Crossword Compilation with Horn Clauses

H. BERGHEL

Department of Computer Science, University of Arkansas, Fayetteville, AR 72701, USA

Because of the widespread interest in crossword puzzles, serious attention has been given in recent years toward the development of efficient algorithms for their generation and solution. In this paper we describe a new approach which is based upon an analysis of the problem expressed in first-order predicate logic. This analysis is then used to construct a crossword compiler in Prolog. The kernel of the program is then presented and described.

Received November 1985

1. INTRODUCTION

A crossword puzzle is a word game defined upon an **m × n** matrix where most, if not all, of the cells are filled in with characters which comprise words along horizontal and/or vertical axes. The puzzle solver uses 'clues' provided with the puzzle to narrow the range of acceptable characters and words. In finding a solution, the player seeks to associate with each clue a related word or phrase with desirable orthographical properties.

Such puzzles may be described in terms of at least three characteristics: geometry, density and degree of interlocking. The *geometry* of a crossword puzzle is defined by the size of the matrix and the distribution of closed and open cells. *Closed cells*, which usually appear as solid boxes, are not actually parts of the puzzle, but are used to mark internal word boundaries. *Open cells*, on the other hand, are the business part of the puzzle. They are filled in with characters which make up the words. In Romance languages, these works are formed by the concatenating horizontally or vertically contiguous characters, reading from top to bottom or left to right. We refer to these contiguous cells as *word slots*.

Although each cell has an implicit address associated with it (row and column indices), by custom only those open cells which begin words are numbered in the puzzle. The termination of the word slot may be indicated by a closed cell, the border of the puzzle or a numbered cell which signifies the start of a new word on the same line or column.

The *density* of a puzzle refers to the percentage of open cells. In the extreme case where there are no closed cells, the puzzle will be described as *full*.

The last parameter by which we may compare crossword puzzles is the *degree of interlocking* of words. We say that two word slots are *interlocked* when they share at least one open cell. Typically, interlocking is *orthogonal*. In this case, a horizontal word slot intersects a vertical word slot. We will refer to the open cell at which this intersection occurs as the *orthogonal intercept*.

The percentage of open cells which are shared provides the *degree of interlocking*. If all open cells of a crossword puzzle are shared, the puzzle will be said to be *completely interlocked*. Generally speaking, modern crossword puzzles exhibit a high degree of interlocking, and most of the American puzzles that we have seen are completely interlocked. The reader should note the nomenclature used in this section is slightly different from that employed in Smith and Steen.[8] For an alternative treatment, see Mazlack.[6]

Fig. 1 is an illustration of a typical American crossword puzzle.

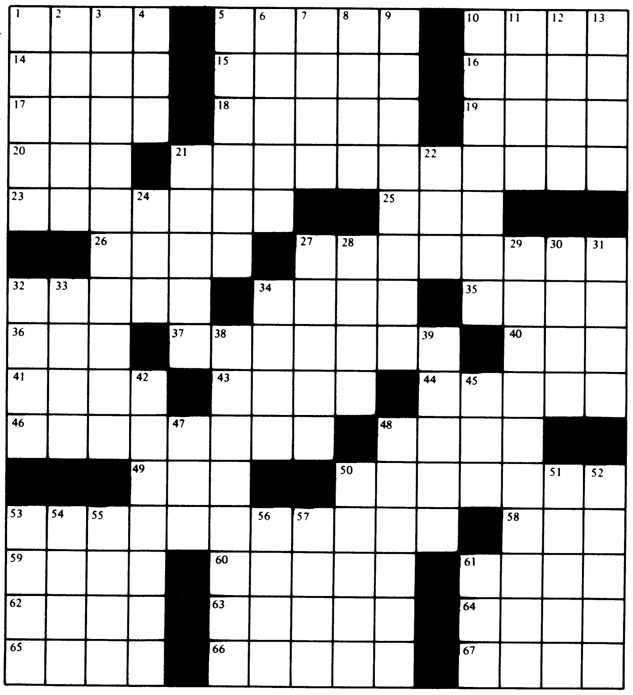

Figure 1. Puzzle defined on 15 × 15 matrix. Degree of interlocking: complete; density: 84%.

