the Correction of Spelling Errors in Electronic Documents", Info. Proc. and Mgmt., V. 23, pp. 477-494, 1987.

[3] Berghel, H., "Crossword Compilation with Horn Clauses", The Computer Journal, V. 30, N. 2, pp. 183-188, 1987.

[4] Berghel, H. and C. Andreu, "TALISMAN: A Prototype Expert System for the Detection and Correction of Spelling Errors:, Proc. 1988 ACM Symp. on Small Systems, pp. 107-113, 1988.

[5] Berghel, H. and R. Rankin, "A Proposed Standard for Measuring Crossword Compilation Efficiency", The Computer Journal [in press].

[6] Berghel, H., R. Rankin and D. Roach, "Efficient Lexical Processing with V-C Pattern Filtration", Proceedings of the Third Oklahoma Symposium on Artificial Intelligence, pp. 149-154, 1989.

[7] Berghel, H., D. Roach and J. Talburt, "The Mechanical Cruciverbalist", PC/AI, Vol. 3, No. 6, pp. 45-47, 1989.

[8] Berghel, H., D. Roach and J. Talburt, "The Logic of Spelling", PC/AI, Vol. 4, No. 1, pp. 24-27, 1990.

[9] Berghel, H. and C. Yi, "Crossword Compiler Compilation", The Computer Journal, V. 32, N. 3, pp. 276-280, 1989.

[10] Faulk, R., "An Inductive Approach to Language Translation", Communications of the ACM, Vol. 7, pp. 647-653, 1964.

[11] Hall, P. and G. Dowling, "Approximate String Matching", Comp. Surv., V. 12, pp. 381-402, 1980.

[12] Harris, G., "Generalization of Solutions Sets for Unconstrained Crossword Puzzles", Proceedings of the 1990 Symposium on Applied Computing [in press].

[13] Harris, G. and J. Forster, "On the Bayesian Estimation and Computation of the Number of Solutions to Crossword Puzzles", Proceedings of the 1990 Symposium on Applied Computing [in press].

[14] Harris, G. and J. Spring, "An Efficient Algorithm fro Crossword Puzzle Solutions", Division of CAD, Griffith University, Nathan, Australia, 1989 [manuscript].

[15] Mazlack, L., "The Use of Applied Probability in the Computer Construction of Crossword Puzzles", Proceedings of the IEEE Conference on Decision and Control, pp. 497-506, 1973.

[16] Mazlack, L., "Computer Construction of Crossword Puzzles Using Precedence Relationships", Artificial Intelligence, V. 7, N. 1, pp. 1-19, 1976.

[17] Mazlack, L., "Machine Selection of Elements in Crossword Puzzles: An Application of Computational Linguistics", SIAM Journal on Computing, V. 5, N. 1, pp. 51-72, 1976.

[18] Pollock, J. and A. Zamora, "Automatic Spelling Correction in Scientific and Scholarly Text", CACM, V. 27, pp. 358-368, 1984.

[19] Rankin, R. "Increasing the Efficiency of Prolog Lexical Databases with N-Gram Boolean Cubes", Proc. 1988 ACM Symp. on Small Systems, pp. 161-166, 1988.

[20] Rankin, R., H. Berghel and T. Xu, "Efficient Generation of Lexical Palindromes", Proceedings of the 1990 ACM Symposium on Small Systems [in press].

[21] Roach, D., H. Berghel and J. Talburt, "Intelligent Problem Solving with Strings: New Directions in Approximate String Matching", PC/AI, Vol. 3, No. 5, pp. 24-27, 1989.

[22] Roach, D., H. Berghel and J. Talburt, "Multi-level ASM", PC/AI, Vol. 4, No. 2, pp. 17-19 & 47, 1990.

[23] Smith, G. and J. duBoulay, "The Generation of Cryptic Crossword Clues", The Computer Journal, V. 29, N. 3, pp. 282-284, 1986.

[24] Smith, P. and S. Steen, "A Prototype Crossword Compiler", The Computer Journal, V. 24, N. 2, pp. 107-111, 1981.

[25] Smith, P., "XENO: Computer-Assisted Compilation of Crossword Puzzles", The Computer Journal, V. 26, N. 4, pp. 296-301, 1983.

[26] Smith, P., An Introduction to Text Processing, Cambridge, M.I.T. Press, 1990.

[27] Williams, P. and D. Woodhead, "Computer Assisted Analysis of Cryptic Crosswords", The Computer Journal, Vol. 22, No. 1, pp. 67-70, 1979.

[28] Wilson, J., "Crossword Compilation Using Integer Programming", The Computer Journal, Vol. 32, No. 3, pp. 273-275, 1989.