2. CROSSWORD COMPILATION

'Crossword compilation' is the phrase used by Smith and Steen[8] to refer to the stages in the creation of crossword puzzles. In the terminology of the previous section, crossword compilation can be seen to involve the following operations:

(1) creation of the host matrix,
(2) determination of the overall design (i.e. pattern of open and closed cells) within the matrix,
(3) specification of word slots,
(4) identification of shared cells,
(5) construction of one or more solution sets, and
(6) composition of a clue set for each solution set.

The solution of the crossword puzzle amounts to performing the sequence (5)–(6) in reverse order.

For present purposes we will concentrate upon stage (5). In particular, we intend to describe a simple procedure which will derive all solution sets from a crossword puzzle with a certain geometry.

3. A LOGICAL APPROACH TO FINDING SOLUTION SETS

Assume that we have a crossword puzzle which employs only orthogonal interlocking. Suppose that we have two word slots, of length m and n $(m, n \geqslant 1)$, respectively. Further suppose that the orthogonal intercept is cell i $(1 \leqslant i \leqslant m)$ in the first word slot and j $(1 \leqslant j \leqslant n)$ in the second. In attempting to solve or create this part of the puzzle, we in effect make the following claim: there exist two words in our dictionary which share the same character in positions i and j, respectively. Formally, we say

$$(\exists X_1, \ldots, X_i, \ldots, X_m)(\exists Y_1, \ldots, Y_j, \ldots, Y_n)$$
$$(word(X_1, \ldots, X_i \ldots, X_m)$$
$$\& \; word(Y_1, \ldots, Y_j, \ldots, Y_n) \,\&\, X_i = Y_j)$$

where X_i/Y_j is the orthogonal intercept, the predicate, 'word', is interpreted as 'the result of concatenating the following argument names is a word', and the domain of discourse is essentially the set of characters of the Roman alphabet. Further, we note that the identity operator is actually unneeded, for we can say the same thing by using a common variable in both arguments. This is,

$$(\exists X_1, \ldots, X_m)(\exists Y_1, \ldots, Y_n)(\exists Z)(word(X_1, \ldots, Z, \ldots, X_m)$$
$$\& \, word(Y_1, \ldots, Z, \ldots, Y_n),$$

which is related to the earlier expression by bound substitution. Since the two occurrences of the shared variable are bound by the same quantifier, any substitution instance which holds true for the first predicative expression is guaranteed to hold true for the second as well.

We now generalise this description for the entire puzzle. Assume that there are m 'across' words and n 'down' words. We may associate each word slot with a word by means of a naming predicate, 'name', such that an expression of the form

$$name(slot\_k, [X_1, X_2, \ldots, X_i])$$

will be interpreted as 'the string of open cells, $X_1, X_2, \ldots, X_i$, will be called slot_k', for $1 \leqslant k \leqslant m+n$. For each of the j $(1 \leqslant j \leqslant i)$ word slots which intersect slot_k, we will substitute a new existentially quantified variable, Z_j at the orthogonal intercept. If we continue this process for all intersecting word slots, we will have converted the two-dimensional puzzle into a linear list of predicative expressions. (In fact, all of these expressions are Horn clauses, hence the title of this article.)

At this point we wish to employ a derivation technique to determine whether the entire set of expressions can be jointly satisfied with respect to some dictionary.

4. THE PROLOG PROGRAM

We suggest that the foregoing is a literal description of our intention in stage (5) of crossword compilation. Any inference technique which can jointly satisfy the predicates defined above can, *inter alia*, provide a solution to the puzzle. Further, each set of substitution instances for the variables will provide another solution to the puzzle.

Fortunately, the descriptive language, Prolog,[2, 3] is available for these sorts of problems. Prolog has a built-in theorem prover which operates upon sets of Horn clauses in a left–right, depth-first manner. In the program we describe below, we shall determine whether a solution exists to a crossword puzzle by determining whether the expressions defined above are consistent with a set of lexical axioms.

The lexical axioms will be predicative expressions similar to those above, except that constants (characters) will replace the variables in the argument, and there will be no slot identifiers. That is, they will be statements of the form 'word$(c_1, c_2, \ldots, c_n)$', where each c_i, $1 \leqslant i \leqslant n$, is a character of the Roman alphabet, perhaps augmented with delimiters (e.g. hyphens) and special symbols (e.g. apostrophes). These axioms define our lexicon; they explicitly state which character strings constitute words. The lexicon which is currently in use consists of 9734 words. The word distribution by length is depicted graphically in Fig. 2.

The program proper (excluding I/O) consists of one procedure, or rule, called 'solution'. The head of the procedure has a list structure as its sole argument. Each element of the list is a variable which stands for one of the word slots. The body of the procedure consists of a set of procedure pairs, again one for each word slot. This body is the literal translation (in Prolog) of the list of predicative expressions referred to in the previous section. A brief example should make these remarks clearer.

Consider the puzzle presented in Fig. 1. We observe that this puzzle is represented on a 15 × 15 matrix which is 84% dense. Further, it is completely interlocked, orthogonally. Each of the word slots is terminated by either a closed cell or the matrix border.

Let us now construct the clause set for the solution of the upper left-hand corner of the puzzle. We observe that this entails filling nine word slots (1, 2, 3, 4 down and 1, 14, 17, 20, 23 across). Further, the open cells of these word slots are interlocked. Thus, the solution of this part of the puzzle involves the following Prolog clause:

```
solution([SLOT_1d, SLOT_2d, SLOT_3d, SLOT_4d,
         SLOT_1a, SLOT_14a, SLOT_17a, SLOT_20a,
         SLOT_23a]):-
  word(C_1, C2_1, C3_1, C4_1, C5_1),
    name(SLOT_1d, [C1_1, C2_1, C3_1, C4_1, C5_1]),
  word(C1_2, C2_2, C3_2, C4_2, C5_2),
    name(SLOT_2d, [C1_2,C2_2,C3_2, C4_2, C5_2]),
  word(C1_3, C2_3, C3_3, C4_3, C5_3,
         C6_3, C7_3, C8_3, C9_3, C10_3),
    name(SLOT_3d, [C1_3, C2_3, C3_3, C4_3,
         C5_3, C6_3, C7_3, C8_3, C9_3, C10_3]),
  word(C1_4, C2_4, C3_4),
    name(SLOT_4d, [C1_4, C2_4, C3_4]),
  word(C1_1, C1_2, C1_3, C1_4),
    name(SLOT_1a, [C1_1, C1_2, C1_3, C1_4]),
  word(C2_1, C2_2, C2_3, C2_4),
    name(SLOT_14a. [C2_1, C2_2, C2_3, C2_4]),
  word(C3_1, C3_2, C3_3, C3_4),
    name(SLOT_17a. [C3_1, C3_2, C3_3, C3_4]),
  word(C4_1, C4_2, C4_3),
    name(SLOT_20a, [C4_1, C4_2, C4_3]),
  word(C5_1, C5_2, C5_3, C5_4, C5_5, C5_6, C5_7),
    name(SLOT_23a, [C5_1, C5_2, C5_3, C5_4, C5_5,
      C5_6, C5_7]).
```

For convenience in exposition, we here identify each cell variable by its row and column coordinates in the

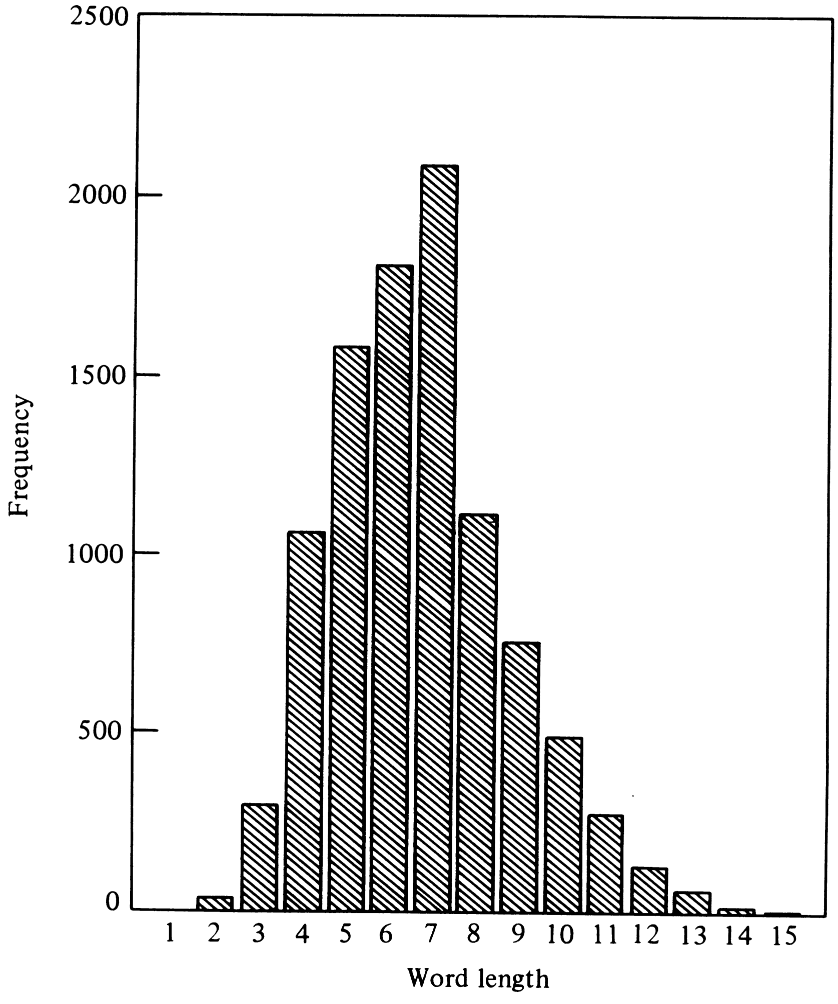

Figure 2. Distribution of words by length.

matrix. Thus '$C3\_4$' is the variable which stands for the cell in the third row and fourth column. We interpret this '*SLOT_2d*',...,'*SLOT_23a*' will be found *if* some substitution instance for the cell variables exists such that each of the resulting strings is a word (i.e. matches with some entry in the dictionary). It is important to recall from Section 3 that each occurrence of a given variable, no matter which goal it is in, is bound by the same implicit quantifier. This guarantees that multiple occurrences of the same variable in the clause set will always be instantiated to the same character for any given solution. This is precisely the meaning of shared cells.

This representational schema is then re-applied for the remaining word slots in the puzzle. The lexicon is a set of clauses of the form '$word(c_1, c_2, \ldots, c_n)$' where the characters $c_1, c_2, \ldots, c_n$ are treated as individual atoms in the argument of the predicate. Aside from a few I/O predicates, some control procedures and the lexicon, the above clause is the entire program. (Note that the identification of a word slot is independent from the recognition of a word in that position. We will return to this point again in Sections 6 and 7.)

5. ILLUSTRATION OF THE METHOD

The program which we have developed is a literal implementation of the procedure described above. For convenience we have added a few simple, interactive routines which help create the clause sets in order to limit the amount of typing. In addition, we employ several techniques for limiting the size of the search space (see below). However, the backbone of the program remains the clause discussed in the previous section. Given enough time, and an adequate dictionary, a program with this format will generate all solutions for any given crossword puzzle.

To illustrate the effectiveness of the program, we began by solving one of the crossword puzzles which appeared in Smith and Steen (see Fig. 3). We terminated the program after 87 hours of operation on an IBM/PC using an interpreted Prolog. At the time of termination, 1286 solutions had been found, the last of which appears as Fig. 4. Despite the fact that the program ran on a microcomputer, the efficiency was on a par with the Smith and Steen approach (on average, one solution every 4.06 minutes). While we had intended to run the program long enough to derive the Smith and Steen solution, it quickly became evident that the solution set was too large to make this goal viable. In fact, even when all the across words were inserted into the program as constants, there were 24 different solutions resulting from variations on the down words alone, of which the Smith and Steen solution was eighth. As an aside, the latter solution set was found in 18.4 minutes.

As our next test, we solved completely interlocked, full puzzles. These puzzles are especially interesting, for the

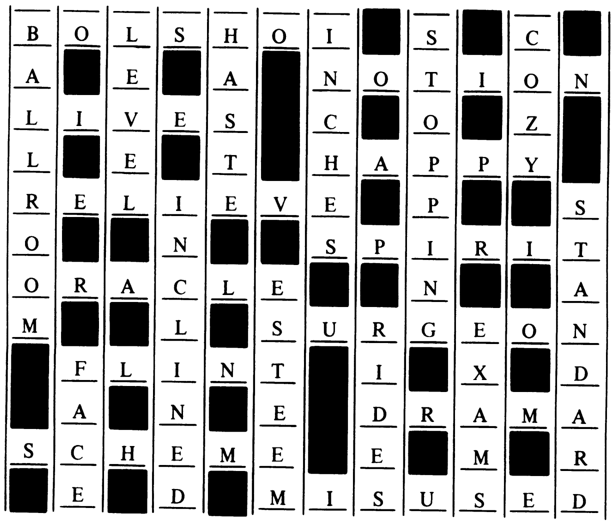
Figure 3. (adapted from Smith and Steen.)[8]

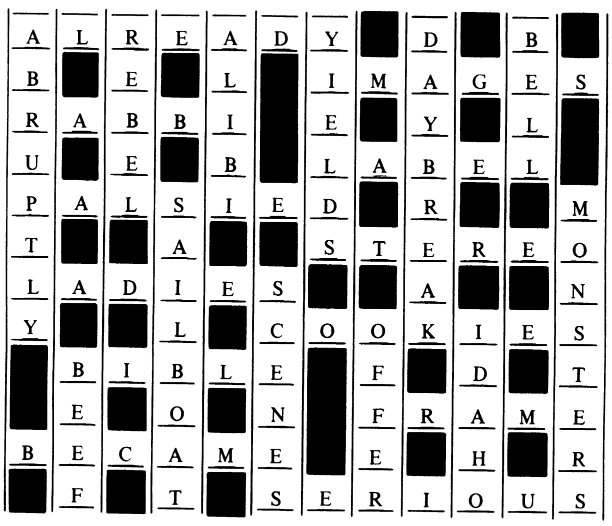
Figure 4. Solution 1286 (run time: 87 hours).

high degrees of interlocking and density increase the search times considerably, while restricting the size of the solution set. Once again, a microcomputer was employed in the solution. In this case, due to the increase in search time, we moved to an IBM/PC-AT and compiled our code.

We solved puzzles defined on 2 × 2, 3 × 3 and 4 × 4 matrices. The sizes of the solution sets were 22, 1246 and 1824, respectively. The run times were, in order, 11 seconds, 35 minutes and 30 hours. The average time required to find each solution was thus 0.5 seconds for the 2 × 2, 1.7 seconds for the 3 × 3 and 1 minute for the 4 × 4. Sample solutions appear in Figs 5–7.

6. HEURISTICS

In Section 5 we alluded to control procedures which increase the efficiency of our program. We now discuss them in greater detail for they illustrate another advantage of the logical approach: the programmer has considerable latitude when it comes to introducing constraints which guide the search strategy. Further, the programmer does not have to rely on *ad hoc* or unintuitive procedures in order to exercise this control.

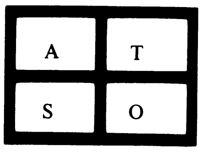

Figure 5. 2 × 2 full puzzle with complete interlocking. Solution 11 of 22.

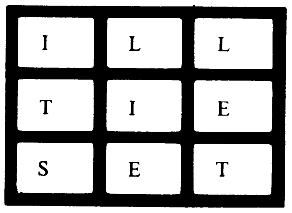

Figure 6. 3 × 3 full puzzle with complete interlocking. Solution 452 of 1246.

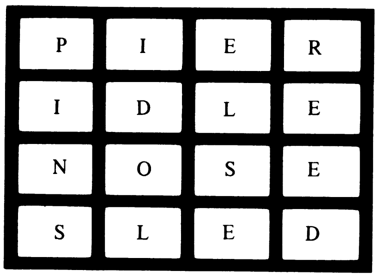

Figure 7. 4 × 4 full puzzle with complete interlocking. Solution 906 of 1824.

It is easy to appreciate the importance of a viable strategy in puzzle solution if one considers the potentially enormous size of the search space. Suppose, for example that one uses a simple, generate-and-test approach to compile a full, $n \times n$ crossword with complete orthogonal interlocking. If k is the size of the alphabet from which the words are drawn, this strategy would create k^{n^2} possible solutions, each one of which would require $2n$ tests. If we restrict ourselves to the Roman alphabet, a simple 4 × 4 puzzle would have 10^{22} possible solutions.

Two basic strategies have been proposed for reducing the search space. Mazlack[7] refers to then as the 'whole word' and 'letter by letter' methods. Under the whole-word approach, each cell of a word slot is filled at the same time, by means of lexical insertion. Under the letter-by-letter approach the cells are filled individually, without immediate reference to the work of which it is a part. The advantage of the whole-word method is that every inserted character and substring is guaranteed to be a part of a word in at least one direction. The disadvantage is that words in parallel word slots may form illegitimate character combinations for subsequent intersecting work slots. Conversely, the strength of the letter-by-letter approach is that illegal letter combinations in both directions can be largely avoided, but there is no guarantee that any given cluster of characters will eventually grow into complete, intersecting words. The letter-by-letter technique was used by Mazlack,[7] while the whole-word method can be found in Smith and Steen.[8]

Our method is a hybrid of the two. It is 'whole word' in so far as insertions are concerned, for entire word slots are filled at once. However, the word insertions are constrained according to whether the substrings which they impose on the intersecting word slots are legitimate. This manner of constraint is in principle the same as that used in letter-by-letter generation. A simple example should illustrate this technique.

Suppose that we were to solve the upper, right-hand part of the puzzle in Fig. 1. Note that it would be unreasonable to insert 'cozy' below 'lazy' in word slots 10 and 16 across, respectively, for this would make solutions for 10, 12 and 13 down impossible (no English word begins with 'lc', 'zz' or 'yy'). We accomplish this checking by means of an additional set of goals of the form

$$word(X_1, X_2, \_, \ldots, \_),$$

where the underscore symbol is taken to be an anonymous, or 'don't care' variable. This predicative expression will succeed only when there is at least one word in the dictionary which has the characters currently instantiated to the variables in the initial two character positions. In practice, we will place such expressions between the 'word' and 'name' goals, one for each intersecting word slot.

The advantage of this type of constraint is that 'naïve backtracking' can be avoided. Backtracking is extremely time-consuming in Prolog for it is inextricably linked to the resolution/unification mechanism. Thus, if left to its own devices, Prolog will backtrack to the point of last success and try another alternative. However, this point may be too far down in the search tree.

To illustrate, consider the partially solved puzzle in Fig. 8. Assume that we have placed the goals in the following order: 1 across, 1 down, 2 across, 2 down, etc. When we fail for 3 down ('ZLA_'), naïve backtracking would take us back to the point of last success (3 across) and try again. Thus we would try in turn 'SEES', 'SELL', 'SEND', and so forth. However, the problem arises farther up in the search tree: by inserting 'ABLE' below 'LAZY', one creates the illegal character combination 'ZL__'. We prevent naïve backtracking by ensuring that no word is inserted which imposes such illegal combinations on the remainder of the puzzle.

The implications of naïve backtracking on efficiency are dramatic. By Smith and Steen's estimate, if there are k n-letter words in the dictionary, a word slot of length n with m characters fixed will have approximately $(k^{1/n})^{n-m}$ alternatives. In our case, where there are approximately 1000 4-letter words in the dictionary, the problem depicted in Fig. 8 will require roughly $((1000^{0.25})^2)^2$ substitutions before the cause of the problem is addressed and a new word is substituted for 2 across. It is easy to see how failure to include constraints like the one mentioned above can account for

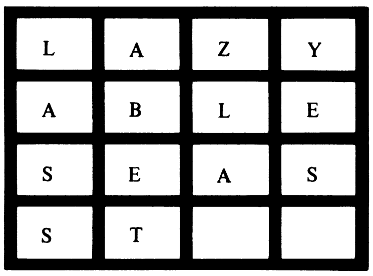

Figure 8. Partially completed 4 × 4 puzzle.

a decrease in performance of several orders of magnitude, a hypothesis confirmed by our own experience.

Our proposal also has implications for other areas of heuristics. For example, the goals in the clause set are portable, and can be arranged for additional efficiency. intuitively, the shortest and longest word slots should be filled in first because there are fewer words of that length to choose from. This prioritization is accomplished by simply shuffling the goals within the clause. The same reasoning would apply to attempts to fill in the most dense or most interlocked portions of the puzzles first.

Further, we may easily avoid the duplication of words in any particular puzzle by collecting the chosen words in a set, and then insisting that a word can only be a candidate for insertion if it is in the dictionary but not in that set.

Lastly, these procedures do not require that the dictionary be laid out in any particular way. In our dictionary, the length of a word is also the arity of the corresponding lexical predicate. For this reason, the search for a candidate for a given word slot will always be restricted to words that do indeed fit, and are in fact appropriate.

7. EXTENSIBILITY

The intuitiveness and simplicity of the method we describe should be obvious from the discussion in Sections 3 and 4. The logic of the problem is directly addressed by the program. All tests are made with respect to the lexical axioms. This means that no character string is considered for insertion into a word slot unless it is actually a word, and no word is considered unless it has the desired orthographical properties (at least, as far as that can be determined at any given stage in the solution). The advantage of using Prolog lies in the fact that the language allows the programmer to solve the problem at a conceptual level rather than a procedural one.

The directness of the method also has important side effects. Perhaps the most important of these is the ability to generalise upon the principles involved to include logically related puzzle relationships. Consider, for example, the phenomena of *slot linking*. This is described by Smith and Steen as the case where a single answer to a question is distributed over several word slots. This is easly handled by our approach, for the notion of a word is logically independent from that of a word slot.

Refer to Fig. 1. Suppose that one answer is divided between the word slots 1 down, 43 across and 61 down, in that order. The clause structure required to recognise this word or phrase is simply

```
solution([..., SLOT_1d, ..., SLOT_43a, ...,
                                    SLOT_61d, ...]):-
    word(C1_1, C1_2, C1_3, C1_4, C9_6, C9_7,
                C9_8, C9_9, C13_12, C14_12, C15_12),
      name(SLOT_1d, [C1_1, C1_2, C1_3, C1_4]),
      name(SLOT_43a, [C9_6, C9_7, C9_8, C9_9]),
      name(SLOT_61d, [C13_12, C14_12, C15_12]),       .
```

Or perhaps we might wish to add the extra condition that the diagonal of a square puzzle must also be a word. This can be handled by adding the following test

```
solution([..., DIAG, ...]):-
    word(C1_1, C2_2, ..., Cn_n),
      name(DIAG, [C1_1, C2_2, ..., Cn_n]),
```

Further we could extend our method to include *linear interlocking* as well. In this situation, one word slot overlaps another on the same row or column (indeed, one may be contained within the other). This involves a non-orthogonal type of cell sharing. In this case the common cells, no matter where they appear, would simply be placed in two separate argument lists. In Fig. 1, for example, we might insist that word slot 9 down be filled by an eight-character work with ends in the four-character word, 25 down. The appropriate Prolog fragments necessary to add this requirement would be:

```
solution([..., SLOT_9d, ..., SLOT_25d, ...,]):-
  word(C1_10, C2_10, C3_10, C4_10, C5_10,
                          C6_10, C7_10, C8_10),
  word(C5_10, C6_10, C7_10, C8_10),
    name(SLOT_9d, [C1_10, C2_10, C3_10,
                C4_10, C5_10, C6_10, C7_10, C8_10]),
    name(SLOT_25d, [C5_10, C6_10, C7_10, C8_10]),
```

The combinations and variations are endless: one can easily imagine the generation of symmetrical puzzles, puzzles whose word slots wrap around, puzzles in three or more dimensions, puzzles with unusual geometries (e.g. triangles, circles), and so forth. In addition, the program can be used to complete puzzles which are partially worked through. This involves nothing more than substituting characters for some of the variable positions. All of these variations are easily accomplished because the program is a direct representation of the underlying logic of crossword puzzles.

8. CONCLUSION

In our opinion the strategy of crossword compilation described in this article has several distinct advantages. For one, the code is a literal transcription of the logical analysis of the problem. This makes the code perspicuous and easy to maintain and modify. The word slots are clearly identified, and the conditions for word status are plainly noted. Once one comes to appreciate the fact that this method solves crossword puzzles by proving that the listed goals are consequences of the lexical axioms, the function of each goal is obvious.

Further, as we say above, myriad possible extensions of the basic crossword puzzle are easily implemented. This is a direct result of the 'naturalness' of the methodology. The program is capable of virtually any extension which is consistent with the underlying logic of crossword puzzles.

Another advantage is simplicity. The 'business part' of the program is presented in a single procedure, with a goal for each word and word slot. Since the program relies upon the theorem-proving mechanism of Prolog, little consideration need be given to such things as search techniques, data structures and data formats. This distinguishes this method from those of Smith and Steen and of Mazlack.[4-7] Our method is a direct and intuitive approach to the problem, with a minimal amount of distracting code.

Perhaps the most important advantage, though, is that the techniques described here have implications for approximate string matching and spelling verification. Brief reflection will show that the logic of crossword puzzle compilation carried over to these other areas as well. For further discussion see Ref. 1.

REFERENCES

1. H. Berghel and E. Traudt, Spelling verification in Prolog. *SIGPLAN Notices*, **21** (1), 19–27 (1986).
2. W. Clocksin and C. Mellish, *Programming in Prolog*. Springer-Verlag, New York (1981).
3. R. Kowalski, *Logic for Problem Solving*. North-Holland, New York (1979).
4. L. Mazlack, The use of applied probability in the computer construction of crossword puzzles. *Proceedings of IEEE Conference on Decision and Control*, pp. 497–506. IEEE Press, Washington (1973).
5. L. Mazlack, Data structures required for crossword puzzle construction. *Proceedings of the 36th Annual Meeting of the American Society for Information Science, American Society for Information Science, Washington*, pp. 141–142 (1974).
6. L. Mazlack, Computer construction of crossword puzzles using precedence relationships. *Artificial Intelligence* **7** (1), 1–19 (1976).
7. L. Mazlack, Machine selection of elements in crossword puzzles – an application of computational linguistics. *SIAM Journal on Computing*, **5** (1), 51–72 (1976).
8. P. Smith and S. Steen, A prototype crossword compiler. *The Computer Journal*, **24** (2), 107–111 (1981).